
	<p style="text-align: center;">Rechnerorganisation Praktikum Prof. Dr.-Ing. W. Rehm</p> <hr/> <p style="text-align: center;">Praktikum 4 Bearbeiter: Sven Lasch Letzte Änderung: René Oertel, 25.11.2011</p>	
---	--	---

1 Ziele

Hier zunächst ein Überblick über die Ziele dieses Praktikums:

- Entwurf eines trivialen von Neumann-Rechners in Form einer 2-Bit CPU
- Verständnis des Steuerwerks und des funktionellen Ablaufs in der CPU
- Selbstständige Implementierung sämtlicher Grundbausteine des Rechners ausgehend von einem Strukturmodell

2 Grundlagen

2.1 Einführung

Im Mittelpunkt dieses Praktikums steht der Entwurf einer sehr einfachen 2-Bit Architektur. Denn gerade die Überschaubarkeit dieses vereinfachten Rechners erlaubt es, die wesentlichen Funktionseinheiten vollständig zu durchdringen und die Phasen des Entwurfs einer solchen Architektur zu verstehen. Im Laufe der schon absolvierten Praktika sind bereits einzelne Grundbausteine eines Digitalrechners und ihre Realisierung behandelt worden. Darauf aufbauend soll nun ein größeres Projekt – eine 2-Bit CPU – angegangen werden.

2.2 Konzepte der 2-Bit Architektur

Um einen ersten Eindruck zu erhalten, soll die grundlegende Architektur zunächst etwas näher vorgestellt werden. Es handelt sich hierbei um einen 2-Bit Rechner mit folgender beschränkter Funktionalität:

- Datenbus und Adressbus haben eine Breite von jeweils 2 Bit (\Rightarrow 4 Speicherstellen á 2 Bit).
- Es handelt sich um eine sogenannte *Load-Store*-Architektur, d. h. Speicherzugriffe erfolgen ausschließlich von einem Register in den Speicher (schreiben) oder vom Speicher in ein Register (lesen).
- Es wird ausschließlich direkte Adressierung angewandt, d. h. die Adresse, von der gelesen bzw. auf die geschrieben werden soll, wird im Befehl als Argument mit angegeben.
- Es gibt ein einziges allgemeines Rechenregister – den Akkumulator AA. Ein weiteres Register BB enthält permanent den Wert 1. Weitere in der CPU enthaltene Register sind nicht direkt (d. h. mittels eines Programmes) zugreifbar und dienen lediglich der Funktionalität der CPU.
- Es existiert ein relativer Sprungbefehl. Als Argument wird der relative Abstand von der Zieladresse im Befehl mit übergeben.
- Die ALU beherrscht als einzigen Befehl die Addition zweier Binärzahlen ohne Beachtung von Überträgen. Die Berechnung relativer Sprungziele während des oben erwähnten Sprungbefehls erfolgt ebenfalls mithilfe der ALU.

2.3 Befehlssatz der CPU

Aufgrund der Verarbeitungsbreite von 2 Bit und einem Umfang von einem Befehlswort pro Befehl stehen insgesamt 4 Befehle zur Verfügung. Diese sind in der folgenden Tabelle zusammengefasst.

Mnemonic	Argument	Befehlscode	Bedeutung
READ	direkte Adr.	01 XX	Lesen eines Datenwortes an der Speicheradresse XX in den Akkumulator
WRITE	direkte Adr.	10 XX	Schreiben des Akkumulatorinhaltes in die Speicherzelle XX
ADD	—	11	Addieren von 1 (permanentes Register) zum Akkumulatorinhalt
JUMP	relative Adr.	00 XX	Relativer Sprung, d. h. XX wird zum aktuellen Befehlszähler hinzuaddiert

2.4 Struktureller Aufbau der CPU

Das in Abbildung 1 dargestellte Blockschaltbild zeigt die interne Struktur der CPU bestehend aus Steuer- und Datenpfad. Als Steuerpfadkomponente ist das in der Mitte erkennbare Steuerwerk zu nennen. Der Datenpfad wird durch einige Register, Multiplexer und die ALU repräsentiert. Im Folgenden werden die einzelnen Register und ihre Bedeutung für die Funktion der CPU erklärt:

- Das Befehlsregister **BR** speichert den Befehlscode des gerade in der Abarbeitung befindlichen Befehls.
- Der Instruction-Pointer **IP** beinhaltet diejenige Speicheradresse, von der entweder der nächste Befehl oder das Argument des gerade in Abarbeitung befindlichen Befehls zu lesen ist.
- Der Akkumulator **AA** fungiert als zentrales Operandenregister der ALU und ist somit eine Quelle und Ziel jeder arithmetischen Operation der CPU.
- Das Register **BB** ist ein konstantes Hilfsregister mit dem Wert 1 (ist also gar kein richtiges Register). Es wird in Verbindung mit dem Befehl ADD zur Addition des Wertes 1 zum Akkumulator benötigt.
- Das Adressregister **AR** dient der Aufnahme des Adressargumentes bei den Befehlen READ und WRITE. Bei dem Befehl JUMP wird in **AR** ebenfalls das Argument (also der relative Abstand zum Sprungziel) aufgenommen.

2.5 Steuerwerk nach der Sequence-Counter Methode

Das Steuerwerk ist wesentlicher Bestandteil jeder CPU-Architektur. Es ist für die Steuerung des zeitlichen Ablaufs aller Aktionen innerhalb der CPU verantwortlich. Basis eines jeden Steuerwerks ist in der Regel ein getakteter Automat. Für die konkrete Implementierung eines solchen Steuerwerks gibt es verschiedene Möglichkeiten. Die hier zunächst angewandte bedient sich eines einfachen Sequence-Counters, welcher die aktuelle Befehlsausführungsphase durch einen einfachen 1-aus-3-Code darstellt (wie wir gleich sehen werden, besteht die Befehlsausführung aus zwei bis drei Phasen). Mit Hilfe eines speziellen Modulo-3 Zählers (praktisch eine Art 3-Bit Schieberegister in dem eine 1 rotiert) werden hier die Signale F1, F2 und F3 generiert, welche die jeweils aktive Abarbeitungsphase anzeigen.

In Abhängigkeit dieser Signale und dem Inhalt des Befehlsregisters **BR** wird durch eine kombinatorische Logik eine Menge von Steuersignalen erzeugt. Diese Signale steuern sowohl den Datenfluss über die Multiplexer als auch die Eingänge der Register.

2.6 Funktioneller Ablauf in der CPU

Der funktionelle Ablauf in der CPU basiert auf dem in Abbildung 2 dargestellten Flussdiagramm.

Die Abarbeitung jedes Befehls der CPU wird dabei in mehrere Befehlsphasen unterteilt, von denen jede einen Takt beansprucht. Dabei haben die einzelnen Abarbeitungsphasen unterschiedliche Aufgaben. Die erste Phase bezeichnen wir auch als *FETCH*-Phase. Hier wird der Befehlscode gelesen. Die weiteren Phasen fasst man oft unter dem Begriff *EXECUTION*-Phase zusammen. Bei uns umfasst dies bei den Befehlen READ, WRITE und JUMP 2 Takte (F2+F3), und bei ADD lediglich einen Takt (F2).

Die Abarbeitung eines Befehls beginnt stets mit der *FETCH*-Phase, welche mit **F1** gekennzeichnet ist. In ihr wird der Befehlscode des nächsten abzuarbeitenden Befehls aus dem Programmspeicher in das Befehlsregister **BR** geladen (engl. fetch: holen). Dabei wird wie folgt verfahren:

- über den Multiplexer M1_S wird der Inhalt des Instruction-Pointers an den Adressausgang ADDR der CPU gelegt. **IP** zeigt ja in dem Moment auf die Speicheradresse, von der der Befehlscode gelesen werden soll.
- Mit einer 1 am RD-Signal wird angezeigt, dass aus dem Speicher gelesen werden soll.

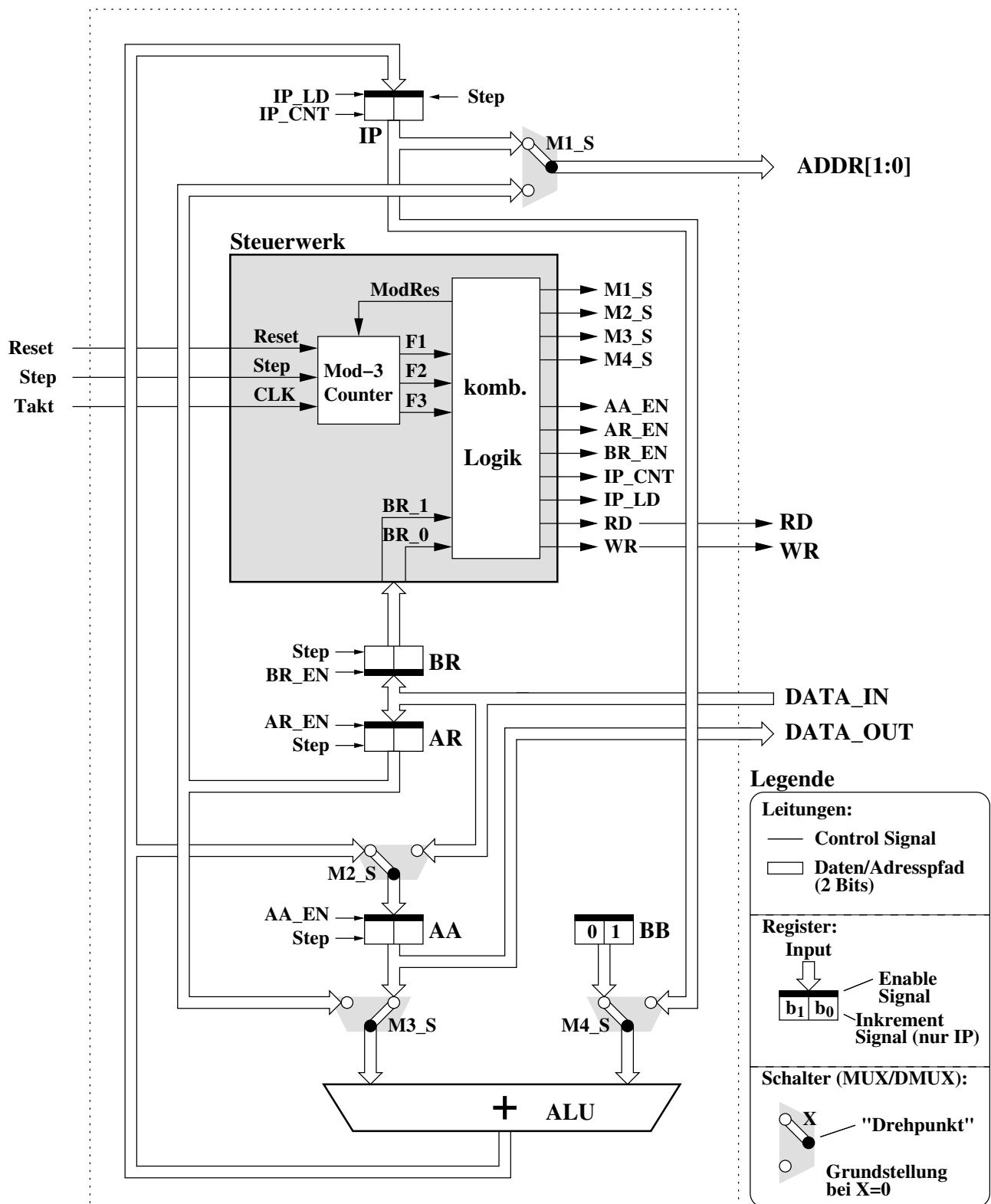


Abbildung 1: Blockschaltbild der 2-Bit CPU.

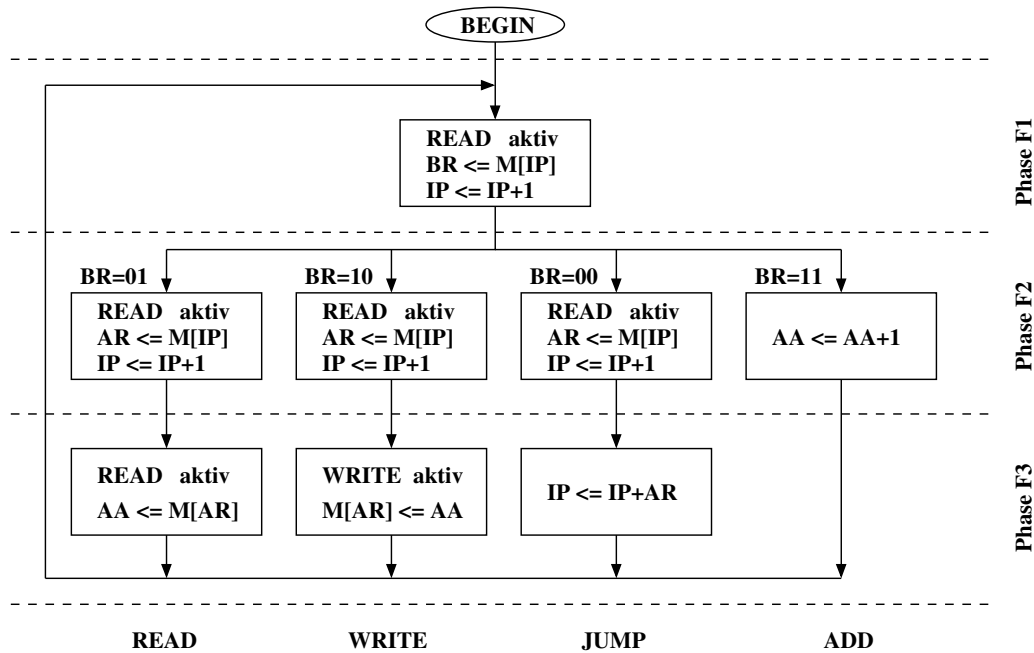


Abbildung 2: Flussdiagramm der CPU.

- Durch ein aktives BR_EN-Signal wird die Übernahme des an DATA_IN anliegenden Datums in das Befehlsregister vorbereitet.
- Mit dem aktiven Signal IP_CNT wird angezeigt, dass der Instruction-Pointer um eins erhöht werden soll.

Mit der nächsten steigenden Taktflanke werden diese Operationen schließlich ausgeführt (sofern sie getaktete Komponenten betreffen). Der Opcode des Befehls befindet sich nun im Befehlsregister und der Instruction-Pointer zeigt auf die nächste Speicherstelle. Diese kann entweder das Argument des Befehls im Fall von READ und WRITE, oder bereits den nächsten Befehl enthalten. Letzteres ist der Fall, wenn gerade ein ADD-Befehl gelesen wurde. Denn dieser besitzt kein Argument.

Anhand des WRITE-Befehls soll nun erklärt werden, wie sich der weitere Ablauf während der Phasen **F2** und **F3** gestaltet. In der Phase **F2** wird zunächst das Adressargument des WRITE-Befehls in das Adressregister geladen. Dazu wird wie folgt vorgegangen:

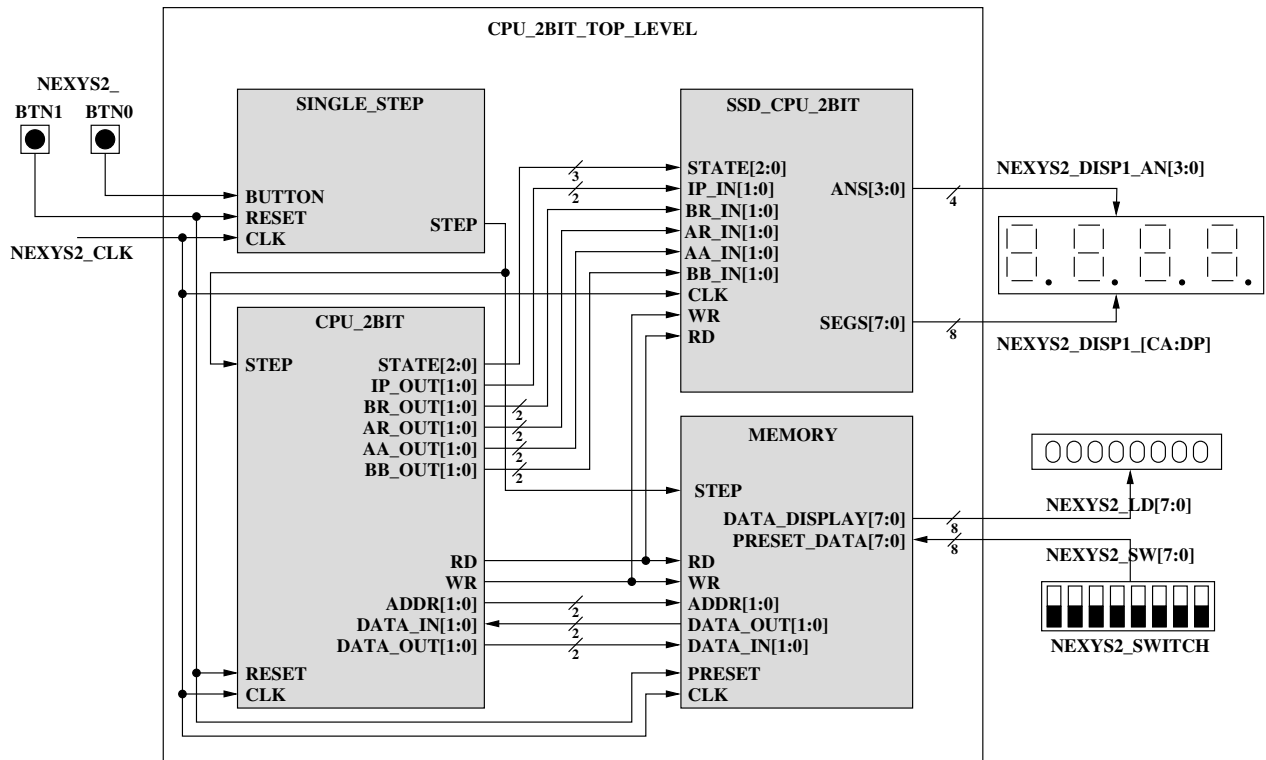
- Der Instruction-Pointer, der inzwischen auf das Adressargument des WRITE-Befehls im Programmspeicher verweist, wird wie in **F1** über den Multiplexer M1_S an den Adressausgang ADDR der CPU gelegt.
- Durch ein aktives RD-Signal wird wiederum die Richtung des Datentransfers vom Speicher zur CPU festgelegt.
- Mittels aktivem AR_EN-Signal wird diesmal das Adressregister **AR** als Ziel des Datentransfers ausgewählt.
- Das Signal IP_CNT zeigt an, dass der Instruction-Pointer wiederum um eins erhöht werden soll.

Mit der nächsten steigenden Taktflanke wird das an DATA_IN liegende Datum in das Adressregister **AR** transferiert und der Instruction-Pointer um eins erhöht. **IP** zeigt damit auf den Folgebefehl.

Die Phase **F3** der Befehlsabarbeitung dient zur eigentlichen Ausführung des Befehls. Im Fall des WRITE-Befehls wird dabei der Inhalt des Akkumulators in die durch das Adressregister adressierte Speicherzelle geschrieben. Im Einzelnen gestaltet sich der interne Ablauf wie folgt:

- Der Inhalt des Adressregisters **AR** wird über den Multiplexer M1_S an den Adressausgang der CPU gelegt, um die zu schreibende Speicherzelle zu adressieren.
- Der Inhalt des Akkumulators liegt am Datenausgang DATA_OUT der CPU bereit, um in den Programmspeicher übernommen zu werden.
- Durch ein aktives WR-Signal wird dem Speicher mitgeteilt, dass geschrieben werden soll.

Da der ADD-Befehl keine dritte Phase benötigt (schließlich muss kein Argument gelesen werden), wird in diesem Fall in unserer Implementierung der Sequence-Counter mit Hilfe des synchronen Reset Signals ModRes von **F2** nach **F1** zurückgesetzt, um mit dem Lesen des nächsten Befehls fortzufahren.



Das Schreiben in den Speicher ist synchron ausgelegt, d. h. es wird mit der steigenden Taktflanke geschrieben. Weiterhin finden Schreibvorgänge nur dann statt, wenn das STEP-Signal aktiv ist.

Über die Schiebeschalter *SW0–SW7* wird eine Voreinstellung des Speichers ermöglicht. Befindet sich einer der Schalter auf der unteren Stellung, so wird dasjenige Bit auf den Wert logisch 0, sonst auf logisch 1 gesetzt. Die Zuordnung der einzelnen Schiebeschalter zu den Bitstellen im Speicher wurde folgendermaßen getroffen:

SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0
Word 3 MSB	Word 3 LSB	Word 2 MSB	Word 2 LSB	Word 1 MSB	Word 1 LSB	Word 0 MSB	Word 0 LSB

Der Taster *BTN1* dient zur Generierung des Reset-Signals für die CPU. Bei Betätigung wird die CPU in den Anfangszustand zurückversetzt und der an den Schiebeschaltern eingestellte Bitvektor wird in den Speicher der CPU übertragen.

Der komplette Speicherinhalt wird durch die Leuchtdioden (*LD0 ... LD7*) repräsentiert. Dabei ist die Nummerierung analog zu den Schiebeschaltern gewählt, d. h. *LD0* stellt ebenfalls LSB von Word 0 und *LD7* MSB von Word 3 dar.

Das Befehlsregister **BR** wird auf den beiden oberen, senkrechten Segmenten (*CF*, *CB*) der Anzeige *AN3* dargestellt. Die beiden senkrechten Segmente rechts daneben (Anzeige *AN2*) repräsentieren den Inhalt des Adressregisters **AR**. Die beiden unteren, senkrechten Segmente (*CE*, *CC*) von Anzeige *AN3* stellen den Akkumulator **AA** dar. Wiederum rechts daneben auf Anzeige *AN2* befindet sich die Ausgabe des B-Registers **BB**. Weiterhin wird über die Sieben-Segment-Anzeige *AN1* die jeweilige Phase der Befehlsausführung der CPU und das RD-Signal dargestellt. Dabei bedeutet ein unterer Querstrich (Segment *CD*) Phase F1, ein mittlerer Querstrich (Segment *CG*) Phase F2 und ein oberer Querstrich (Segment *CA*) Phase F3. Die rechte Anzeige *AN0* stellt den Inhalt des Instruction-Pointers dezimal und das WR-Signal als Dezimalpunkt dar. Abbildung 4 gibt eine grafische Übersicht zu dieser Belegung.

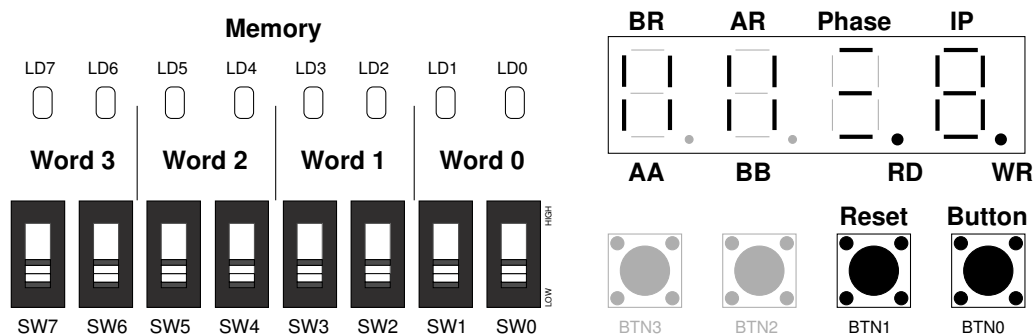


Abbildung 4: Zuordnung der Ein- und Ausgaben auf dem Nexys 2 Board.

Hinweis: Wer sich das Interface der CPU_2BIT-Komponente in Abbildung 3 aufmerksam angeschaut hat, und dies mit dem in Abbildung 1 suggerierten Interface vergleicht, der wird festgestellt haben, dass die CPU in Abbildung 3 ein paar mehr Ausgänge besitzt. Dabei handelt es sich um den Block von STATE[2:0] bis BB_OUT[1:0]. In unserem Fall müssen wir diese normalerweise internen Signale jedoch aus Visualisierungsgründen nach außen führen. Andernfalls würden wir nicht sehen, was innen abläuft.

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!

Zu dieser Versuchsanleitung wird ein VHDL-Quelltext mitgeliefert, der neben dem Grundgerüst der CPU einige bereits oben vorgestellte Module enthält. An den markierten Stellen im Quelltext sind dabei entsprechende Ergänzungen vorzunehmen.

Um das gesamte Design nicht in viele kleine Dateien zerbrechen zu lassen, sind sämtliche benötigten Komponenten (Entities/Architectures) in einer Datei hintereinander aufgeführt. Zwar wird die Datei dadurch relativ groß und in gewisser Weise auch unübersichtlich, aber egal ...

4.1 Vertrautmachen mit der 2-Bit CPU

Um mit dem eigenen Entwurf der CPU zu beginnen ist es notwendig, die Funktion aller Teilschaltungen der CPU und des Gesamtsystems verstanden zu haben. Dabei sollte wie folgt vorgegangen werden:

- Verstehen der internen Struktur 2-Bit CPU anhand des Blockschaltbildes in Abbildung 1
- Analyse der Funktionsweise des Steuerwerkes
- Durchspielen der internen Vorgänge anhand von selbstgewählten Beispielbefehlen/Befehlssequenzen

Kontrollfragen

- Wodurch wird ein Bitmuster im Speicher einmal als Befehlscode, ein anderes mal als Datum interpretiert?
- Wie reagiert die CPU bei einem Überlauf (in IP und AA)?

4.2 Implementation des Instruction-Pointer Registers

Bei dem Instruction-Pointer handelt es sich um ein Register mit einer Breite von 2 Bit. Die vorgegebene Schnittstelle des Moduls ist in Listing 1 dargestellt.

Listing 1: Entity ip_counter

```
1 ENTITY ip_counter IS
2
3   PORT (
4       clk      : IN    STD_LOGIC;
5       reset    : IN    STD_LOGIC;
6       step     : IN    STD_LOGIC;
7       ip_ld    : IN    STD_LOGIC;
8       ip_cnt   : IN    STD_LOGIC;
9       ip_in    : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
10      ip_out   : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0));
11
12 END ip_counter;
```

Die oben aufgeführten Signale des Moduls sollen bei Ihrer Implementation die folgenden Funktionalität besitzen:

Signal	Bedeutung
clk	globaler Takteingang des Registers
reset	asynchrones Reset-Signal (high-aktiv)
step	zusätzliches globales Enable-Signal für den Einzelschrittbetrieb (high-aktiv)
ip_in	Dateneingang des Instruction-Pointer Registers
ip_out	Datenausgang des Registers
ip_ld	Laden des an ip_in liegenden Vektors in das Register (high-aktiv)
ip_cnt	Inhalt des Registers um eins erhöhen (high-aktiv)

4.3 Implementation eines 2-Bit Registers

Die zu entwerfende 2-Bit Registerkomponente bildet die Grundlage für alle weiteren in der CPU vorhandenen Register außer dem Instruction-Pointer (also Akkumulator, Befehlsregister und Adressregister). Die Schnittstelle ist definiert, wie sie in Listing 2 dargestellt ist.

Listing 2: Entity register_2bit

```
1 ENTITY register_2bit IS
2
3   PORT (
4       clk      : IN    STD_LOGIC;
5       reset    : IN    STD_LOGIC;
```

```

6      step    : IN    STD_LOGIC;
7      reg_en  : IN    STD_LOGIC;
8      reg_in  : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
9      reg_out : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0));
10
11 END register_2bit;

```

Die oben aufgeführten Signale sollen die folgende Funktionalität besitzen.

Signal	Bedeutung
clk	globaler Takteingang des Registers
reset	asynchrones Reset-Signal (high-aktiv)
step	zusätzliches globales Enable-Signal für den Einzelschrittbetrieb (high-aktiv)
reg_en	Laden des an reg_in liegenden Vektors in das Register (high-aktiv)
reg_in	Dateneingang des Registers
reg_out	Datenausgang des Registers

4.4 Implementation eines 2x2-Bit Multiplexers

Bei dem nun zu entwerfenden Multiplexer handelt es sich um eine rein kombinatorische Einheit, bei der in Abhängigkeit des Steuersignals *sel* der am Eingang *a* liegende Vektor oder der an *b* liegende Vektor an den Ausgang *y* des Multiplexers durchgeschaltet wird. Anschließend ist in Listing 3 die vorgegebene Schnittstelle des Moduls dargestellt.

Listing 3: Entity mux

```

1 ENTITY mux IS
2
3   PORT (
4       sel : IN    STD_LOGIC;
5       a   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
6       b   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
7       y   : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0));
8
9 END mux;

```

Der Multiplexerausgang soll im Fall von *sel*='0' den Wert von *a* besitzen und sonst den Wert von *b* besitzen.

4.5 Entwurf der ALU

Die ALU der 2-Bit CPU ist durch die Schnittstelle in Listing 4 gegeben.

Listing 4: Entity alu

```

1 ENTITY alu IS
2
3   PORT (
4       a : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
5       b : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
6       y : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0));
7
8 END alu;

```

Bei der ALU handelt es sich um einen einfachen 2-Bit-Adder ohne Übertragsein- und Ausgang. Also um eine kombinatorische Logik, welche die beiden Eingangsinformationen *a* und *b* additiv ohne Berücksichtigung des Übertrages verknüpft.

4.6 Entwurf und Implementation des Steuerwerks der CPU

Wie in Abbildung 1 dargestellt, soll es sich dabei um einen Modulo-3 Zähler mit einer angeschlossenen Kombinatorik handeln. Der Zähler soll durch das *Reset*-Signal asynchron und durch das *ModRes*-Signal (kein Steuersignal im eigentlichen Sinne der CPU) synchron rücksetzbar sein. Er erzeugt die Steuersignale **F1**, **F2** und **F3**, welche die interne Phase der

Befehlsabarbeitung der CPU repräsentieren. Für den Entwurf der Kombinatorik, welche für die Generierung der Steuersignale zuständig ist, soll die nachfolgende und auch später wichtige Tabelle behilflich sein. Diese Tabelle muss im Rahmen der Übung vervollständigt und die benötigten Logikgleichungen aus ihr extrahiert werden.

Befehl	Phase	BR_EN	AR_EN	AA_EN	IP_LD	IP_CNT	RD	WR	ModRes	M1_S	M2_S	M3_S	M4_S
READ	F1	1	0	0	0	1	1	0	0	0	0	0	0
	F2	0	1	0	0	1	1	0	0	0	0	0	0
	F3	0	0	1	0	0	1	0	0	1	1	1	1
WRITE	F1	1	0	0	0	1	1	0	0	0	0	0	0
	F2	0	1	0	0	1	1	0	0	0	0	0	0
	F3	0	0	0	0	0	0	1	0	1	1	1	1
ADD	F1	1	0	0	0	1	1	0	0	0	0	0	0
	F2	0	0	1	0	0	1	0	1	0	0	0	0
JUMP	F1												
	F2												
	F3												

Zur Implementierung des Steuerwerks die in Listing 5 dargestellte Schnittstelle zu verwenden.

Listing 5: Entity steuerwerk

```

1 ENTITY steuerwerk IS
2
3   PORT (
4       clk      : IN  STD_LOGIC;
5       reset    : IN  STD_LOGIC;
6       step     : IN  STD_LOGIC;
7       br       : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
8       state    : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
9       br_en    : OUT STD_LOGIC;
10      ar_en    : OUT STD_LOGIC;
11      aa_en    : OUT STD_LOGIC;
12      ip_ld    : OUT STD_LOGIC;
13      ip_cnt   : OUT STD_LOGIC;
14      rd       : OUT STD_LOGIC;
15      wr       : OUT STD_LOGIC;
16      m1_s     : OUT STD_LOGIC;
17      m2_s     : OUT STD_LOGIC;
18      m3_s     : OUT STD_LOGIC;
19      m4_s     : OUT STD_LOGIC);
20
21 END steuerwerk;
```

4.7 Zusammenfügen aller Teilkomponenten zur fertigen CPU

Nachdem alle Teilkomponenten entworfen wurden, kann daraus nun die eigentliche CPU zusammengesetzt werden. Dabei besitzt die fertige CPU das in Listing 6 abgebildete Interface.

Listing 6: Entity cpu_2bit

```

1 ENTITY cpu_2bit IS
2
3   PORT (
4       clk      : IN  STD_LOGIC;
5       reset    : IN  STD_LOGIC;
6       step     : IN  STD_LOGIC;
7       rd       : OUT STD_LOGIC;
8       wr       : OUT STD_LOGIC;
9       addr     : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
10      state    : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
11      ar_out    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
12      br_out    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
13      ip_out    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
14      aa_out    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
15      bb_out    : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
16      data_in   : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
```

```

17   data_out : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
18
19 END cpu_2bit;

```

Die oben aufgeführten Signale haben dabei die folgende Funktionalität:

Signal	Bedeutung
clk	globaler Takteingang des Registers
reset	asynchrones Reset-Signal (high-aktiv)
step	zusätzliches globales Enable-Signal für den Einzelschrittbetrieb (high-aktiv)
rd	lesender Speicherzugriff (high-aktiv)
wr	schreibender Speicherzugriff (high-aktiv)
addr	Adressausgabe der CPU
state	Ausgabe der einzelnen Phasen im 1-aus-3-Code

Signal	Bedeutung
ar_out	Ausgabe des Adressregisterinhaltes
br_out	Ausgabe des Befehlsregisterinhaltes
ip_out	Ausgabe des Instruction-Pointer Inhaltes
aa_out	Ausgabe des Akkumulatorinhaltes
bb_out	Ausgabe des Inhalts von Register B
data_in	Dateneingang der CPU
data_out	Datenausgang der CPU

Die einzelnen Komponenten sind nun unter Verwendung einer Strukturbeschreibung zusammenzufügen. Dabei sollte das Blockschaltbild der CPU aus Abbildung 1 behilflich sein.

5 Aufgaben

Hinweis: In den folgenden Aufgaben soll die entworfene CPU für den Spartan-3E FPGA synthetisiert werden. Dabei sind folgende Schritte zu vollziehen:

1. Ergänzen des beiliegenden VHDL-Quelldateien um die oben beschriebene CPU. Gegebenenfalls sollte man sich per Simulation von der Korrektheit der einzelnen Module vergewissern. Die beiliegende Datei `cpu_2bit_top_level.ucf` sollte mit übernommen werden, da hier bereits Pinbelegungen voreingestellt sind. Synthetisieren Sie Ihr Design und laden Sie es in das FPGA.
2. Um den Entwurf zu testen ist wie folgt vorzugehen. Zunächst ist ein geeignetes Programm mittels der Schalter *SW7-0* einzustellen. Durch Drücken des Tasters *BTN1* wird die CPU in den Ausgangszustand versetzt. Dabei wird das an den Schaltern eingestellte Programm in den Hauptspeicher der CPU übernommen und kann nun abgearbeitet werden. Der Inhalt des Hauptspeichers wird durch die LED-Reihe visualisiert. Durch Drücken des Tasters *BTN0* kann schrittweise Befehlsphase für Befehlsphase abgearbeitet werden. Dabei wird bei jedem Drücken und wieder Loslassen fortgefahren.

Es sollten geeignete Testprogramme eingestellt werden, um jeden der vier Befehle mindestens einmal auf seine Funktionstüchtigkeit hin zu überprüfen.

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>

25. November 2011