RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

LEHRSTUHL INFORMATIK XI SOFTWARE FÜR EINGEBETTETE SYSTEME

Professor Dr.-Ing. Stefan Kowalewski
Hilal Diab, M.Sc.
Kamal Barakat, M.Sc
Dipl.-Inform. Dominik Franke

Aachen, 22. Januar 2010
SWS: V4/Ü2, ECTS: 7

Einführung in die Technische Informatik

WS 2009/2010

Blatt 12: Musterlösung

Aufgabe 1: VHDL

a) Gegeben sei folgender Ausschnitt aus einer VHDL Spezifikation:

```
entity E2 is
  port(x: in std_logic_vector(3 downto 0);
    clk: in std_logic;
    f: out std_logic;
    g: out std_logic);
end E2;
```

Erstellen Sie ein Blockdiagramm (ähnlich Aufgabe 5), welches das Modul E2 und seine Ein- und Ausgänge darstellt.

b) Das Verhalten des Moduls E2 ist in folgendem VHDL Code spezifiziert:

```
architecture P3 of E2 is
begin
  process(x, clk)
  begin
    if rising_edge(clk) then
      if(x = "1100" or x = "0110") then
        f <= '1';
      else
        f <= '0';
      end if;
    end if;
  end process;
  g <= (x(3) and x(2)) or (x(1) and x(0));
end P3;
```

Welche Funktionen werden an den Ausgängen f und g des Moduls realisiert? Was haben sie gemeinsam und worin unterscheiden sie sich?

Aufgabe 2: (⋆)von-Neumann-Rechnermodell

1946 wurde das von-Neumann-Rechnermodell vorgestellt, das die Rechnerarchitektur bis heute maßgeblich beeinflusst. Arbeiten Sie die grundlegenden Organisationsprinzipien und Besonderheiten dieses Modells heraus, indem Sie folgende Fragen möglichst prägnant und in

eigenen Worten beantworten. Zur Hilfe können Sie das Lehrbuch von Oberschelp/Vossen oder das Internet nehmen.

a) Mit dem von-Neumann-Rechnermodell wurde erstmalig das Konzept für einen echten 'general-purpose Computer' vorgeschlagen. Was ist darunter zu verstehen?

b) Das von-Neumann-Rechnermodell setzt sich aus drei Hauptbestandteilen zusammen. Welche Bestandteile sind dies und welchem Zweck dienen sie?

c) Im von-Neumann-Rechnermodell ist der Datenprozessor ein Bestandteil der CPU. Welche Aufgaben werden von welchen Komponenten dieses Prozessors erfüllt?

d) Das von-Neumann-Rechnermodell unterscheidet zwischen Daten- und Adressbus.Warum macht das Sinn? Es ergeben sich auch Zusammenhänge zwischen der Größe (in Bits) des MAR, des MBR, des Speichers, einer Speicherzelle sowie der Speicherzellenanzahl. Welche sind dies?

e) Bahnbrechend neu am von-Neumann-Rechnermodell war das Konzept einer quasi universellen Programmierbarkeit. Erörtern Sie in diesem Zusammenhang die Begriffe Maschinencode, Assemblersprachen sowie Ein- und Mehr-Adress-Befehle.

f) Charakteristisch für das von-Neumann-Rechnermodell ist ein Zwei-Phasen-Konzept der Befehlsverarbeitung. Welches Problem wird damit auf welche Weise gelöst?

Lösungsvorschlag

a) **general-purpose Computer:** Bahnbrechend neu am von Neumann-Rechnermodell war das Konzept eines general-purpose Computers, der versehen mit dem geeigneten Programm jede prinzipiell von einem Computer durchführbare Aufgabe ausführen kann. Bis dahin waren vor allem special-purpose Computer entworfen worden, die nur eine einzige Aufgabe lösen konnten, z.B. Berechnung balistischer Kurven.

b) **Hauptbestandteile des von Neumann-Rechnermodells:**

- Zentraleinheit (CPU) bestehend aus aus zwei Prozessoren, dem Datenprozessor und dem Befehlsprozessor (s.w.u.).
- Speicher bestehend aus Random Access Memory = RAM und Read Only Memory = ROM (s.u.).
- Ein-Ausgabe-Einheit (PIO) als notwendige Schnittstelle nach außen (Eingaben und Ausgaben können teilweise parallel erfolgen).

Die Teile sind durch Busse verbunden (s.w.u.). Die Busse können Daten transportieren. Es gibt billige 1-Bit-Leitungen und schnellere und teure Parallel-Busse. Die CPU führt Befehle aus und steuert den Ablauf (s.w.u.). Daten und Programme werden als Bitfolgen im Speicher abgelegt (s.o.). Der Speicher besteht aus einer festen Anzahl von Plätzen, von denen jeder durch seine Adresse angesprochen werden kann (s.w.u.). Dabei unterscheiden wir zwischen ROM und RAM. Das ROM ist in das Steuerwerk des Befehlsprozessors (s.w.u.) integriert. Hier stehen feste Informationen, z.B. solche Befehle,

die eine CPU häufig auszuführen hat (vgl. Mikroprogrammierung). [1] Demgegenüber enthält das RAM die Daten und die Programme. Hier kann gelesen und geschrieben werden, d.h. an dieser Stelle wechselt der Speicherinhalt mit der jeweiligen Aufgabe.

c) **Bestandteile des Datenprozessors:** Den Mittelpunkt im Datenprozessor bildet ein Rechenwerk, die Arithmetic Logical Unit (kurz: ALU). Hinzu kommen Register zur Speicherung von Operanden. Im einzelnen sind dies: ein Akkumulator A, ergänzt um ein Link-Register L, das zum Beispiel den Übertrag bei der Addition aufnimmt, dazu ein Multiplikator-Register MR, das zum Beispiel zur Aufnahme von Multiplikationsergebnissen dient, und schließlich noch ein Memory-Buffer-Register MBR für die Kommunikation mit dem Speicher.

   Im wesentlichen entspricht der Datenprozessor also den von uns bereits „gebauten" Rechnerbausteinen. Man vergleiche den Aufbau mit unseren Addierwerken.

d) In der Interpretations-Phase eines von Neumann-Rechners (s.w.u.) wird der Inhalt der Speicherzelle mit der im MAR gespeicherten Adresse über den Adressbus abgefragt und über den Datenbus in das MBR gebracht.

   Im allgemeinen ist die Länge einer Speicherzelle verschieden von der Länge ihrer Adresse. Dies motiviert die Unterscheidung zwischen Daten- und Adressbus.

   > „Falls eine Speicherzelle m Bits aufnehmen kann, zur Darstellung ihrer Adresse n Bits erforderlich sind und $m \neq n$ gilt, so ist es nicht sinnvoll, den gleichen Bus zur Übertragung von Daten und Adressen zu verwenden, da etwa im Fall $m > n$ bei einem parallelen m-Bit-Bus ein gewisser Teil während der Übertragung ungenutzt bleibt." (Oberschelp/Vossen 1998, S. 239)

   Daraus ergibt sich ein Zusammenhang zwischen RAM-Größe und Adreßbus-Breite. Beträgt die Speichergröße $L = m \cdot 2^n$ Bits, dann macht es Sinn, einen n-Bit-Adreßbus zu wählen, um damit jede der m-Bit Speicherzellen direkt (d.h. ohne weitere Kunstgriffe) über n Bits adressieren zu können. Das MAR-Register muß dementsprechend über n Speicherplätze verfügen, um die direkte Adressierung zu ermöglichen (bei $n + n'$ Speicherplätzen, blieben n' Speicherplätze im MAR ungenutzt).

   Eine analoge Betrachtung für den Datenbus und die Länge einer Speicherzelle führt dazu, daß es Sinn macht, die Datenbusbreite an die Speicherzellengröße anzulehnen. Das MBR-Register muß dementsprechend über m Speicherplätze verfügen, damit der komplette Inhalt einer Speicherzelle in einem Abruf der CPU zugeführt werden kann.

e) **Maschinencode, Assemblersprachen, Ein-Adreß-Befehle:** Ein Programm ist eine Befehlsfolge und als solche eine Folge von Binärzahlen. Die Umsetzung von Befehlen in Binärzahlen wird als Maschinencode bezeichnet. Nur der Maschinencode ist von der CPU interpretierbar, für Menschen dagegen ziemlich unlesbar. Daher werden neben den „höheren" Programmiersprachen noch Assemblersprachen entworfen, die einerseits der maschinennahen Programmierung dienen und andererseits lesbar sind.

   An Berechnungen ausführenden (Assembler-)Befehlen ist der Akkumulator beteiligt, er muß daher nicht adressiert werden. Einstellige Operatoren benötigen daher keine Adresse. Der Operand steht im Akku und das Ergebnis wird dann im Akku abgelegt. Dabei

[1] Heutzutage sind ROM's nicht mehr unveränderlich, d.h. sie können ebenfalls programmierbar oder aber überschreibbar sein. In diesem Fall lassen sich auch die Mikroprogramme eines Rechners erneuern.

geht der Operand nicht unbedingt verloren, da er in der Regel auch noch im Speicher steht. Für zweistellige Operationen (Addition, Multiplikation) reicht die Angabe (der Adresse) des 2-ten Operanden, sofern der 1-te Operand im Akku steht.

Wir kommen also mit Ein-Adreß-Befehlen aus. Oft nur der Bequemlichkeit halber werden auch Zwei- oder Drei-Adreß-Befehle (manchmal) betrachtet.

Im Falle einer hardwaretechnischen Realisierung werden dann neben dem Akku noch zusätzliche Register benötigt, so daß im Prinzip auch Effizienzgewinne zu erzielen sind. Üblicherweise erfolgt eine Umsetzung von Mehr-Adreß-Befehlen schon auf der Software-Seite (die Sprache wird im allgemeinen passend zur Hardware entwickelt und nicht umgekehrt), so daß sich in diesem Fall — außer einer dann (vielleicht) leichteren Programmierung — im wesentlichen nichts ändert.

f) **Zwei-Phasen-Konzept:** Aus b) und g) ergibt sich insbesondere das Problem, daß der Rechner aus dem zeitlichen Kontext erkennen muß, ob ein aus dem Speicher geholtes Bit-Muster als „Befehl“ oder als „Datum“ zu interpretieren ist. Im klassischen Anzatz wurde dies technisch über zwei Phasen in der Befehlsverarbeitung gelöst:[2]

(1) Interpretationsphase (Fetch-Phase)

1.1 Der Inhalt des Befehlszählers PC wird in das Speicheradreßregister MAR geschrieben.

1.2 Der Inhalt der Speicherzelle mit der in MAR gespeicherten Adresse wird über den Adreßbus abgefragt und über den Datenbus in das Memory-Buffer-Register (MBR) und dann in das Befehlsregister IR gebracht.

1.3 Der Decodierer interpretiert den in IR abgelegten Befehl.

1.4 Der Befehlszähler PC wird aktualisiert.
In der Regel heißt dies, daß der Zähler um 1 erhöht wird, da Programme Folgen nacheinander auszuführender Befehle sind. Ausnahmen bilden hier Sprungbefehle (Schleifenende, if-tests, Unterprogramm-Aufrufe).

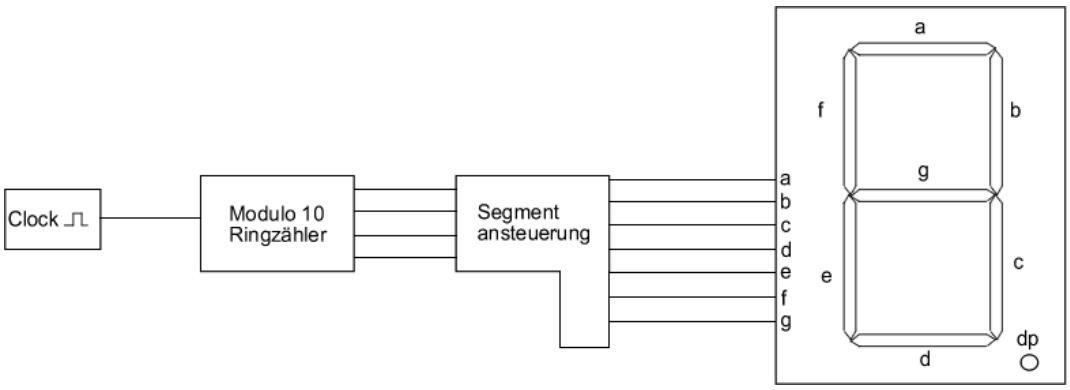
(2) Execution-Phase

Hier erfolgt die eigentliche Befehlsausführung, die vom Steuerwerk gesteuert und in der ALU ausgeführt wird. Über MAR können die benötigten Daten ausgewählt werden und über MBR in die ALU gebracht werden. Die ALU hat dann Zugriff zum Programm und zu den Daten in den Registern des Datenprozessors.

Aufgabe 3: 7 Segmentanzeige

Gegeben sei folgende Schaltung zur Ansteuerung eines 7-Segmentdisplays. Der Modulo-10-Ringzähler gibt Zahlen von 0-9 aus, wobei der Ringzähler nach jedem Takt inkrementiert wird. Die aus dem Ringzähler gewonnene Zahl in BCD-Darstellung wird an eine Segmentansteuerungsschaltung weitergeleitet. Dort wird die Zahl so umgewandelt, dass die jeweilige Repräsentation der BCD-Zahl auf der Segmentanzeige sichtbar wird. Die Segmentanzeige besteht aus 7 Segmenten (oftmals mit zusätzlichem Punkt, der hier nicht betrachtet wird). Jedes Segment kann unabhängig vom anderen durch Anlegen einer logischen 1 bzw. 0 am entsprechenden Eingang an- und ausgeschaltet werden.

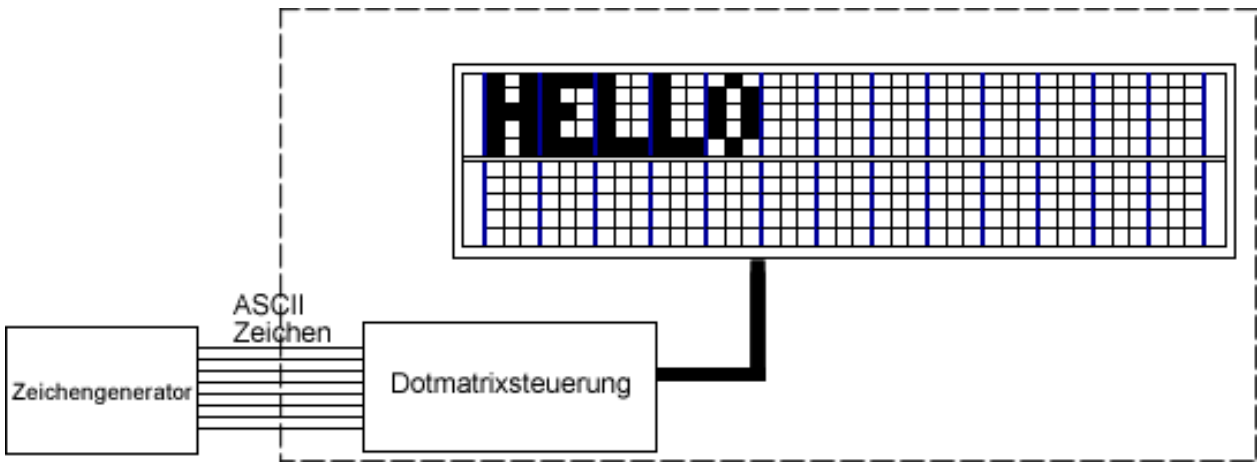
[2]Modelle mit mehr als zwei Phasen sind ebenfalls denkbar und später auch umgesetzt worden.



a) Stellen Sie eine Boolesche Funktion für den Ringzähler auf und minimieren Sie diese mit Hilfe von Karnaugh-Diagrammen. Zeichnen Sie ein Schaltnetz für diese Funktion (Leiterbahnen beschriften!). Alle Bauteile sollen einen Fan-In von 2 besitzen. Welche Bauteile werden benötigt, um den Ringzähler zu realisieren?
(Tipp: Es gibt Eingabewerte in der Funktion, für die keine Ausgabewerte definiert sind.)

b) Schreiben Sie die Funktionstabelle für die Segmentansteuerungsanzeige auf und minimieren Sie jede Funktion mit Hilfe von Karnaugh-Diagrammen.

Aufgabe 4: ($\star$)Punktmarixdisplay
Ein Punktmatrixdisplay ist eine Anzeige zum Darstellen von Zeichen. Die Zeichen werden mit Hilfe einer Punktmatrix dargestellt, in der, je nach Buchstabe, ein Punkt an bzw. ausgeschaltet wird. Das Display hat Eingänge zum Empfangen der Zeichen bzw. zur Steuerung des Cursors (Cursorsteuerug wird hier nicht betrachtet). Das Punktmatrixdisplay verfügt über eine Punktmatrixsteuerung, die ankommende ASCII-Zeichen in ihre entsprechende Matrixvariante umwandelt - der Zeichensatz (alle Buchstaben sind Großbuchstaben!) des Displays ist unten angegeben. Ein 3x5 Punktmatrixdisplay mit einem angeschlossenen Zeichengenerator:



(Bsp.: Wenn das Zeichen A ausgegeben werden soll, müssen die Punkte 0,1,2,3,5,6,7,8,9,B,C,E auf logisch 1 gesetzt werden, während die Punkte 4, A, D auf logisch 0 gesetzt werden müssen.)

a) Realisieren Sie die Schaltfunktion der Punktmatrixsteuerung, die eine ASCII-Bitfolge in eine Bitfolge zum An- und Ausschalten der Punkte abbildet, mittels eines PLAs. Überlegen Sie dabei, wie viele Eingänge, Ausgänge und Spalten das PLA haben soll. (Tipp: Minimieren von Funktionen ist hier nicht zu empfehlen!)

b) An das Punktmatrixdisplay ist ein Zeichengenerator angeschlossen, der das Wort 'FROHE OSTERN' mit einem anschließenden Leerzeichen ausgeben soll, indem der Generator jedes einzelne Zeichen der Zeichenkette zur Punktmatrixsteuerung überträgt. Zur Verfügung stehen Ihnen ein PLA und ein 7 Bit breites Register. Realisieren Sie den Generator mit Hilfe der beiden Bauteile. Gehen Sie davon aus, dass der Generator beim Start initial die Bitfolge '0000000' in seinem Register gespeichert hat.
(Tipp: Auch hier ist Minimieren von Funktionen nicht zu empfehlen!)

Lösungsvorschlag

a) Um den Zeichensatz des Display anzeigen zu können, benötigt man lediglich 7 Eingänge, da das höchste Bit, das gesetzt ist bei den ASCII-Zeichen, Bit 7 ist. Das PLA benötigt 15 Ausgänge, da 15 Punkte unabhängig voneinander angesteuert werden sollen. Die Anzahl der Spalten des PLA entspricht der Anzahl der Zeichen des Displayzeichensatzes, also 27 Spalten. Das PLA sieht dann wie folgt aus:

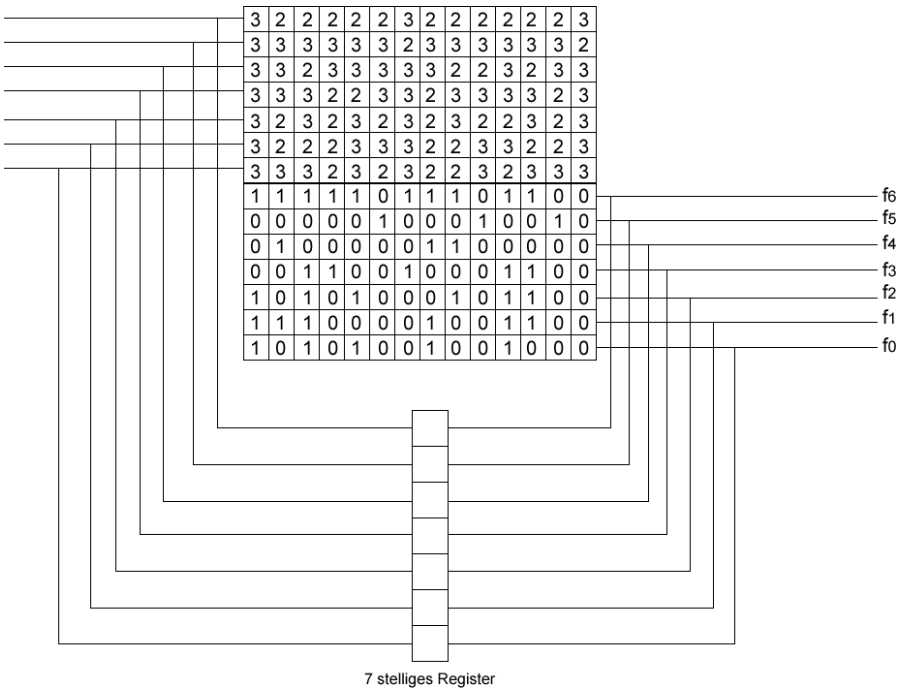
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_	
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	2	3	
3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	2	2	2	3	
3	3	3	2	2	2	2	3	3	3	3	2	2	2	2	3	3	3	3	2	2	2	2	3	3	3	3	
3	2	2	3	3	2	2	3	3	2	2	3	3	2	2	3	3	2	2	3	3	2	2	3	3	2	3	
2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	3	
1	1	1	1	1	1	0	1	0	1	1	1	1	0	0	1	0	1	0	1	1	1	1	0	1	1	0	0
1	1	1	1	1	1	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	1	0	1
1	1	1	0	1	1	1	1	0	1	1	0	1	0	0	1	0	1	1	1	1	1	1	0	1	1	0	2
1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0	3
0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	4
1	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	5
1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0	0	0	6
1	1	0	0	1	1	1	1	1	0	1	0	0	1	0	1	0	1	1	1	0	0	0	1	1	1	0	7
1	1	0	1	1	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	1	1	1	0	0	0	0	8
1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	9
0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	1	0	1	0	0	1	1	1	0	0	A
1	1	0	1	0	0	1	1	0	1	1	0	1	1	1	0	1	0	1	0	1	1	1	0	0	0	0	B
1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	0	1	1	0	1	0	1	1	0	1	0	C
0	1	1	1	1	0	1	0	1	1	0	1	0	0	1	0	0	0	1	1	1	1	0	0	1	1	0	D
1	1	1	0	1	0	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1	0	1	1	0	1	0	E

In der UND-Ebene sind jeweils die Minterme für die ASCII-Repräsentation des jeweiligen Buchstabens abgetragen.
In der ODER-Ebene wird jetzt genau nur dann eine 1 gesetzt, wenn ein Punkt beim Anzeigen eines Buchstabens angeschaltet wird. (Z.B. Punkt 0 wird genau dann angeschaltet, wenn die Buchstaben A-F,H,J-M,P,R,T-W,Y,Z angezeigt werden sollen).

b) Der Generator ähnelt dem Ringzähler vom Prinzip her, nur dass beim Generator nicht um eins inkrementiert wird, sondern der nächste Buchstabe als ASCII-Bitfolge am Ausgang bereitsteht. Die Funktionstabelle:

Eingangsbuchstabe	x_6 - x_0	f_6	f_5	f_4	f_3	f_2	f_1	f_0	Ausgangsbuchstabe
NULL	0000000	1	0	0	0	1	1	1	F
F	1000110	1	0	1	0	0	1	0	R
R	1010010	1	0	0	1	1	1	1	O
O	1001111	1	0	0	1	0	0	0	H
H	1001000	1	0	0	0	1	0	1	E
E	1000101	0	1	0	0	0	0	0	_
_	0100000	1	0	0	1	0	0	0	O
O	1001111	1	0	1	0	0	1	1	S
S	1010011	1	0	1	0	1	0	0	T
T	1010100	0	1	0	0	0	0	0	E
E	1000101	1	0	0	1	1	1	1	R
R	1010010	1	0	0	1	1	1	0	N
N	1001110	0	1	0	0	0	0	0	_
_	0100000	0	0	0	0	0	0	0	NULL

Wichtig hier ist, wenn der Generator gestartet wird, so ist der Anfangswert, den der Generator gespeichert hat, 0. Falls also das Register die Zahl 0 gespeichert hat, so muss natürlich als erstes der Buchstabe 'F' ausgegeben werden. Dies steht in der erste Zeile der Funktionstabelle.
Das PLA sieht dann wie folgt aus:



Das Register speichert jeweils den zuletzt ausgegebenen Buchstaben und führt diesen dem PLA immer wieder erneut zu. Auch hier reichen 7 Eingänge für das PLA(Erklärung wie in Aufgabenteil a). Der Generator sollte auch 7 Ausgänge haben, da das Punktmatrixdisplay 7 Eingänge hat. Die Anzahl der Spalten ist, die Anzahl Zeichen des Wortes 'FROHE_OSTERN_' + das NULL-Zeichen (das beim Starten des Generators initial in den Registern vorhanden ist).

Aufgabe 5: PLA's

Mit PLA's lassen sich Schaltnetze einfach realisieren durch die Angabe des Bausteintyps innerhalb der PLA-Matrix. In dieser Aufgabe sind PLA's von verschiedene Bauteilen angegeben. Geben Sie für jedes dieser PLA an, welche Funktion realisiert wird bzw. um welches Bauteil es sich handelt.

a)

x_0	2	
x_1	2	
	1	f_0

b)

x_0	2	0	
x_1	0	2	
	1	1	f_0

c)

x_0	3	3	2	
x_1	3	2	3	
	1	1	1	f_0

d)

x_0	3	2	
x_1	2	3	
	1	1	f_0

e)

x_0	3	
	1	f_0

f)

x_0	3	
x_1	3	
	1	f_0

g)

x_0	2	2	2	2	
x_1	3	2	3	2	
x_2	3	3	2	2	
	1	0	0	0	f_0
	0	1	0	0	f_1
	0	0	1	0	f_2
	0	0	0	1	f_3

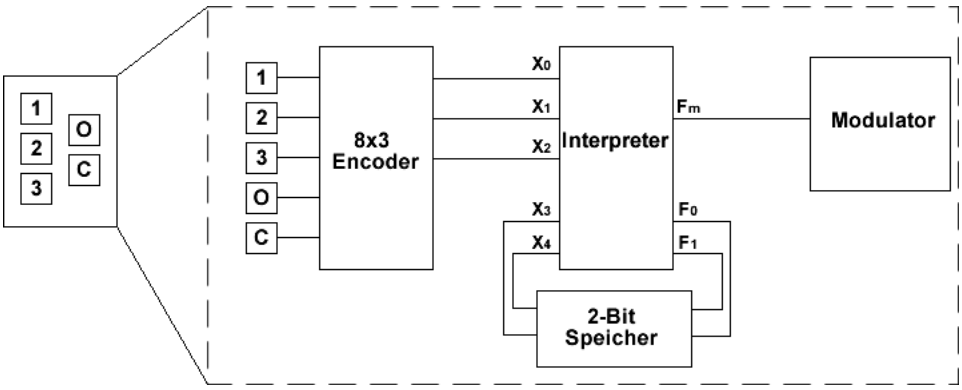
h)

x_0	2	0	0	0	
x_1	0	2	0	0	
x_2	0	0	2	0	
x_3	0	0	0	2	
	0	1	0	1	b_0
	0	0	1	1	b_1

i)

x_0	2	2	2	2	
x_1	3	2	3	2	
x_2	3	3	2	2	
	1	0	0	0	f_0
	0	1	0	0	f_1
	0	0	1	0	f_2
	0	0	0	1	f_3

Aufgabe 6: ($\star$)Garagentor-Fernbedienung



In dieser Aufgabe sollen Sie einen Teil einer Garagentor-Fernbedienung entwerfen. Die Fernbedienung hat folgende Funktionen:
Tor 1 öffnen/schließen, Tor 2 öffnen/schließen, Tor 3 öffnen/schließen, alle Tore öffnen (O), alle Tore schließen (C). Sie sollen dabei konkret den Interpreter-Baustein realisieren. Die Fernbedienung funktioniert wie folgt:
Die Tasten der Fernbedienung sind an einen 8x3 Encoder angeschlossen, der die jeweilige Taste in eine Binärzahl umwandelt. Taste 1 ist an Eingang 0 des Encoders angeschlossen, Taste 2 an Eingang 1, Taste 3 an Eingang 2 usw.. Dabei ist zu bedenken, dass nicht alle Eingänge des Encoders benutzt werden und somit nicht alle Binärzahlen am Ausgang anliegen (z.B. wird das Wort $(101)_2$ nie am Ausgang ausgegeben, da es eine sechste Taste an der Fernbedienung nicht gibt).

Die 3 Ausgänge des Encoders sind mit dem Interpreter-Baustein verbunden (x_2,x_1,x_0). Der Ausgang F_m des Interpreter-Bausteins geht direkt zum Modulator, welcher vom Interpreterbaustein ein 4-Bit breites Datenwort seriell zugeschickt bekommt und das anschließend in ein Funksignal umwandelt. Dabei wird das LSB immer zuerst gesendet.
Die Eingänge x_3 und x_4 dienen dazu, in Kombination mit den Eingängen x_0, x_1 und x_2, das aktuell zu sendende Bit, das an F_m anliegen soll, zu bestimmen. Das Wort $(x_4, x_3)_2$ wird dabei inkrementiert und direkt an den Ausgang $(F_1, F_0)_2$ weitergeleitet. Die Ausgänge F_0 und F_1 werden anschließend in ein 2-Bit Register gespeichert und stehen im nächsten Takt wieder als x_3 und x_4 zur Verfügung, damit bestimmt werden kann, welches Bit als nächstes an F_m anliegt.

Beispiel (siehe hierzu Tabelle unten): Wenn Taste 3 gedrückt wird($(x_2, x_1, x_0)_2 = (011)_2$), soll die Zahl $(14)_{10} = (1110)_2$ an den Modulator seriell übertragen werden.
x_3 und x_4 sind anfangs beide 0. Es muss also das $(x_4, x_3)_2 = (00)_2 = (0)_{10}$ -te Bit übertragen werden - eine 0. An F_m liegt deswegen eine 0 an. Das Wort $(x_4, x_3)_2$ wird inkrementiert.

Im nächsten Takt ist $(x_4, x_3)_2 = (01)_2$. Es soll also das $(x_4, x_3)_2 = (01)_2 = (1)_{10}$ -te Bit übertragen werden - eine 1. An F_m liegt deswegen eine 1 an. Das Wort $(x_4, x_3)_2$ wird wieder inkrementiert usw.

Wenn $(x_4, x_3)_2 = (11)_2$ ist und inkrementiert wird, wird $(x_4, x_3)_2$ wieder auf $(00)_2$ gesetzt und der Vorgang beginnt von Neuem. Insgesamt werden nacheinander die Bits 0 (LSB), 1, 1, 1 (MSB) an den Modulator gesendet.
Die folgende Tabelle zeigt für jede Taste an, welcher Wert an den Modulator gesendet werden

soll:

Taste	x_4	x_3	x_2	x_1	x_0	F_1	F_0	F_m	Zu sendender Dezimalwert
1									6
2									2
3	0	0	0	1	0	0	1	0	14
	0	1	0	1	0	1	0	1	
	1	0	0	1	0	1	1	1	
	1	1	0	1	0	0	0	1	
O									9
C									7

a) Füllen Sie die obige Tabelle aus.

b) Konstruieren Sie den Interpreter-Baustein mit Hilfe eines PLA.

Lösungsvorschlag

a)

Taste	x_4	x_3	x_2	x_1	x_0	F_1	F_0	F_m	Zu sendender Dezimalwert
1	0	0	0	0	0	0	1	0	6
	0	1	0	0	0	1	0	1	
	1	0	0	0	0	1	1	1	
	1	1	0	0	0	0	0	0	
2	0	0	0	0	1	0	1	0	2
	0	1	0	0	1	1	0	1	
	1	0	0	0	1	1	1	0	
	1	1	0	0	1	0	0	0	
3	0	0	0	1	0	0	1	0	14
	0	1	0	1	0	1	0	1	
	1	0	0	1	0	1	1	1	
	1	1	0	1	0	0	0	1	
O	0	0	0	1	1	0	1	1	9
	0	1	0	1	1	1	0	0	
	1	0	0	1	1	1	1	0	
	1	1	0	1	1	0	0	1	
C	0	0	1	0	0	0	1	1	7
	0	1	1	0	0	1	0	1	
	1	0	1	0	0	1	1	1	
	1	1	1	0	0	0	0	0	

b) Der PLA Baustein hat 5 Eingänge für x_4 - x_0, 3 Ausgänge für die Funktionen F_1, F_0 und F_m. Weiterhin besteht der Baustein aus 20 Spalten, denn es gibt insgesamt 20 Zeilen in der Funktionstabelle. Somit sieht der Baustein wie folgt aus:

x4	3	3	2	2	3	3	2	2	3	3	2	2	3	3	2	2	3	3	2	2	
x3	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	3	2	
x2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	
x1	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	3	3	3	3	
x0	3	3	3	3	2	2	2	2	3	3	3	3	2	2	2	2	3	3	3	3	
	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	F1
	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	F0
	0	1	1	0	0	1	0	0	0	1	1	1	1	0	0	1	1	1	1	0	Fm

Aufgabe 7: ($\star$)Einführung der MMIX-Umgebung

Diese Aufgabe soll in die Benutzung der MMIX-Umgebung einführen. Für weitere Informationen (Datenblätter, MMIX-Befehle, etc.) schauen Sie bitte auf folgende Webseite:

http://www-kbsg.informatik.rwth-aachen.de/fileskbsg/legacy/teaching/SS2004/RS/MMIX_info/index.html

Erstellen eines MMIX-Programms:

1. Quellcode erstellen und mit der Endung .mms speichern
2. Assemblieren durch: mmixal name.mms
3. Objektdatei ausführen mit: mmix name.mmo

a) Erzeugen Sie die Datei `einfach.mms`, die folgendes Programm enthält (Zeilennummern und Kommentare nicht mit eingeben):

```
 1          LOC #100          Start des Programms im Speicher
 2 x        IS $42            Anweisung, 'x' im Programm durch $42
                              zu ersetzen
 3 y        IS $43            ...
 4 z        IS $44            ...
 5 Main     SET x,40          Main: hier beginnt das Hauptprogramm
                              Setze x (also das Register 42, $42)
                              auf den Wert 40
 6          SET y,7           Setze y auf 7
 7          MUL $44,$42,$43   Multipliziere Reg. 42 mit Reg. 43,
                              schreibe das Ergebnis nach Reg. 44
 8          ADD z,z,3         ...
 9          DIV y,z,13        ...
10          TRAP 0,Halt,0     Ende des Programms
```

b) Erzeugen Sie den assemblierten Code `einfach.mmo`.

c) Welchen Wert enthalten die Register 42, 43 und 44 nach der Ausführung von Zeile 9 durch MMIX? Wie kommt man schnell an diese Information mit Hilfe von `mmix`?

d) Ändern Sie das Programm so ab, dass anfangs $x = 6$, $y = 7$ und $z = 20$ gilt, der Term $xy(x + y)/(z - y)$ berechnet wird und das Ergebnis in Register 14 steht.

Lösungsvorschlag

a)

b) `mmixal einfach.mms`

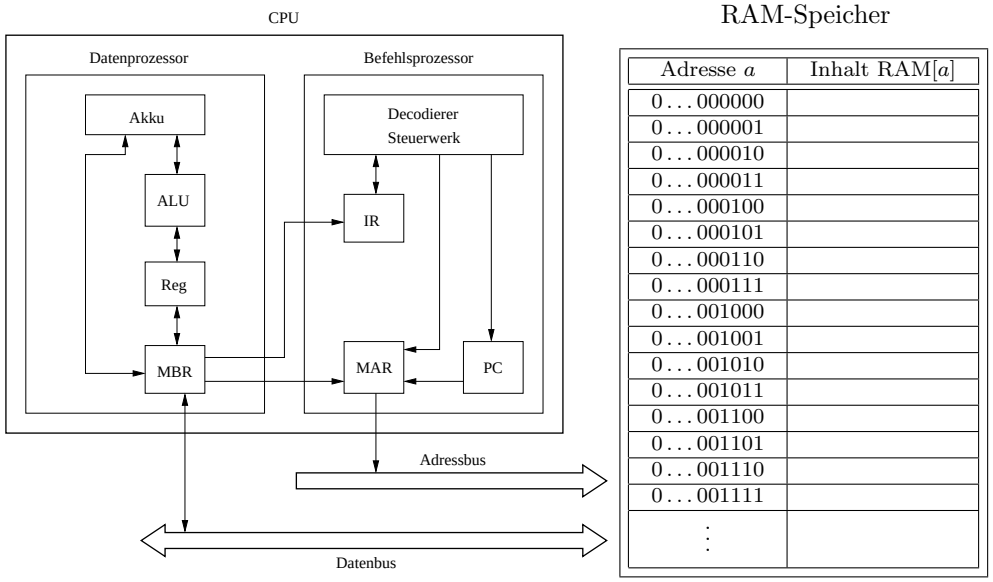
c)
- Register $42: 40
- Register $43: 21 (283div13)
- Register $44: 283 ($40 \cdot 7 + 3$)

`mmix -I einfach` führt `einfach.mms` bis zum HALT aus und schaltet in den interaktiven Modus. Hier lassen sich die Registerinhalte mit `l<n><t>` inspizieren (z.B. `l42`). Näheres in der Hilfe (im interaktiven Modus durch `h` aufrufbar).

d)
```
        LOC #100
x       IS $42
y       IS $43
z       IS $44

t1      IS $50
t2      IS $51

res     IS $14

Main    SET x,6
        SET y,7
        SET z,20

        MUL t1,x,y      % x*y nach t1
        ADD t2,x,y      % x+y nach t2
        MUL t1,t1,t2    % (x*y)*(x+y)=t1*t2 nach t1
        SUB t2,z,y      % z-y nach t2
        DIV res,t1,t2   % xy*(x+y) / (z-y) = t1/t2 nach res
        TRAP 0,Halt,0
```

Aufgabe 8: ($\star$) Arbeitsweise einer CPU

Die folgende Abbildung zeigt eine CPU-Struktur und ihre Verbindung zum RAM-Speicher. Im Vergleich zu der in der Vorlesung gezeigten Struktur einer CPU (VL 17 Folie 8), besitzt diese CPU ein zusätzliches Register Reg. Dafür wird hier zur Vereinfachung der genaue Aufbau des Registers Akku vernachlässigt: statt MR, L und A wird einfach nur Akku verwendet.



Für eine Speicheradresse a bezeichne RAM[a] die zugehörige Speicherzelle bzw. deren Inhalt. Nehmen Sie an, dass diesem Rechner die in der folgenden Tabelle durch ihre Operationscodes identifizierten Befehle/Operationen zur Verfügung stehen:

Operationscode	Wirkung(en) der Operation	
0	$(keineOperation)$	$\mathrm{PC} \leftarrow \mathrm{PC} + 1$
1	$\mathrm{RAM}[\mathrm{RAM}[\mathrm{PC} + 1]] \leftarrow \mathrm{Reg}$	$\mathrm{PC} \leftarrow \mathrm{PC} + 2$
2	$\mathrm{Reg} \leftarrow \mathrm{RAM}[\mathrm{RAM}[\mathrm{PC} + 1]]$	$\mathrm{PC} \leftarrow \mathrm{PC} + 2$
3	$\mathrm{Reg} \leftarrow \mathrm{RAM}[\mathrm{PC} + 1]$	$\mathrm{PC} \leftarrow \mathrm{PC} + 2$
4	$\mathrm{Reg} \leftarrow \mathrm{Akku}$	$\mathrm{PC} \leftarrow \mathrm{PC} + 1$
5	$\mathrm{Akku} \leftarrow \mathrm{Reg}$	$\mathrm{PC} \leftarrow \mathrm{PC} + 1$
6	$\mathrm{Akku} \leftarrow \mathrm{Akku} + \mathrm{Reg}$	$\mathrm{PC} \leftarrow \mathrm{PC} + 1$
7	$\mathrm{Akku} \leftarrow \mathrm{Akku} - \mathrm{Reg}$	$\mathrm{PC} \leftarrow \mathrm{PC} + 1$
8	$\mathrm{Akku} \leftarrow \mathrm{Akku} \times \mathrm{Reg}$	$\mathrm{PC} \leftarrow \mathrm{PC} + 1$
9	$\mathrm{Akku} \leftarrow \mathrm{Akku} \div \mathrm{Reg}$	$\mathrm{PC} \leftarrow \mathrm{PC} + 1$
10	$(unbedingterSprung)$	$\mathrm{PC} \leftarrow \mathrm{RAM}[\mathrm{PC} + 1]$
11	$\begin{cases} \text{Falls Akku} = 0: \\ \text{Falls Akku} \neq 0: \end{cases}$	$\mathrm{PC} \leftarrow \mathrm{RAM}[\mathrm{PC} + 1]$ $\mathrm{PC} \leftarrow \mathrm{PC} + 2$

Der Rechner kenne nur ganze Zahlen (keine Gleikomma-Zahlen). Dementsprechend steht $\div$ für die ganzzahlige Division (z. B. $26 \div 7 = 3$).

a) Beschreiben Sie, wozu die Operationen mit den Operationscodes 1, 2, 3, 10, 11 dienen.

Im folgenden „Speicher-Auszug“ sind neben den Dualzahlen (mit denen „echte“ Rechner arbeiten) auch die entsprechenden Dezimalzahlen (klein und in Klammern) geschrieben.

In Ihren Lösungen brauchen Sie aber nur Dezimalzahlen anzugeben.

Es sei angenommen, dass die CPU hier bei jeder Befehlsverarbeitung die folgenden „Zuweisungen“ (in dieser Reihenfolge) durchführt:

MAR ← PC
MBR ← RAM[MAR]
IR ← MBR
Wirkung(en) der Operation IR
(PC-Änderung zuletzt!)

b) Protokollieren Sie den Inhalt der Register Akku, Reg, IR und PC nach jeder Befehlsverarbeitung bis PC den Wert 25 enthält, wobei zu Anfang alle Register 0 enthalten und der Speicher wie nebenstehend gefüllt ist.

Hinweis: PC sollte den Wert 25 nach 14 Befehlsverarbeitungen enthalten.

c) Betrachten Sie nun den nebenstehenden Speicherinhalt von Adresse 0 bis Adresse 21 (einschließlich) und interpretieren Sie ihn als ein Programm. Was wird durch dieses Programm berechnet?

Adresse a	Inhalt RAM[a]
(0) 0...000000	0...00000010 (2)
(1) 0...000001	0...00010110 (22)
(2) 0...000010	0...00000101 (5)
(3) 0...000011	0...00000010 (2)
(4) 0...000100	0...00010111 (23)
(5) 0...000101	0...00000111 (7)
(6) 0...000110	0...00000100 (4)
(7) 0...000111	0...00000001 (1)
(8) 0...001000	0...00011000 (24)
(9) 0...001001	0...00001011 (11)
(10) 0...001010	0...00011001 (25)
(11) 0...001011	0...00000011 (3)
(12) 0...001100	0...01100100 (100)
(13) 0...001101	0...00000101 (5)
(14) 0...001110	0...00000010 (2)
(15) 0...001111	0...00011000 (24)
(16) 0...010000	0...00001001 (9)
(17) 0...010001	0...00000100 (4)
(18) 0...010010	0...00000001 (1)
(19) 0...010011	0...00011000 (24)
(20) 0...010100	0...00001010 (10)
(21) 0...010101	0...00011001 (25)
(22) 0...010110	0...00001011 (11)
(23) 0...010111	0...00000100 (4)
(24) 0...011000	0...00000000 (0)
⋮	⋮

Sie kennen vermutlich schon die folgende Anekdote aus dem Leben von C. F. Gauß (bzw. eine Variante davon):

> Um seine Schüler (bzw. Gauß) eine Zeitlang zu beschäftigen (bzw. zu bestrafen), gab der Lehrer ihnen (bzw. ihm) die Aufgabe, die Zahlen von 1 bis 100 zu addieren. Doch der kleine Gauß hatte die Lösung 5050 schon nach kurzer Zeit gefunden.
>
> Gauß hatte nämlich erkannt, dass man statt $1 + 2 + 3 + \cdots + 98 + 99 + 100$ zu rechnen auch 50 „Paare“ $1 + 100$, $2 + 99$, $3 + 98$, ..., $50 + 51$ bilden kann, die jeweils 101 ergeben, und sich so die Lösung $50 \cdot 101 = 5050$ leicht berechnen lässt.

D. h.: Gauß benutzte die (nach ihm benannte) Gaußsche Summenformel $\sum_{k=1}^{n} k = \frac{1}{2}n(n+1)$.

Schreiben Sie je ein Programm, das die Summe s der ersten n natürlichen Zahlen berechnet ...

d) ... durch *fortgesetzte Addition* (also nach der Formel $s = \sum_{k=1}^{n} k$);

e) ... in sog. *geschlossener Form* (z. B. nach der Formel $s = \frac{1}{2}n(n+1)$).

Das Ergebnis s soll bei Adresse 3 gespeichert werden, wobei n bei Adresse 2 gespeichert sei und das Programm selbst bei Adresse 4 beginnt. Sie können davon ausgehen, dass $n \geq 0$ gilt. Kommentieren Sie Ihr Programm jeweils unbedingt ausführlich!

Adresse a	Inhalt RAM[a]
(0) 0...000000	0...00001010 (10)
(1) 0...000001	0...00000100 (4)
(2) 0...000010	*...******** (*n*)
(3) 0...000011	*...******** (*s*)
⋮	⋮

Hinweis zu Aufgabenteil d):
Beachten Sie, dass die Summationsreihenfolge durch $\sum_{k=1}^{n} k$ nicht festgelegt ist.[3]

Hinweis zu Aufgabenteil e):
Beachten Sie, dass sich $\frac{1}{2}n(n+1)$ auf verschiedene Arten berechnen lässt.[3]

Lösungsvorschlag
Eine Zuordnung von Mnemocodes zu den Operationen könnte wie folgt aussehen:

Op.code	Wirkung(en) der Operation		'Mnemocode'
0		PC ← PC + 1	`skip`
1	RAM[RAM[PC + 1]] ← Reg	PC ← PC + 2	`store_REG`
2	Reg ← RAM[RAM[PC + 1]]	PC ← PC + 2	`load_REG`
3	Reg ← RAM[PC + 1]	PC ← PC + 2	`assign_REG`
4	Reg ← Akku	PC ← PC + 1	`copy_Akku_REG`
5	Akku ← Reg	PC ← PC + 1	`copy_REG_Akku`
6	Akku ← Akku + Reg	PC ← PC + 1	`add`
7	Akku ← Akku − Reg	PC ← PC + 1	`sub`
8	Akku ← Akku ∗ Reg	PC ← PC + 1	`mul`
9	Akku ← Akku ÷ Reg	PC ← PC + 1	`div`
10		PC ← RAM[PC + 1]	`goto`
11	Falls Akku = 0: Falls Akku ≠ 0:	PC ← RAM[PC + 1] PC ← PC + 2	`branch_on_zero`

a) Operation 1 = Speichern des Reg-Inhalts in den Speicher, wobei die Speicher-Adresse im auf den Operationscode folgenden Speicherwort enthalten ist

Operation 2 = Laden des Reg mit einem Speicherwort, wobei die Speicher-Adresse im auf den Operationscode folgenden Speicherwort enthalten ist

Operation 3 = Setzen des Reg auf den Wert des auf den Operationscode folgenden Speicherwortes (= Laden des Reg mit dem auf den Operationscode folgenden Speicherwort)

[3]Nutzen Sie die Assoziativität, Kommutativität und ggf. Distributivität der Rechenoperationen aus, um möglichst einfache und schnelle (aber natürlich dennoch korrekte) Programme zu schreiben.

Operation 10 = Sprungbefehl, wobei die Sprung-Ziel-Adresse im auf den Operationscode folgenden Speicherwort enthalten ist

Operation 11 = Verzweiungsbefehl = bedingter Sprungbefehl, wobei die Verzweiungs/Sprung-Ziel-Adresse im auf den Operationscode folgenden Speicherwort enthalten ist und der Verzweiungs-Sprung genau dann ausgeführt wird, wenn der Akku den Inhalt 0 hat

b)

	Akku	Reg	MAR	IR	PC	
0.	0	0	0	0	0	
1.	"	11	0	2	2	
2.	11	"	2	5	3	
3.	"	4	3	2	5	
4.	7	"	5	7	6	
5.	"	7	6	4	7	
6.	"	"	7	1	9	RAM[24] = 7
7.	"	"	9	11	11	
8.	"	100	11	3	13	
9.	100	"	13	5	14	
10.	"	7	14	2	16	
11.	14	"	16	9	17	
12.	"	14	17	4	18	
13.	"	"	18	1	20	RAM[24] = 14
14.	"	"	20	10	25	

c) (Notation: $\&v$ = Adresse von v, $*v$ = Wert von v)

a	RAM[a]	Assemblercode	Befehl
0	2	`load_REG`	Reg $\leftarrow$ RAM[22] $=: x$
1	22	22 $\widehat{=}\ \&x$	
2	5	`copy_REG_Akku`	Akku $\leftarrow x$
3	2	`load_REG`	Reg $\leftarrow$ RAM[23] $=: y$
4	23	23 $\widehat{=}\ \&y$	
5	7	`sub`	Akku $\leftarrow x - y$
6	4	`copy_Akku_REG`	Reg $\leftarrow x - y$
7	1	`store_REG`	RAM[24] $\leftarrow x - y$
8	24	24 $\widehat{=}\ \&z$	
9	11	`branch_on_zero`	springe zu 25, falls Akku $= 0$
10	25	25 $\widehat{=}$ ...	
11	3	`assign_REG`	Reg $\leftarrow 100$
12	100	100	
13	5	`copy_REG_Akku`	Akku $\leftarrow 100$
14	2	`load_REG`	Reg $\leftarrow$ RAM[24] $= x - y$
15	24	24 $\widehat{=}\ \&z$	
16	9	`div`	Akku $\leftarrow 100 \div (x - y)$
17	4	`copy_Akku_REG`	Reg $\leftarrow 100 \div (x - y)$
18	1	`store_REG`	RAM[24] $\leftarrow 100 \div (x - y)$
19	24	24 $\widehat{=}\ \&z$	
20	10	`goto`	springe zu 25
21	25	25 $\widehat{=}$ ...	
22	11	11 $\widehat{=}\ *x$	
23	4	4 $\widehat{=}\ *y$	
24	0	0 $\widehat{=}\ *z$	
$\vdots$	$\vdots$		

d) $\underline{[(n{+}1) * n] \div 2}$:

```
begin:
    load_REG n
    copy_REG_Akku
    assign_REG 1
    add
    load_REG n
    mul
    assign_REG 2
    div
    copy_Akku_REG
    store_REG s
end:
```

$\underline{[(1{+}n) * n] \div 2}$:

```
begin:
    assign_REG 1
    copy_REG_Akku
    load_REG n
    add

    mul
    assign_REG 2
    div
    copy_Akku_REG
    store_REG s
end:
```

$\underline{[[n * n] + n] \div 2}$:

```
begin:

    load_REG n
    copy_REG_Akku
    mul

    add
    assign_REG 2
    div
    copy_Akku_REG
    store_REG s
end:
```

$[\cdots[n+(n-1)]+\cdots+2]+1$:

```
begin:
    assign_REG 0
    store_REG s

loop:
    load_REG n
    copy_REG_Akku


    branch_on_zero end
compute_sum:
    load_REG s
    add
    copy_Akku_REG
    store_REG s
decrement_n:
    load_REG n
    copy_REG_Akku
    assign_REG 1
    sub
    copy_Akku_REG
    store_REG n
    goto loop
end:
```

$[\cdots[1+2]+\cdots+(n-1)]+n$:

```
begin:
    assign_REG 0
    store_REG s
    store_REG k
loop:
    load_REG k
    copy_REG_Akku
    load_REG n
    sub
    branch_on_zero end
increment_k:
    load_REG k
    copy_REG_Akku
    assign_REG 1
    add
    copy_Akku_REG
    store_REG k
compute_sum:
    load_REG s
    add
    copy_Akku_REG
    store_REG s
    goto loop
end:
```

e)

a	RAM[a]	Assemblercode	Befehl
0	10	`goto`	} springe zu 4
1	4	4 $\widehat{=}$ begin	
2		$\widehat{=} *n$	
3		$\widehat{=} *s$	
		begin:	
		numbersum:	
4	2	`load_REG`	} Reg $\leftarrow n$
5	2	2 $\widehat{=} \&n$	
6	5	`copy_REG_Akku`	Akku $\leftarrow n$
7	8	`mul`	Akku $\leftarrow n * n$
8	6	`add`	Akku $\leftarrow [n * n] + n$
9	3	`assign_REG`	} Reg $\leftarrow 2$
10	2	2	
11	9	`div`	Akku $\leftarrow [[n * n] + n] \div 2$
12	4	`copy_Akku_REG`	Reg $\leftarrow [[n * n] + n] \div 2$
13	1	`store_REG`	} RAM[3] $\leftarrow [[n * n] + n] \div 2 \quad =: s$
14	3	3 $\widehat{=} \&s$	
		end:	
$\vdots$	$\vdots$		