1 Ziele

Hier zunächst ein Überblick über die Ziele dieses Praktikums:

- Die im vorherigen Praktikum entworfene 2-Bit CPU soll an einen vergrößerten Adressraum angepasst werden. Dabei soll die Änderung der internen Struktur der CPU und die damit verbundene Anpassung der Befehle nachvollzogen und verstanden werden.
- Das Steuerwerk der 2-Bit CPU soll zu einem mikroprogrammierbaren Steuerwerk umgebaut werden.
- Aufbau und Merkmale verschiedener Befehlssatz-Architekturen sollen besprochen und implementiert werden. Dabei soll es speziell um die beiden folgenden Varianten gehen:
  - Stack-Architektur
  - Akkumulator-Architektur

2 Grundlagen

2.1 Einführung

Anhand der im letzten Praktikum entworfenen 2-Bit CPU werden nun weitere wichtige Prinzipien und Konzepte besprochen. Dies betrifft unter anderem die Erweiterung des Adressraumes der CPU. Anstatt lediglich 4 Worte, soll die CPU nun 16 Worte á 2 Bit adressieren können. Diese Erweiterung und die damit verbundene Änderung der Befehlssyntax wird in dieser Übung erarbeitet.

Hier soll zunächst auf das Prinzip der *Mikroprogrammierung* von Steuerwerken eingegangen werden, was auch Gegenstand eines Praktikumsversuches mit eigener Implementierung sein soll. Weiterhin werden zwei Befehlssatz-Architekturen näher beleuchtet – die sog. Stack-Architektur und die Akkumulator-Architektur.

2.2 Mikroprogrammierung von Steuerwerken

Die bisher in der 2-Bit CPU verwendete Sequence-Counter Methode in Verbindung mit einem kombinatorischen Schaltwerk zur Generierung der Steuersignale ist nur für relativ kleine Architekturen praktikabel. Denn sobald die Komplexität der CPU und die damit verbundene Anzahl der Steuersignale steigt, wird das Steuerwerk immer schwerer beherrschbar. Dies trifft insbesondere für Architekturen mit vielen verschiedenen Befehlsarten zu. Um diesem Problem aus dem Weg zu gehen wird ein Prinzip angewandt, das unter der Bezeichnung der *Mikroprogrammierung* bekannt ist.

Dabei wird nach folgendem Grundsatz verfahren: Die für die Abarbeitung einer Befehlsphase notwendigen Steuersignale werden als Steuervektor zusammengefasst. Es existiert somit für jeden Maschinenbefehl eine Menge von Steuervektoren, die sequentiell abgearbeitet der Funktionalität eines Maschinenbefehls entsprechen. Da ein Steuervektor sozusagen angibt, wie sich sämtliche Datenpfadkomponenten in dem Moment zu verhalten haben (welche Schalter sind wie eingestellt, welche Enable-Signale sind gesetzt etc.), spricht man hier auch von einem sogenannten *Mikrobefehl*.

Die Semantik eines Maschinenbefehls definiert sich nun durch die Abarbeitung mehrerer solcher Mikrobefehle. Je nach Befehl werden unter Umständen verschieden viele Mikrobefehle benötigt. In diesem Sinne ist dies nichts weiter als das zeilenweise Durchgehen durch die Steuerwerkstabelle, welche wir bereits kennen.

Innerhalb der Steuerwerkskomponente werden diese Mikrobefehle in einem Speicher untergebracht, der auch als *Mikroprogrammspeicher* bezeichnet wird. Dieser Speicher kann „read only“ als sog. ROM oder auch vom Anwender veränderbar als RAM realisiert sein.

Abbildung 1 zeigt den prinzipiellen strukturellen Aufbau eines besonderen mikroprogrammierbaren Steuerwerks. Die Belegung der Mikrobefehle ist in der Abbildung willkürlich und hat nichts mit unserer 2-Bit CPU zu tun!

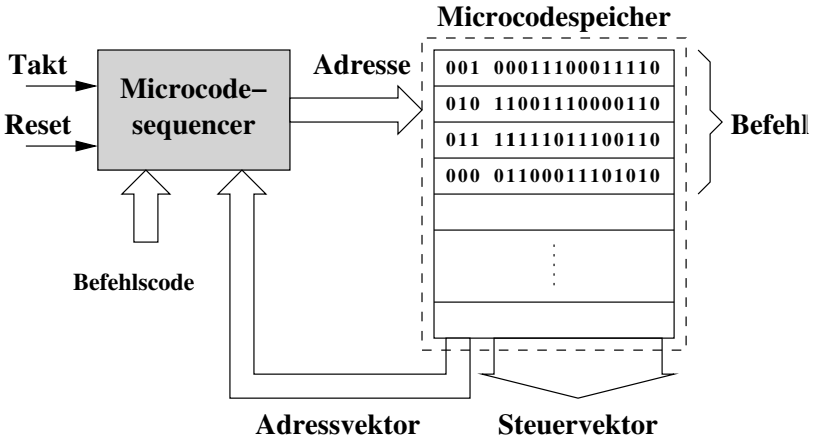

Abbildung 1: Prinzip eines Microcode Steuerwerkes

Das Besondere an dem hier gezeigten mikroprogrammierten Steuerwerk ist, dass der Mikrocode neben dem Steuervektor noch eine weitere Komponente enthält: Den Adressvektor. Darin wird die Adresse des Mikrobefehls angegeben, welcher in der nächsten Ausführungsphase ausgewählt werden soll. Man spricht hier auch vom *direkt gefädelten Mikrocode*.

Um nun Mikrobefehl für Mikrobefehl „auszuführen“ wird ein Automat benötigt, der auch als *Mikrocodesequencer* bezeichnet wird (sozusagen wiederum ein Mini-Steuerwerk). Dieser erzeugt aus den ihm zugeführten Eingangssignalen wie Befehlscode und Folgeadresse die Adresse des nächsten Mikrobefehls. Die Weiterschaltung der Mikrobefehle erfolgt dabei wie gehabt durch das allgemeine Taktsignal.

Die Abarbeitung eines Maschinenbefehls mit Hilfe des mikroprogrammierbaren Steuerwerks gestaltet sich wie folgt:
Nach dem Lesen des Befehlscodes, welches in der Regel mit dem gleichen Mikrocode geschieht, wird in Abhängigkeit vom Inhalt des Befehlsregisters zu der Mikrocodesequenz für eben diesen aktuellen Befehl verzweigt. Die einzelnen Mikrobefehle des Maschinenbefehls werden nun schrittweise abgearbeitet. Wie in Abbildung 1 dargestellt, ist dabei die Folgeadresse in der Regel direkt im Mikrocode enthalten. Nur in bestimmten Fällen muss die Adresse ggf. gesondert ermittelt werden. So z. B. nach dem Lesen des Befehls wie oben beschrieben. Nach der Abarbeitung einer Mikrocodesequenz für einen Maschinenbefehl wird wiederum zu einem Mikrocodestück verzweigt, welches für das Laden eines neuen Befehles verantwortlich ist. Damit beginnt die Befehlsabarbeitungsschleife des Prozessors von vorn.

Dieses allgemeine Verfahren bildet die Grundlage vieler in realen CPUs verwendeter mikroprogrammierbarer Steuerwerke. Die im nächsten Praktikum behandelte 4-Bit CPU besitzt ebenfalls eine auf diese Art und Weise konstruierte Steuerwerkskomponente. Für die 2-Bit CPU wollen wir zunächst ein einfacheres Mikroprogrammsteuerwerk verwenden (siehe Versuchsvorbereitung).

2.3 Befehlssatz-Architekturen

Es bestehen verschiedene Organisationsmöglichkeiten für die interne Registerstruktur einer CPU. Neben der Variante des sogenannten *Registerfiles*, bei dem mehrere meist gleichberechtigte CPU Register in Form einer Art zusammenhängenden Speicherbereiches realisiert sind, gibt es noch weitere bekannte Architekturen. Dazu zählen u. a. die Stack- und die Akkumulator-Architektur. Ihre Anwendung bringt eine Reihe von Veränderungen für den strukturellen Aufbau der CPU und den Aufbau des Befehlssatzes mit sich. Im Folgenden soll die Stack- und die Akkumulator-Architektur in Aufbau und Funktion vorgestellt werden und später selbstständig implementiert werden.

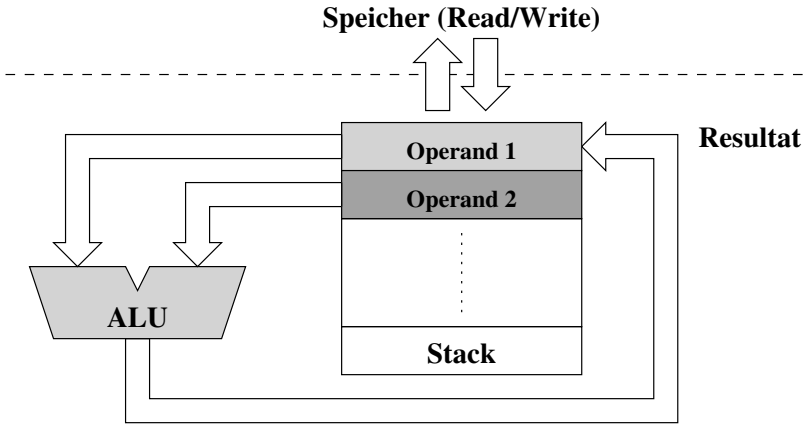

Abbildung 2: Prinzip einer Stack-Architektur

2.3.1 Stack-Architektur

Die einfachste mögliche Befehlssatz-Architektur ist die sog. *Stack-Architektur*, welche auch als *Nulladressmaschine* bezeichnet werden kann. Der Name Nulladressmaschine kommt daher, da bei einer arithmetischen Operation der CPU keine Registeradresse angegeben werden muss.[1] Die Operanden stehen sozusagen von vorn herein fest. In diesem Kontext können wir die „normale“ 2-Bit CPU ebenfalls als eine *Nulladressmaschine* bezeichnen, da bei dem einzigen arithmetischen Befehl (Add) die Quelloperanden (`AA` und `BB`) sowie der Zieloperand (`AA`) von vorn herein feststehen.

Bei der Stack-Architektur werden Quell- und Ergebnisoperanden in einem Stack gespeichert. Als Stack bezeichnet man eine Registerstruktur von linear miteinander verketteten Speicherelementen, welche die folgende Funktionalität realisiert: Das Einschreiben eines Datums in den Stack und das Lesen eines Datums geschieht stets an der obersten Stelle des Stack. Im Fall des Einschreibens (Lesen eines Wortes aus dem Speicher und Schreiben auf den Stack) rutschen alle bisher im Stack enthaltenen Daten eine Position tiefer. Das Datum an der tiefsten Stelle wird verworfen. Beim Lesen eines Datums vom Stack (bzw. dem Schreiben eines Datums vom Stack in den Speicher) ist der Vorgang analog. Das an der obersten Position im Stack befindliche Datum wird in den Speicher geschrieben und verworfen. Alle tiefer im Stack liegenden Daten rutschen eine Position nach oben. Das an der untersten Stelle im Stack liegende Datum bleibt dabei unverändert (wird praktisch dupliziert). Das Schreiben in den Stack wird auch als **Push** und das Lesen als **Pop** bezeichnet.

Im einfachsten Fall könnten sich für ALU-Operationen beide Quelloperanden wie in Abbildung 2 dargestellt auf den beiden obersten Plätzen im Stack liegen und von dort direkt entnommen werden (also ohne eine Pop-Operation). Gleichfalls könnte der oberste Platz des Stacks direkt mit dem Ergebnis überschrieben werden (also ohne eine Push-Operation). Natürlich sind auch andere Semantiken denkbar. Z. B. zweimal Pop und einmal Push (beide Operanden werden zerstört), oder Operanden direkt entnehmen und einmal Push (beide Operanden bleiben erhalten), ...

Zusätzlich zu den arithmetischen Operationen müssen noch Befehle vorhanden sein, die es ermöglichen den Registerstack mit Werten aus dem Speicher zu laden (`READ`) und die Daten aus dem Registerstack zurück in den Arbeitsspeicher zu schreiben (`WRITE`). Im Folgenden soll an einem Programmbeispiel gezeigt werden, wie die Addition zweier Werte mit Hilfe der Stackarchitektur realisiert wird.

Achtung: Hier wird bereits vom vergrößerten Adressraum aus Versuchsvorbereitung 3.2 ausgegangen! D. h. die Argumente sind nicht 2, sondern 4 Bit breit. Dabei ist außerdem zu beachten, dass bei 4 Bit breiten Argumenten zuerst die beiden niederwertigen Bits im Speicher stehen, und danach die beiden höherwertigen Bits (→ Little-Endian).

Weiterhin ist der Übersichtlichkeit halber die Kodierung der Befehle horizontal angegeben. Also die Zeile

```
0000 :  01 01 11
```

bedeutet, dass an Adresse `0000` das Datum `01` steht, an Adresse `0001` ebenfalls ein `01`, und an Adresse `0010` das Datum `11` steht.

[1] Im Fall von Read bzw. Write-Befehlen muss weiterhin eine Adresse zum Lokalisieren des Datums im Speicher angegeben werden.

```
Adresse   Inhalt      Mnemonik
 0000 :  01 01 11    READ  1101       # Lade Datum von Adresse 1101 in den Registerstack
 0011 :  01 10 11    READ  1110       # Lade Datum von Adresse 1110 in den Registerstack
 0110 :  11          ADD
                                      # Das Ergebnis der Addition befindet sich nun auf dem
                                      # obersten Platz im Stack
 0111 :  10 11 11    WRITE 1111       # Schreibe den Inhalt der obersten Stackposition
 ...                                  # nach Adresse 1111
 1101 :  01                           # Operand 1 im Hauptspeicher
 1110 :  11                           # Operand 2 im Hauptspeicher
 1111 :  00                           # Platzhalter für Ergebnis der Addition
```

Eine der bekanntesten Anwendungen dieser Stack-Architektur besteht in der Berechnung von Ausdrücken, die in der „Umgekehrten Polnischen Notation“ bzw. Postfixnotation verfasst wurden. Diese Notation hat die Eigenschaft, dass die einzelnen Ausdrücke zur korrekten Berechnung nicht geklammert werden müssen.

2.3.2 Akkumulator-Architektur

Eine weitere Befehlssatz-Architektur ist die sogenannte *Akkumulator-Architektur*. Sie wird auch als *Einadressmaschine* bezeichnet. Bei der Einadressmaschine existiert im Gegensatz zur Stack-Architektur nur der Akkumulator als einziges Universalregister. Damit muss bei einer arithmetischen Operationen der CPU einer der beiden Operanden stets dem Akkumulator entnommen werden. Der zweite Operand befindet sich im Hauptspeicher und wird durch Angabe der Operandenadresse im Befehl adressiert. Das Ergebnis der arithmetischen Verknüpfung wird im Akkumulator gespeichert. Weiterhin

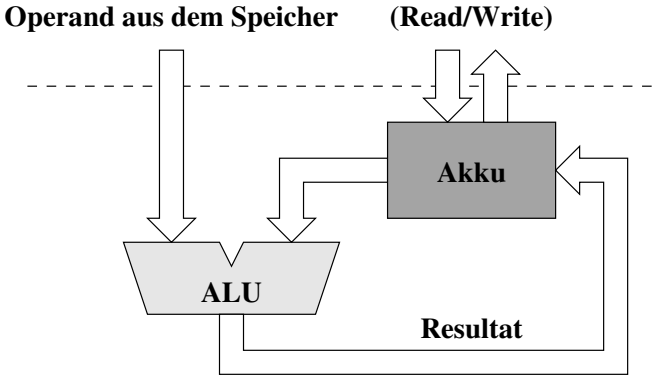

Abbildung 3: Prinzip einer Akkumulator-Architektur

gibt es in der Regel nach wie vor die Möglichkeit, Daten mittels `READ`- und `WRITE`-Befehlen zwischen Hauptspeicher und Akkumulator zu transferieren.

Ein kurzes Beispielprogramm zeigt die Addition zweier Binärzahlen bei Verwendung der Akkumulator-Architektur (wieder bereits mit 4 Bit Argumenten):

```
Adresse   Inhalt      Mnemonik
 0000 :  01 01 11    READ  1101       # Lade Datum von Adresse 1101 in den Akkumulator
 0011 :  11 10 11    ADD   1110       # Addiere das Datum von Adresse 1110 zum Akkumulator
                                      # Das Ergebnis der Addition befindet sich nun im Akku
 0110 :  10 11 11    WRITE 1111       # Schreibe den Inhalt des Akkumulators
 ...                                  # nach Adresse 1111
 ...
 1101 :  01                           # Operand 1 im Hauptspeicher
 1110 :  11                           # Operand 2 im Hauptspeicher
 1111 :  00                           # Platzhalter für Ergebnis der Addition
```

3 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!

3.1 Kontrollfragen

- Wie schon erwähnt kann der Mikrocodespeicher entweder als ROM oder als RAM implementiert sein. Worin könnte eine mögliche Anwendung der Variante mit veränderlichem Mikrocode (RAM) bestehen.
- Wie gesehen macht die Form der stackbasierten Registerorganisation eine Angabe der Registeradresse im Befehl überflüssig. Ansonsten müsste ja bei mehreren Registern immer angegeben werden, welche Register betroffen sind. Diese „Sparmaßnahme“ führt zu einer Reduktion der Befehlsbreite (Anzahl der benötigten Bits) und sollte ebenfalls unweigerlich zu einer höheren Codedichte[2] führen. Warum ist diese Aussage nur teilweise richtig?

3.2 Vergrößerung des Adressraumes der 2-Bit CPU auf 4 Bit

Bisher hat der eingeschränkte Adressraum von 4 Worten á 2 Bit kaum sinnvolle Programme ermöglicht. Aus diesem Grund besteht nun die Aufgabe darin, die CPU so abzuändern, sodass ein Speicherbereich mit einer Größe von 16 Worten á 2 Bit adressiert werden kann. Dabei gilt es sich zunächst zu überlegen, welche Teile der CPU von Änderungen betroffen sind. Die Datenbusbreite von 2 Bit soll jedoch unverändert bleiben. Die bereits bestehenden Befehle sind aber wie folgt zu ändern:

- `READ XX YY`, Der Befehl soll ein 4 Bit Argument besitzen um den vollen Speicherbereich adressieren zu können. Dabei stellt `XX` den niederwertigen und `YY` den höherwertigen Adressanteil dar.
- `WRITE XX YY`, besitzt ebenfalls ein 4 Bit Argument. Die Aufteilung in niederwertigen und höherwertigen Anteil ist äquivalent zum Read-Befehl.
- `JUMP XX YY`, Der Befehl soll ebenfalls ein 4 Bit Argument besitzen (wie gehabt als relative Sprungweite)
- Der `ADD`-Befehl bleibt von den Änderungen unberührt, da nicht auf den Speicher zugegriffen wird.

3.3 Ausstattung der 2-Bit CPU mit einem mikroprogrammierbarem Steuerwerk

Als Basis für das mikroprogrammierte Steuerwerk soll die Version der 2-Bit CPU mit dem erweiterten Addressraum dienen!

Die Aufgabe besteht darin, das in der 2-Bit CPU integrierte Steuerwerk auf der Basis eines Sequence-Counters in ein mikroprogrammierbares Steuerwerk zu verwandeln. Dazu wurde in der Einführung bereits das Prinzip einer solchen Steuerwerkskomponente vorgestellt.

Wir wollen uns hier auf eine etwas einfachere Realisierungsmöglichkeit konzentrieren, welche bei kleinen Steuerwerken zur Anwendung kommen kann. Anstatt die Adresse des nächsten Mikrobefehls im Mikrocode mitzuführen, kann diese direkt generiert werden.

Und zwar verwendet man den Befehlsphasenzähler um den niederwertigen Teil der Mikrocodeadresse zu erzeugen, und den Inhalt des Befehlsregisters für den höherwertigen Teil der Adresse. Den Befehlsphasenzähler können wir hier als 2 Bit Zähler auslegen, um zwischen den vier Befehlsphasen zu unterscheiden. Aus den zwei Bits des Befehlsphasenzählers und den zwei Bits des Befehlsregisters entsteht somit ein Adressbereich von 16 Worten. Dabei kann jeder der 4 Maschinenbefehle aus 4 Mikrobefehlen bestehen. Sinnvollerweise sollten die Mikrobefehle wie folgt im Mikrocodespeicher abgelegt sein:

[2] Bei gleicher Semantik wird weniger Programmspeicher verbraucht.

Opcode (bin)	Adressbereich (hex)	Bedeutung
00	0 . . . 3	Mikrocode des Befehls `JUMP`
01	4 . . . 7	Mikrocode des Befehls `READ`
10	8 . . . B	Mikrocode des Befehls `WRITE`
11	C . . . F	Mikrocode des Befehls `ADD`

Wegen der Verwendung eines Zählers zur Adressgenerierung müssen die einzelnen Codes für einen Befehl aufeinanderfolgend und auf festgelegten Adressen im Microcodespeicher abgelegt sein. Eine mehrfache Nutzung gleicher Mikrocodesequenzen für unterschiedliche Maschinenbefehle ist somit nicht möglich.

Hinweise:

- Da der `ADD`-Befehl weiterhin weniger als 4 Takte (Maschinenzyklen) benötigt, werden nicht alle dafür verfügbaren Mikrobefehle benötigt. Das ist aber nicht weiter schlimm.
- Zum Rücksetzen des Befehlsphasenzählers im Falle von `ADD` wird wie gehabt ein Steuersignal aus dem Mikrocode benötigt.
- Der erste Mikrobefehl eines jeden Maschinenbefehls muss jeweils identisch sein. Der Mikrobefehl, welcher zum Laden des nächsten Maschinenbefehles verwendet wird, hängt immer vom letzten Befehl ab. Also bei einer Folge von

  ```
  READ ...
  ADD
  ```

  wird der erste (nullte) Mikrobefehl des `READ`-Befehls verwendet, um den `ADD`-Befehl zu laden. Das kommt daher, da das Befehlsregister mit dem Ende eines Befehles den alten Wert beibehält.

3.4 Veränderung der CPU zu einer Stack-Architektur

Die „normale“ bzw. bereits auf 4 Bit Adressraum erweiterte 2-Bit CPU soll in eine Stack-Architektur umgebaut werden. Der Einfachheit halber soll der Registerstack eine Tiefe von 2 besitzen.

Die Befehlssemantik der neuen CPU soll wie folgt definiert sein:

Mnemonik	Argument	Befehlscode	Bedeutung
`READ`	direkte Adr.	`01 XX YY`	Lesen eines Datums von Speicheradresse YYXX in den Stack.
`WRITE`	direkte Adr.	`10 XX YY`	Scheiben eines Datums aus dem Stack nach Speicheradresse YYXX.
`ADD`	—	`11`	Addieren der beiden im Stack befindlichen Operanden und Rückschreiben des Ergebnisses auf die oberste Stackposition.
`JUMP`	relative Adr.	`00 XX YY`	Relativer Sprung, d. h. YYXX wird zum aktuellen Befehlszähler hinzuaddiert

Dazu sollten zumindest folgende Dinge vorbereitet werden:

- ein modifiziertes Blockschaltbild der 2-Bit CPU bzw. des betroffenen Teils
- eine modifizierte Steuerwerkstabelle
- nach Möglichkeit der VHDL-Code

Hinweise:

- Der Stack sollte sich aus den Registern `AA` und `BB` zusammensetzen, welche in geeigneter Weise verschaltet sind. `BB` wird dabei nun zu einem „richtigen“ Register, und kann auch andere Werte als `01` annehmen.
- Die Semantik des `ADD`-Befehls bzgl. des Umganges mit dem Registerstack soll so sein, wie in Abbildung 2 (Seite 3) dargestellt. Also die beiden obersten (und einzigsten) Register werden **direkt** an die ALU geleitet, und in das oberste Register wird das Ergebnis geschrieben.
  Die Implementation und der Test anderer Organisationen kann als Zusatzaufgabe angesehen werden.

3.5 Veränderungen der CPU zu einer Akkumulator-Architektur

Die „normale“ bzw. bereits auf 4 Bit Adressraum erweiterte 2-Bit CPU soll in eine Akkumulator-Architektur umgebaut werden. Dabei soll die in der folgenden Tabelle aufgeführte Befehlssemantik zur Verfügung stehen.

Mnemonik	Argument	Befehlscode	Bedeutung
`READ`	direkte Adr.	`01 XX YY`	Lesen eines Datums von Speicheradresse `YYXX` in den Akkumulator.
`WRITE`	direkte Adr.	`10 XX YY`	Scheiben eines Datums aus dem Akkumulator nach Speicheradresse `YYXX`.
`ADD`	direkte Adr.	`11 XX YY`	Addieren des Datums von Speicheradresse `YYXX` zum Akkumulator und Rückschreiben des Ergebnisses in den Akkumulator.
`JUMP`	relative Adr.	`00 XX YY`	Relativer Sprung, d. h. `YYXX` wird zum aktuellen Befehlszähler hinzuaddiert

Dazu sollten zumindest folgende Dinge vorbereitet werden:

- ein modifiziertes Blockschaltbild der 2-Bit CPU bzw. des betroffenen Teils
- eine modifizierte Steuerwerkstabelle
- nach Möglichkeit der VHDL-Code

Hinweise:

- Das `BB`-Register wird nicht unbedingt benötigt. Es ist eine Architektur denkbar, bei der `BB` überhaupt nicht mehr existiert, sowie eine Architektur bei der `BB` als reguläres (ladbares) Register fungiert.

4 Durchführung

Die entworfenen Varianten der CPU sollen synthetisiert und funktionellen Tests unterzogen werden.

Hinweise:
Aufgrund des erweiterten Adressbereiches der CPU waren einige Änderungen bei der Darstellung der internen Register notwendig. Das Adressregister `AR` wird nun auf der Sieben-Segment-Anzeige *AN1* hexadezimal angezeigt. Die aktuelle Befehlsphase wird binär durch die beiden oberen, senkrechten Segmente von *AN3* dargestellt (`00` entspricht Phase F1, `01` Phase F2 usw.). Das Befehlsregister `BR` ist nun auf die oberen, senkrechten Segmente von *AN2* abgebildet. Die restlichen Anzeigen blieben unverändert.

Weiterhin wird der Inhalt des Speichers nicht mehr mit den Schiebeschaltern eingestellt, sondern muss vor der Synthese fest im `MEMORY`-Modul eingetragen werden. Außerdem wird weiterhin auf den LEDs *LD0-7* der Inhalt der untersten 4 Speicheradressen angezeigt. Abbildung 4 fasst diese Änderungen noch mal zusammen.

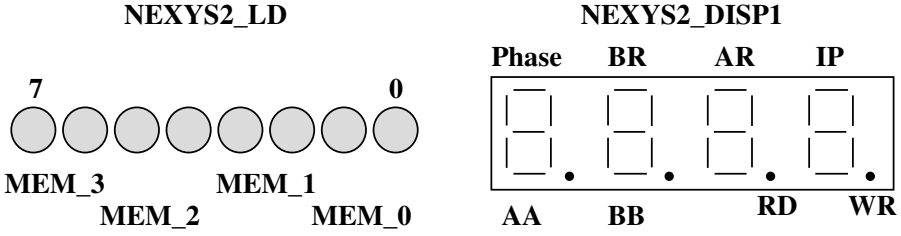

Abbildung 4: Veränderte LED-Belegung

1. Zunächst soll die CPU, deren VHDL-Quelltext den Praktikumsunterlagen bei liegt, mit erweitertem Adressbereich durch ein selbstgewähltes Beispielprogramm getestet werden. Dieses muss aufgrund des größeren Speicherbereiches fest in den Speicher der CPU eingetragen werden. Dazu sind entsprechende Änderungen im Modul `MEMORY` notwendig. Der Inhalt des Speichers wird mit einem Reset mit dem dort eingestellten Programm initialisiert.

2. Weiterhin soll die um das mikroprogrammierbare Steuerwerk erweiterte CPU getestet werden. Für den Funktionstest sollte das Testprogramm aus der vorhergehenden Aufgabe verwendet werden. Hinsichtlich der Funktion der CPU und der Ergebnisse des Testprogramms sollten dabei natürlich keine Unterschiede zu verzeichnen sein.

3. Es soll nun die CPU als Stack-Architektur getestet werden. Dabei ist die 2-Bit CPU mit einem geeigneten Testprogramm zu versehen, welches eine einfache Überprüfung der Funktionalität ermöglicht. Dazu kann das in der Einführung vorgestellte Programmfragment beliebig erweitert werden.

4. Abschließend muss noch die Version mit Akkumulator-Architektur einem Funktionstest unterzogen werden. Dazu ist ebenfalls ein geeignetes Testprogramm zu entwerfen und in das Design einzubinden.

5. ***Zusatzaufgabe:*** Diverse Variationen der Stackarchitektur (siehe Text) können implementiert und getestet werden.

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