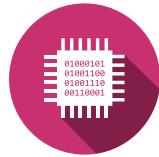


# Mikroprozessor-interne Datenbusse

Labor Digitales Design

## Inhalt

1	Ziel .....	2
1.1	Verbindung der Register zu den Datenbussen .....	4
1.2	Verbindung zum Ein-/Ausgabe-Bus .....	4
1.3	Daten aus der Instruktion .....	5
1.4	Implementierung .....	5
2	Software-erstellung eines seriellen Ports .....	6
2.1	Serielle Übermittlung .....	6
2.2	Algorithmus .....	6
2.3	Implementierung .....	7
2.4	Simulation .....	7
3	Checkout .....	8
	Glossar .....	9



# 1 | Ziel

Dieses Labor zielt darauf ab, die Verwendung von Tri-State-Schaltungen zu üben, insbesondere im Kontext von gemeinsam genutzten Datenbussen auf einem Xilinx **PicoBlaze**  $\mu$ Prozessor.

Es bietet Einblicke in die interne Funktionsweise eines  $\mu$ Prozessor und konzentriert sich auf die Interaktion zwischen seinen Komponenten. Ein besonderer Schwerpunkt liegt auf der Registerdatei (auch bekannt als Datenspeicherregister), die die Register  $s_0, s_1, s_2$  und  $s_3$  umfasst.

Durch die Erkundung der **Arithmetic and Logical Unit (ALU)**-Operationen, der Registerdatei und ihrer Verbindung zu einem gemeinsamen Bus zeigt dieses Labor, wie Tri-State-Logik es mehreren Komponenten ermöglicht, über einen gemeinsamen Datenpfad zu kommunizieren, ohne sich gegenseitig zu stören.

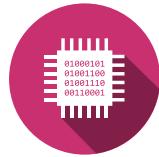
**Bibliothek IO**

**i**

In diesem Labor werden Sie Tristate Komponenten benutzt. Diese befinden sich in der Bibliothek **IO**.

Falls diese im Komponentenbrowser nicht sichtbar ist, können Sie sie wie folgt hinzufügen.

The screenshot shows the Xilinx ISE software interface. At the top, there is a toolbar with various icons. Below it is a menu bar. On the left, there is a 'Component Browser' window showing categories like 'common', 'gates', and 'sequential'. On the right, there is an 'Explore Library' window showing a list of components including 'COR', 'IND', 'ID' (which is checked), 'MIB', 'MUL', 'NUM', 'operators', and 'PhA'. The 'ID' component is highlighted with a red box. At the bottom of the interface, there are 'OK' and 'Cancel' buttons.



Die Abbildung 1 zeigt einen Teil des Xilinx PicoBlaze  $\mu$ Prozessors, der aus folgenden Komponenten besteht:

- einer ALU
- einem Registerfile mit 4 Registern ( $s_0, s_1, s_2, s_3$ )
- einer Schnittstelle zu einem Ein-/Ausgabe-Bus (Input/Output (I/O))
- und einer Verbindung zum  $\mu$ Prozessor-Befehl.

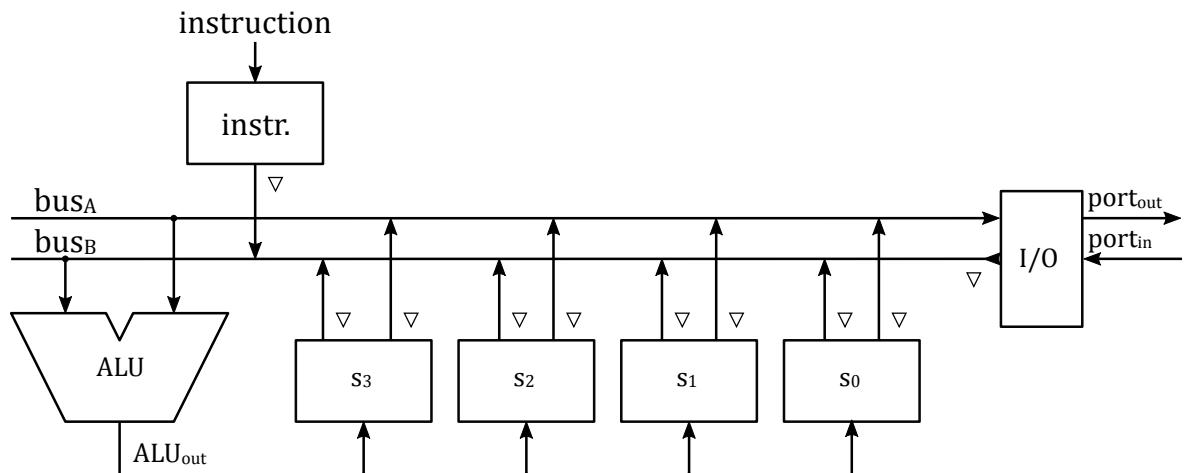


Abbildung 1 - Komponenten des  $\mu$ Prozessors, die durch die Datenbusse  $bus_A$  und  $bus_B$  verbunden sind.

Die Komponenten sind durch zwei Datenbusse,  $bus_A$  und  $bus_B$ , verbunden, die zum Übertragen von Daten zwischen den Komponenten verwendet werden. Auf dem  $bus_A$  können die 4 Register Daten an die ALU übertragen und wird für den *ersten Operand* einer Operation verwendet. Während der  $bus_B$  mit den 4 Registern, dem I/O-Block und dem Instruktionsblock verbunden ist und für den *zweiten Operand* einer Operation verwendet wird.

```

1 ADD s0 s1 # Adds the contents of register s1 to register s0. s0 = s0 + s1
2 #   ^   ^   ^
3 #   |   |   --- Second operand, register s1
4 #   |   +--- First operand, register s0
5 #   +----- Operation, ADD
  
```

Listing 1 - Beispiel für eine Assembler-Anweisung ADD.

Der erste Operand ist das Register  $s_0$ , der zweite ist das Register  $s_1$ .

Der Datenfluss für  $bus_A$  verläuft:

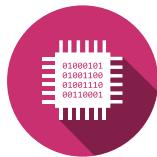
- von einem der 4 Register  $s_0-s_3$  zur ALU für den ersten Operand einer Operation.
- von einem der 4 Register  $s_0-s_3$  zum I/O-Block, um Daten an ein externes Gerät zu schreiben.

Der Datenfluss für  $bus_B$  verläuft:

- von einem der 4 Register  $s_0-s_3$  zur ALU für den zweiten Operand einer Operation.
- vom I/O-Block zur ALU für den zweiten Operand einer Operation.
- vom Instruktionsblock zur ALU für den zweiten Operand einer Operation.

Der Datenfluss für  $ALU_{out}$  verläuft:

- von der ALU zu einem der 4 Register  $s_0-s_3$  zum Schreiben des Ergebnisses einer Operation.



## 1.1 Verbindung der Register zu den Datenbussen

Abbildung 2 zeigt zwei Register  $s_1$  und  $s_2$  mit ihrem Steuerblock zum Schreiben in die Register oder zum Lesen aus den Registern auf entweder bus<sub>A</sub> oder bus<sub>B</sub>.

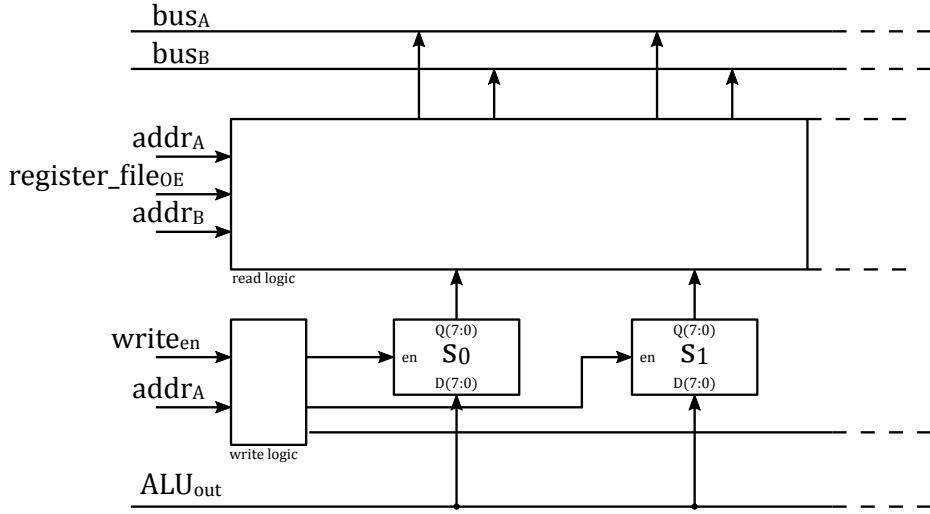


Abbildung 2 - Datenregister  $s_X$

Der Schaltkreis von Abbildung 2 ermöglicht es, dass die Ausgänge der Register sowohl mit den Bussen bus<sub>A</sub> als auch bus<sub>B</sub> verbunden werden können. Das erstellte System sollte es ermöglichen, dass ein Register seine Daten auf bus<sub>A</sub> und ein anderes auf bus<sub>B</sub> legt.

Die Zahlen addr<sub>A</sub> und addr<sub>B</sub> geben an, welches Register seine Informationen auf bus<sub>A</sub>, bzw. bus<sub>B</sub> überträgt. Das Signal register\_file<sub>OE</sub> gibt an, ob Daten aus dem ausgewählten Register auf den bus<sub>B</sub> gebracht werden sollen und verhindert einen Konflikt mit Daten, die vom I/O-Port oder dem Instruktionenblock kommen.

Ebenso wird das Signal write<sub>en</sub> und addr<sub>A</sub> verwendet, um in die Register zu schreiben. Somit wird bei einer Operation, deren erster Operand das Register ist, das durch addr<sub>A</sub> ausgewählt wurde, das Ergebnis in dieses Register geschrieben. Wie im Assembler-Code in Listing 1 zu sehen ist, ist der erste Operand das Register  $s_0$ , und das Ergebnis der Operation wird in dieses Register geschrieben.



Entwickeln Sie die Lese- und Schreiblogik für die Register  $s_0$ ,  $s_1$ ,  $s_2$  und  $s_3$  im Block **MIB/aluAndRegister**.

## 1.2 Verbindung zum Ein-/Ausgabe-Bus

Beim Lesen von Daten von aussen muss das Steuersignal port<sub>in\_OE</sub> aktiviert werden, und dann werden die Daten vom Bus port<sub>in</sub> auf den bus<sub>B</sub> geschrieben. Beim Schreiben von Daten nach aussen werden die Daten vom bus<sub>A</sub> auf den Bus port<sub>out</sub> geschrieben, und ein externes Signal an die ALU, write<sub>strobe</sub>, wird über die Testbench aktiviert. Dadurch können diese Daten in einem Register außerhalb des µProzessors aufgezeichnet werden.

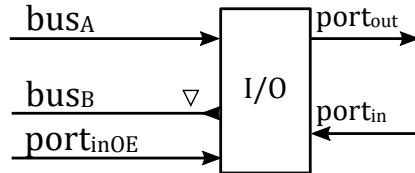
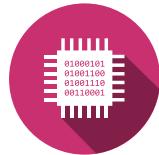


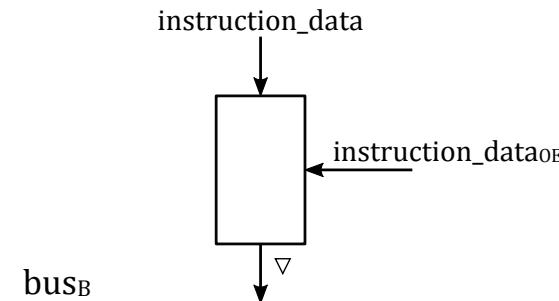
Abbildung 3 - Ein- / Ausgangsblock



Entwickeln Sie das interne Schema des I/O-Blocks in **MIB/aluAndRegisters** aus Abbildung 3.

### 1.3 Daten aus der Instruktion

Für den zweiten Operanden einer Operation kann ein konstanter Wert in der Instruktion kodiert und auf den  $bus_B$  der ALU gebracht werden. Der Block, der diesen Transfer verwaltet, ist in Abbildung 4 dargestellt.

Abbildung 4 - Daten aus dem  $\mu$ Prozessor-Befehl stammend

```

1 LOAD s0 10 # Loads the constant 10 to register s0. s0 = 10
2 #   ^   ^
3 #   |   |   +- immediate (constant) value 10
4 #   |   +--- First operand, register s0
5 #   +----- Operation, LOAD

```

Listing 2 - Beispiel für eine Assembler-Anweisung **LOAD**.

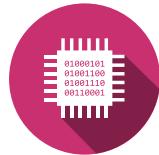
Der erste Operand ist das Register  $s_0$ , der zweite Wert ist ein unmittelbarer Wert (Konstante).



Entwickeln Sie das interne Schema des Instruktionsblocks aus Abbildung 4 in **MIB/aluAndRegisters**.

### 1.4 Implementierung

Basierend auf den vorherigen Punkten, ist der interne Busschaltkreis des  $\mu$ Prozessors in **MIB/aluAndRegisters** vervollständigt.



## 2 | Software-erstellung eines seriellen Ports

### 2.1 Serielle Übermittlung

Die Abbildung 5 gibt das zeitliche Verhalten der seriellen Übermittlung eines Datenwortes.

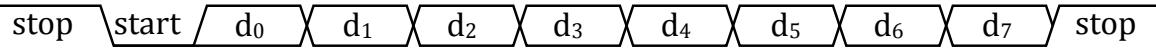


Abbildung 5 - Serielle Übermittlung

Bei einer seriellen Übertragung hat das Signal auf dem Bus standardmäßig ein hohes Niveau (Logik 1). Die Übertragung beginnt mit einem Startbit (Logik 0), gefolgt von den Datenbits, und endet wieder mit einem Stoppbit (Logik 1). Die Datenbits werden nacheinander gesendet, beginnend mit dem am wenigsten signifikanten Bit (LSB) bis zum am meisten signifikanten Bit (MSB).

In unserer Anwendung wird das serielle Signal auf dem am wenigsten signifikanten Bit des  $\text{port}_{\text{out}}$ -Busses resp.  $\text{port}_{\text{out}}[0]$  übertragen. Im Testbench ist dieser Bus mit einem externen Register **MIB\_test/MIB\_tb/I2** verbunden. Der  $\text{write}_{\text{strobe}}$ -Befehl aus dem Testbench steuert das Schreiben in dieses Register.

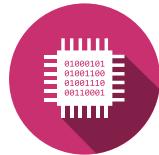
### 2.2 Algorithmus

Der zu programmierende Algorithmus ist wie folgt:

```

1 LOAD    s3, FF          # load stop bit
2 OUTPUT  s3              # output stop bit
3 LOAD    s3, s3          # no operation
4 LOAD    s3, s3          # no operation
5 LOAD    s3, s3          # no operation
6 LOAD    s3, s3          # no operation
7 LOAD    s0, 00          # load start bit
8 OUTPUT  s0              # output start bit
9 INPUT   s1              # load word to send
10 OUTPUT  s1             # output word, LSB is considered
11 SR0    s1              # shift word, bit 1 -> LSB
12 OUTPUT  s1             # output bit 1
13 SR0    s1              # bit 2 -> LSB
14 OUTPUT  s1             # output bit 2
15 SR0    s1              # bit 3 -> LSB
16 OUTPUT  s1             # output bit 3
17 SR0    s1              # bit 4 -> LSB
18 OUTPUT  s1             # output bit 4
19 SR0    s1              # bit 5 -> LSB
20 OUTPUT  s1             # output bit 5
21 SR0    s1              # bit 6 -> LSB
22 OUTPUT  s1             # output bit 6
23 SR0    s1              # bit 7 -> LSB
24 OUTPUT  s1             # output bit 7
25 LOAD    s3, s3          # no operation
26 OUTPUT  s3              # output stop bit
  
```

Listing 3 - Software-Implementierung des seriellen Übertragungsprotokolls



Studieren und verstehen Sie den Algorithmus des seriellen Übertragungsprotokolls Listing 3.

## 2.3 Implementierung

Jede Zeile oder Anweisung muss im Testbench-Testerblock implementiert werden.



Vervollständigen Sie den Testbench-Tester **MIB\_test/MIB\_tester**, um die Anweisungsfolge für die serielle Übertragung Listing 3 auszuführen.



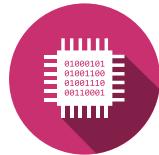
Es ist wichtig, keine Busse im Hochimpedanzzustand zu belassen. Programmieren Sie den Algorithmus so, dass immer ein Signal auf  $\text{bus}_A$  und  $\text{bus}_B$  vorhanden ist, auch wenn keine Informationen von ihnen abgefragt werden.

## 2.4 Simulation



Simulieren Sie den Testbench **MIB\_test/MIB\_tb** mit der Simulationsdatei **\$SIMULATION\_DIR/MIB.do**.

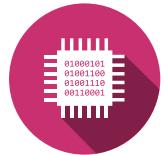
Wie viele Bits und welcher Datenwert wird übertragen?



## 3 | Checkout

Dies ist das Ende des Labors, Sie haben erfolgreich die interne Struktur des minimalistischen  $\mu$  Processors Xilinx **PicoBlaze** aufgebaut. Bevor Sie das Labor verlassen, stellen Sie sicher, dass Sie die folgenden Aufgaben abgeschlossen haben:

- Schaltkreisdesign
  - Überprüfen Sie, ob der Block **MIB/aluAndRegisters** mit den genannten Funktionen entworfen und getestet wurde.
- Simulationen
  - Stellen Sie sicher, dass Sie den Algorithmus der seriellen Übertragung Listing 3 verstanden haben.
  - Die spezifischen Anweisungen wurden in **MIB\_test/MIB\_tester** implementiert.
  - Der Wert und die Anzahl der übertragenen Bits wurden aus der Simulation gelesen.
- Dokumentation und Projektdateien
  - Stellen Sie sicher, dass alle Schritte (Design, Konvertierungen, Simulationen) gut in Ihrem Laborbericht dokumentiert sind.
  - Speichern Sie das Projekt auf einem USB-Stick oder dem gemeinsamen Netzlaufwerk (\\\filer01.hevs.ch).
  - Teilen Sie die Dateien mit Ihrem Laborpartner, um die Kontinuität der Arbeit zu gewährleisten.



# Glossar

**ALU – Arithmetic and Logical Unit** [2](#), [3](#), [3](#), [3](#), [3](#), [3](#), [3](#), [3](#), [4](#), [5](#)

**I/O – Input/Output** [3](#), [3](#), [3](#), [3](#), [4](#), [5](#)

**PicoBlaze:** PicoBlaze is a small, 8-bit microcontroller designed by Xilinx for use in FPGAs. It is often used in educational settings to teach basic microcontroller concepts. [2](#), [3](#), [8](#)