Besondere Lernleistung
Berufliche Schulen Groß-Gerau

Implementierung eines Mikrorechners in VHDL auf einem FPGA

Technikwissenschaft & Technologie

Autor: Markus Schneider

Klasse: 13BGDE

Wohnort: Mörfelden-Walldorf

Prüfer: Uwe Homm, Friedrich Ernst

Abgabedatum: 30.03.2017

Inhaltsverzeichnis

Abkürzungsverzeichnis

EPU Educational Processing Unit

CPU Central Processing Unit

FPGA Field-programmable gate array

IC Integrated Circuit

LUT Look-Up-Table

UART Universal Asynchronous Receiver Transmitter

ALU Arithmetic logic unit

PC Program Counter

VHDL **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

Abbildungsverzeichnis

Tabellenverzeichnis

Selbstständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Besondere Lernleistung selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmitteln angefertigt wurde. Mir ist bekannt, dass bei nachgewiesenen Täuschungsversuchen die Prüfung als „nicht bestanden“ erklärt werden kann.

Unterschrift Autor	Datum

1. Einleitung

Diese Dokumentation beschreibt den Aufbau und die Funktionsweise der Educational Processing Unit (EPU). Das Projekt kam dadurch zustande, dass die Struktur und die Arbeitsweise eines Computers, insbesondere der Central Processing Unit (CPU) besser verstanden werden soll. Um dieses Ziel zu erreichen, wurde die EPU gebaut, da sie als lehrreicher Mikrorechner, wobei der Hauptteil der EPU nur aus der CPU besteht, die Funktionsweise und den Aufbau eines Alltagscomputer erklärt und somit Verständnis für die Komplexität unserer heutingen Rechner einbringt.

1.1. Was ist ein FPGA?

Ein Field-programmable gate array (FPGA) ist ein Integrated Circuit (IC), welcher zum Aufbau digitaler Schaltungen dient. Er besteht meist aus mehr als 100.000 Logikblöcken [1, S. 8]. Ins Deutsche übersetzt bedeutet FPGA soviel wie 'im Feld programmierbare [Logik-]Gatter-Anordnung', wobei 'im Feld' sich dabei auf den Konsumenten bezieht.

Die Logikschaltungen eines FPGA sind entweder über elektronische 'Schalter' der Konfiguration entsprechend verknüpft oder es werden sogenannte Look-Up-Tables (LUTs) benutzt, mit denen die Logikfunktion explizit realisiert werden kann. Eine LUT kann verschiedene kombinatorische Funktionen (NAND, XOR, AND, NOT, Multiplexer, etc.) aus den Eingangssignalen realisieren. Die meisten LUTs besitzen zwischen 4 und 6 Eingangssignale. Es ist auch möglich, mehrere LUTs in Serie zu schalten und die Limitierung durch die Eingangssignale zu verhindern. [2]

Da der FPGA aber nach Verlust des Stromanschlusses die Konfiguration der Logikelemente nicht von selbst speichert, wird meist zusätzlich noch ein Flash-Speicher verbaut, damit nach einem Stromverlust die alte Konfiguration wieder neu geladen werden kann. Dies hat auch den Vorteil, dass dadurch der Status des FPGA zurückgesetzt wird und somit ein 'Neustart' schnell möglich ist.

Anders als bei üblicher Programmierung von Computern kann bei einem FPGA keine herkömmliche Programmiersprache verwendet. Das 'Programmieren' wird üblicherweise als Konfiguration bezeichnet und wird mithilfe einer Hardwarebeschreibungssprache wie z.B. VHDL oder Verilog erledigt. Auch wird der geschriebene Code nicht kompiliert, wie es bei der normalen Programmierung ist, sondern synthetisiert, was soviel bedeutet wie 'künstlich herstellen'.

1.2. Beschreibung des genutzten FPGA

Vor der Suche nach einem FPGA-Board waren bereits einige Anforderungen klar, welche das Board erfüllen muss:

- Videoausgang
- Konfiguration über USB
- UART (Serielle Schnittstelle)
- 64KiB Speicher

Nach etwas längere Suche wurde das 'Waxwing Spartan 6 FPGA Development Board' von Numato Lab ausgewählt, da es alle Anforderungen auf einem Entwicklungsboard vereint. Einige Merkmale dieses Entwicklungsboards sind:

- Xilinx Spartan 6 FPGA
- Taktrate 100MHz
- Videoausgang (HDMI und VGA)
- LCD-Display
- Drei 7-Segment-Anzeigen
- USB-UART
- MicroSD-Support
- Größe: 180mm x 120mm

2. Übersicht der Hardwarekomponenten

2.1. Komponentenbeschreibung

Im Folgenden soll eine Übersicht aller Hardwarekomponenten erfolgen. Dadurch soll eine grobe Vorstellung der Funktionsweise des Mikrorechners entstehen, welche in Kapitel 3 vertieft wird. Vor der Beschreibung der einzelnen Hardwarekomponenten, soll durch Abbildung 2.1 das Zusammenspiel aller Komponenten gezeigt werden, auf welche während dieses Kapitels zurückgegriffen werden kann.

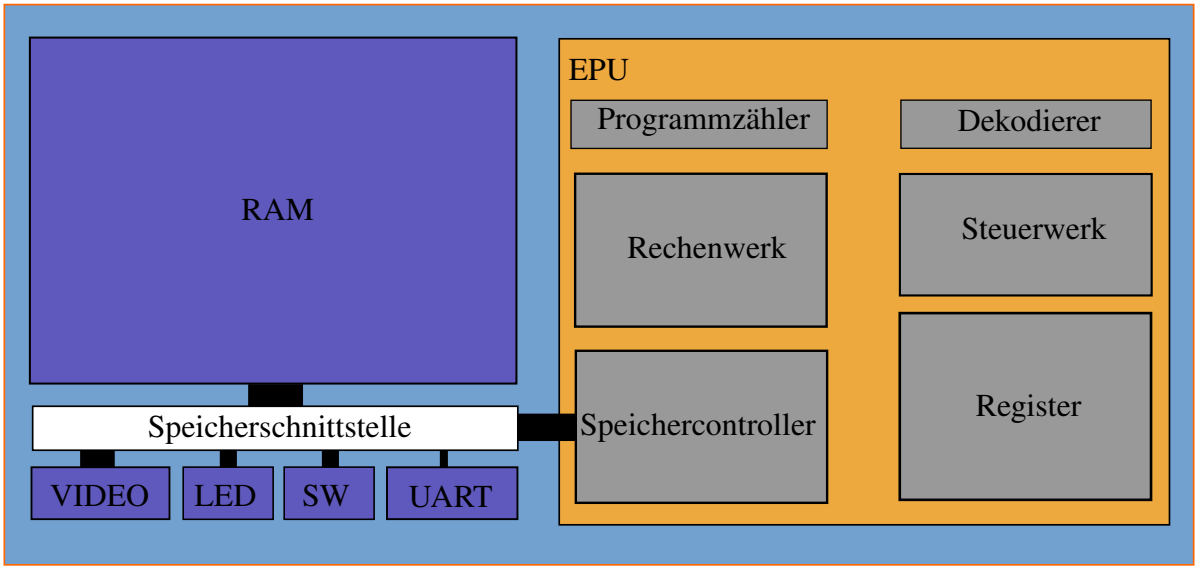


Abbildung 2.1.: Hardwarekomponentenübersicht

2.1.1. Steuerwerk

Das Steuerwerk ist dafür zuständig, die einzelnen Zustände, die bei der Ausführung jedes Befehls ausgeführt werden, zum richtigen Zeitpunkt zu aktivieren. Dabei können einzelne Zustände je nach Befehl übersprungen werden.

Als Abstraktion kann man sich das Steuerwerk auch als Zustandsautomat vorstellen. Mit Hilfe von Abbildung 2.2 soll die Funktionsweise des Steuerwerkes durch ein Zustandsdiagramm dargestellt werden.

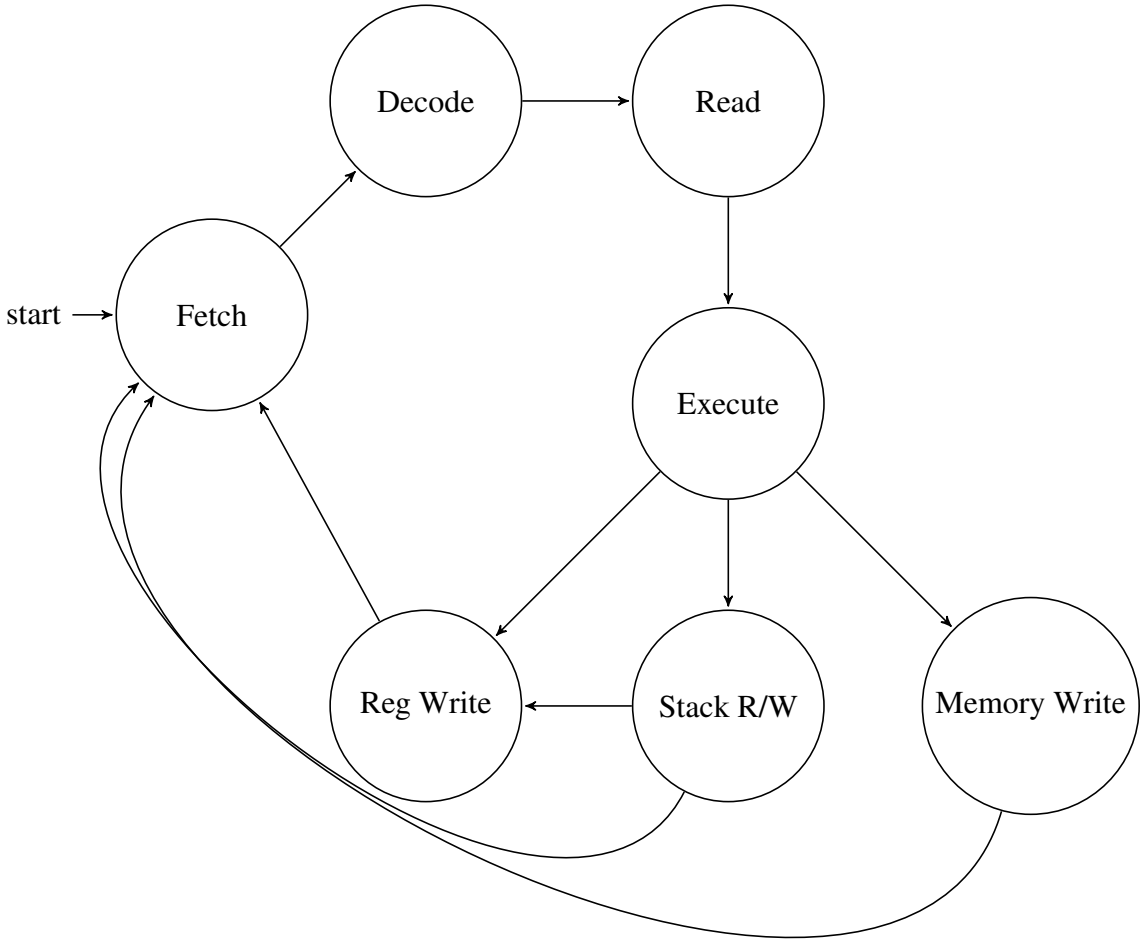


Abbildung 2.2.: Zustandsdiagramm des Steuerwerks

Die einzelnen Zustände des Steuerwerks werden im Folgenden beschrieben.

Fetch

Der zunächst auszuführende Befehl wird aus dem Hauptspeicher in den Prozessor geladen. Die Adresse des Befehls wird durch den Wert des Programmzählers (siehe 2.1.5) bestimmt.

Decode

In diesem Zustand wird das erste Byte des Befehl dekodiert und dabei festgestellt, ob der Befehl noch mehr Bytes, als bisher gelesen wurde, besitzt. Ist dies der Fall, wird wieder zum 'Fetch'-Zustand gewechselt. Außerdem wird der eigentlich auszufhrende Befehl festgestellt, mit welchen Parametern er auszuführen ist, aus welchen Registern die Daten herkommen und wo das Resultat gespeichert werden soll. Eine ausführlichere Beschreibung ist bei der Beschreibung des Dekodiers in Abschnitt 2.1.3 zu finden.

Read

Die Register für die Ein- und Ausgabe des Befehls werden nun ausgewählt und haben dementsprechend genug Zeit, ihre Daten an den Ausgängen auszugeben, ohne dass korrupte Daten verwendet werden.

Execute

Das Rechenwerk (siehe 2.1.2) wird bei diesem Zustand aktiviert und führt den vorher dekodierten Befehl aus. Das Resultat des Rechenwerks wird ausgegeben, damit in den weiteren Zuständen das Ergebnis an die gewünschte Stelle geschrieben werden kann.

Memory Write

Das Resultat des Rechenwerks wird an die vom Befehl vorgebene Speicheradresse im Arbeisspeicher (siehe 2.1.7) gespeichert.

Stack Read/Write

Je nach Befehl wird entweder das Resultat des Rechenwerks auf dem Stack gelegt oder der oberste Wert des Stacks genommen, um es im nächsten Zustand weier verarbeiten zu können. Eine detaillierte Erklärung der Funktionsweise des Stacks findet in Abschnitt 2.1.5 statt.

Register Write

Ähnlich wie beim 'Stack Read/Write'-Zustand, wird auch hier je nach Befehl das Ergebnis des Rechenwerks oder der Wert, der vom Stack entnommen wurde, in das Register geschrieben, welches der Befehl vorschreibt.

2.1.2. Rechenwerk

Das Rechenwerk ist dafür zustandig, dass die eigentliche Operation des Befehls ausgeführt wird. Es ist zu unterscheiden, ob man von der Arithmetic logic unit (ALU) oder vom Rechenwerk spricht, denn die ALU ist nur das 'Rechenzentrum' des Rechenwerks, gestützt von z.B. Hilfsregistern.

Der Befehlscode (auch Opcode) besteht aus 5 Bits und daher können $2^5 = 32$ verschiedene Befehle ausgeführt werden. Mit Hilfe der Tabelle 2.2 soll ein Überblick über die verfügbaren Befehle vermittelt werden. Um den Aufbau simpel zu halten, wurde entschieden, dass jeder Befehl die gleiche Bearbeitungszeit bekommt und damit jeder Befehl gleich schnell (bezogen nur auf die Zeit beim Rechenwerk) ausgeführt wird.

Das Rechenwerk ist auch zuständig, bestimmte Statusflaggen wie z.B. 'SB' zu setzen oder zu löschen, welche angibt, ob ein Sprungbefehl ausgeführt wurde und dieser auch die Sprungbedingung erfüllt, da auch der Programmzähler in diesem Fall einen neuen Wert bekommt,

2.1.3. Dekodierer

Der Dekodierer übernimmt die Aufgabe, den Befehl zu dekodieren, damit die Operation für das Rechenwerk ausgewählt werden kann und die dafür benötigten Register angesprochen werden. Da der Dekodierer einen einfachen Aufbau haben sollte, wurde der Aufbau eines Befehls selbst einfach gehalten.

Der Dekodierer ist sehr stark an den Befehlssatz angelehnt, welcher in Abschnitt 2.2 genauer beschrieben wird.

2.1.4. Programmzähler

Der Programmzähler (engl. Program Counter (PC)) ist hardwaretechnisch gesehen ein Register, welches aber im Vergleich zu 'normalen' Registern ihren Wert um eine bestimmte Byteanzahl je nach Länge des ausgeführten Befehls erhöht. Somit hält der PC immer die Adresse für den nächsten auszuführenden Befehl. Bei einem Sprungbefehl wird der Wert des Programmzählers mit dem vom Befehl vorgeschriebenen Wert überschrieben.

Im Ganzen lässt sich der Programmzähler als Zustandsautomat realisieren. Dazu wurden vier verschiedene Zustände für den PC gewählt:

NOP Keine Operation

INC Erhöhe den Zähler um die Byteanzahl des vorherigen Befehls

ASSIGN Setze den Zähler auf die Sprungadresse

RESET Setze den Zähler auf null zurück

Je nach Befehl wird bei der Ausführung dann der passende Zustand ausgewählt und somit sichergestellt, dass die Adresse für den folgenden Befehl korrekt ist.

2.1.5. Stack

Der Stack(deutsch: Stapel) ist ein Modell zum Speichern von Daten. Dabei sind zwei Operationen zum Speichern und Abrufen der Daten vorhanden: PUSH und POP. Bei PUSH wird oben auf den Stapel ein neues Element hinzugefügt, bei POP wird das oberste Element vom Stapel heruntergenommen. Somit ist der Zugriff auf das oberste Element beschränkt. Der Stack wird hauptsächlich von Programmen als Zwischenspeicher genutzt, wenn nicht genügend Register zur Verfügung sthen, aber auch beim Aufrufen von Funktionen (Befehl CALL) wird auf dem Stack die Rückkehraddresse gespeichert, welche beim Verlassen der Funktion (Befehl RET) wieder vom Stapel entnommen wird.

Besonders für den Funktionsaufruf eignet sich das Prinzip des Stacks sehr gut, da es keine Begrenzung (außer der physischen Größe des Stacks) gibt, wie viele Funktionen innerhalb einer

Funktionen aufgerufen werden können. Es sind also theoretisch unendlich viele Verkettungen möglich, wobei die Theorie nur durch die Größe des Speichers limitiert wird.

2.1.6. UART

Der Universal Asynchronous Receiver Transmitter (UART), welcher die Implementierung der seriellen Schnittstelle enthält, wird genutzt, um Eingabe von dem Benutzer zu bekommen. Dabei wird über ein selbst geschriebenes Skript die Tastatureingabe von einem anderen Rechner übernommen und über die serielle Schnittstelle an das Gerät weitergeleitet. So ist es möglich, eine Tastatur 'anzuschließen'. Die Übertragung über die serielle Schnittstelle findet mit einer Bitrate von 9600 bit/s, 8 Datenbits, 1 Stopbit und keinem Paritätsbit statt.

Hardwaretechnisch ist der UART über die Speicheradresse 0xEF00 zu erreichen. Sollte diese Adresse gelesen werden, wird das nächste Byte des UARTs abgefragt und dadurch die CPU temporär gestoppt, bis ein Byte angekommen ist. Dadurch ist die Eingabe blockierend, also können keine weiteren Operationen während der Eingabe durchgeführt werden.

2.1.7. Speichercontroller

Der Arbeitsspeicher ist als kontinuierlicher Speicherblock realisert worden. Es sind zwei Ports für die Ein- und Ausgabe zur Verfügung, wobei der zweite Port ausschließlich für den Videocontroller vorgesehen ist, welcher damit Zugriff auf den Videospeicher bekommt. Dies bedeutet auch, dass der Videospeicher nicht seperat geändert werden muss und so über allgemein gültige Lese- und Schreibbefehle angesprochen werden kann.

Insgesamt wurde der mit 16 Bit verfügbare maximal adressierbare Speicher von $2^{16} = 65536$ Bytes ausgenutzt, wobei einzelne Abschnitte, wie beispielsweise der Videospeicher oder auch die Speicheraddresse für den UART, nicht allgemein nutzbar sind.

2.1.8. Videocontroller

Der Videocontroller selbst ist nach außen hin unabhängig von allen anderen Elementen der EPU, nur der Videospeicher, welche die Informationen über die anzuzeigenden Pixel enthält, wird vom Speichercontroller geliefert. Dennoch hat der Speichercontroller keine Kontrolle über den Zugang des Videcontrollers zu dem Videospeicher, denn wie bereits in 2.1.7 gesagt, ist der Port für den Videocontroller unabhängig vom Speichercontroller.

2.1.9. Registerbelegung

Die EPU besitzt 16 Register, welche durch Selektion von $\log_2(16) = 4$ Adressbits angesprochen werden. Mithilfe der Tabelle 2.1 soll eine Übersicht aller Register dargestellt werden.

Selektion	Name	Zweck
0000	R0	Akkumulator
0001	R1	Allgemeine Verwendung
0010	R2	Allgemeine Verwendung
0011	R3	Allgemeine Verwendung
0100	R4	Allgemeine Verwendung
0101	R5	Allgemeine Verwendung
0110	R6	Allgemeine Verwendung
0111	R7	Allgemeine Verwendung
1000	R8	Allgemeine Verwendung
1001	R9	Allgemeine Verwendung
1010	R10	Allgemeine Verwendung
1011	R11	Allgemeine Verwendung
1100	R12	Allgemeine Verwendung
1101	R13	Allgemeine Verwendung
1110	R14	Flagregister
1111	R15	Programmzähler (Kopie)

Tabelle 2.1.: Registerbelegung

2.2. Befehlssatz

Der Befehlssatz beschreibt den Aufbau und die Menge aller Befehle der EPU. Nachfolgend soll sowohl der Aufbau und dessen Entstehung erläutert werden.

2.2.1. Aufbau

Der Befehl wird byteweise gelesen und die Anzahl an Bytes in einem Befehl sind durch die beiden letzten Bits des ersten Bytes angegeben, was es leichter macht, die richtige Byteanzahl einzulesen. Alle weiteren Abschnitte jedes Bytes werden je nach Opcode (die ersten 5 Bits) bestimmt, also die Register, die Rechenoperation, ob das Ergebnis auf dem Arbeitsspeicher geschrieben werden soll, etc. Wie in Abbildung 2.3 zu sehen, ist die Angabe von Konstanten in 4, 8 und 16 Bit möglich. Der Grund für die doch sehr platznehmende 16-Bit-Konstante ist, dass ein Register mit einer Konstante in einem Befehl befüllt werden soll, wobei alle Register der EPU 16 Bit groß sind.

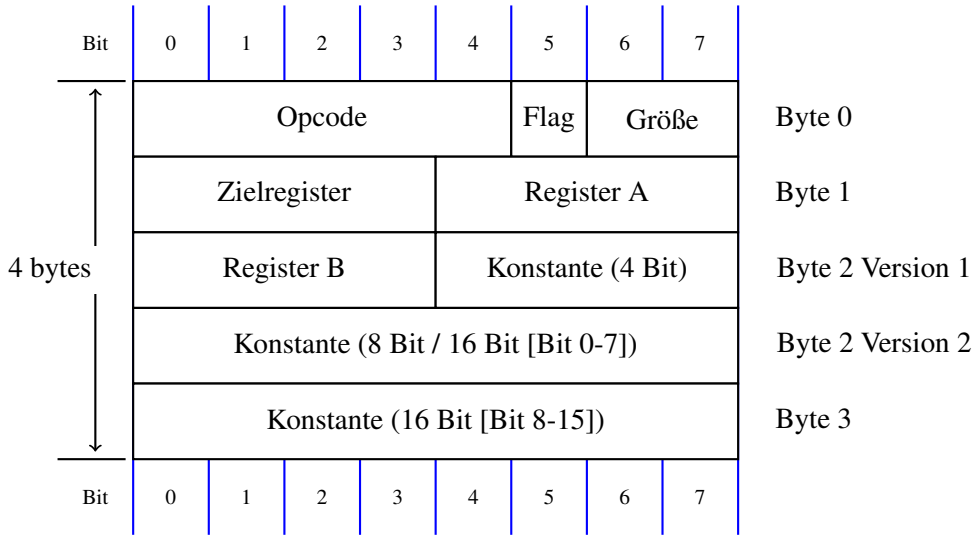


Abbildung 2.3.: Befehlsaufbau

2.2.2. Befehlsübersicht

Die folgende Tabelle 2.2 soll einen Überblick über die verschiedenen Befehle der EPU liefern. Durch die Größe des Opcodes (5 Bit) sind insgesamt $2^5 = 32$ Befehle möglich. Dennoch kann ein Befehl durch bestimmte Parameter (wie z.B. der Flag im ersten Byte) um gewisse Funktionen erweitert werden. Dies ermöglicht beispielsweise vorzeichenbehaftete und vorzeichenlose Addition, obwohl der Opcode beider Operationen der gleiche ist. Eine ausführlichere Beschreibung jedes Befehls ist im Anhang A zu finden.

Beschreibung	Kurzform	Opcode
Keine Operation	NOP	00000
Addition	ADD	00001
Subtraktion	SUB	00010
Logisches UND	AND	00011
Logisches ODER	OR	00100
Logisches XOR	XOR	00101
Logisches NOT	NOT	00110
Laden eines Registers	LOAD	00111
Verschieben eines Registerwertes	MOV	01000
Lesevorgang im Arbeisspeicher	READ	01001
Schreibvorgang im Arbeisspeicher	WRITE	01010
Bitweises Verschieben nach links	SHL	01011
Bitweises Verschieben nach rechts	SHR	01100
Vergleichen zweier Register	CMP	01101
Springen zu einer Adresse	JMP	01110
Bedingtes Springen zu einer Adresse	JC	01111
Nicht benutzt	—	10000
Aufrufen einer Funktion	CALL	10001
Zurückkehren von einer Funktion	RET	10010
Ein Element auf den Stack schieben	PUSH	10011
Von dem Stack ein Element entnehmen	POP	10100
Nicht benutzt	—	10101
Flags für die Beziehung zweier Register setzen	TEST	10110
Nicht benutzt	—	10111
Nicht benutzt	—	11000
Stoppen der Ausführung des Prozessors	HLT	11001
Setzen eines bestimmten Bits eines Registers	SET	11010
Löschen eines bestimmten Bits eines Registers	CLR	11011
Nicht benutzt	—	11100
Nicht benutzt	—	11101
Nicht benutzt	—	11110
Nicht benutzt	—	11111

Tabelle 2.2.: Befehlsliste

3. Implementierung in VHDL

3.1. Einführung in VHDL

In diesem Abschnitt soll eine kurze Einführung in VHDL gegeben werden, damit die nachfolgende Erklärung der Implementierung leichter zu verstehen ist.

3.1.1. Was ist VHDL

VHDL ist eine sogenannte Hardware Description Language (HDL), mit welcher es möglich ist, digitale Schaltungen in Form von Quellcode zu schreiben, um diesen dann von einem Computer zu Hardware weiterverarbeiten zu lassen. VHDL ist eine Abkürzung für VHSIC HDL (**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage). Der geschriebene VHDL-Code wird dann synthesiert zu einer Netzliste, welche die einzelnen Verbindungen zwischen den Komponenten beschreibt. Diese Netzliste wird daraufhin für den jeweiligen FPGA übersetzt, sodass als letzter Schritt eine Konfigurationsdatei erstellt werden kann, welche auf den FPGA geladen werden kann.

Eine Besonderheit bei VHDL ist, dass nicht jeder geschriebene Quellcode synthesierbar ist, was bedeutet, dass mancher Quellcode nur in der Simulation funktioniert. In dem Umfang dieses Projektes ist aber der geschriebene Quellcode vollkommen synthesierbar und soll nicht nur in der Simulation funktionieren.

3.1.2. Notation

Allgemein endet jede Anweisung immer mit einem Semikolon. Kommentare werden mit einem doppelten Bindestrich (--) eingeleitet und sind gültig bis zum Ende der Zeile. Einzelne Bitwerte werden mit einfachen Anführungszeichen und Werte von Bitvektoren (mehreren Bits) mit doppelten Anführungszeichen umrahmt.

Außerdem gibt es noch mehrere Sprachelemente, welche häufig Verwendung haben. Eine (nicht vollständige) Beschreibung von häufig genutzten Sprachelementen:

signal Verbindung zweier Module. Beschreibt die Leitung zwischen den Modulen.

variable Ein Zwischenspeicher für Werte, welcher **nicht** als Signal übersetzt wird.

entity Schlüsselwort zur Deklaration eines Moduls (logische Einheit).

in Deklaration eines Eingabesignals eines Moduls.

out Deklaration eines Ausgabesignals eines Moduls.

architecture Schlüsselwort zur Definition eines Moduls

component Signatur eines Moduls. Wird verwendet, um ein Modul innerhalb eines anderen Moduls zu verwenden.

package Eine Sammlung (Bibliothek) von **components**.

3.2. Beschreibung wichtiger Module

3.2.1. top – Verbindung der Ein- und Ausgänge

Das Topmodul dient als oberste Ebene, welche die EPU mit der ‘Außenwelt’ verbindet. Die dazugehörigen Ein- und Ausgangssignale werden in der Datei *top.vhdl* festgelegt.

Wie in Abbildung 3.1 zu erkennen, wird zuerst der Takt in Zeile drei als Eingang definiert, in den Zeilen vier bis acht die 7-Segment-Anzeige, die LEDs und Taster definiert. Die Zeilen zehn und elf beschreiben die beiden Signale des seriellen Schnittstelle und Zeilen dreizehn bis siebzehn beschreiben die Signale der VGA–Verbindung.

```
1  entity top is
2      port(
3          MainClk : in  std_logic;
4          RST     : in  std_logic;
5          SEGEn   : out  std_logic_vector (2 downto 0);
6          LED     : out  std_logic_vector (7 downto 0);
7          SEG     : out  std_logic_vector (7 downto 0);
8          SW      : in  std_logic_vector (6 downto 0);
9
10         TX      : out  std_logic;
11         RX      : in  std_logic;
12
13         hs      : out  std_logic;
14         vs      : out  std_logic;
15         red     : out  std_logic_vector (2 downto 0);
16         green   : out  std_logic_vector (2 downto 0);
17         blue    : out  std_logic_vector (1 downto 0)
18     );
19 end top;
```

Abbildung 3.1.: Topmodul

3.2.2. core – Topmodul der CPU

Das ‘core’-Modul verbindet die einzelnen Baugruppen der CPU. Dazu gehören Rechenwerk, Steuerwerk, Dekodierer, Programmzähler und der Stack, welche bereits in Kapitel 2 beschrieben wurden.

In Abbildung 3.2 ist die Definition des Moduls dargestellt. Die Signalnamen wurden dem Zweck angepasst, dass heißt das Verbindungen zum Speichercontroller mit ‘MEM’ und interne Signale mit ‘CORE’ bezeichnet. Das Signal ‘O_LED’ ist dabei eine Ausnahme, da dieser zu Testzwecken verwendet wird und damit nicht fest eingeordnet werden kann.

```
1  entity core is
2      port(
3          -- Eingaenge
4          I_CORE_Clk      : in  std_logic;
5          I_CORE_Reset    : in  std_logic;
6
7          I_MEM_Ready     : in  std_logic;
8          I_MEM_Data      : in  std_logic_vector(7 downto 0);
9
10         -- Ausgaenge
11         O_CORE_HLT      : out std_logic;
12
13         O_MEM_Reset     : out std_logic;
14         O_MEM_En        : out std_logic;
15         O_MEM_We        : out std_logic;
16         O_MEM_Data      : out std_logic_vector(7 downto 0);
17         O_MEM_Addr      : out std_logic_vector(15 downto 0);
18
19         O_LED           : out std_logic_vector(7 downto 0)
20     );
21 end core;
```

Abbildung 3.2.: ‘core’-Modul

3.2.3. memory_control – Speichercontroller

Der Speichercontroller wurde bereits in Abschnitt 2.1.7 beschrieben und hier soll nur die Implementation dessen dargestellt werden.

Wie auch bei dem ‘core’-Modul (siehe 3.2.2) sind hier die Signale nach Zugehörigkeit bezeichnet und interne Signale mit ‘MEM’ und Videosignale mit ‘VID’ beschrieben. Außerdem sind die letzten drei Signale (Zeilen 22–24) für die serielle Schnittstelle vorgesehen, welches im Speicher abgebildet wird.

```
1  entity memory_control is
2      port (
3          -- Eingänge
4          I_MEM_Clk       : in  std_logic ;
5          I_MEM_Reset     : in  std_logic ;
6          I_MEM_En        : in  std_logic ;
7          I_MEM_We        : in  std_logic ;
8          I_MEM_Data      : in  std_logic_vector (7 downto 0);
9          I_MEM_Addr      : in  std_logic_vector (15 downto 0);
10         I_VID_Clk       : in  std_logic ;
11
12         -- Ausgänge
13         O_MEM_Ready     : out std_logic ;
14         O_MEM_Data      : out std_logic_vector (7 downto 0);
15         O_LED           : out std_logic_vector (7 downto 0);
16         O_VID_Red       : out std_logic ;
17         O_VID_Green     : out std_logic ;
18         O_VID_Blue      : out std_logic ;
19         O_VID_HSync     : out std_logic ;
20         O_VID_VSync     : out std_logic ;
21
22         UClk            : in  std_logic ;
23         TX              : out std_logic ;
24         RX              : in  std_logic
25     );
26 end memory_control;
```

Abbildung 3.3.: Speichercontroller

3.2.4. alu – Rechenwerk

Das Rechenwerk führt die eigentliche logische bzw. mathematische Operation aus. Hier soll zum Verständnis nur ein Teil des Quellcodes gezeigt werden.

Die nachfolgende Abbildung 3.4 beschreibt den Ablauf der ALU bei dem Befehl 'ADD'.

```
1                   when OPCODE_ADD =>
2                       if I_AluOp(IFO_REL_FLAG) = '0' then
3                           if I_AluOp(IFO_REL_LENGTH_END) = '0' then
4                               -- Nicht vorzeichenbehaftet + Form RRR
5                               S_Res(16 downto 0) <= std_logic_vector (unsigned('0' & I_DataA)
6                                                   + unsigned('0' & I_DataB));
7                           else
8                               -- Nicht vorzeichenbehaftet + Form RRI
9                               S_Res(16 downto 0) <= std_logic_vector (unsigned('0' & I_DataA)
10                                                  + unsigned('0' & I_Imm));
11                          end if;
12                      else
13                          if I_AluOp(IFO_REL_LENGTH_END) = '0' then
14                              -- Vorzeichenbehaftet + Form RRR
15                              S_Res(15 downto 0) <= std_logic_vector (signed(I_DataA) + signed(I_DataB
    ));
16                              S_Res(17) <= (I_DataA(15) and I_DataB(15) and not S_Res(15)) or
17                                           (not I_DataA(15) and not I_DataB(15) and S_Res(15));
18                          else
19                              -- Vorzeichenbehaftet + Form RRI
20                              S_Res(15 downto 0) <= std_logic_vector (signed(I_DataA) + signed(I_Imm))
    ;
21                              S_Res(17) <= (I_DataA(15) and I_Imm(15) and not S_Res(15)) or
22                                           (not I_DataA(15) and not I_Imm(15) and S_Res(15));
23                          end if;
24                      end if;
25                      S_SB <= '0';
```

Abbildung 3.4.: Addition aus Sicht der ALU

Der Quellcode ist recht einfach zu lesen, denn die eigentliche Logik, die hier passiert, basiert auf Abfragen nach bestimmten Flags. So gibt es die Möglichkeit beim 'ADD'-Befehl einerseits vorzeichenbehaftet und andererseits vorzeichenunbehaftet zu addieren (siehe Abfrage Zeile 2). Danach wird noch bei beiden Fällen noch überprüft, ob zwei Register oder ein Register und eine Konstante addiert werden sollen (siehe Zeilen 3 und 13). Die je nach Fall ausgeführte Addition ist dann trivial, da sie bereits in die Sprache eingebunden ist und nur die Werte in die richtigen Datentypen (*signed* & *unsigned*) konvertiert werden müssen (siehe Zeilen 5-6, 9-10, 15 und 20). Bei der vorzeichenbehafteten Addition muss die Erkennung eines **Überlaufs** seperat durchgeführt werden (siehe Zeilen 16-17 und 21-22), wohingegen bei der vorzeichenunbehafteten Addition die Erkennung einfach am siebzehnten Bit zu erkennen ist und damit innerhalb der eigentlichen Rechnung stattfindet.

4. Programmstruktur

In diesem Kapitel soll eine Übersicht über die Struktur und den Aufbau der Programme, welche die Software der EPU bilden, dargestellt werden. Die Voraussetzung für das Verständnis der Umsetzung der Software in die Hardware ist bereits in Kapitel 2, oder genauer bei der Erklärung des Befehlssatzes in Abschnitt 2.2, geschehen.

4.1. Einführung in EPU-Assembly

Zur Übersetzung der eigenen Assembly-Sprache in Maschinencode, welche die CPU verstehen, wurde ein eigener Assembler entwickelt. Der Assembler ist in der Interpretersprache Python geschrieben und erlaubt die Kompilierung von einer .easm-Datei, wobei eine Vorlage mit vordefinierten Funktionen als externe Datei auch noch übergeben werden kann. Die Ausgabe erfolgt durch das Schreiben der einzelnen Bytes des Maschinencodes in eine Datei im .coe-Format. Der Grund für das .coe-Format ist, dass dieses genutzt werden kann, um den RAM-Block der EPU zu initialisieren und somit das Programm direkt ausgeführt werden kann.

Der Aufbau der Befehle, welche der Assembler versteht, wurde recht simpel gehalten, um eine leicht zu verstehende Sprache zu erstellen, wobei viele Ähnlichkeiten zur x86-Assembly (genutzt ist den meisten Desktopcomputern heutzutage) bestehen, da somit das anfängliche Verstehen der Befehle leichter wird. Der Aufbau eines Befehls richtet sich nach folgender Regel, wobei in eckigen Klammern eingerahmte Parameter optional sind bzw. nur bei bestimmten Befehlen notwendig sind:

Mnemonic[.Option] [Operand1] [,Operand2] [,Operand3]

Um einen Befehl aufzurufen, wird das obere Schema verwendet, was bedeutet, dass der Befehl selbst mit seiner **Mnemonic** startet. Das Wort Mnemonic bedeutet auf Deutsch so viel wie 'Merkspruch'. Es wird meist bei Assemblersprachen verwendet, um beim Programmieren das Auswendiglernen des Maschinencodes jedes Befehls zu sparen. An die Mnemonic kann eine Option nach einem Punkt angehängt werden. Dies ist befehlsspezifisch und muss für jeden Befehl nachgeschlagen werden. Je nach Befehl werden auch ein bis zwei weitere Operanden benötigt, welche selbst noch in verschiedenen Formen vorkommen können (abhängig von der gewählten Option). Um dies zu verdeutlichen, sollen nun ein paar Beispiele das Verständnis für den Befehlsaufbau gezeigt werden.

```
1     load r1, 0xAFFE
2     jmp.i $test
3
4     data:
5         .data 0xFF
6
7     test:
8         addi r2, r1, 0d10
9         load r1, $data
10        write.l r1, r2, 0x0
```

Abbildung 4.1.: Befehlsaufbau - Assembly

In Abbildung 4.1 wird zuerst in Zeile 1 eine hexadezimale Konstante (Präfix '0x' steht für hexadezimal) in das Register R1 geladen. Danach wird ein Sprungbefehl mit der Option 'i' ausgeführt, wobei das 'i' für 'Immediate' bzw. Konstante steht und damit die Adresse, zu welcher gesprungen werden soll, im Befehl vorhanden ist. Die Konstant ist hier mit dem Adressoperator '$' vorangestellt, welcher die Adresse des **Labels**, hier test' (Zeile 7), enthält. Dadurch wird die Ausführung in Zeile 8, also direkt nach der Definition des Labels, fortgeführt.

Davor ist aber noch ein in Zeile 5 eine sogennante **Assemblerdirektive** zu sehen, welche an dem '.' vor der Mnemonic zu erkennen ist. Assemblerdirektiven sind Anweisungen an den Assembler und werden nicht direkt in Maschinencode umgewandelt, sondern getrennt behandelt. In diesem Beispiel ist die Assemblerdirektive 'data' verwendet worden, welche die angegebene Konstante im ersten Operanden unmittelbar in die Byteausgabe des Maschinencodes übernimmt. So ist es beispielsweise möglich, dass vordefinierte Variablen bereits einen Wert beim Start des Programms zugewiesen bekommen können.

Zurück bei der Ausführung des Beispieles in Zeile 8 wird mit dem Befehl 'addi' eine Konstante zu dem Register R1 (zweiter Operand) eine dezimale Konstante (Präfix '0d' für dezimale Konstanten) addiert. Das Ergebnis wird im ersten Operand gespeichert - hier R2. In Zeile 9 wird die Adresse des 'data'-Labels in das Register R1 geladen und danach in Zeile 10 wird an die Speicheradresse in R1 der Wert im Register R2 geschrieben. Der dritte Operand wird hier nicht benötigt und daher null gesetzt, kann aber für einen Offset der Speicheradresse dienen.

4.2. Funktionsaufruf

In diesem Abschnitt soll der Aufruf von Funktonen demonstriert werden. Bei dem Aufbau wurde eine ähnliche Methode wie beim x86-Befehlssatz verwendet. Die einfachste Methode des Funktionsaufrufs ist das Aufrufen einer Funktion ohne Parameter, wie unten in Abbildung 4.2 zu sehen ist. Dabei wird einfach der Befehl 'call' aufgerufen mit dem ersten Operan-

```
1 funktion :
2       ret
3
4 _start :
5       call . i  $funktion
```

Abbildung 4.2.: Funktionsaufruf ohne Parameter

den hier als Adresse zum Label 'funktion'. Nachdem die Funktion mit dem Befehl 'ret' wieder zurückkehrt, wird automatisch der nächste Befehl nach dem Funktonsaufruf ausgeführt.

Damit beim Zurückkehren von einer Funktion die Rücksprungadresse für den nächsten Befehl gefunden werden kann, wird beim Aufruf von 'call' nicht nur ein Sprungbefehl zur Funktion aufgerufen, sondern auch die Rückkehradresse auf den Stack (siehe 2.1.5) gelegt. Beim Ausführen von 'ret' wird dann einfach das Element vom Stack heruntergenommen und in den PC (siehe 2.1.4) geschrieben.

Bei Funktionen mit Parametern und Rückgabewert ist der Aufbau ähnlich. Das folgende Beispiel in Abbildung 4.3 soll einen solchen Funktionsaufruf illustrieren:

Zuerst werden die zu übergebenen Argumente in beliebige Register gespeichert (Zeile 15-16) und dann in **umgekehrter** Reihenfolge auf den Stack gelegt (Zeile 17-18). Danach findet der eigentliche Funktionsaufruf statt und die Funktion wird aufgerufen, wobei unter anderem die Rückkehradresse wird auch auf dem Stack abgelegt. Da die Rückkehradresse ganz oben auf dem Stack abgelegt ist, muss diese erst vom Stack heruntergenommen und in einem Register zwischengespeichert werden, damit die Argumente vom Stack geholt werden können (Zeile 2). Da sie in umgekehrter Reihenfolge auf den Stack gelegt wurden, dreht sich beim Holen der Argumente die Reihenfolge wieder um, sodass die Reihenfolge wieder richtig ist (Zeile 3-4). Danach kann die Rückkehradresse wieder zurück auf den Stack gelegt werden (Zeile 5). Danach kann die eigentliche Operation der Funktion durchgeführt werden; dies ist hier die Subtraktion (Zeile 7). Da nun der Rückgabewert auf dem Stack abgelegt werden soll, aber auch zugriffsbereit für die aufrufende Funktion, muss zuerst wieder die Rückkehradresse vom Stack entfernt (Zeile 9) und dann der Rückgabewert auf den Stack gelegt werden (Zeile 10). Danach kann die Rückkehradresse wieder auf den Stack (Zeile 11) und die Funktion kann wie gewohnt zurückkehren (Zeile 12).

Durch die Nutzung des Stacks für den Rückgabewert ist es theoretisch möglich, mehrere Rückgabewert einer Funktion zuzuweisen. In dem Beispiel ist dies aber nicht der Fall. Den-

```
1  sub:
2      pop r15
3      pop r0
4      pop r1
5      push r15
6
7      sub.u r0, r0, r1
8
9      pop r15
10     push r0
11     push r15
12     ret
13
14 _start :
15     load r4, 0x0005
16     load r5, 0x0002
17     push r5
18     push r4
19     call.i $sub
20     pop r3
```

Abbildung 4.3.: Funktionsaufruf mit Paramtern

noch ist zu erwähnen, dass über die Register auch ein oder mehrere Rückgabewerte theoretisch erfolgen können. Genauere Informationen sind der jeweiligen Funktion, welche man aufrufen möchte, zu entnehmen.

4.3. Vordefinierte Funktonen

Einige allgemein nützige Funktionen sind bereits vorgeschrieben worden und können als Vorlage an den Assembler übergeben werden. Dadurch können diese vordefinierten Funktionen in allen Programmen genutzt werden. Unter anderem sind folgende Funktionen verfügbar:

_print(*data, start) Gebe eine null-terminierte Zeichenkette an der Stelle **data** auf dem Bildschirm an Startpositon **start** aus.

_setcursor(pos) Setze den Bildschirmcursor an Position **pos**, wobei der Wert 0xAABB die x-Koordinate 0xAA und die y-Koordinate 0xBB setzt.

_getinput Liest ein Byte vom Benutzer ein und gibt es zurück

_getstring Liest einen 'Enter'-terminierten String vom Benutzer ein und gibt diesen zurück

Die Benutzung dieser Vorlage wird empfohlen, ist aber keine Voraussetzung. Denn die Vorlage definiert auch, dass der Start des eigentlichen Programms beim Label '_start' stattfindet und dieser wird auch von der Vorlage aufgerufen. Wie wahrscheinlich bereits aufgefallen ist, starten bei der Erstellung der Vorlage alle genutzten Labels mit einem Unterstrich, was den Grund hat, alle vordefinierten Funktionen und die Funktionen des Programms visuell zu trennen. Daher ist es auch nicht empfohlen, dass im Programm Labels mit einem Unterstrich starten.

4.4. Debugging

Mithilfe des Debugging ist es möglich, Fehler in Programmen zu erkennen und diese auch zu beseitigen. Damit dies auch bei der EPU möglich ist, soll dieser Abschnitt beispielhaft anhand eines Signal-Zeit-Diagrammes zeigen, wie die Ausführung eines Befehls aussehen sollte. Das auszuführende Befehl sei folgender, wobei angenommen wird das Register R14 bereits den Wert 0x1234 hat:

```
push r14
```

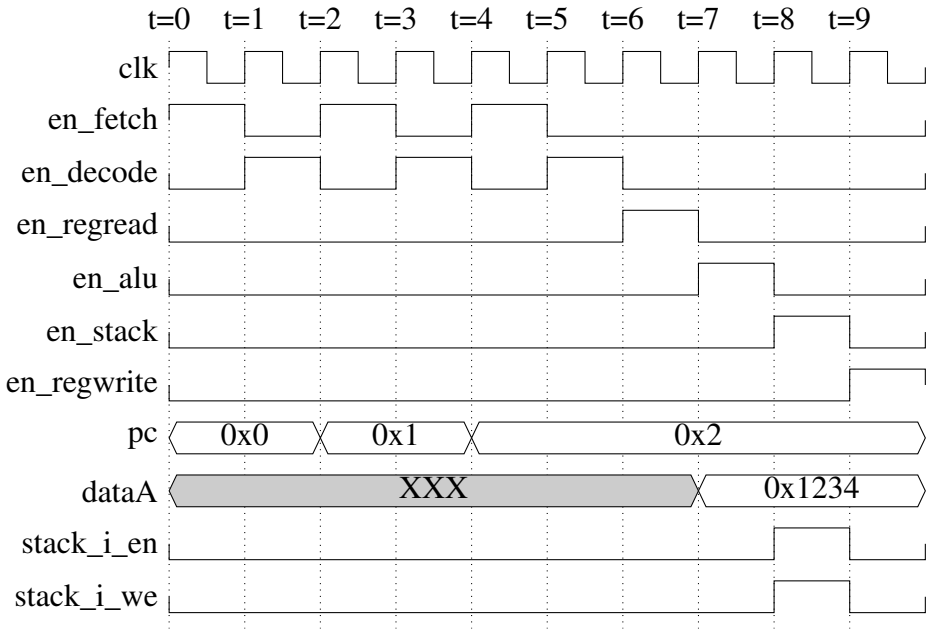


Abbildung 4.4.: Signal-Zeit-Diagramm

Wie in Abbildung 4.4 zu sehen, beschreiben die ersten 6 Signale die einzelnen Zustände des Steuerwerks (siehe 2.1.1), danach folgt der Programmzähler (siehe 2.1.4) und der Wert des übergebenen Registers Ra, gefolgt von dem 'Enable'- und 'WriteEnable'-Eingang des Stacks (siehe 2.1.5).

Im ersten Abschnitt von t=0 bis t=6 ist zu sehen, wie byteweise der Befehl gelesen und dekodiert wird. Dabei wird bei jeder positiven Flanke des Signals 'en_fetch' der Programmzähler inkrementiert, damit er auf das nächste zu dekodierende Byte zeigt. Im zweiten Abschnitt von t=6 bis t=9 werden zuerst die Werte der übergebenen Register gelesen und sind dann nach der negativen Flanke von 'en_regread' verfügbar (siehe Signal 'dataA'). Als nächstes wird das Rechenwerk aktiviert und legt in diesem Fall nur fest, dass kein Sprungbefehl ausgeführt werden muss.Danach wird nun 'en_stack' auf 1 gesetzt (t=8) und damit gleichzeit auch die 'Enable'- und 'Write-Enable'-Eingänge des Stacks. Der Stack nimmt daraufhin den Wert von 'dataA' und schreibt ihn als oberstes Element auf den Stapel. Im letzten Schritt (t=9) wird nun noch 'en_regwrite', da jedoch kein Ergebnis in ein Register geschrieben werden muss, passiert hier nichts.

5. Fazit

5.1. Umfang und Aufwand

Der Aufwand der besonderen Lernleistung war insgesamt deutlich höher als anfänglich gedacht. Der Einstieg in das Thema ging ziemlich schnell, sodass die Simulation des ersten Programm bereits ein Monat nach Beginn funktionierte.

Nachfolgend gab es kleinere Probleme, bei denen die Behebung durch die Simulation deutlich vereinfacht wurde. Als es dann zur ersten Übertragung des Projektes auf den FPGA kam, trat ein größeres Problem auf. Zuerst konnte die Datei für den FPGA nicht übertagen werden, da der Hersteller nur ein Programm für Windows zur Übertragung anbietete, aber der Rest des Projektes bereits vollständig auf Linux erstellt wurde. Nach etwas Recherche stellte sich dann aber raus, dass jemand bereits das selbe Problem für ein ähnliches Board des selben Herstellers hatte und seinen Code für ein Programm zur Übertragung veröffentlicht hatte. Dies war eine große Hilfe, da nun nur noch die Beschreibung des Boards im Code angepasst werden musste und somit auch die Übertagung funktionierte.

Wenig später sollte auch der RAM-Block des FPGA genutzt werden, wobei vorher nur für die Simulation der Speicher aus Flip-Flops erstellt wurde. Dabei stellte sic das Problem, dass die Dokumentation des DDR-RAM-Chips auf dem Board nicht sehr ausführlich ist und keine große Hilfe bei der Implementierung bietet. Deshalb wurde kein DDR-RAM verwendet, sondern der interne Speicher des FPGA-Chip. Da nur 64 Kilobyte an Speicher für die 16-Bit-Adressierung der EPU nötig sind, reicht dieser interne Speicher vollkommen aus und vereinfacht damit auch den Aufbau der EPU, da kein komplexer DDR-Speichercontroller implementiert werden musste.

Auch wurde anfangs überlegt, alle Grundrechenarten als eigene Befehle zu implementieren. Nach langer Überlegung wurde aber dagegen entschieden und nur die Addition und Subtraktion in die Hardware implementiert, da die anderen Rechenarten über die Software später implementiert werden können und nicht essentiell sind. Grundsätzlich lohnte sich nicht der notwendige Zeitaufwand der Implementierung für die Multiplikation und Division, da diese Befehle nicht häufig genug Verwendung bekommen.

5.2. Ziele und Lernerfolg

Hauptziel der besonderen Lernleistung war es, einen funktionsfähigen Mikrorechner bzw. zu produzieren und dieser Ziel ist auch sehr gut gelungen. Außerdem sollte eine Ausgabe über einen Bildschirm als auch eine Eingabe über eine Tastatur möglich sein, was beides funktioniert.

Andere anfangs sehr übernommene Ziele, wie eine Netzwerkverbindung, Soundtreiber oder externer Speicher über den MicroSD-Kartenslot, mussten aus Zeitgründen leider gestrichen werden. Dennoch sind diese Ziele als mögliche Erweiterungen zu sehen, welche im Nachhinein noch implementiert werden könnten.

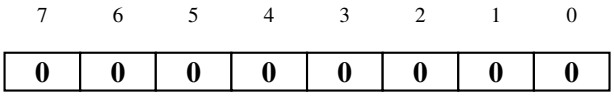
A. Befehlsliste

Erklärungen

Alle Befehle, die mit **.u** (engl. *unsigned*) enden, werden als vorzeichenunbehaftet betrachtet und alle Befehle, die mit **.s** (engl. *signed*) enden, werden als vorzeichenbehaftet betrachet.

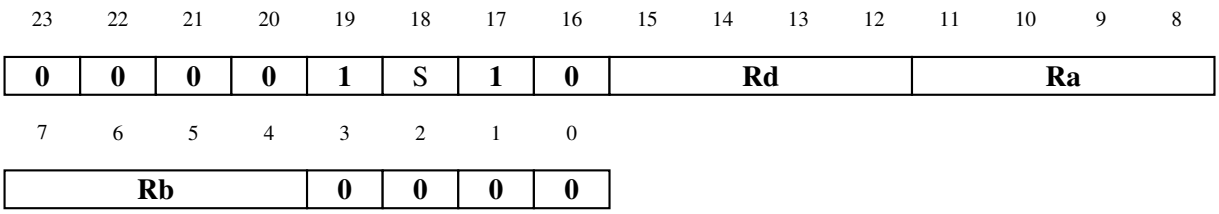
Wenn ein Befehl auf Vorzeichen achtet und dies nicht angegeben wird (also kein **.u** oder **.s**), ist der Befehl äquivalent zum vorzeichenunbehafteten Befehl, wenn es nicht explizit angegeben wird.

nop



Dieser Befehl führt keine Operation aus.

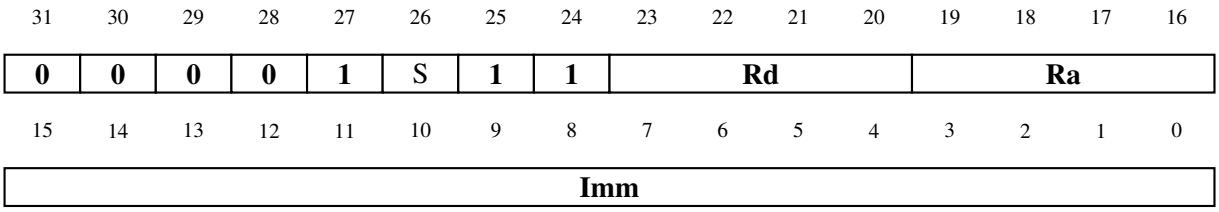
add.u, add.s



Addiere zwei Register miteinander. Das Bit **S** zeigt an, ob die Addition vorzeichenbehaftet(1) oder vorzeichenunbehaftet(0) stattfinden soll.

Regs[Rd] = Regs[Ra] + Regs[Rb]

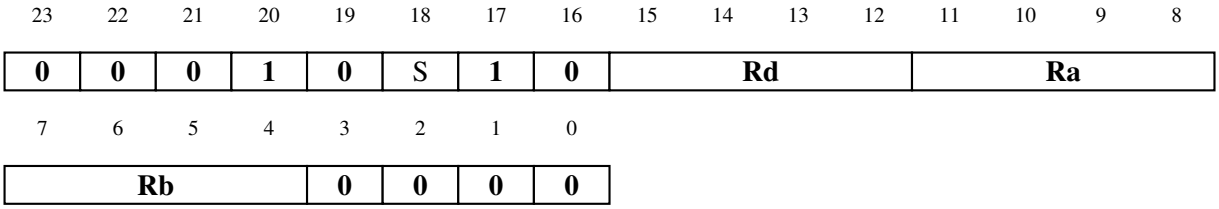
addi.u, addi.s



Addiere ein Register mit einer 16-Bit-Konstanten. Das Bit **S** zeigt an, ob die Addition vorzeichenbehaftet(1) oder vorzeichenunbehaftet(0) stattfinden soll.

Regs[Rd] = Regs[Ra] + Imm

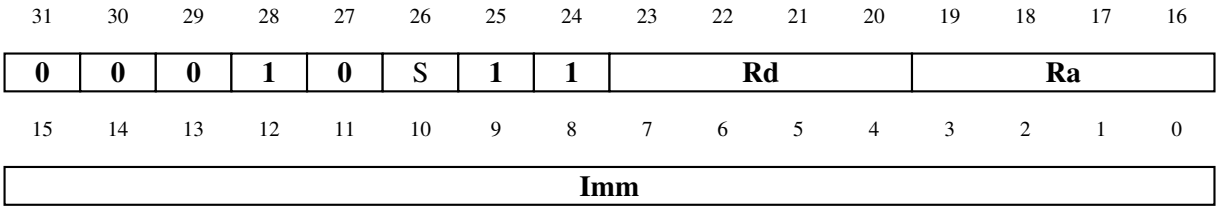
sub.u, sub.s



Subtrahiere die Werte zweier Register. Das Bit **S** zeigt an, ob die Subtraktion vorzeichenbehaftet(1) oder vorzeichenunbehaftet(0) stattfinden soll.

Regs[Rd] = Regs[Ra] - Regs[Rb]

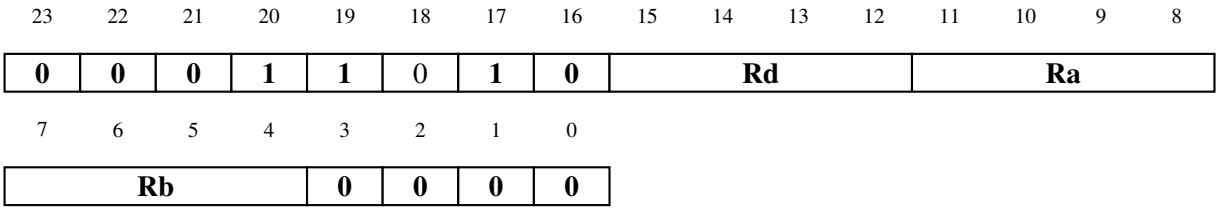
subi.u, subi.s



Subtrahiere eine 16-Bit-Konstante von einem Register. Das Bit **S** zeigt an, ob die Subtraktion vorzeichenbehaftet(1) oder vorzeichenunbehaftet(0) stattfinden soll.

Regs[Rd] = Regs[Ra] - Imm

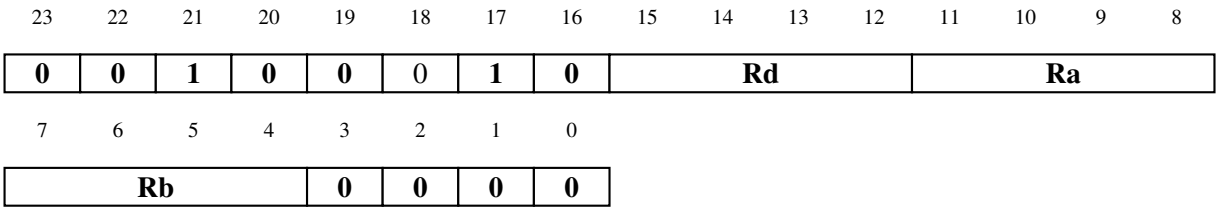
and



Berechne die bitweise UND-Verknüpfung zweier Register.

Regs[Rd] = Regs[Ra] $\wedge$ *Regs[Rb]*

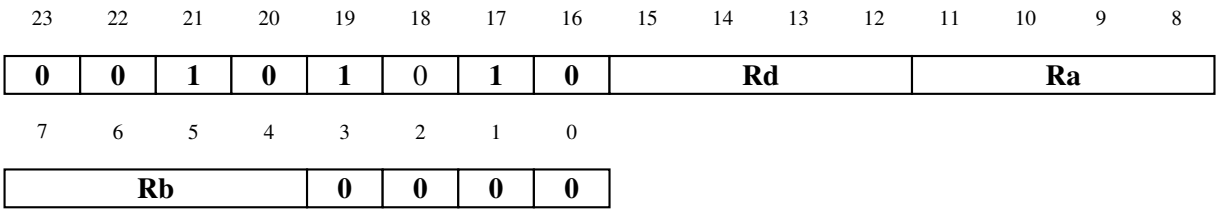
or



Berechne die bitweise ODER-Verknüpfung zweier Register.

Regs[Rd] = Regs[Ra] $\vee$ *Regs[Rb]*

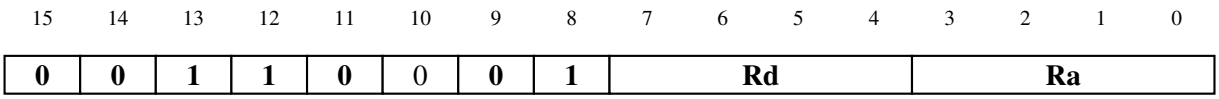
or



Berechne die bitweise XOR-Verknüpfung zweier Register.

Regs[Rd] = Regs[Ra] $\oplus$ *Regs[Rb]*

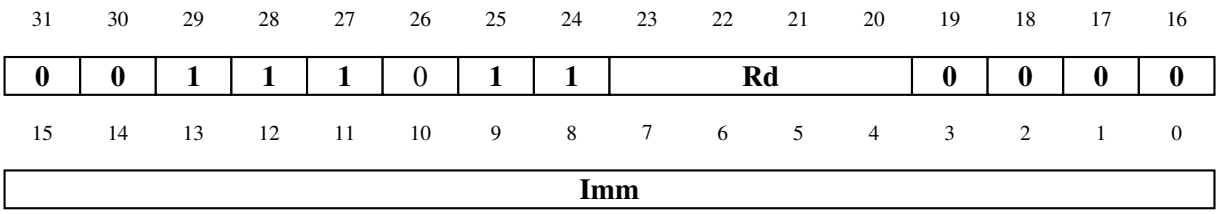
not



Berechne das logische NICHT eines Registers.

Regs[Rd] = ¬Regs[Ra]

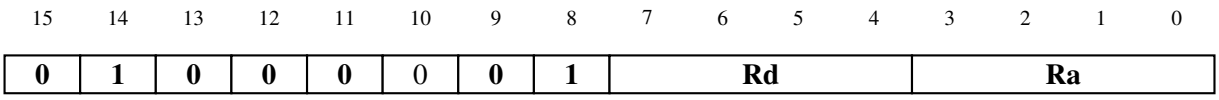
load



Lade eine Konstante in ein Register.

Regs[Rd] =Imm

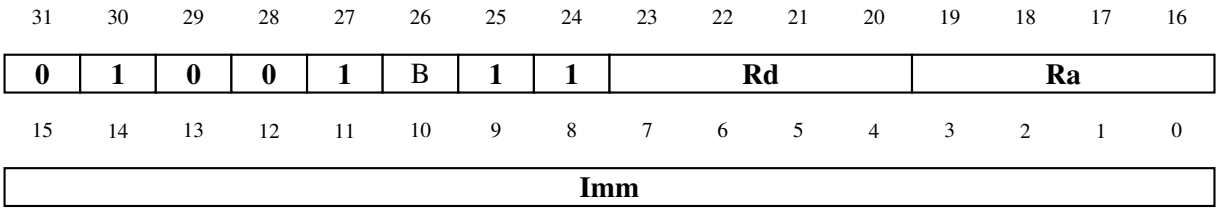
mov



Kopiere den Wert eines Registers in ein anderes Register.

Regs[Rd] = Regs[Ra]

read.l, read.h



Hole den Wert vom Speicher an der angegebenen Adresse in ein Register. Die Adresse ergibt sich aus der vorzeichenunbehafteten Addition von dem Register *Ra* und der Konstanten *Imm*.

Da der Speicher immer nur ein Byte adressiert, wird mit Hilfe des Bits **B** entschieden, ob der Wert in das untere Byte des Registers (**B=0** bzw. **read.l**) oder in das obere Byte des Registers (**B=1** bzw. **read.h**) geschrieben werden soll.

Regs[Rd] = Mem[Regs[Ra] + Imm]

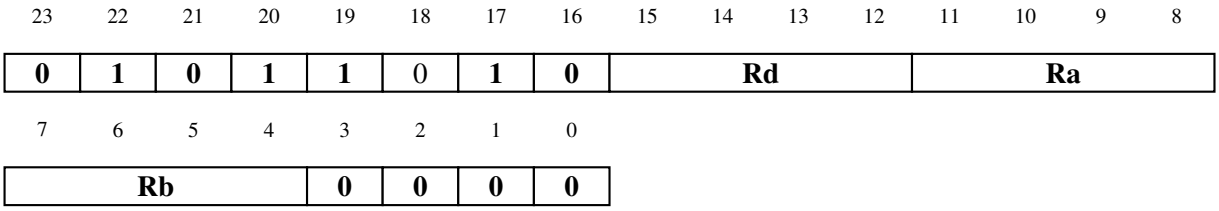
write.l, write.h

Der Wert in Register *Rb* wird an eine Adresse im Speicher geschrieben. Diese Adresse ergibt sich aus der vorzeichenunbehafteten Addition von dem Register *Ra* und der Konstanten *Imm*, welche aufgrund von Optimisierungen im Dekodierer (siehe 2.1.3) getrennt vorkommt.

Da eine Speicheradresse immer nur ein Byte adressiert, wird mit dem Bit **B** entschieden, ob das untere Byte (**B=0** bzw. **write.l**) oder das obere Byte (**B=1** bzw. **write.h**) des Registers *Rb* geschrieben wird.

Mem[Regs[Ra] + Imm] = Regs[Rb]

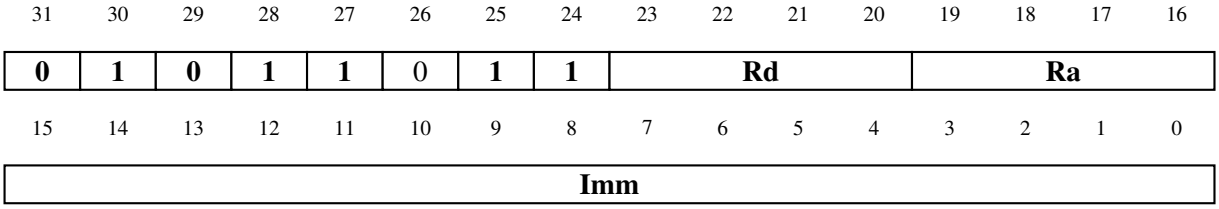
shl.r



Dieser Befehl führt eine logische Verschiebung nach links um einen bestimmten Wert aus. Dieser Wert wird aus dem Register *Rb* genommen.

Regs[Rd] = Regs[Ra] « Regs[Rb]

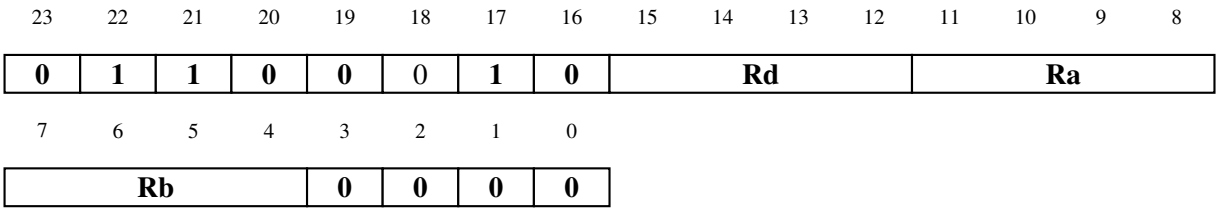
shl.i



Dieser Befehl führt eine logische Verschiebung nach links um einen bestimmten Wert aus. Dieser Wert wird von der Konstante *Imm* genommen.

Regs[Rd] = Regs[Ra] « Imm]

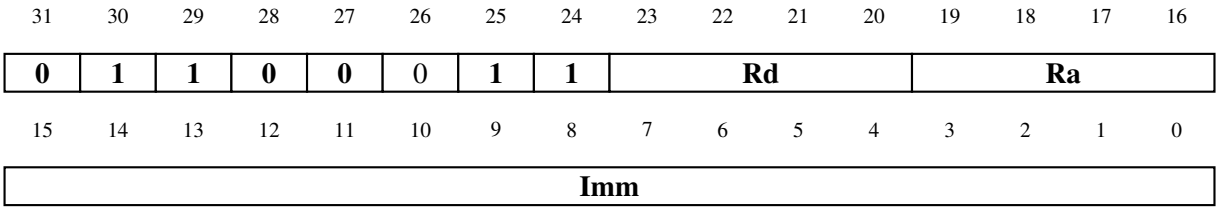
shr.r



Dieser Befehl führt eine logische Verschiebung nach rechts um einen bestimmten Wert aus. Dieser Wert wird aus dem Register *Rb* genommen.

Regs[Rd] = Regs[Ra] » Regs[Rb]

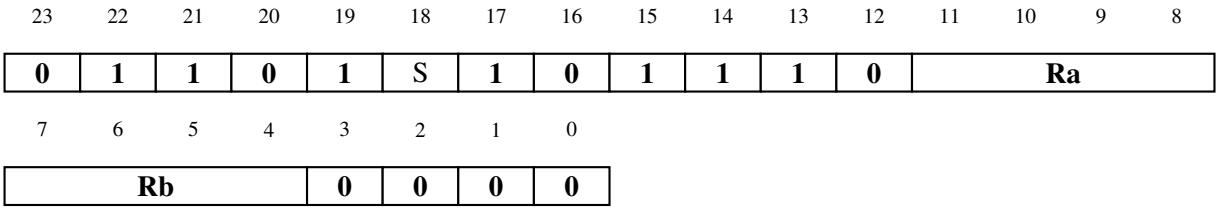
shr.i



Dieser Befehl führt eine logische Verschiebung nach rechts um einen bestimmten Wert aus. Dieser Wert wird von der Konstante *Imm* genommen.

Regs[Rd] = Regs[Ra] » Imm

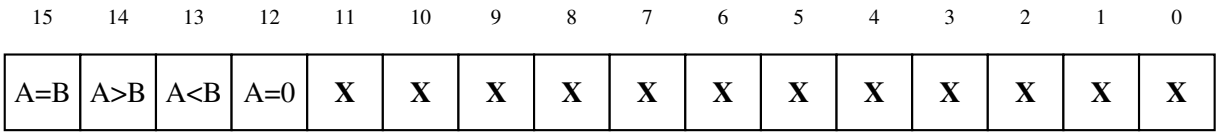
cmp.u, cmp.s



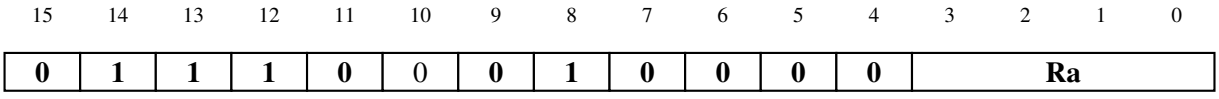
Vergleiche zwei Register und speichere die Vergleichsbitmaske in Register **R14**.

Regs[14] = cmp(Regs[Ra], Regs[Rb])

Vergleichsbitmaske



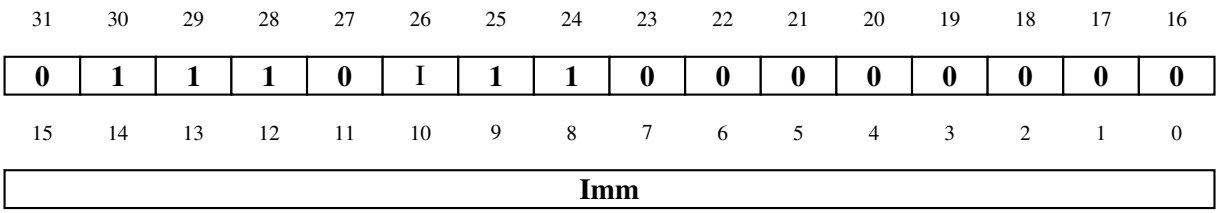
jmp.r



Springe zu Adresse. Die Adresse wird durch das Register *Ra* gegeben.

PC = Regs[Ra]

jmp.o, jmp.i

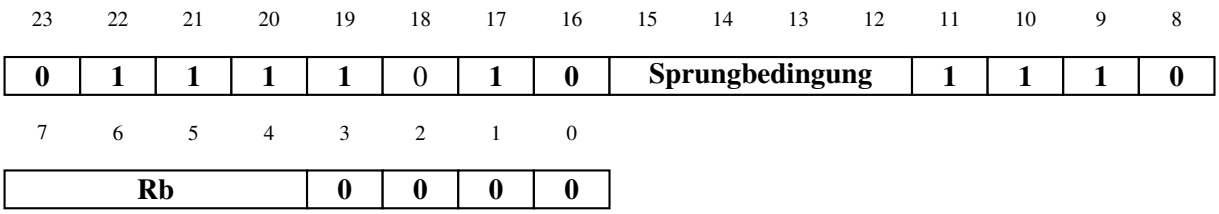


Springe zu Adresse. Wenn das Bit **I** gesetzt ist, wird der Wert der Konstante *Imm* als Adresse übernommen (1), andernfalls wird der Wert zum derzeitigen Wert des PC's addiert (2). Dabei wird im zweiten Fall immer von vorzeichenbehafteten Zahlen ausgegangen, damit auch Sprünge rückwarts stattfinden können.

(1) *PC = Imm*

(2) *PC = PC + Imm*

je.r, jne.r, jg.r, jl.r jge.r, jle.r, jp.r, jnp.r, jc.r, jnc.r, jo.r, jno.r, jz.r, jnz.r, jb.r, jnb.r



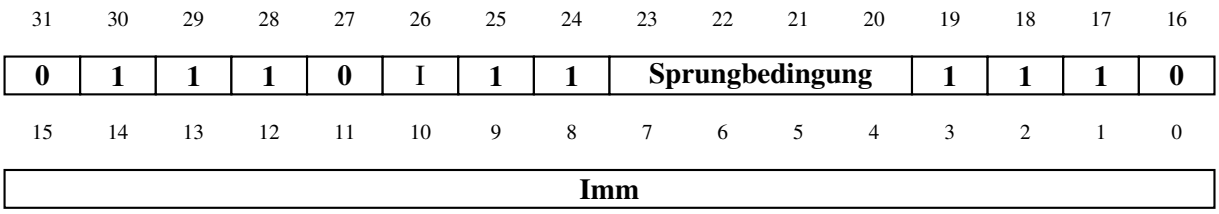
Springe zu Adresse, wenn die *Sprungbedingung* erfüllt ist. Die Adresse wird aus dem Register *Rb* gelesen. Zur Überprüfung der Sprungbedingung wird diese mit dem Register **R14** verglichen, welches das Ergebnis der **Vergleichsbitmaske** bzw. **Testbitmaske** enthält (siehe Befehl *cmp*). Mehr Informationen zur Sprungbedingung können der Tabelle A.1 entnommen werden.

Regs[14] = cmp(Regs[Ra], Regs[Rb])

Bitmuster	Befehl	Bedingung
0000	je	A == B
0001	jne	A != B
0010	jg	A > B
0011	jl	A < B
0100	jge	A >= B
0101	jle	A <= B
0110	jp	Paritätsbit von A ist gesetzt
0111	jnp	Paritätsbit von A ist nicht gesetzt
1000	jc	Carrybit gesetzt
1001	jnc	Carrybit nicht gesetzt
1010	jo	Überlaufbit gesetzt
1011	jno	Überlaufbit nicht gesetzt
1100	jz	A == 0
1101	jnz	A != 0
1110	jb	Bit gesetzt
1111	jnb	Bit nicht gesetzt

Tabelle A.1.: Sprungbedingung

je.o, je.i, jne.o, jne.i, jg.o, jg.i, jl.o jge.o, jl.i, jle.o, jle.i, jp.o, jp.i, jnp.o, jnp.i, jc.o, jc.i, jnc.o, jnc.i, jo.o, jo.i, jno.o, jno.i, jz.o, jz.i, jnz.o, jnz.i, jb.o, jb.i, jnb.o, jnb.i

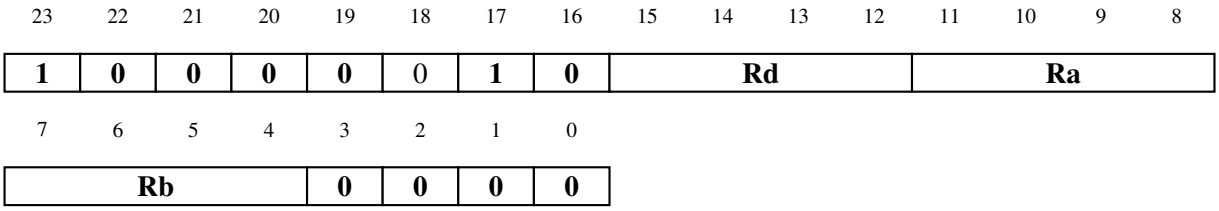


Springe zu Adresse, wenn die **Sprungbedingung** erfüllt ist. Dafür wird die **Sprungbedingung** mit dem Register **R14** verglichen. Wenn das Bit **I** gesetzt ist, wird der Wert der Konstante *Imm* als Adresse übernommen (1), andernfalls wird der Wert zum derzeitigen Wert des PC's addiert (2). Dabei wird im zweiten Fall immer von vorzeichenbehafteten Zahlen ausgegangen, damit auch Sprünge rückwarts stattfinden können. Weitere Informationen zur **Sprungbedingung** sind der Tabelle A.1 zu entnehmen.

(1) *PC = Imm (wenn Sprungbedingung erfüllt ist)*

(2) *PC = PC + Imm (wenn Sprungbedingung erfüllt ist)*

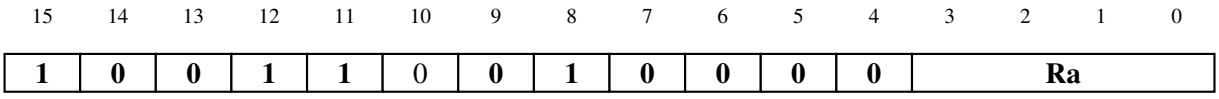
mul



Zwei Register werden miteinander multipliziert in das Ergebnis in ein drittes Register geschrieben.

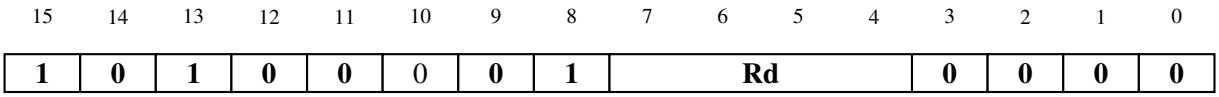
Regs[Rd] = Regs[Ra] · Regs[Rb]

push



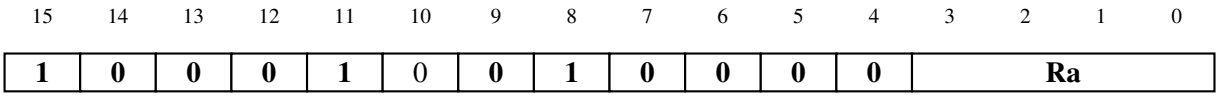
Schiebe den Wert des Registers *Ra* auf den Stack (Stapel).

pop



Hole das oberste Element des Stacks und schreibe es in das Register *Rd*.

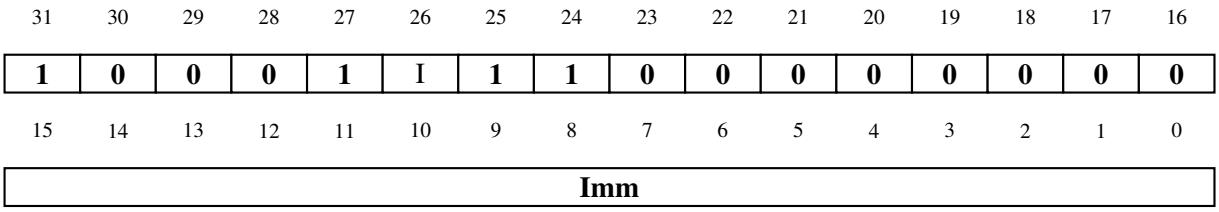
call.r



Rufe die Funktion an der Adresse *Ra* auf.

Hinweis: Dieser Befehl tut gleichzeitig die Rückkehraddresse auf den Stack schieben, damit die Funktion wieder ordnungsgemäß zurückkehren kann.

call.o, call.i



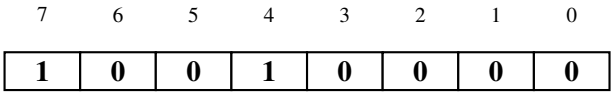
Rufe die Funktion an der Adresse *Ra* auf. Wenn das Bit **I** gesetzt ist, wird der Wert der Konstante *Imm* als Adresse übernommen (1), andernfalls wird der Wert zum derzeitigen Wert des PC's addiert (2).

(1) *PC = Imm (wenn Sprungbedingung erfüllt ist)*

(2) *PC = PC + Imm (wenn Sprungbedingung erfüllt ist)*

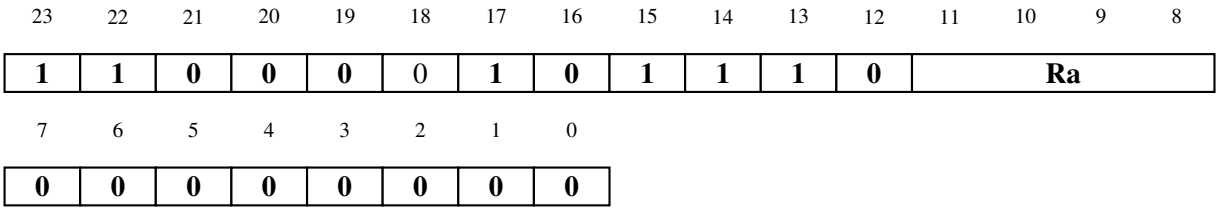
Hinweis: Dieser Befehl tut gleichzeitig die Rückkehraddresse auf den Stack schieben, damit die Funktion wieder ordnungsgemäß zurückkehren kann.

ret



Beendet die Funktion und kehrt damit zurück zum Aufruf der Funktion. Die Rückkehraddresse wird aus dem Stack entnommen.

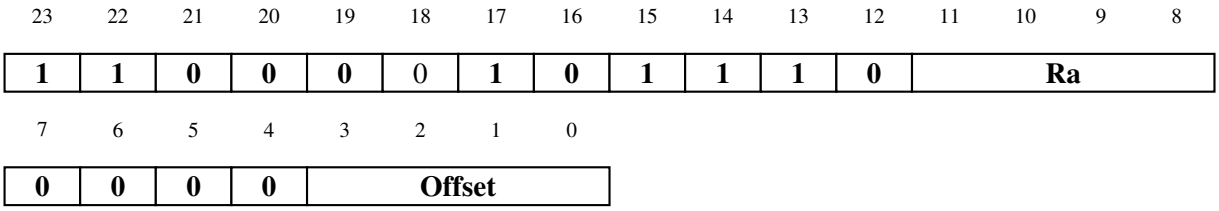
test.r



Teste das Register *Ra* auf besondere Eigenschaften und schreibe das Ergebnis als **Testbitmaske** in das Register **R14**.

Regs[14] = test(Regs[Ra])

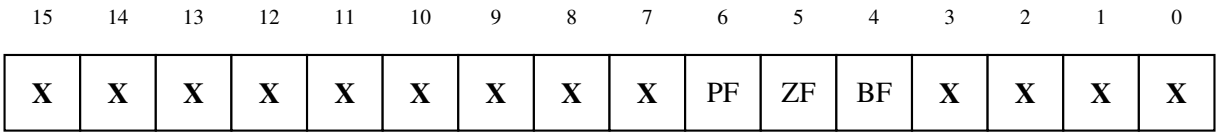
test.b



Teste ein Bit des Registers *Ra*. Die Eigenschaften werden mit der **Testbitmaske** festgehalten und in das Register **R14** geschrieben.

Regs[14] = test(Regs[Ra<Offset>])

Testbitmaske

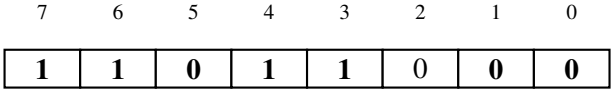


PF: Paritätsbit (gerade Parität)

ZF: A == 0

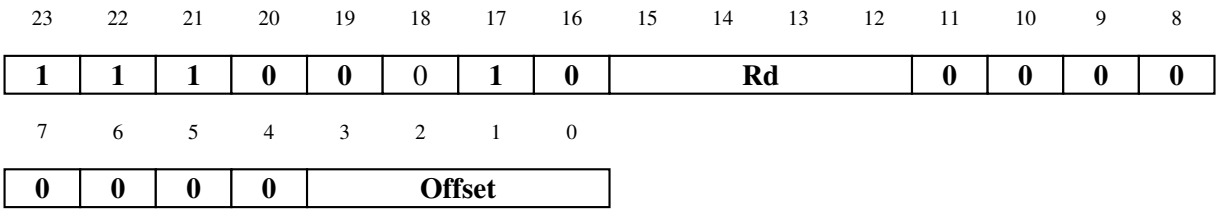
BF: Bit null (für *test.b*)

hlt



Stoppt die CPU. Keine weiteren Befehle werden ausgeführt.

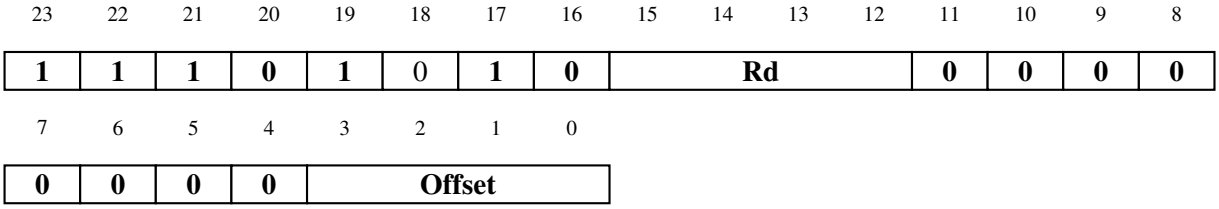
set



Setze ein Bit des Registers *Rd* auf den (logischen) Wert 1. Das zu setzende Bit wird durch den **Offset** angegeben.

Regs[Rd<Offset>] = 1

clr



Lösche ein Bit des Registers *Rd*, wodurch das Bit auf den (logischen) Wert 0 gesetzt wird. Das zu löschende Bit wird durch den **Offset** angegeben.

Regs[Rd<Offset>] = 0

Literaturverzeichnis

[1] Christopher Isreal und Marcel Jaoks. Implementierung einer 4-Bit MiniCPU in VHDL auf einem FPGA, Juli 2009.

[2] FPGA - Mikrocontroller.net. https://www.mikrocontroller.net/articles/FPGA#Aufbau. Eingesehen am 18.03.2017.