1 Ziele

Hier zunächst ein Überblick über die Ziele dieses Praktikums:

- Einführung in Aufbau und Funktionsweise der 4-Bit CPU
- Verständnis und Implementierung einiger erweiterter Architekturmerkmale
  - Implementierung einer Ein-/Ausgabeeinheit (IO-Unit) zur Kommunikation mit der Testumgebung
  - Erweiterung des Befehlssatzes um spezielle Ein-/Ausgabebefehle
  - Realisierung einiger Testprogramme unter Verwendung des erweiterten Befehlssatzes und der E/A-Einheit

2 Grundlagen

2.1 Einführung

In den bisherigen Praktika wurden die entworfenen Prozessorkonzepte als eigenständig und von der Umgebung unabhängig betrachtet. Die zu verarbeitenden Daten befanden sich bereits im Hauptspeicher, was eine Kommunikation mit der Umgebung nicht erforderlich machte. Ein Rechner ohne jegliche Kommunikationsmöglichkeit mit der Außenwelt ist jedoch praktisch komplett nutzlos. Aus diesem Grund muss ein Rechnersystem diverse Ein-/Ausgabeeinheiten beinhalten, welche einen definierten Übergang zwischen Prozessor und externen Geräten schaffen.

Im Rahmen dieses Praktikums soll eine solche Ein-/Ausgabeeinheit entworfen werden. Weiterhin soll die vorgegebene 4-Bit CPU um zwei spezielle Ein-/Ausgabebefehle zur Programmierung dieser E/A-Einheit erweitert werden. Dabei soll jeweils ein Ein- und ein Ausgabebefehl zur Verfügung stehen.

Die softwareseitige Programmierung der E/A-Einheit wird den zweiten Teil dieses Praktikums darstellen. Im Gegensatz zur 2-Bit CPU lassen sich mit der 4-Bit CPU aufgrund des vergrößerten Befehlssatzes und einem Adressraum von 256 Worten á 4 Bit wesentlich sinnvollere Programme realisieren.

Übrigens: Prozessoren mit einer Verarbeitungsbreite von 4 Bit werden oft in kleineren „Eingebetteten Systemen“ verwendet, wo die Anwendung von größeren Prozessoren aufgrund des höheren Preises oder zu hoher Leistungsaufnahme nahezu unmöglich ist.

2.2 Merkmale der 4-Bit Architektur

Zunächst soll anhand einiger Stichpunkte die momentan implementierte Funktionalität der im Praktikum verwendeten CPU vorgestellt werden.

- Die allgemeine Architektur ähnelt sehr stark der bereits behandelten 2-Bit CPU.
- Der Datenbus und die Verarbeitungsbreite der CPU betragen jeweils 4 Bit.
- Der Adressbus besitzt eine Breite von 8 Bit, was einen Adressraum von 256 Worten entspricht. Daraus resultiert, dass der Instruction-Pointer und das Adressregister ebenfalls eine Breite von jeweils 8 Bit besitzen müssen.

- Die größere Anzahl möglicher Befehle ermöglicht es, einige grundlegende ALU-Operationen wie z. B. Negation, bitweise Rechtsverschiebung sowie Befehle zur Flag[1]-Manipulation mit in den Befehlssatz aufzunehmen.
- Als einzige mögliche Art ein Speicherwort zu adressieren existiert wie bei der 2-Bit CPU die direkte Adressierung. Dabei wird die Adresse als Argument im Befehl direkt angegeben.
- Als weiterer entscheidender Zusatz wurde die Möglichkeit der verzweigten Programmausführung eingeführt. Dass heißt, Sprungbefehle werden nur unter bestimmten Umständen ausgeführt.

2.3 Befehlssatz des Prozessors

In der folgenden Tabelle ist der komplette Befehlssatz der vorgegebenen 4-Bit CPU zusammengefasst.

Hex-Code	Bin-Code	Mnemonik	Bedeutung
0	0 0 0 0	--	momentan unbenutzt
1	0 0 0 1	--	momentan unbenutzt
2	0 0 1 0	CCF	Löschen des Carry-Flag
3	0 0 1 1	SCF	Setzen des Carry-Flag
4	0 1 0 0	JPZ ADDR	Sprung zu Adresse ADDR, wenn Zero-Flag gesetzt ist
5	0 1 0 1	JPC ADDR	Sprung zu Adresse ADDR, wenn Carry-Flag gesetzt ist
6	0 1 1 0	--	momentan unbenutzt
7	0 1 1 1	--	momentan unbenutzt
8	1 0 0 0	LD (ADDR)	Registerstack mit Datum an Speicher-Adresse ADDR laden
9	1 0 0 1	--	momentan unbenutzt
A	1 0 1 0	STO (ADDR)	Datum aus Registerstack in Speicher-Adresse ADDR schreiben
B	1 0 1 1	--	momentan unbenutzt
C	1 1 0 0	ADC	A und B mit Carry addieren, Ergebnis nach A schreiben, CF und ZF updaten
D	1 1 0 1	NOT	Inhalt von Register A bitweise negieren, ZF updaten
E	1 1 1 0	--	momentan unbenutzt
F	1 1 1 1	RRC	Register A bitweise über Carry-Flag rechtsrotieren, CF und ZF updaten

2.4 Strukturmodell

In Abbildung 1 ist der vereinfachte strukturelle Aufbau der 4-Bit CPU dargestellt. Dabei sind Datenpfad mit sämtlichen Registern und der Steuerpfad innerhalb der CPU ersichtlich.

2.4.1 Komponenten des Datenpfades

Befehlsregister (BR): Das Befehlsregister speichert wie gehabt den momentan abzuarbeitenden Befehl.

Adressregister (AR): Das Adressregister beinhaltet im Fall einer Transportoperation (`LD, STO`) die Quell- bzw. Zieladresse des Datums. Bei Verzweigungsbefehlen (`JPC, JPZ`) wird die Adresse des Sprungzieles im Fall erfüllter Sprungbedingung gespeichert.

Instruction-Pointer Register (IP): Der Instruction-Pointer beinhaltet die momentan durch die CPU adressierte Speicherzelle zum Lesen des nächsten Befehls bzw. zum Lesen eines Arguments.

Registerstack: Die vom Anwender programmierbaren Register der 4-Bit CPU sind wie bei einer der Erweiterungsvarianten der 2-Bit CPU als Registerstack mit zwei Registern A und B realisiert. Die Verwendung eines Registerstack hat den Vorteil, dass Quell- und Zieladressen der Register nicht als Argument im Befehl angegeben werden müssen, sondern implizit vereinbart sind. Als Nachteil kann man die Tatsache anführen, dass sich der Zugriff insbesondere auf Register B bei weitem umständlich gestaltet. Die folgende Tabelle verdeutlicht die möglichen Operationen mit ihren Auswirkungen auf den Inhalt des Registerstack.

[1] Mit *Flags* werden Register spezieller Statusinformationen mit einer Breite von jeweils einem Bit bezeichnet. Im Abschnitt 2.4.1 wird dazu noch genauer eingegangen.

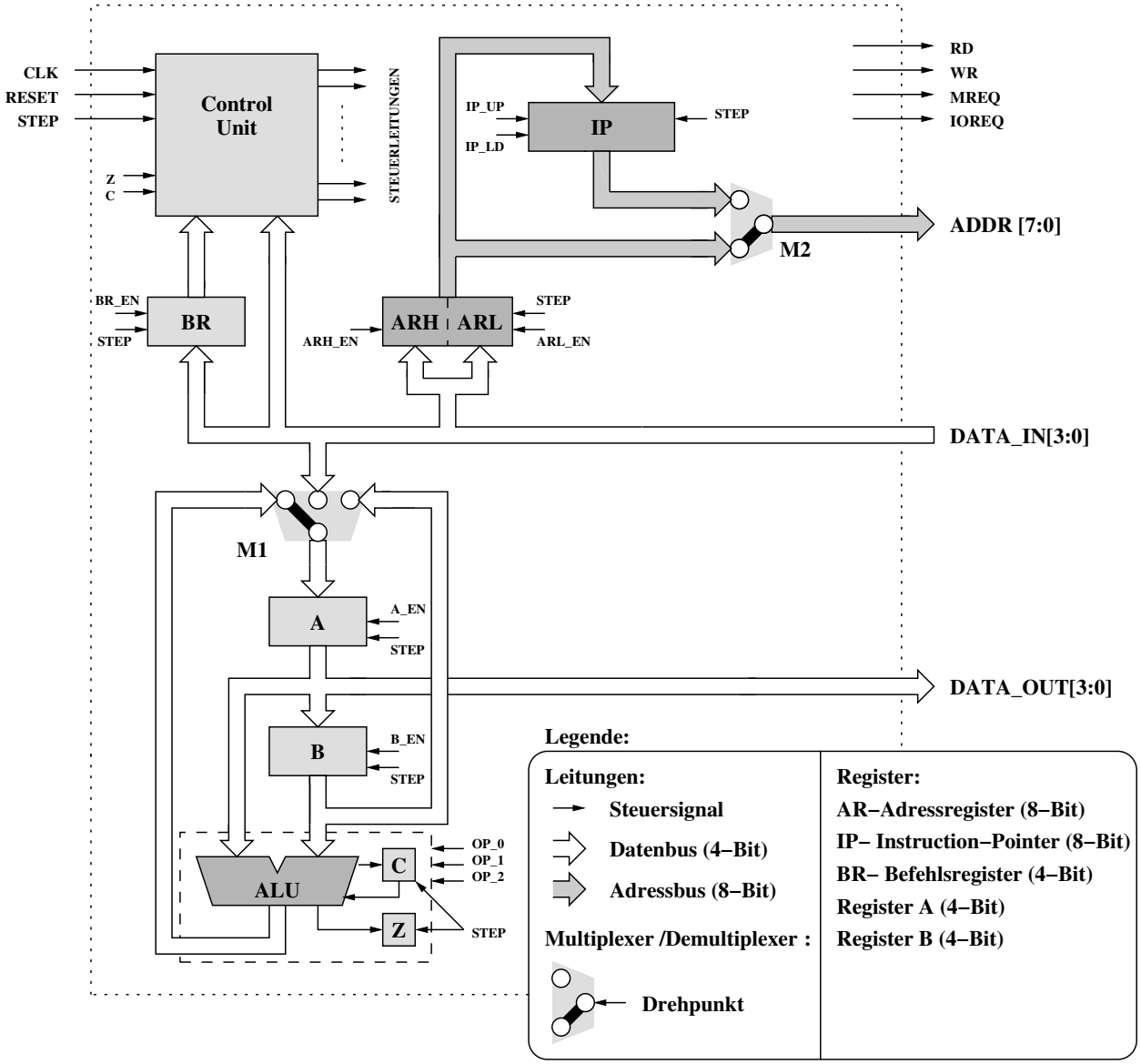

Abbildung 1: Datenpfad und vereinfachter struktureller Aufbau der 4-Bit CPU

Operation	Vorgänge im Registerstack	
einschreiben	x → A,	A → B
auslesen	A ← B,	B ← B
binäre Operation	A ← A OP B,	B ← B
unäre Operation	A ← OP A,	B ← B

ALU: Die „Arithmetic Logic Unit“ der 4-Bit CPU bringt eine Reihe von Operationen mit, die es ermöglichen, auch komplexere Aufgaben zu programmieren. So wurde z. B. neben der Addition und Negation die bitweise Rechtsverschiebung bzw. Rechtsrotation implementiert. Weiterhin werden zusätzlich Statusinformationen in den beiden Flags gespeichert. Dabei wird das *Zero*-Flag auf 1 gesetzt, wenn das Ergebnis der letzten ALU-Operation Null war. Das *Carry*-Flag speichert den Übertrag der letzten Additionsoperation. Bei der Rotationsoperation (`RRC`) wird es ebenfalls mit einbezogen (Rotation über Carry). Dabei wird das höchstwertige Bit (3) des Registers A mit dem Inhalt des Carry-Flags geladen. Nach der Operation enthält das Carry-Flag den Wert des niederwertigsten Bit (0) von A. Bei den verbleibenden Bits wird der Inhalt von Bit *n* durch den Inhalt von Bit *n+1* ersetzt.

Die folgende Tabelle enthält alle ALU-Operationen mit den beeinflussten Flags.

Operation	beeinflusste Flags	Carry-Flag	Zero-Flag	Bedeutung
ADC	C,Z	Übertrag	Ergebnis = 0	Addition mit Übertrag
NOT	Z	—	Ergebnis = 0	bitweise Negation
CCF	C	0	—	Carry-Flag löschen
RRC	C,Z	A(0)	Ergebnis = 0	Rechtsrotation über Carry-Flag
SCF	C	1	—	Carry-Flag setzen

2.5 Der Steuerpfad

In Abbildung 2 ist der Ablaufgraph des Steuerwerkes zu sehen. Man erkennt deutlich die Unterteilung in 4 Maschinenzyklen. Während bei der 2-Bit CPU lediglich ein Takt pro Maschinenzyklus benötigt wurde, werden hier 3 Takte benötigt. Auf den Grund dafür soll hier nicht näher eingegangen werden. Es ist jedoch durchaus möglich, die 4-Bit CPU so umzubauen, dass ebenfalls nur ein Takt pro Maschinenzyklus gebraucht wird.

Maschinenzyklus 1: In der ersten, der sogenannten Fetch-Phase wird der durch den Instruction-Pointer adressierte Opcode in das Befehlsregister geladen. Im Fall der ALU-Operationen wird die Berechnung des Ergebnisses am Ende dieses Zyklus durchgeführt.

Maschinenzyklus 2 und 3: Diese beiden Zyklen dienen im Allgemeinen zum Lesen der Argumente des Befehls. Da einigen Opcodes (z. B. LD, STO) 8-Bit breite Argumente (Hauptspeicheradressen) folgen, muss der Zugriff auf die beiden Adressanteile nacheinander erfolgen. Dabei wird stets zuerst der niederwertige Teil und anschließend der höherwertige Teil der Adresse gelesen ($\Rightarrow$ Little Endian).

Maschinenzyklus 4: In diesem Maschinenzyklus erfolgt bei allen Transportoperationen das Schreiben des Datums auf das Ziel.

2.6 Mikroprogrammsteuerwerk

2.6.1 Aufbau und Funktionsweise

Da sich eine der Praktikumsaufgaben mit der Mikroprogrammierung des Steuerwerkes beschäftigt, soll dessen Aufbau und Funktionsweise zunächst kurz vorgestellt werden.

Wie in Abbildung 3 dargestellt, handelt es sich dabei prinzipiell um ein *direkt gefädeltes* Mikroprogrammsteuerwerk, wie es bereits im letzten Praktikum zur 2-Bit CPU erläutert wurde. Der Mikroprogramm–ROM umfasst insgesamt 64 Worte mit einer Breite von jeweils 19 Bit. 6 Bit davon werden zum Kodieren der Folgeadresse benötigt (wegen Tiefe von 64), und die restlichen 13 Bit stellen den eigentlichen Steuervektor dar.

Durch die Verwendung des direkt gefädelten Mikroprogrammsteuerwerkes ergibt sich eine Platzersparnis gerade bei den Befehlen, die nur aus wenigen Mikrobefehlen bestehen.

Da die einzelnen Mikroprogramme jedes Befehls nun nicht mehr auf regelmäßigen Anfangsadressen liegen ist es notwendig, diese Startadressen in einer Tabelle separat zu speichern. Diese Tabelle (im Folgenden als *MC_Table* bezeichnet) liefert für den aktuellen Befehlscode die Startadresse seines zugehörigen Mikroprogrammes.

Zu Beginn jeder Befehlsabarbeitung muss wie gehabt der durch den Instruction-Pointer adressierte Befehlscode aus dem Hauptspeicher in das Befehlsregister gelesen werden. Das dafür zuständige Mikroprogramm befindet sich auf den beiden Adressen 0 und 1 im Mikroprogrammspeicher. Mit Ausführung des Befehls auf Adresse 0 wird zunächst der Inhalt des Instruction-Pointers an den Adressausgang der CPU gelegt. Damit wird der nächste zu lesende Befehlscode im Hauptspeicher adressiert. Im nächsten Schritt (Mikroprogrammadresse 1) werden die Vorbereitungen getroffen, um den Befehlscode in das Befehlsregister zu laden. Dazu werden die Steuersignale BR_EN und MREQ aktiviert. Zur gleichen Zeit wird durch Auslesen der Tabelle *MC_Table* die Startadresse des Mikroprogrammes für den aktuellen Befehl ermittelt. Dazu wird der am Dateneingang DATA_IN liegende Befehlscode verwendet.

Diese Vorgehensweise – also das direkte Auswerten von DATA_IN – spart bei der Befehlsausführung einen Mikrobefehl gegenüber der Verwendung des Befehlscodes aus dem Register BR. Denn der Inhalt des Befehlsregisters wird ja ebenfalls erst mit diesem Schritt aktualisiert. Somit steht dessen Inhalt erst bei der Abarbeitung des nächsten Mikrobefehls zur Verfügung.

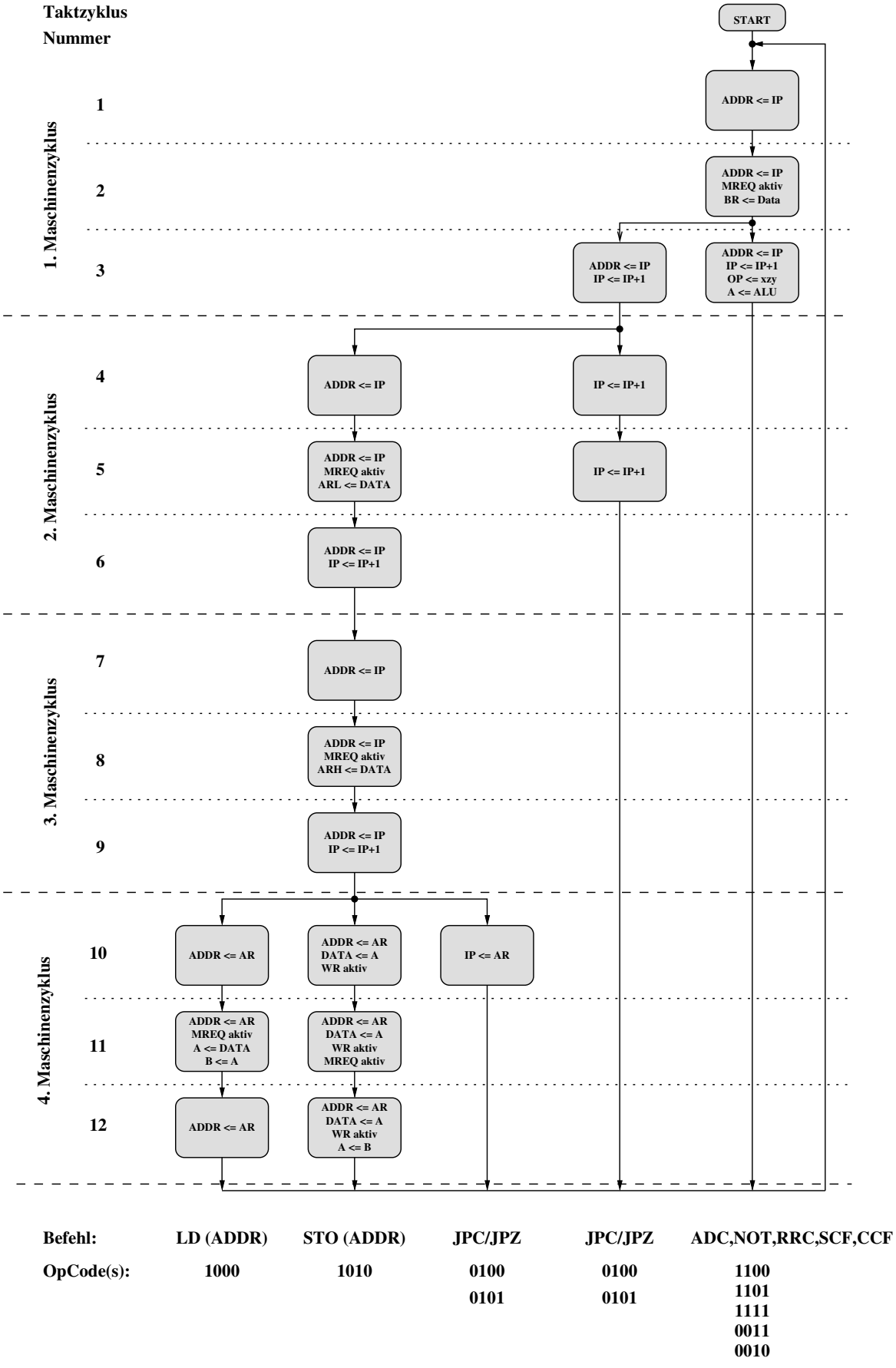

Abbildung 2: Steuerflussdiagramm der 4-Bit CPU

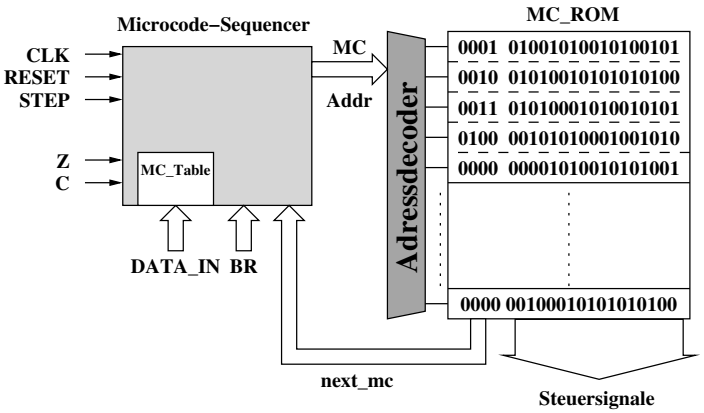

Abbildung 3: Struktur des Mikroprogrammsteuerwerkes der 4-Bit CPU

An der ermittelten Startadresse des Mikroprogrammes wird dann im folgenden Schritt fortgefahren. Durch die Speicherung der Folgeadressen in den Mikrobefehlen wird der Programmablauf durch den Mikrocode definiert.

Um nach Abarbeitung aller Mikrobefehle des Maschinenbefehls mit dem nächsten Maschinenbefehl fortzufahren ist es notwendig, als Folgeadresse des letzten Mikrobefehls stets die Adresse 0 anzugeben. Dies leitet wiederum die Fetch-Phase ein und der nächste Befehlscode kann gelesen werden.

2.6.2 Struktur des Mikroprogramm-ROM

Um das Mikroprogramm der CPU in dem Spartan-3E FPGA zu speichern wird ein Bereich auf dem Chip als ROM deklariert, der während der Konfiguration mit den vordefinierten Werten beschrieben wird. Wie schon erwähnt besitzt der Mikroprogramm-ROM eine Größe von 64 Worten á 19 Bits. In Listing 1 ist der Aufbau der Datei des Mikroprogrammes auszugsweise dargestellt. Das vollständige Mikroprogramm ist der Datei `microcode.vhd` zu entnehmen.

Listing 1: Auszug aus dem Mikroprogramm-ROM

```
1     B"0000010000000000001", --00-- % FETCH %
2     B"0000100101000000001", --01--
3     B"0000000000000010001", --02-- % CCF, SCF %
4     B"0000000000001010001", --03-- % ADC, NOT, RRC %
5     B"0001010000000010001", --04-- % JPC, JPZ %
6     B"0001100000000010001", --05--
7     B"0000000000000010001", --06--
8     B"0010000000000000001", --07-- % JPC, JPZ (carry, zero gesetzt) %
9     B"0010010100100000001", --08--
10    B"0010100000000010001", --09--
11    B"0010110000000000001", --0A--
12    B"0011000100010000001", --0B--
13    B"0011010000000010001", --0C--
14    B"0000000000000001001", --0D--
15    B"0011110000000010001", --0E-- % LD (M) %
16    B"0100000000000000001", --0F--
17    B"0100010100100000001", --10--
18    B"0100100000000010001", --11--
19    B"0100110000000000001", --12--
20    B"0101000100010000001", --13--
21    B"0101010000000010001", --14--
22    B"0101100000000000010", --15--
23    B"0101110100001100010", --16--
24    B"0000000000000000010", --17--
```

Folgende Tabelle enthält eine Auflistung aller im Steuervektor beinhalteten Signale, sowie ihre Position und Bedeutung.

Bitposition	Signalname	Bedeutung
18 . . . 13	`next_mc[5...0]`	Adresse des nächsten Mikrobefehls
12	`wr`	write enable
11	`mreq`	Zugriff auf Speicherbereich
10	`ioreq`	Zugriff auf IO-Bereich
9	`br_en`	Befehlsregister enable
8	`arl_en`	Adressregister(Bits 3 . . . 0) enable
7	`arh_en`	Adressregister(Bits 7 . . . 4) enable
6	`aa_en`	Register A enable
5	`bb_en`	Register B enable
4	`ip_cnt`	Instruction-Pointer + 1
3	`ip_ld`	Instruction-Pointer laden
2 . . . 1	`m1_s[1...0]`	Steuervektor Multiplexer 1
0	`m2_s`	Steuervektor Multiplexer 2

Diese Information ist auch in der Datei `microcode.vhd` zu finden (oben den Kommentar vertikal von oben nach unten lesen).

2.6.3 Struktur der MC_Table

Wie schon erwähnt enthält die Tabelle *MC_Table* die Startadressen der Mikroprogramme für die einzelnen Maschinenbefehle. Da im Fall der 4-Bit CPU bei Verwendung eines einzelnen 4-Bit Wortes für die Befehlskodierung maximal 16 Befehle möglich sind, besitzt die Tabelle 16 Einträge. Jeder dieser Einträge speichert ein 6-Bit Wort, welches die jeweilige Startadresse repräsentiert. Die Tabelle wird auf die gleiche Weise wie das Mikroprogramm in einem vorher deklarierten Speicherbereich ebenfalls in einem ROM untergebracht. Die Generierung des ROM-Inhaltes geschieht mit Hilfe der in Listing 2 teilweise dargestellten Datei `mctable.vhd`.

Listing 2: Auszug aus der MC_Table

```
1     B"000000", --0-- % - unused -      %
2     B"000000", --1-- % - unused -      %
3     B"000010", --2-- % 02: CCF         %
4     B"000010", --3-- % 02: SCF         %
5     B"000100", --4-- % 04: JPZ ADDR    %
6     B"000100", --5-- % 04: JPC ADDR    %
7     B"000000", --6-- % - unused -      %
8     B"000000", --7-- % - unused -      %
9     B"001110", --8-- % 0E: LD (ADDR)   %
10    B"000000", --9-- % - unused -      %
11    B"011000", --A-- % 18: STO (ADDR)  %
12    B"000000", --B-- % - unused -      %
13    B"000011", --C-- % 03: ADC         %
14    B"000011", --D-- % 03: NOT         %
15    B"000000", --E-- % - unused -      %
16    B"000011"  --F-- % 03: RRC         %
```

3 Testumgebung

In Abbildung 4 ist der strukturelle Aufbau der Testumgebung dargestellt. Der Taster *BTN1* bewirkt das Rücksetzen der CPU in den Anfangszustand. Dabei bleiben jedoch durch etwaige Schreibbefehle veränderte Speicherzellen im Block RAM unberührt. Für ein vollständiges Reset muss der Spartan-3E FPGA daher leider neu konfiguriert werden.

Der Taster *BTN0* bewirkt bei kurzem Drücken eine schrittweise Abarbeitung. Bei längerem Drücken wird ein kontinuierlicher Takt des `STEP`-Signals von ca. 3 Hz erzeugt. Bei Drücken des Tasters *BTN2* wird das mit den Schiebeschaltern `SW[3..0]` eingestellte Bitmuster in das Register `IO_0` der IO-Unit übernommen. Die Schiebeschalter `SW[7..4]` sind mit dem IO-Register `IO_1` verbunden (siehe auch Abschnitt 3.2).

Mit Taster *BTN3* kann die Ausgabe auf den Sieben-Segment-Anzeigen und auf den LEDs ausgewählt werden. In Normalzustand erfolgt die Darstellung des Adressregisters AR auf den Anzeigen *AN3* und *AN2*. Die Darstellung des Carry- bzw. Zero-Flags erfolgt zusätzlich auf den Dezimalpunkten der selben Anzeigen. Die Register A bzw. B sind auf den Anzeigen *AN1* und *AN0* dargestellt. Die zugehörigen Dezimalpunkte *DP1* bzw. *DP0* sind mit dem Signal `RD` bzw. `WR` verbunden. Die LEDs *LD7* bis *LD4* visualisieren das Signal `DATA`, welches mit dem Eingang `DATA_IN` der 4-Bit CPU verbunden ist. Auf den LEDs *LD3* bis *LD0* hingegen erfolgt die binäre Darstellung des Befehlsregister BR.
Wird der Taster *BTN3* gedrückt gehalten, dann ändern sich die vorherigen Ausgaben folgendermaßen: Auf den Sieben-Segment-Anzeigen *AN3* und *AN2* wird nun das Instruction-Pointer Register dargestellt und auf den Anzeigen *AN1* bzw. *AN0* die aktuelle Microcode-Adresse (`STATE`). Die Dezimalpunkte *DP3* bzw. *DP2* sind jetzt mit den Signalen `MEMRQ` und `IOREQ` verknüpft. Das Register `IO_3` wird nun binär auf den LEDs *LD7* bis *LD4* und `IO_2` auf den LEDs *LD3* bis *LD0* abgebildet.

Hinweis: Ein Entprellen der Taster *BTN[3..1]* und ist nicht unbedingt notwendig, da eine mehrfache Änderung dieser Signale keine schlimmen Auswirkungen auf das Verhalten der Schaltung haben.

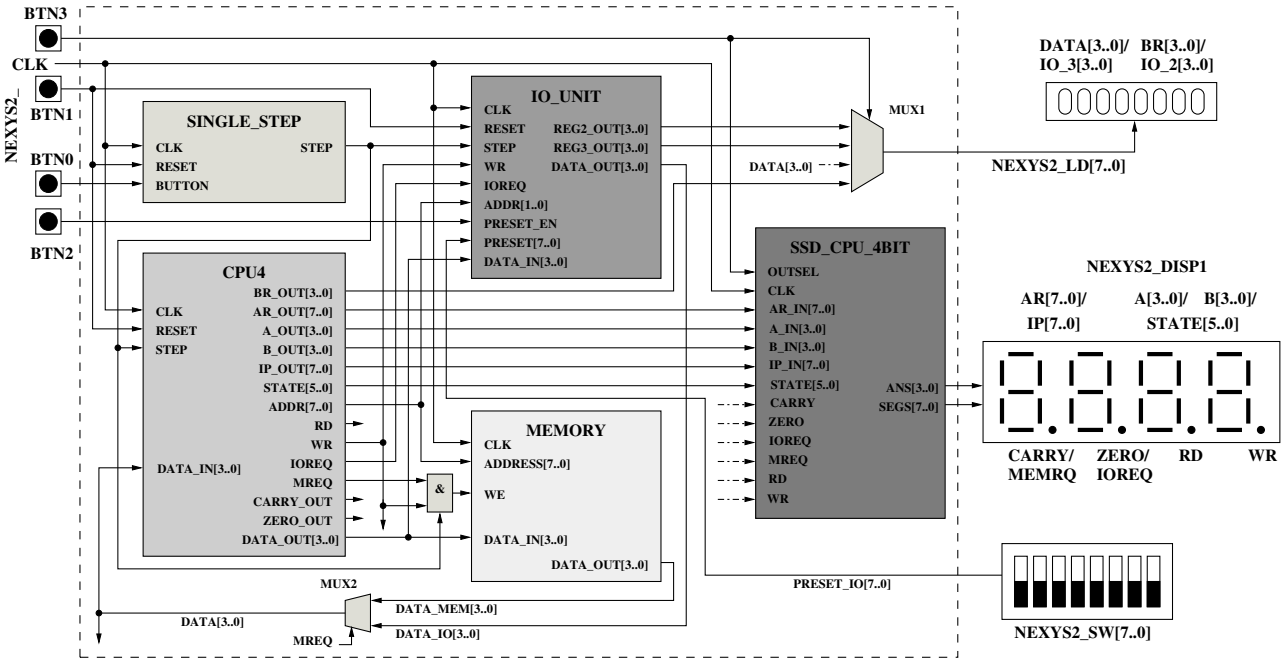

Abbildung 4: Testumgebung der 4-Bit CPU auf dem Digilent Nexys 2 Board

3.1 Single-Step

Bei dem verwendeten Single-Step-Modul handelt es sich um eine zu den vorherigen Praktika erweiterte Variante. Neben dem bereits bekannten Einzelschrittmodus wird nun zusätzlich bei längerem Drücken des Tasters (ca. 1 s) am Ausgang `STEP` ein kontinuierliches Signal mit einer Frequenz von ca. 3 Hz erzeugt. Längere Programmsequenzen können so durch Halten des Tasters einfacher abgearbeitet werden.

3.2 Ein-/Ausgabeeinheit (IO-Unit)

Zum Test der zu implementierenden Befehle soll eine Ein-/Ausgabeeinheit entworfen werden. Der prinzipielle Aufbau der Ein-/Ausgabeeinheit ist in Abbildung 5 dargestellt.

Die beiden Register `IO_2` und `IO_3` dienen der Speicherung der Ausgabedaten, die im entsprechenden Modus auf den LEDs *LD7* bis *LD0* ausgegeben werden können. Auf diese beiden Register kann nur schreibend zugegriffen werden.

Über die beiden Register `IO_0` und `IO_1` können Daten eingelesen werden. Das Register `IO_1` ist dabei, wie bereits erwähnt, fest mit den Schiebeschaltern *SW[7..4]* verdrahtet und spiegelt deren Stellung wider. Insofern muss es nicht einmal ein echtes Register sein, weil ein paar „Drähte“ auch ausreichen würden. Aber davon muss die eigentliche CPU nichts wissen...

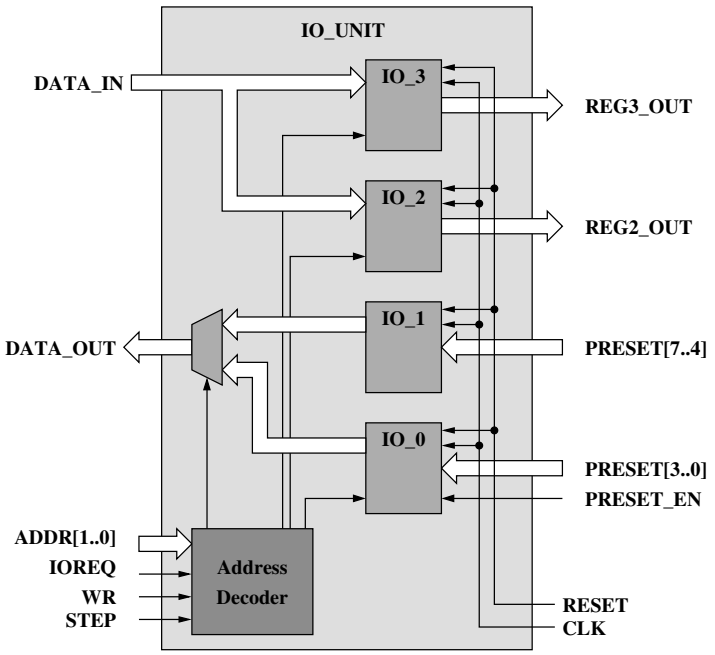

Abbildung 5: Struktur der E/A-Einheit (IO-Unit)

Das Register `IO_0` soll eine besondere Funktionalität besitzen. Durch Drücken des Tasters `BTN2` wird der an den Schiebeschaltern *SW[3..0]* eingestellte Vektor in das Register übernommen. Nach Auslesen des Registers durch die CPU wird dessen Inhalt gelöscht. In der folgenden Tabelle sind die einzelnen Register mit ihren Adressen innerhalb der IO-Unit dargestellt.

Portadresse	Datenrichtung	Funktion
00	Input	Register `IO_0`
01	Input	Register `IO_1`
02	Output	Register `IO_2`
03	Output	Register `IO_3`

3.3 Hauptspeicher

Der Hauptspeicher der CPU ist bei der 4-Bit CPU auf ähnliche Weise wie der Mikroprogramm-ROM, jedoch als Block RAM auf dem Spartan-3E FPGA implementiert. Er besitzt einen Umfang von 256 Worten á 4 Bit. Um nun Programme im Hauptspeicher unterzubringen wird wie bei der Programmierung des Mikroprogrammes verfahren. In der Datei `memory.vhd` wird der auszuführende Programmcode, wie im folgenden Beispiel (Listing 3) zu sehen, eingetragen. Im Anschluss daran muss das Design wie gehabt synthetisiert werden. Die Initialisierung des RAM-Inhaltes geschieht während der Konfiguration des FPGAs.

Listing 3: Auszug aus der Speicherkonfiguration der 4-Bit CPU

```
1     -- The following INIT_xx declarations specify the initial contents of the RAM
2     -- INIT_00: 255 to 0
3     --                          3               2               1               0
4     --           FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210
5     INIT_00 => X"0000000000000000000000000000000000000000000000000FFAFEAFD8FC8")
6     -- Progam:
7     -- 00: 8 C F -- LD $FC (IO_0)
8     -- 03: 8 D F -- LD $FD (IO_1)
9     -- 06: A E F -- STO $FE (IO_2)
10    -- 09: A F F -- STO $FF (IO_3)
```

4 Versuchsvorbereitung

Hinweis:
In Vorbereitung auf den Versuch ist es empfehlenswert, dass die folgenden Vorarbeiten von jeder Versuchsgruppe durchgeführt und die schriftlich ausgearbeiteten stichpunktartigen Antworten zu diesen Fragen und ggf. vorbereitete VHDL-Dateien zur praktischen Übung mitgebracht werden!

4.1 Kontrollfragen

1. Warum wird neben dem Befehlsregister auch `DATA_IN` in das Steuerwerk der 4-Bit CPU gebracht?
2. Wie aus der Befehlsbeschreibungstabelle ersichtlich ist, versteht die 4-Bit CPU lediglich bedingte Sprünge (springe wenn Zero und springe wenn Carry). Wie lässt sich ein unbedingter Sprung realisieren?
3. Überflüssige Leseoperationen auf den Speicher schaden prinzipiell nicht. Was ist jedoch zu beachten, wenn spezielle IO-Register im Spiel sind (besonders `IO_0` in unserer IO-Unit)?

4.2 Entwurf der IO-Unit

Die IO-Unit selbst soll nach der Spezifikation von Abbildung 5 entworfen werden. Dabei soll die in der Datei `io_unit.vhd` enthaltene Entity-Deklaration benutzt werden und die Datei entsprechend erweitert werden.

Wie bereits erwähnt, sollen die Register der IO-Unit folgende Funktionalität besitzen:

Registeradresse	Register	Datenrichtung	Funktion
`00`	`IO_0`	Input	Dateneingabe von *SW[3..0]* bei Drücken von *BTN2* Löschen bei lesendem Zugriff
`01`	`IO_1`	Input	Dateneingabe von *SW[7..4]*
`02`	`IO_2`	Output	Datenausgabe
`03`	`IO_3`	Output	Datenausgabe

Hinweis: Bei den IO-Registern `IO_2` und `IO_3`, die ausschließlich zur Ausgabe dienen, muss das Datum bei lesendem Zugriff nicht dem Inhalt des IO-Registers entsprechen. Dies könnte aber als Zusatzaufgabe angesehen werden.

4.3 Memory-Mapped IO-Unit

Eine häufig verwendete Methode des Zugriffs auf Register externer Bausteine ist das sogenannte *Memory-Mapping*. Dabei wird ein definierter Bereich des Hauptspeicheradressraumes als Adressbereich für IO-Register benutzt. Der an dieser Stelle befindliche Hauptspeicher ist in den häufigsten Fällen nicht mehr nutzbar, da er vom IO-Adressbereich überlagert wird. Ein lesender Zugriff auf die IO-Register ist in diesem Fall einfach über einen `LD`-Befehl, ein schreibender Zugriff über einen `STO`-Befehl möglich. Die Auswahl der jeweiligen Komponente (Speicher oder IO-Einheit) wird durch einen Adressdecoder geregelt, der jeweils nur IO-Register oder nur Hauptspeicher in Abhängigkeit der Adresse aktiviert.

Die entworfene IO-Unit soll nun so eingebunden werden, dass ein Zugriff auf die IO-Register über den Speicherbereich von `FCh` . . . `FFh` erfolgen kann. Die Zuordnung der einzelnen Adressen zu den IO-Registern ist der folgenden Tabelle zu entnehmen.

Adresse	Register
`FC`	`IO_0`
`FD`	`IO_1`
`FE`	`IO_2`
`FF`	`IO_3`

Also immer wenn die CPU z. B. auf Adresse `FCh` zugreift, dann soll nicht auf die Speicheradresse `FCh` zugegriffen werden, sondern auf Register `IO_0` (analog für `IO_1` bis `IO_3`).

Um nun das Memory-Mapping der IO-Unit zu realisieren, muss bei Speicherzugriffen der 4-Bit CPU je nach Adresse einmal der Speicher selbst, und einmal die IO-Unit aktiviert werden. Dies erfordert in der Datei `cpu4_top_level.vhd`

eine geeignete Manipulation des WE-Eingangs des Speichers sowie des IOREQ-Eingangs der IO-Unit. Der Schalter MUX2 muss ebenfalls so angesteuert werden, dass er je nach ausgewählter Adresse Daten aus dem Speicher oder Daten aus der IO-Unit zur CPU bringt (siehe auch Abbildung 4). Das von der CPU gelieferte Signal IOREQ hat hier keine Bedeutung (weil ungenutzt).

4.4 Testprogramm

Es soll ein kleines Testprogramm aufgestellt werden, welches die folgende Funktionalität besitzt und zunächst auf der Memory-Mapped Version der IO-Unit basiert. In einer Schleife soll folgendes gemacht werden:

1. Register IO_0 soll ausgelesen werden
2. - ist Bit 0 des gelesenen Wertes gleich 1, so soll der Wert aus Register IO_1 auf Register IO_2 kopiert werden
   - ist Bit 1 des gelesenen Wertes gleich 1, so soll der Wert aus Register IO_1 auf Register IO_3 kopiert werden
3. Es wird wieder von vorn begonnen.

Hinweis: Der Befehl RRC ist hier unabkömmlich!

4.5 Erweiterung der CPU um die beiden Befehle IN und OUT sowie IO-Mapped Verschaltung der IO-Unit

Eine weitere Variante des Zugriffs auf externe Register ist das sogenannte *IO-Mapping*. Im Gegensatz zu der Memory-Mapped Variante wird ein zusätzliches Steuersignal IOREQ eingeführt, das in Verbindung mit speziellen Ein- und Ausgabebefehlen den Zugriff auf externe Einheiten realisiert. Der somit entstandene IO-Adressbereich existiert sozusagen parallel zum Hauptspeicher. Ob auf Hauptspeicher oder IO-Register zugegriffen wird, hängt dabei einzig und allein von den beiden Steuersignalen MREQ und IOREQ ab, welche von der CPU erzeugt werden.

Um diese Unterscheidung zu gewährleisten, werden nun jedoch spezielle IO-Befehle benötigt, die den Zugriff auf den IO-Adressraum erlauben. Aus diesem Grund soll die oben beschriebene CPU um jeweils einen Eingabebefehl (IN) und einen Ausgabebefehl (OUT) erweitert werden. Dabei sollen die Befehle die folgende Syntax bzw. Semantik besitzen.

Hex-Code	Bin-Code	Mnemonik	Bedeutung
9	1 0 0 1	IN (ADDR)	Registerstack mit Datum von IO-Adresse ADDR laden
B	1 0 1 1	OUT (ADDR)	Datum aus Registerstack auf IO-Adresse ADDR schreiben

Hinweis: Um die Befehle IN und OUT nur minimal im Vergleich zu LD und STO zu ändern, kann das Adressargument der beiden Befehle der Einfachheit ebenfalls eine Breite von jeweils 8 Bit besitzen. Damit stände ein IO-Adressbereich von ebenfalls 256 Worten zur Verfügung. In unserem Fall wäre auch ein IO-Adressbereich von nur 16 Worten mehr als ausreichend, da wir eigentlich nur 4 Adressen benötigen würden.

Beim Entwurf der beiden Ein-/Ausgabebefehle könnte wie folgt vorgegangen werden:

1. Erweiterung des Steuerflussgraphen um die beiden Befehle IN und OUT.
2. Notieren aller Stellungen der Multiplexer und Aktivitäten der Enable-Eingänge der Register.
3. Ergänzen des Mikroprogrammes (microcode.vhd) der 4-Bit CPU, um die erweiterte Funktionalität zu erhalten.
4. Ergänzen der Mikroprogrammstartadressen-Tabelle (mctable.vhd) der CPU um die beiden zusätzlichen Befehle.

Beide Befehle funktionieren analog zu LD und STO, es wird lediglich das Signal IOREQ anstatt MREQ aktiviert.

Um die Sache zu komplettieren, muss die CPU wieder von der Memory-Mapped Version der IO-Unit auf die IO-Mapped Version zurückgebaut werden. Das heißt, das Ausgangssignal IOREQ der 4-Bit CPU muss mit dem Eingang IOREQ der IO-Unit mittels des gleichnamigen Signals verbunden werden, und der Speicher wird ebenfalls normal angesprochen – praktisch so wie in Abbildung 4 dargestellt.

Hinweis: Da die IO-Unit nur die beiden niederwertigsten Adressbits auswertet, wiederholen sich die vier IO-Register immer wieder (insgesamt 64 mal bei einer 8 Bit Adresse). Das soll uns nicht weiter stören.

5 Durchführung

1. Die hier weitestgehend besprochene IO-Unit soll zunächst als Memory-Mapped Variante in die 4-Bit CPU integriert werden, und mit dem kleinen Testprogramm (siehe Abschnitt 4.4) getestet werden.
   Das Testprogramm muss dabei wie gehabt per Hand assembliert, d. h. in Maschinencode überführt und in die Datei `memory.vhd` gebracht werden.

   Beim Test sollte darauf geachtet werden, ob das IO-Register `IO_0` auch richtig funktioniert. Also beim Auslesen sollte es auf `0000` gesetzt werden und erst durch Drücken von *BTN2* einen neuen Wert erhalten.

2. Nachdem die Funktionsweise der Memory-Mapped Version der IO-Unit erfolgreich getestet wurde, sollen nun die beiden IO-Befehle `IN` und `OUT` integriert werden. Weiterhin soll die IO-Unit so eingebunden werden, dass sie im IO-Adressraum der CPU liegt (Verbindungen wie in Abb. 4).

3. Zum Schluss soll die Funktionalität der beiden neuen Befehle getestet werden, indem das kleine Testprogramm minimal geändert wird. Das heißt, anstatt der Befehle `LD` und `STO` wird nun `IN` und `OUT` verwendet.

Hinweise, Berichtigungen und Kritik zu den Übungsunterlagen bitte an:

- René Oertel <rene.oertel@cs.tu-chemnitz.de>

Literatur und wichtige Links

[1] *The VHDL Cookbook, First Edition*
Peter J. Ashenden
http://www.vhdl.org/misc/VHDL-Cookbook.ps.tar.Z
http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf

[2] *Schaltungsdesign mit VHDL*
Gunther Lehmann, Bernhard Wunder, Manfred Selz
Eine ausführliche Einführung in VHDL und nebenbei ein gutes Referenzhandbuch. Ein „must-have“ ;-) Das Buch, bestehend aus mehreren `.pdf`-Dateien und Beispielen, ist kostenlos im Internet verfügbar:
http://www.itiv.kit.edu/653.php

17. Oktober 2011