

**Universität Duisburg-  
Essen**

Fak. 5, Abt. IIMT, Institut für  
Informationstechnik (IT)

## **PRAKTIKUM ZUR VORLESUNG**

# **Rechnergesteuerte Systeme 2**

## **VERSUCH 5**

### **RECHNERMODELL UND MASCHINENPROGRAMMIERUNG**

Name:	Matrikelnummer:
Vorname:	Gruppennummer:
Betreuer:	Datum:

## Hinweise

- Arbeiten Sie den anliegenden Vorbereitungstext sorgfältig durch.
- Lösen Sie die im Vorbereitungstext gestellten Aufgaben und bringen Sie Ihre Ergebnisse zum Versuchstermin mit.

---

## Inhalt

1	Einleitung .....	3
2	Die Ablaufsteuerung .....	3
2.1	Schnittstelle .....	4
2.2	Befehlsdekoder .....	4
2.3	Automatenmodellierung .....	5
2.4	Aufgabe 1 .....	7
3	Die CPU .....	9
3.1	Schnittstelle .....	9
3.2	Aufgabe 2 .....	10
4	Der Speicher .....	10
4.1	Schnittstelle .....	10
4.2	Implementierung .....	10
4.3	Aufgabe 3 .....	12
5	Das System .....	12
5.1	Aufgabe 4 .....	12
6	Maschinenprogrammierung .....	13
6.1	Aufgabe 5 .....	15

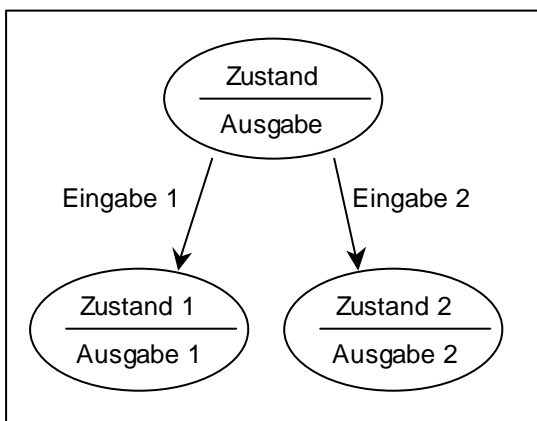
## 1 Einleitung

In diesem Versuch wird zunächst die Ablaufsteuerung des Prozessors entworfen. Danach wird die Schnittstelle des Prozessors spezifiziert und dessen Architektur aus den in dieser Versuchsreihe bisher vorgestellten Komponenten erstellt. Diesem Schritt folgend wird auch ein einfaches Modell eines Speichers(RAM) entworfen. Der Prozessor und das Speichermodell können dann zu einem Gesamtsystem zusammengefasst werden. Anschließend werden kleinere Programme auf Maschinensprachebene erstellt, anhand derer das Gesamtsystem simuliert wird.

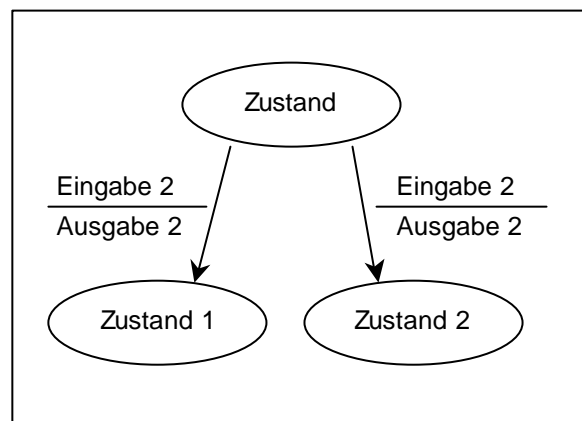
## 2 Die Ablaufsteuerung

Die Prozessor-Struktur wurde ja bereits im Versuch 4 mit der schrittweisen Ausführung einiger Befehle getestet. Die genaue Analyse aller Prozessorbefehle anhand der schrittweisen Ausführung ist der erste Schritt zur Aufstellung der gesamten Ablaufsteuerung. Bei der Realisierung der Ablaufsteuerung wird zwischen zwei verschiedenen Arten von Steuerung unterschieden: Mikroprogrammsteuerung und festverdrahtete Steuerung. Gegenstand des ersten Teils dieses Versuchs ist der Entwurf einer festverdrahteten Steuerung für die bekannte Prozessor-Struktur.

Allgemein gesehen ist jede Ablaufsteuerung als Schaltwerk bzw. Automaten implementiert, dessen Zustand eine Situation im Steuerungsablauf repräsentiert. Es gibt zwei grundsätzliche Typen von Schaltwerken(Automaten): Bei einem *Moore-Automaten*(Abb. 1) hängt die Ausgabe nur vom aktuellen Zustand ab, beim *Mealy-Automaten*(Abb. 2) hängt sie sowohl vom Zustand als auch von der aktuellen Eingabe ab. Der Moore-Automat ist somit ein Spezialfall des Mealy-Automaten.



**Abb. 1: Moore-Automat**



**Abb. 2: Mealy-Automat**

Ohne weiter auf die Automatentheorie einzugehen sei hier angemerkt, dass bei der hiesigen Implementierung der Ablaufsteuerung ein Mealy-Automat zu Grunde gelegt wird, da er, im Vergleich zu einem funktionsgleichen Moore-Automaten, mit einer kleineren Anzahl an Zuständen auskommt.

Zur Beschreibung von Schaltwerken/Automaten gibt es verschiedene Möglichkeiten: üblich sind Zustandsübergangsdigrammen(siehe Abb. 1 und 2) und Automatentabellen. Die Automatentabelle der Ablaufsteuerung enthält neben Zuständen und Folgezuständen den erkannten Opcode als Eingabe und die Steuersignale als Ausgabe. Um eine einfach zu überblickende Tabelle zu bilden, ist es zunächst vonnöten den Befehlssatz des Prozessors genau unter die Lupe zu nehmen und Befehlsgruppen zu bilden, die gleiche Arbeitsphasen aufweisen. Hierzu ist es hilfreich die bereits im Versuch 4 beschriebenen drei groben Verarbeitungsschritte des Prozessors zu vergegenwärtigen. Die dort beschriebenen Arbeitsphasen kann man als Zustandsübergänge des nun zu entwerfenden Mealy-Automaten auffassen.

Die ersten Arbeitsphasen(Befehl holen, Befehl dekodieren) sind offensichtlich für alle Befehle gleich:

### 1. Befehl holen

Hierbei werden anhand der entsprechenden Steuersignale folgende Maßnahmen durchgeführt:

- Bidirektionalen Buffer(BB) auf „lesen“ stellen
- Instruktionsregister(IR) laden
- Program Counter(PC) erhöhen

## 2. Befehl dekodieren

An dieser Stelle übernimmt der Befehlsdekoder, der bei der vorliegenden Prozessor-Architektur innerhalb der Einheit Ablaufsteuerung(CON) implementiert werden soll, das Instruktionswort aus dem IR und erkennt um welchen Befehl es sich dabei handelt. In dieser Arbeitsphase braucht also weder der PC erhöht, noch andere Register geladen zu werden. Lediglich sollte der bidirektionale Buffer im Modus „lesen“ bleiben.

Die dritte grob beschriebene Arbeitsphase „**Befehl ausführen**“ muss weiter in zwei Verarbeitungsschritte unterteilt werden.

(3a.) Bei den 1-wort-(NOT, SLL und SRA), Immediate- und Sprungbefehlen ist die Befehlsausführung im dritten Schritt abgeschlossen, somit ist der Prozessor wieder bereit für die Arbeitsphase „Befehl holen“.

(3b.) Bei den Direktbefehlen(bzw. Absolutbefehlen) hingegen muss ein vierter Arbeitsschritt angehängt werden, um den Operanden adressieren zu können. Die zu setzenden Steuersignale sind ab dem dritten Arbeitsschritt natürlich unterschiedlich je nach Befehle bzw. Befehlsgruppen.

*Hinweis:* Zur Erinnerung seien die Begriffe *immediate* und *absolute* im Zusammenhang mit den Befehlsgruppen kurz erläutert. Sowohl Immediate- als auch Absolutbefehle sind Zwei-Wort-Befehle, wobei im ersten Wort immer das Instruktionswort steht. Während dieses Instruktionswort dekodiert wird, zeigt der PC durch die Auswirkung der ersten Arbeitsphase bereits auf das darauffolgende Wort.

Bei den Immediate-Befehlen steht im zweiten Wort bereits der Operand.

Bei den Absolutbefehlen steht im zweiten Wort die Adresse des Operanden.

### 2.1 Schnittstelle

Die Schnittstellensignale der Ablaufsteuerung wurden bereits im letzten Versuch beschrieben. Zur Erinnerung sei die Schnittstelle hier noch einmal genau spezifiziert.

Die Ablaufsteuerung soll den `entity`-Namen **controller** tragen.

Anschluss[Breite]	Richtung	Typ
i[4]	Eingang	std_logic_vector
res,clk,cf,zf	Eingang	std_logic
inc,lac,lad,lf,lir,lpc,smx,samx,rw	Ausgang	std_logic

### 2.2 Befehlsdekoder

Zunächst sollen die Befehle des Prozessors in Gruppen eingeteilt werden, um die VHDL-Modellierung übersichtlich zu halten. Folgende Tabelle zeigt drei Gruppen von Befehlen, die jeweils einen Gruppennamen tragen. Die restlichen Befehle werden weiterhin durch die Mnemonik aus der bereits im Versuch 4 vorgestellten Befehlssatz repräsentiert.

Gruppenname	Befehle
ALU	NOT, SLL, SRA
ALU_IMM	ADDIMM, SUBIMM, ANDIMM, ORIMM
ALU_ABS	ADDABS, SUBABS, ANDABS, ORABS

Zur Verwendung dieser Mnemonik kann innerhalb der Architektur von **controller** ein Aufzählungstyp `opcode_Type` deklariert werden. Ein lokales Signal `opcode` von diesem Typ dient dazu, den erkannten Opcode festzuhalten.

```
type Opcode_Type is (ALU,ALU_IMM,ALU_ABS,LDIMM,LDABS,STABS,JMP,JZ,JC);
signal opcode : Opcode_Type;
```

Der Befehlsdekoder wird als Prozess modelliert, der von den Signalen `res` und `clk` angestoßen wird. Initialisiert wird der Dekoder mit dem Befehlstyp `ALU`. Mittels eines `case`-Konstrukts wird abhängig vom Instruktionswort der Opcode „erkannt“. Folgender Quellcode-Abschnitt zeigt einen Vorschlag zur Modellierung des Befehlsdekoders:

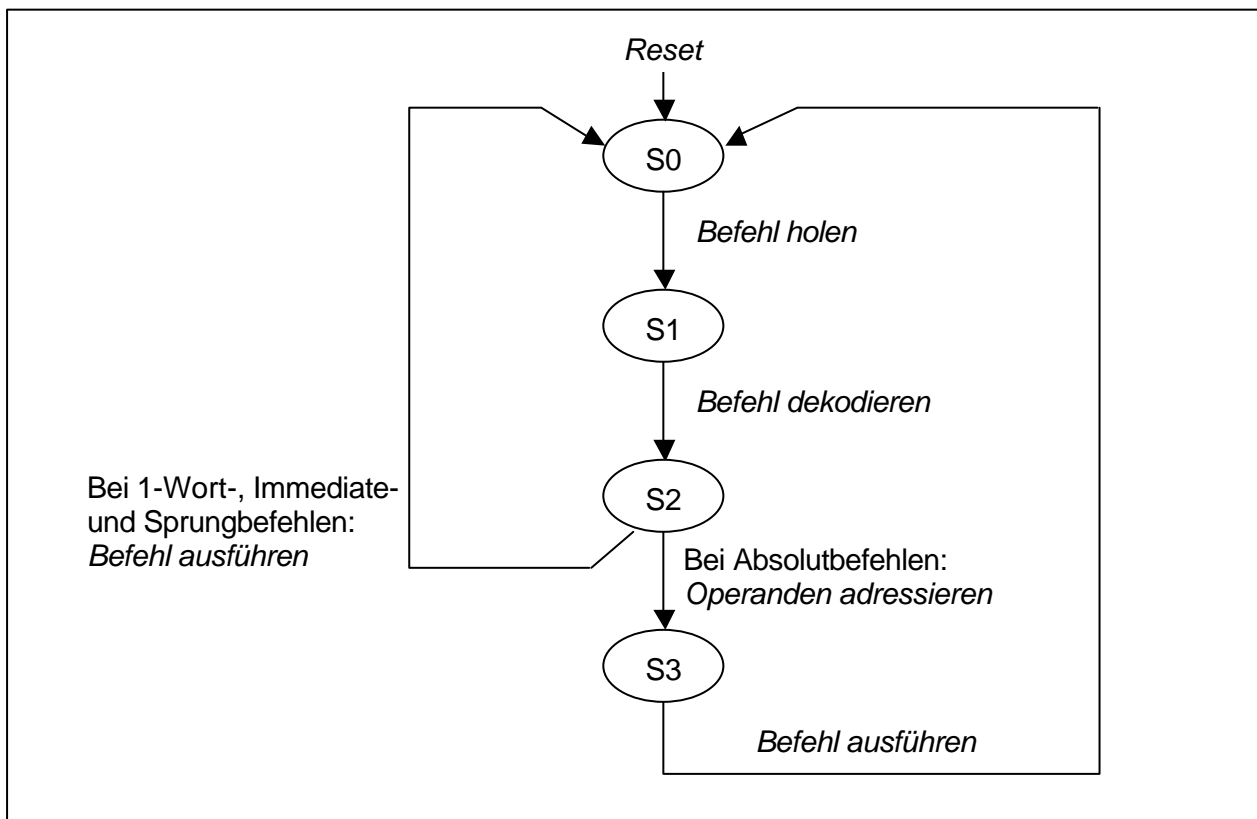
```

opcode_decode: process(clk,res) is -- Befehlsdekoder
begin
  if res = '1' then
    opcode <= ALU;
  elsif clk'event and clk = '1' then
    case i is
      -- NOT,SLL,SRA
      when "0000" => opcode <= ALU;
      -- ADDIMM,SUBIMM,ANDIMM,ORIMM
      when "0001"|"0011"|"0101"|"0111" => opcode <= ALU_IMM;
      ...
      when "1001" => opcode <= LDIMM;
      ...
      when "1110" => opcode <= JC;
      when others => opcode <= ALU;
    end case;
  end if;
end process opcode_decode;

```

### 2.3 Automatenmodellierung

Zur Modellierung der Ablaufsteuerung in Form eines Mealy-Automaten müssen nun alle Zustände und Zustandsübergänge spezifiziert werden. Aus den oben beschriebenen Arbeitsphasen geht hervor, dass es Befehle gibt, die maximal vier Arbeitsphasen bzw. Zustandsübergänge benötigen. Daraus ergibt sich die Anzahl der nötigen Zustände zu vier. Diese seien S0, S1, S2 und S3 benannt. Für jeden Zustandsübergang braucht der Automat genau einen Takt; bei dem gegebenen Befehlssatz des Prozessors benötigen die Absolutbefehle also vier Takte, während alle anderen Befehle mit drei Takten auskommen. Folgende Abbildung gibt einen Überblick.



**Abb. 3 : Zustandsübergangsdiagramm**

Dieses Zustandsübergangsdiagramm kann nun beliebig verfeinert werden, sodass an den Übergängen genaue Angaben über Ein- und Ausgangssignale stehen. Zur besseren Handhabung empfiehlt es sich,

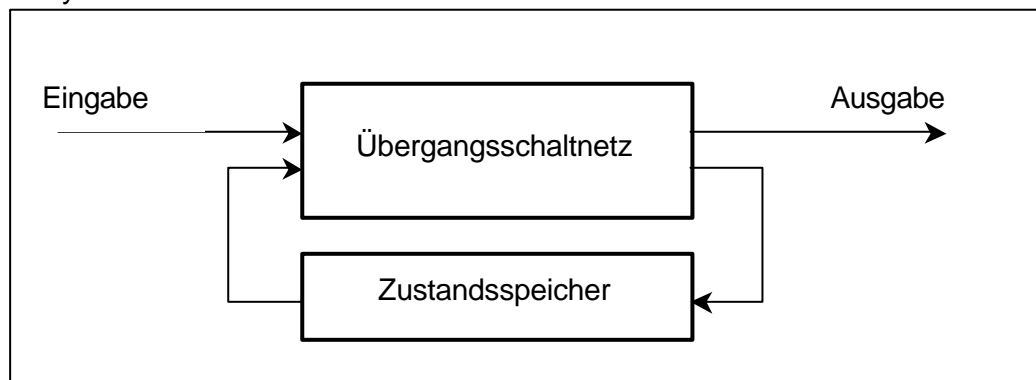
eine Automatentabelle zu verwenden, in der die Zustände und Folgezustände, sowie die zugehörigen Ein- und Ausgangssignale angegeben sind. Solch eine Tabelle lässt sich folgendermaßen aufbauen:

state	opcode	next_state	inc	lac	lad	lf	lir	Lpc	smx	samx	rw
S0	-	S1	1	0	0	0	1	0	0	0	1
S1	-	S2	0	0	0	0	0	0	0	0	1
S2	ALU	S0	0	1	0	1	0	0	0	0	1
S2	ALU_IMM	S0	1	1	0	1	0	0	0	0	1
S2	LDIMM	S0									
S2	JMP	S0	0	0	0	0	0	1	0	0	1
S2	JZ	S0	$\neg$ ZF	0	0	0	0	ZF	0	0	1
S2	JC	S0									
S2	ALU_ABS	S3	0	0	1	0	0	0	0	0	1
S3	ALU_ABS	S0	1	1	0	1	0	0	0	1	1
S2	LDABS	S3	0	0	1	0	0	0	0	0	1
S3	LDABS	S0									
S2	STABS	S3	0	0	1	0	0	0	0	0	1
S3	STABS	S0	1	0	0	0	0	0	0	1	0

**Tabelle 1: Automatentabelle**

Die Spalte „opcode“ stellt hier das Eingangssignal des Automaten dar; die Steuersignale *inc*, *lac*, *lad*, *lf*, *lir*, *lpc*, *smx*, *samx* und *rw* sind als Ausgabe des Automaten zu betrachten.

Automaten können auf verschiedene Weise aufgebaut werden. Es gibt jedoch zwei Grundelemente, die alle Automaten gemeinsam haben: Zustandsspeicher und Schaltnetz(e). Die Abb. 4 zeigt einen üblichen Aufbau eines Mealy-Schaltwerks bzw. -Automaten.



**Abb. 4 : Aufbau eines Mealy-Automaten**

Die dargestellten Elemente des Automaten, der Zustandsspeicher und das Übergangsschaltnetz, können jeweils mittels eines Prozesses modelliert werden. Zur Kommunikation zwischen diesen Prozessen müssen lokale Signale definiert werden. Neben dem oben im Zusammenhang mit dem Befehlsdeko-der definierten Signal *opcode* müssen nun lokale Signale für den Zustand und den Folgezustand definiert werden.

```

type State_Type is (S0,S1,S2,S3);
signal state, next_state : State_Type;
  
```

Die Architektur der Ablaufsteuerung, die sowohl den Befehlsdekoder als auch den Mealy-Automaten beherbergt, enthält also drei miteinander über lokale Signale kommunizierenden Prozesse:

```
architecture arch of controller is

    type Opcode_Type is (ALU,ALU_IMM,ALU_ABS,LDIMM,LDABS,STABS,JMP,JZ,JC);
    signal opcode : Opcode_Type;

    type State_Type is (S0,S1,S2,S3);
    signal state, next_state : State_Type;
begin
    opcode_decode: process(clk,res) is -- Befehlsdekoder
    begin
        ...
    end process opcode_decode;

    memory: process(clk,res) is -- Zustandsspeicher
    begin
        if res = '1' then
            state <= S0; -- ersten Befehl holen
        elsif clk'event and clk = '1' then
            state <= next_state;
        end if;
    end process memory;

    transition: process(state,opcode,cf,zf) is -- Übergangsschaltnetz
    begin
        ...
    end process transition;
end arch;
```

## 2.4 Aufgabe 1

Realisieren Sie die Ablaufsteuerung anhand der oben gegebenen Spezifikationen. Gehen Sie dabei wie folgt vor.

- 1.1 Erstellen Sie die entity **controller** mit den im Abschnitt 2.1 beschriebenen Schnittstelle.
- 1.2 Realisieren Sie den Befehlsdekoder(als eingebetteter Prozess), indem Sie im Abschnitt 2.2 gegebenen VHDL-Code vervollständigen. Schauen Sie hierzu den Befehlssatz des Prozessors (Versuch 4, Tabelle 1) genau an.
- 1.3 Vervollständigen Sie die Automatentabelle im Abschnitt 2.3
- 1.4 Ausgehend von der Automatentabelle erstellen Sie eine neue Tabelle nach der folgenden Vorlage.

state	opcode	next_state	inc	lac	lad	lf	lir	lpc	smx	samx	rnw
S0/Init	-	S1	1	0	0	0	1	0	0	0	1
S1	-	S2	0	-	-	-	0	-	-	-	-
S2	ALU	S0	0	1	-	1	0	-	-	-	-
S2	ALU_IMM	S0	-	1	-	1	0	-	-	-	-
S2	LDIMM	S0									
S2	JMP	S0	0	-	-	-	0	1	-	-	-
S2	JZ	S0	$\neg$ ZF	-	-	-	0	ZF	-	-	-
S2	JC	S0									

S2	ALU_ABS	S3	0	-	1	-	0	-	-	-	-
S3	ALU_ABS	S0	-	1	-	1	0	-	-	1	-
S2	LDABS	S3									
S3	LDABS	S0									
S2	STABS	S3	0	-	1	-	0	-	-	-	-
S3	STABS	S0	-	-	-	-	0	-	-	1	0

Tabelle 2: Automatentabelle (zur VHDL-Umsetzung)

Diese Tabelle ist zur Umsetzung des Automaten in VHDL sehr hilfreich. Das Zeichen "-" hat hierbei die Bedeutung: *Keine Änderung des Steuersignals bezogen auf den Wert, auf den das betreffende Steuersignal beim Zustandsübergang von S0 nach S1 gesetzt wurde.*

- 1.5 Zur Modellierung des Übergangsschaltnetzes sei nun folgende Vorlage gegeben. Vervollständigen Sie den VHDL-Code unter Zuhilfenahme der im Unterpunkt 1.4 erstellten Tabelle.

```

transition: process(state,opcode,cf,zf) is          -- Übergangsschaltnetz
begin
  -- S0/init
  inc <= '1';
  lac <= '0';
  lad <= '0';
  lf <= '0';
  lir <= '1';
  lpc <= '0';
  smx <= '0';
  samx <= '0';
  rnw <= '1';
  case state is
    when S0 =>
      next_state <= S1;
    when S1 =>
      next_state <= S2;
      inc <= '0';
      lir <= '0';
    when S2 =>
      lir <= '0';
      next_state <= S0;
      case opcode is
        when ALU =>
          inc <= '0';
          lac <= '1';
          lf <= '1';
        when ALU_IMM =>
          lac <= '1';
          lf <= '1';
        when LDIMM =>
          ...
        when JMP =>
          inc <= '0';
          lpc <= '1';
        when JZ =>
          inc <= not zf;
          lpc <= zf;
        when JC =>
          ...
      end case;
    when others =>          -- ALU_ABS | LD_ABS | STABS

```

```

        next_state <= S3;
        inc <= '0';
        lad <= '1';
    end case;
when others =>          -- S3
    next_state <= S0;
    lir <= '0';
    case opcode is
        when ALU_ABS =>
            lac <= '1';
            lf <= '1';
            samx <= '1';
        when LDABS =>
            ...
        when STABS =>
            samx <= '1';
            rnw <= '0';
        when others => null;
    end case;
end case;
end process transition;

```

1.6 Stellen Sie nun die komplette Architektur der Ablaufsteuerung zusammen.

### 3 Die CPU

Da nun alle Komponenten des Prozessors vorhanden sind, kann die `entity cpu` erstellt werden. Bevor die Schnittstelle genau spezifiziert wird, soll auf Folgendes aufmerksam gemacht werden: Das Richtungssteuerungssignal(`rw`) der Ablaufsteuerung dient sowohl dem bidirektionalen Buffer als auch der Memory als Eingangssignal; siehe folgende Abbildung.

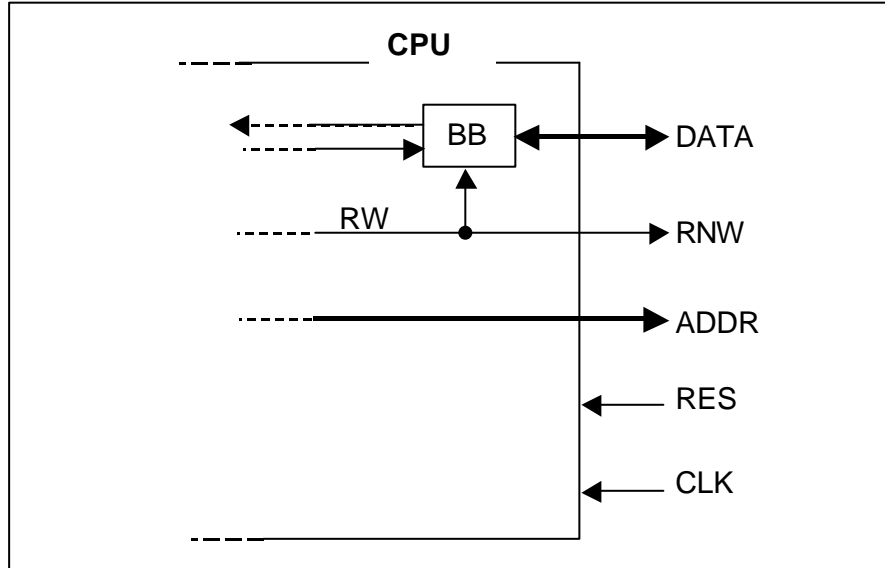


Abb. 5: Schnittstelle der CPU

#### 3.1 Schnittstelle

Anschluss[Breite]	Richtung	Typ
addr[8]	Eingang	std_logic_vector
res,clk	Eingang	std_logic
rnw	Ausgang	std_logic
data[8]	Bidirektional	std_logic_vector

### 3.2 Aufgabe 2

2.1 Erstellen Sie die `entity` **cpu** anhand der Schnittstellenbeschreibung.

2.2 Erstellen Sie die Architektur von **cpu** gemäß Abb. 2 und 3 in der Anleitung zu Versuch 4.

*Hinweis:* Zum Aufbau der Architektur kann der VHDL-Code aus Aufgabe 2 im Versuch 4 weitgehend übernommen werden; einzige neue Komponente ist die Ablaufsteuerung(CON).

## 4 Der Speicher

Im vorherigen Versuch wurden die Daten/Befehle über die Testbench „manuell“ an das Bussystem des Prozessors gelegt, nun soll dies durch ein Speichermodell erzielt werden. Der Speicher soll als RAM (Random Access Memory) arbeiten, d.h., sowohl Lese- als auch Schreiboperationen sind bei beliebiger Adressierung jederzeit möglich. Um die Maschinenprogrammierung zu erleichtern, soll der Speicherinhalt bei einem Reset-Signal aus einer Textdatei geladen und in ein Speicherzellenarray geschrieben werden.

### 4.1 Schnittstelle

Schnittstellen des Speichermodells sollen folgende Leitungen bilden:

Anschluss[Breite]	Richtung	Typ
ma[8]	Eingang	std_logic_vector
md[8]	Bidirektional	std_logic_vector
rnw	Eingang	std_logic
reset	Eingang	std_logic

### 4.2 Implementierung

Es soll ein neuer Variablentyp Namens `Mem_Type` deklariert werden, der einen Array von `std_logic_vector` bildet, die als Speicherzellen fungieren sollen. Die Deklaration und ein Beispiel zur Adressierung sind in den nächsten Code-Zeilen wiederzufinden:

```
type Mem_Type is array(0 to 255) of std_logic_vector(7 downto 0);

variable mem : Mem_Type := (others => x"00");

begin
    mem(0) := "00000001";
    mem(1) := "00000010";
    mem(2) := "00000011";
    mem(3) := "00000100";
    mem(4) := "00000101";
    mem(5) := "00000111";
    ...
```

Beim Rücksetzen(Reset) des Speichers soll die Prozedur `mem_read_in` aktiviert werden, um den Speicherinhalt aus der Datei „memin.log“ in das Zellenarray zu laden. Diese Prozedur verwendet Werkzeuge aus dem Paket `Textio` der Bibliothek `Std` zur Manipulation von Text-Dateien und baut sich wie folgt auf:

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use std.textio.all;

...
type Mem_Type is array(0 to 255) of std_logic_vector(7 downto 0);
```

```

procedure mem_read_in( mem_box : inout Mem_Type) is
  file login : text open read_mode is "memin.log";
  variable akt_line : line;
  variable address : integer;
  variable mem_line : std_logic_vector(7 downto 0);
begin
  mem_box := (others => x"00");
  while not endfile(login) loop
    readline(login,akt_line);
    read(akt_line,address);
    read(akt_line,mem_line);
    mem_box(address) := mem_line;
  end loop;
end procedure mem_read_in;
...

```

Mit der Zeile

```

file login : text open read_mode is "memin.log";

```

wird ein Dateizeiger mit dem Namen `login` deklariert, der auf eine Datei im Text-Format zeigt, die im Lesemodus geöffnet ist und den Dateinamen `memin.log` trägt. Zur Auslese ist weiterhin eine Variable des Typs `line` notwendig. Der Auslesevorgang erfolgt Zeilenweise mit dem Befehl `readline` unter Angabe des Dateizeigers und einer Zeilenvariable: `readline (login,akt_line)`. Jede Zeile wird vollständig in einem Schritt ausgelesen und dann mit dem Befehl `read` durchkämmt. Dabei werden Abschnitte der Zeile in andere lokale Variablen abgespeichert, womit die Auslese Variablen verschiedenen Typs in einer einzigen Zeile möglich werden. Hier wird zunächst eine Integervariable bis zum ersten Leerzeichen ausgelesen, daran anschließend wird ein `std_logic_vector` ausgelesen. Die Ausleseschleife wird beendet, wenn das Ende der Datei erreicht ist: `endfile(login)`.

Inhalt der Datei "memin.log" sind mit Speicheradressen versehenen Daten und Befehle. Die Formatierung dieser Datei sei wie folgt zu wählen: Linksbündig soll in der ASCII Datei zunächst die Adresse im Dezimalformat stehen. Daran anschließend ein (oder mehrere) Leerzeichen und der Inhalt der Speicherzelle im Binärformat (8 Bit). Diese Datei kann mit jedem beliebigen Text-Editor geschrieben werden, solange die festgelegte Formatierung beibehalten wird. Als Beispiel sei gegeben:

```

000 00011001
010 10101010
011 11111111
012 00001111

```

Das Speichermodell kann nach folgendem Algorithmus erstellt werden:

```

Prozess ram, sensitiv auf (ma,rnw und reset)
Mem_Type deklarieren und initialisieren
Variable mem initialisieren

Begin des Prozesses
wenn reset gleich '1' dann
  starte Prozedur mem_read_in mit der Variable mem;
sonst
  wenn (fallende Flanke von rnw) dann
    md in Hochohmigen Zustand -- notwendig, sonst Buskonflikt 'X' beim Schreiben
  sonst
    wenn rnw gleich '1' dann
      Inhalt von mem an der Adresse ma nach md
    sonst_wenn rnw gleich '0' dann
      Inhalt von md an Adresse ma des Feldes mem
    ende wenn;

```

```
    ende wenn;  
  ende wenn;  
ende Prozess ram;
```

### 4.3 Aufgabe 3

Entwerfen Sie ein Speichermodell in VHDL mit den oben beschriebenen Eigenschaften.

## 5 Das System

Ein funktionsfähiges Rechnersystem kann nun mit den bisher behandelten Bausteinen – Prozessor, Speicher und Taktgenerator – aufgebaut werden.

### 5.1 Aufgabe 4

Modellieren Sie einen Rechner in VHDL, indem Sie das Prozessormodell, das Speichermodell und einen Taktgenerator zu einem System zusammenfügen. Orientieren Sie sich an folgender Vorlage:

```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity system is  
end system;  
  
architecture arch of system is  
  
  signal clk,reset,rnw : std_logic;  
  signal addr,data : std_logic_vector(7 downto 0);  
  
  component cpu is  
  ...  
end component;  
  
  component memory is  
  ...  
end component;  
  
  for CPU1: ...  
  for MEM1: ...  
  
begin  
  CPU1: cpu port map ...  
  MEM1: memory port map ...  
  
  gen_clock: process is  
  begin  
    clk <= '0';  
    wait for 50 ns;  
    clk <= '1';  
    wait for 50 ns;  
  end process gen_clock;  
  
  start: process is  
  begin  
    reset <= '1';  
    wait for 100 ns;  
    reset <= '0';  
    wait;  
  end process start;  
end arch;
```

## 6 Maschinenprogrammierung

Bei der hier simulierten Hardware handelt es sich um einen "programmgesteuerten Universalrechner", der einen statischen - von den Aufgabenstellungen unabhängigen - Aufbau aufweist. Die Anpassung an die Aufgabenstellung erfolgt über die im Speicher abgelegten Programme, sog. Maschinenprogramme. Diese bestehen aus Maschinenbefehlen, die zum Befehlssatz des Prozessors gehören und interne Abläufe desselben steuern. Mit der Maschinenprogrammierung ist es möglich, komplexe Abläufe auf einer höheren Ebene der Abstraktion zu realisieren. Ein noch höheres Abstraktionsniveau kann durch höhere Programmiersprachen erreicht werden, deren Anweisungen über einen Compiler in Maschinenprogramme übersetzt werden.

Im hiesigen Versuch soll auf der Maschinenprogrammebene gearbeitet werden, somit muss der Speicher des Rechners mit Maschinenbefehlen und Daten geladen werden. Um das Rechnermodell „universal“ zu gestalten, wird der Speicherinhalt aus einer Text-Datei geladen.

Mit dem schon bekannten Befehlsatz

Assembler	Mnemonic	Codierung	Kurzerklärung
NOT	NOTA	0000101	$\neg$ ACCU $\rightarrow$ ACCU
SLL	SLLA	0000110	(ACCU $\ll$ 1) $\rightarrow$ ACCU
SRA	SRAA	0000111	(ACCU $\gg$ 1) $\rightarrow$ ACCU
ADD #value	ADDIMM	00010001 value	ACCU + value $\rightarrow$ ACCU
ADD addr	ADDABS	00100001 addr	ACCU + MEM(addr) $\rightarrow$ ACCU
SUB #value	SUBIMM	00110010 value	ACCU - value $\rightarrow$ ACCU
SUB addr	SUBABS	01000010 addr	ACCU - MEM(addr) $\rightarrow$ ACCU
AND #value	ANDIMM	01010011 value	ACCU $\wedge$ value $\rightarrow$ ACCU
AND addr	ANDABS	01100011 addr	ACCU $\wedge$ MEM(addr) $\rightarrow$ ACCU
OR #value	ORIMM	01110100 value	ACCU $\vee$ value $\rightarrow$ ACCU
OR addr	ORABS	10000100 addr	ACCU $\vee$ MEM(addr) $\rightarrow$ ACCU
LD #value	LDIMM	10010000 value	value $\rightarrow$ ACCU
LD addr	LDABS	10100000 addr	MEM(addr) $\rightarrow$ ACCU
ST addr	STABS	10110000 addr	ACCU $\rightarrow$ MEM(addr)
JMP addr	JMP	11000000 addr	Absoluter Sprung nach addr
JZ addr	JZ	11010000 addr	Sprung nach addr falls ZF=1
JC addr	JC	11100000 addr	Sprung nach addr falls CF=1

**Tabelle 3: Befehlssatz des Prozessors**

sollen nun Maschinenprogramme zur Lösung komplexerer Aufgaben geschrieben werden.

Als Beispiel sei folgende Aufgabe mit dem Rechnermodell gelöst:

Bilde die Summe über  $i$  in den Grenzen  $a$  und  $b$  mit: 
$$s = \sum_{i=a}^b i$$

Die Lösung des Problems ist auf mehrere Wege möglich; da jedoch hier die Multiplikation nicht definiert ist, muss die Summe über eine Schleife erfolgen. Eine mögliche Lösung folgt:

An den Adressen 00100001 und 00100010 (dezimal 33 und 34) liegen die Grenzen  $a$  bzw.  $b$  an, wobei das Ergebnis  $s$  auf Adresse 00100011 (dezimal 35) geschrieben wird, die zuvor mit dem Wert 0 initialisiert sein muss.

Die Befehlsfolge sei dabei:

Sprungmarke / Variable	Assembler	Mnemonic	Adresse	Speicherinhalt	Kommentar
LOOP:	LD b	LDABS	000	10100000	Lade den Wert der oberen Grenze
		$b$	001	00100010	
	ADD s	ADDABS	002	00100001	Addiere den Wert zum Ergebnis
		$s$	003	00100011	
	ST s	STABS	004	10110000	Speichere Ergebnis
		$s$	005	00100011	
	LD b	LDABS	006	10100000	Lade den Wert der oberen Grenze
		$b$	007	00100010	
	SUB #1	SUBIMM	008	00110010	Subtrahiere den Wert 1
		1	009	00000001	
	ST b	STABS	010	10110000	Speichere die neue obere Grenze ab
		$b$	011	00100010	
	SUB a	SUBABS	012	01000010	Subtrahiere den Wert der unteren Grenze
		$a$	013	00100001	
	JZ END	JZ	014	11010000	Springe auf Ende falls Subtraktion gleich 0
		END	015	00010010	
	JMP LOOP	JMP	016	11000000	Springe auf den Schleifenanfang
		LOOP	017	00000000	
END:	0		018	00000000	
			...		
a			033		Untere Grenze
b			034		Obere Grenze
s			035		Ergebnis

**Tabelle 4: Programmbeispiel**

Belegt man den Speicher mit diesen Daten, so hätte die Datei „memin.log“ den folgenden Inhalt:

```
000 10100000
001 00100001
002 00100001
003 00100011
004 10110000
005 00100011
006 10100000
007 00100001
008 00110010
009 00000001
010 10110000
011 00100001
012 01000010
013 00100010
```

```
014 11010000
015 00010010
016 11000000
017 00000000
018 00000000
033 00001100
034 00000010
035 00000000
```

### 6.1 Aufgabe 5

Entwerfen Sie ein Maschinenprogramm nach dem obigen Beispiel, das die Funktion des im zweiten Versuch vorgestellten Programms NumBits („Anzahl der gesetzten Bits im Datenwort“) erfüllt.

5.1 Programmieren Sie zunächst mit Assemblerbefehlen und Sprungmarken bzw. Variablennamen.

5.2 Übersetzen Sie Ihr Assemblerprogramm nun in Maschinencode.