Prozessortechnik - Mikrocomputertechnik 3

Vorlesung und Übungen

mit Seminaraufgaben

Ausgabe 0.2, 23.11.2015
Autor: Stephan Rupp

Inhaltsverzeichnis

1. Serielle Schnittstellen

Serielle Schnittstellen haben parallele Bussysteme zunehmend abgelöst. Grund hierfür ist die Möglichkeit der Einsparung vieler Anschlusskontakte an den Bausteinen, sowie der hiermit verbundene Aufwand für Leitungen. Die für die serielle Verbindung erforderlichen hohen Datenraten sind keine technische Herausforderung mehr. Im Unterschied zu Parallelbussystemen vermeidet die serielle Übertragung Laufzeitunterschiede zwischen den einzelnen Signalen. Daher sind serielle Schnittstellen in der Praxis leistungsfähiger als Parallelbussysteme. Ein Beispiel hierfür ist die Ablösung von PCI durch PCI-Express auf Mikroprozessorsystemen. In diesem Abschnitt werden einfache serielle Schnittstellen zur Anbindung von Schnittstellenmodulen an FPGAs näher beschrieben.

1.1. SPI - Serielle Schnittstelle für Peripherie

Das Serial Peripheral Interface (SPI) ist ein synchrones serielles Protokoll zur Kommunikation eines Mikrocontrollers mit peripheren Geräten. Mit serieller Kommunikation ist gemeint, dass die Daten nacheinander über eine gemeinsames Medium geschickt werden, im Unterschied zur parallelen Kommunikation. Synchrone Kommunikation bedeutet, dass Sender und Empfänger hierbei synchron arbeiten. In der Praxis wird hierzu eine Taktleitung verwendet. Folgende Abbildung zeigt das Prinzip der synchronen seriellen Kommunikation.

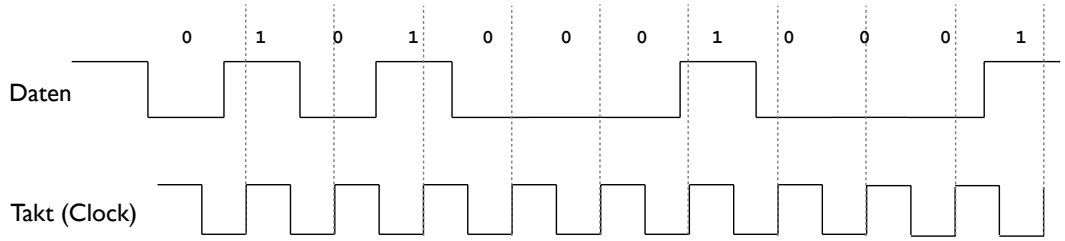
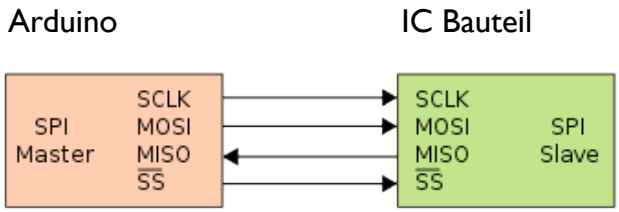


Bild 1.1 Synchrone serielle Kommunikation

Für die Daten ist eines der Systeme der Sender, das benachbarte System der Empfänger. Mit Hilfe des Taktes werden die Zeitpunkte definiert, an denen die Daten gültig sind und der Empfänger jeweils ein Datum übernehmen kann. Hierzu lassen sich z.B. die ansteigenden Flanken des Taktes verwenden, wie in der Abbildung gezeigt. Wer die Taktleitung betreibt, spielt für das Konzept keine Rolle. Allerdings muss es eine Stelle geben, die den Takt bereit stellt und die Kommunikation steuert.

Nach diesem Konzept werden also für eine bidirektionale Kommunikation zwischen Sender und Empfänger drei Leitungen benötigt: (1) die Taktleitung, (2) eine Leitung zum Senden, (3) eine Leitung zum Empfangen. Auf diese Weise miteinander verbundene Komponenten würden also Daten austauschen, solange ein Taktsignal gegeben wird. Gibt es mehr als zwei Kommunikationspartner, lässt sich der jeweils gewünschte Baustein durch ein weiteres signal auswählen. Bei SPI ist diese Auswahl durch eine weitere Leitung gelöst, wir in folgender Abbildung dargestellt.



Quelle: Wikimedia Commons

Bild 1.2 SPI als Kommunikationsschnittstelle

In der Abbildung zu sehen sind die bereits beschriebene Taktleitung (SCLK für SPI Clock), sowie die Datenleitungen in Senderichtung (MOSI für Master Out - Slave In), die Datenleitung in Empfangsrichtung (MISO für Master In - Slave Out). Die vierte Leitung ist mit SS bezeichnet für System Select (bzw. auch etwas neutraler als CS für Chip Select). Der Oberstrich auf der Bezeichnung ist als boolsche Negation zu interpretieren und bedeutet, dass das System durch den logischen Zustand LOW selektiert wird (als sogenannten active low Leitung).

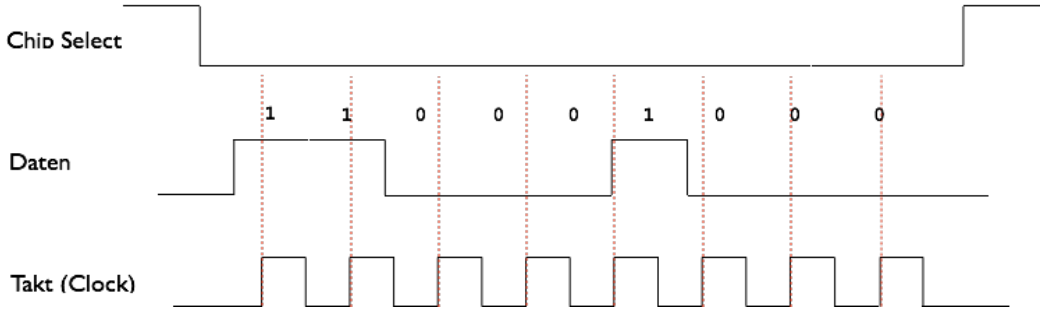


Bild 1.3 Ablauf der Kommunikation mit Hilfe der Chip Select Leitung

Eines der beiden Systeme muss die Taktleitung betreiben und die Kommunikation über die Chip Select Leitung organisieren. Diese Funktion ist als Master bezeichnet und ist in der Regel die Rolle des Mikrocontrollers. Die Bezeichnungen der beiden Datenleitungen beziehen sich auf dem Master der Kommunikation. Aus Sicht der Bauteile sind die Bezeichnungen jeweils DI (für Data In) oder SDI (für Serial Data In) bzw. DO (für Data Out) oder SDO (für Serial Data Out).

Die SPI-Spezifikation lässt Freiheitsgrade in der Interpretation des aktiven Taktsignals (Ruhewert bei LOW oder Ruhewert bei HIGH), sowie in der Wahl der aktiven Taktflanke (lesen der Daten bei steigender oder bei fallender Flanke). Hierzu tauchen also folgende Begriffe im Zusammenhang mit der SPI-Schnittstelle auf: CPOL (Clock Polarity) und CPHA (Clock Phase). Die vier kombinatorischen Möglichkeiten stellen die möglichen Betriebsmodi der Schnittstelle dar. Die folgende Übersicht verdeutlicht die Begriffe und den Zusammenhang.

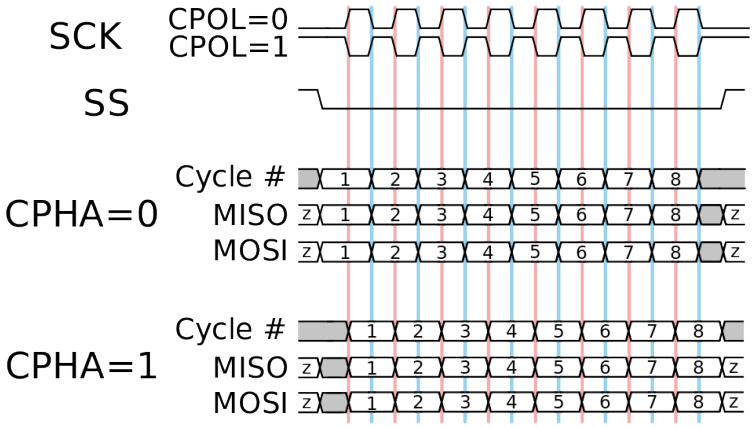


Bild 1.4 Betriebsmodi der SPI-Schnittstelle

Weitere Protokolldefinitionen sind Sache des Herstellers von SPI Bausteinen. Hierzu gehört die Organisation der Daten für die Konfiguration des Bausteins (durch Setzen von Registern), die Datenformate für den Austausch von Informationen, die Organisation der Daten in Bytes, ggf. Quittungsnachrichten etc. Das generelle Vorgehen ist dabei immer gleich: (1) den Baustein konfigurieren, (2) Daten austauschen. Welche genauen Informationen in welchem Format benötigt werden findet sich im Datenblatt des Herstellers.

Ansteuerung der SPI Schnittstelle

Bei der Erstellung eines Konzepts für die SPI-Schnittstelle ist ein Blockschaltbild hilfreich. Hierbei sind zu unterscheiden die externe Schnittstelle der Schaltung, d.h. der SPI-Bus, sowie die interne Schnittstelle der Schaltung zu einem eigenen, übergeordneten Block bzw. Programm. Die externe Schnittstelle ist hierbei vorgegeben: für einen SPI Master, der Daten an einen SPI-Empfänger überträgt, gehören hierzu die Signale MOSI (die serielle Datenleitung zum Empfänger), die SPI-Clock (SCLK) zum Eintakten der Bots in dem Empfänger, sowie das Chip Select-Signal (CS), das den Empfänger aktiviert. Die folgende Abbildung zeigt eine solche Anordnung.

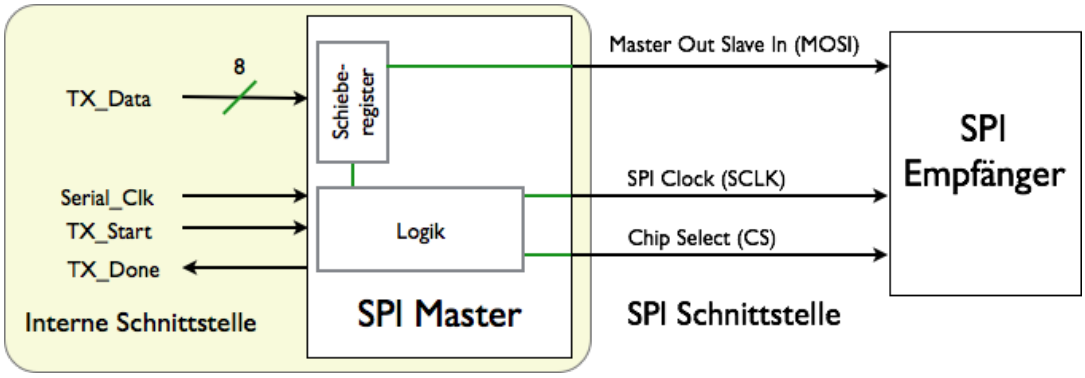


Bild 1.5 Blockdiagramm zur Orientierung

Die Schnittstelle zu einer Schaltung, die den SPI-Master bedient, ist hierbei eine interne Schnittstelle in dem Sinne, dass die Gestaltung völlig frei ist. Im in der Abbildung gezeigten Beispiel

wurden hierfür folgende Signale gewählt: Ein Byte an Daten (TX_Data), das parallel in den SPI-Master übertragen wird. Der SPI-Master überträgt diese Daten dann seriell über die SPI-Schnittstelle. Das Startsignal zur Übertragung (TX-Start) aktiviert die serielle Übertragung. Sobald die serielle Übertragung beendet ist, schickt der SPI-Master als Quittung das Signal TX_Done. Dem SPI-Master wird eine Takt zur seriellen Übertragung bereitgestellt: Serial_Clk.

Aus dieser Anordnung ergibt sich dann folgender Ablauf für den SPI-Master: (1) Beim Startsignal TX_Start mit der Übertragung der im Schieberegister enthaltenen Daten beginnen. (2) Übertragung: Hierzu erzeugt der SPI-Master das Chip-Select (CS) Signal für den SPI-Empfänger. Der SPI-Master zählt die benötigten Anzahl Takte mit und erzeugt eine passende Anzahl Takte (SCLK) für den Empfänger. (3) Das Ende der Übertragung signalisiert der SPI-Master mit Hilfe des Signals TX_Done. Folgender Programmtext beschreibt eine mögliche Realisierung.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Settings:
-- SPI-Mode 0: CPOL=0, CPHA=0
-- Serial_Clk : serial clock as input for SPI Clock
-- SCLK: SPI Clock output to slave

entity SPI_Master is
    Port ( TX_Data : in STD_LOGIC_VECTOR (7 downto 0); -- Transmit Register
           TX_Start   : in  STD_LOGIC;    -- Start Transmission
           Serial_Clk : in  STD_LOGIC;    -- Serial clock in at SCLK rate
           TX_Done : out STD_LOGIC; -- Handshake signal (transmission done)
           MOSI       : out STD_LOGIC;    -- SPI Master out signal
           SCLK       : out STD_LOGIC;    -- SPI clock signal out
           SS         : out STD_LOGIC     -- SPI chip select signal
         );
end SPI_Master;

architecture RTL of SPI_Master is

-- internal signals and variables
signal txreg        : std_logic_vector(7 downto 0) := (others=>'0');
signal counter      : integer := 0;

begin

  -- start transaction with new sample
 process(Serial_Clk, TX_Start) begin

 if (rising_edge(TX_Start)) then
   counter <= 8;    -- initialize counter for 8 bits transmission (9 - 1)
   TX_Done <= '0';  -- initialize handshake signal to not done
   SS <='0';        -- chip select is active low
   SCLK <= '0';     -- SCLK operated in CPOL = 0 (active low, rising edge)
   txreg <= TX_Data;
```

```
 elsif (TX_Start = '1') then
    if (falling_edge(Serial_Clk) and (counter > 0)) then
        SCLK <= '0'; -- SCLK follows input SPI clock for 8 counts
        MOSI <= txreg(7);
        for i in 0 to 6 loop
          txreg(7-i) <= txreg (6-i);
        end loop;
    elsif(rising_edge(Serial_Clk) and (counter > 0))then
        SCLK <= '1'; -- SCLK follows input SPI clock for 8 counts
        counter <= counter - 1;
    end if;

    if (counter = 0) then
      TX_Done <= '1'; -- handshake signal for transmission completed
      SS <='1';       -- deselect chip
    end if;

   end if;

  end process;
end RTL;
```

Bei dieser Implementierung wird die Synchronisation des SPI-Masters mit der internen Schnittstelle zur Steuerung des Datentransfers über SPI durch folgende Massnahmen erreicht: (1) Start der Übertragung durch Trägern auf die steigende Flanke des Steuersignals TX_Start. In diesem Zustand wird dann auch das Chip-Select-Signal bedient, sowie ein interner Bitzähler counter gesetzt. (2) Solange TX-Start gesetzt bleibt, folgt der SPI-Master der seriellen Taktrate Ser_Clk, überträgt jeweils ein Bit, erzeugt die SPI-Clock SCLK und zählt mit jedem Takt den Bitzähler herunter. Synchronität ist nur dadurch gewährleistet, dass die SPI-Clock aus dem Zustand Low gestartet wird, d.h. dass auf das Signal TX_Start zunächst eine steigende Taktflanke der SPI-Clock SCLK folgen muss. Hierzu muss auf TX-Start zunächst eine fallende Flanke der seriellen Eingangstaktes Ser_Clk folgen.

Testprogramm für den SPI Master

Zum Testen der Schaltung wird die interne Schnittstelle des SPI-Masters bedient. Diese Schnittstelle stellt die serielle Taktrate für die Übertragung bereit. Mit dem in der letzten Abbildung gewählten Konzept ist der Ablauf wie folgt: (1) Daten parallel in das Schieberegister des SPI-Masters schreiben. (2) Die Übertragung mit Hilfe des Startsignals TX-Start starten. (3) Auf das Ende der Übertragung warten (Signal TX_Done des SPI-Masters). Nach der Sendebestätigung kann das nächste Byte übertragen werden. Folgender Programmtext beschreibt eine mögliche Realisierung.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY test_SPI_Master IS
END test_SPI_Master;
```

```
ARCHITECTURE behaviour OF test_spi_master IS

  -- Component Declaration for the Unit Under Test (UUT)
  COMPONENT SPI_Master
     Port (TX_Data : in STD_LOGIC_VECTOR (7 downto 0); -- Transmit Register
           TX_Start   : in  STD_LOGIC;    -- Start Transmission
           Serial_Clk : in  STD_LOGIC;    -- serial clock in at SCLK rate
           TX_Done    : out STD_LOGIC;    -- Handshake signal (transm. done)
           MOSI       : out STD_LOGIC;    -- Master out signal
           SCLK       : out STD_LOGIC;    -- SPI clock signal out
           SS         : out STD_LOGIC     -- chip select signal
          );
  END COMPONENT;

  -- Signals
  SIGNAL T_TX_Data    :  std_logic_vector(7 downto 0) := (others=>'0');
  SIGNAL T_TX_Start   :  std_logic := '0';
  SIGNAL T_Serial_Clk :  std_logic;
  SIGNAL T_TX_Done    :  std_logic;
  SIGNAL T_MOSI       :  std_logic;
  SIGNAL T_SCLK       :  std_logic;
  SIGNAL T_SS         :  std_logic;
  SIGNAL rxreg        :  std_logic_vector(7 downto 0) := x"00";

BEGIN
  -- Connect DUT to testbenck (port map)
  DUT: SPI_Master PORT MAP(
    TX_Data => T_TX_Data,
    TX_Start=> T_TX_Start,
    Serial_Clk => T_Serial_Clk,
    TX_Done => T_TX_Done,
    MOSI    => T_MOSI,
    SCLK    => T_SCLK,
    SS      => T_SS
  );


  -- operate DUT
  clock : process begin
    T_Serial_CLK <= '1';
    wait for 10 ns;
    T_Serial_CLK <= '0';
    wait for 10 ns;
  end process clock;

  test : process begin

    T_TX_Data <= x"00"; -- transmit control byte
    T_TX_Start <= '1';
    wait until T_TX_Done ='1';
    wait for 5 ns;
    T_TX_Start <= '0';
```

```
    wait for 50 ns;

    T_TX_Data <= x"bb"; -- transmit data byte
    T_TX_Start <= '1';
    wait until T_TX_Done ='1';
    wait for 5 ns;
    T_TX_Start <= '0';
    wait for 100 ns;

  END PROCESS test;

END Behavior;
```

Das Textprogramm hat folgende parallele Prozesse: (1) Die Erzeugung des seriellen Taktes, (2) Die Übertragung zweier Bytes. In der gewählten Realisierung zählt der SPI-Master die seriell übertragenen Bits eigenständig mit und erzeugt selber das Chip-Select Signal. Folgende Abbildung zeigt den Test der Schaltung auf dem Simulator.

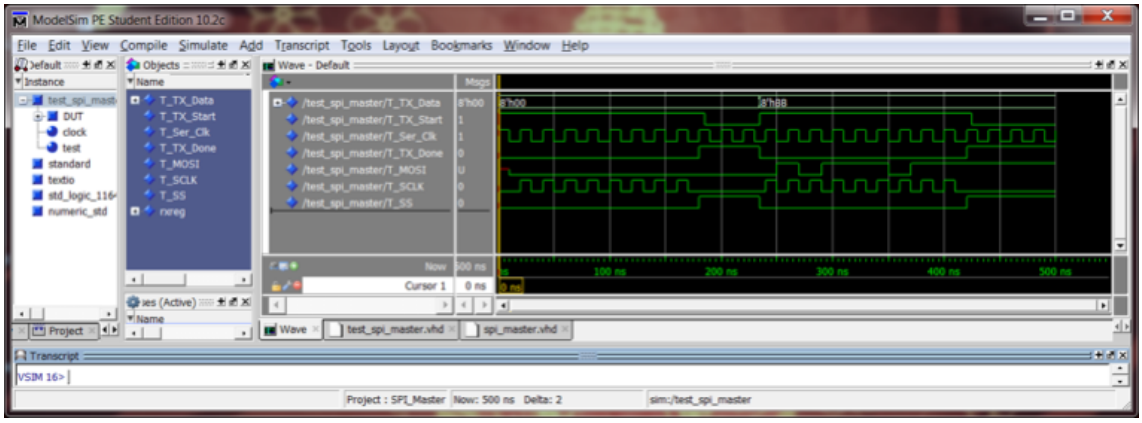
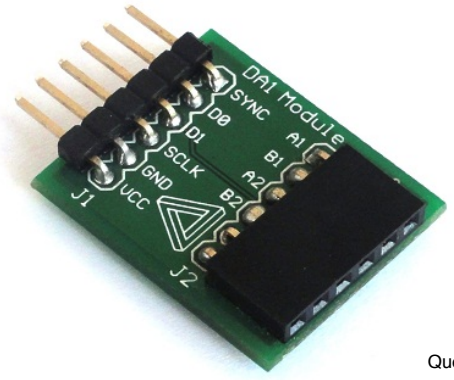


Bild 1.6 Test der SPI-Schnittstelle im Simulator

Man erkennt die Zustände der Datenleitung und kann jeweils 8 Taktsignale für den SPI-Clock (CSLK) abzählen. Ein Empfänger würde die Signale der Datenleitung mit jeder steigenden Flanke der SPI-Clock (CSLK) in sein Eingangs-Schieberegister einlesen. Am Ende jedes übertragenen Bytes setzt der SPI-Master das Chip-Select Signal (CS) eigenständig auf den inaktiven Zustand (High).

Ein praktisches Beispiel

Als Beispiel sei ein D/A Wandler Modul gewählt, hier das PmodDA1 des Herstellers Digilent, das an der DHBW im Einsatz ist. Für eigene Experimente stellt Ihnen der Dozent gerne ein Exemplar zur Verfügung. Folgende Abbildung zeigt das Modul und das Blockschaltbild.



Quelle: Digilent Inc.

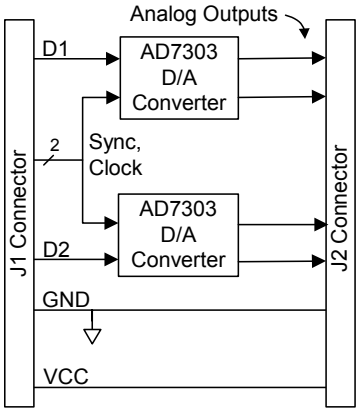
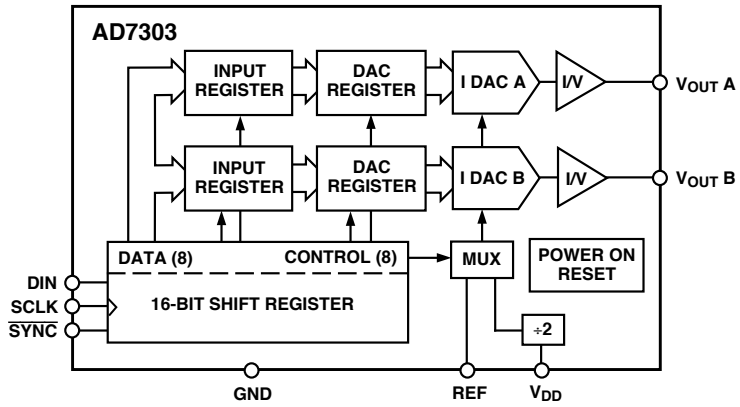


Bild 1.7 D/A Wandler Modul PmodDA1

Auf dem Modul finden sich zwei D/A Wandler Bausteine AD7303. Der Weg führt also weiter zum Datenblatt des Herstellers Analog Devices. Dort findet sich das Blockschaltbild des Bausteins und ein Zeitdiagramm der SPI-Schnittstelle, wie in folgender Abbildung gezeigt.



Quelle: Analog Devices

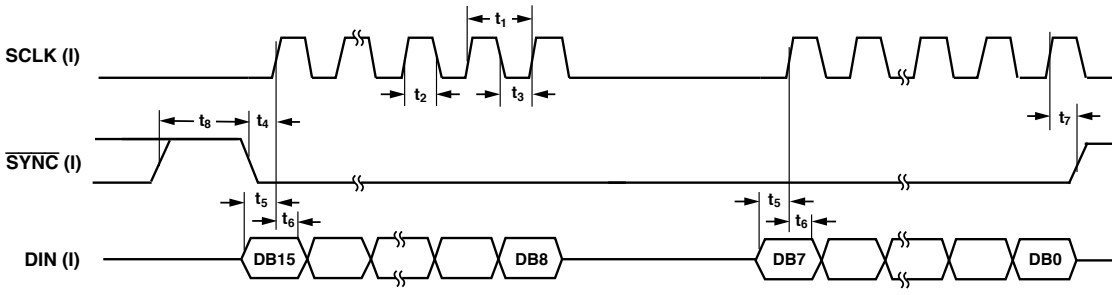


Bild 1.8 Blockschaltbild und Zeitdiagramm des D/A Wandler Bausteins

Man erkennt, dass der Baustein zwei Bytes an Daten erwartet: (1) ein Kommando (Control mit 8 Bit) zur Steuerung der Betriebsart, (2) Daten (mit ebenfalls 8 Bit). Das Schieberegister am Eingang des Chips ist dementsprechend 16 Bit lang. Das höchstwertigste Bit DB15 wird zuerst gesendet, d.h. die Reihenfolge ist MSB-LSB. Ausserdem erkennt man am Zeitdiagramm, dass der Ruhezustand der Clock 0 ist und die aktive Flanke die ansteigende Flanke (entsprechend CPHA = 0). Der SPI-Modus wäre demnach 0.

Übung 1.1: Zum Betrieb an diesem Baustein wären 16-Bits zu übertragen, d.h. 2 Bytes zwischen zwei Chip-Select Signalen. Modifizieren Sie die Schaltung hierfür und testen Sie am Simulator.

Übung 1.2: Im Falle der Übermittlung von Abtastwerten an den D/A-Wandler wäre die Schaltung mit einer Frequenz für die Abtastwerte F_{sample} zu betreiben, sowie mit einer Bit-Clock für die serielle Schnittstelle F_{serial}. Das FPGA stellt eine Quarzclock F_{Qaurz} zur Verfügung. Wie wären die beiden Takte sinnvoll hieraus zu gewinnen?

Einsatz eines Zustandsautomaten

Die bisherige Implementierung kennt nur die Zustände Übertragung starten (mit Hilfe des Steuersignals TX_Start). Nach dem Start wartet man auf das Quittungssignal TX_Done, mit dem der SPI-Master das Ende der Übertragung anzeigt. Nach Ende der Übertragung befindet man sich in einem Wartezustand. Um die Übertragung mehrerer Bytes zu ermöglichen, sonn ein weiterer Zustand eingeführt werden: die Initialisierung des seriellen Busses mit Hilde des Chip-Select-Signals. Folgende Abbildung zeigt das Zustandsdiagramm des Automaten.

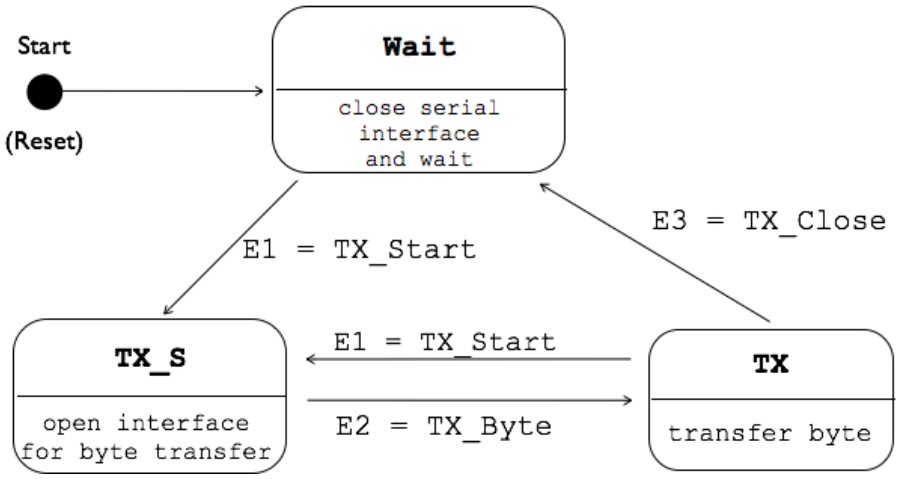


Bild 1.9 Zustandsdiagramm des Automaten zur Steuerung der Schnittstelle

Das Steuersignal TX_Start führt nun vom Wartezustand in den Zustand der Initialisierung des seriellen Busses (TX_S). Die Übertragung eines Bytes findet in diesem Zustand noch nicht statt, sondern wird durch ein weiteres Steuersignal TX_Byte veranlasst, das in den Zustand der Übertragung (TX) führt. Nach Übertragung des Bytes aus dem Zustand TX lässt sich entweder (1) wiederum in den Zustand TX_S wechseln (durch erneuten Senden des Steuersignals TXStart), (2) oder die Übertragung mit Hilfe des Steuersignals TX_Close beenden. In diesem Fall wechselt der Automat in den Wartezustand Wait.

Zur Synchronisation der Steuerung soll der Automat aus jedem Zustand heraus Quittungen schicken, sobald er die jeweiligen Zustandsaktionen ausgeführt hat. Folgende Abbildung zeigt die erweiterte interne Schnittstelle des Automaten mit allen Steuersignalen und Quittungen. Hierbei wird das Steuersignal TX_Start quittiert durch QX_Ready (Automat hat Zugriff auf die seriellen Schnittstelle und ist bereit zur Übertragung), das Signal TX_Byte wird quittiert durch QX_Byte (Automat hat ein

Byte übertragen), das Signal TX_Close wird quittiert durch QX_Closed (Automat hat die serielle Schnittstelle wieder freigegeben).

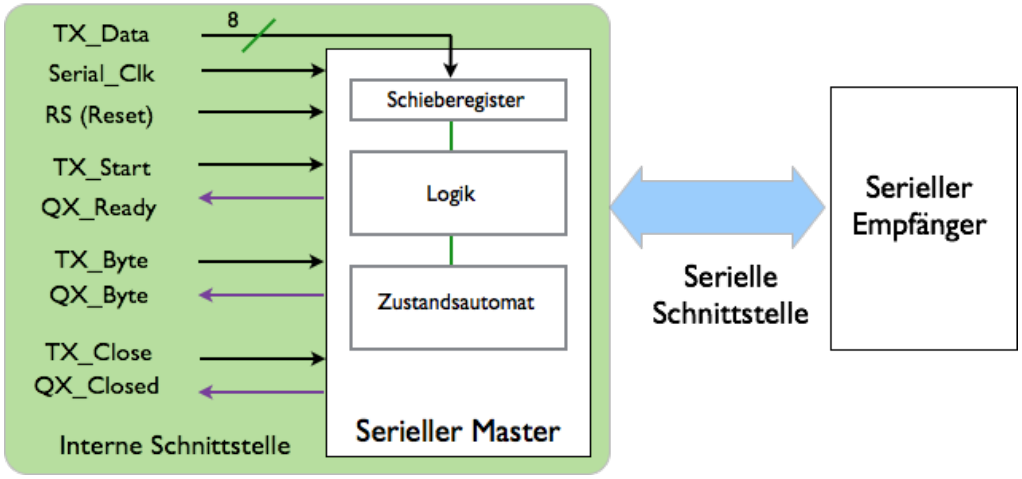


Bild 1.10 Blockdiagramm der erweiterten Steuerschnittstelle

Mit Hilfe dieser Vorgabe lässt sich nun ein Zustandsautomat zur Steuerung der Schnittstelle realisieren. Hierbei kann man der Vorgehensweise aus der Vorlesung Entwurf digitaler Systeme folgen (siehe (1) im Literaturverzeichnis).

Übung 1.3: Entwerfen Sie ein Testprogramm des Automaten, das die interne Schnittstelle bedient. Steuern Sie den Automaten so, dass mehrere Bytes übertragen werden.

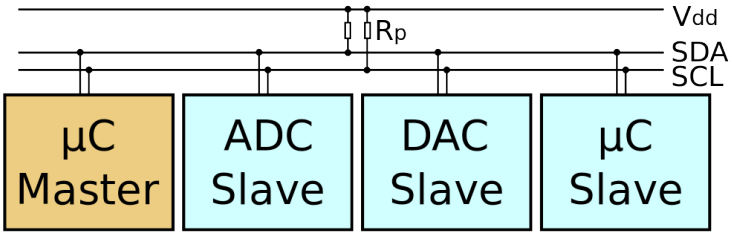
Übung 1.4: Entwerfen Sie den Zustandsautomaten zur Steuerung der SPI-Schnittstelle. Hinweis: Die Implementierung der Schnittstelle kann im Ausgangsschaltwerk des Automaten erfolgen.

Übung 1.5: Testen Sie den Automaten mit Hilfe des im Übung 1.4 erstellten Testprogramms.

1.2. I2C Bus

Während SPI insgesamt 4 Leitungen für eine bidirektionale Übertragung benötigt (bzw. 3 Leitungen für eine unidirektionale Übertragung), ist kommt der I²C Bus mit insgesamt 2 Leitungen aus. Den Unterschied macht die Adressierung der angeschlossenen Geräte: SPI benötigt für jedes Gerät am Controller (Master) eine eigene Adressleitung (Chip Select). I²C arbeitet mit Geräteadressen und benötigt daher keine eigene Leitung zur Auswahl eines Gerätes. Ausserdem wird nur eine einzige Datenleitung für beide Richtungen verwendet.

Ein I²C System hat die in folgender Abbildung gezeigte Struktur. Wegen der gemeinsamen Datenleitung spricht man hier auch von einer Bus-Struktur. Da die Schnittstelle seriell arbeitet, ist die Funktion der beiden Busleitungen eindeutig: eine der Leitungen ist die Taktleitung (SCL - Serial Clock Line), die andere Leitung ist die Datenleitung (SDA - Serial DAta Line).

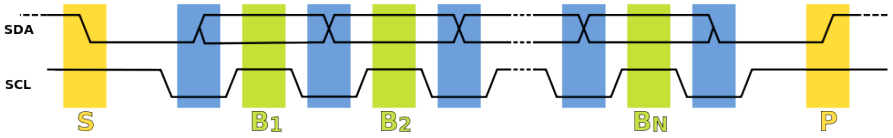


Quelle: Wikimedia Commons

Bild 1.11 I2C Zweitdraht Bussystem

Den Nachrichtenaustausch steuert beim Zweidrahtbus I²C ebenfalls ein Master, in aller Regel der Mikrocontroller. Elektrisch haben alle angeschlossenen Bauteile offene Kollektorausgänge und sind über die Pull-up Widerstände an beiden Leitungen auf definiertem Potential. Dieser Anschluss stellt eine verdrahtete ODER-Verknüpfung der Ausgänge dar und wird zur Signalisierung benutzt. Hohe Datenraten sind bei diesem Konzept nicht vorgesehen. Die Taktrate liegt üblicherweise bei maximal 100 kHz im Standardmodus, bzw. bei maximal 400 kHz im sogenannten Fast Mode.

Für die Kommunikation gibt der Master den Takt vor. Das Protokoll gibt vor, dass jeweils Einheiten von 8 Bits übertragen werden. Bei der moderaten Taktrate lassen sich Phasenübergänge der Signale auf beiden Leitungen zur Steuerung der Kommunikation nutzen. Ein Startsignal gibt der Master durch eine fallende Flanke auf der Datenleitung, währen die Taktleitung noch im Ruhezustand ist (Ruhezustand = HIGH). Als Stopp-Signal zieht der Master die Datenleitung auf HIGH, nachdem die Taktleitung bereits in den Ruhezustand übergegangen ist. Folgende Abbildung zeigt den Ablauf der Kommunikation.



Quelle: Wikimedia Commons

Bild 1.12 Kommunikation über den Zweidrahtbus I²C

Auffällig ist, dass Daten nicht bei fallenden oder steigenden Taktflanken interpretiert werden, sondern im Zustand HIGH der Taktleitung. Während dieses Zustandes dürfen sich die Daten auf der Datenleitung nicht ändern. Nach 8 Datenbits wird ein weiteres Protokollbit übertragen, das sogenannte Bestätigungsbit (Acknowledge Bit), das den Empfänger der Nachricht (vorausgegangene Bits) zu einer Quittierung des Empfangs motivieren soll. Wurde die Nachricht korrekt empfangen, legt der Empfänger die Datenleitung auf LOW und hält die Datenleitung in diesem Zustand während der folgenden HIGH Phase der Taktleitung. Die Taktleitung verbleibt in diesem Zustand (Ruhezustand). Der Übergang der Datenleitung in den Zustand LOW muss vor dem Übergang der Taktleitung in den Zustand HIGH erfolgen, um Missverständnisse zu vermeiden. Der Master hebt im Anschluss an die Interpretation der Quittung die Datenleitung auf HIGH und meldet hiermit das Stopp-Signal.

Wie werden nun Geräte adressiert? I²C verwendet üblicherweise 7-Bit Adressen, die im Byte Format als Nachricht verschickt werden. Das achte Bit kennzeichnet, ob die folgende Nachricht vom gewünschten Empfänger gelesen werden soll, oder ob der Empfänger eine Nachricht an den Master senden soll. Für Geräte, die entweder nur lesen oder nur schreiben können, spielt diese Unterscheidung allerdings keine Rolle. Die Geräteadressen werden vom Hersteller festgelegt, wobei

einige Adressbits vom Anwender durch Verdrahtung konfiguriert werden können, z.B. um mehrere baugleiche Geräte zu betreiben.

Ein praktisches Beispiel

Als I²C Baustein soll der 12-Bit Digital-Analog-Konverter MCP4725 zum Einsatz kommen. Dem Datenblatt entnimmt man, dass der Baustein über eine Spannungsversorgung (V_{DD}) mit Masse (V_{SS}) verfügt, und das analoge Ausgangssignal (V_{OUT}) herausgeführt wird. Die digitalen Abtastwerte werden über die I²C-Schnittstelle mit den Anschlüssen SCL und SDA angeschlossen. Ausserdem findet sich ein Eingang A0 zur Auswahl der Geräteadresse am I²C-Bus. Der Baustein kann im Standard Modus mit 100 kHz Datenrate, bzw. im sogenannten Fast-Mode mit 400 kHz Datenrate betrieben werden.

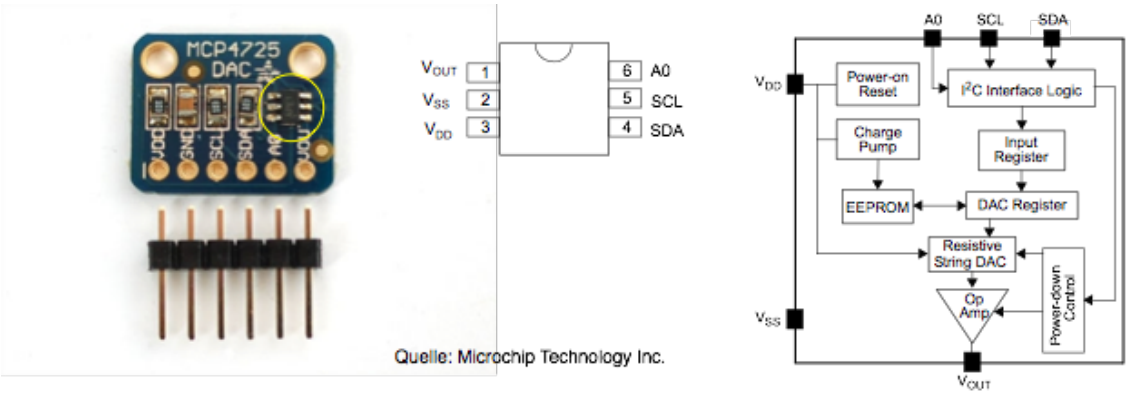


Bild 1.13 MCP4725 12-Bit DAC

Zum Speichern der Konfiguration (2 Bits), sowie der digitalen Eingangssignale (12-Bits) enthält der Baustein ein E-PROM. Die Konfiguration dient zum durch die Software gesteuerten Deaktivieren des Bausteins (Power-Down Modus), wobei der Ausgang auf unterschiedliche Impedanzen geschaltet werden kann. Ein DAC-Register ermöglich die Konfiguration des Übertragungsmodus am Bus. Hier enthalten sind 3 Bits zur Einstellung des Übertragungsmodus (Standard, Fast-Mode), ob das DAC-Register bzw. das EPROM beschrieben werden soll, bzw. ob das Statusbit (Ready/Busy) ausgelesen werden soll. Folgende Abbildung zeigt ein Beispiel für die Kommunikation.

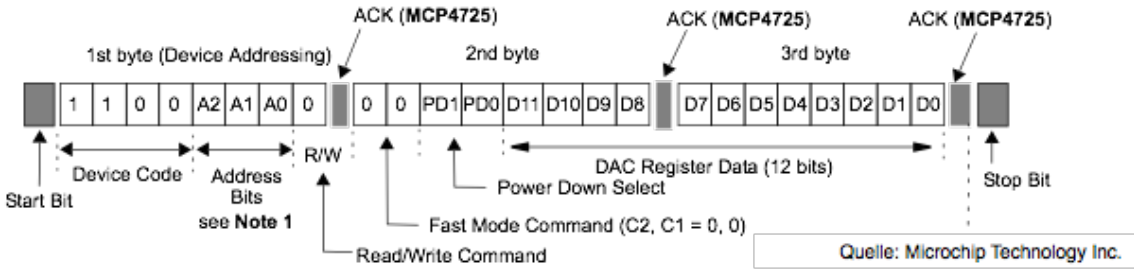


Bild 1.14 Beispiel für die Kommunikation über die serielle Schnittstelle

Das erste übertragene Byte dient der Auswahl des Bausteins. Hierbei sind die Bits A2 und A1 der Adresse vom Hersteller fest voreingestellt (als Standard „00“, bzw. auf Kundenwunsch), nur das Bit A0 wird durch die Beschaltung auf logisch Null oder Eins eingestellt. Die Konfigurationsdaten werden im zweiten Byte übertragen, zusammen mit den obersten 4 Bits des digitalen Signals. Mit einem dritten Byte endet die Übertragung eines Abtastwertes. Gemäß Spezifikation des I²C-Busses quittiert der Baustein die Übertragung im Anschluss an jedes Byte.

Seminararbeit S1: Erstellen Sie ein Konzept zur Anbindung eines I2C-Bausteins auf Ihrem FPGA. Erstellen Sie einen Schaltungsentwurf und Test Sie den Entwurf. Synthetisieren Sie die Schaltung und testen Sie die Implementierung. Für den Test stehen Ihnen DAC-Bausteine MCP4725 DAC zur Verfügung.

Emulation eines Mikrocontrollers

Bei den relativ bescheidenen Übertragungsraten von I²C lässt sich diese Schnittstelle auch durch Mikrocontroller bedienen, die man auf dem programmierbaren Baustein (FPGA) emuliert. Bei gängigen Mikrocontrollern finden sich in diesem Fall fertige Bibliotheken bzw. fertige HDL-Bausteine (IP - Intellectual Property) für solche Anwendungen. Die Emulation eines Mikrocontrollers auf FPGA ist Teil von Abschnitt 3 dieses Manuskripts.

1.3. USB

Seminararbeit S2: Recherchieren Sie nach der Funktion und nach Realisierungsmöglichkeiten für eine USB-Schnittstelle auf Ihrem FPGA. Erstellen Sie ein Konzept. Erstellen Sie einen Schaltungsentwurf und Test Sie den Entwurf. Synthetisieren Sie die Schaltung und testen Sie die Implementierung durch Anbindung des FPGAs an Ihrem PC via USB.

2. Signalverarbeitung

2.1. Signale und Variablen in HDL

In der digitalen Signalverarbeitung werden Berechnungen in zeitdiskreten Systemen durchgeführt, wobei unter den Signalen die Abtastwerte bzw. die berechneten Werte verstanden werden. In diese Abschnitt geht es um Signale und Variablen als Schlüsselworte in VHDL. Beide können mit geeigneten Datentypen zur Signalverarbeitung eingesetzt werden. Signale in VHDL werden als globale Variablen verwendet, die über mehrere Prozesse hinweg sichtbar sind. Die sogenannten Variablen verwendet man als lokale Variablen innerhalb eines Prozesses. Folgende Beispiele sollen die Unterschiede im Gebrauch von Signalen und Variablen zeigen.

```
--- Demonstrate Signals and Variables (VHDL)
library ieee;
use ieee.std_logic_1164.all;

entity SigVarDemo is
      port ( Clk : in std_logic;   -- clock
               D : in std_logic;   -- one bit in
               Q : out std_logic); -- one bit out
end SigVarDemo;

architecture RTL of SigVarDemo is
      signal Qi: std_logic_vector (2 downto 0) := (others =>'0');
      begin
```

```
      process (Clk)                  -- first process using Qi
          begin
            if rising_edge(Clk) then
               Qi(2) <= Qi(1);
               Qi(1) <= Qi(0);
               Qi(0) <= D;
            end if;
      end process;

      Q <= Qi(2);                    -- second process using Qi

end RTL;
```

Übung 2.1: Analysieren Sie den HDL-Text. Welches Verhalten erwarten Sie von dieser Schaltung?

Übung 2.2: Schreiben Sie ein Testprogramm und testen Sie die Schaltung. Entspricht das Ergebnis Ihren Erwartungen?

Im gleichen HDL-Programm wird nun die Reihenfolge der Signalzuweisungen innerhalb des taktsensitiven Prozesses process (Clk) vertauscht:

```
      process (Clk)
          begin
            if rising_edge(Clk) then
               Qi(0) <= D;
               Qi(1) <= Qi(0);
               Qi(2) <= Qi(1);
            end if;
      end process;
```

Übung 2.3: Welches Verhalten erwarten Sie nun von der Schaltung?

Übung 2.4: Testen Sie die Schaltung mit dem Testprogramm aus Übung 1.7. Entspricht das Ergebnis Ihren Erwartungen? Erklären Sie, warum sich die Schaltung so verhält. Hinweise: (1) Der Prozess process (Clk) ist taktsensitiv und reagiert nur auf steigende Flanken. (2) Das Register Qi ist als Signal innerhalb des Architektur-Blocks der Entity prozessübergreifend definiert. Ein weiterer Prozess, der auf das Register zugreift, findet sich in der Zeile Q <= Qi(2).

Beide Testläufe zeigen folgende Ergebnisse (das Testprogramm schreibt ein Muster 010 010 … auf den Eingang des Prüflings):

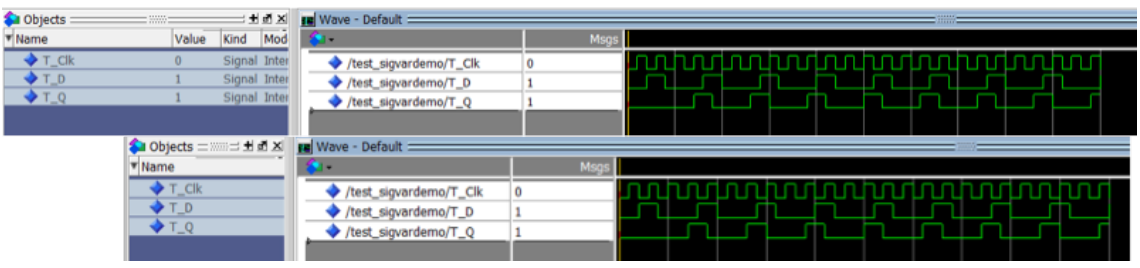


Bild 2.1 Testläufe beider Varianten

In folgendem HDL-Text wird nun statt des Signals Qi eine Variable Vi mit identischem Datentyp (3-Bit breiter Vector) verwendet. Da die Variable nur innerhalb eines Prozesses Gültigkeit hat, wird sie im Prozessblock process (Clk) deklariert. Der Datentyp ist identisch mit dem vorher verwendeten Signal Qi. Damit der Prozess einen Wert von Vi nach außen kommunizierten kann, wird als Signal Qi ein einzelnes Bit beibehalten. Diesem Signal übergibt das Prozess process (Clk) dann ein Bit der Variablen Vi. Um Variablen und Signale besser auseinander halten zu können, ist die Syntax bei der Wertübergabe unterschiedlich („:=„ bei Variablen, siehe Programmtext).

```
--- Demonstrate Signals and Variables (VHDL)
library ieee;
use ieee.std_logic_1164.all;

entity SigVarDemo is
      port ( Clk : in std_logic;   -- clock
               D : in std_logic;   -- one bit in
               Q : out std_logic); -- one bit out
end SigVarDemo;

architecture RTL of SigVarDemo is
      signal Qi: std_logic := '0'; -- one bit internal signal
      begin

      process (Clk)                -- first process using Vi and Qi
        variable Vi: std_logic_vector (2 downto 0) := (others =>'0');
        begin
            if rising_edge(Clk) then
               -- Version 1:            -- Version 2:
               Vi(2) := Vi(1);          -- Vi(0) := D;
               Vi(1) := Vi(0);          -- Vi(1) := Vi(0);
               Vi(0) := D;              -- Vi(2) := Vi(1);
            end if;
            Qi <= Vi(2);
      end process;

      Q <= Qi;                    -- second process using Qi

end RTL;
```

Übung 2.5: Testen Sie beide Programmversionen, die Variablen benutzen mit Hilfe des Testprogramms aus Übung 1.7. Hinweis: Die zweite Variante findet sich als Kommentar im Programmtext oben. Entsprechen die Ergebnisse Ihren Erwartungen? Erklären Sie das Verhalten der Schaltungen in beiden Fällen. Welche Unterschiede ergeben sich bzgl. der Verwendung von Signalen und Variablen durch Prozesse?

Die Testläufe beider Varianten zeigen folgende Ergebnisse (das Testprogramm schreibt wiederum ein Muster 010 010 … auf den Eingang des Prüflings):

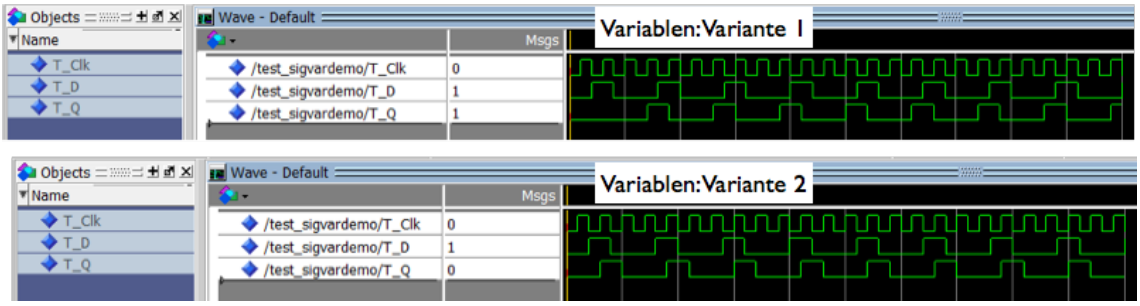


Bild 2.2 Testläufe beider Varianten

Hier ergeben sich gravierende Unterschiede: Variante 1 verhält sich wie ein 3-Bit Schieberegister, Variante 2 wie ein D-Flip-Flop (einfache Abtastung mit steigender Taktflanke). Der Vollständigkeit halber folgt an dieser Stelle noch das verwendete Testprogramm.

```
--- Test SigVarDemo (VHDL)
library ieee;
use ieee.std_logic_1164.all;

entity Test_SigVarDemo is
end Test_SigVarDemo;

architecture Behavioural of Test_SigVarDemo is

component SigVarDemo is
      port ( Clk : in std_logic;   -- clock
               D : in std_logic;   -- one bit in
               Q : out std_logic); -- one bit out
end component;

-- testbench internal signals
signal T_Clk: std_logic;
signal T_D : std_logic := '0';
signal T_Q : std_logic;

begin

-- connect DUT to testbench
DUT: SigVarDemo port map (Clk => T_Clk, D => T_D, Q => T_Q);

-- run tests

count: process -- write 010 010 ... pattern to D
  begin
  T_D <= '0';
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

  T_D <= '1';
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
```

```
  T_D <= '0';
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

end process count;

end Behavioural;
```

2.2. FIR Filter

Die Faltung einer zeitdiskreter Funktionen x(i) mit der Impulsantwort h_k wird durch die Faltungssumme

$$y[n] = \sum h_k \cdot x[n-k] \qquad (2.1)$$

beschrieben, wobei der Index k über alle vorhandenen Stützstellen h_k verläuft. Für eine Impulsantwort mit insgesamt 5 Stützstellen ergibt sich folgende Gleichung.

$$y[n] = h_0 \cdot x[n] + h_1 \cdot x[n-1] + h_2 \cdot x[n-2] + h_3 \cdot x[n-3] + h_4 \cdot x[n-4] \qquad (2.2)$$

Bei der Realisierung werden wiederholte Multiplikationen und Additionen erforderlich. Wegen der Möglichkeit zur Parallelverarbeitung und der Realisierbarkeit individuell zugeschnittener Rechenwerke eignen sich der HDL-Entwurf für Filter zur Signalverarbeitung.

Übung 2.6: Geben Sie als Eingangssignal einen Impuls vor, d.h. x(0) = 1 und x(n) = 0 für n ≠ 0. Berechnen Sie mit Hilfe von Gleichung (2.2) die Impulsantwort y(n).

Übung 2.7: Geben Sie als Eingangssignal eine Sprungfunktion vor, d.h. x(n) = 1 für n ≥ 0 und x(n) = 0 für n < 0. Berechnen Sie mit Hilfe von Gleichung (2.2) die Sprungantwort y(n).

Bei der Realisierung von Filtern bedeuten die Stützstellen h_k die Filterkoeffizienten. Da direkt nach Gleichung (3.2) implementierte Filter stets eine endliche Anzahl von Filterkoeffizienten haben, ist auch die Impulsantwort endlich. Solche Filter werden als FIR-Filter bezeichnet, wobei FIR auf die endliche Impulsantwort hinweist (Finite Impulse Response).

Rechenwerk für FIR Filter

Für die in Gleichung (3.2) wiederholt auftretenden Additionen und Multiplikationen lässt sich ein spezielles Rechenwerk verwenden, wie in folgender Abbildung gezeigt.

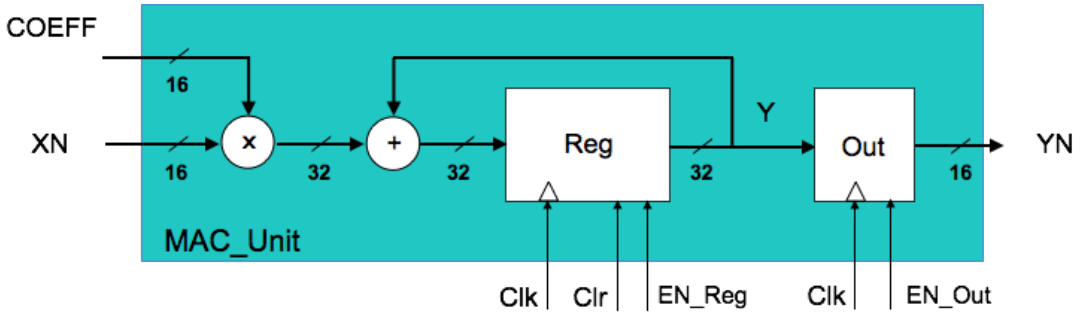


Bild 2.3 Rechenwerk für kumulierte Multiplikationen

Ein Filterkoeffizient h_k st in der Abbildung als COEFF bezeichnet, ein Eingangswert x(n) mit XN, sowie ein berechneter Ausgangswert y(n) bzw. ein Zwischenergebnis mit YN. Mit Hilfe der Signale EN_Reg und EN_Out lässt sich das Rechenwerk steuern. EN_Reg aktiviert das Register Reg, das in diesem Fall als Akkumulator arbeitet: Mit jedem Takt wird eine Multiplikation ausgeführt und das Ergebnis auf das vorher berechnete Ergebnis addiert. Es ergibt sich pro Takt somit $y[n] = h_k \cdot x[n-k] + y[n-1]$. Hierbei sind die Eingangswerte x[n -k] mit den passenden Koeffizienten h_k durch ein geeignetes Steuerwerk bereit zu stellen. Eine Implementierung des speziellen Rechenwerks zeigt folgender HDL-Text.

```
--- MAC Unit for FIR Filters (VHDL)
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity MAC_Unit is
     port (Clk, Clr : in std_logic;     -- clock, Clear MAC register
           EN_Reg   : in std_logic;     -- Enable MAC register
           EN_Out   : in std_logic;     -- Enable output register
           XN : in std_logic_vector (15 downto 0);    -- input sample
           COEFF : in std_logic_vector (15 downto 0); -- filter coeff.
           YN : out std_logic_vector (15 downto 0));  -- output sample
  end MAC_Unit;

architecture RTL of MAC_Unit is

signal Y : signed (31 downto 0);  -- intermediate results

begin

MAC : process (Clk)     -- perform Multiply-ACcumulate operation
  begin
    if rising_edge(Clk) then
      if (Clr = '1') then Y <= (others => '0'); -- clear MAC
      elsif (EN_Reg = '1') then
        Y <= Y + signed(COEFF) * signed(XN);
      end if;
    end if;
end process MAC;

Sample_Out : process (Clk) -- transfer result to output register
  begin
    if rising_edge(Clk) and (EN_Out = '1') then
      YN <= std_logic_vector(Y(31 downto 16));
    end if;
end process Sample_Out;

end RTL;
```

Übung 2.8: Erläutern Sie Funktionen der beiden Prozesse des Rechenwerkes (inklusive der Steuersignale EN_Reg und EN_Out).

Übung 2.9: Skizzieren Sie einen Ablauf (identisch mit den Prozessen eines Testprogramms), mit dem sich das Rechenwerk verwenden lässt, um eine Impulsantwort zu berechnen.

Übung 2.10: Skizzieren Sie einen Ablauf (identisch mit den Prozessen eines Testprogramms), mit dem sich das Rechenwerk verwenden lässt, um eine Sprungantwort zu berechnen.

Festkomma-Arithmetik

Das im obigen HDL-Text beschriebene Rechenwerk arbeitet mit dem Datentyp signed, das als vorzeichenbehaftete ganze Zahl mit einer vorgegebenen Anzahl Bits interpretiert wird. Dieser Datentyp ist zusammen mit arithmetischen Operationen in der Bibliothek IEEE.numeric_std.all definiert. Rechenoperationen im Festkommaformat sind hiermit sehr einfach zu beschreiben, wie das beschriebene Rechenwerk zeigt (siehe Multiplikationen und Additionen).

Hierbei ist zu beachten, dass sich bei der Addition zweier Zahlen das Ergebnis verdoppeln kann. Das Ergebnis der Addition zweier Zahlen benötigt also in aller Regel ein weiteres Bit. Bei der Multiplikation zweier Zahlen verdoppelt sich die Zahl der benötigten Bits. Diese Regeln folgen im binären Zahlensystem der geläufigen Arithmetik im dezimalen Zahlensystem.

Die arithmetischen Operationen wie Addition bzw. Multiplikation sind übrigens nicht auf ganze Zahlen beschränkt, sondern gelten generell für Zahlen mit einem fest definierten Dezimalpunkt bzw. Komma, den sogenannten Festkommazahlen. Das Komma (bzw. der Dezimalpunkt im 10-er System) ist hierbei eine reine Frage der Interpretation und hat keinerlei Auswirkung auf die Rechenoperationen bzw. Rechenwerke. Ob man im dezimalen System eine Zahl „200“ als „200,“, „20,0“, „2,00“ bzw. als „0,200“ interpretiert, ist nur eine Vereinbarung.

Die gleiche Interpretation gilt im binären Zahlenformat, wobei hierbei allerdings die dezimalen Zahlen in binäre Zahlen umzuwandeln sind. Durch fortgesetzte Division durch 2 und Mitschreiben des jeweiligen Rests (0 oder 1) erhält man aus 100 beispielsweise die binäre Zahl 0110 0100. Die folgende Abbildung zeigt das Rechenschema (beginnend mit der kleinsten binären Stelle). Je nach Datentyp, wird das höchstwertige Bit (MSB) entweder als Zahl (Datentyp Unsigned) oder als Vorzeichen (Datentyp Signed) interpretiert.

Dezimal		100								
Binär	Unsigned	128	64	32	16	8	4	2	1	
	Signed	VZ	64	32	16	8	4	2	1	
x/2		0	0	1	3	6	12	25	50	
Rest		0	1	1	0	0	1	0	0	
Test:	Binär -> Dezimal	0	64	32	0	0	4	0	0	100

Dezimal		0,5625							
Binär	Signed	VZ	0,5	0,25	0,125	0,0625	0,03125	0,015625	0,0078125
x/2			0	0,0078125	0,015625	0,03125	0,0703125	0,140625	0,28125
Rest			0,0078125	0	0	0,0078125	0	0	0
	Signed	0	1	0	0	1	0	0	0
Test:	Binär -> Dezimal		0,5	0	0	0,0625	0	0	0	0,5625

Bild 2.4 Umwandlung dezimal nach binär

Die Interpretation des Festkommas ist von dieser Darstellung unabhängig: In der Abbildung oben wurde die kleinste binäre Stelle als 1 interpretiert, das Komma steht somit rechts von der kleinsten Stelle. Stellt man das Komma vor die binäre Zahl, wäre die kleinste binäre Stelle als 1/128

zu interpretieren. In dieser Darstellung sind dann Zahlen darstellbar, deren Betrag kleiner als 1 ist. Die dezimale Zahl 0,5625 wird bei dieser Sichtweise in die binäre Zahl 0100 1000 gewandelt. Die Wandlung zurück ins dezimale Format erhält man wiederum als 1* 64/128 + 1* 8/128 = 72/128 = 0,5625.

Das Festkomma gilt als Vereinbarung, wie die dezimalen bzw. binären Stellen einer Zahl zu interpretieren sind. In die binäre Kodierung der Zahl muss diese Vereinbarung nicht übernommen werden, hierfür ist also kein eigenes Bit erforderlich. Allerdings erfordert das Vorzeichen beim Datentyp signed ein eigenes Vorzeichenbit (erstes Bit, MSB). Die Darstellung vorzeichenbehafteter binärer Zahlen folgt dem Zweierkomplement.

Übung 2.11: Erstellen Sie mit Hilfe Ihres Programms zur Tabellenkalkulation eine Tabelle zur Umrechnung dezimaler Festkommazahlen in binäre 16-Bit Festkommazahlen vom Typ. Das Komma soll hierbei unmittelbar auf das Vorzeichen der 16-Bit Zahl folgen. Wandeln Sie zum Test die Zahl 0,5625 in das 16-Bit Format. Hinweis: Verwenden Sie die Funktion Rest zur Berechnung des Restes der Division. Verwenden Sie zunächst nur positive Zahlen.

Berechnung der Filterkoeffizienten

In vorliegenden Manuskript stehen Rechenwerke in HDL im Vordergrund. Die Berechnung von Filterkoeffizienten ist Teil der Theorie zeitdiskreter Systeme. An dieser Stelle erfolgt daher nur eine Plausibilisierung der Filterkoeffizienten durch den Vergleich mit einem realen System. Als Beispiel wird hierfür ein mit diskreten Bauteilen aufgebautes LC-Filter untersucht, wie in folgender Abbildung dargestellt.

Hierbei lässt sich die Ausgangsspannung $u_2(t)$ in Abhängigkeit der Eingangsspannung $u_1(t)$ in Form einer Differentialgleichung wie folgt beschreiben.

$$u_2(t) = u_1(t) - LC\, d^2u_2(t)/dt^2 - (L/R)\, du_2(t)/dt \qquad (2.3)$$

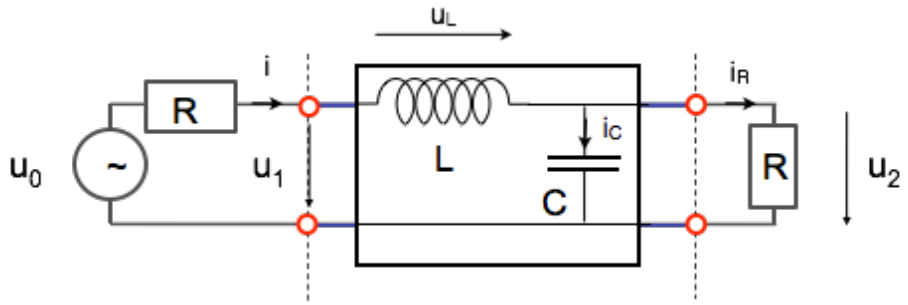


Bild 2.5 LC-Filter

Übung 2.12: Leiten Sie die Differentialgleichung (2.3) her. Hinweis: Verwenden Sie hierzu die Beziehungen $u_L(t) = L\, di(t)/dt$ und $i_C(t) = C\, du_2(t)/dt$.

Gleichung (2.3) lässt sich so interpretieren, dass die Ausgangsspannung $u_2(t)$ der eingeprägten Eingangsspannung $u_1(t)$ folgt, bis auf die Ausdrücke mit den Ableitungen von $u_2(t)$. Da die Ableitungen nur auf Veränderungen reagieren, klingen diese Beiträge bei einer konstant verlaufenden Eingangssignal ab. Gibt man als Eingangsspannung $u_1(t)$ beispielsweise eine Sprungfunktion vor ($u_1(t) = \hat{u}$ für t

≥ 0 und $u_1(t) = 0$ für $n < 0$), so wird sich das Ausgangssignal $u_2(t)$ nach Abklingen der transienten Beiträge ebenfalls auf den konstanten Wert û einschwingen.

Der transiente Anteil des Ausgangssignals folgt hierbei einer gedämpften Schwingung. Für $u_1(t) = 0$ beschreibt die Differentialgleichung (2.3) einen gedämpften Schwingkreis und entspricht der folgender allgemeinen Form.

$$\ddot{y}(t) + 2\delta\,\dot{y}(t) + \omega_0^2\,y(t) = 0 \qquad (2.4)$$

Die Konstante δ wird hierbei als Abklingkoeffizient bezeichnet. Mit ω_0 ist die Eigenfrequenz des ungedämpften Schwingkreises bezeichnet. Beide Parameter lassen sich aus den Werten von R, L, und C berechnen. Die Gleichung der gedämpften Schwingung gemäß (2.4) wird durch Funktionen folgenden Typs gelöst:

$$y(t) = \hat{y}_0\, e^{-\delta t} \sin(\phi) = \hat{y}_0\, e^{-\delta t} \sin(\omega_d t + \phi_0) \qquad (2.5)$$

Zusätzlich zum periodischen Anteil sin(ϕ) enthält diese Funktion einen mit der Zeit exponentiell abklingenden Anteil $e^{-\delta t}$. Die Frequenz der Schwingung ist ebenfalls gedämpft. Die Kreisfrequenz der gedämpften Schwingung in Gleichung (2.5) mit ω_d bezeichnet (Eigenfrequenz der gedämpften Schwingung). Der gedämpfte Schwingkreis schwingt mit der Frequenz $\omega_d^2 = \omega_0^2 - \delta^2$. Die Dauer einer Periode beträgt folglich $T_d = 2\pi / \omega_d = 2\pi / \sqrt{(\omega_0^2 - \delta^2)}$. Die Sprungantwort hat daher qualitativ den in folgender Abbildung gezeigten Verlauf.

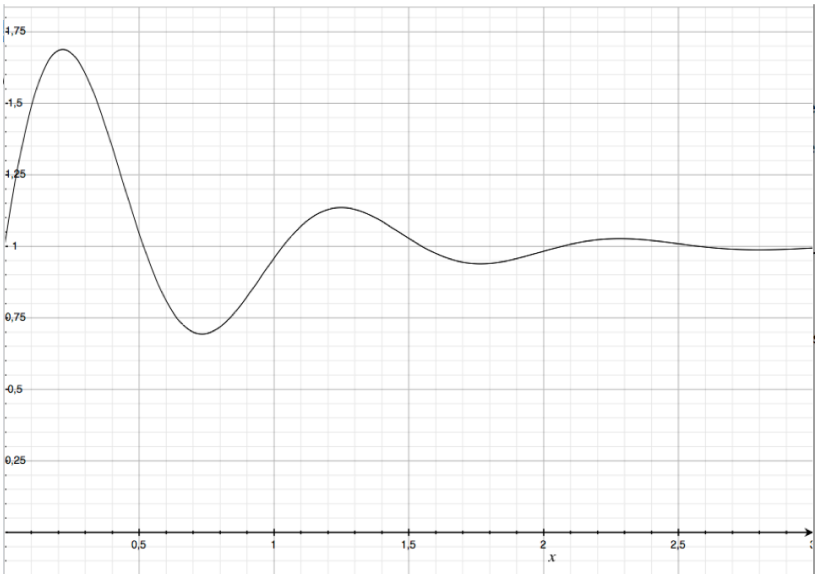


Bild 2.5 Sprungantwort des LC-Filters

Bei einem zeitdiskreten System erhält man die Eingangssignale und Ausgangssignale durch Abtastung mit einem festen Zeitintervall. Diskretisiert man den oben gezeigten qualitativen Verlauf der Sprungantwort, so ergeben sich die in folgender Abbildung gezeigten Werte. Hierbei wurde das Abtastintervall völlig willkürlich gewählt. Die obere Kurve entspricht der Sprungantwort.

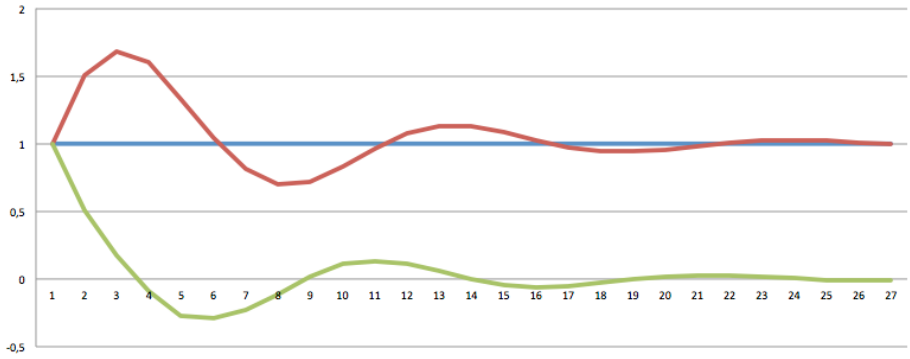


Bild 2.6 Sprungantwort und Impulsantwort des zeitdiskreten Systems

Aus der Sprungantwort lässt sich in einem zeitdiskreten System mit geringem Aufwand die Impulsantwort rekonstruieren. Mit Blick auf die Gleichungen (2.1) und (2.2), sowie die Übungen 2.6 und 2.7 erkennt man folgenden Zusammenhang: $y_{Imp}(i) = y_{Sprung}(i) - y_{Sprung}(i-1)$. Die Impulsantwort ist in der unteren Kurve in der Abbildung gezeigt.

Die Stützstellen der Impulsantwort entsprechen den Filterkoeffizienten eines FIR-Filters. Mit Hilfe der Faltungssumme lässt sich aus diesen Filterkoeffizienten die Reaktion des Systems auf beliebige Eingangssignale berechnen. Bei einem System, das das Eingangssignal nicht verstärkt, ist die Summe aller Filterkoeffizienten = 1 (bzw. der Betrag der Summe). Auch dieses Verhalten zeigt das oben genannte Beispiel: Da sich die Sprungantwort auf den Wert 1 einschwingt, addieren sich die Filterkoeffizienten zu diesem Wert.

Ein praktisches Beispiel

Mit Hilfe der im vorausgegangenen Abschnitt gezeigten Filterkoeffizienten soll mit Hilfe des eingangs beschriebenen Rechenwerkes die Sprungantwort des Systems berechnet werden. Für die Sprungantwort ist ab der ersten Stützstelle das Eingangssignal konstant (=1). Daher müssen dem Rechenwerk nur noch die Filterkoeffizienten zugeführt werden. Durch das Steuersignal EN_Reg kumuliert das Rechenwerk die Ergebnisse selbsttätig gemäß dem Verlauf der Sprungantwort (siehe Übung 2.7). Folgender Programmtext zeigt den grundsätzlichen Ablauf.

```
-- run tests
MAC : process -- calculate sample MAC
begin

  -- clear MAC
  T_Clr <= '1';
  T_Clk <= ,0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
  T_Clr <= '0';
  T_Clk <= ,0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

  -- operate MAC for sample signal (step response)
  T_EN_Reg <= '1';               -- Enable MAC
  T_EN_Out <= '1';               -- Enable Output register
  T_XN <= "0111 1111 1111 1111"; -- set input to 1 for step response

  T_COEFF <= "0111 1111 1111 1111";      -- h0 (one clock cycle)
```

```
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

  T_COEFF <= "0100 0001 0000 0110";      -- h1 (one clock cycle)
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

  T_COEFF <= "0001 0110 1010 0111";      -- h2 (one clock cycle)
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

  T_COEFF <= "1101 1101 1011 0011";      -- h3 (one clock cycle)
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

  . . . continue with next coefficient

  T_COEFF <= "1111 1111 1011 1111";      -- hlast (one clock cycle)
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

  -- stop accumulation and wait
  T_EN_Reg <= '0';
  T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
  wait;

end process MAC
```

Zur Berechnung der Filterkoeffizienten wird folgende Tabelle verwendet. Die Umrechnung der im dezimalen Format gegebenen Filterkoeffizienten in das binäre Format mit Vorzeichen für den Datentyp signed erfolgt mit der im vorausgegangenen Abschnitt beschriebenen Methode (siehe Übung 2.11). Hierbei kann man sich mit guter Näherung auf die ersten 13 Stützstellen beschränken (mit Index 0 bis 12).

Sprungantwort und Impulsantwort

Index	Sprungfkt	Sprungantwort	Impulsantwort	Binärformat 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1,508	0,508	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	0
2	1	1,685	0,177	0	0	0	1	0	1	1	0	1	0	1	0	0	1	1	1
3	1	1,599	-0,086	1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	0
4	1	1,331	-0,268	1	1	0	1	1	1	0	1	1	0	1	1	0	0	1	1
5	1	1,042	-0,289	1	1	0	1	1	0	1	1	0	0	1	1	0	0	1	1
6	1	0,813	-0,229	1	1	1	0	0	0	1	0	1	0	1	1	0	0	0	1
7	1	0,7	-0,113	1	1	1	1	0	0	0	1	1	0	0	0	1	0	0	1
8	1	0,72	0,02	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	1
9	1	0,829	0,109	0	0	0	0	1	1	0	1	1	1	1	1	0	0	1	1
10	1	0,963	0,134	0	0	0	1	0	0	0	1	0	0	1	0	0	1	1	1
11	1	1,073	0,11	0	0	0	0	1	1	1	0	0	0	0	1	0	1	0	0
12	1	1,131	0,058	0	0	0	0	0	1	1	1	0	1	1	0	1	1	0	1
13	1	1,129	-0,002	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1

Bild 2.7 Stützstellen der Impulsantwort als Filterkoeffizienten

Ein Testlauf des Programms die Sprungantwort an den Zwischenergebnissen im Zeitdiagramm anschauen lässt, siehe folgende Abbildung. Das Eingangssignal verbleibt als Sprungfunktion auf konstanten Niveau. Da zuvor alle internen Register gelöscht wurden, startet die Berechnung ab der ersten Stützstelle der Eingangsfunktion. Die durch das Testprogramm jeweils vorgegebenen Filterkoeffizienten entsprechen hierbei in Ihrer Reihenfolge der Impulsantwort des Systems.

Mit der Vorgabe EN_Reg kumuliert das Rechenwerk die Stützstellen wieder zur Sprungfunktion, was der Verlauf des Signals im internen Register Y in der Abbildung ganz unten zeigt. Die Ausgabe auf das Ausgangssignal YN wird durch das Steuersignal EN_Out vorgegeben und erfolgt einen Takt später.

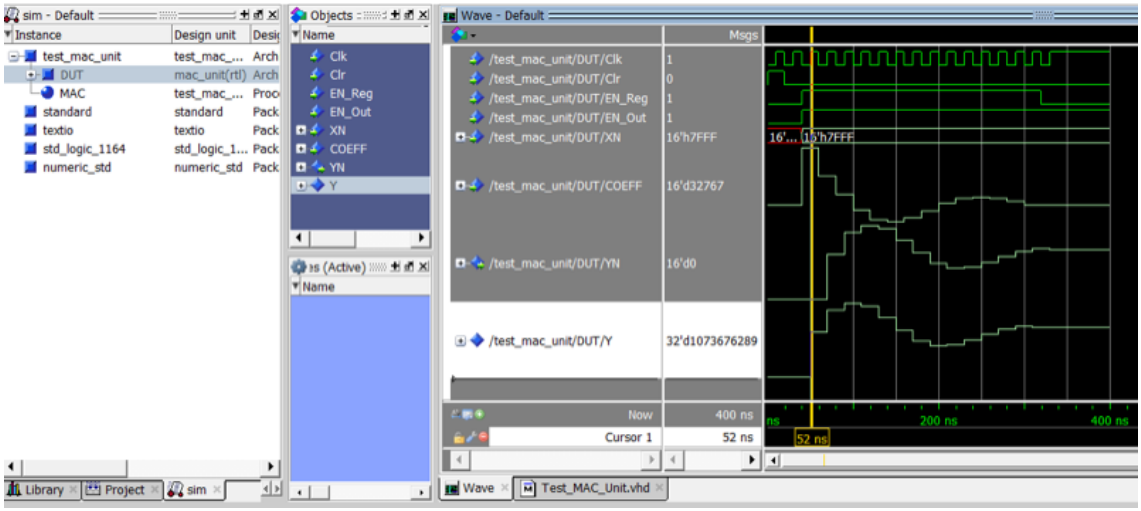


Bild 2.8 Berechnung der Sprungantwort mit Hilfe des Testprogramms

Übung 2.13: Erstellen Sie ein Testprogramm für das Rechenwerk, das die Sprungantwort mit den im oben genannten Beispiel gegebenen Filterkoeffizienten mit 12-Bit Wortbreite für die Sprungfunktion und die Filterkoeffizienten berechnet. Testen Sie das Programm am Simulator.

Übung 2.14: Erstellen Sie mit Hilfe des Testprogramms ein Filter, das mehrere Stützstellen des Eingangssignals mittelt, das also eine Impulsantwort in Form eines Rechtecks hat. Durch diese Mittelung ergibt sich ebenfalls eine Tiefpasswirkung. Wählen Sie für das Filter beispielsweise 5 Stützstellen und skalieren Sie die Filterkoeffizienten so, dass sich die Sprungantwort auf die Höhe des Eingangssignals einschwingt (Verstärkung = 1). Berechnen Sie die Sprungantwort.

FIR Filter als Systemmodell

FIR bilden das Verhalten eines Systems durch Reproduktion der Impulsantwort nach. Im Vergleich zum realen physikalischen System, wie der in der vorausgegangenen Abschnitten gezeigten LC-Schaltung, erschient die Nachbildung jedoch wenig effizient. Für die Beschreibung des realen Systems genügen sehr wenige Parameter, nämlich die Werte von R, L und C. Im Vergleich hierzu benötigt das FIR-Filter selbst für eine recht grobe Näherung bereits mehr als 12 Systemparameter (Filterkoeffizienten).

Die Ursache hierfür liegt darin, dass mit Hilfe der Differentialgleichung (2.3) beschriebene reale System Rückkopplungen verwendet. Der FIR-Algorithmus gemäß Gleichung (2.1) bzw. Gleichung (2.2) berechnet die Stützstellen des Ausgangssignals jedoch ausschließlich aus den Stützstellen des Eingangssignals mit Hilfe der Filterkoeffizienten. Somit ist für jede Stützstelle der Impulsantwort ein Filterkoeffizient erforderlich. Mit einem Filteralgorithmus folgender Form ließe sich die Zahl der erforderlichen Parameter zur Nachbildung des Systems deutlich reduzieren.

$$y[n] = \sum a_k \cdot x[n-k] \quad - \sum b_l \cdot y[n-l] \qquad (2.6)$$

Während der erste Teil mit den Koeffizienten a_k der Struktur des FIR-Filters, koppelt der zweite Teil des Algorithmus vergangene Werte des Ausgangssignals y[n–l] mit Hilfe der Koeffizienten b_l zurück. Die Wirkungsweise der Gleichung verdeutlicht die folgende vereinfachte Form.

$$y[n] = a_0 \cdot x[n] \; - \; b_1 \cdot y[n-1] \tag{2.7}$$

Wählt man beispielsweise $a_0 = 1$ und $b_1 = 0{,}8$, so klingt die Impulsantwort des Filters ohne Beschränkung sehr lange aus, unabhängig von der Anzahl der Filterkoeffizienten. Wegen der im Vergleich zum FIR-Filter unbegrenzten Dauer der Impulsantwort werden solche Algorithmen auch als IIR Filter bezeichnet (wobei IIR für Infinite Impulse Response steht).

Die Nachbildung von Systemen mit rückgekoppelten Algorithmen erfordert belastbare Kenntnisse der Systemtheorie, sowohl was die Modellierung betrifft, als auch die Stabilität der Modelle (um ein Aufschwingen zu verhindern). Weiterhin setzt auch die Implementierung der Algorithmen in Festkomma-Arithmetik belastbare Kenntnisse des numerischen Verhaltens voraus. Davon abgesehen stellt die Implementierung in HDL keine Anforderungen, die über die Implementierung der FIR-Algorithmen hinaus gehen.

2.3. Steuerwerk für das FIR Filter

Das in Abschnitt 2.2 gezeigte Rechenwerk soll nun um ein geeignetes Steuerwerk erweitert werden. Hierbei sollen die Filterkoeffizienten innerhalb des Filters gespeichert werden. Von aussen werden die abgetasteten Eingangssignale zugeführt. Das Filter soll den Faltungsalgorithmus gemäß Gleichung (2.1) für alle vorhandenen Koeffizienten eigenständig durchführen. Der Ablauf wäre dann wie folgt: (1) den neuen Eingangswert x(n) einlesen, (2) die Faltungssumme ausführen, (3) den aktuellen Ausgangswert y(n) ausgeben.

Da der Faltungsalgorithmus auch die vergangenen Eingangssignale für die gegebene Anzahl von Filterkoeffizienten benötigt, sind die eingelesenen Eingangssignale ebenfalls zu speichern. Demnach besitzt das FIR-Filter folgende Komponenten: (1) das Rechenwerk (MAC_Unit), (2) einen Speicher für Koeffizienten (COEFF_RAM), (3) einen Speicher für Eingangswerte (X_RAM), (4) ein Steuerwerk (FSM). Folgendes Blockschaltbild zeigt einen Entwurf des Filters.

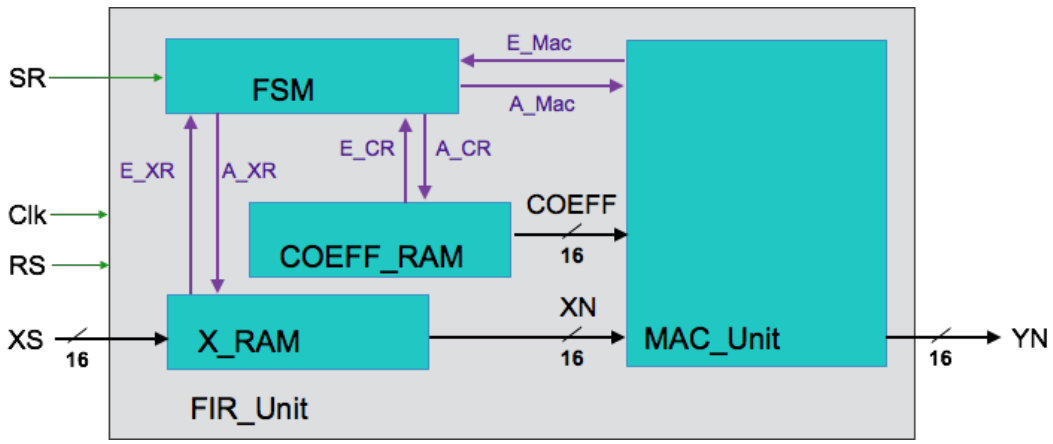


Bild 2.9 Blockschaltbild des Filters mit Rechenwerk und Steuerwerk

Dem Filter wird ein abgetasteter Wert des Eingangssignals XS zugeführt. Mit Hilfe der Koeffizienten berechnet das Filter hieraus den aktuellen Abtastwert des Ausgangssignals YN. Das

Steuersignal SR (Sample Ready) zeigt dem Filter an, dass ein neuer Abtastwert als Eingangssignal vorhanden ist. Ausserdem sind als Eingänge ein Takt (Clk), sowie ein Reset (RS) vorhanden. Die Weiterführung dieser Signale an die einzelnen Komponenten ist der Übersichtlichkeit halber nicht dargestellt.

Die Steuerung erfolgt durch einen Zustandsautomaten (FSM), der Eingänge und Ausgänge zu allen übrigen Komponenten besitzt. Diese Signale sind noch nicht näher spezifiziert, bis auf die bereits vorhandenen Steuersignale der MAC-Unit. Diese Steuersignale (EN_Reg, EN_Out und Clr) sind in der Abbildung unter dem Vektor A_MAC zusammengefasst. Für den eingangs beschriebenen Ablauf sind innerhalb der einzelnen Komponenten Zähler vorzusehen, die ein taktsynchrones Arbeiten ermöglichen. Die Anzahl der Filterkoeffizienten wird hierfür fest vorgegeben, beispielsweise für 32 Koeffizienten. Hieraus folgt dann die Dimensionierung der Speicher und Zähler.

Aufbau der Speicher

Die Speicher werden als Arrays definiert, die das Synthesewerkzeug je nach der auf dem Zielbaustein vorhandenen Infrastruktur dann entweder in Tabellen (Look-up Tables) oder Speicherzellen (RAM) abbildet. Der grundsätzliche Aufbau wäre wie folgt. Der Speicher stellt ein Array von Worten dar. Da Arrays mit dem Datentyp Integer adressiert werden, ist für den Zugriff auf die gegebene Speicheradresse eine Typenkonversion von std_logic_vector nach Integer erforderlich.

```
--- RAM Module

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity RAM is

       generic (L_BITS : natural;   -- L bits wide (Address Space)
                M_BITS : natural);  -- M bits wide (Word Width)

       port ( ADDR : in  std_logic_vector(L_BITS-1 downto 0);
              DATA : out std_logic_vector(M_BITS-1 downto 0));

end RAM;

architecture RTL of RAM is

 -- array of 2**L samples, each M bits wide
 type RAM_array is array(0 to 2**L_BITS - 1) of
              std_logic_vector(M_BITS - 1 downto 0);

       -- fill array with 2**M sample values
       signal memory : RAM_array := ( x"0." -- fill in values here );

       -- read access: y = data(address)
       begin
       DATA <= memory(to_integer(unsigned(ADDR)));
```

```
end RTL;
```

Das Beispiel beschreibt einen Speicher, der statisch beschrieben und nur gelesen wird (also ein ROM, wie es für die Koeffizienten benötigt wird). Für ein RAM wäre eine Schreibfunktion zu ergänzen, d.h. die Zuweisung von Eingangsdaten an die Speicheradresse. Für ein Filter mit 16 Stützstellen wären L = 4 Adressleitungen ausreichend. Da die Wortbreite 16 Bit betragen soll, wäre M = 16. Diese Angaben gelten sowohl für den Koeffizientenspeicher als auch für den Speicher der Eingangswerte.

Folgende zusätzlichen Anforderungen bestehen: (1) Beide Speicher sollen takt-synchron arbeiten, d.h. die Ausgabe der Daten erfolgt einen Takt nach Vorgabe der Adressen, (2) der Koeffizientenspeicher soll zusätzlich einen Zähler erhalten, mit dem sich die Bereitstellung passender Paare von Koeffizienten und Eingangswerte herstellen lässt, (3) der Speicher für die Eingangssignale muss mit Null initialisieren lassen, (4) im Speicher für die Eingangs-signale werden Werte, die mehr als 32 Stützstellen in der Vergangenheit liegen, mit jeweils aktuellen Werten fortlaufend überschrieben.

Aufbau des Speichers für die Eingangswerte

Der Speicher für die Eingangswerte soll als Dual-Ported RAM ausgeführt werden, d.h. mit separaten Adressen zum Lesen und zum Schreiben von Daten. Außerdem erhält der Speicher ein Steuersignal zum Schreiben (Write-Enable), das unnötige Schreiboperationen beim Auslesen des Speichers vermeiden soll.

```
--- X_RAM Module (VHDL)

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity X_RAM is

       generic (L_BITS : natural;   -- L bits wide (Address Space)
                M_BITS : natural);  -- M bits wide (Word Width)

       port ( Clk, WR_EN : in std_logic; -- Clock, Write_Enable
              RADDR : in  std_logic_vector(L_BITS-1 downto 0);
              DTOUT : out std_logic_vector(M_BITS-1 downto 0);
              WADDR : in  std_logic_vector(L_BITS-1 downto 0);
              DATIN : in  std_logic_vector(M_BITS-1 downto 0));

end X_RAM;

architecture RTL of X_RAM is

 -- array of 2**L samples, each M bits wide
 type RAM_array is array(0 to 2**L_BITS - 1) of
              std_logic_vector(M_BITS - 1 downto 0);

       -- instantiate memory object of X_RAM
       signal memory : RAM_array;
```

```
                -- to initialize use := (others => x"0000");

    begin
          -- read & write process
          process (Clk) begin
            if (rising_edge(Clk)) then
               if (WR_EN = '1') then            -- write to WADDR
                   memory(to_integer(unsigned(WADDR))) <= DATIN;
          end if;                            -- read from RADDR
          DTOUT <= memory(to_integer(unsigned(RADDR)));
       end if;
     end process;

   end RTL;
```

Je nach Vorgabe des Write_Enable Signals (WR_EN) führt der Speicher bei einer steigenden Taktflanke eine Schreiboperation und eine Leseoperation durch, bzw. nur eine Leseoperation. Beim Schreiben und Lesen auf die gleiche Adresse benötigt das Speicher zwei Taktflanken, da mit der ersten Taktflanke noch der alte Wert vorhanden ist, der zu diesem Zeitpunkt erst aktualisiert wird. Folgendes Testprogramm zeigt die Arbeitsweise des Speichermoduls.

```
--- Testbench für X_RAM (test cases only)
-- write
T_WR_EN <= '1';
T_WADDR <= x"1";
T_DATIN <= x"BBBB";

T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

-- write & read simultaneously at different addresses
T_WR_EN <= '1';
T_WADDR <= x"2";
T_DATIN <= x"EEEE";
T_RADDR <= x"1";

T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;

-- write & read simultaneously at the same address
T_WR_EN <= '1';
T_WADDR <= x"1";
T_DATIN <= x"CCCC";
T_RADDR <= x"1";

T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
```

Das Zeitdiagramm zeigt, das simultanes Schreiben und Lesen an unterschiedlichen Adressen mit der nächsten steigenden Taktflanke erfolgt. Simultanes Schreiben und Lesen der gleichen Adresse benötigt erwartungsgemäß zwei Taktflanken.

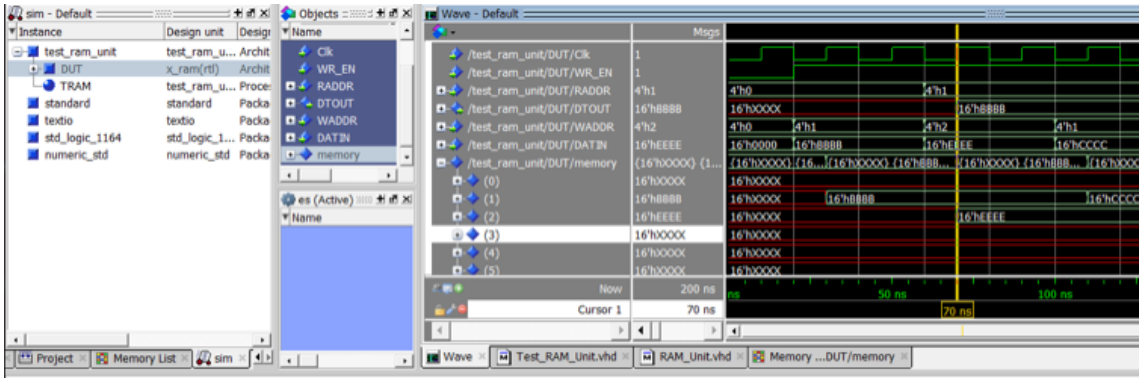


Bild 2.10 Zeitverhalten des X_RAM

Aufbau des Speichers für Koeffizienten

Der Speicher für die Koeffizienten kann als reines ROM ausgeführt werden. Wenn die Koeffizienten fest vorgegeben werden können, kann die Vorgabe direkt im Programmtext erfolgen, wie in folgenden HDL-Text beschrieben.

```
--- COEFF_ROM Module (VHDL)

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity COEFF_ROM is

        generic (L_BITS : natural;    -- L bits wide (Address Space)
                 M_BITS : natural);   -- M bits wide (Word Width)

        port ( Clk    : in std_logic; -- Clock
               CADDR : in  std_logic_vector(L_BITS-1 downto 0);
               CFOUT : out std_logic_vector(M_BITS-1 downto 0));

end COEFF_ROM;

architecture RTL of COEFF_ROM is

 -- array of 2**L samples, each M bits wide
 type ROM_array is array(0 to 2**L_BITS - 1) of
               std_logic_vector(M_BITS - 1 downto 0);

        -- instantiate memory object of COEFF_ROM
        signal cmemory : ROM_array   := (
          0 => "0111111111111111",  -- h0
          1 => "0100000100000110",  -- h1
```

```
              2 => "0001011010100111",  -- h2
              3 => "1111010011111110",  -- h3
              4 => "1101110110110011",  -- h4
              5 => "1101101100110011",  -- h5
              6 => "1110001010110001",  -- h6
              7 => "1111000110001001",  -- h7
              8 => "0000001010001111",  -- h8
              9 => "0000110111110011",  -- h9
             10 => "0001000100100111",  -- h10
             11 => "0000111000010100",  -- h11
             12 => "0000011101101101",  -- h12
             13 => "1111111110111111",  -- h13
             14 => "0000000000000000",  -- h14
             15 => "0000000000000000"); -- h15

     begin
           -- read process
           process (Clk) begin
             if (rising_edge(Clk)) then
           CFOUT <= cmemory(to_integer(unsigned(CADDR)));
        end if;
      end process;

    end RTL;
```

Zähler zur Berechnung der Faltungssumme

Es verbleibt noch die Steuerung der Adressen für die Koeffizienten und die passenden Stützstellen der Eingangsfunktion aus dem X_RAM. Mit Blick auf die Faltungssumme gemäß Gleichung (2.1) bzw. (2.2) wird hierfür ein Zähler benötigt, der einerseits die Koeffizienten von Index 0 bis 15 adressiert, andererseits die Stützstellen beginnend vom aktuellen Eingangswert x(n) abwärts bis x(n-15) adressiert. Diese Aufgabe kann durch folgende Beschreibung gelöst werden.

```
signal CCOUNT : unsigned ((LBITS -1) downto 0); -- starting from x"0";
signal XCOUNT : unsigned ((LBITS -1) downto 0); -- starting from WADDR

counter : process (Clk)
    begin
       if rising_edge(Clk) then
           CCOUNT <= CCOUNT + 1;
           XCOUNT <= XCOUNT - 1;
       end if;
       CADDR <= CCOUNT; -- coefficient hi read address
       RADDR <= XCOUNT; -- sample x(n-i)read address
end process counter;
```

Für 16 Koeffizienten sind beide Zähler 4-Bit-Zähler mit automatischem Überlauf bzw. Unterlauf. Der Zähler für die Eingangswerte startet bei der letzten beschriebenen Speicheradresse, die den aktuellen Wert von x(n) enthält und zählt von dort aus abwärts. Da das X-RAM ebenfalls als

Ringpuffer organisiert ist, erreicht dieser Zähler von der aktuellen Stützstelle aus die vergangenen 15 Werte. In diesem Fall bietet es sich an, die beiden Speicher einem Modul unterzuordnen, das die Zähler enthält und die Durchführung einer Faltungssumme eigenständig übernimmt.

Die Speicher mit den Zählern und ihrer Steuerlogik werden im folgenden zu einem Modul zusammengefasst (Memory-Unit), wie in der folgenden Abbildung gezeigt. Hierbei übernimmt eine Steuerlogik im Datenpfad die Aufgabe, das Schreiben und Lesen der beiden Speicher zu organisieren und zum Zustandsautomaten zu kommunizieren. Der Zustandsautomat steuert bei dieser Vorgabe jede einzelne Transaktion zum Lesen beider Speicher, bzw. zum Schreiben des Speichers für die Eingangswerte.

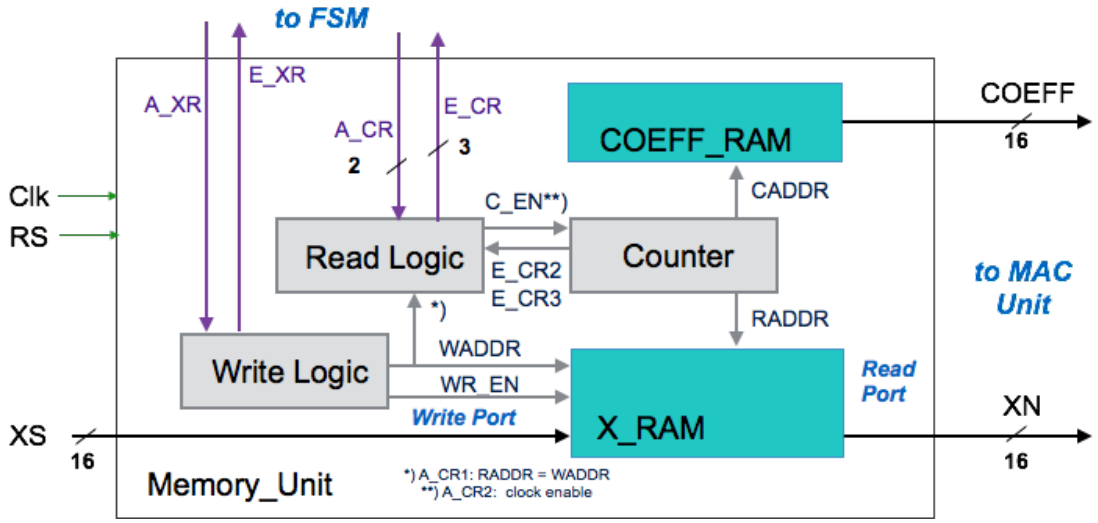


Bild 2.11 Steuerung im Datenpfad

Die Memory-Einheit führt auf Anweisung des Zustandsautomaten folgende Operationen aus:

(1) Schreiben eines neuen Eingangswertes in das XRAM an die Adresse WADDR. Diese Adresse wird von der Lese-Logik als Startpunkt für das erste Faltungsprodukt $h_0 * x(n)$ verwendet. Für den nächsten Eingangswert Wert wird die Adresse inkrementiert. Zum Schreiben wird das Write-Enable Signal des X_RAMs verwendet. Die Schreiboperation wird vom Zustandsautomaten mit dem Signal A_XR ausgelöst und die Ausführung vom Datenpfad mit E_XR quittiert.

(2) Durchführung der Faltungssumme durch die Memory-Einheit und MAC-Einheit: Der Zustands-automat steuert jeden einzelnen Schritt über seine Eingänge und Ausgänge. Für jeden Koeffizienten wird folgende Transaktion ausgeführt:
- Synchronisation der Adresse (RADDR) zum Auslesen des ersten Wertes x(n) mit der zuletzt beschriebenen Speicheradresse (WADDR). Die Synchronisation wird mit dem Signal A_CR1 ausgelöst und mit E_CR1 quittiert.
- Auslesen eines Paares h_i und x(n-i) aus der Memory-Unit. Hierfür wird als Clock-Enable-Signal (C_EN) die Steueranweisung A_CR2 verwendet, mit E_CR2 als Quittung.
- Multiplikation der bereitgestellten Werte und Akkumulation der Ergebnisse in der MAC-Einheit.
- Die Bereitstellung des letzten Wertepaares signalisiert die Memory-Einheit mit Hilfe ihres Zählerstandes durch das Signal E_CR3.

(3) Ausgabe des Ergebnisses in der MAC-Einheit.

Entwurf des Zustandsautomaten

Mit der oben beschriebenen Abfolge ergibt sich für den Zustandsautomaten folgendes Diagramm. Initialzustand ist das Warten auf den ersten neuen Eingangswert. Die Ankunft eines neuen Wertes wird durch das Ereignis E= E_SR (Sample Ready) gemeldet. Daraufhin wechselt der Automat in den Zustand Z1 und veranlasst das Speichern des Wertes mit Hilfe der Anweisung A_XR. Die Quittung E_XR führt den Automaten in den nächsten Zustand Z2, in dem er die Leseadresse für dien aktuellen Wert im X_RAM aktualisiert. Hierfür wird gemäß der letzten Abbildung Die Anweisung A_CR1 verwendet.

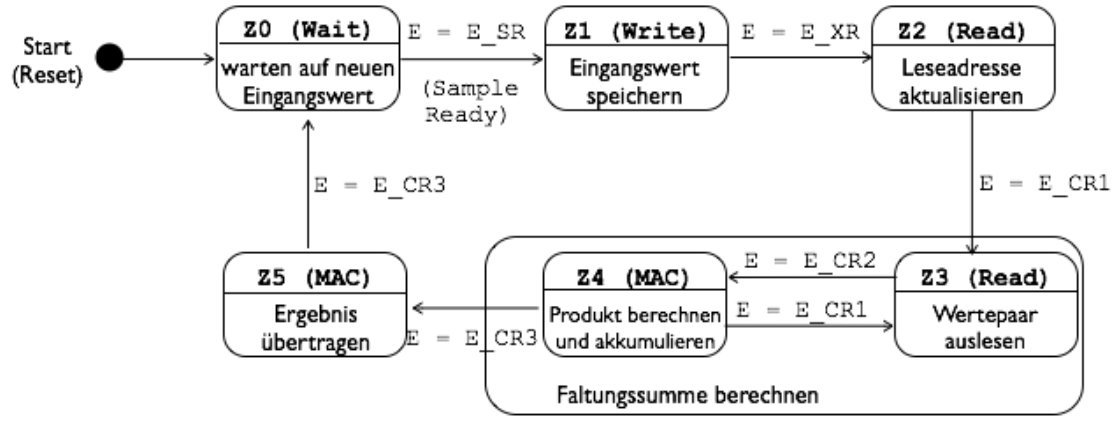


Bild 2.12 Zustandsdiagramm des Automaten

Der Automat steuert darauf den Speicher und das Rechenwerk mit Hilfe von Clock-Enable Signalen, d.h. durch Anschluss der Module an die Taktversorgung und Trennung der Module von der Taktversorgung. Eine besondere Herausforderung ist hierbei die zeitliche Synchronisation. Die Zustände während der Berechnung der Faltungssumme müssen im Taktzyklus wechseln, wo bei jeweils abwechselnd die Speichereinheit und die MAC-Einheit getaktet werden.

Das Ende der Faltungskette signalisiert das Zählwerk der Speichereinheit mit dem Signal E_CR3, das gesetzt wird, sobald der letzte Koeffizient ausgelesen wurde. Nach einer finalen MAC-Operation wechselt der Automat dann in den Zustand Z5, in dem er die MAC-Einheit anweist, das Ergebnis aus dem internen Register auf den Ausgang zu schreiben. Nach der Ausgabe schaltet der wieder Automat in den Wartezustand.

Aufbau der Memory-Unit

Folgender HDL-Text beschreibt die Memory-Einheit. Der Aufbau deklariert die beiden Speicher X_RAM und Koeffizienten-RAM als Komponenten und bindet diese ein. Die Funktion der Schaltung findet sich dann ein den beiden Prozessen counters und write_logic.

```
--- Memory_Unit (VHDL)

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Memory_Unit is
```

```
  -- define constants
  generic (  L_BITS : natural;   -- L bits wide (Address Space)
             M_BITS : natural);  -- M bits wide (Word Width)
  -- ports
  port (Clk, RS : in std_logic;    -- Clock, Reset
    A_XR, A_CR1, A_CR2 : in std_logic;            -- controls from FSM
    E_XR, E_CR1, E_CR2, E_CR3 : out std_logic := '0'; -- events to FSM
    MCOEFF, MXN : out std_logic_vector(M_BITS - 1 downto 0); -- MAC
    XS : in std_logic_vector(M_BITS - 1 downto 0)); -- sample from ADC
end Memory_Unit;

architecture RTL of Memory_Unit is

component X_RAM is
       generic (L_BITS : natural;   -- L bits wide (Address Space)
                M_BITS : natural);  -- M bits wide (Word Width)
       port (Clk, WR_EN : in std_logic; -- Clock, Write_Enable
              RADDR : in  std_logic_vector(L_BITS - 1 downto 0);
              DTOUT : out std_logic_vector(M_BITS - 1 downto 0);
              WADDR : in  std_logic_vector(L_BITS - 1 downto 0);
              DATIN : in  std_logic_vector(M_BITS - 1 downto 0));
end component;

component COEFF_ROM is
       generic (L_BITS : natural;   -- L bits wide (Address Space)
                M_BITS : natural);  -- M bits wide (Word Width)
       port (Clk : in std_logic; -- Clock, Write
             CADDR : in  std_logic_vector(L_BITS-1 downto 0);
             CFOUT : out std_logic_vector(M_BITS-1 downto 0));
end component;

-- memory_unit internal signals
signal WR_EN : std_logic := '0';
-- initialized with x"FF" in order to test synchonization
signal RADDR : std_logic_vector(L_BITS-1 downto 0):= (others => '1');
signal  WADDR : std_logic_vector(L_BITS-1 downto 0):= (others => '1');
signal CADDR : std_logic_vector(L_BITS-1 downto 0):= (others => '1');

signal CCOUNT : unsigned (L_BITS-1 downto 0) := (others => '0');
signal XCOUNT : unsigned (L_BITS-1 downto 0) := (others => '1');
signal WRTCNT : unsigned (L_BITS-1 downto 0) := (others => '0');

begin

-- connect components
XR: X_RAM generic map(L_BITS => L_BITS, M_BITS => M_BITS)
port map (Clk => Clk, WR_EN => WR_EN, RADDR => RADDR, DTOUT => MXN,
WADDR => WADDR, DATIN => XS);

CR: COEFF_ROM generic map(L_BITS => L_BITS, M_BITS => M_BITS)
port map (Clk => Clk, CADDR => CADDR, CFOUT => MCOEFF);
```

```
    -- run processes

    counter : process (Clk, A_CR1, A_CR2) -- address counters & read logic
        begin
          if ((A_CR1 = '1') and rising_edge(Clk)) then
              XCOUNT <= unsigned(WADDR);
              E_CR1 <= '1';
          end if;
          if ((A_CR2 = '1') and rising_edge(Clk)) then
                CADDR <= std_logic_vector(CCOUNT);
                RADDR <= std_logic_vector(XCOUNT);
                CCOUNT <= CCOUNT + 1;
                XCOUNT <= XCOUNT - 1;
                if (CCOUNT = x"F") then E_CR3 <= '1';
                                   else E_CR3 <= '0';
                end if;
                E_CR2 <= '1';
          end if;
    end process counter;

    write_logic : process(Clk, A_XR) -- write logic (returns E_XR)
      begin
      if rising_edge (Clk) then
        if (A_XR = '1') then
          WR_EN <= '1';
          WADDR <= std_logic_vector(WRTCNT);
          WRTCNT <= WRTCNT + 1;
          E_XR <= '1';
        else
          WR_EN <= '0';
          E_XR <= '0';
        end if;
      end if;
    end process write_logic;

    end RTL;
```

Im Vergleich mit dem Blockschaltbild in Abbildung 2.11 umfasst der Prozess counters die beiden Blöcke Read-Logic und Counter. Dieser Prozess läuft parallel zum Prozess write_logic, der den gleichnamigen Block in der Abbildung beschreibt. Der Zustandsautomat sollte zunächst mit Hilfe der Anweisung A_XR mit der nächsten steigenden Taktflanke den Prozess write_logic anstossen, der mit dem Clock Enable Signal WR_EN des Schreibport des X_RAMs aktiviert. Mit der nächsten Taktflanke wird der Wert dann abgespeichert. Der Prozess write_logic inkrementiert die Schreibadresse für den nächsten Eingangswert und quittiert den Vorgang mit dem Signal E_XR.

Der Prozess counter erfüllt folgende Aufgaben: (1) die Aktualisierung der Leseadresse für den Fall, dass ein neuer Eingangswert abgespeichert wurde, (2) das Bereitstellen der Adressen für ein Wertepaar (Eingangswert x(n-i) und Koeffizienten h_i). Die Aktualisierung der Leseadresse erfolgt auf die Anweisung A_CR1. Die Bereitstellung der Adressen erfolgt auf die Anweisung A_CR2. Sobald alle Adressen der Koeffizienten durchgezählt wurden, wird die Meldung E_CR3 abgesetzt, die das Ende der Faltungssumme für den gegebenen Eingangswert kennzeichnet.

Test der Schaltung

Bevor der Zustandsautomat weiter detailliert wird, erfolgt ein Test der Schaltung in einem Top-Modul, welches die Memory-Einheit und die MAC-Einheit enthält. Das Testprogramm verwendet alle Zustandsübergänge, bevor ein Automat eingesetzt wird. Diese Vorgehensweise arbeitet vom Groben ins Feine, indem erst einmal die Steuerung auf ihre Funktion überprüft wird. Folgender HDL-Text beschreibt das Top-Modul mit den Tests.

```
--- FSM and Top Module of FIR Filter (VHDL)

library ieee;
use ieee.std_logic_1164.all;

entity FSM_and_Top_Module is
   generic (L_BITS : natural := 4;   -- L bits wide (Address Space)
            M_BITS : natural := 16); -- M bits wide (Word Width)

  -- later: Clk, RS, SR, XS (sample in) as input ports,
  -- YS (sample out) as output port

end FSM_and_Top_Module ;

architecture RTL of FSM_and_Top_Module  is

component Memory_Unit is
   generic (L_BITS : natural;   -- L bits wide (Address Space)
            M_BITS : natural);  -- M bits wide (Word Width)
   port ( Clk, RS : in std_logic;                    -- Clock, Reset
          A_XR, A_CR1, A_CR2 : in std_logic;         -- from FSM
          E_XR, E_CR1, E_CR2, E_CR3 : out std_logic; -- to FSM
          MCOEFF, MXN : out std_logic_vector(M_BITS - 1 downto 0);
          XS : in std_logic_vector(M_BITS - 1 downto 0));
end component;

component MAC_Unit is
  port (Clk, Clr : in std_logic;     -- clock, Clear MAC register
    EN_Reg, EN_Out : in std_logic;   -- Enable MAC register, output
    XN : in std_logic_vector (15 downto 0);    -- input sample
    COEFF : in std_logic_vector (15 downto 0); -- filter coefficient
    YN : out std_logic_vector (15 downto 0));  -- output sample
end component;

-- Top_Module internal signals
signal Clk: std_logic := '0';                -- later: input port
signal RS : std_logic := '0';                -- later: input port
signal FXN: std_logic_vector (15 downto 0);  -- later: input port
signal FYN: std_logic_vector (15 downto 0);  -- later: output port
signal SR : std_logic := '0';                -- later: input port

signal Clr   : std_logic := '0';
signal EN_Reg: std_logic := '0';
```

```
    signal EN_OUT: std_logic := '0';
    signal FCOEFF: std_logic_vector (15 downto 0);
    signal A_XR, A_CR1, A_CR2 : std_logic := '0'; -- control inputs of FSM
    signal E_XR, E_CR1, E_CR2, E_CR3 : std_logic; -- control events to FSM
    signal FXS : std_logic_vector(M_BITS - 1 downto 0); -- sample from ADC

    begin

    -- connect Memory_Unit and MAC_Unit to Top Module
    MyMem: Memory_Unit generic map(L_BITS => L_BITS, M_BITS => M_BITS)
           port map (Clk=>Clk, RS=>RS, A_XR => A_XR, A_CR1 => A_CR1,
                A_CR2 => A_CR2, E_XR => E_XR, E_CR1=>E_CR1, E_CR2=>E_CR2,
                E_CR3=>E_CR3, MCOEFF=>FCOEFF, MXN=>FXN, XS=>FXS);

    MyMAC: MAC_Unit port map (Clk=>Clk, Clr=>Clr, EN_Reg=>EN_Reg,
                         EN_Out=>EN_Out, XN=>FXN, COEFF=>FCOEFF, YN=>FYN);

    -- run processes

    Test : process -- test MAC and Mem w/o FSM (each step one clock cycle)
    begin

      --- write sample value to memory
      FXS <= x"AAAA";
      A_XR <= '1';
      EN_Out <= '0';
      Clr <= '1'; -- clear MAC unit
      Clk <= '0'; wait for 10 ns; Clk <= '1'; wait for 10 ns;

      -- synchronize XCOUNT/RADDR to WADDR (read from last write address)
      if (E_XR = '1') then
          A_XR <= '0';
          A_CR1 <= '1';
          Clr <= '0'; -- deselect clear MAC unit
      end if;
      Clk <= '0'; wait for 10 ns; Clk <= '1'; wait for 10 ns;

      A_CR1 <='0';

      -- start MAC operations
      L1: for i in 1 to 16 loop
        A_CR2 <= '1';
        EN_Reg <= '0';
        Clk <= '0'; wait for 10 ns; Clk <= '1'; wait for 10 ns;
        if (E_CR2 = '1') then A_CR2 <= '0'; end if;
        EN_Reg <= '1';
        Clk <= '0'; wait for 10 ns; Clk <= '1'; wait for 10 ns;
      end loop L1;

      -- copy result to MAC output register
      if (E_CR3 = '1') then
        A_CR2 <= '0';
```

```
        EN_Reg <= '0';
        EN_Out <= '1';
        Clk <= '0'; wait for 10 ns; Clk <= '1'; wait for 10 ns;
        EN_Out <= '0';

    end if;

      -- more clock cycles to check further acticities
      Clk <= '0'; wait for 10 ns; Clk <= '1'; wait for 10 ns;
      Clk <= '0'; wait for 10 ns; Clk <= '1'; wait for 10 ns;

    wait;

    end process Test;

    end RTL;
```

Die Schaltung bindet folgende Module ein: (1) die Memory-Einheit mit den beiden Speichern und der Logik, (2) die MAC-Einheit. Memory- und MAC-Einheit sind über interne Signale des Top-Moduls miteinander verbunden. Damit die Schaltung als Testprogramm eingesetzt werden kann, sind zunächst die externen Signale des Filters (Clk, RS, SR, XS und YN) noch nicht nach Außen geführt. Folgende Abbildung zeigt den Aufbau der Schaltung insgesamt.

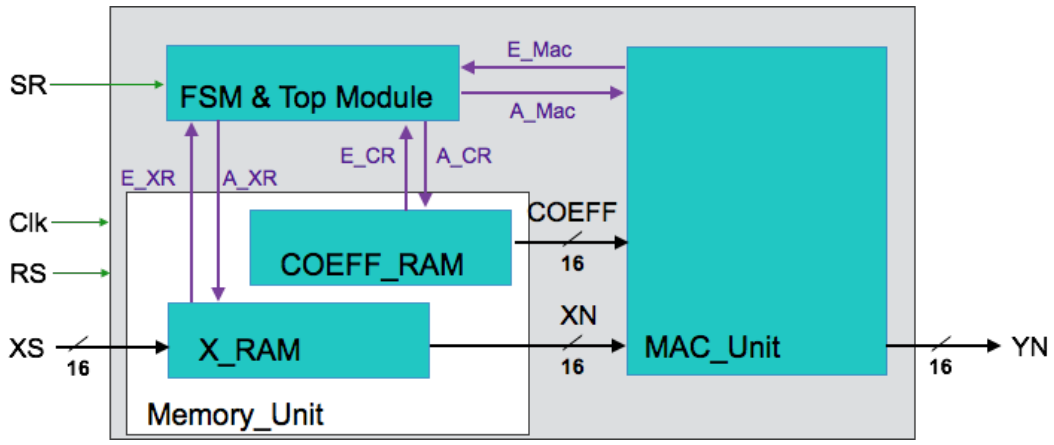


Bild 2.13 Blockschaltbild des Filters

Alle Testschritte sind so ausgeführt, dass nach dem Setzen bzw. nach der Abfrage der Steuersignale jeweils ein Takzyklus folgt. Der Testablauf ist wie folgt: (1) Bereitstellung des ersten Abtastwertes zusammen mit der Anweisung zum Speichern (A_XR) und der Anweisung zum Löschen der MAC-Register (Clr). (2) Synchronisation der Leseadresse des Speichers für die Eingangswerte auf den zuletzt geschriebenen Wert (A_CR1). (3) Schleife zur Bereitstellung eines Wertepaares durch die Memory-Einheit und anschließender Berechnung in der MAC-Einheit (A_CR2, EN_Reg). Hierfür ist jeweils ein eigener Taktzyklus erforderlich, wobei die jeweils nicht aktive Einheit von der Taktversorgung getrennt wird. (4) Nach Abschluss der letzten Berechnung erfolgt die Übertragung des kumulierten Ergebnisses der MAC-Einheit auf den Ausgang YN (Steuersignal: EN_Out).

Damit folgt der Test dem Ablauf des Zustandsdiagramms. Folgende Abbildung zeigt das Ergebnis eines Testlaufs im Simulator. Innerhalb der Bearbeitung der Filterkette erkennt die jeweils abwechselnd aktiven Steuersignale A_CR2 (Bereitstellung eines Wertepaares durch die Memory-Einheit) und EN_Reg (Verarbeitung des Wertepaares in der MAC-Einheit). Am Ende erfolgt die Übergabe des Ergebnisses der MAC-Einheit.

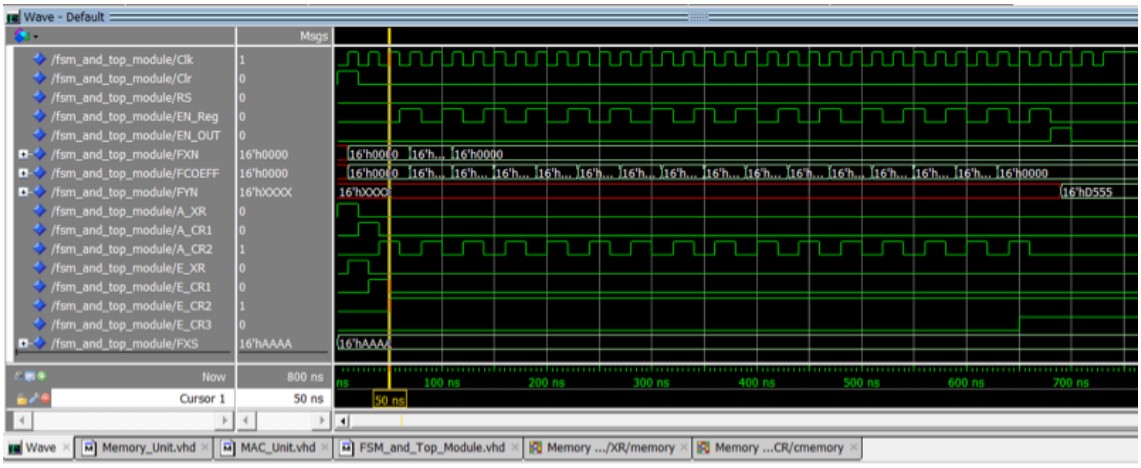


Bild 2.14 Testlauf des Top-Moduls

Der Simulator stellt übrigens auch einen Einblick in die Speicher bereit (Memory-List für das X_RAM und Koeffizienten-Rom). Im Zeitdiagramm erkennt man ausserdem, dass die Koeffizienten und der vorher eingelesen Wert korrekt bereit gestellt werden. Die üblich ist ausser dem Top-Modul auch ein Blick in das Innere der beiden eingebundenen Module (Memory-Unit und MAC-Unit) möglich. Da die MAC-Unit ja bereits im vergangenen Abschnitt getestet wurde, ist speziell die korrekte Initialisierung der Leseadresse in der Memory-Einheit von Interesse. Die folgende Abbildung zeigt die diesbezüglichen Testsignale.

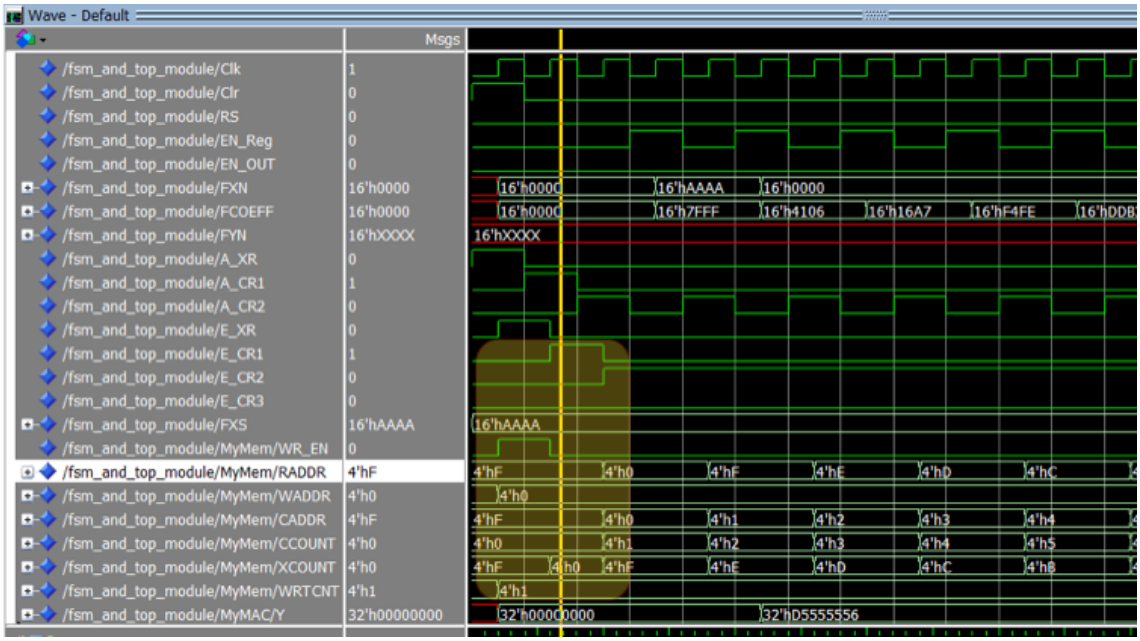


Bild 2.15 Initialisierung der Leseadresse

Man erkennt (1) den mit A_XR ausgelösten Schreibprozess (Wert unter FXS, Schreibadresse WADDR, Zähler WRTCNT der Schreibadresse wird inkrementiert, (2) die durch A_CR1 ausgelöste Synchronisation (Zähler XCOUNT wird an der Curserposition auf den Wert WADDR gesetzt, die Adresse RADDR folgt dann mit dem ersten aktiven Zyklus des X-RAMs, der durch A_CR2 aktiviert wird). Ausserdem erkennt man auch die Werte der jeweils bereit gestellten Filterkoeffizienten.

Zum Takzyklus ist zu bemerken, dass das Testprogramm die Steuersignale zur fallenden Taktflanke generiert (bzw. abfragt). Die Signale stehen dann den anderen Modulen zur folgenden steigenden Taktflanke bereit. Um diesen Fall zu reproduzieren, müsste man den Zustandsautomaten ebenfalls auf fallende Taktflanken programmieren. Arbeitet der Zustandsautomat wie die übrigen Module mit steigenden Taktflanken, so verdoppeln sich die Taktzyklen.

Übung 2.15: Erklären Sie diesen Effekt (Verdopplung der Taktzyklen). Wie verarbeitet eine Simulationsumgebung bzw. ein Synthesewerkzeug die zur jeweiligen Taktflanke zu bedienenden Signale?

Implementierung des Zustandsautomaten

Die Testschritte des Top-Moduls sollen nun durch einen Zustandsautomaten ersetzt werden. Um diesen Automaten dann zu testen, werden die Eingänge und Ausgänge des Top-Moduls (= Eingänge und Ausgänge des FIR-Filters gemäß Abbildung 2.13 als Ports der Entity definiert und von einem übergeordneten Testprogramm als Testeingänge und Ausgänge verwendet.

Übung 2.16: Implementieren Sie den Zustandsautomaten und testen Sie die Schaltung. Hinweis: Ergänzen Sie das Top-Modul aus dem letzten Abschnitt um den Automaten und führen Sie dann die Eingänge und Ausgänge wie in Abbildung 2.9 gezeigt aus der Entity heraus (SR, Clk, RS, XS, YN). Erstellen Sie dann ein Testprogramm für das FIR-Filter.

Übung 2.17: Betreiben Sie das Filter mit insgesamt 32 Koeffizienten. Verwenden Sie hierzu die Koeffizienten aus Übung 2.14 und betreiben Sie das Filter daher mit 12-Bit Wortbreite. Verdoppeln Sie die Anzahl der Koeffizienten dadurch, dass Sie die vorhanden Koeffizienten h_0 bis h_{12} auf die Stützstellen h_{13} bis h_{25} verschieben und an h_{13} so spiegeln, dass $h_{12} = h_{14}$, $h_{11} = h_{15}$, …, $h_0 = h_{25}$ gilt. Nicht benötigte Koeffizienten ab h_{26} werden zu Null gesetzt. Das Filter wird durch die nun symmetrische Impulsantwort linearphasig (d.h. es erneut eine konstante Verzögerung).

Betrachtungen zur Rechenleistung

Da der Automat den Filteralgorithmus für jeden neuen Eingangswert sequentiell abarbeitet, benötigt er pro Stützstelle des Eingangssignals ca 40 Taktzyklen. Ein FPGA mit einer Taktrate von 40 MHz kann somit eine maximale Abtastrate von 1 MSample/s (Megasample pro Sekunde) verarbeiten. Diese Datenrate entspricht einer Grenzfrequenz von ca. 500 kHz. Damit wären Audio-Anwendungen ohne Probleme realisierbar, auch mit deutlich mehr Filterkoeffizienten. Ebenso realisierbar sind anspruchsvolle Aufgaben aus der Regelungstechnik, beispielsweise zur Steuerung von Leistungshalbleitern.

Zur Verarbeitung höhere Datenraten wäre der Anteil der parallelen Verarbeitung zu erhöhen. Statt also alle Stützstellen pro Abtastwert sequentiell zu bearbeiten, wären Filterstrukturen geeignet, wie im Entwurf digitaler Systeme beschrieben (siehe Literatur (1)).

Seminararbeit S3: Legen Sie das Filter für Audio-Bereich aus (Grenzfrequenz = 10 kHz, d.h. Abtastrate 20 kSamples/s). Synthetisieren und testen Sie das Filter. Realisieren Sie hierfür geeignete FIR-Filterkoeffizienten, deren Auslegung Sie aus der Literatur entnehmen.

3. Mikroprozessoren

Die Emulation von Mikroprozessoren auf FPGA ist von Interesse für die Implementierung programmierbarer Steuerungen, beispielsweise mit Hilfe von Verzweigungen bzw. Unterprogrammen, ohne dass jedes Mal hierfür der Entwurf eines eigenes Steuerwerks erforderlich ist. Ein anderer Grund ist vorhandene Software für einen bekannten Mikrocontroller (bzw. vorhandene Expertise über einen Mikrocontroller) der statt als Hardware in Form einer Emulation zusammen mit der sonst benötigten Schaltung auf einem FPGA implementiert wird.

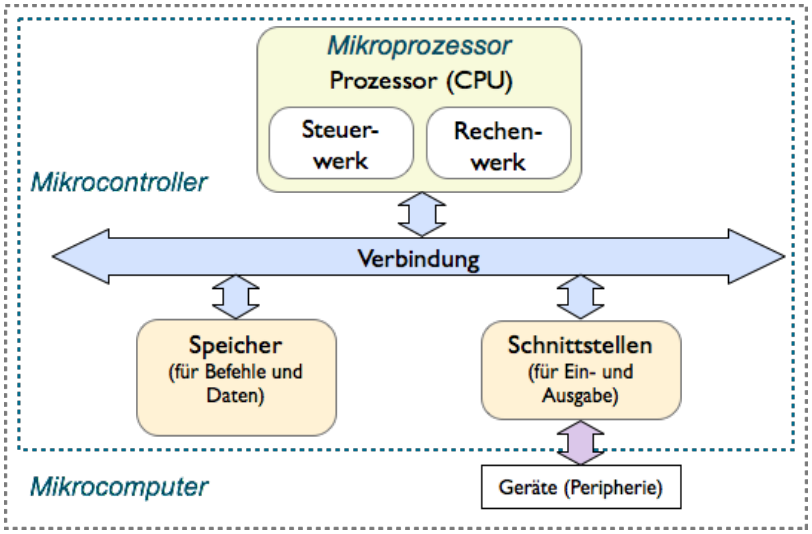


Bild 3.1 Begriffe: Mikroprozessor, Mikrocontroller und Mikrocomputer

Für einen Mikrocontroller charakteristisch ist die Programmierbarkeit, der Speicher für Programm und Daten, sowie Schnittstellen zum Anschluss externer Geräte. Die Abbildung oben zeigt die grundsätzliche Architektur. Wesentlicher Bestandteil des Mikrocontrollers ist der Mikroprozessor, bestehend aus Rechenwerk und Steuerwerk. Zu einem Computer bzw. Mikrocomputer gehören dann noch Geräte zur Eingabe (Tasten, Tatstur, Maus, berührungsempfindliche Anzeigte, ...) und Ausgabe (LEDs, Anzeige, Bildschirm, ...), sowie Schnittstellen zur Vernetzung mit anderen Systemen.

3.1. Prozessorarchitektur

Unter der Architektur eines Prozessors oder Mikrocontrollers versteht man alle Komponenten und Abstraktionen, die zu seiner Programmierung benötigt werden. Hierzu gehört speziell der Befehlssatz und die Möglichkeiten zur Übergabe und Speicherung der Operanden. Ein einfacher Mikroprozessor ist in Akkumulator-Architektur aufgebaut. In dieser Architektur funktioniert der Akkumulator (kurz Akku) als einziges Register, wie bei der Rechenmaschine aus dem letzten Abschnitt. Operanden werden direkt aus dem Datenspeicher entnommen und mit dem Inhalt des Akkus verrechnet. Ergebnisse finden sich im Akku wieder und können von dort aus weiter verarbeitet werden, bzw. in den Datenspeicher geschrieben werden.

Folgende Abbildung zeigt den grundsätzlichen Aufbau des Mikrocontrollers bestehen aus dem Prozessor, dem Programmspeicher und dem Datenspeicher. Bei Mikrocontrollern ist es üblich, dien letztgenannten Komponenten direkt auf dem Chip unterzubringen. In diesem Fall verfügt der Controller über einen vom Datenbus getrennten Programmbus. Mikroprozessoren für den Desktop-Bereich bzw. für Server verfügen über externe Speicher. Um Anschlussleitungen zu sparen, ist hier in der Regel der Programmspeicher mit dem Arbeitsspeicher kombiniert und verfügt über einen gemeinsamen Adressbus.

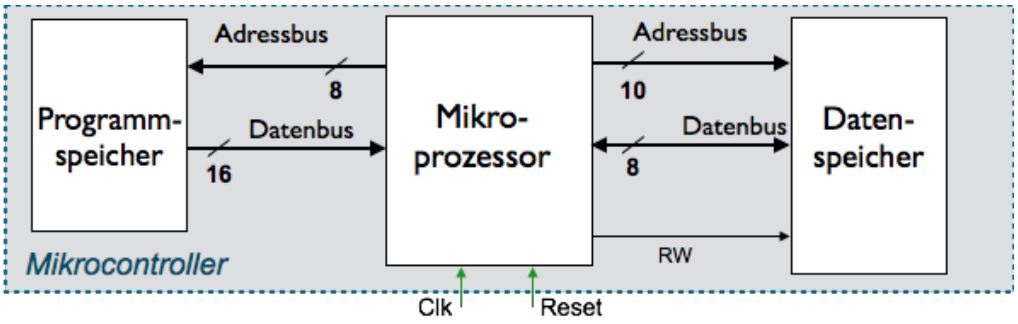


Bild 3.2 Grundsätzlicher Aufbau des Mikrocontrollers

Zur Prozessorarchitektur gehören die Wortbreiten für Programm und Daten, sowie die hierfür gewählten Breiten der Adressbusse. Im Beispiel verwendet der Prozessor 16-Bit breite Befehle, und 8 Bit breite Daten. Der Programmspeicher wird mit einem 8-Bit breiten Adressbus adressiert, ausreichend für kleine Programme mit bis zu 256 Worten. Der Datenspeicher wird mit 10 Bit adressiert, ausreichend für 1024 Bytes an Daten.

Befehlssatz

Die folgende Abbildung zeigt den Befehlssatz des Prozessors.

Befehlssatz			Beschreibung	Prozessor Status Register (PSR) Flags				
Assembler	Kürzel	Opcode		Zero (Z)	Carry (C)	oVerflow (V)	Sign (S)	Neg. (N)
nop	NOP	0	No Operation	-	-	-	-	-
lsl	LSL	1	Akku: C <- Akku <- 0, logic. shift left	x	x	x	x	x
lsr	LSR	2	Akku: 0 -> Akku -> C, logic. shift right	x	x	x	x	x
ldi K	LDI	3	Akku<=K, load immediate 8bit constant	-	-	-	-	-
ld Addr	LDA	4	Akku<=(Addr), load Akku from memory	-	-	-	-	-
st Addr	STR	5	(Addr)<=Akku, store Akku to memory	-	-	-	-	-
add Addr	ADD	6	Akku<=Akku+(Addr), add	x	x	x	x	x
sub Addr	SUB	7	Akku<=Akku-(Addr), subtract	x	x	x	x	x
and Addr	ANDA	8	Akku<=Akku AND (Addr), logical AND	x	-	0	x	x
eor Addr	EOR	9	Akku<=Akku XOR (Addr), logical XOR	x	-	0	x	x
or Addr	ORA	A	Akku<=Akku OR (Addr), logical OR	x	-	0	x	x
jmp K	JMP	B	jump to address constant (K)	-	-	-	-	-
brbc bit, offset	BRBC	C	branch if bit clear (C, Z)	-	-	-	-	-
brbs bit, offset	BRBS	D	branch if bit set (C, Z)	-	-	-	-	-

Bild 3.3 Befehlssatz des Mikroprozessors

Die Tabelle in der Abbildung ist wie folgt aufgebaut: Neben dem Befehl in Assembler-Sprache (z.B. ldi K) findet sich ein Kürzel, das z.B. in Zeitdiagrammen verwendet wird. Daneben ist in hexadezimaler Schreibweise der Befehlscode aufgeführt (Opcode kurz für engl. Operation Code). Die

Kodierung wird nur für die Implementierung in die Maschinensprache verwendet und spielt für die Programmierung in Assembler keine Rolle.

Assembler Sprache

Daneben findet sich eine kurze Beschreibung der Befehle (z.B. für ldi K: Akku<= K, d.h. der Akku übernimmt den Wert der Konstanten K). Die folgenden Spalten zeigen die Bits des Prozessor-Status-Registers. Ein Kreuz an einer der Bits bedeutet, dass der Befehl einen Einfluss auf das besagte Bit hat, d.h. dieses Bit je nach Ergebnis der Operation verändert. Der Befehl ldi K hat als reine Ladeoperation keinen Einfluss auf die Statusbits, ebenso wie die anderen Befehle, die nur Daten verschieben, bzw. die Sprungbefehle. Einfluss auf die Statusbits haben hingegen die arithmetischen und logischen Operationen.

Die durch den Befehlssatz und durch den grundsätzlichen Aufbau gegebene Prozessorarchitektur ist bereits ausreichend, um Assembler-Programme zu schreiben. Eine genauere Kenntnis der Implementierung ist hierfür nicht erforderlich. Mit Hilfe der Befehle soll ein Programm erstellt werden, das zwei Zahlen addiert und das Ergebnis im Datenspeicher hinterlegt.

```
; ASL (Assembler Language) für den DHBW MCT-Mikroprozessor
; Einfaches Testprogramm: c = a + b
; mit den Werten a = 5 und b = 3

start:      ldi  5          ; Akku <= A (mit Wert A = 5)
            st   $0         ; DS(0) <= Akku (Wert A=5 abspeichern)
            ldi  3          ; Akku <= B (mit Wert B = 3)
            add  $0         ; Akku <= Akku + DS(0) (A + B)
end:        st   $1         ; DS(1)<= Akku
```

Der eigentliche Programmtext ist hier in der mittleren Spalte zu sehen. Die Markierungen (Labels) „start“ und „end“ dienen nur der Übersicht. Der mit einem Strichpunkt abgesetzte Text in der Spalte rechts ist als Kommentar zu lesen und soll die Programmschritte erläutern. Im folgenden wird nun untersucht, wie das Programm innerhalb des Prozessors abläuft. Zu diesem Blick auf das Innenleben muss man sich allerdings auf eine mögliche Implementierung der Prozessorarchitektur festlegen.

Ein wesentliches Merkmal der gewählten Prozessorarchitektur ist die Verwendung eines Akkumulators (kurz Akku) als einzigen Register. Das Rechenwerk funktioniert also genauso, wie in der im letzten Abschnitt gezeigten Rechenmaschine (siehe Abbildung 2.8). Der Akku dient zur Aufbewahrung eines Operanden als Quelle und als Ergebnis der Rechenoperation. In der durch den Befehlssatz hier beschriebenen Prozessorarchitektur wird der zweite Operand jedoch aus einem Datenspeicher geladen: der Befehl add $0 addiert beispielsweise den Inhalt des Akkus zum Inhalt der Speicherzelle DS(0). Mit dem Kürzel $0 wird also die Speicherzelle mit der Adresse 0 bezeichnet.

Der Akku selbst kann mit einer Konstanten aus dem Programmspeicher geladen werden (siehe z.B. ldi 5), bzw. mit einem Ladebefehl aus dem Datenspeicher (im Befehlssatz siehe lda $0). Ebenso kann der Inhalt des Akkus in den Datenspeicher geschrieben werden (siehe z.B. str $1). Die Verbindung zwischen dem Datenspeicher und dem Rechenwerk wird somit über den Akku hergestellt. Die Adressierung des Datenspeichers erfolgt hierbei direkt aus dem Assembler-Kode heraus.

Übung 3.1: Erstellen Sie mit dem gegebenen Befehlssatz ein Assembler-Programm, das eine vorgegebene Zahl A (kleiner als 64) in die Speicherzelle 0 schreibt. Anschliessend soll folgende Berechnung ausgeführt werden: B = 16 * A + A. Das Ergebnis soll in Speicherzelle 2 geschrieben werden. Hinweis: Verwenden Sie eine Schiebeoperation für die Multiplikation.

3.2. Implementierung der Prozessorarchitektur

Die Assembler-Sprache dient zwar der maschinennahen Programmierung, kann aber von der Maschine nur in bitkodierter Form ausgeführt werden. Hierzu ist zu definieren, wie die einzelnen Befehle zu kodieren sind. Hierbei gelten die bereits getroffenen Vorgaben über

- die Wortlänge der Befehle (16 Bits)
- die Wortlänge der Daten (8 Bits)
- die Adressbreiten des Befehlsspeichers (8 Bit) und Datenspeichers (10 Bit).

Kodierung der Befehle

Darüber hinaus muss definiert werden, wie viele unterschiedliche Befehle der Prozessor ausführen soll. Hierfür wird eine Zahl von 32 unterschiedlichen Befehlen vorgegeben. Somit werden für die Kodierung jeden Befehls 5 Bits benötigt. Von den 16 Bits eines Befehls verbleiben somit 11 Bits, die für Operanden bzw. Adressen verwendet werden können. Folgende Abbildung zeigt eine Vorgabe für die Befehlsformate.

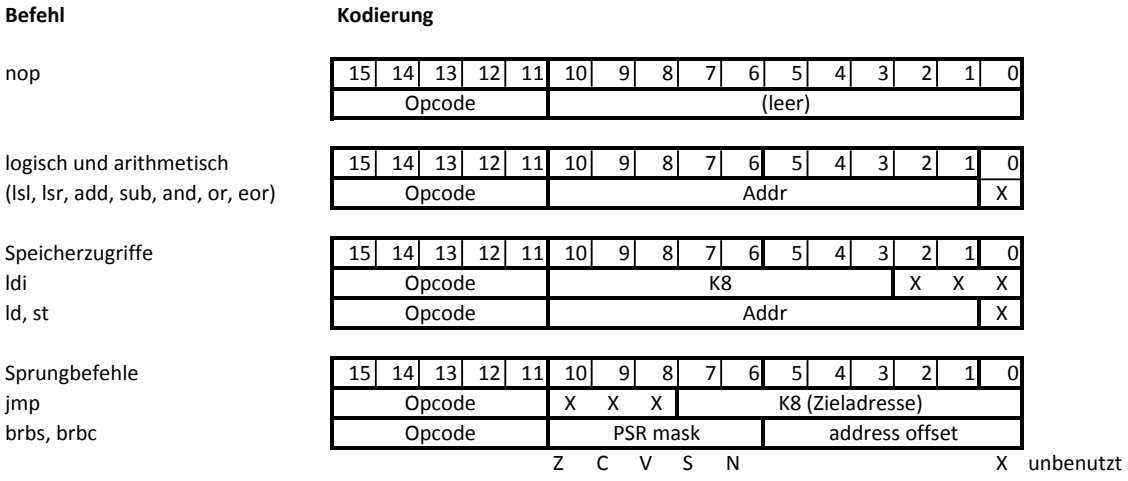


Bild 3.4 Befehlsformate des Prozessors

Der einfachste Befehl führt überhaupt keine Funktion aus (NOP für engl. No Operation) und hat nur zur Folge, dass der nächste Befehl geladen wird. Die übrigen Befehle wurden nach ihrer Funktion bzw. nach ihrem Befehlsformat gruppiert.

Die größte Gruppe ist hierbei die der logischen und arithmetischen Operationen. Bei allen diesen Befehlen befindet sich ein Operand bereits im Akku, der zweite Operand im Datenspeicher. Im Assembler-Befehl wird die Adresse dieses Operanden im Speicher angegeben. Diese Adresse Addr folgt nun im Befehlsformat dem Befehlscode (Opcode). Da der Datenbus in der gewählten Architektur 10 Bits breit ist, genügen als Adresse die auf den Opcode folgenden 10 Bits (vom 10 bis 1).

Der Befehl zum Laden des Akkus aus dem Datenspeicher, sowie der Befehl zum Speichern des Inhalts des Akkus in den Datenspeicher sind ebenso kodiert: Auf den Opcode folgt die Adresse der

gewünschten Speicherzelle. Beim Befehl zum direkten Laden des Akkus LDI folgt auf den Befehlscode die 8-Bit Konstante K. Die übrigen Bits werden nicht benötigt.

Bei den Sprungbefehlen ist zu unterscheiden zwischen dem unbedingten Sprung JMP und den Sprüngen mit Bedingung. Beim unbedingten Sprung folgt auf den Befehlscode die Zieladresse. Da der Adressbus des Programmspeichers mit 8 Bit vorgegeben wurde, genügt für eine absolute Sprungadresse eine 8-Bit Konstante. Dass die Sprungadresse absolut vorgegeben wurde, ist hierbei eine willkürliche Festlegung. Eine Alternative wäre eine Adresse relativ zur Adresse des aktuellen Befehls im Programmspeicher.

Die bedingten Sprungbefehle BRBS (für engl. branch if bit set) und BRBC (für engl. branch if bit clear) führen nur zu einem Sprung (bzw. zu einer Verzweigung), wenn die genannte Bedingung erfüllt ist. Andernfalls erfolgt keine weitere Aktion und es wird der nächste Befehl in der Reihe ausgeführt. Als Bedingung gilt, ob ein bestimmtes Bit im Prozessor-Status-Register gesetzt ist (BRBS), bzw. nicht gesetzt ist (BRBC). Welches Bit hierfür betrachtet werden soll, ist im Feld PSR des Befehls angegeben. Die Auswahl erfolgt bei der hier gewählten Kodierung einfach durch eine Maske über die 5 Bits des Prozessor-Status-Registers. Mit dem Kürzel PSR Mask im Befehlsformat ist diese Maske gemeint (d.h. durch eine 1 an einer der Stellen wird das betreffende Bit ausgewählt).

Die Vorgaben für die Befehlsformate folgen zwar der Prozessorarchitektur, allerdings haben diese Festlegungen im Detail auf die Prozessorarchitektur überhaupt keinen Einfluss. Auf diese Art kann ein in Assembler geschriebenes Programm auch auf einer unterschiedlichen Implementierung des Prozessors funktionieren, der als Variante ein Mitglied der Prozessorfamilie dar stellt. Das Assembler-Programm muss dann auf die unterschiedliche Kodierung des Prozessors übersetzt werden.

Maschinensprache

Folgender Text zeigt die Kodierung des Assembler-Programms mit den in der Abbildung oben gezeigten Befehlsformaten. Im Unterschied zur Assembler-Sprache ist dieses Programm nun direkt von der Maschine ausführbar (es ist in Maschinensprache kodiert).

```
-- Maschinensprache für den DHBW MCT-Mikroprozessor
-- Einfaches Testprogramm: c = a + b
-- mit den Werten a = 5 und b = 3
-- aus dem Assembler-Programm manuell übersetzt

0 => "0001100000101000",        -- LDI 5
1 => "0010100000000000",        -- STR $0
2 => "0001100000011000",        -- LDI 3
3 => "0011000000000000",        -- ADD $0
4 => "0010100000000010",        -- STR $1
```

Auch dieser Programmtext enthält noch Textkommentare (hier mit „--“ eingeleitet), sowie die Zeilennummern für die Ablage der Befehle im Programmspeicher am Anfang jeder Zeile (z.B. „0 =>“) gefolgt von einem 16Bit-Wert in doppelten Hochkommas). In den Programmspeicher werden nur die 16-Bit Werte geladen. In den ersten 5 Bits erkennt man die Befehlscodes (Opcodes) der Befehle aus Abbildung 3.2 (z.B. 00011 für hexadezimal 3 bzw. den Befehl LDI). In den folgenden Bits sind die Konstanten bzw. die Adressen enthalten, wie in Abbildung 3.4 vereinbart.

Übung 3.2: Übersetzen Sie Ihr Programm aus Übung 3.1 in Maschinensprache. Hinweis: Verwenden Sie ggf. ein Tabellenkalkulationsprogramm der Übersicht halber.

Prozessor

Die folgende Abbildung zeigt das Innenleben des Mikrocontrollers als Blockschaltbild. Der Prozessor besteht aus einem Steuerwerk und einem Rechenwerk. Die übrigen Komponenten sind der Programmspeicher und der Datenspeicher. Wie bereits im Befehlssatz festgelegt, verfügt das Rechenwerk über eine Akku-Architektur. Der Akku als einziges Register ist die Brücke zum Datenspeicher. Das Rechenwerk entspricht somit ganz der im letzten Abschnitt bereits diskutierten Rechenmaschine. Allerdings wird das Rechenwerk nun mit Daten aus dem Datenspeicher versorgt.

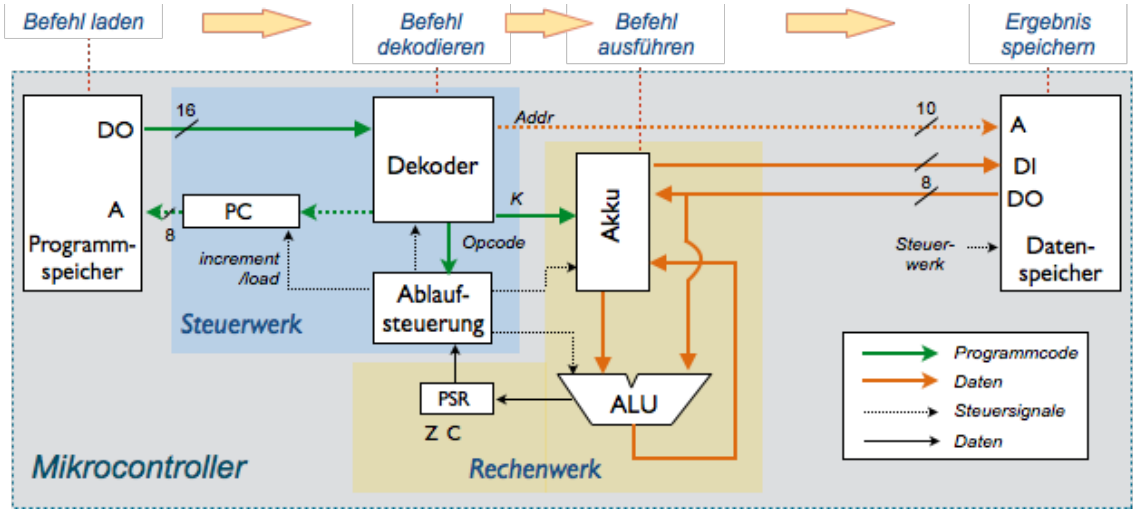


Bild 3.5 Prozessor mit Akkumulator-Architektur

Bei der im Befehlssatz verwendeten direkten Adressierung wird die Adresse des jeweiligen Operanden aus dem Datenspeicher direkt im Befehl codiert. Aus diesem Grund entspringt im Blockschaltbild die Adresse des Datenspeichers dem Dekoder, der die Adresse dem Befehlscode entnimmt. Ausgang und Eingang des Datenspeichers sind direkt mit dem Akku verbunden. Die Auswahl übernimmt der Akku in Abhängigkeit des jeweils vorliegenden Befehls. Das direkte Laden des Akkus mit einer Konstanten im Befehl ist in der Abbildung als Verbindung zwischen Dekoder und Akku dargestellt.

Zum Steuerwerk gehören neben der bereits bekannten Ablaufsteuerung und dem Befehls-Dekoder der Programmzähler (PC für engl. Program Counter). Wie der Name bereits ausdrückt, ist dieser Zeiger in den Programmspeicher als Zähler ausgeführt, der den jeweils folgenden Befehl adressiert. Im Falle eines Sprungbefehls wird der Programmzähler von der Ablaufsteuerung neu geladen.

Die Ablaufsteuerung verfügt nun über Steuerleitungen zu allen anderen Komponenten. Der grundsätzliche Ablauf ist hierbei wie folgt: (1) der Dekoder übergibt den Steuerwerk den Befehlscode des aktuellen Befehls. Ja nach Befehl stellt der Dekoder auch die Adresse in den Datenspeicher bzw. die in den Akku zu ladende Konstante zur Verfügung. (2) das Steuerwerk bedient in Abhängigkeit des Befehlscodes die Steuerleitungen zu den Komponenten. Rechenwerk und Steuerwerk arbeiten taktgesteuert. Wie die Abbildung zeigt, laufen bzgl. des Kontrollflusses alle Fäden in der Ablaufsteuerung zusammen. Mit dem Datenpfad hat die Ablaufsteuerung hingegen nichts zu tun.

Ablauf des Programms

Folgende Abbildung zeigt den Programmablauf. Dargestellt sind neben dem Takt (Clk) und einem Reset-Signal der Programmzähler (PC) als Adresse am Programmspeicher, sowie der Datenausgang des Programmspeichers (PS-DO). Man erkennt dass mit steigenden Taktflanken die Adresse des Programmspeichers hochgezählt wird. Mit jeder folgenden Taktflanke gibt der Programmspeicher den zugehörigen Befehl aus (unter PS-DO als 16-Bit Wort in hexadezimaler Schreibweise notiert).

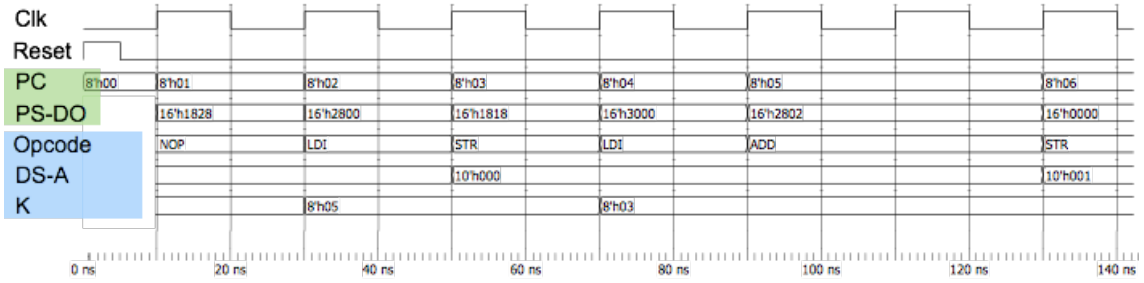


Bild 3.6 Programmablauf am Steuerwerk

Darunter dargestellt sind drei Ausgänge des Dekoders: (1) der Befehlscode (Opcode), den der Dekoder aus dem 16-Bit Programmwort entnimmt, (2) ggf. eine Adresse für den Datenspeicher (DS-A), sowie (3) ggf. eine Konstante K zum Laden des Akkus. In der Zeile Opcode erkennt man den Ablauf des vorher beschriebenen Testprogramms (wobei durch den Reset als initialer Wert am Ausgang des Dekoders der Befehl NOP vorliegt):

- LDI K=5
- STR $0
- LDI K=3
- ADD $0
- STR $1.

Man kann dem Verlauf des Programms unmittelbar folgen. Etwas irritierend ist möglicherweise der Vorlauf des Programmzählers gegenüber dem aktuell dekodierten Opcode: Der Zähler ist jeweils 2 Takte voraus. Der erste Takt Verzögerung kommt durch den Programmspeicher zustande, wie die Zeile unmittelbar unter dem Programmzähler zeigt. Einen weiteren Takt benötigt der Dekoder zum Dekodieren des Befehls. Zum Zeitpunkt n eilt der Befehlszähler also bereits 2 Schritte vor.

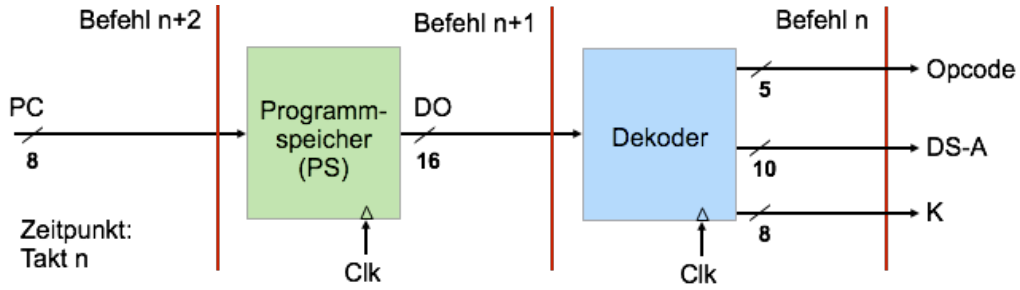


Bild 3.7 Programmspeicher und Dekoder als Schieberegister

Im Grunde genommen entspricht die Anordnung einem zweistufigen Schieberegister, wie in der Abbildung oben gezeigt. Wenn am Ausgang des Dekoders der erste Befehl LDI K=5 von Adresse 0 dekodiert ist, zeigt der Befehls-zähler (PC) bereits auf die Adresse 2. Der Befehl am Adresse 1 wurde

bereits ausgelesen und wird noch am Ausgang des Programmspeichers vorgehalten, bis er mit der nächsten Taktflanke in den Dekoder übernommen wird. Die Reihenfolge der Befehle wird hierdurch nicht geändert. Sobald die Kette einmal angelaufen ist, folgt auch mit jedem Takt ein neuer Befehl.

Folgende Abbildung zeigt nun den Ablauf des Programms zusammen mit dem Rechenwerk und dem Datenspeicher. Außer dem bereits bekannten Ablauf um den Programmspeicher und Dekoder sind die Ausgänge der ALU und des Akkus dargestellt, sowie die Adressleitung des Datenspeichers (DS-A), der Eingang des Datenspeichers (DS-DI), der Ausgang des Datenspeichers (DS-DO), sowie die ersten beiden Speicherzellen DS(0) und DS(1).

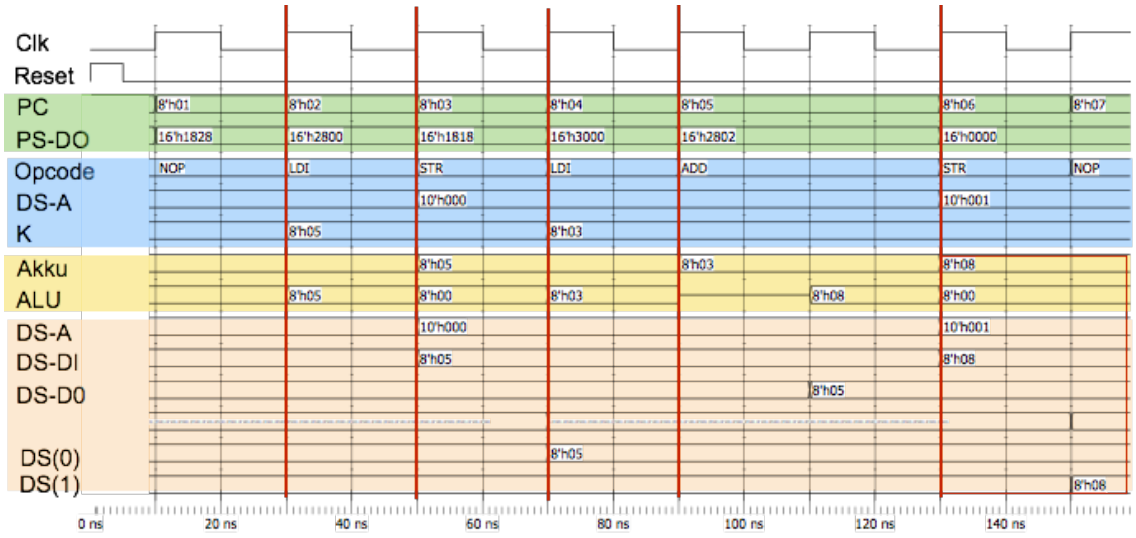


Bild 3.8 Programmablauf mit Rechenwerk und Datenspeicher

Der prinzipieller Ablauf des Programms ist wie folgt:

- LDI K=5: Die Konstante J liegt mit dem Opcode am Dekoder an, somit nach der Schaltung im Blockschaltbild (siehe Abbildung 3.5) auch am Eingang des Rechenwerks (ALU in der Abbildung oben). Im folgenden Takt wird der Wert dann in den Akku übernommen. In diesem folgenden Takt wurde bereits der nächste Befehl dekodiert.

- STR $0: Mit dem Opcode liegt am Ausgang des Dekoders auch die Zieladresse DS-A im Datenspeicher zur Aufnahme des Akku-Inhalts (10‘h000 in der Abbildung). Mit dem folgenden Takt wird der Inhalt des Akkus übernommen, wie der Inhalt der Speicherzelle DS(0) zeigt. Mit diesem Takt wurde bereits der nächste Befehl dekodiert.

- LDI K=3: Der Akku wird mit der Konstante K=3 geladen. Ablauf wie im ersten Befehl.

- ADD $0: Ein Operand (K=3) befindet sich bereits im Akku, der zweite muss erst aus dem Datenspeicher geladen werden. Mit dem Opcode liegt am Ausgang des Decoders bereits die Adresse der gewünschten Speicherzelle (DS-A wiederum 10‘h000). Es benötigt jedoch einen weiteren Takt, um des zugehörigen Wert aus dem Datenspeicher zu laden. Der Wert findet sich dann auf der Leitung DS-DO, die zum Eingang für den zweiten Operanden der ALU führt (siehe Signal ALU am Eingang des Rechenwerkes). Die ALU als reines Schaltnetz stellt das Ergebnis unmittelbar zur Verfügung. Das Laden des Ergebnisses in den Akku benötigt jedoch einen weiteren Takt. Insgesamt benötigt der

Befehl ADD (gemessen von der Dekodierung bis zur Fertigstellung) also 3 Takte, bzw. einen Takt länger als die übrigen Befehle. Mit dem 3. Takt kann aber bereits der folgende Befehl dekodiert werden.

- STR $1: Mit dem Opcode liegt die Zieladresse DS-A am Datenspeicher (mit Wert 10‘h 001). Der Inhalt des Akkus wird mit dem folgenden Takt in die Speicherzelle DS(1) geschrieben.

Die Ausführung des Programms zeigt folgende Eigenschaften des Prozessors: (1) Auch bei der Ausführung reicht ein Befehl in den folgenden hinein: Während des Abspeichern eines Ergebnisses in den Akku bzw. in den Datenspeicher kann bereits der folgende Befehl dekodiert werden. (2) Akku und Datenspeicher sind hierbei ebenfalls als Schieberegister hintereinander geschaltet. Folgende Abbildung zeigt diese Anordnung.

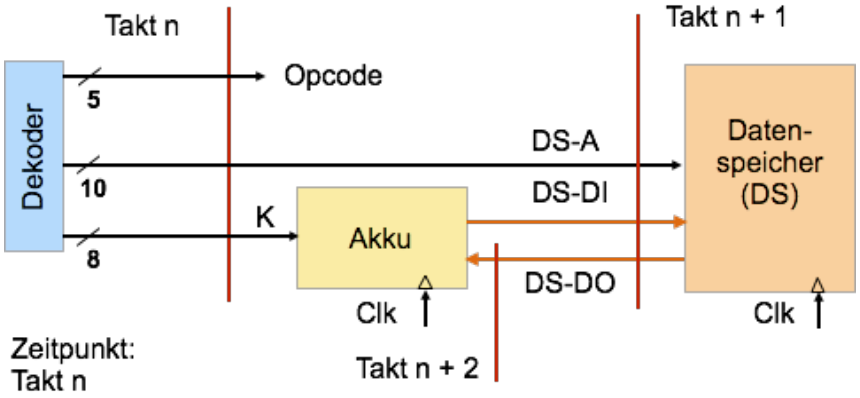


Bild 3.9 Akku und Datenspeicher als Schieberegister

Zum Zeitpunkt n sei der aktuelle Befehl dekodiert und die Zieladresse DS-A bzw. Konstante K liege vor. Es benötigt einen weiteren Takt, um die Konstante K im Akku zu speichern, bzw. um einen Wert aus der adressierten Speicheradresse zu lesen. Ist letztere Operation Teil eines arithmetischen oder logischen Befehls wie ADD $0, so wird ein weiterer Takt benötigt, um das Ergebnis im Akku zu speichern.

Übung 3.3: Skizzieren Sie den zeitlichen Ablauf Ihres Programms aus Übung 3.1 in einen Zeitdiagramm. Hinweis: Übernehmen Sie die Struktur aus Abbildung 3.8 und verwenden Sie der Übersichtlichkeit halber ggf. ein Programm zu Tabellenkalkulation.

Fliessbandverarbeitung

Wie die Ausführung des Programmbeispiels zeigt, sind die Schritte zur Ausführung der Befehle so aneinander gekettet, dass möglichst in keiner der beteiligten Stationen ein Leerlauf entsteht. Diese Art der Ausführung ist auch als Fliessbandverarbeitung bekannt (engl. pipeline processing). Wie in der Serienproduktion entsprechen Programmspeicher, Dekoder, Rechenwerk und Datenspeicher einzelnen Stationen im Fertigungsprozess. Mit dem Systemtakt als Arbeitstakt wird wie an einem Förderband das Werkstück im Prozessor der Befehl an der nächsten Station weiter verarbeitet.

Damit diese Methode funktioniert, müssen alle Stationen im Arbeitsprozess ihren Arbeitsschritt im Takt beenden. Folgend Abbildung zeigt das Prinzip. Als Arbeitsstationen sind Programmspeicher,

Dekoder, Rechenwerk und Datenspeicher eingezeichnet. Es wird angenommen, dass der Arbeitsprozess von links nach rechts verläuft. Mit jedem Arbeitstakt wird das Werkstück von einer Arbeitsstation zur nächsten übergeben. Das Fliessband läuft also nicht kontinuierlich, sondern befördert alle Werkstücke mit dem nächsten Arbeitstakt eine Station weiter.

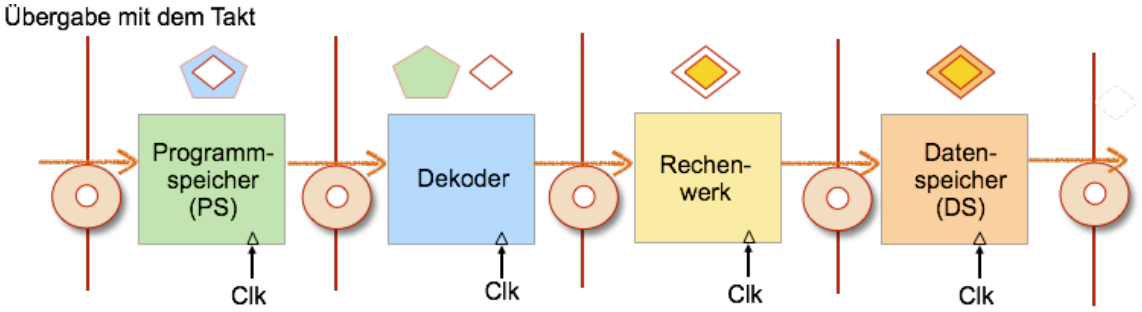


Bild 3.10 Fliessbandverarbeitung

Auf diese Methode lassen sich alle Arbeitsstationen gleichzeitig auslasten. Leerlauf gibt es nur mit dem Anlaufen der Strecke mit dem ersten Werkstück. Was geschieht aber, wenn eine der beteiligten Stationen für bestimmte Operationen mehr als einen Arbeitstakt benötigt? Im Programmbeispiel war das beim Befehl ADD der Fall. Hier benötigt das Rechenwerk einen zusätzlichen Takt.

Während dieses zusätzlich benötigten Taktes müssen alle Stationen vor dem Rechenwerk einen Takt aussetzen. Das Fliessband vor dem Rechenwerk wird hierzu einen Takt lang angehalten. Wie man im Zeitdiagramm in Abbildung 3.8 erkennt, zählt während dieses zusätzlich benötigten Taktes der Befehlszähler nicht weiter, und der Dekoder dekodiert auch nicht seinen bereits geladenen nächsten Befehl.

Ohne diese Massnahme (Fliessband vor der Station anhalten) käme der komplette Ablauf durcheinander. An einem realen Fliessband bekäme das Rechenwerk das nächste Werkstück zugeschoben, bevor es mit dem aktuellen Werkstück fertig ist. Hier fällt also eins der Werkstücke vom Band und der Prozess gerät völlig durcheinander.

Welche Einheit ist nun dafür zuständig, zu erkennen, dass ein aktueller Arbeitsschritt in einer der beteiligten Stationen einen zusätzlichen Arbeitstakt benötigt? In der Praxis übernimmt diese Rolle am besten die Komponente, die auch in den Prozess eingreifen kann und das Fliessband für alle vorderen Stationen für die benötigte Zeit anhält. Im Blockschaltbild ist diese Einheit die Ablaufsteuerung des Steuerwerks. Diese Komponente besitzt Steuerleitungen zu allen übrigen Komponenten. Sie erkennt am Opcodes des Befehls den Bedarf nach einem zusätzlichen Arbeitstakt und hält für diese Zeit die voraus produzierenden Einheiten an.

3.3. Ablauf der einzelnen Befehle

Wie werden die einzelnen Befehle nun abgearbeitet? Im folgenden wird der Ablauf der Befehle der Gruppen NOP, Schiebeoperationen, logische und arithmetische Operationen, Speicherzugriffe und Sprungbefehle Schritt für Schritt erläutert.

No Operation

Dieser Befehl bewirkt nur, dass der Programmzähler inkrementiert wird, so dass der nächste Befehl geladen wird.

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

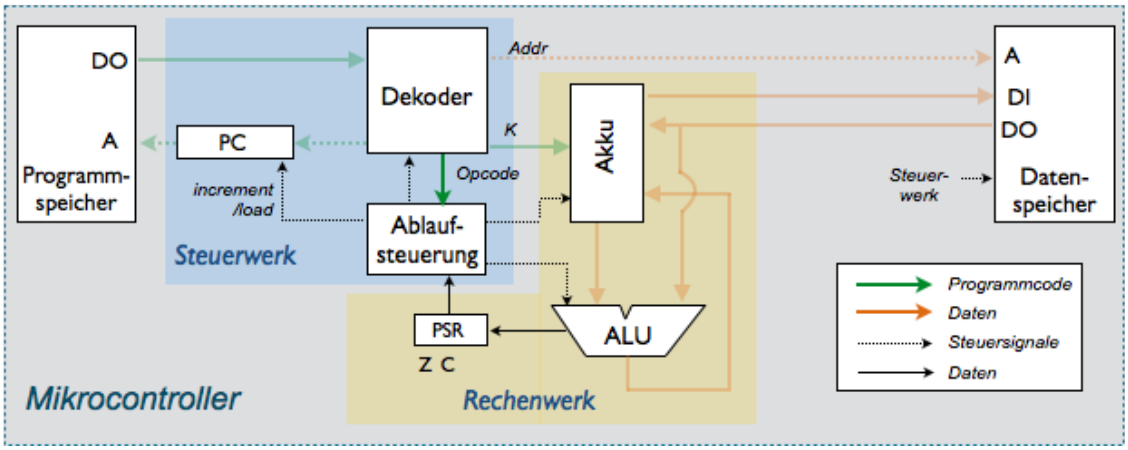


Bild 3.11 Ausführen des Befehls NOP

Aktion: Das Steuerwerk veranlasst keine Aktion. Das Inkrement-Signal für den Programmzähler (PC) bleibt gesetzt, so dass der PC nach dem Laden des folgenden Befehls weiter inkrementiert wird.

Benötigte Takte: ein Takt.

Schiebeoperationen

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

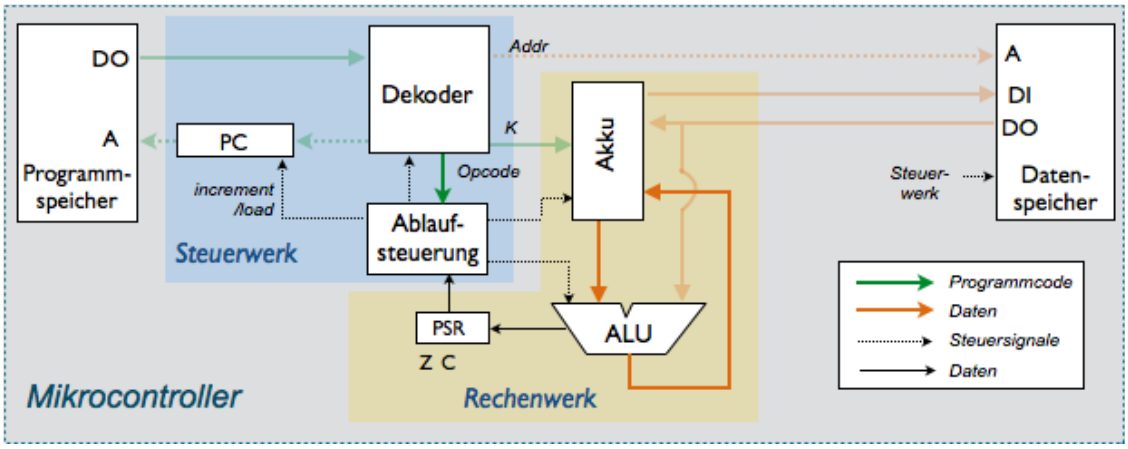


Bild 3.12 Ausführung der Schiebeoperation

Aktion: Die Schiebeoperationen lsl und lsr wirken auf ALU und Akku und benötigen keine weiteren Operanden (der Operand befindet sich bereits im Akku und steht am Eingang der ALU). Das Steuerwerk signalisiert der ALU die Verschiebeoperation.

Die ALU arbeitet als Schaltnetz, führt die Verschiebung unmittelbar durch und setzt das PSR entsprechend. Das Ergebnis steht am Ausgang der ALU zur Verfügung und wird mit dem nächsten Takt in den Akku übernommen.

Benötigte Takte: ein Takt.

Arithmetische und Logische Operationen

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

Logische und arithmetische Operationen enthalten neben dem Opcode die Adresse des zweiten Operanden, der mit dem Inhalt des Akkus verarbeitet wird. Diese Adresse legt der Dekoder an den Adressbus des Datenspeichers.

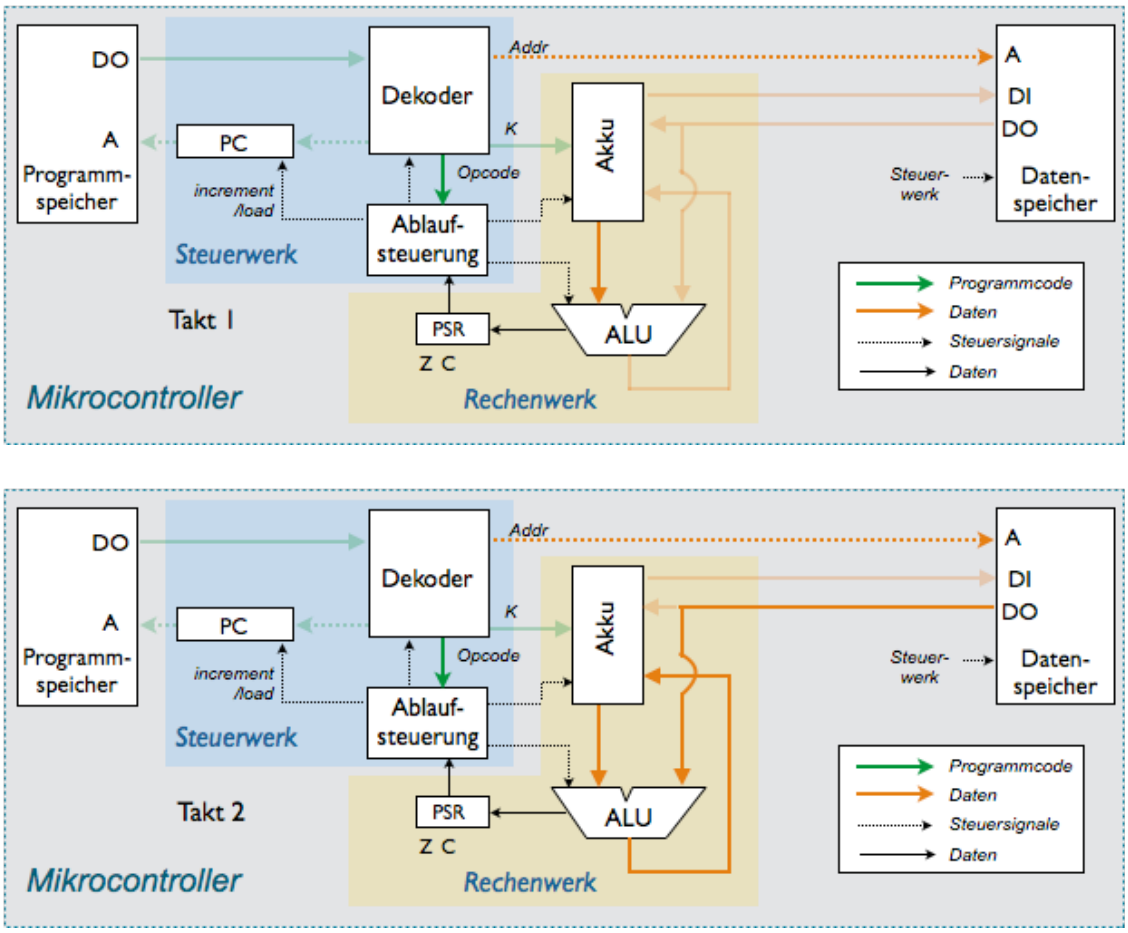


Bild 3.13 Ausführung logischer und arithmetischer Operationen

Aktion: Ein Takt wird benötigt, um den Operanden aus dem Datenspeicher auszulesen. Das Steuerwerk legt im ersten Takt das Lesesignal an den Datenspeicher. Der Akku wird während dieses Taktes nicht benötigt. Sein Inhalt enthält bereits den anderen Operanden der Operation. Ebenso führt die ALU im ersten Takt keine Operationen durch. Damit kein weiterer Befehl aus dem Programmspeicher geladen wird, wird das Inkrementieren des PC deaktiviert (der PC zeigt ja bereits auf den

nächsten Befehl), sowie der Programmspeicher und der Dekoder angehalten. Das Fliessband der vorgelagerten Arbeitsabschnitte steht somit in diesem Takt still.

Im zweiten Takt steht der zweite Operand aus dem Datenspeicher zur Verfügung. In der ALU erfolgt unmittelbar die logische bzw. arithmetische Operation mit den nunmehr vorliegenden beiden Operanden. PSR und Ergebnis stellt die ALU als Schaltnetz unmittelbar zur Verfügung, so dass mit dem folgenden Takt der Akku das Ergebnis aufnehmen kann.

Damit das Steuerwerk den zweitaktigen Modus korrekt bearbeitet, schaltet es beim ersten Takt in einen Zweitaktmodus (als Zustand eines Zustandsautomaten). In diesem zweiten Zustand verhält es sich dann bei gegebenen Opcode anders als im ersten Takt und führt die benötigten Operationen aus.

Benötigte Takte: Zwei Takte.

Speicherzugriffe

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

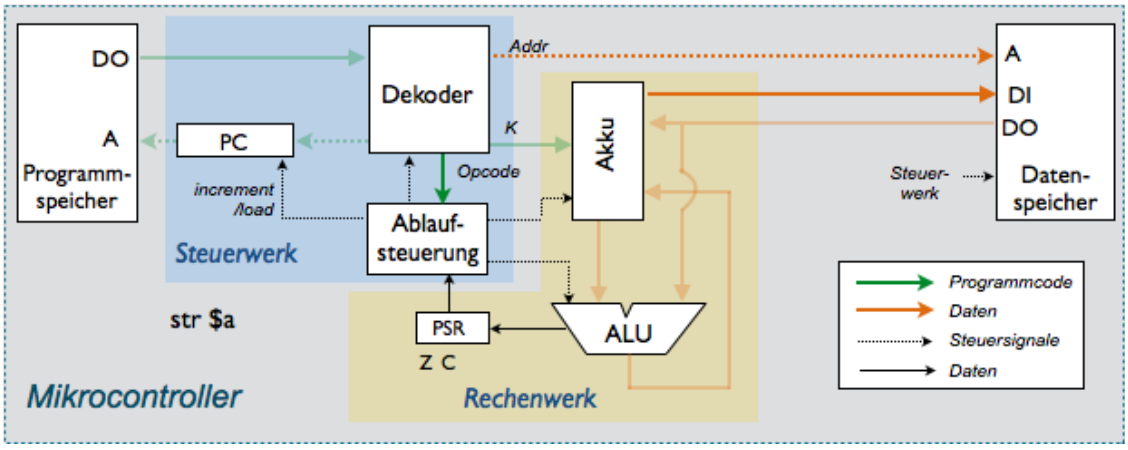


Bild 3.14 Schreiboperation ausführen

Aktion: (A) Schreiboperation: Die Zieladresse liegt zusammen mit dem dekodierten Opcode bereits vor. Der Akku enthält bereits den zu speichernden Inhalt. Der Datenspeicher erhält ein Lesesignal und kann den Inhalt des Akkus mit dem folgenden Takt übernehmen.

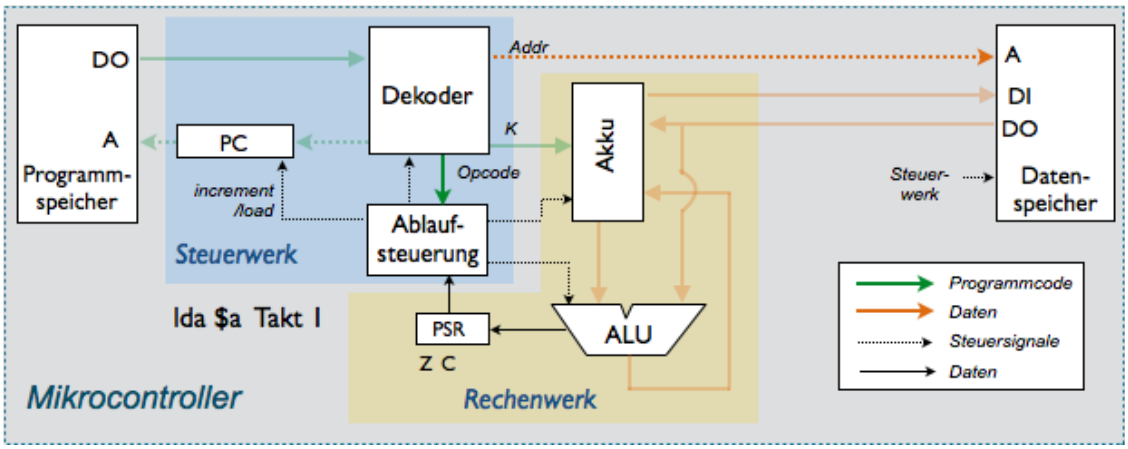


Bild 3.15 Leseoperation ausführen

Aktion (B) Leseoperationen: Die Quelladresse liegt zusammen mit dem dekodierten Opcode bereits vor. Der folgende Takt wird benötigt, um die Quelle aus dem Datenspeicher auszulesen. Der Akku kann den Wert dann mit dem nächsten folgenden Takt übernehmen.

Benötigte Takte:

- Schreiboperation: Ein Takt
- Leseoperation: Zwei Takte

Übung 3.4: Funktionen der Ablaufsteuerung: Gehen Sie den Ablauf des Beispielprogramms aus Abbildung 3.8 Befehl für Befehl durch. Welche Aktionen muss die Ablaufsteuerung jeweils durchführen? Vermerken Sie die Aktionen im Zeitdiagramm. Definieren Sie passende Steuersignale für die einzelnen Komponenten des Mikrocontrollers.

Übung 3.5: Funktionen der Ablaufsteuerung: Gehen Sie den Ablauf Ihres Programms aus Übung 3.1 Befehl für Befehl durch. Welche Aktionen muss die Ablaufsteuerung jeweils durchführen? Vermerken Sie die Aktionen im Zeitdiagramm. Hinweis: Verwende Sie der Übersichtlichkeit halber ggf. ein Programm zur Tabellenkalkulation.

Sprungbefehle

Ausgangssituation: Der Befehl wurde vor zwei Takten aus dem Programmspeicher in den Dekoder geladen. Der Befehlszähler zeigt bereits auf den Befehl n+2. Der Ausgang des Dekoders stellt dem Steuerwerk den aktuellen dekodierten Befehl zum Zeitpunkt n zur Verfügung.

Aktion: Das Steuerwerk liest das PSR aus und entscheidet, ob die Sprungbedingung ausgeführt wird. Ergebnisse: (1) kein Sprung: keine Aktion, mit dem folgenden Takt könnte der nächste Befehl geladen werden. (2) Sprung: der Offset für den Programmzähler (bzw. der absolute Wert) liegt bereits am Ausgang des Dekoders an. Das Steuerwerk gibt dem PC Anweisung, seinen Inhalt mit dem nächsten Takt mit dem Offset zu erhöhen (bzw. den absoluten Wert zu laden). Programmspeicher und Dekoder werden für diesen Takt deaktiviert, damit während der Aktualisierung des PCs nicht der bisher adressierte Befehl aus dem Programmspeicher gelesen wird.

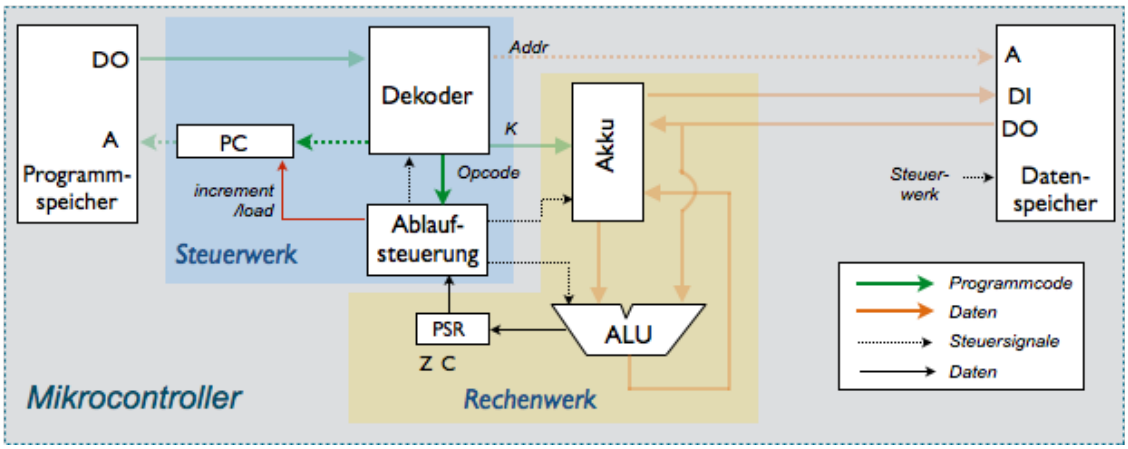


Bild 3.16 Sprungbefehl ausführen

Benötigte Takte: Zwei Takte.

Da ja vor Prüfung der Bedingung nicht feststeht, ob ein Sprung stattfindet. Werden Programmzähler, Programmspeicher und Dekoder auf jeden Fall angehalten. Im folgenden Takt wird dann für den Fall, dass kein Sprung erfolgt, die Bearbeitung fortgesetzt. Da die Pipeline nur angehalten wurde, ist die Fortsetzung ohne Störungen möglich.

Im Fall, dass gesprungen wird, hat der Programmzähler im zweiten Takt den neuen Stand. Mit dem folgenden Takt liest der den Befehl an der Sprungadresse aus dem Programmspeicher. Allerdings befinden sich auf dem Weg vom Programmspeicher zum Dekoders noch vor dem Anhalten der Pipeline bereits geladenen, nun nicht mehr korrekte Befehl.

Diese Situation lässt sich z.B. dadurch entschärfen, dass man im Programm im Anschluss an einen Sprungbefehl grundsätzlich einen Leerlaufbefehl (NOP) anbringt, der nur der Fließbandverarbeitung geschuldet ist. In einer Entwicklungsumgebung übernimmt solche Massnahmen der Compiler für das Assembler-Programm bzw. der Compiler für die Programmierung in Hochsprachen.

3.4. Programme mit Sprungbefehlen

Als Beispiel für ein programm mit Sprungbefehlen soll das Maximum zweier Zahlen A und B bestimmt werden. Die beiden Zahlen seien im Datenspeicher in den Speicherzellen DS(1) und DS(2) abgelegt. Das Ergebnis soll in Speicher DS(0) abgelegt werden. Den Ablauf beschreibt folgende Programmtext.

```
; ASL (Assembler Language) für den DHBW MCT-Mikroprozessor
; Maximum zweier Zahlen A und B ermitteln
; Bedingung: A und B sind im Datenspeicher bei $01 und $02 abgelegt
;
             ld  $01          ; Akku <= A
checkAB:     sub $02          ; Akku <= A - B
             brbc N, maxB     ; if (A ≥ B) then branch to maxA
maxB:        ld  $02          ; Akku <= B
             st  $00          ; DS(0) <= B
             ldi 0            ; Akku <= 0
             st  $03          ; (DS(3) <= 0
```

```
                anda $03          ; ALU <= 0 AND 0 = Zero
                brbs Z, end       ; branch to end
    maxA:       ld  $01           ; max(A,B) = A
                st  $00           ; DS(0) <= A
    end:        ld  $00           ; Akku <= max (A,B)
```

Der erste Sprungbefehl verwendet das Vorzeichenbit des Differenz von A-B. War A größer oder gleich B, so bleibt das Ergebnis positiv, also das Vorzeichenbit N = 0. Diese Bedingung wird mit der Anweisung BRBC (engl. für branch if bit clear) mit Verweis auf das Vorzeichenbit geprüft. In diesem Fall ist A das Maximum, und das Programm verzweigt zur Markierung maxA. An der Markierung wird das Ergebnis max(A,B) = A in der Speicherzelle DS(0) abgelegt.

Im anderen Fall (B > A) findet kein Sprung statt, der nächste Befehl wird geladen. Der besseren Lesbarkeit wegen ist an dieser Stelle die Markierung maxB eingefügt. In diesem Zweig wird das Ergebnis max (A,B) = B an der Speicherzelle DS(0) abgelegt. Die Markierung maxA muss dann aber übersprungen werden. Statt eines unbedingten Sprungbefehls wird hierfür ein bedingter Sprung mit erzwungener Bedingung verwendet: (1) ldi 0 lädt den Wert Null in den Akku, (2) die folgende Prüfung auf das Zero-Bit führt dann auf jeden Fall zu einem Sprung ans Ende des Programms.

Übung 3.6: Übersetzen Sie das Programm in die Maschinensprache. Wie erfolgt die Kodierung der Sprungbefehle? Erläutern Sie Ihr Vorgehen.

Die Abbildung auf der folgenden Seite zeigt den Ablauf des Programms. Hierbei wurden vorab folgende Werte im Datenspeicher hinterlegt A= 7 in DS(1), B= 5 in DS(2). Man erkennt im Ablauf das Laden des Wertes A in den Akku. Hierauf folgt die Subtraktion von B. Da der Inhalt des Akkus mit den vorgegebenen Werten positiv bleibt, ist das Maximum beider Werte A. Der folgenden Sprungbefehl prüft das Vorzeichen des Ergebnisses und wird somit ausgeführt.

Im Ablauf erkennt man, das im zweiten Takt des Sprungbefehls (BRBC) der Programmzähler (PC) auf die Zieladresse erhöht wird. Das Programm springt zur Markierung maxA und lädt den Befehl ld $01, d.h. den Wert A in den Akku. Dieser wird dann in die Speicherzelle DS(0) geschrieben.

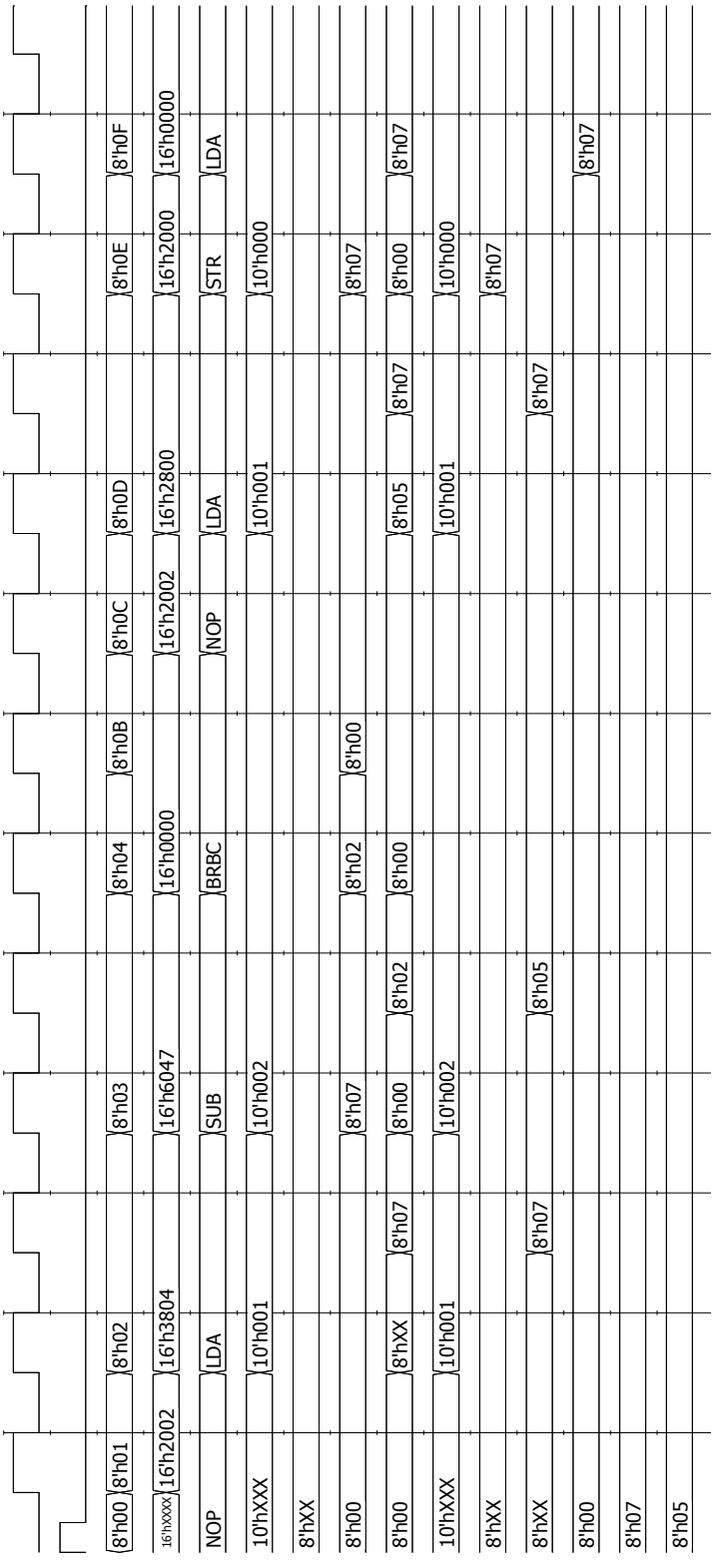


Bild 3.17 Ablauf des Programms mit Sprungbefehlen

Im Akku und im Speicher DS(0) verbleibt das Maximum der beiden Werte. Im Zeitdiagramm erkennt man ebenfalls, dass in das Programm in Maschinensprache zwei Leerlaufbefehle (NOP) vor den Sprungbefehl eingefügt wurden. Diese dienen dazu, zu vermeiden, dass auf dem Weg vom Programmzähler zur Ablaufsteuerung bei einem Sprung nicht mehr passende Befehle in die Verarbeitung geraten.

Übung 3.7: Vertauschen Sie die Werte A und B im Programmbeispiel (d.h. DS(1) = A = 5 und DS(1) = B = 7. Wie sollte das Programm jetzt ablaufen? Erstellen Sie ein Zeitdiagramm. Hinweis: Verwenden Sie der Übersichtlichkeit halber ggf. ein Programm zur Tabellenkalkulation.

Übung 3.8: Erstellen Sie in Assembler-Sprache unter Verwendung des gegebenen Befehlssatzes ein Programm zur Berechnung des Maximums dreier Zahlen max(A, B, und C). Hierbei sei angenommen, dass sich die Zahlen im Datenspeicher unter den Adressen 1, 2 und 3 befinden. Das Ergebnis wird im Speicher unter der Adresse 0 abgelegt. Hinweis: Erstellen Sie zunächst ein Aktivitätsdiagramm bzw. einen Ablaufplan für das Vorgehen. Verwenden Sie Sprungbefehle und geeignete Prüfbedingungen hierfür.

Übung 3.9: Skizzieren Sie den zeitlichen Ablauf des Programms aus Übung 3.8 für ein ausgewähltes Szenario. Erstellen Sie ein Zeitdiagramm. Hinweis: Verwenden Sie der Übersichtlichkeit halber ggf. ein Programm zur Tabellenkalkulation.

3.5. Implementierung in HDL

Bei der Implementierung in HDL dient das bereits eingangs in Abschnitt 3 gezeigte Blockschaltbild sowie die Abläufe für die einzelnen Befehle der Orientierung. Einige Komponenten wie die ALU mit dem PSR, sowie der Registerblock werden erst bei der Implementierung näher detailliert. Hierzu gehört auch die Definition der jeweils benötigen Steuersignale für die einzelnen Befehle.

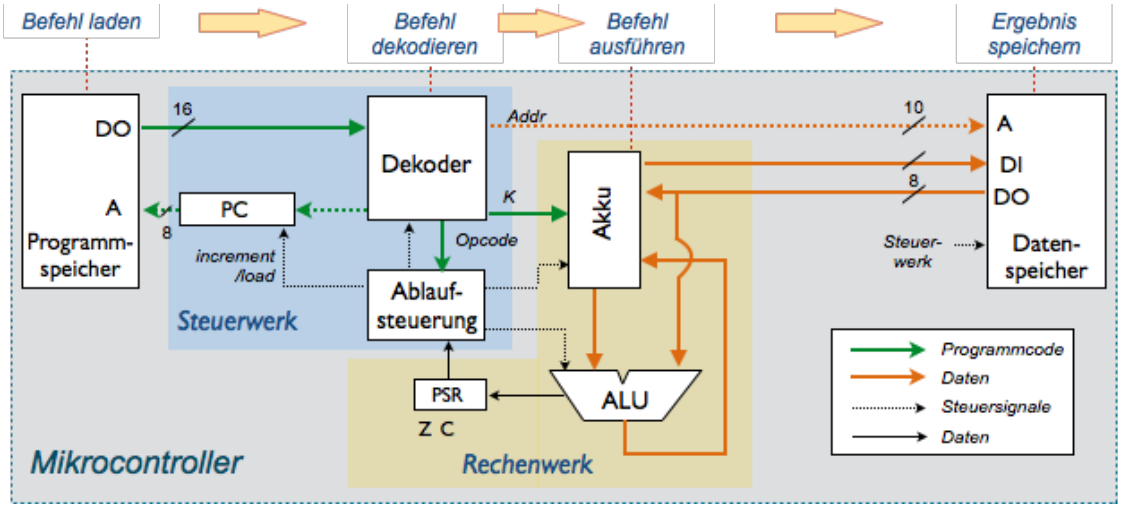


Bild 3.18 Blockschaltbild des Prozessors als Basis der Implementierung

Das Instruktionsregister wurde mit dem Programmzähler zu einem Funktionsblock zusammengefasst. Weitere Funktionsblöcke bilden die ALU mit dem PSR, sowie der Registerblock mit dem Stapelzeiger. Die beiden Speicher bilden ebenfalls Funktionsblöcke (Vorlagen hierfür finden sich in den Speicherblöcken aus Kapitel 2). Es verbleibt die Ablaufsteuerung des Steuerwerk. Die Ablauf-

steuerung wird als Zustandsautomat ausgelegt (Finite State Machine) und bildet außerdem den übergeordneten Funktionsblock (Top Modul), in den alle anderen Einheiten als Komponenten eigebettet werden.

Paket zur Typendeklaration

Um häufig benötigte Deklarationen wie z.B. den Befehlssatz einfacher in die diversen Komponenten einzubinden, wird ein sogenanntes Package verwendet. Diese Sprachkonstruktion entspricht z.B. einer Header-Datei in einem C-Programm. Das Paket befindet sich mit den übrigen HDL-Dateien im gleichen Projektverzeichnis.

```
--- Package for the DHBW MCT controller (VHDL)

library ieee;
use ieee.std_logic_1164.all;

package MCT_Pack_uP is
-- types representing opcodes
type OPTYPE is (NOP, LSL, LSR, LDI, LDA, STR, ADD,
  SUB, ANDA, EOR, ORA, JMP, BRBC, BRBS, BCLR, ERR);

-- data type and address type
subtype P_Type is std_logic_vector(15 downto 0);
-- 16-bit of programm code
subtype D_Type is std_logic_vector (7 downto 0);
--  8-bit of data
subtype DA_Type is std_logic_vector(9 downto 0);
-- 10-bit addresses of data memory
subtype PA_Type is std_logic_vector(7 downto 0);
--  8-bit addresses of program memory

-- constants
constant P_Width  : integer := 16;
constant D_Width  : integer := 8;
constant DA_Width : integer := 10;
constant PA_Width : integer := 8;

end MCT_Pack_uP;
```

Wichtigster Bestandteil des Paketes ist die Typendeklaration der Opcodes. Diese Kürzel können im folgenden als leicht lesbare Beschreibung für das Steuerwerk und die ALU verwendet werden. Um Konflikte mit HDL-Schlüsselworten zu vermeiden, weichen die Kürzel stellenweise etwas vom Assemblercode ab (ANDR statt „and“, ORR statt „or“).

Programmspeicher

Der Programmspeicher entspricht dem ROM im vorausgegangenen Kapitel und enthält keine Besonderheiten. Für einen Test des Prozessors sollte der aus dem Assembler-Format in das Maschinenformat übersetzte Programmcode hier untergebracht werden.

Den HDL-Text des Programmspeichers finden Sie in Anhang C.

Datenspeicher

Der Datenspeicher entspricht dem RAM im vorausgegangenen Kapitel, ist allerdings nicht als Version mit separaten Leseadressen und Schreibadressen ausgeführt (Dual-Ported RAM), sondern als einfaches RAM, das entweder ausgelesen oder beschrieben wird.

Den HDL-Text des Datenspeichers finden Sie in Anhang C.

Befehlsdekoder mit Programmzähler (Instruktionsregister)

Für das Instruktionsregister mit dem Dekoder zeigt das Blockschaltbild bereits die wichtigsten Eingänge und Ausgänge. Zu den Eingängen zählen ist die nächste Instruktion aus dem Programmspeicher, sowie die noch näher zu definierenden Steuersignale des Steuerwerks. Zu den Ausgängen gehören der Opcode (als Vorgabe für das Steuerwerk), sowie die Konstante K zum Laden des Registerblocks. Ausserdem teilt das IR dem Befehlszähler noch Zieladressen für Sprungbefehle mit. Zur Vereinfachung wird der Befehlszähler (PC) mit in das IR integriert.

Als Eingänge werden definiert:

- Clock und Reset
- nächster Befehl aus dem Programmspeicher
- Steuersignale vom Steuerwerk: IR_EN (IR Enable), PC_EN (PC Enable), PC_INCR (PC inkrementieren), PC_LOADA (PC mit absoluter Adresse laden), PC_LOADR (PC mit relativer Adresse laden.

Als Ausgänge sind verfügbar:

- Adresse für den Programmspeicher (PC, Programmzähler)
- Opcode für das Steuerwerk
- Konstante K8 für den Befehl ldi Rb, K8 (Register mit Konstante laden)

Den HDL-Text für das Instruktionsregister finden Sie in Anhang C.

Im HDL-Text findet sich ein Prozess für den Dekoder. Dieser Prozess ist taktabhängig und dekodiert den Opcode gemäß der Vorgabe (siehe Abbildung 3.3). Die Konstante K8 wird gemäß der in Abbildung 3.5 gegebenen Befehlskodierung dem Programmcode entnommen. Ebenso werden die Adressen der Operanden dekodiert. Diese Informationen sind zusammen mit dem Opcode für das Steuerwerk bestimmt.

Als weiterer, ebenfalls taktabhängiger Prozess findet sich der interne Programmzähler. Je nach Steuersignal wird dieser aktualisiert oder nicht (PC_EN). Die Aktualisierung kann je nach Steuersignal als Inkrement oder als Sprungadresse ausgeführt werden. Als dritter Prozess erfolgt in der vorletzten Zeile die Übergabe des internen Zählers an den Ausgang PC.

Rechenwerk (ALU mit PSR und Akku)

Für das Rechenwerk (ALU mit Statusregister und Akku) soll das Bit C (Carry) des Prozessor-Status-Register einen Überlauf oder Unterlauf im Anschluss an eine arithmetische Operation

anzeigen, bzw. das überlaufende Bit bei Schiebeoperationen aufnehmen. Das Bit Z (Zero) soll anzeigen, dass im Anschluss an eine logische oder arithmetische Operation das betreffende Register leer ist.

Das Zero-Bit ergibt sich bei einem Rechenwerk mit Vorzeichen (Format Signed) aus einer NOR-Verknüpfung aller Bits des Registers (alle Bits müssen gleich Eins sein). Für ein vorzeichenloses Format (Unsigned) müssen alle Bits des Registers gleich Null sein, d.h. man erhält das Zero-Bit aus einer NAND-Verknüpfung. Das Innenleben der ALU muss man für eine Beschreibung mit HDL nicht weiter detaillieren, da alle benötigten logischen und arithmetischen Operationen in der Sprache vorliegen.

Den HDL-Text der ALU mit PSR finden Sie in Anhang C.

Als Eingangsgrößen der ALU mit PSR dienen der Opcode, sowie die beiden Operanden. Ausgänge sind das Ergebnis RdOut zum Abspeichern im Datenspeicher mit dem nächsten Takt, sowie die beiden Statusbits C, Z und N. Die ALU selbst funktioniert als Schaltnetz in Abhängigkeit ihrer Eingangsgrößen. Gemäß ihrer Bezeichnung als arithmetisch-logische Einheit führt die ALU nur die diesbezüglichen Befehle aus. Sprungbefehle, Registertransfers bzw. Stapeloperationen betreffen die ALU nicht.

Steuerwerk mit SP

Das Steuerwerk stellt als Top-Modul alle benötigten Verknüpfungen zwischen den Komponenten her. Als Basis für die Steuerung erhält das Steuerwerk vom Instruktionsregister den Opcode, sowie die Adressen der Operanden (Register). In der ALU sind die Eingänge und Ausgänge der Operanden fest verdrahtet. Hier genügt es, wenn das Steuerwerk den Opcode an die ALU weiter gibt.

Das Rechenwerk wird in Abhängigkeit des Opcodes vom Steuerwerk mit Hilfe von Signalen gesteuert. Hierzu gehören die Adressen der benötigten Ziele bzw. Quellen (jeweils ein Eingang zum Schreiben, sowie zwei Ausgänge zum Lesen), sowie die Selektion der korrekten Eingangsleitungen und Ausgangsleitungen.

Den HDL-Text der ALU mit PSR finden Sie in Anhang C.

Ein grosser Teil des Codes dient der Verschaltung der eingebetteten Module. Der Zustandsautomat schaltet dann in Abhängigkeit des vom IR erhaltenen Opcodes. Während die ALU den weiter gegebenen Opcode selbsttätig interpretiert, wird das Speichern und Holen der Operanden durch den Automaten gesteuert.

Da die Anzahl der Befehle die der Zustände weit überwiegt (Zustand 1: ein Takt pro Befehl, Zustand 2: zwei Takte pro Befehl), wurden die kombinatorischen Logiken (Übergangsschaltnetz und Ausgangsschaltnetz) zusammengefasst. Bei zweitaktigen Befehlen ist als Folgezustand der Zustand 2 definiert. Auf Zustand 2 folgt dann wiederum Zustand 1.

3.6. Tests

Da das Top-Modul als einzige Schnittstelle nur über einen Reset und ein Taktsignal verfügt, fällt das Testprogramm recht einfach aus und beschränkt sich auf das Generieren eines initialen Reset-Signals, sowie der periodischen Taktsignale. Den HDL-Text der Testumgebung mit einem einfachen Testprogramm im Programmspeicher finden Sie in Anhang C.

Ein aussagekräftiger Test ist hiermit allein jedoch nicht möglich. Das eigentliche Testprogramm ist das Assembler-Programm für den Mikrocontroller, beispielsweise das Programm aus Abschnitt 3.1.

Um dieses Programm in den Programmspeicher zu laden, ist eine Umwandlung des Assembler-Codes in Maschinensprache erforderlich. Das Testprogramm wird anschliessend vorab in den Programmspeicher geladen (und somit als ROM gespeichert). Der eigentliche Funktionstest besteht in der Analyse des Ablaufs des Testprogramms im Simulator,

Folgende Abbildung zeigt einen Test im Simulator mit einem auf diese Weise in den Programmspeicher geladenen Testprogramm. In der Zeile Opcode erkennt man die Sequenz der Befehle aus dem Testprogramm. Ebenso dargestellt sind die für jeden Befehl dekodierten Adressen der Operanden. Einige Schwierigkeiten bei der Interpretation des Zeitdia-gramms ergeben sich durch die Pipeline-Struktur des Controllers. Jeweils ein Takt wird benötigt, um den Befehl aus dem Programmspeicher zu laden, ein weiterer Takt für das Dekodieren des Befehls, ein weiterer Takt für die Ausführung des Befehls. Der Befehlszähler eilt also der Ausführung stets zwei Schritte voraus. Sofern der Mikrocontroller die korrekte Sequenz der Befehle für die vorgegebenen Werte wieder gibt, darf man davon ausgehen, dass die Implementierung funktioniert.

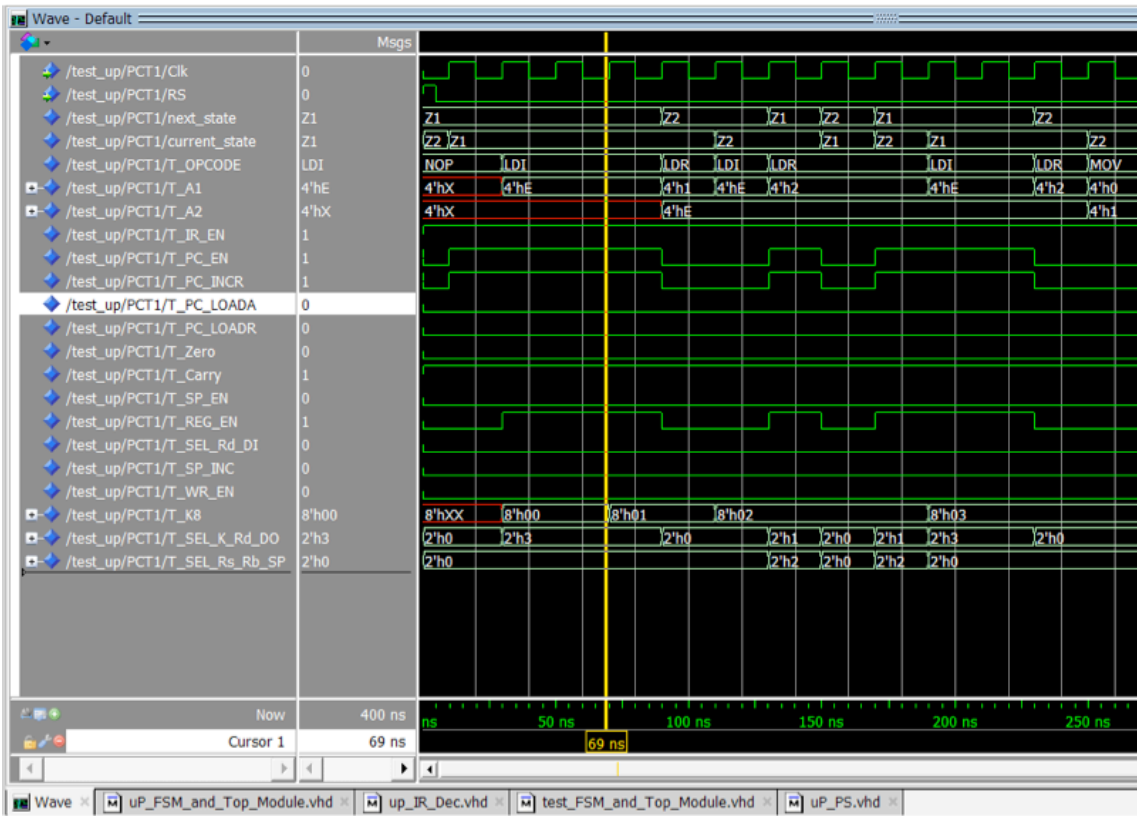


Bild 3.19 Testlauf im Simulator

4. Erweiterungen des Mikroprozessors

4.1. Unterprogramme

Durch strukturierte Programmierung lassen sich Programme übersichtlicher gestalten. Für häufig verwendete Aktivitäten lassen sich beispielsweise in Makros bzw. Unterprogrammen unterbringen. Bei Makros werden wiederkehrende Aktivitäten für den Assembler mit speziellen Anweisungen geklammert. Im Programmtext wird dann stellvertretend für die Aktivität der Name des Makros aufgerufen. Diese Methode hat auf den Prozessor keinerlei Einfluss. Der Assembler fügt jedes Mal, wenn das Makro aufgerufen wie, den Programmcode des Makros in den Maschinencode ein.

Eine andere Methode zur strukturierten Programmierung ist die Verwendung von Unterprogrammen. Auch hier wird eine wiederkehrende Aktivität als Einheit verpackt, statt als Makro allerdings als Unterprogramm. Im Assemblerprogramm wird dort, wo die besagte Aktivität benötigt wird, der Name des Unterprogramms aufgerufen. Es ist hierzu also eine Erweiterung des Befehlssatzes des Prozessors erforderlich.

Die Realisierung erfolgt so, dass der Assembler den Programmcode des Unterprogramms an einer geeigneten Stelle im Programmspeicher ablegt. Bei Aufruf des Unterprogramms erfolgt ein Sprung an diese Adresse. Zur Unterstützung von Unterprogrammen sind im Prozessor folgende Erweiterungen erforderlich:

- Befehl zum Aufrufen eines Unterprogramm (aus dem Hauptprogramm)
- Befehl zur Rückkehr ins Hauptprogramm (aus dem Unterprogramm)
- Rettung (Speicherung) des Programmzählers beim Sprung ins Unterprogramm, damit das Hauptprogramm bei der Rückkehr an dieser Stelle weiter arbeiten kann.
- Rettung von Zwischenergebnissen (Inhalt des Akkus) des Hauptprogramms, die bei der Rückkehr ins Hauptprogramm wieder benötigt werden.

Abstrakt ausgedrückt muss also der Kontext (Befehlszähler und ggf. Inhalt des Akkus) bei Start des Unterprogramms gesichert werden. Bei der Rückkehr aus dem Unterprogramm wird dieser Kontext wieder in den Prozessor geladen. Diese Auslagerung und Wiederherstellung des Kontexts wird in englischer Sprache als Context Switching bezeichnet. Je nach Art des Prozessors gehören zum Kontext neben der Rücksprungadresse (Programmzähler) und dem Akku als einzigem Register auch sonstige benötigte Registerinhalte. Folgende Abbildung zeigt den Kontrollfluss bei Makros bzw .Unterprogrammen.

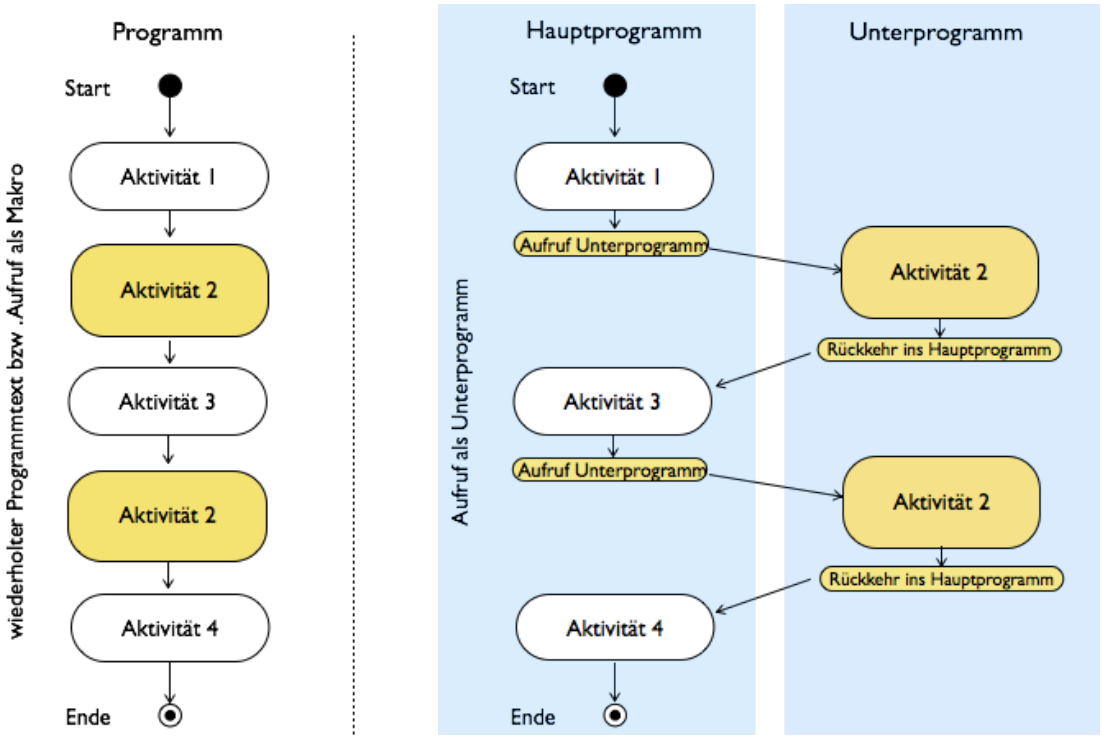


Bild 4.1 Kontrollfluss bei Makros und Unterprogrammen

In Falle der Verwendung von Makros bleibt der Programmablauf im gezeigten Beispiel linear und folgt dem Kontrollfluss. Bei der Verwendung von Unterprogrammen bleibt der Kontrollfluss zwar

linear, der Programmablauf verzweigt jedoch in ein Unterprogramm. Das Unterprogramm löst die Rückkehr ins Hauptprogramm aus. Aus dem Unterprogramm liesse sich ein weiteres Unterprogramm aufrufen. Auf diese Weise entsteht ein ineinander verschachtelter Ablauf.

Wohin wird der Kontext des aufrufenden Programms (Programmzähler, ggf. Akku-Inhalt) ausgelagert? Die nahe liegende Möglichkeit ist der Datenspeicher. Da bei der Rückkehr die dort abgelegten Informationen in genau der umgekehrten Reihenfolge benötigt werden, wie sie abgelegt werden, lässt sich dieser Bereich als Stapel (engl. Stack) organisieren: neue Informationen werden übereinander gestapelt und bei Bedarf von oben nach unten wieder abgetragen. Wie wird der ausgelagerte Bereich (Stapel) adressiert? Hierzu wird ein spezielles Register definiert, das als Zeiger auf den nächsten freien Platz auf dem Stapel dient: der Stapelzeiger (engl. Stack Pointer)

Stapel und Stapelzeiger

Folgende Abbildung zeigt die Funktionsweise des Stapels. Vor Aufruf des Unterprogramms zeigt der Stapelzeiger SP auf das nächste freie Feld im Stapel. Bei Aufruf des Unterprogramms wird dort zunächst der aktuelle Stand des Befehlszeigers abgelegt (genauer: die Adresse des auf den den Aufruf folgenden Befehls). Der Stapelzeiger wird anschliessend inkrementiert. Es folgt die Ablage des Akkus aus dem Hauptprogramm. Der Stapelzeiger wird erneut inkrementiert. Diese Situation ist auf der rechten Seite der Abbildung zu sehen.

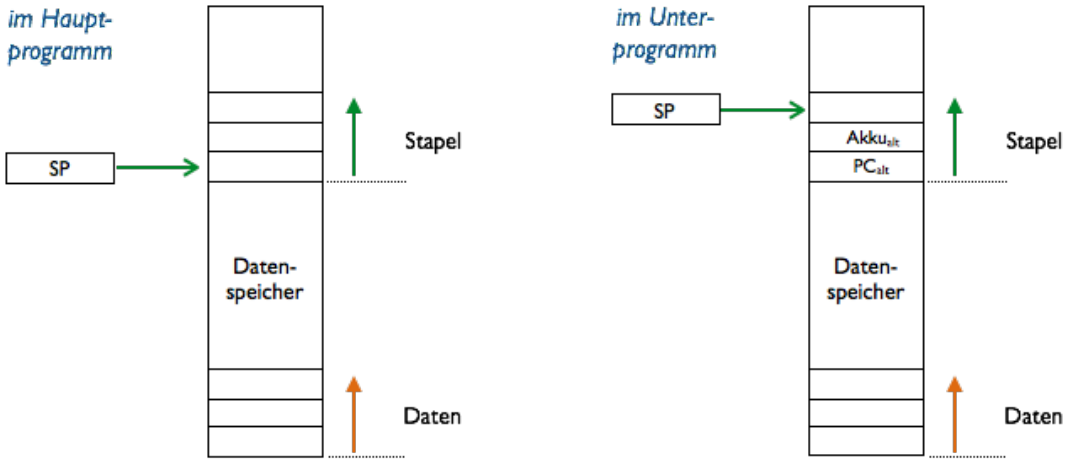


Bild 4.2 Funktionsweise des Stapels

Bei der Rückkehr in das Hauptprogramm wird zunächst der Akku-Inhalt wieder hergestellt (vom Stapel zurück in den Akku transferiert). Anschliessend übernimmt der Befehlszähler den Stand aus dem Stapel. Somit erfolgt ein Rücksprung ins Hauptprogramm an die Stelle unmittelbar nach dem Aufruf des Unterprogramms.

Stapel und Daten teilen sich den Datenspeicher, haben jedoch unterschiedliche Einsprungadressen. Beim hier betrachteten Musterprozessor wurde die Startadresse der Daten willkürlich an der Adresse Null (d.h. DS(0)) festgelegt. Der Stapel beginnt mit der Startadresse 32 (d.h. DS(32). In der Praxis sind je nach Prozessor ebenfalls solche Festlegungen erforderlich.

Neue Befehle

Es wird folgende Arbeitsaufteilung zwischen Hauptprogramm und Unterprogramm gewählt:

- Hauptprogramm: ruft Unterprogramm auf. Hierzu wird ein neuer Befehl definiert: call Ziel. Der Ablauf ist wie bei einem Sprungbefehl an die Adresse Ziel, jedoch wird zusätzlich der Stand des PC (Programmzählers) auf dem Stapel gesichert.
- Unterprogramm: sichert Inhalt des Akkus. Hierzu wird ein neuer Befehl definiert: push. Als einziges Register dient der Akku als impliziter Operand. Sein Inhalt wird auf dem Stapel gespeichert, der Stapelzeiger inkrementiert.
- Unterprogramm: erledigt seine Aktivität.
- Unterprogramm: Bereitet die Rückkehr ins Hauptprogramm vor. Hierzu wird der Inhalt des Akkus aus dem Stapel zurück in den Akku transferiert. Als Gegenspieler des Befehls push wird hierzu der Befehl pop definiert. Der Stapelzeiger wird dekrementiert und zeigt auf den alten Stand des Akkus im Stapel. Implizites Ziel der Operation ist der Akku als einziges Register.
- Unterprogramm: löst die Rückkehr ins Hautprogramm aus. Hierzu wird ein neuer Befehl definiert: ret. Auch dieser Befehl funktioniert wie ein Sprungbefehl, wobei die Zieladresse der auf dem Stapel abgelegte Stand des PC ist. Der Stapelzeiger wird dekrementiert. Der Befehlszähler wird auf den alten Stand zurück gesetzt.
- Hauptprogramm: setzt seine Tätigkeit fort.

Folgendes Beispiel zeigt den Aufruf eines Unterprogramms.

```
; ASL (Assembler Language) für den DHBW MCT-Mikroprozessor
; Muster zum Aufruf eines Unterprogramms
;
main:         ldi  0            ; Akku <= 0
              st  $01           ; DS(1) <= 0
              call count        ;
              call count        ;
              call count        ;
              ...
; count.asm Muster für ein Unterprogramm
count:        push              ; (SP)<= Akku, SP <= SP +1
              ldi 1             ; Akku <= 1
              add $01           ; Akku <= Akku + DS(1)
              st  $01           ; DS(1) <= Akku
              pop               ; SP <= SP -1; Akku <= (SP)
              ret               ; return
```

Übung 4.1: Erläutern Sie die Funktion des Unterprogramms. Was geschieht bei jedem Aufruf?

Übung 4.2: Erläutern Sie den Ablauf im Hauptprogramm. Skizzieren Sie die Inhalte des Stapels mit jedem Aufruf. Skizzieren Sie den Inhalt von DS(1). Skizzieren Sie den Inhalt des Akkus im Hauptprogramm (vor und nach jedem Aufruf des Unterprogramms).

Das Musterprogramm verwendet bereits die neuen Befehle. Folgende Abbildung zeigt die für die Implementierung gewählte Kodierung.

Erweiterungen

Befehlssatz			Beschreibung	Prozessor Status Register (PSR) Flags				
Assembler	Kürzel	Opcode		Zero (Z)	Carry (C)	oVerflow (V)	Sign (S)	Neg. (N)
push	PUSH	E	(SP)<= Akku; SP <= SP+1	-	-	-	-	-
pop	POP	F	SP<SP-1; Akku<= (SP)	-	-	-	-	-
call K	CALL	10	(SP)<= PC; SP<=SP +1; jump to address K	-	-	-	-	-
ret	RET	11	SP<= SP-1; PC<=(SP); jump to PC	-	-	-	-	-

Erweiterungen

nop, push, pop, ret

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					(leer)										

Sprungbefehle
jmp, call

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					X	X	X	K8 (Zieladresse)							

Bild 4.3 Neue Befehle und Befehlsformate

Mit den neuen Befehlen wird die Liste der bisher vorhandenen Befehle verlängert. Der Befehlssatz umfasst nun also 18 Befehle (hexadezimal x11 plus 1). Die Befehle Push und Pop verhalten sich wie Speicherbefehle, wobei im Vergleich zu Ld und St der Operand wiederum implizit der Akku ist, jedoch die direkte Adresse entfällt, da diese implizit durch den Stapelzeiger gegeben ist. Die Befehle Call K und Ret verhalten sich wie Sprungbefehle, wobei Call K den Sprung vom Haupt-programm bzw. dem übergeordneten Programm aus durchführt, und Ret vom Unterprogramm aus startet.

Für die neuen Befehle sind keine neuen Befehlsformate erforderlich. Push, Pop und Ret haben keine expliziten Operanden und sind wie die Leerlaufoperation Nop kodiert. Der Befehl Call K folgt der Kodierung des absoluten Sprungbefehls Jmp K.

Erweiterungen des Prozessors

Neben den neuen Funktionen durch den erweiterten Befehlssatz ist der Prozessor nun mit einem Stapelzeiger zu erweitern, wie folgende Abbildung zeigt.

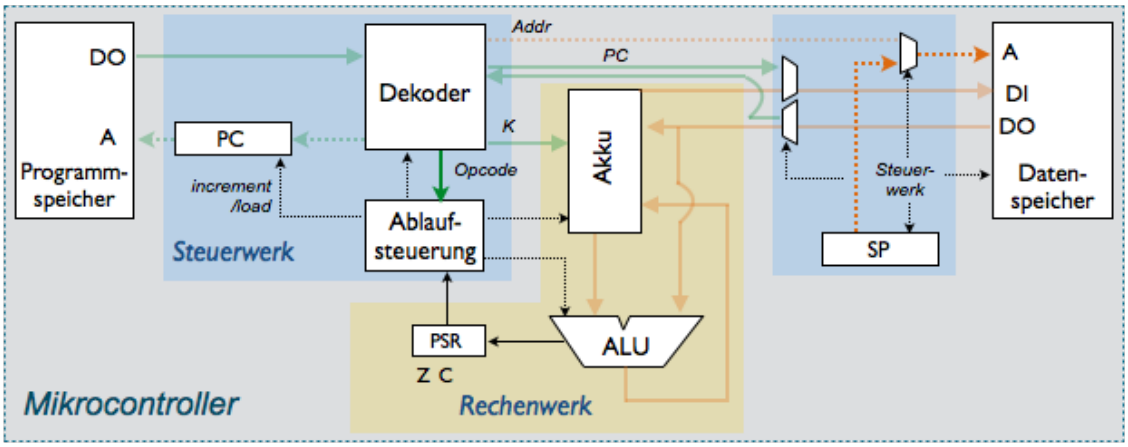


Bild 4.4 Implementierung des Stapels und Stapelzeigers

Der Stapelzeiger (SP) ist als Register ausgeführt, ebenso wie der Befehlszeiger (PC). Ziel des Stapelzeigers ist jedoch der Datenspeicher, nicht der Programmspeicher. Das Inkrementieren zw. Dekrementieren des Stapelzeigers übernimmt die Ablaufsteuerung im Steuerwerk. Die Ablaufsteuerung stellt auch die Weichen für die korrekte Adresse am Datenspeicher durch den hierfür

vorgesehenen Multiplexer. Push- und Pop-Operationen erfolgen nun wie Ladezugriffe oder Speicherzugriffe vom Akku aus, jedoch mit der Adresse aus dem Stapelzeiger (SP).

Das Laden bzw. Speichern des Programmzählers für die Befehle Call K bzw. Ret übernimmt ebenfalls die Ablaufsteuerung im Steuerwerk. Die diesbezüglichen Datenpfade sind im Blockdiagramm nicht dargestellt. Auch diese Datentransfers verhalten sich wie Ladeoperationen bzw. Speicheroperationen des Akkus, wobei die Adressen vom Stapelzeiger bereit gestellt werden.

Aus dem Blockdiagramm lassen sich für die einzelnen Befehle folgende Taktzyklen ableiten:

- Call K: 2 Takte: (1) Sperren der Pipeline, (2) Speichern und neu Laden des PC, Aktivieren der Pipeline
- Ret: 2 Takte: (1) Sperren der Pipeline und Auslesen des Stacks, (2) neu Laden des PC, Aktivieren der Pipeline
- Push: 1 Takt (wie Speichern des Akkus mit St Addr)
- Pop: : 2 Takte (wie Laden des Akkus mit Ld Addr)

Programm mit Stapeloperationen

Um den Ablauf des Musterprogramms mit den Operationen zum Sichern und Wiederherstellen des Akkus auf dem Stack, sowie für den Aufruf eines Unterprogramms genauer zu verfolgen, wird der Programmtext in die Maschinensprache mit den genauen Adressen der einzelnen Instruktionen übersetzt. Folgender Abschnitt zeigt den Programmtext.

```
-- Maschinensprache für den DHBW MCT-Mikroprozessor
-- Aufruf des Unterprogramms „count": ab Zeile 12

 0 => "0001100000000000", -- LDI 0 : Hauptprogramm
 1 => "0010100000000010", -- STR $1
 2 => "1000000000001100", -- CALL Count (Zeile 12)
 3 => "0000000000000000",  -- NOP_3
 4 => "1000000000001100", -- CALL Count (Zeile 12)
 5 => "0000000000000000",  -- NOP
 6 => "1000000000001100", -- CALL Count (Zeile 12)
 7 => "0000000000000000",  -- NOP
 8 => "0000000000000000",  -- NOP
 9 => "0000000000000000",  -- NOP
10 => "0000000000000000",  -- NOP
11 => "0000000000000000",  -- NOP
12 => "0111000000000000", -- PUSH: Unterprogramm Count
13 => "0001100000001000", -- LDI 1
14 => "0011000000000010",  -- ADD $1
15 => "0010100000000010", -- STR $1
16 => "0111100000000000", -- POP
17 => "1000100000000000", -- RET
18 => "0000000000000000",  -- NOP_1  (ab hier sind nur noch NOPs im PS)
19 => "0000000000000000",  -- NOP_2
```

Mit Zeile Null startet das Hauptprogramm. Die einzige Aktion besteht darin, den Wert Null in die Speicherzelle mit der Adresse 1 zu speichern. Dann erfolgt der Aufruf des Unterprogramms Count, das im Beispiel bei Zeile 12 beginnt. Da der Aufruf des Unterprogramms einen Sprungbefehl ausführt, wird nach dem Aufruf von Call eine Leerlaufanweisung NOP eingefügt. Diese Anweisung ist zum

Zeitpunkt der Dekodierung des Befehls Call bereits aus dem Programmspeicher ausgelesen worden und wird daher als nächste Anweisung ausgeführt, unabhängig von der neuen Adresse im Programmzähler.

Der Programmzähler sollte nun den Befehl in Zeile 12 adressieren, d.h. im Anschluss an den NOP in Zeile 5 sollte der Befehl Push in Zeile 12 dekodiert werden. Die Instruktionen im Unterprogramm count dienen dazu, jeden Aufruf zu zählen und in Speicherzelle 1 zu hinterlegen. Hierzu wird zunächst der Akku mit dem im Hauptprogramm ggf. benötigten Inhalt auf dem Stapel gespeichert (Befehl Push). Anschliessend wird der alte Stand aus Speicherzelle 1 ausgelesen, im Akku inkrementiert und in Speicherzelle zurückgeschrieben. Vor dem Rücksprung ins Hauptprogramm mit Hilfe der Anweisung Ret erfolgt die Wiederherstellung des Akkus vom Stapel mit Hilfe der Anweisung Pop.

Der Rücksprung sollte nun an die Adresse 3 erfolgen, d.h. den nächsten Befehl, der im Hauptprogramm auf den Aufruf Call folgt. Dann erfolgt in Zeile 4 der erneute Aufruf des Unterprogramms Count. Der Rücksprung sollte dieses Mal an die Adresse 5 erfolgen (als nächster Befehl nach dem Programmaufruf). In Zeile 6 erfolgt schliesslich der dritte Aufruf des Unterprogramms. Der Rücksprung sollte nun an die Adresse 7 erfolgen. Das Zeitdiagramm auf der folgenden Seite zeigt den Ablauf des Musterprogramms.

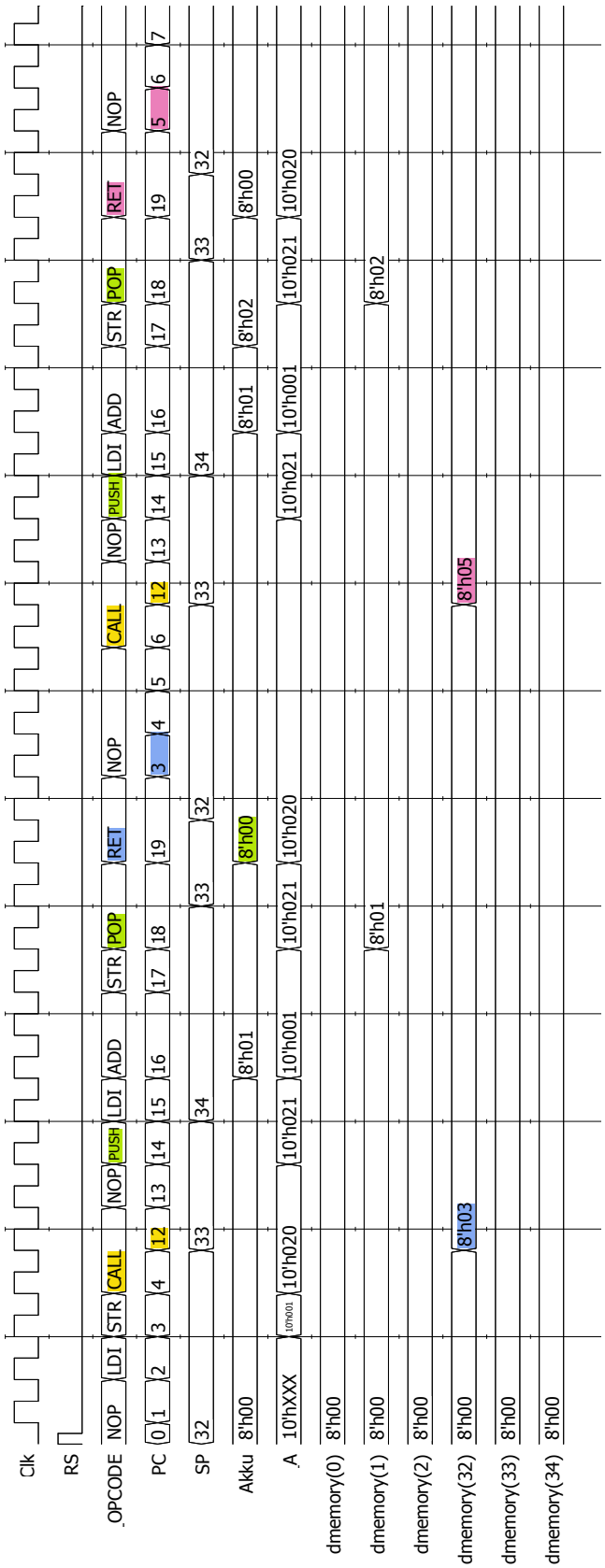


Bild 4.5 Ablauf des Musterprogramms mit Stapeloperationen

Im Zeitablauf dargestellt sind der Opcode, die Stände des Programmzählers (PC) und Stapelzeigers (SP), sowie der Inhalt des Akkus. Darunter findet sich die Adresse A des Datenspeichers, sowie die Inhalte der Speicherzellen 0 bis 2, sowie 32 bis 33. Der Stapelzeiger ist hier so eingestellt, dass er im initialen Zustand auf die Speicherzelle 32 zeigt. Der Stapel wächst also von Adresse 32 aufwärts. Der Ablauf des Programms gestaltet sich wie folgt:

- Aufruf des Unterprogramms mit Opcode Call (gelbe Markierung): Der Programmzeiger zeigt bereits auf die Adresse 4, wenn Call aktuell dekodierte wird (der Befehl Call findet sich an Adresse 2). Das Vorauseilen des PC um 2 ist ein Effekt der bereits erläuterten Fließbandverarbeitung. Die Adresse 4 im Programmzähler wird nun überschrieben mit der Adresse 12 des Unterprogramms. Zur gleichen Zeit wird der alte Stand des PC in die oberste Zeile des Stapels nach DS(32) geschrieben (blaue Markierung). Der Stapelzeiger (SP) wird inkrementiert und zeigt nun auf DS(33).
- Dekodieren des Befehls Push (grüne Markierung) an Adresse 12: Der Programmzähler zeigt hier bereits auf Adresse 14, eilt also wiederum 2 Zeilen vor. Die Leerlaufanweisung NOP vor dem Befehl Push stammt aus Zeile 3 und ist ein Effekt der Fließbandverarbeitung. Mit dem Befehl Push wir der Inhalt des Akkus (Null) im Stapel in die Speicherstelle DS(33) geschrieben (die nach der Initialisierung bereits den Wert 0 hatte). Der Stapelzeiger wird inkrementiert und zeigt nun auf DS(34)
- Ausführung des Unterprogramms: Das Unterprogramm zählt seinen ersten Aufruf. Als Ergebnis wird der Wert 1 nach DS(1) geschrieben. Vor dem Rücksprung ins Hauptprogramm wird mit der Anweisung Pop der alte Inhalt des Akkus vom Stapel gelesen (aus DS(34)) und somit der Akku wieder hergestellt (siehe grüne Markierung). Der Stapelzeiger wird hierbei dekrementiert und zeugt nun auf DS(33) als nächsten freier Platz im Stapel.
- Rücksprung ins Hauptprogramm mit Opcode Ret (blaue Markierung): Aus dem Stapel (Speicherzelle DS(32)) wird der alte Stand des Programmzählers ausgelesen und wieder hergestellt. Der PC zeigt nun auf Zeile 3, d.h. den nächsten Befehl nach dem Aufruf des Unterprogramms in Zeile 2.
- Nächster Aufruf des Unterprogramms mit Opcode Call (gelbe Markierung): Der Programmzähler steht bei 6, wenn Zeile 4 (Call) dekodiert wird, und wird nun wieder auf den Programmtext des Unterprogramms in Zeile 12 gesetzt.
- Der weitere Ablauf folgt den bereits beschriebenen Mechanismen. Jeder Aufruf des Unterprogramms wird durch das Unterprogramm gezählt und in DS(1) gespeichert. Der Rücksprung erfolgt jeweils auf den Befehl, der dem letzten Aufruf folgt. Beim zweiten Aufruf des Programms also nach Zeile 5 (siehe violette Markierung).

Übung 4.3: Skizzieren Sie den Ablauf folgender Befehle im Blockdiagramm: (1) Call K, (2) Ret, (3) Push, (4) Pop. Erläutern Sie den jeweiligen Bedarf an Taktzyklen. Hinweise: Startpunkt ist der dekodierte Befehl. Verwenden Sie die Vorlagen in Anhang A.

Übung 4.4: Erläutern Sie den zeitlichen Ablauf der Befehle aus Übung 4.3 mit Hilfe eines Zeitdiagramms nach dem Muster in Bild 4.5. Hinweis: Verwenden Sie eine handschriftliche Skizze bzw. ein Programm zur Tabellenkalkulation.

Übung 4.5: Rekursive Programmierung. Erläutern Sie die Verwendung des Stapels bei einem rekursiven Programmaufruf. Verwenden Sie ein Beispiel, z.B. die Berechnung der Fakultät n! Testen Sie den rekursiven Aufruf durch Implementierung des Beispiels.

4.2. Unterbrechungssystem

Seminararbeit S4: Interrupts. Erweitern Sie Ihren Mikrocontroller um ein Unterbrechungssystem. Testen Sie Ihren Entwurf an einem Beispiel. Hinweis: Beim Unterbrechungssystem erfolgt der Aufruf eines Unter-programms (der Interrupt-Service Routine) nicht durch einen regulären Programmaufruf (CALL, RET), sondern durch ein Ereignis an einem Eingangsport (Interrupt). Die Mechanismen des Aufrufs eines Unterprogramms können somit wiederverwendet werden.

4.3. Ports für Geräte

Seminararbeit S5: Digitale Eingänge und Ausgänge. Erweitern Sie Ihren Mikrocontroller um I/O Ports. Testen Sie Ihren Entwurf in einer Simulation mit Hilfe eines Beispiels.

4.4. Timer

Seminararbeit S6: Mikrocontroller wie der im Arduino verwendetet ATTiny bzw ATMega (siehe Mikrocom-putertechnik 1 und 2) verfügen über Timer, mit deren Hilfe sich Programme zeitlich steuern lassen. Recherchieren Sie die Realisierung der Timer-Funktion für einen gängigen Mikrocontroller. Erweitern Sie Ihren Mikrocontroller um Timer. Testen Sie Ihren Entwurf in einer Simulation mit Hilfe eines Beispiels.

4.5. Serielle Schnittstellen

Seminararbeit S7: Mikrocontroller wie der im Arduino verwendetet ATTiny bzw ATMega (siehe Mikrocom-putertechnik 1 und 2) besitzen serielle Schnittstellen wie z.B. SPI oder I^2C, mit deren Hilfe sich externe Geräte über passende Bausteine anbinden lassen (z.B. Pulsweitenmodulation, Port-Extender, Anzeigen, ADCs). Erweitern Sie Ihren Mikrocontroller um eine serielle Schnittstelle, z.B. SPI. Testen Sie Ihren Entwurf in einer Simulation mit Hilfe eines Beispiels.

5. Signalprozessoren

Als Signalprozessoren (abgekürzt DSP für Digitale Signal-Prozessoren) werden spezialisierte Mikroprozessoren bezeichnet, die mit speziellen Rechenwerken ausgestattet sind und die zur Signal-verarbeitung eingesetzt werden. Zu den Anwendungsgebieten gehören Audiosignale, Videosignale, Bildverarbeitung, Radar, Sonar etc. Viele der genannten Anwendungen lassen sich inzwischen auch mit leistungsfähigen Mikroprozessoren für Server abdecken.

Aus der Perspektive der programmierbaren Bausteine (FPGA) spielen solche Unterschiede in der Terminologie bzw. Technologie ohnehin keine Rolle. Anwendungsorientierte Rechenwerke in Kombination mit der Programmierbarkeit eines Mikrocontrollers sind hier auf jeden Fall interessant.

5.1. Architektur

Der Mikrocontroller aus dem vergangenen Abschnitt ist nur mit einem recht bescheidenem Rechenwerk ausgestattet: Die Arithmetisch-Logische Einheit (ALU) beherrscht nur einige Schiebe-

operationen, logische Verknüpfungen, sowie Addition und Subtraktion von Festkommazahlen. Eine Multiplikation muss bereits als Makro oder Unterprogramm mit Hilfe wiederholter Verschiebeoperationen und Additionen realisiert werden.

Um Berechnungen wie die in Abschnitt 2 dargestellten Faltungssumme ausführen zu können, soll das Rechenwerk des Mikrocontrollers mit der MAC Unit aus Abschnitt 2 ausgerüstet werden. Ausserdem wird ein Unterbrechnungsregister (Interrupt-Register) eingeführt, mit dem ein externes System wie z.B. ein ADC (Analog-Digital-Wandler) signalisieren kann, ob ein neuer Abtastwert vorliegt. Um den Abtastwert aufzunehmen, bzw. um Rechenergebnisse nach Aussen zu kommunizieren, wird der Mikrocontroller ausserdem um Eingabeports und Ausgabeports ergänzt. Folgende Abbildung zeigt einen Überblick über die Architektur.

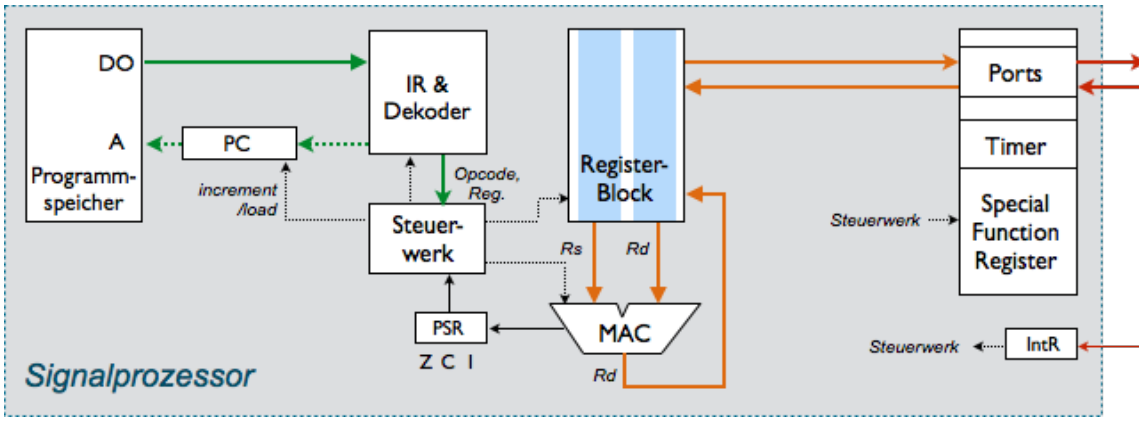


Bild 4.1 Architektur des Signalprozessors

Alle zusätzlichen Komponenten lassen sich natürlich als Erweiterungen des in Abschnitt 3 vorgestellten Mikrocontrollers realisieren (zusätzliches Rechenwerk mit zusätzlichen Registern). Damit die HDL-Texte jedoch überschaubar bleiben, wurde der Umfang auf die zur Demonstration der Anwendung des Filteralgorithmus nötigen Komponenten reduziert. Insbesondere der Registerblock wurde durch eine vereinfachte, anwendungsorientierte Struktur ersetzt. Im Registerblock ist jeweils eine Bank von 16 Registern Rs_0 bis Rs_{15} vorgesehen, die die Filterkoeffizienten aufnehmen können, sowie eine Bank von 16 Registern Rd_0 bis Rd_{15} für die eingelesenen Abtastwerte.

Auf den Datenspeicher wurde in der gewählten Untermenge verzichtet. Stattdessen findet sich ein Bereich spezieller Register (SFR - Special Functional Register), in denen Timer, sowie Ports für Schnittstellen nach aussen untergebracht wurden. Über einen Eingangsport kann ein externes Gerät (AD-Wandler) einen Neuen Wert in den Prozessor schreiben. In diesem Fall wäre die Verfügbarkeit des neuen Wertes über ein Interrupt-Bit zu signalisieren. Hierfür steht ein weiteres Register zur Verfügung. Ein fertiges Rechenergebnis kann in einen Ausgangsport des SFR übertragen werden und dort von einem angeschlossenen Gerät abgeholt werden. Die Eingabe und Ausgabe ist so in den Adressraum des Datenspeichers eingebunden (für das SFR verwendete Adressen müssen im Datenspeicher ausgenommen werden).

Das Unterbrechungssystem ist in der gewählten Form etwas vereinfacht ausgefallen. Das Steuerwerk wird über das Eintreffen eines neuen Wertes benachrichtigt. Das Prozessor-Status-Register wurde um ein Bit erweitert, dass anzeigt, ob Unterbrechungen zugelassen sind. Beim Ausführen einer Unterbrechnungsroutine wird dieses Bit gelöscht, um zu verhindern, dass in dieser

Zeit eine weitere Unterbrechung auftritt. Im Falle einer Unterbrechung würde allerdings der Befehlszähler auf dem Stapel gesichert und die Sprungadresse der Unterbrechung geladen. Nach Ausführen der Unterbrechungsroutine kehrt der Prozessor wieder in das ursprüngliche Programm zurück, indem er den Befehlszähler wiederum vom Stapel lädt. Diese Erweiterungen sind mit dem in Abschnitt 3 verwendetet Stapelzeiger und Datenspeicher möglich. Der Einfachheit halber wurde hier auf diesen Aufwand verzichtet, d.h. der Prozessor bedient nur den Filteralgorithmus und steht für keine anderen Aufgaben zur Verfügung.

Befehlssatz

Folgende Tabelle zeigt den Befehlssatz des Prozessors. Der Befehlssatz erweitert den Befehlsumfang aus Abschnitt 3. Allerdings sind nur die Befehle dargestellt, die für das Filter benötigt werden.

Befehlssatz			Beschreibung	Prozessor Status Register			
Assembler	Kürzel	Opcode		Zero (Z)	Carry (C)	Interrupt (I)	...
nop	NOP	0	No Operation	-	-	-	
ldi K	LDI	4	Akku<=K11, load immediate 11bit co	-	-	-	
and Rd, Rs	ANDR	B	Rd<=Rd AND Rr, logical AND	x	-	-	
brbc bit, offset	BRBC	F	branch if bit clear (C, Z)	-	-	-	
brbs bit, offset	BRBS	10	branch if bit set (C, Z)	-	-	-	
bclr bit	BCLR	11	clear bit in PSR	x	x	x	
bset bit	BSET	12	set bit in PSR	x	x	x	
mac R, *Rb	MAC	13	akku<= akku + Rd*Rs; R<= akku	x	x	-	
mul Rd, Rs	MUL	14	Rd<= Rd * Rs, multiply	x	x	-	
dec Rd	DECR	15	Rd<= Rd-1, decrement	x	x	-	
clear akku	CLRA	16	akku <= 0	x	-	-	
wait K8	WAITI	17	wait for interrupt K8	-	-	-	
shift	SHIFT	18	shifts register bank Rd down	-	-	-	
in Rd, SFR	INP	19	Rd<= SFR, read from port	-	-	-	
out Rs, SFR	OUTP	1A	SFR<= Rs, write to port	-	-	-	

Abbildung 4.2 Befehlssatz des Signalprozessors

Die Erweiterungen betreffen einerseits das Unterbrechungssystem: Das Interrupt-Bit im PSR zeigt an, ob Unterbrechungen zugelassen sind. Um das Bit zu setzen, bzw. zurückzusetzen, wurden die Befehle bclr und bset eingeführt, die sich allgemein auf die Status-Bits im PSR anwenden lassen. Ebenfalls neu eingeführt wurde der Befehl wait K8, der den Prozessor in den Wartezustand versetzt, bis das mit der Konstante K8 angegebene Bit im Interrupt-Register gesetzt wurde.

Andere Erweiterungen betreffen das Ein- und Ausgabesystem: Der Befehl „in Rd, SFR“ liest den unter der Adresse SFR befindlichen Port ins angegebene Register Rd ein. Der Befehl „out SFR, Rs“ beschreibt den unter der Adresse SFR befindlichen Port mit dem Inhalt von Rs. Alle weiteren Befehle dienen der Berechnung: mit Hilfe von „dec Rd“ lässt sich das Register Rd als Zähler verwenden, der Befehl mac multipliziert den Inhalt der angegebenen Register (wobei registerindirekt adressiert wird) und sammelt das Ergebnis in einem Akkumulator-Register.

Um die Indizierung zu vereinfachen, wurde außerdem eine Schieberegister-Operation eingeführt, die die gesamte Register-bank Rd um eine Position nach unten verschiebt, d.h. Rd(15) <= Rd(14) usw. Auf diese Weise kann nach Beenden der Berechnung der nächste Eingangswert immer wieder an Position Rd(0) geschrieben werden. Zusätzlich zu dem spezialisierten mac-Befehl wurde noch ein Befehl für eine reguläre Festkomma-Multiplikation eingeführt.

Filteralgorithmus

Beim Filteralgorithmus geht man davon aus, dass alle Koeffizienten in die Register Rd geladen wurden. Mit dem erweiterten Befehlssatz aus Abschnitt 3 wäre natürlich auch ein Laden aus dem Datenspeicher möglich. Der Ablauf des Algorithmus mit den oben genannten neuen Befehlen wäre somit wie folgt.

```
; ASL (Assembler Language) für den DHBW PCT1-Mikroprozessor
; FIR Filteralgorithmus
; Voraussetzungen: Koeffizienten h0 bis h15 in Rs0 bis Rs15,
; Rd0 bis Rd15 = 0, erster Wert wird nach Rd0 geschrieben
;
             ldi R15H, $0      ; R15_High<= 0
interrupt:   wait $01          ; wait for interrupt 1
             bclr $4           ; disable interrupts
             in R32, PortA     ; Rd0<= x(0)
             clra              ; akku<= 0
             ldi R15L, $0F     ; R15_Low<= 15 (counter)
next:        mac R14, *R15     ; akku<= akku + Rd*Rs; R14<= akku
             dec R15           ; counter = counter - 1
             tst R15           ; Zero = true
             brne next         ; branch to end
end:         out R14, PortB    ; PortB<= result
             shift Rd          ; shift Rd register bank
             tst R15           ; Zero = true
             bset $4           ; enable interrupts
             breq interrupt    ;
```

Das Programm geht von einigen Voraussetzungen aus, die bei der Organisation der Register berücksichtigt werden müssen. Folgende Abbildung zeigt die diesbezügliche Organisation.

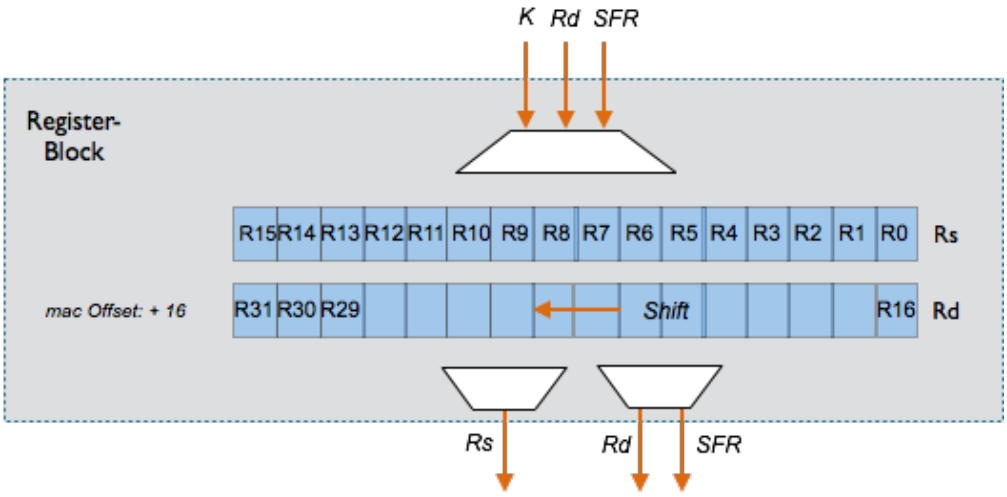


Abbildung 4.3 Organisation des Registerblocks

Die ursprünglichen 16 Register R0 bis R15 wurden beibehalten. Allerdings wurde die Anbindung des Datenspeichers und Stapelzeigers in der Darstellung nicht berücksichtigt. Zur Aufnahme der Filterkoeffizienten werden Register R0 bis R16 verwendet, die mit Registerbank Rs bezeichnet sind.

Zur Aufnahme der Eingangswerte dienen die Register R16 bis R31 (Registerbank Rd). Für diese Registerbank ist die Schieberegister-Operation Shift Rd realisiert.

Da bei der gewählten 5-Bit Adressierung nur insgesamt 32 Register verfügbar sind, werden die Register R14 und R15 nicht für die Speicherung von Koeffizienten verwendet, sondern als allgemeine Register. Um die Adressierung beim Befehl mac zu Vereinfachen, arbeitet dieser Befehl die Adressdifferenz von +16 selber ein, wenn in oben gezeigtem Programmausschnitt der Befehl als „mac R14, *R15“ aufgerufen wird. Hierbei wird der Inhalt von Register R15 als Zähler verwendet. Die Ergebnisse jeder Operation werden im Register R14 abgelegt.

5.2. Parallelverarbeitung

Das bisherige Rechenwerk besitzt zwar einen Festkomma-Multiplizierer mit Akkumulator, arbeitet aber immer noch völlig sequenziell. Unter Parallelverarbeitung lassen sich folgende Alternativen verstehen:

- Das Rechenwerk arbeitet parallel und unterstützt beispielsweise die Verarbeitung von Vektoren (z.B. das Skalarprodukt $s = \sum a(i) * b(i)$, das eine ähnliche Struktur wie die Faltungssumme besitzt).
- Es werden mehrere mehrere Rechenwerke parallel eingesetzt. Für die Verarbeitung eines Stereosignals könnte man z.B. das im letzten Abschnitt beschriebene Rechenwerk mit zugehörigem Registerblock duplizieren. Unter Umständen können auch in beiden Rechenwerken unterschiedliche arithmetische und logische Operationen erfolgen. Jedoch gibt es nur einen gemeinsamen Kontrollfluss, d.h. Verzweigungen und Schleifen können von einem gemeinsamen Steuerwerk aus erfolgen.
- Es werden mehrere Prozessoren bzw. Prozessorkerne parallel eingesetzt. In diesem Fall gibt es unabhängige Steuerwerke, die Verarbeitung kann völlig nebenläufig geschehen.

Alle beschriebenen Verfahren eignen sich zur Realisierung auf FPGAs.

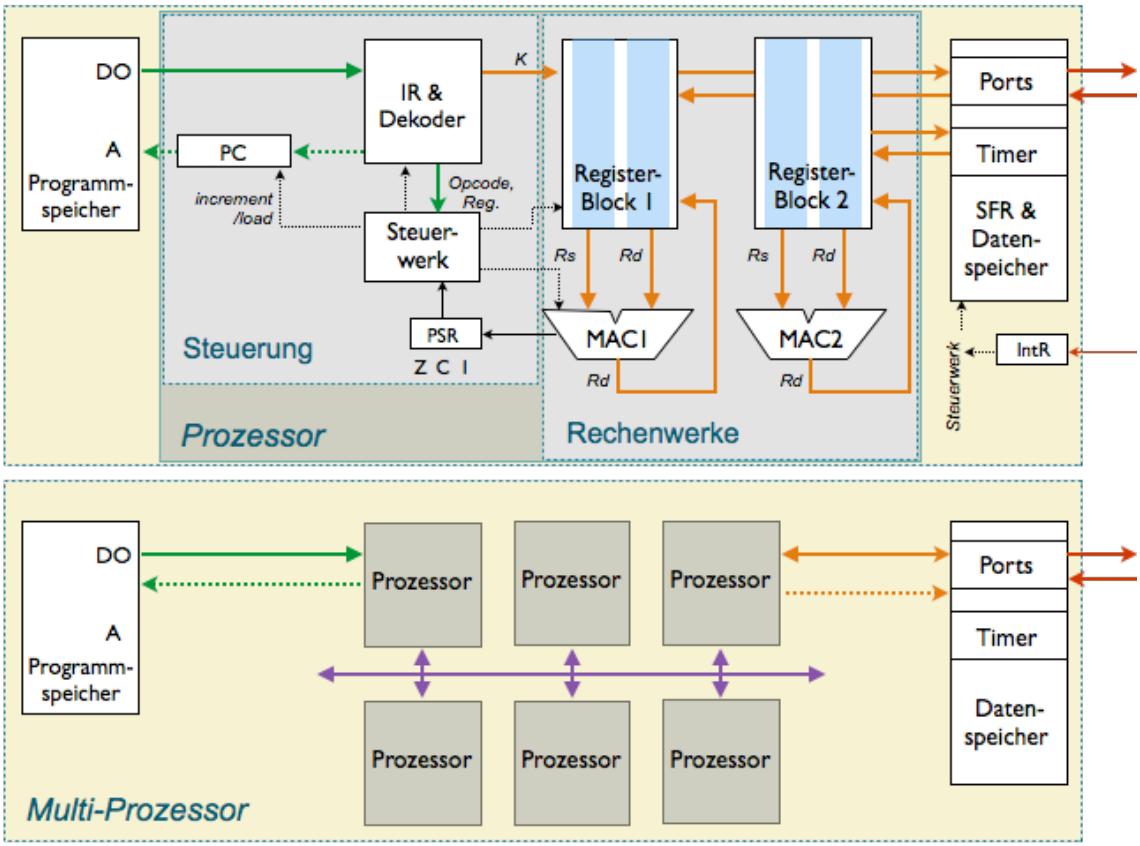


Abbildung 4.7 Arten der Parallelverarbeitung

Parallelverarbeitung im Rechenwerk

Als Rechenwerk mit interner Paralellverarbeitung soll der FIR-Filteralgorithmus als Vektor-Operation in einem Schritt umgesetzt werden. In der einfachsten Form setzt der Algorithmus die Faltungssumme unmittelbar um, wie in den folgenden Gleichungen gezeigt.

$$y[n] = \sum h_k \cdot x[n-k] \qquad (4.1)$$

Der Index k läuft hierbei über alle vorhandenen Stützstellen h_k. Für eine Impulsantwort mit insgesamt 5 Stützstellen ergibt sich also folgende Gleichung.

$$y[n] = h_0 \cdot x[n] + h_1 \cdot x[n-1] + h_2 \cdot x[n-2] + h_3 \cdot x[n-3] + h_4 \cdot x[n-4] \qquad (4.2)$$

Diese Gleichung lässt sich mit einem Rechenwerk wie in folgender Abbildung gezeigt umsetzten. Benötigt werden hierzu Addierer (Symbol „+“), Multiplizierer (Symbol „*“), sowie Register für die Speicherung vergangener Werte, wie hier die Stützstellen der Eingangswerte x(n). Das Rechenwerk arbeitet so, dass mit jedem Takt in Art eines Schieberegisters die gespeicherten Werte um eine Position nach rechts verschoben werden, und der aktuelle Eingangswert im ersten Register oben rechts gespeichert wird.

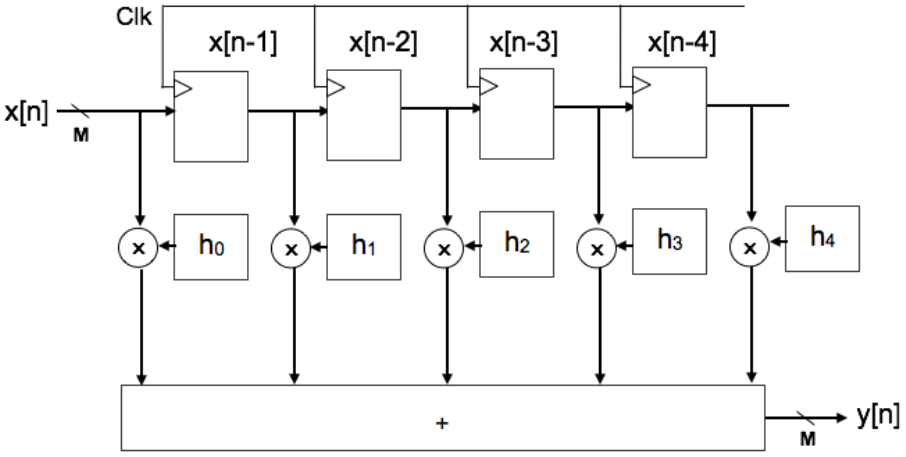


Abbildung 4.8 Paralleles Rechenwerk für FIR-Filter

Die Multiplizierer und Addierer arbeiten als Schaltnetz auf Basis der im Schieberegister gespeicherten Werte der Eingangsfunktion, sowie der intern gespeicherten Filterkoeffizienten. Nach jedem Takt steht nach Durchlauf dieses Schaltnetzes am Ausgang ein neuer Ausgangswert zur Verfügung. Die Schaltung arbeitet also parallel und bearbeitet die komplette Faltungssumme mit jedem Takt. Ein Nachteil bei der Realisierung ist das Addierwerk am Ende der Schaltung. In der Praxis müsste man hier wieder eine Kette von Addierwerken mit jeweils zwei Eingängen verwenden, deren Laufzeiten untereinander ausgeglichen, bzw. durch einen separaten Systemtakt gesteuert werden müssten.

Durch mathematische Umformung der Faltungssumme lässt sich jedoch eine effizientere Form der Realisierung finden. So lässt sich Gleichung 4.1 auf die folgende Weise schreiben, wobei nur der erste Produktterm gesondert aufgeführt wird.

$$y[n] = \sum h_k \cdot x[n-k] = h_0 \cdot x[n] + \sum h_{k+1} \cdot x[n-k-1] \qquad (4.1)$$

Der Struktur des letzen Ausdruck entspricht hierbei der ursprünglichen Gleichung, jedoch beginnend mit dem vergangenen Wert x(n-1) und dem Koeffizienten h_1. Dieses Produkt lässt sich einen Schritt vorher aus dem Wert dann aktuellen Wert x(n) und dem Koeffizienten h_1 berechnen und in einem Register aufbewahren. Zum folgenden Takt wird dieses Produkt dann als h_1 * x(n-1) zum aktuellen Produkt h_0 * x(n) addiert.

Mit den vorausgegangenen Eingangswerten verfährt man genau so bis zum letzten verfügbaren Koeffizienten h_K. Es ergibt sich dann ein Rechenwerk für den transponierter Filteralgorithmus, wie in folgender Abbildung gezeigt. Die vergangenen Produkte werden mit jedem Schritt aufsummiert, d.h. in jeder Stufe erhält man ein Zwischenergebnis Summe(i) = Summe (i+1) + Produkt(i).

Im Vergleich zu der in der vorausgegangenen Abbildung gezeigten Struktur vermeidet diese transponierte Form die Kaskade von Addierwerken. Die Kumulation der Ergebnisse geschieht hier stufenweise in den Registern für die Zwischensummen.

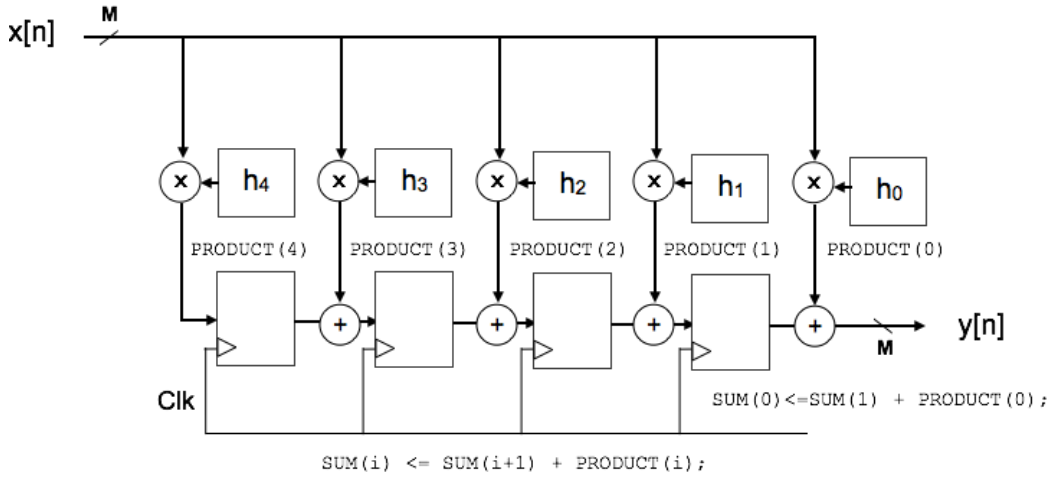


Abbildung 4.9 Paralleles Rechenwerk in transponierter Form

Implementierung in HDL

Das Rechenwerk lässt sich in HDL mit den in folgendem Text gezeigten Prozessen realisieren. Der erste Prozess ist taktabhängig und liest einen neuen Eingangswert ein. Es folgen ein Schaltnetz für die Berechnung der Produktterme. Die Berechnung startet, sobald ein neuer Eingangswert verfügbar ist. Ein weiterer Prozess berechnet mit jedem Takt einmal alle Summenterme. Der Summenterm Sum(0) wird als Ausgangswert übergeben.

```
new_sample : process (Clk) begin          --- new input value
     if rising_edge(Clk) then
          XD <= signed(XN(11 downto 0));
     end if;
end process new_sample;

multiply : process (COEFF, XD) begin      -- products
     for i in 0 to 4 loop
          PRODUCT(i) <= XD * COEFF(i);
     end loop;
end process multiply;

add_stages : process (CLK) begin          --- sums
     if rising_edge(Clk) then
          for i in 0 to 4 loop
               SUM(i) <= SUM(i+1) + PRODUCT(i);
          end loop;
     end if;
end process add_stages;

YN <= SUM(0)(23 downto 12);               --- output = sum(0)
```

Insgesamt benötigt die Schaltung für die komplette Faltungssumme tatsächlich nur einen Takt, was eine deutliche Steigerung gegenüber den bisher vorgestellten Lösungen bedeutet. Gemessen vom Zeitpunkt der Verfügbarkeit eines neuen Eingangswertes beträgt die Latenz von einem Takt, bzw. einem Abtastintervall. Diese Latenz liesse sich verringern, indem die Berechnung mit einem

Systemtakt erfolgt, der schneller ist als die Abtastrate. In diesem Fall beträgt die Latenz nur einen Systemtakt statt eines Abtastintervalls.

Nun soll das Schaltwerk als Rechenwerk des Signalprozessors realisiert werden. Bei der Übersetzung in das Umfeld des Prozessors erfolgt die Bereitstellung des neuen Eingangswertes über den Eingangsport in Kombination mit einem Unterbrechungssignal (Bit im Interrupt-Register). Die beiden Prozesse mit den Berechnungen müssen in den Kontext der Befehle und Zustandswechsel übersetzt werden. Damit die Berechnung der Produktterme vor der Berechnung der Summenterme erfolgt, wird erste in den ersten Teil eines zweitaktigen Befehls CONV verlegt (für Faltung, engl. convolution). Die Summenterme in den zweiten Takt des Befehls berechnet. Die Latenz dieser Realisierung beträgt somit zwei Systemtakte, insgesamt also eine deutliche Steigerung gegenüber der sequentiellen Berechnung.

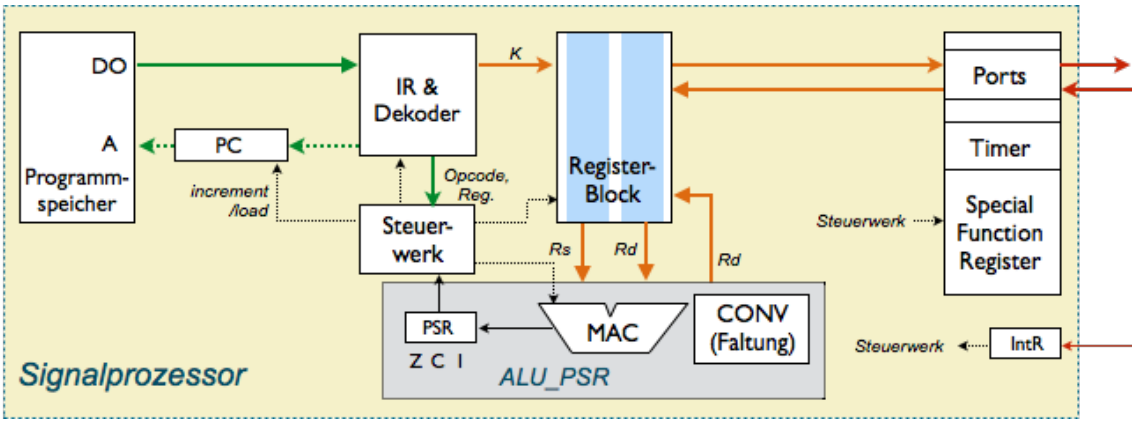


Bild 4.10 Erweiterung um ein Rechenwerk zur Parallelverarbeitung

Die Realisierung erfolgt wie in der Abbildung oben gezeigt. Der Filteralgorithmus wird als völlig eigenständiges Rechenwerk in der ALU untergebracht. Dieser mit CONV bezeichnete Block enthält auch alle benötigten Filterkoeffizienten in einem eigenen Register. Anders ist die parallele Verarbeitung nicht realisierbar. Allerdings liesse sich der Befehlsvorrat des Prozessors um Instruktionen zum Befüllen dieser Koeffizienten, sowie zum Setzen der Filterordnung (Anzahl der Koeffizienten) erweitern. Den Befehl mit Opcode und Kodierung zeigt die folgende Abbildung.

DSP Erweiterung

Befehlssatz			Beschreibung	Prozessor Status Register			
Assembler	Kürzel	Opcode		Zero (Z)	Carry (C)	Interrupt (I)	...
conv Rd	CONV	1B	Rd<= FIR(Rd, hk), Faltungssumme	x	x	-	

Befehl
logisch und arithmetisch
conv

Kodierung

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					Rd					(leer)					

Bild 4.11 Befehlserweiterung und Kodierung für das Rechenwerk

Hierbei wurde vorausgesetzt, dass der Eingangswert x(n) im Register Rd zur Verfügung steht. Dieses Register wird im Anschluss an die Verarbeitung mit dem Ergebnis der Berechnung überschrieben. Folgender Programmtext beschreibt den Algorithmus mit Hilfe der Befehlserweiterung.

```
    ; ASL (Assembler Language) für den DHBW PCT1-Mikroprozessor
```

```
; FIR Filteralgorithmus mit Parallelverarbeitung
;
             ldi R15H, $0      ; R15_High<= 0
             ldi R15L, $0      ; R15_Low <= 0
interrupt:   wait $01          ; wait for interrupt 1
             bclr $4           ; disable interrupts
             in R14, PortA     ; Rd0<= x(0)
             conv R14          ; R14<= FIR(R14, hk)
             out R14, PortB    ; PortB<= R14 (result)
             tst R15           ; Zero = true
             bset $4           ; enable interrupts
             breq interrupt    ;
```

Zähler und Schleifen mit Sprungadressen entfallen bei dieser Art der Realisierung. Es verbleibt nur der Rücksprung in den Wartezustand für den nächsten Interrupt (bzw. der Rücksprung aus der Unterbrechungsroutine in die normale Programmschleife bei einem vollständig vorhandenen Unterbrechungssystem). Folgender HDL-Text beschreibt die Erweiterung der ALU mit dem Rechenwerk.

```
...
architecture RTL of ALU_PSR is
signal Zero : std_logic_vector(15 downto 0) := (others => '0');
...

-- CONV
type COEFF_TYPE is array (0 to 15) of signed (15 downto 0);
type PRODUCT_TYPE is array (0 to 15) of signed (31 downto 0);
type SUM_TYPE is array (0 to 15) of signed (31 downto 0);
signal COEFF : COEFF_TYPE :=(            -- coefficients array
  0 => "0111111111111111",   1 => "0100000100000110", -- h0, h1
  2 => "0001011010100111",   3 => "1111010011111110", -- h2, h3
  4 => "1101110110110011",   5 => "1101101100110011", -- h4, h5
  6 => "1110001010110001",   7 => "1111000110001001", -- h6, h7
  8 => "0000001010001111",   9 => "0000110111110011", -- h8, h9
  10 => "0001000100100111", 11 => "0000111000010100", -- h10, h11
  12 => "0000011101101101", 13 => "1111111110111111", -- h12, h13
  others => "0000000000000000");
signal PRODUCT : PRODUCT_TYPE :=(        -- product array
  others => "00000000000000000000000000000000");
signal SUM : SUM_TYPE :=(                -- sum array
  others => "00000000000000000000000000000000");
--/CONV

begin
...
case OPCODE is
...
-- CONV
   when CONV => if (MAC_EN ='0') then
                  for i in 0 to 15 loop
                    PRODUCT(i) <= signed(RdVar)* COEFF(i);
```

```
                    end loop;
                  elsif (MAC_EN ='1') then
                    for i in 0 to 14 loop
                      SUM(i) <= SUM(i+1) + PRODUCT(i);
                    end loop;
                    Akku:= SUM(0);
                    RdOutVar := std_logic_vector(Akku(31 downto 16));
                  end if;
    --/CONV
       when  others => null; -- no action
    end case;
    if (RdOutVar = Zero) then ZFlag <='1'; end if;
    RdOut <= RdOutVar;
    end process;
    end RTL;
```

Die Erweiterung fällt also einigermassen überschaubar aus. Die Filterkoeffizienten sind nun also direkt im Rechenwerk untergebracht. Der Eingangswert x(n) wird vom Top-Modul in den Registerblock geschrieben und von dort aus als Rd ausgelesen. Das Ergebnis wird als Ausgangswert RdOut wiederum dem Registerblock übergeben. Am Registerblock sind keine Änderungen erforderlich. Folgende HDL-Beschreibung zeigt die Erweiterungen im Dekoder (Instruktionsregister).

```
    --- Instruction_Register (IR) and Decoder for DHBW PCT1 (VHDL)
    ...
    when "11011" => OPCODE <= CONV; A1 <= unsigned(DIN(10 downto 6));
    ...
```

Die Erweiterung besteht hier gerade aus einer Teile Text. Das Quell- und Zielregister der Operation wird mit der Adresse des Operanden (Rd) angegeben. Selbstverständlich muss der Opcode CONV auch in der Paketdefinition aufgeführt werden. Die Erweiterung des Top-Moduls mit dem Zustandsautomaten beschreibt folgender HDL-Text.

```
    --- FSM and Top Module for the DHBW PCT1_SP controller (VHDL)
        ...
        when Z1 => -- one clock cycle per operation

           case T_OPCODE is
           ...
           -- CONV
             when CONV =>  -- step 1: products
                       T_MAC_EN<='0'; -- indicate step 1 to ALU
                       T_Pipe_EN<='0'; T_PC_EN<='0';
                       T_AOUT2<=T_A1; T_SEL_Rd_SFR<="11";
                       next_state<= Z2;
           --/CONV
           ...

         when Z2 => -- second clock cycle per operation

           case T_OPCODE is
```

```
        ...
        --CONV
          when CONV =>  -- step 2: sums
                    T_MAC_EN<='1';
                    T_PC_EN<='1'; T_Pipe_EN<='1'; T_REG_EN<='1';
                    T_AIN<=T_A1; T_SEL_K_Rd_SFR<="10";
                    next_state<= Z1;
             --/CONV

    ...
```

Für die Erweiterung wurde das bereits vorher verwendete Signal MAC_EN verwendet, um die Operation über zwei Taktzyklen zu verteilen. Da der Eingangswert sowie der Ausgangswert über die vorhandenen Kanäle und Adressen zum Registerblock kommuniziert werden, müssen nur diese nur korrekt ausgewählt werden. Die Vorgehensweise entspricht der bei allen anderen arithmetischen und logischen Operationen. Zum Start des Programms ist schliesslich der Maschinencode im Programmspeicher abzulegen, den folgender HDL-Text zeigt.

```
--- Program_Memory for DHBW PCT1 controller (VHDL)

use work.PCT1_Pack_SP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity P_ROM is
       port ( Clk, Pipe_EN : in std_logic; -- Clock
              PADDR : in  PA_TYPE; PDOUT : out D_TYPE);
end P_ROM;

architecture RTL of P_ROM is

 -- array of 2**8 samples, each 16 bits wide (reduced for tests)
 type ROM_array is array(0 to 32) of D_TYPE;

       -- instantiate memory object with sample program
       signal pmemory : ROM_array   := (
         0 => "0010011100000000", -- LDI 0 to R15H
         1 => "0010011000000000", -- LDI 0 to R15L
         2 => "1011100000000001", -- WAITI
         3 => "1100101110000010", -- INP   to R14 PortA
         4 => "1101101110000000", -- CONV R14
         5 => "1101001110000100", -- OUTP R14 to PortB
         6 => "0101101111011110", -- ANDR (TST) R15
         7 => "1000010000100111", -- BREQ (-7)
         others => "0000000000000000");

 begin
       -- read process
       process (Clk, Pipe_EN) begin
         if (Pipe_EN = '1') then
```

```
            if (rising_edge(Clk)) then
          PDOUT <= pmemory(to_integer(unsigned(PADDR)));
        end if;
      end if;
    end process;

end RTL;
```

Der Programmcode umfasst gerade acht Zeilen (wobei auf das Setzen und Rücksetzen der Interrupt-Flags verzichtet wurde, da kein vollständiges Unterbrechungssystem vorhanden ist.) Folgende Abbildung zeigt einen Testlaufs des Programms.

Nach dem Reset läuft das Programm im Zustand Z1 hoch. Der Programmzähler beginnt mit dem Laden der Befehle, bis er bei Schritt 4 angekommen ist. Inzwischen sind die beiden Ladebefehle ausgeführt worden (Schritt 0 und Schritt 1). Der Befehl WAIT-Interrupt wurde in Schritt 2 geladen. Wegen der Fließbandverarbeitung (Pipeline) steht der Programmzähler an dieser Stelle bereits bei Schritt 4. Da das Signal „Interrupt“ inzwischen angekommen ist werden die folgenden Instruktionen INP und CONV, die bereits geladen sind, ausgeführt.

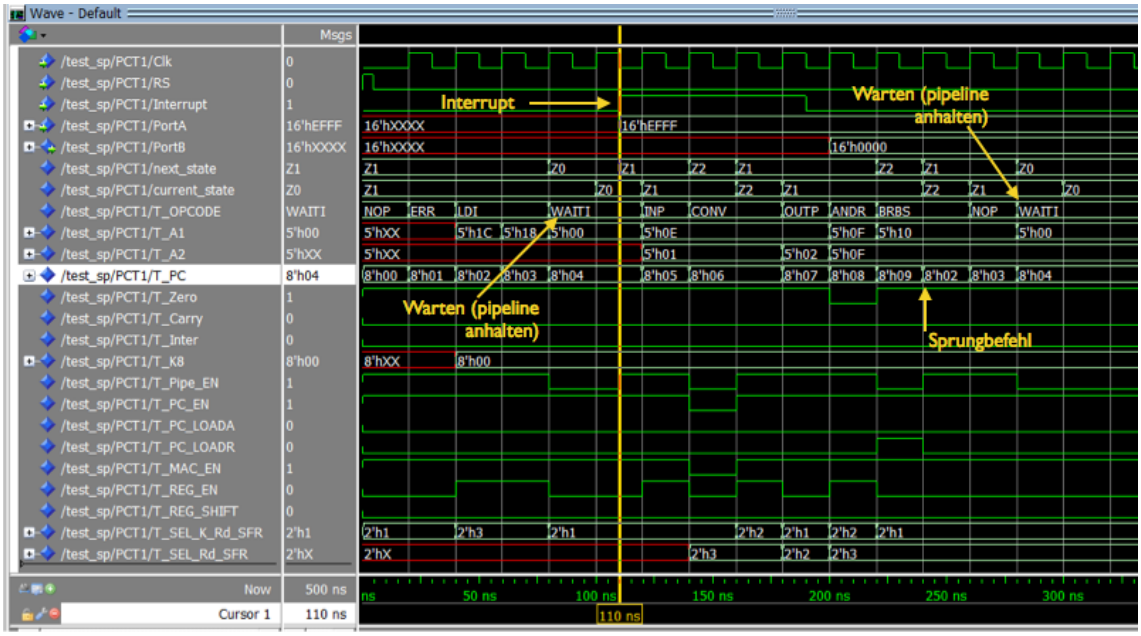


Bild 4.12 Ablauf des Filterprogramms

Anschliessend muss der Programmzähler weitere Instruktionen nachladen und zählt über das Programmende bei Schritt 7 hinaus (im Programmspeicher finden sich hier Nullen, die als NOP interpretiert werden). Wenn der Programmzähler bei Schritt 9 angekommen ist, befindet sich der Sprungbefehl BRBS in der Ausführung. Der Programmzähler wird programmgemäß um 7 Schritte dekrementiert, womit er wiederum auf die Instruktion WAIT-Interrupt zeigt. Wegen der Pipeline werden die nächsten beiden Befehle (INP und CONV) bereits geladen.

Die Ausführung des Befehls CONV mit der Faltungssumme benötigt zwei Takte. Es folgt die Abfrage der Bedingung tst R15, die über die Instruktion ANDR mit der Adresse von R15 realisiert ist. Dass der Sprungbefehl ausgeführt wird, erkennt man am auf den Wert 2 (auf den Befehl WAITI) zurückgesetzten Programmzähler. In der Pipeline befindet sich jedoch noch die Instruktion mit der

Adresse 08, die im Programmspeicher auf die Instruktion NOP trifft und daher hier nicht stört. Der Prozessor führt anschließend die Instruktion WAITI an der Adresse 02 aus, wodurch die Pipeline bis zur nächsten Unterbrechung angehalten wird.

Bei Sprungbefehlen findet sich im Falle der Ausführung immer ein nicht mehr gültiger Befehl in der Pipeline. Mit dieser Situation lässt sich leben, wenn man im Programmtext grundsätzlich vor jedem Sprungbefehl eine Instruktion NOP als Füllung für die Pipeline unterbringt. Für Instruktionen wie WAIT, bzw. für Instruktionen, die zwei Takte benötigen, ist eine solche Maßnahme nicht erforderlich, da die Bearbeitung ja nur einen Takt angehalten wurde und der bereits geladene folgende Befehl den Ablauf wieder fortsetzt.

Seminararbeit S7: Implementieren Sie den Faltungsalgorithmus als Rechenwerk auf Ihrem Mikrocontroller. Verwenden Sie hierzu die einfache Akku-Architektur ohne Register. Hinweis: Die Rechenoperation kann in einem einzigen Schritt ausgeführt werden, wenn die Filterkoeffizienten vorab als Konstanten ins Rechenwerk geladen wurden. Testen Sie Ihre Implementierung mit einem Beispiel.

5.3. Gleitkomma-Arithmetik

Bei Berechnungen im Festkommaformat ist der Wertebereich immer sehr eingeschränkt. Für Festkommazahlen im Dezimalsystem mit 10 Stellen beträgt der Bereich von der kleinsten bis zur größten Zahl 10^{10}. Bei Festkommazahlen mit Vorzeichen wäre der Bereich auf +/- 10^{9} beschränkt. Ausserdem ist die Darstellung kleiner Zahlen (bzw. von Zahlen mit kleinem Betrag) wegen der beschränkten Anzahl verfügbarer Stellen sehr ungenau (beispielsweise 175 003 011 im Vergleich zu 365). Für Berechnungen mit physikalischen Hintergrund (Multiplikationen, Wertebereich von Mikro bis Giga) ist diese Darstellung mit sehr vielen Einschränkungen verbunden.

Besser geeignet ist hierfür die Darstellung der Zahlen im Gleitkomma-Format. Hier liesse sich 175 003 011 darstellen als 1,75003011 * 10^{8} und ebenso die kleine Zahl ohne Verluste an Genauigkeit z.B. als 3,65435472 * 10^{2}. Diese Darstellung als Kombination einer Festkommazahl als Mantisse mit einem Exponenten wird als Gleitkomma-Format bezeichnet. Diese Darstellung lässt sich ebenso im binären Format verwenden, wobei als Basis statt der Zahl 10 eben die Zahl 2 verwendet wird.

Für eine Darstellung mit insgesamt 32 Bit soll das in der Abbildung gezeigte Schema verwendet werden. Hierbei wird ein Bit für das Vorzeichen verwendet, 8 Bits für den Exponenten, sowie die verbleibenden 23 Bits für die Mantisse.

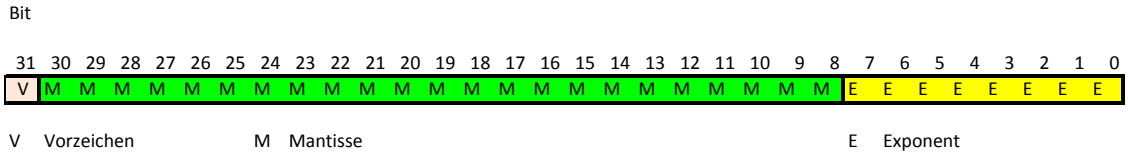


Bild 4.13 Gleitkomma-Format

Die Rechenoperationen Addition, Subtraktion, Multiplikation und Division folgen den im Zehnersystem geläufigen Regeln. Für Additionen und Subtraktionen werden die Exponenten angeglichen, wodurch sich für die Mantissen die gleichen Verhältnisse ergeben wie im Festkommaformat. Bei der Multiplikation werden die Mantissen multipliziert und die Exponenten addiert. Hier zeigt das Gleitkommaformat seine Stärke gegenüber dem Festkommaformat.

Übung 5.1: Welchen Wertebereich deckt die Gleitkomma-Darstellung im genannten Format ab?

Übung 5.2: Für die Darstellung des Exponenten wird vereinbart, dass der Exponent im Bereich 0 bis 255 eine Abweichung (engl. Bias) von -127 besitzt. Auf diese Weise lassen sich ohne Einführung eines Vorzeichenbits Exponenten im Bereich von -127 bis +128 darstellen. Statt der Bezeichnung Exponent wird im Gleitkomma-Format die Bezeichnung Charakteristik verwendet im Sinne von Exponent = Charakteristik -127. Welchen Wertebereich deckt das so definierte Format ab? Welchen Einfluss hat dieses Format auf die genannten Rechenoperationen? Wie werden die Rechenoperationen ausgeführt (bitte Beispiele ausführen)?

Übung 5.3: Entwerfen Sie ein Rechenwerk für Gleitkomma-Zahlen mit den Befehlen Addition, Subtraktion, Multiplikation und Division. Testen Sie Ihren Schaltungsentwurf im Simulator.

Seminararbeit L8: Implementieren Sie einen Prozessor mit Gleitkomma-Rechenwerk auf dem FPGA. Testen Sie die Funktion der Schaltung. Untersuchen Sie das Ergebnis der Synthese (zu welchen Schaltungen wurden die Rechenwerke synthetisiert). Welcher Bedarf an Logikbausteinen hat die Realisierung im Vergleich zu einem Festkomma-Rechenwerk?

6. Verschlüsselung

Verschlüsselte Daten bzw. verschlüsselte Verbindungen erlauben die vertrauliche Kommunikation über öffentliche Netze. FPGAs lassen sich hierbei direkt auf den Schnittstellen der Kommunikation nutzen. Auch für Angriffe auf verschlüsselte Daten bieten FPGAs wegen der Möglichkeit zur Parallelisierung die benötigte Rechenleistung.

6.1. Die Pfadfindermethode

Eine einfache Methode der Verschlüsselung besteht darin, die nach Zeichen eines Textes nach einem vereinbarten Muster gegeneinander auszutauschen, z.B. „E“ gegen „U“, „U“ gegen „H“ etc. Das Muster für den Austausch der Zeichen entspricht hierbei dem Schlüssel.

Klartext	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Schlüssel	Q	Z	S	A	U	C	B	I	Y	D	V	R	.		E	F

Klartext	Q	R	S	T	U	V	W	X	Y	Z		.
Schlüssel	G	T	W	X	H	M	K	J	O	N	L	P

Bild 5.1 Verschlüsselung nach der Pfadfindermethode

Durch Anwendung dieser Tabelle wird aus einem Klartext ein verschlüsselter Text. Aus dem Satzteil „Als man dies im Dorf erfuhr ...“ beispielsweise wird „Qrwl.q ayuwly.laetclutchitl ...“. Die Buchstaben sind verwürfelt worden, allerdings nicht zufällig, sondern nach dem im Schlüssel angegebenen Schema. Wenn man den Schlüssel kennt, d.h. im Besitz der in der Abbildung gezeigten Tabelle ist, kann man den Text wieder entschlüsseln.

Übung 6.1: Wie lässt sich sich ein verschlüsselter Text entziffern, wenn man nicht im Besitz des Schlüssels ist? Nennen Sie unterschiedliche Ansätze. Welches Ziel verfolgt diese Art der Entschlüsselung? Wodurch werden solche Angriffe auf die Verschlüsselung begünstigt?

Folgende Abbildung zeigt das grundsätzliche Szenario bei der Verschlüsselung, Kommunikation und Entschlüsselung. Auf der Seite A (links) wird ein Dokument verschlüsselt. Das Verfahren (der Verschlüsselungsalgorithmus) ist hierbei als Prozess E bezeichnet (für engl. Encryption). Über die Funktionsweise des Verfahrens und den benötigten Schlüssel sind hierbei keine speziellen Vorgaben enthalten, die Beschreibung des Verfahrens ist allgemein bzw. abstrakt. Das verschlüsselte Dokument kann nun als Nachricht übermittelt werden.

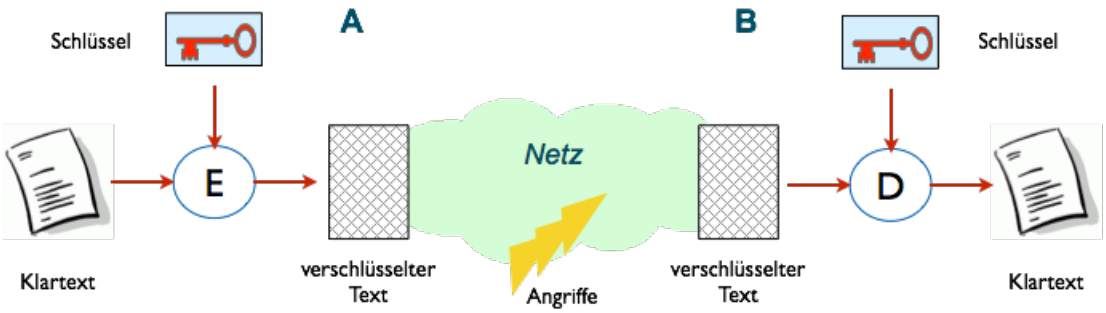


Bild 5.2 Verschlüsselte Nachrichtenübermittlung

Über den Weg zum Empfänger, beispielsweise über ein Kommunikationsnetz, haben die Korrespondenten an den Stellen A und B keine Kontrolle. Man geht davon aus, dass das verschlüsselte Dokument bzw. die verschlüsselte Nachricht abgefangen und analysiert werden kann. Der Empfänger am Anschluss B entschlüsselt das Dokument wieder mit Hilfe des ihm bekannten Schlüssels. Auch dieser Prozess D (engl. für Decryption) ist allgemeingültig dargestellt. Schlüssel und Verschlüsselungsverfahren können unterschiedlich aufwändig und raffiniert sein. Der Ablauf ist grundsätzlich gleich.

Kodierung von Texten

Auch die Pfadfindermethode ist mit maschineller Unterstützung auf binär kodierten Texten anwendbar und wird durch Einsatz eines digitalen Schaltwerks deutlich vereinfacht. Folgende Abbildung zeigt ein einfaches Schema zur Kodierung von Zeichen inklusive Leerzeichen und Punkt.

Kodierung (hex)	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Q	R	S	T	U	V	W	X	Y	Z		.				

Bild 5.3 Kodierung von Zeichen

Nach diesem Schema wäre z.b. das Zeichen A als hexadezimal x“00“ kodiert, das Zeichen Q als x“10“. Es wird vorausgesetzt, dass das Kodierungsschema bekannt und gebräuchlich ist (in der Praxis beispielsweise als ASCII-Kodierung bzw. Unicode). Das in der Tabelle gezeigte Kodierungsschema unterscheidet keine Gross- und Kleinschreibung, kennt keine Zahlen und soll nur zur Illustration einiger Beispiele dienen. Für die Kodierung der insgesamt 27 Zeichen werden 5 Bits Wortbreite benötigt.

Verschlüsselungsmaschine für Pfadfinder

Texte lassen sich nun verschlüsseln, in dem man Zeichen nach einem vereinbarten Schlüssel vertauscht. Diesen Schlüssel kann man z.B. als Tabelle vereinbaren, die mit den kodierten Zeichen

Adressiert wird. Als Ausgabe erhält man ein Zeichen, das nach dem vorgegebenen Schlüssel transformiert wurde (siehe Abbildung 5.1). Folgender HDL-Text beschreibt die Verschlüsselung.

```
--- Encryption (VHDL)

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ENCRYPT is
  port (Clk : in std_logic);
end ENCRYPT;

architecture RTL of ENCRYPT is

  component ENCR_ROM is
     port ( CADDR : in  unsigned(5 downto 0);
            CDOUT : out unsigned(4 downto 0));
  end component;

  component D_RAM is
            port ( WADDR : in  unsigned (5 downto 0);
                   DATIN : in  unsigned (4 downto 0));
  end component;

  signal ADDR : unsigned(5 downto 0) := "000000";
  signal DReg, DOut : unsigned(4 downto 0);

  -- Look-up table of 64 samples, each 5 bits wide
  type LUT is array(0 to 63) of unsigned(4 downto 0);
       -- instantiate LUT with key
       signal key : LUT  := (
       0 => '1'&x"0", 1 => '1'&x"9", 2 => '1'&x"2", 3 => '0'&x"0",
       4 => '1'&x"4", 5 => '0'&x"2", 6 => '0'&x"1", 7 => '0'&x"8",
       8 => '1'&x"8", 9 => '0'&x"3", 10 => '1'&x"5", 11 => '1'&x"1",
       12 => '1'&x"B", 13 => '1'&x"A", 14 => '0'&x"4", 15 => '0'&x"5",
       16 => '0'&x"6", 17 => '1'&x"3", 18 => '1'&x"6", 19 => '1'&x"7",
       20 => '0'&x"7", 21 => '0'&x"C", 22 => '0'&x"A", 23 => '0'&x"9",
       24 => '0'&x"E", 25 => '0'&x"D", 26 => '0'&x"B", 27 => '0'&x"F",
       others => "00000");

  begin
  -- instantiate and connect memories
  Source: ENCR_ROM port map (CADDR => ADDR, CDOUT => DReg);
  Destination : D_RAM port map (WADDR => ADDR, DATIN => DOut);

  -- read, encrypt and write data
  encrypt : process (Clk) begin
     for i in 0 to 63 loop
       if rising_edge(Clk) then
         DOut <= key(to_integer(DReg));
```

```
            ADDR <= ADDR + 1;
          end if;
        end loop;
    end process encrypt;
end RTL;
```

Der Schlüssel wurde hierzu in der Tabelle Key hinterlegt. Der Quelltext wurde im Speicher ENCR-ROM untergebracht. Der verschlüsselte Text wird in den Speicher D_RAM geschrieben. Folgende HDL-Texte beschreibt die beiden Speicher.

```
--- Memory for Encryption(VHDL)

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ENCR_ROM is
       port (CADDR : in  unsigned(5 downto 0);
             CDOUT : out unsigned(4 downto 0));
end ENCR_ROM;

architecture RTL of ENCR_ROM is

 -- array of 32 samples, each 5 bits wide
 type ROM_array is array(0 to 63) of unsigned (4 downto 0);

 -- instantiate memory object with sample text
 signal pmemory : ROM_array   := (
 -- Als man dies im Dorf erfuhr, war von Trauer keine Spur.
 0 => '0'&x"0", 2 => '0'&x"B", 3 => '1'&x"C", 4 => '1'&x"A",
 5 => '0'&x"C", 6 => '0'&x"0", 7 => '0'&x"D", 8 => '1'&x"A",
 9 => '0'&x"3", 10 => '0'&x"8", 11 => '0'&x"4", 12 => '1'&x"2",
 13 => '1'&x"A", 14 => '0'&x"8", 15 => '0'&x"C", 16 => '1'&x"A",
 17 => '0'&x"3", 18 => '0'&x"E", 19 => '1'&x"1", 20 => '0'&x"5",
 21 => '0'&x"5", 22 => '1'&x"A", 23 => '0'&x"4", 24 => '1'&x"1",
 25 => '0'&x"5", 26 => '1'&x"4", 27 => '0'&x"7", 28 => '1'&x"1",
 29 => '1'&x"A", 30 => '1'&x"6", 31 => '0'&x"0", 32 => '1'&x"1",
 33 => '1'&x"A", 34 => '1'&x"5", 35 => '0'&x"E", 36 => '0'&x"D",
 37 => '1'&x"A", 38 => '1'&x"3", 39 => '1'&x"1", 40 => '0'&x"0",
 41 => '1'&x"4", 42 => '0'&x"4", 43 => '1'&x"1", 44 => '1'&x"A",
 45 => '0'&x"A", 46 => '0'&x"4", 47 => '0'&x"8", 48 => '0'&x"D",
 49 => '0'&x"4", 50 => '1'&x"A", 51 => '1'&x"2", 52 => '0'&x"F",
 53 => '1'&x"4", 54 => '1'&x"1", 55 => '1'&x"A", 56 => '1'&x"A",
 others => "00000");

 begin
  process (CADDR) begin
       CDOUT <= pmemory(to_integer(CADDR));
  end process;
end RTL;
```

```
--- Data Memory for Encryption (VHDL)

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity D_RAM is
       port ( WADDR : in  unsigned (5 downto 0);
              DATIN : in  unsigned (4 downto 0));
end D_RAM;

architecture RTL of D_RAM is

 -- array of 32 samples, each 5 bits wide
 type RAM_array is array(0 to 63) of unsigned(4 downto 0);

-- instantiate memory object of X_RAM
signal dmemory : RAM_array  := (others => "00000");

 begin
  process (WADDR) begin
      dmemory(to_integer(WADDR)) <= DATIN;
  end process;
end RTL;
```

Nach einem Testlauf findet man den verschlüsselten Text im Datenspeicher. Folgende Abbildung zeigt die Schlüsseltabelle (key), den Klartext (source), sowie den transformierten Text (destination).

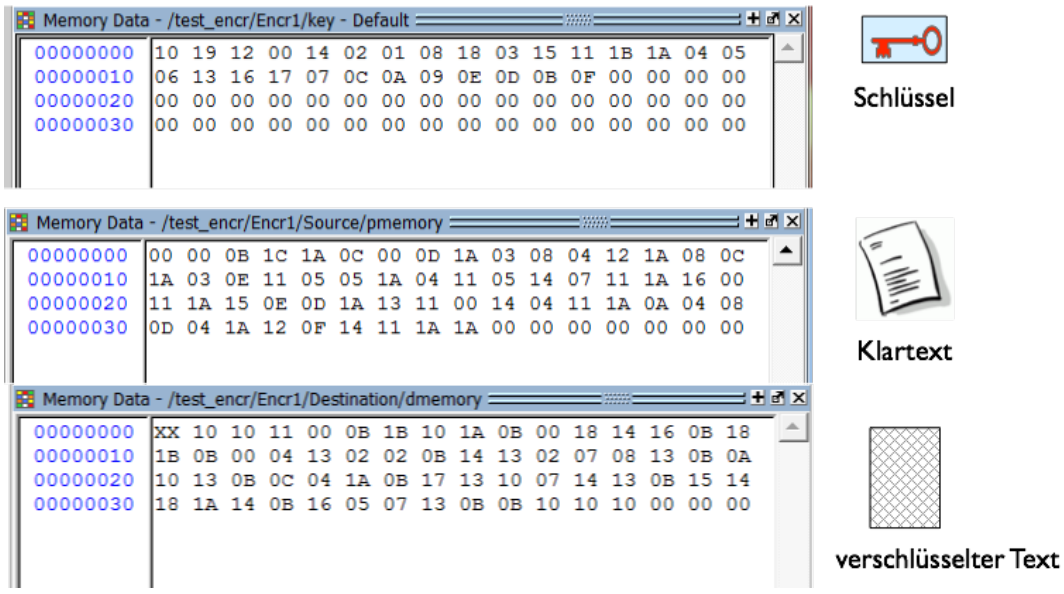


Bild 5.4 Schlüssel, Klartext und verschlüsselter Text

Um den Text wieder zu entschlüsseln, muss man die durch die Tabelle gegebene Transformation wieder umkehren. Ausserhalb des Pfadfinderlebens ist diese Methode nur sehr eingeschränkt nutzbar, da der Schlüssel durch Textanalyse leicht erraten werden kann. Mehr Sicherheit erreicht man durch Verwendung mehrerer, unterschiedlicher Tabellen zur Verschlüsselung. Die Vereinbarung, wann welche Tabelle verwendet werden soll, ist dann ebenfalls Bestandteil des Schlüssels (z.B. 5 Zeichen mit Tabelle 1, dann 7 Zeichen mit Tabelle 2 etc).

Übung 6.2: Entwerfen Sie eine Schaltung als HDL-Beschreibung, die einen verschlüsselten Text mit Hilfe der Tabelle wieder entschlüsseln kann. Testen Sie die Schaltung auf dem Simulator.

Seminararbeit S9: Entwerfen Sie eine Schaltung zum Verschlüsseln und Entschlüsseln von Texten. Überprüfen Sie Ihre Schaltung mit Hilfe eines Beispiels.

6.2. Stromziffern

Eine andere recht einfache Methode zur Verschlüsselung beruht auf der Exklusiv-Oder-Verknüpfung eines Textzeichens des Klartextes mit einem weiteren Textzeichen als Schlüssel. Nimmt man in der oben angenommenen Kodierung beispielsweise das Zeichen B = „00001“ und verknüpft es mit dem Zeichen C = „00010“ so erhält man B XOR C = 00001 XOR 00010 = 00011. Dieses verschlüsselte Zeichen überträgt man vom Absender A an den Empfänger.

Wenn der Empfänger den Schlüssel kennt (hier also B), so kann der das Zeichen durch erneute Anwendung der Exklusiv-Oder-Verknüpfung wieder entschlüsseln. Aus dem übertragenes Zeichen „00011“ und dem Schlüssel C = „00010“ erhält man durch Anwendung von 00011 XOR 00010 = 00001 = B. Für einen Text mit N-Zeichen benötigt man mit diesem Verfahren einen weiteren Text mit N-Zeichen als Schlüssel. Die Länge des Schlüssels ist somit gleich der Länge des Klartextes. Der Schlüssel in Länge des Klartexts wird auch als Stromziffer (engl. Stream Cipher) bezeichnet.

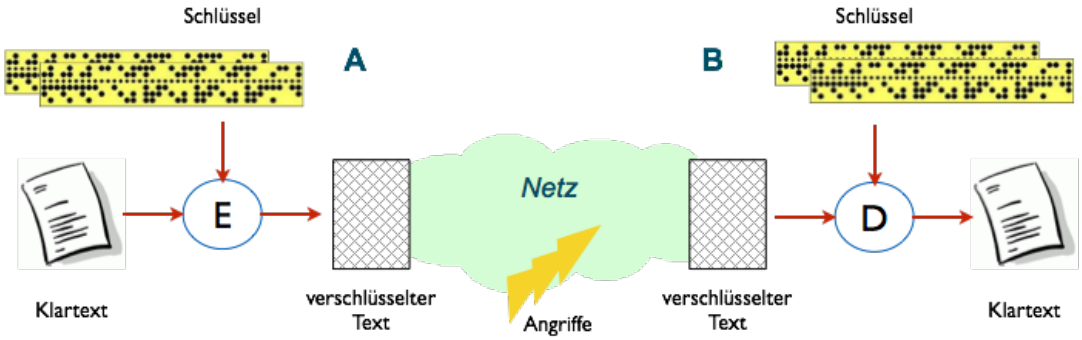


Bild 5.5 Stromziffern als Schlüssel

Im Unterschied zur Pfadfindermethode ist die Verwendung von Stromziffern sehr sicher. Grund hierfür ist die Tatsache, dass man von der Verschlüsselung eines Zeichens nicht auf die Verschlüsselung eines anderen Zeichens schliessen kann. Das gleiche Zeichen erhält jedes Mal einen zufällig ausgewählten neuen Schlüssel. Eine Vorhersage ist somit nicht möglich, unabhängig von der Länge des bekannten Klartextes. Die einzige Einschränkung des Verfahrens besteht darin, dass man die gleiche Stromziffer nicht mehrmals verwenden sollte. Als Stromziffer kommt jeder beim Sender

und Empfänger bekannte Text in Frage, beispielsweise ein Buch, bzw. sichererer ein für diesen Zweck vorhandener Zufallstext (Textrolle), von dem bereits verwendete Abschnitte vernichtet werden.

Übung 6.3: Erstellen Sie eine Schaltung, die das Textmuster aus der letzten Übung mit Hilfe einer zufällig gewählten Stromziffer kodiert. Testen Sie die Schaltung, in dem Sie den verschlüsselten Text erneut dekodieren.

Das Verfahren lässt sich auch unmittelbar bitweise anwenden. In diesem Fall benötigt man eine zufällige Bitfolge als Stromziffer. Auch hier kann ein Angreifer von einem erfolgreich dekodierten Bit nicht auf ein anderes Bit schliessen. Die Wahrscheinlichkeit, ein einzelnes Bit zu erraten, bleibt bei 50%, unabhängig von der Länge eines bereits bekannten Klartextes und seinem Schlüssel.

Pseudo-Zufallszahlen

Für ein wirklich sicheres Verfahren muss eine identische Folge von Zufallszahlen beim Sender und beim Empfänger bekannt sein. Diese Folge muss auch einem sicheren Weg miteinander ausgetauscht werden. Eine vereinfachte Methode verwendet auf beiden Seiten einen baugleichen Zufallsgenerator. Beide Generatoren werden mit dem gleichen Startwert initialisiert und produzieren dann eine identische Folge von Zufallszahlen. Hierbei spielt nun der Startwert die Rolle des geheimen Schlüssels, der auf sicherem Wege zwischen beiden Seiten vereinbart werden muss. Die folgende Abbildung zeigt das Funktionsprinzip.

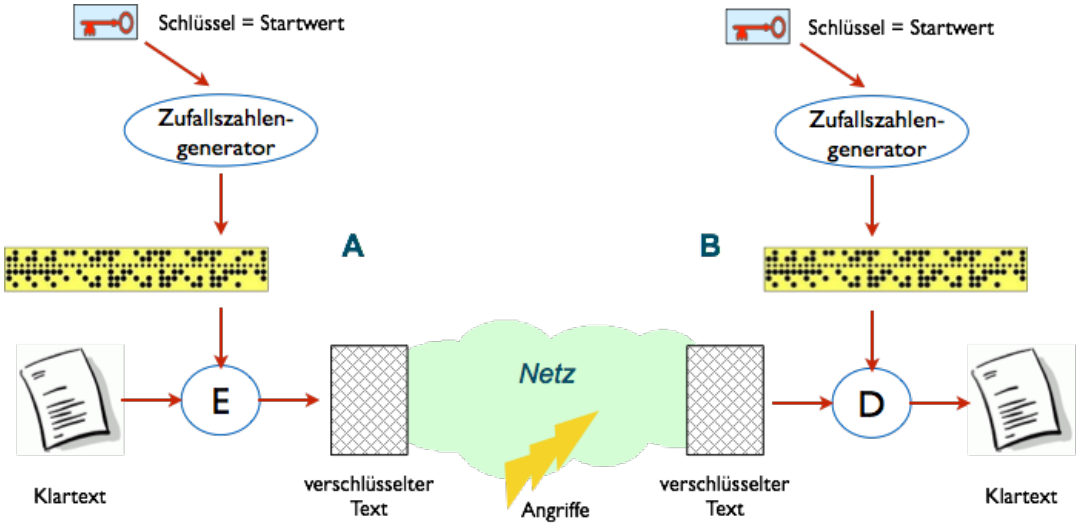


Bild 5.6 Stromziffern mit Zufallszahlen-Generator

Zufallsgeneratoren lassen sich mit rückgekoppelten Schieberegistern realisieren. Folgende Abbildung zeigt das Prinzip.

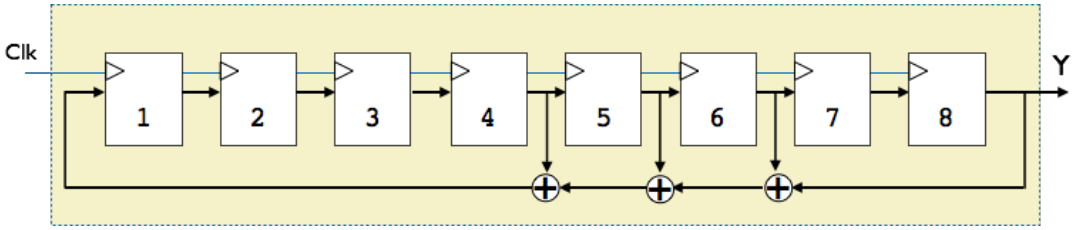


Bild 5.7 Zufallszahlen-Generator

Der Generator besteht aus einem 8-Bit Schieberegister mit Rückkopplungen an Bit 8 (bei aufsteigender Num-merierung Richtung Ausgang), Bit 6, Bit 5 und Bit 4. Die rückgekoppelten Signale werden über XOR miteinander Verknüpft (als Symbol wurde das Zeichen ⊕ verwendet). Folgender HDL-Text beschreibt diese Schaltung.

```
--- Pseudo Noise 8 bits (VHDL)
library ieee;
use ieee.std_logic_1164.all;

entity Pseudo_Noise_8 is
      port ( Clk, RS : in std_logic;  -- clock, reset (active low)
             Q : out std_logic_vector (8 downto 1)); -- 8 bits out
end Pseudo_Noise_8;

architecture RTL of Pseudo_Noise_8 is
      signal Qreg: std_logic_vector (8 downto 1);   -- internal signal
      begin
   process (Clk, RS) begin
     if (RS = '0') then
        Qreg <= "01001001";
     elsif rising_edge(Clk) then
       for i in 8 downto 2 loop
          Qreg(i) <= Qreg(i-1);
       end loop;
       Qreg(1)<=Qreg(8) XOR Qreg(6) XOR Qreg(5) XOR Qreg(4);
     end if;
   end process;

   Q <= Qreg;
end RTL;
```

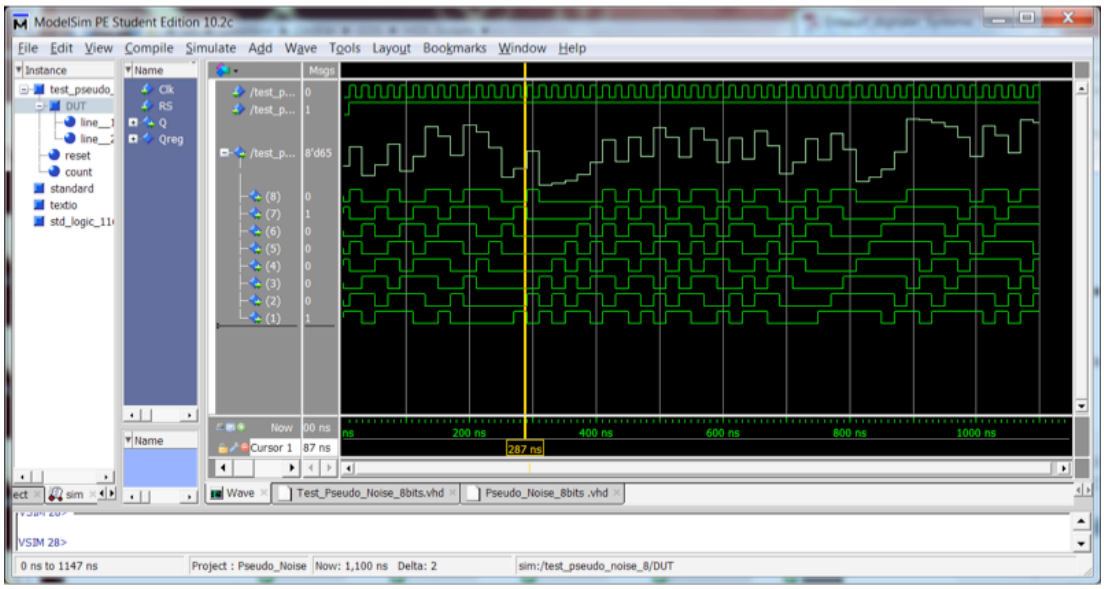


Bild 5.8 Generator für Zufallszahlen

Der Testlauf zeigt die zufällig generierte Bitfolge zusammen mit dem als Byte interpretierten Inhalt des Schieberegister.

Übung 6.4: Erweitern Sie die Schaltung so, dass der Initialwert des Generators als Schlüssel übergeben werden kann.

Übung 6.5: Verwenden Sie den Generator zur Verschlüsselung einer Zeichenfolge bzw. Bitfolge. Entschlüsseln Sie die Zeichenfolge bzw. Bitfolge mit Hilfe des verwendeten Schlüssels.

Übung 6.6: Erstellen Sie eine Schaltung für einen 16-Bit Zufallsgenerator. Verwenden Sie hierzu ein 16-Bit Schieberegister mit Rückkopplungen an den Bits 13, 12, 10 und 6. Gezählt wurde hierbei in aufsteigender Reihenfolge vom Eingang (Bit 1) bis zum Ausgang (Bit 16). Vergleichen Sie beide Generatoren.

Seminararbeit S10: Entwerfen Sie eine Schaltung zum Verschlüsseln und Entschlüsseln von Texten mit Hilfe von Stromziffern. Überprüfen Sie Ihre Schaltung mit Hilfe eines Beispiels.

6.3. Prüfsummen

Prüfsummen werden verwendet, um Fehler in der Datenübertragung bzw. bei der Speicherung von Daten festzustellen. Sie dienen zur Sicherung der Unversehrtheit bzw. Integrität der Daten. Im Zusammenhang mit möglichen Manipulationen der Daten auf dem Weg zwischen Sender und Empfänger sind spezielle Prüfsummen gebräuchlich, die sogenannten Hash-Funktionen. Die Hash-Funktion dient hierbei als Stellvertreter der übermittelten Daten. Wurde auf dem Weg ein Bit des Dokumentes verändert, stimmt die beim Empfänger aus dem Dokument berechnete Prüfsumme nicht mehr mit der übermittelten Prüfsumme überein.

Das Funktionsprinzip soll mit Hilfe der Divisions-Rest-Methode gezeigt werden. Die Hash-Funktion h(k) des Wertes k wird hierbei nach folgender Vorschrift ermittelt.

$$h(k) = k \bmod m \tag{5.1}$$

Die Funktion Modulo m bezeichnet hierbei den Rest bei einer Division durch m. Mit m=127 ist der durch die Hash-Funktion berechnete Wert also stets kleiner als 127, unabhängig von der Größe von k. Für Zeichenketten z(i) bzw. in Gruppen zusammengefasster Bits einer Bitfolge kann man den Algorithmus in rekursiver Weise anwenden, wie folgender Pseudocode zeigt:

```
for i in 0 to N loop
        h = (h*32 + z(i)) mod m;
end loop;
```

Hierbei wurde angenommen, dass jedes Zeichen z(i) eine Wortbreite von 5 Bits besitzt. Die Multiplikation mit Zweierpotenzen kann hierbei als Schiebeoperation ausgeführt werden (hier um 5 Bits nach links). Die Zwischenergebnisse erreichen bei diesem Beispiel maximal den Wert (m-1) *32 + 31, d.h. näherungsweise m*32. Um Hash-Werte der Größe 127 zu erzeugen (entsprechend 7 Bits), wird m = 127 gewählt. Folgender HDL-Text zeigt die Berechnung der Hash-Funktion für die eingangs erwähnte Zeichenfolge.

```
--- Hash (Divisions-Rest-Methode) (VHDL)

library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ENCRYPT is
  port (Clk : in std_logic);
end ENCRYPT;

architecture RTL of ENCRYPT is

  component ENCR_ROM is
     port ( CADDR : in  unsigned(5 downto 0);
                 CDOUT : out unsigned(4 downto 0));
  end component;

  signal ADDR : unsigned(5 downto 0) := "000000";
  signal DReg : unsigned(4 downto 0);
  signal Hash : unsigned (6 downto 0) :="0000000";

  begin
  -- instantiate and connect memory
  Source: ENCR_ROM port map (CADDR => ADDR, CDOUT => DReg);

  -- read, encrypt and write data
  encrypt : process (Clk)
  variable HVar : unsigned (9 downto 0) :="0000000000";
  begin
     for i in 0 to 55 loop
       if rising_edge(Clk) then
         for j in 9 downto 5 loop
             HVar(j):=HVar(j-5); -- multiply by 32 (shift 5 bits)
             HVar(j-5):='0';
         end loop;
         HVar := (HVar + DReg) mod 127;
         ADDR <= ADDR + 1;
       end if;
     end loop;
     Hash<=HVar(6 downto 0);
  end process encrypt;
end RTL;
```

Folgende Abbildung zeigt einen Testlauf mit den Werten aus der Textdatei. Der Hash-Wert ändert sich hierbei deutlich mit jedem neuen Eingangswert. Wegen der Multiplikation (Schiebeoperation um 5 Bits) gerät hierbei allerdings die Vergangenheit im Sinne der vorausgegangenen Zeichen rasch in Vergessenheit, d.h. der so berechnete Hash-Wert repräsentiert vor allem die letzten Eingangswerte. Auch sind 7 Bits als Wortbreite zu gering. Gängige Wortbreiten für Hash-Werte sind

128 bzw. 160 Bits. Diese Werte werden dann in hexadezimaler Schreibweise mit 32 bzw. 40 Zeichen angegeben.

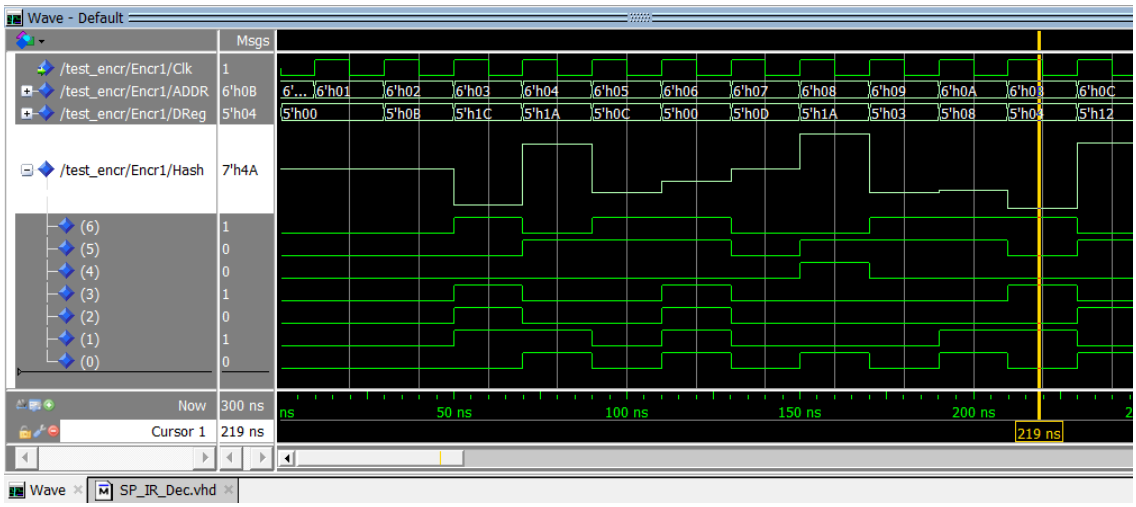


Bild 5.9 Hash-Funktion

Den Ansprüchen zum Schutz der Integrität von Daten bzw. für digitale Signaturen (Eindeutigkeit, Schutz vor Kollisionen) genügen solch einfache Verfahren nicht. Hierfür sind kryptographisch sichere Verfahren wie SHA (Secure Hash Algorithm) im Einsatz, deren Funktionsprinzip auf Kompression und Verwürfelung der Eingangsdaten beruht.

Seminararbeit S11: Entwerfen Sie eine Schaltung zur Berechnung und Überprüfung von Prüfsummen. Recherchieren Sie hierzu gängige Verfahren. Überprüfen Sie Ihre Schaltung mit Hilfe eines Beispiels.

6.4. AES

Ein in der Praxis gängiges Verfahren zur Verschlüsselung ist AES (Advanced Encryption Standard), siehe Literatur (8).

Übung 6.7: Machen Sie sich mit dem AES-Verfahren vertraut. Erläutern Sie das Funktionsprinzip. Skizzieren Sie den Ablauf des Verfahrens zur Verschlüsselung einer vorgegebenen Datei bzw. Nachricht.

Übung 6.8: Entwerfen Sie ein Konzept für die Implementierung der AES-Verschlüsselung mit einer Schlüssellänge von 128 Bit (Komponenten, Blockdiagramm, Ablauf).

Übung 6.9: Entwerfen Sie eine Schaltung zur AES-Verschlüsselung. Testen Sie die Schaltung im Simulator.

6.5. Angriffe mit roher Gewalt

Unter einem Angriff mit roher Gewalt (engl. brute force attack) wird das systematische Ausprobieren von Schlüsseln verstanden, mit dem Ziel, den passenden Schlüssel zu finden. Hierzu wird als Probe ein verschlüsselter Text und der passende Klartext benötigt. Den Klartext verschlüsselt man mit dem jeweils gewählten Schlüssel und vergleicht das Ergebnis mit dem vorliegenden

verschlüsselten Text. Stimmen beide überein, so hat man den passenden Schlüssel gefunden. Proben von Klartext und passendem verschlüsselten Text erhält man beispielsweise aus einer digitalen Signatur, bzw. aus einer entschlüsselten Kommunikation.

Das Gelingen des Angriffs ist abhängig vom jeweils benötigten Rechenaufwand. Umfasst die Länge des binären Schlüssels beispielsweise 48 Zeichen, so muss man alle möglichen Kombinationen des Schlüssels mit 48 Zeichen ausprobieren. Da es insgesamt 2^{48} mögliche Kombinationen gibt, sind hierzu 2^{48} Operationen nötig. Dieser Wert lässt sich näherungsweise ins Dezimalsystem umrechnen: $2^{-2} * 2^{50} \approx 0{,}25 * 10^{15}$ (mit der Annahme $2^{10} \approx 10^3$). Die Dauer, bis der passende Schlüssel gefunden ist, ist abhängig von der Rechenleistung der verwendeten Hardware. Angenommen, der verwendete Rechner kann 10^9 Schlüssel pro Sekunde ausprobieren, werden 10^6 Sekunden benötigt (ca. 11 Tage) um sämtliche Schlüssel auszuprobieren. Mit 50% Wahrscheinlichkeit ist ein passender Schlüssel bereits nach 10^5 Sekunden gefunden.

Die Schlüssellänge entscheidet somit über den Aufwand für einen erfolgreichen Angriff. Durch Erraten eines Teiles des Schlüssels, lässt sich dieser Aufwand reduzieren. Bei der Pfadfindermethode sind Teile des Schlüssel relativ leicht aus den Textproben zu erraten, da sich die Häufigkeitsverteilung der Buchstaben im Text durch die Verschlüsselung nicht ändert. Häufige Buchstaben wie im Deutschen das „E“ sind relativ leicht zu finden. Hierdurch reduziert sich die verbleibende Schlüssellänge. Auch Zahlen lassen sich relativ leicht im verschlüsselten Text ausfindig machen. Solche Angriffe zielen auf Schwachstellen im Algorithmus der Verschlüsselung.

Übung 6.10: Schätzen Sie den Aufwand zum Finden eines passenden AES-Schlüssels ab, wenn die Schlüssellänge 128 Bits beträgt. Welche Rechenleistung wäre nötig, um den Schlüssel durch systematisches Probieren in vertretbarer Zeit zu finden?

Übung 6.11: Wie steigt die Rechenleistung in 15 bzw. 30 Jahren unter der Annahme, dass die Rechenleistung sich alle 18 Monate verdoppelt (Mooresches Gesetz). Wie groß wäre der Aufwand zum Finden eines 128-Bit Schlüssels unter diesen Annahmen? Welchen Gewinn an Sicherheit bringt eine Schlüssellänge von 256 Bit?

7. Netzwerk

Das im Labor Nachrichtentechnik verwendete FPGA (LX9 Board) beinhaltet eine physikalische Ethernet-Schnittstelle, die auf physikalischer Schicht für die Programmierung zugänglich ist. Es besteht also die Möglichkeit, eine eigene Schicht 2 zu programmieren und im Netzwerk zu testen.

Seminararbeit S12: Entwerfen Sie ein standardkonformes Schicht 2 Protokoll und implementieren Sie das Protokoll auf dem FPGA. Testen Sie Ihre Implementierung im Netzwerk.

Seminararbeit S13: Verschlüsseln Sie die Schicht 2 mit einem der Verfahren aus Abschnitt 5. Testen Sie die Verschlüsselung zwischen zwei hiermit ausgestatteten Endpunkten über das Netzwerk.

8. Übungsaufgaben

8.1. Zustandsautomat für die serielle Schnittstelle

Zur Steuerung einer seriellen Schnittstelle soll ein Zustandsautomat verwendet werden. Der Automat soll die in Bild 1.10 im Vorlesungsteil gezeigten Steuersignale verwenden und dem in Bild 1.9 gezeigten Zustandsdiagramm folgen. Zu Erstellen sind ein Testprogramm für den Automaten, sowie ein Schaltungsentwurf, der eine SPI-Schnittstelle realisiert (siehe Übungen 1.3, 1.4 und 1.5).

Lösung: (1) Testprogramm:

```
--- Übliche Struktur eines Testprogramms (Component, Signale, DUT über
Port-Map einspannen), dann folgende Testschritte:

-- generate clock
  clock : process begin
    T_Serial_CLK <= '1';
    wait for 10 ns;
    T_Serial_CLK <= '0';
    wait for 10 ns;
  end process clock;

  test : process begin

    T_RS <= '0';            -- Reset (active low)
    wait for 5 ns;
    T_RS <= '1';
    wait for 5 ns;

    T_TX_Data <= x"00";     -- transmit first byte
    T_E <= "01";            -- TX_Start
    wait until T_A = "01";  -- QX_Ready

    T_E <= "10";            -- TX_Byte
    wait until T_A = "10";  -- QX_Byte
    wait for 5 ns;

    T_TX_Data <= x"bb";     -- transmit next byte
    T_E <= "01";            -- TX_Start
    wait until T_A = "01";  -- QX_Ready

    T_E <= "10";            -- TX_Byte
    wait until T_A = "10";  -- QX_Byte
    wait for 5 ns;

    --  close transmission
    T_E <= "11";            -- TX_Close
    wait until T_A = "11";  -- QX_Closed

    wait;

  end process test;
```

(2) Zustandsautomat: Das Zustandsdiagramm lässt sich in eine Zustandsübergangstabelle übertragen. Zustände, Steuersignale (=Ereignisse) und Quittungen (=Ausgangssignale des Automaten) werden hierbei nach einem geeigneten Schema kodiert. Die Implementierung erfolgt dann z.B. wie in folgendem HDL-Text.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Settings:
-- SPI-Mode 0: CPOL=0, CPHA=0
-- External Clock: 40000000 = 40 MHz
-- Sample_Clk: 40000000/4096 = e.g 10 kHz sampling rate
-- Serial_Clk: 40000000/32   = 1.25 MHz SPI Clock input  (forever)
-- SCLK:       follows Serial_Clk (8 cycles only)

entity Serial_Master is
  Port ( RS   : in STD_Logic;                    -- Reset (active low)
    TX_Data: in STD_LOGIC_VECTOR (7 downto 0);  -- Byte Register
    Serial_Clk    : in  STD_LOGIC;              -- Serial clock

    E       : in  STD_LOGIC_vector (1 downto 0); -- Control Events
    -- "01" TX_Start, "10" TX_Byte,"11" TX_Close

    A  : out  STD_LOGIC_vector (1 downto 0);     -- State Actions
    -- "01" QX_Ready, "10" QX_Byte,"11" QX_Closed

    -- serial bus, e.g. SPI
       MOSI       : out STD_LOGIC;    -- SPI Master out signal
       SCLK       : out STD_LOGIC;    -- SPI clock signal out
       SS         : out STD_LOGIC     -- SPI chip select signal
         );
end Serial_Master;

architecture RTL of Serial_Master is

-- internal signals and variables
signal txreg        : std_logic_vector(7 downto 0) := (others=>'0');
signal counter      : integer := 0;

type state_type is (Z0, Z1, Z2);
 -- Z: Wait; Z1: TX_S; Z2: TX

signal next_state, current_state: state_type;

begin

  -- update state register in engine
  update_state_register: process(Serial_Clk, RS) -- process #1
    begin
      if (RS = '0') then
```

```
          current_state <= Z0;
      elsif rising_edge(Serial_Clk) then
          current_state <= next_state;
      end if;
  end process update_state_register;

  -- implement state logic
  logic_next_state: process(E, current_state) -- process #2
    begin
      case current_state is
        when Z0 =>
            if E =  "01" then next_state <= Z1;
            end if;
       when Z1 =>
            if E =  "10" then next_state <= Z2;
            end if;
       when Z2 =>
            if E =  "01" then next_state <= Z1;
            elsif E = "11" then next_state <= Z0;
            end if;
        when others => next_state <= Z0;
      end case;
    end process logic_next_state;

-- state actions
 comb_logic_out: process(current_state, Serial_Clk) -- process 3
  begin
    case current_state is
      when Z0 =>
          SS <='1';  -- deselect chip
          SCLK <= '0';      -- SCLK operated in CPOL = 0
          A <= "11";        -- handshake signal QX_Closed

      when Z1 =>            -- initialize bus/ serial interface
          counter <= 8;     -- initialize counter for 8 bits transmission
          SS <='0';         -- chip select is active low
          SCLK <= '0';      -- SCLK operated in CPOL = 0
          txreg <= TX_Data;
          A <= "01";        -- handshake signal init. ready (QX_Ready)

      when Z2 =>            -- transfer one byte
        if(falling_edge(Serial_Clk) and (counter > 0))then
          SCLK <= '0'; -- SCLK follows serial clock for 8 counts
          MOSI <= txreg(7);
          for i in 0 to 6 loop
            txreg(7-i) <= txreg (6-i);
          end loop;
        elsif (rising_edge(Serial_Clk) and (counter > 0)) then
          SCLK <= '1'; -- SCLK follows serial clock for 8 counts
          counter <= counter - 1;
        end if;
```

```
        if (counter = 0) then
           A <= "10"; -- handshake signal for byte transfrd. (QX_Byte)
        end if;

      when others => A <= "00";

    end case;
  end process comb_logic_out;

end RTL;
```

8.2. Zustandsautomat für das FIR Filter

Das in Abschnitt 2.3 entworfene Top-Modul für ein FIR-Filter ist um einen Zustandsautomat zu ergänzen. Entwerfen Sie den Automaten und ein Testprogramm für den Automaten.

Lösung: siehe folgender HDL-Text

```
--- FSM and Top Module of FIR Filter (VHDL)

library ieee;
use ieee.std_logic_1164.all;

entity FSM_and_Top_Module is
  generic (L_BITS : natural := 4;    -- L bits wide (Address Space)
           M_BITS : natural := 16);  -- M bits wide (Word Width)

  port (Clk, RS, E_SR : in std_logic;   -- Clock, Reset, Sample Ready
        XS : in std_logic_vector(M_BITS - 1 downto 0); -- Sample Value
        Y : out std_logic_vector(M_BITS - 1 downto 0); -- result of FIR
                    Error : out std_logic);
end FSM_and_Top_Module ;

architecture RTL of FSM_and_Top_Module  is

type state_type is (Z0, Z1, Z2, Z3, Z4, Z5);
signal next_state, current_state: state_type;

component Memory_Unit is
       generic (L_BITS : natural;   -- L bits wide (Address Space)
                M_BITS : natural);  -- M bits wide (Word Width)
       port ( Clk, RS : in std_logic;     -- Clock, Reset
       A_XR, A_CR1, A_CR2 : in std_logic; -- inputs from FSM
       E_XR, E_CR1, E_CR2, E_CR3 : out std_logic; -- events to FSM
       MCOEFF, MXN : out std_logic_vector(M_BITS - 1 downto 0);
       XS : in std_logic_vector(M_BITS - 1 downto 0)); -- from ADC
end component;

component MAC_Unit is
  port (Clk, Clr : in std_logic;  -- clock, Clear MAC register
  EN_Reg, EN_Out : in std_logic;  -- Enable MAC and Output register
  XN : in std_logic_vector (15 downto 0);    -- input sample
```

```
    COEFF : in std_logic_vector (15 downto 0); -- filter coefficient
    YN : out std_logic_vector (15 downto 0));  -- output sample
  end component;

  -- Top_Module internal signals
  signal FXN : std_logic_vector (M_BITS-1 downto 0);
  signal FXS : std_logic_vector (M_BITS-1 downto 0);
  signal FYN : std_logic_vector (M_BITS-1 downto 0);
  signal FCOEFF : std_logic_vector (M_BITS-1 downto 0);

  signal A_XR, A_CR1, A_CR2, Clr, EN_Out, EN_Reg : std_logic := '0';
  signal E_XR, E_CR1, E_CR2, E_CR3 : std_logic;


  begin

  -- connect Memory_Unit and MAC_Unit to Top Module
  MyMem: Memory_Unit generic map(L_BITS => L_BITS, M_BITS => M_BITS)
         port map (Clk=>Clk, RS=>RS, A_XR => A_XR, A_CR1 => A_CR1,
            A_CR2 => A_CR2,  E_XR => E_XR,  E_CR1=>E_CR1, E_CR2=>E_CR2,
            E_CR3=>E_CR3, MCOEFF=>FCOEFF, MXN=>FXN, XS=>FXS);

  MyMAC: MAC_Unit port map (Clk=>Clk, Clr=>Clr, EN_Reg=>EN_Reg,
                  EN_Out=>EN_Out, XN=>FXN, COEFF=>FCOEFF, YN=>FYN);

  -- run processes
  update_state_register: process(Clk, RS)
    begin
         if (RS='1') then
           current_state <= Z0;
         elsif falling_edge(Clk) then
           current_state <= next_state;
         end if;
  end process update_state_register;

  -- state transitions
  logic_next_state: process(E_SR, E_XR, E_CR1, E_CR2, E_CR3,
   current_state)
    begin
        case current_state is
           when Z0 => -- Wait (idle state)
                  if E_SR = '1'  then  next_state <= Z1; end if;
           when Z1 => -- write new sample value XS
                  if E_XR = '1'  then  next_state <= Z2; end if;
           when Z2 => -- synch  read address to last value
                  if E_CR1 = '1' then  next_state <= Z3; end if;
           when Z3 => -- read out COEFF and XN
                  if E_CR2 = '1' then  next_state <= Z4; end if;
           when Z4 => -- MAC COEFF and XN to result YN
                  if E_CR1 = '1' then  next_state <= Z3; end if;
                  if E_CR3 = '1' then  next_state <= Z5; end if;
           when Z5 => -- transfer result YN to output Y
```

```
                  if E_CR3 = '1' then  next_state <= Z0; end if;
            when others => next_state <= Z0;
         end case;
   end process logic_next_state;

   -- state actions
   logic_out: process(current_state)
    begin
         case current_state is
            when Z0 =>
                 EN_Out <= '0'; EN_Reg <= '0'; -- disable MAC
                 A_XR <= '0';                  -- disable X_RAM
                 A_CR1 <= '0'; A_CR2 <= '0';   -- and COEFF_RAM
                 Error <= '0';

            when Z1 =>
                 FXS <= XS;    -- get new sample from input
                 A_XR <='1';   -- trigger writing of XS
                 Clr <= '1';   -- clear MAC unit

            when Z2 =>
                 A_XR <= '0';  -- disable write port of X_RAM
                 Clr <= '0';   -- deselect clearing of MAC unit
                 A_CR1 <= '1'; -- synch X_RAM read address to last sample

            when Z3 =>
                 A_CR1 <= '0';  -- fix RADDR
                 EN_Reg <= '0'; -- disable MAC
                 A_CR2 <= '1';  -- enable XRAM

            when Z4 =>
                 EN_Reg <= '1'; -- enable MAC
                 A_CR2 <= '0';  -- disable XRAM

            when Z5 =>
                 EN_Reg <= '0'; -- disable MAC
                 A_CR2 <= '0';  -- disable XRAM
                 EN_Out <= '1'; -- transfer MAC.YN to FYN
                 Y <= FYN;      -- transfer FYN to output Y

            when others => Error <= '1';
         end case;
   end process logic_out;

   end RTL;
```

Erläuterungen: der Entwurf folgt dem Zustandsdiagramm und enthält keine besonderen Eigenschaften. Wegen der Steuerung durch Clock-Enable-Signale müssen die Zustandswechsel speziell innerhalb eines Taktzyklus erfolgen, daher arbeitet dieser Automat mit fallenden Taktflanken. Für eine Realisierung mit steigenden Flanken wären der Automat bzw. die Module zu modifizieren.

Testprogramm: siehe folgender HDL-Text.

```
--- Testbench for FSM and Top Module of FIR Filter (VHDL)

library ieee;
use ieee.std_logic_1164.all;

entity test_FSM_and_Top_Module is
   generic (L_BITS : natural := 4;   -- L bits wide (Address Space)
            M_BITS : natural := 16);  -- M bits wide (Word Width)
end test_FSM_and_Top_Module;

architecture Behavioural of test_FSM_and_Top_Module  is

component FSM_and_Top_Module is
   generic (L_BITS : natural;   -- L bits wide (Address Space)
            M_BITS : natural);  -- M bits wide (Word Width)
   port ( Clk, RS, E_SR : in std_logic; -- Clock, Reset, Sample Ready
      XS : in std_logic_vector(M_BITS - 1 downto 0);  -- Sample Value
      Y : out std_logic_vector(M_BITS - 1 downto 0);  -- result of FIR
      Error : out std_logic);
end component;

-- Test bench internal signals
signal T_Clk, T_RS, T_E_SR, T_Error : std_logic :='0';
signal T_XS, T_Y : std_logic_vector (M_BITS-1 downto 0);
begin

-- connect FSM_and_Top_Module to testbench
MyTop: FSM_and_Top_Module
          generic map(L_BITS => L_BITS, M_BITS => M_BITS)
          port map (Clk=>T_Clk, RS=>T_RS, E_SR => T_E_SR, XS=>T_XS,
                    Y=>T_Y, Error=>T_Error);

-- run tests

reset_and_first_sample : process begin
   -- reset
   T_RS <= '1'; wait for 10 ns; T_RS <= '0'; wait for 20 ns;
   -- first sample
   T_XS <= x"AAAA"; T_E_SR <= '1'; wait for 20 ns; T_E_SR <= '0';
   wait;
end process reset_and_first_sample;

clock : process begin
   T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
end process clock;

end Behavioural;
```

Die Testfälle sind hier denkbar einfach: (1) Reset und Vorgabe des ersten Abtastwertes mit dem Startsignal E_SR (Sample Ready), (2) den Rest überlässt man dem Automaten, der nur noch ein Taktsignal vorgegeben bekommt. Im realen Fall bekäme man die Abtastwerte mit ihren Startsignalen

von einem Analog-Digitalwandler. Die Ergebnisse der Filterung wären an einen Digital-Analogwandler zu übergeben. Alle erforderlichen Berechnungen sollte das FIR-Filter eigenständig erledigen.

Testlauf im Simulator:

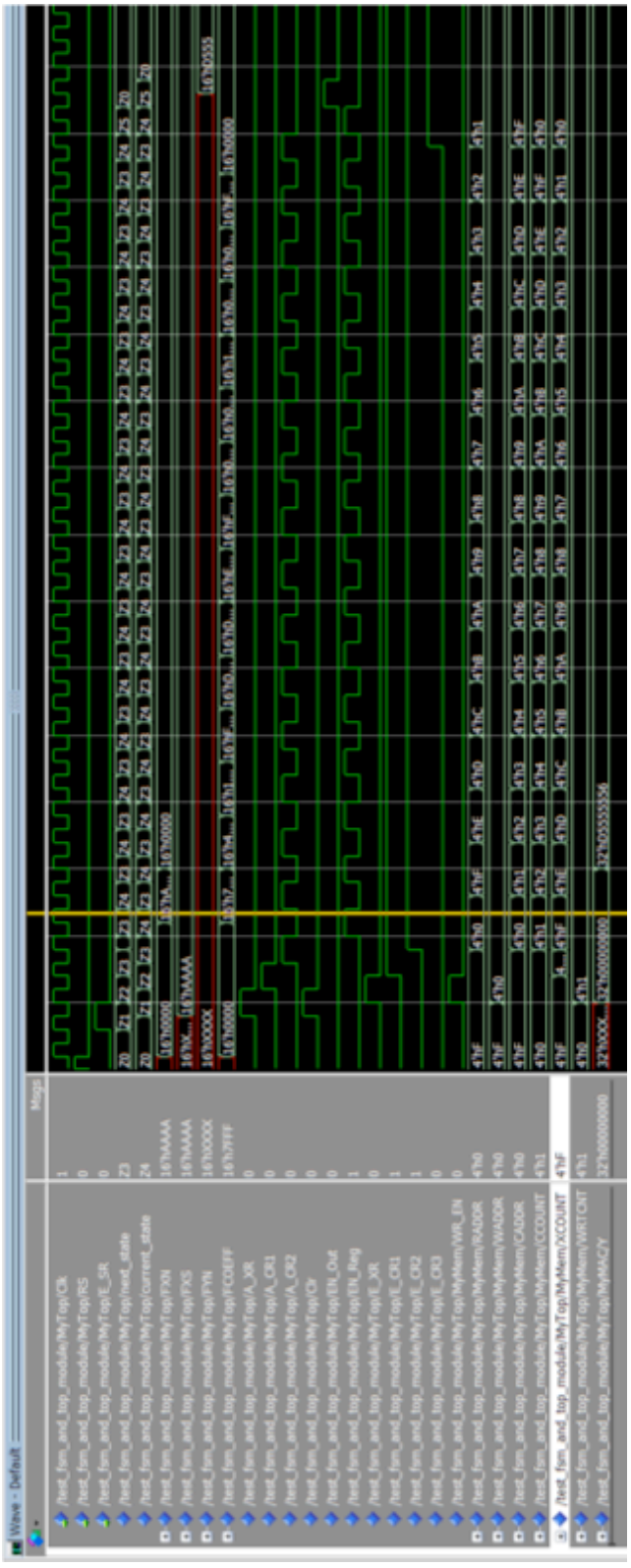


Bild 8.1 Testlauf des Filtern mit Zustandsautomat

Der Testlauf zeigt die Steuersignale und Quittungen, sowie die Zustände des Automaten. Die korrekte Funktion des Automaten erkennt man am Signalverlauf. Ebenfalls dargestellt sind die Adressen der Speichereinheit für das Schreiben des Eingangswertes, sowie das Lesen der Koeffizienten und zugehörigen gespeicherten Abtastwerte. Die korrekte Zählweise lässt sich ebenfalls überprüfen. Wenn der Index des letzten Koeffizienten erreicht ist, meldet die Speichereinheit E_CR3, wechselt in den Zustand Z5 und übergibt das Ergebnis der MAC-Einheit an das bis dahin undefinierte Signal. FYN. Die Zwischenwerte der internen Kumulation sind in der letzten Zeile dargestellt.

Bemerkung: Wenn man den Automaten von fallenden Taktflanken auf ansteigende Taktflanken umstellt, ergibt sich ein Problem: manche Zustandsübergänge dauern nun mehrere Taktzyklen. Grund hierfür ist, dass beim Eintreten des Automaten dieser eine Aktion auslöst, die Quittung der Aktion zwar noch innerhalb des gleichen Zyklus erfolgt, diese jedoch erst mit der nächsten (steigenden) Taktflanke abgefragt wird. Somit verbleibt der Automat in diesem Fall zwei Taktzyklen im Zustand.

Dieses Verhalten ist daher problematisch, da die Module im Datenpfad nun ebenfalls eine weitere Taktflanke erhalten, wodurch in diesem Fall die Adresszähler zu weit laufen (konkret: der Zähler zum Schreiben des nächsten Wertes, wodurch auch der Lesezeiger des ersten Abtastwertes eine Stützstelle zu weit beginnt). Folgende Abbildung zeigt den Effekt nach Umstallung des Automaten auf steigende Taktflanken.

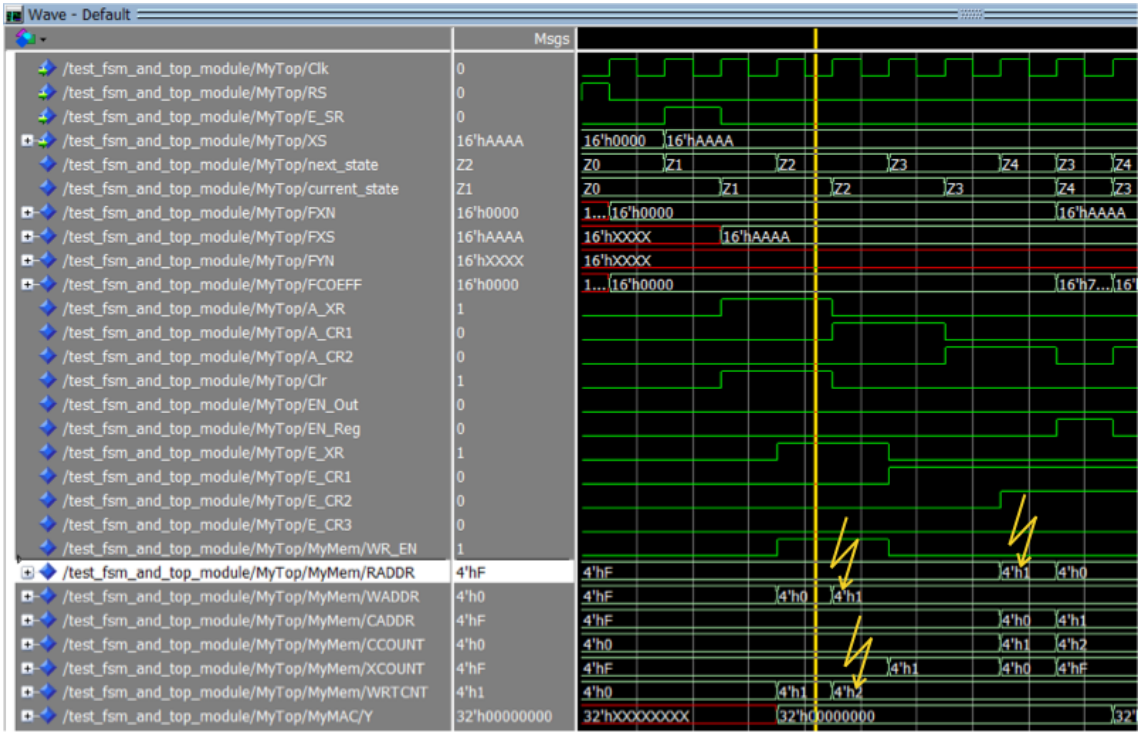


Bild 8.2 Betrieb des Automaten mit steigenden Taktflanken

Abhilfe ist auf folgenden Wegen möglich: (1) Man verändert den Zustandsautomaten so, dass unabhängig vom Takt Eingänge in einem gegebenen Zustand direkt auf die Ausgänge wirken. Im konkreten Fall wären hierfür die Zustände Z1 und Z2 zusammen zu fassen und die Signale E_XR und E_CR1 unabhängig vom Takt abzufragen. Eine Vorlage für eine HDL-Beschreibung hierfür finden Sie in [1] (siehe Literaturverzeichnis) in Aufgabe 6.5, Frage 2.6. Diese Variante des Zustandsautomaten fasst das Übergangsschaltnetz und das Ausgangsschaltnetz zu einem Schaltnetz zusammen und schaltet Ausgänge zwar zustandsspezifisch, jedoch direkt in Abhängigkeit der Eingänge.

Eine andere Möglichkeit (2) besteht darin, den Automaten unabhängig vom Empfang der Quittung nach einem Taktzyklus in den nächsten Zustand zu schicken. Für die Zustände Z1 und Z2 wird hierfür ein Signal E_Carry_On für die Zustandsübergänge eingesetzt, das unverändert auf logisch ‚1' bleibt (anstelle der Signale E_XR und E_CR1). Folgender Programmtext zeigt die Änderung im Automaten.

```
-- Top_Module internal signals
...
signal E_Carry_On : std_logic := '1';
...
-- run processes
update_state_register: process(Clk, RS)
  begin
       if (RS='1') then
         current_state <= Z0;
       elsif rising_edge(Clk) then
         current_state <= next_state;
       end if;
end process update_state_register;

-- state transitions
logic_next_state: process(E_SR, E_XR, E_CR1, E_CR2, E_CR3,
 current_state)
  begin
       case current_state is
         when Z0 => -- Wait (idle state)
                if E_SR = '1'  then  next_state <= Z1; end if;
         when Z1 => -- write new sample value XS
         --     if E_XR = '1'  then  next_state <= Z2; end if;
                if E_Carry_On = '1' then next_state <= Z2; end if;
         when Z2 => -- synch  read address to last value
         --     if E_CR1 = '1' then  next_state <= Z3; end if;
                if E_Carry_On = '1' then   next_state <= Z3; end if;
         when Z3 => -- read out COEFF and XN
                if E_CR2 = '1' then  next_state <= Z4; end if;
         when Z4 => -- MAC COEFF and XN to result YN
                if E_CR1 = '1' then  next_state <= Z3; end if;
                if E_CR3 = '1' then  next_state <= Z5; end if;
         when Z5 => -- transfer result YN to output Y
                if E_CR3 = '1' then  next_state <= Z0; end if;
              when others => next_state <= Z0;
            end case;

end process logic_next_state;
```

Da das Signal E_Carry_On unverändert bleibt, wird es auch nicht im Sensitivitätsbereich (d.h. als Übergabeparameter) des Prozesses benötigt. Mit der Änderung verhält sich der Automat wie in folgender Abbildung gezeigt.

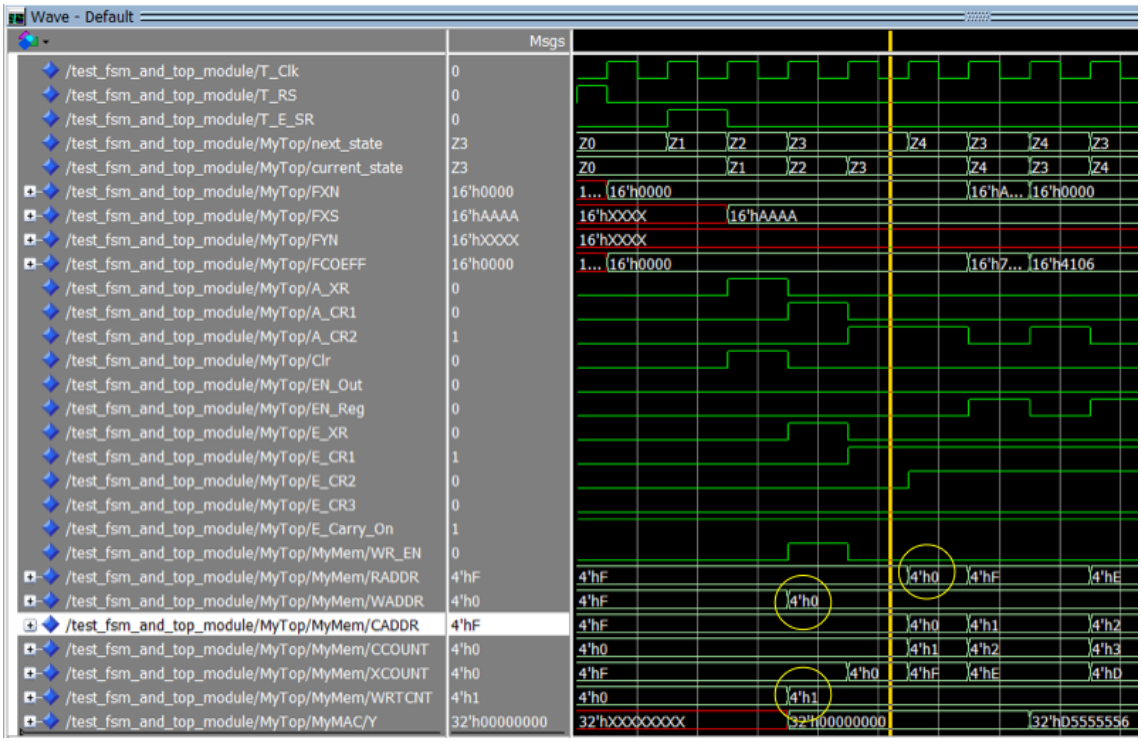


Bild 8.3 Betrieb des Automaten mit erzwungenen Zustandsübergängen für Z1 und Z2

8.3. Prozessor mit Akku-Architektur

Folgender Ausschnitt zeigt die Schleife eines Hochsprachenprogramms.

```
// Lauflicht

// global variables
byte lights;

// program loop
void loop() {

   if (lights == 0) {lights = 128; }  // start from 0b10000000

   PORTB = lights;                    // lighting up

   lights = lights >> 1;              // shift right

   delay(100);                        // wait 100 ms
}
```

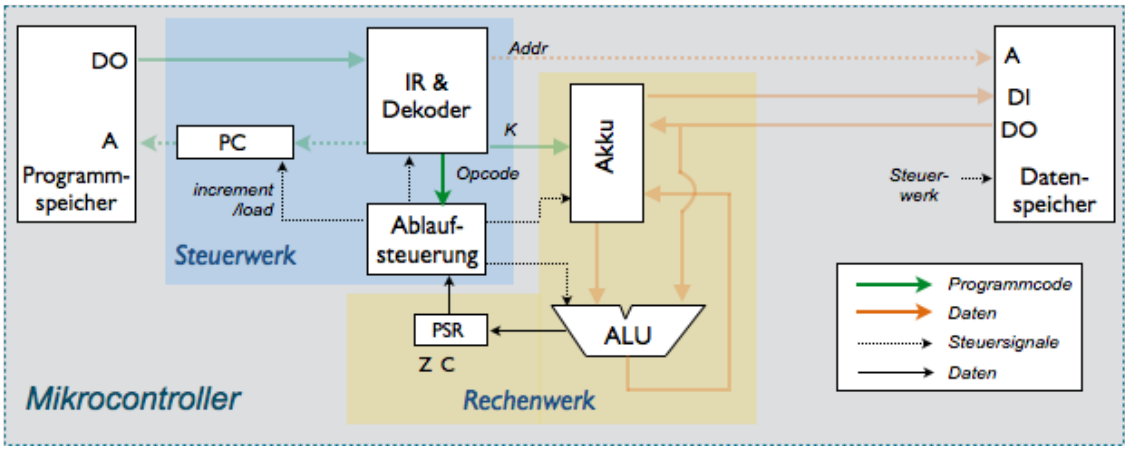
Frage 8.3.1: Übersetzten Sie das Programm in Assemblersprache mit dem Befehls-satz des MCT-Mikrocontrollers. Bitte ergänzen Sie Kommentare zum Ablauf.

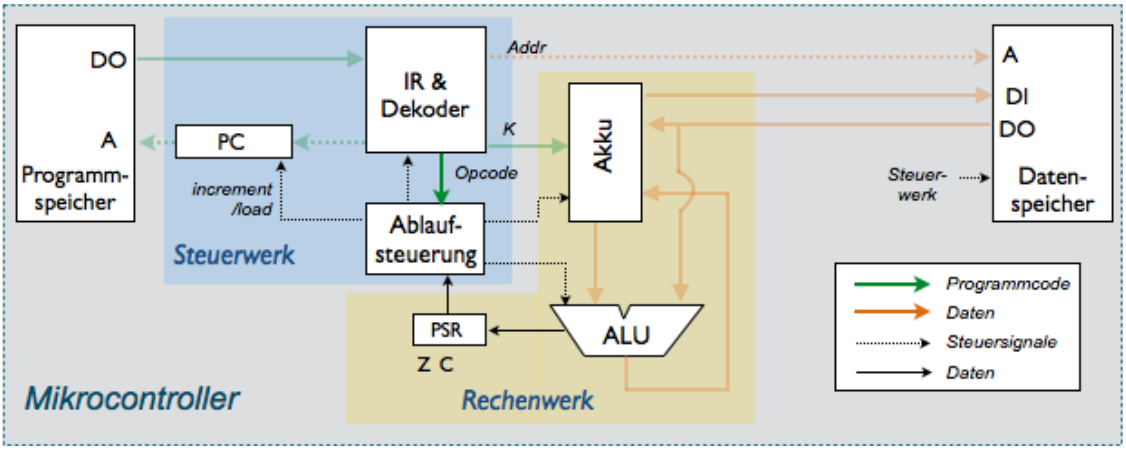
Frage 8.3.2: Zum Ablauf auf dem Mikrocontroller muss das Programm in Maschinen-sprache übersetzt werden. Beschreiben Sie den grundsätzlichen Ablauf dieser Übersetzung. Wie

werden Befehle und Operanden kodiert? Hinweis: Sie brauchen das Programm hierzu nicht in Maschinensprache zu übersetzen.

Frage 8.3.3: Beschreiben Sie den Ablauf Ihres Programms in einer Tabelle über 10 Takte. Die Tabelle soll folgende Signale zeigen: Clock, PC, PS_DO, Opcode (Dekoder-Ausgang), Addr, K, ALU-Out, Carry-Bit, Akku, PortB. Die Bezeichnungen entsprechen dem Blockschaltbild des Prozessors in der folgenden Abbildung.

Frage 8.3.4: Folgende Abbildung zeigt den MCT-Mikrocontroller mit Akku-Architektur. Beschreiben Sie den Ablauf der Schiebeoperation lsr über zwei Takte.

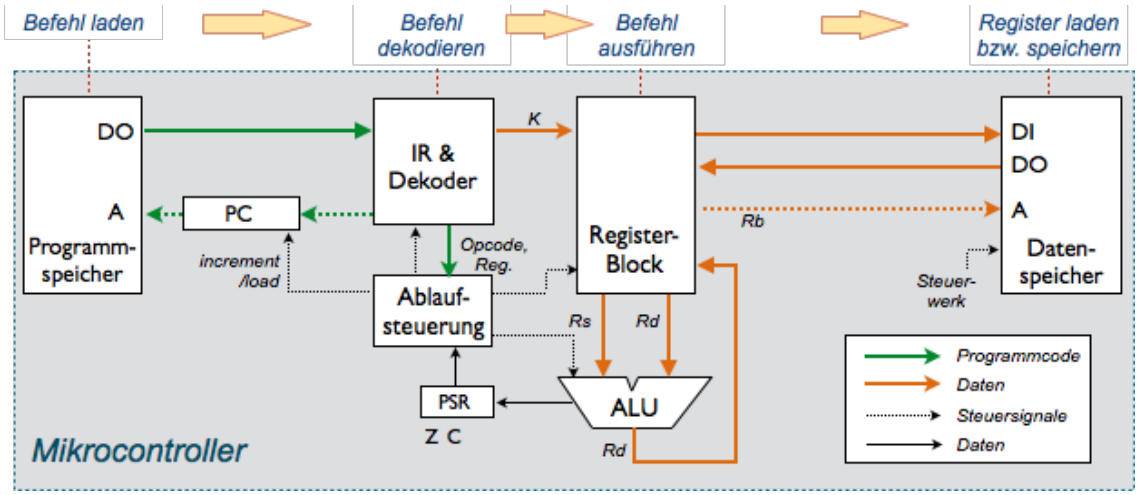




Hinweis: Startpunkt ist der dekodierte Befehl am Ausgang des Dekoders. Zeichnen Sie die jeweils aktiven Signale (Leitungen) im Diagramm nach.

8.4. Prozessor mit Register-Architektur

Anstelle des Akkus wird eine Registerbank mit 16 Registern eingeführt. Jedes Register kann als Akku verwendet werden. Folgendes Blockschaltbild beschreibt die Realisierung.



Frage 8.4.1: Beschreiben Sie die wesentlichen Unterschiede zur Akku-Architektur aus Aufgabe 1 (Laden von Registern, ALU-Operationen, Laden und Speichern von Operan-den im Datenspeicher).

Frage 8.4.2: Nennen Sie Unterschiede im Befehlssatz und in der Befehlskodierung (Beispiele: Sprungbefehle, Laden und Speichern von Ergebnissen, ALU-Operationen).

Frage 8.4.3: Gibt es Vorteile, die die Registerarchitektur gegenüber der Akku-Architektur bringt (Geschwindigkeit, Programmierbarkeit)? Lassen sich die Register als Teil des Datenspeichers realisieren?

Frage 8.4.4: Schreiben Sie Ihr Programm aus Aufgabe 1.1 auf die Registerarchitektur um. Welche Unterschiede gibt es?

8.5. Erweiterungen des Prozessors (Akku-Architektur)

Zur Unterstützung der strukturierten Programmierung durch Unterprogramme wird der Befehlssatz des Mikrocontrollers durch folgende Befehle erweitert:

- CALL Addr: Ruft Unterprogramm auf.
- RET: Rückkehr aus dem Unterprogramm.
- PUSH: Sichert den Inhalt des Akkus im Datenspeicher.
- POP: Stellt den Inhalt des Akkus aus dem Datenspeicher wieder her.

Die Sicherung des Programmzählers und Akkus im Datenspeicher geschieht in einem gesonderten Bereich (Stapel, engl. Stack), der mit Hilfe eines Stapelzeigers (SP für engl. Stack Pointer) adressiert wird. Folgende Übersicht zeigt die Befehlserweiterung.

Erweiterungen

Befehlssatz			Beschreibung	Prozessor Status Register (PSR) Flags				
Assembler	Kürzel	Opcode		Zero (Z)	Carry (C)	oVerflow (V)	Sign (S)	Neg. (N)
push	PUSH	E	(SP)<= Akku; SP <= SP+1	-	-	-	-	-
pop	POP	F	SP<SP-1; Akku<= (SP)	-	-	-	-	-
call K	CALL	10	(SP)<= PC; SP<=SP +1; jump to address K	-	-	-	-	-
ret	RET	11	SP<= SP-1; PC<=(SP); jump to PC	-	-	-	-	-

Erweiterungen

nop, push, pop, ret

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					(leer)										

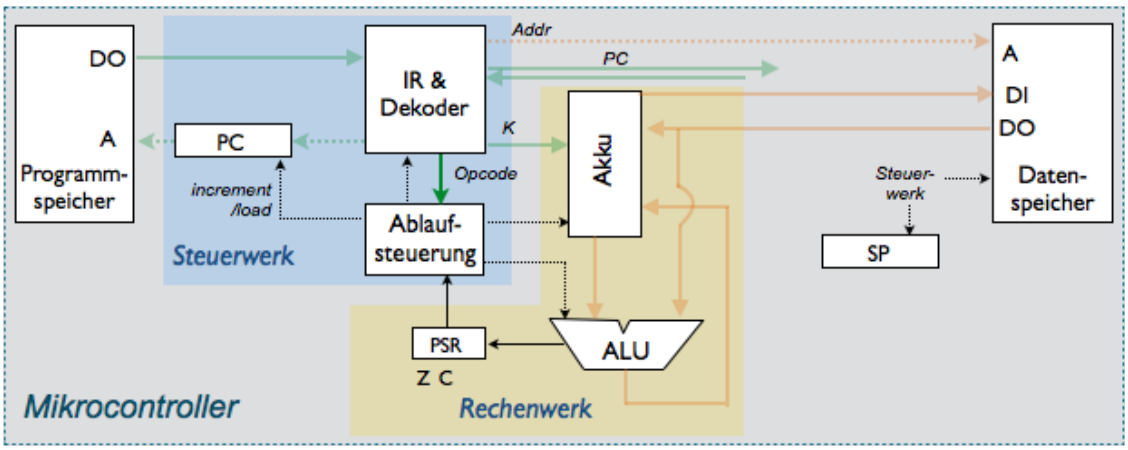
Sprungbefehle
jmp, call

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode					X	X	X	K8 (Zieladresse)							

Frage 8.5.1: Erläutern Sie die Funktion der Befehle im Detail (Akku, Stapel, PC).

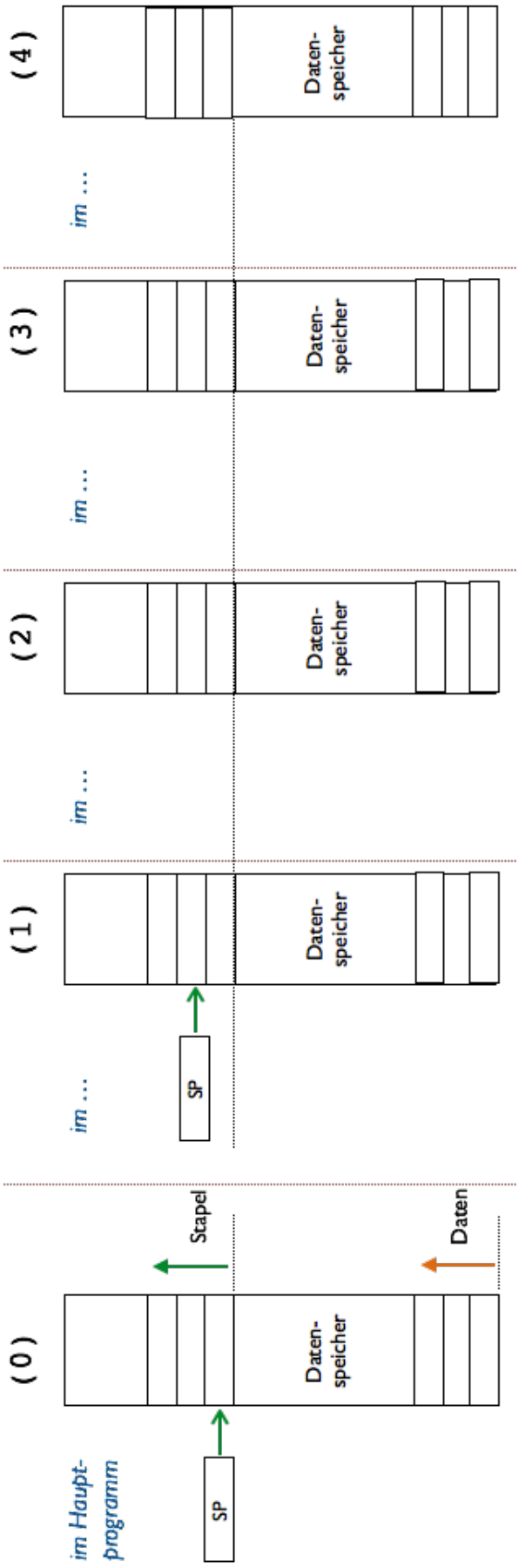
Frage 8.5.2: Welche Befehle werden vom Hauptprogramm aufgerufen? Welche Befehle werden vom Unterprogramm aufgerufen? Skizzieren Sie den Ablauf des Aufrufs eines Unterprogramms in einem Aktivitätsdiagramm.

Frage 8.5.3: Zur Realisierung o.g. Befehle muss der Prozessor erweitert werden. Ergänzen Sie die erforderlichen Erweiterungen im Blockschaltbild.



Frage 8.5.4: Wie viele Takte benötigen die Befehl jeweils: CALL Addr, RET, PUSH, POP. Begründen Sie Ihre Antwort.

Frage 8.5.5: Folgendes Diagramm beschreibt den Stapel im Datenspeicher mit dem SP als Zeiger nach Ablauf eines Befehls. Ergänzen Sie das Diagramm für folgende Schritte: (0) Startpunkt, (1) CALL Addr, (2) PUSH, (3) POP, (4) RET.


(4)
Daten-
speicher
im ...
(3)
Daten-
speicher
im ...
(2)
Daten-
speicher
im ...
(1)
Daten-
speicher
im ...
SP
(0)
Stapel
Daten
Daten-
speicher
im Haupt-
programm
SP

Frage 8.5.6: Das Zeitdiagramm in der folgenden Abbildung (Seite 7) beschreibt den Ablauf eines Programms.

(a) Rekonstruieren Sie den Programmtext aus dem Ablauf (bitte mit Zeilennummern, mit Adressen und Operanden, sowie mit Kommentaren).

(b) Erläutern Sie den Ablauf der Befehle PUSH und POP im Detail (Inhalt Akku und Stapel)

(c) Erläutern Sie den Ablauf der Befehle CALL und RET im Detail (Programmzähler, folgender Befehl, Speicherung und Restaurierung des Programmzählers, Rücksprung an welche Adresse, Ursprung und Zweck des Befehls NOP im Anschluss an CALL).

(d) Erstellen Sie ein Zustandsdiagramm des Prozessors mit den verwendeten Befehlen.

Hinweis: Sie können Zusammenhänge gerne auch im Diagramm mit Pfeilen bzw. Kommentaren markieren.

Englisch - Deutsch

Black box	geschlossenes System
Combinatorial circuit	Schaltnetz
Combinational logic	kombinatorische Logik
Control	Steuerung
Data path	Datenpfad
Decryption	Entschlüsselung
Device under test	Prüfling
Encryption	Verschlüsselung
Finite State Machine	Zustandsautomat
Instruction Pointer	Befehlszeiger
Instruction Register	Befehlsregister
Interrupt	Unterbrechung
Interrupt Routine	Unterbrechungsroutine
Key	Schlüssel
Latch	Auffangregister
Program Counter	Befehlszähler
Run Time Environment	Laufzeitumgebung
Sampled Value	Abtastwert
Sampling Rate	Abtastrate
Sequential circuit	Schaltwerk, getaktete Logik
Stack Pointer	Stapelzeiger
State diagram	Zustandsdiagramm
State event table	Zustandsübergangstabelle
Testbench	Testumgebung
White box	offenes System

...

Abkürzungen

ALU	Arithmetic Logic Unit
CPLD	Complex Programmable Logic Device
DNF	Disjunktive Normalform
DUT	Device under Test
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPIO	General Purpose IO
HDL	Hardware Description Language
IIR	Infinite Impulse Response
IR	Instruction Register
KNF	Konjunktive Normalform
LSB	Least Significant Bit
LUT	Look-up Table
MSB	Most Significant Bit
PLA	Programmable Logic Array
RAM	Random Access Memory
ROM	Read Only Memory
VHDL	Very High Speed Integrated Circuit HDL

...

Literatur

(1) S. Rupp, Entwurf digitaler Systeme, Vorlesungsmanuskript, DHBW, 2013

(2) S. Rupp, VHDL-Dateien zur Implementierung des Mikroprozessors, VHDL, 2014

(3) Jürgen Reichardt, Bernd Schwarz, VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme, Oldenbourg Wissenschaftsverlag; 2012 (6. aktualisierte Auflage), ISBN-13: 978-3486716771

(4) VHDL Online Reference Guide: http://www.hdlworks.com/hdl_corner/vhdl_ref/index.html

(5) HDL-Simulator: http://model.com/content/modelsim-pe-student-edition-hdl-simulation

(6) Mikrocontroller: Günter Schmitt, Mikrocomputertechnik mit Controllern der Atmel AVR-RISC-Familie: Programmierung in Assembler und C - Schaltungen und Anwendungen, Oldenbourg Wissenschaftsverlag, 2010, ISBN-13: 978-3486589887

Anhang A - D/A Wandler mit SPI Schnittstelle

Auszug aus dem Datenblatt, Quelle: Analog Devices

AD7303

DB15 (MSB)															DB0 (LSB)
$\overline{\text{INT}}$/EXT	X	LDAC	PDB	PBA	$\overline{\text{A}}$/B	CR1	CR0	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Control Bits								Data Bits							

Figure 24. Input Shift Register Contents

Bit Location	Mnemonic	Description
DB15	$\overline{\text{INT}}$/EXT	Selects between internal and external reference.
DB14	X	Uncommitted bit.
DB13	LDAC	Load DAC bit for synchronous update of DAC outputs.
DB12	PDB	Power-down DAC B.
DB11	PDA	Power-down DAC A.
DB10	$\overline{\text{A}}$/B	Address bit to select either DAC A or DAC B.
DB9	CR1	Control Bit 1 used in conjunction with CR0 to implement the various data loading functions.
DB8	CR0	Control Bit 0 used in conjunction with CR1 to implement the various data loading functions.
DB7–DB0	Data	These bits contain the data used to update the output of the DACs. DB7 is the MSB and DB0 the LSB of the 8-bit data word.

CONTROL BITS

LDAC	$\overline{\text{A}}$/B	CR1	CR0	Function Implemented
0	X	0	0	Both DAC registers loaded from shift register.
0	0	0	1	Update DAC A input register from shift register.
0	1	0	1	Update DAC B input register from shift register.
0	0	1	0	Update DAC A DAC register from input register.
0	1	1	0	Update DAC B DAC register from input register.
0	0	1	1	Update DAC A DAC register from shift register.
0	1	1	1	Update DAC B DAC register from shift register.
1	0	X	X	Load DAC A input register from shift register and update both DAC A and DAC B DAC registers.
1	1	X	X	Load DAC B input register from shift register and update both DAC A and DAC B DAC registers outputs.

$\overline{\text{INT}}$/EXT	Function
0	Internal $V_{DD}/2$ reference selected.
1	External reference selected; this external reference is applied at the REF pin and ranges from 1 V to $V_{DD}/2$.

PDA	PDB	Function
0	0	Both DACs active.
0	1	DAC A active and DAC B in power-down mode.
1	0	DAC A in power-down mode and DAC B active.
1	1	Both DACs powered down.

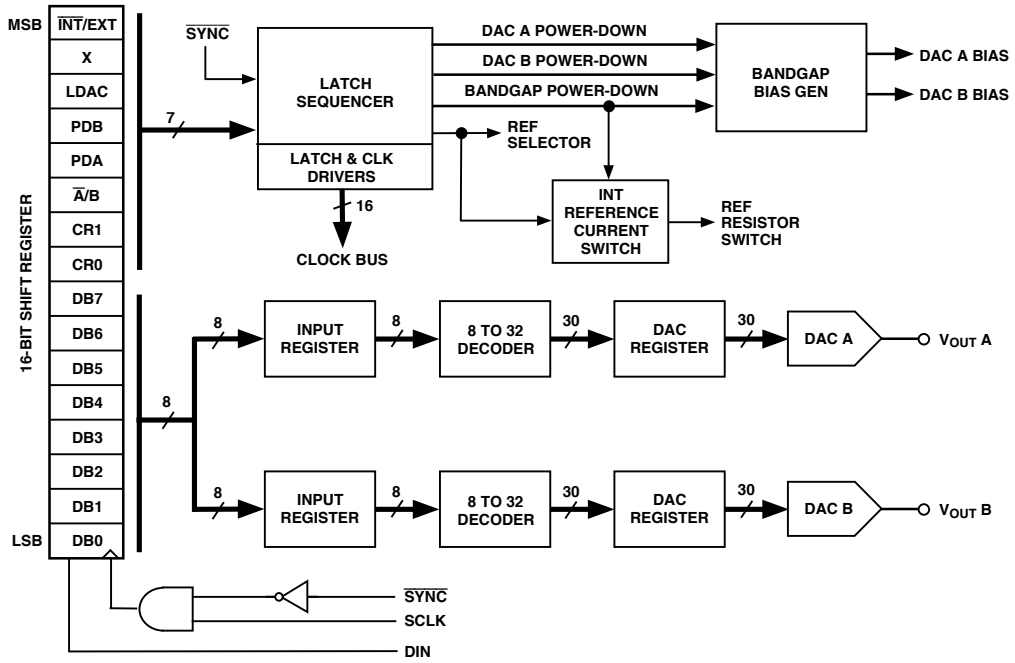
Betriebsarten:

(1) Command = 0b00000000: Parallelbetrieb beider DACs

(2) Beide Kanäle mit unterschiedlichen Signalen nutzen:

Command_1 = 0b00000001: Register A aus Schieberegister laden

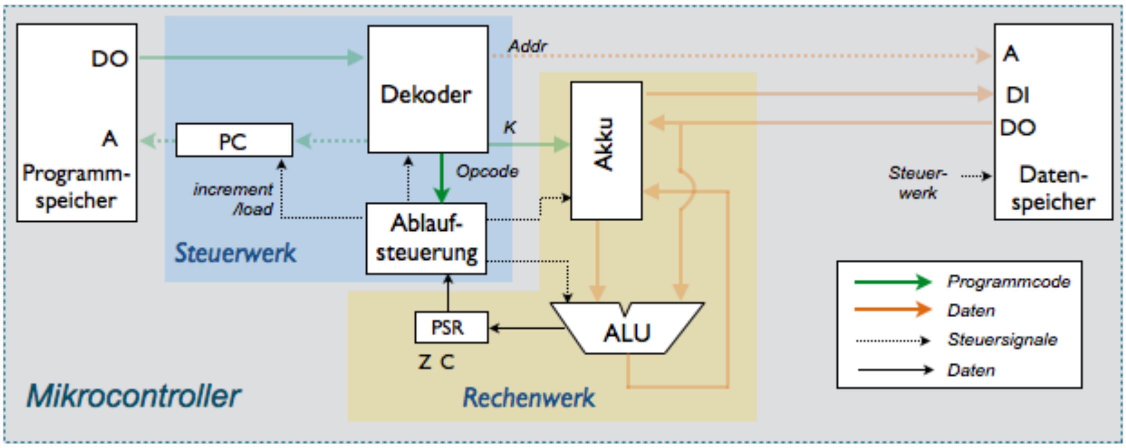
Command_2 = 0b00100100: Register B aus Schieberegister laden, beide Ausgänge aktualisieren

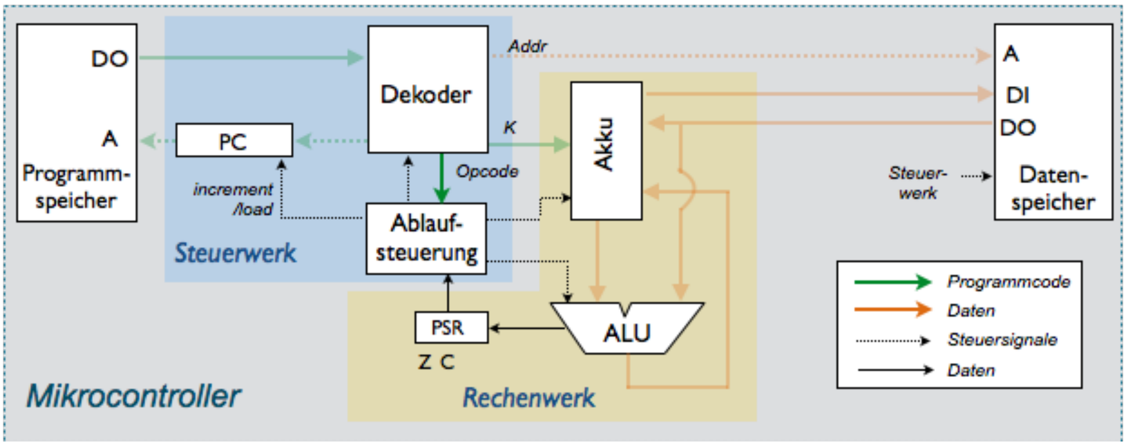


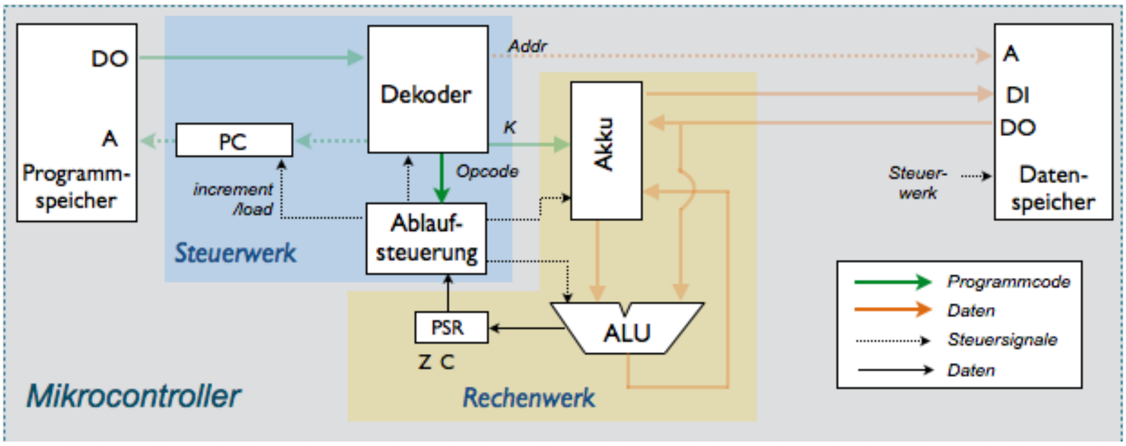
Quelle: Analog Devices

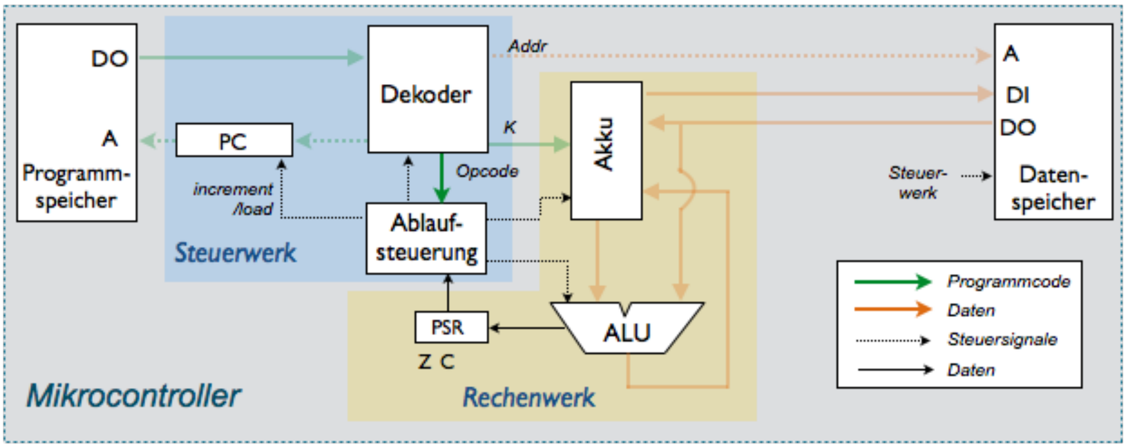
Anhang B - Mikroprozessor mit Akku-Architektur

Folgende Vorlagen sind für Skizzen zum Ablauf von Befehlen verwendbar.

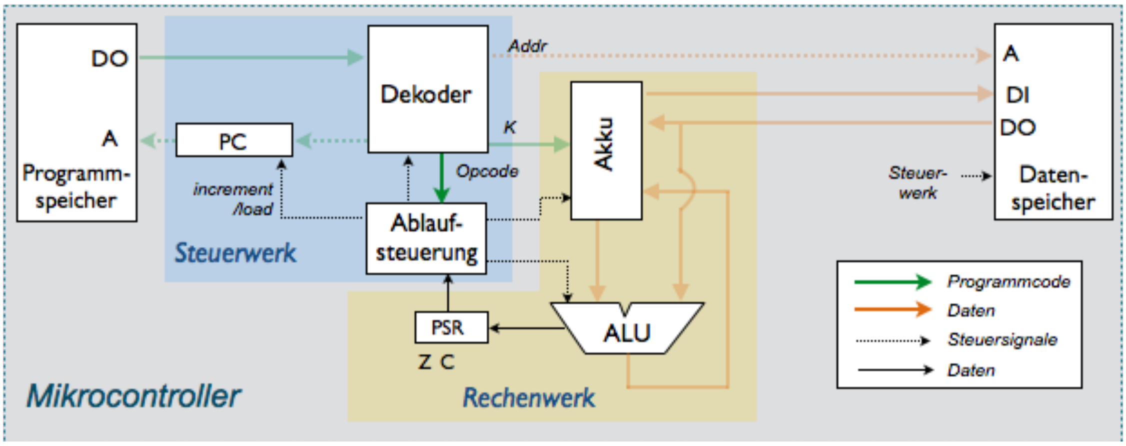




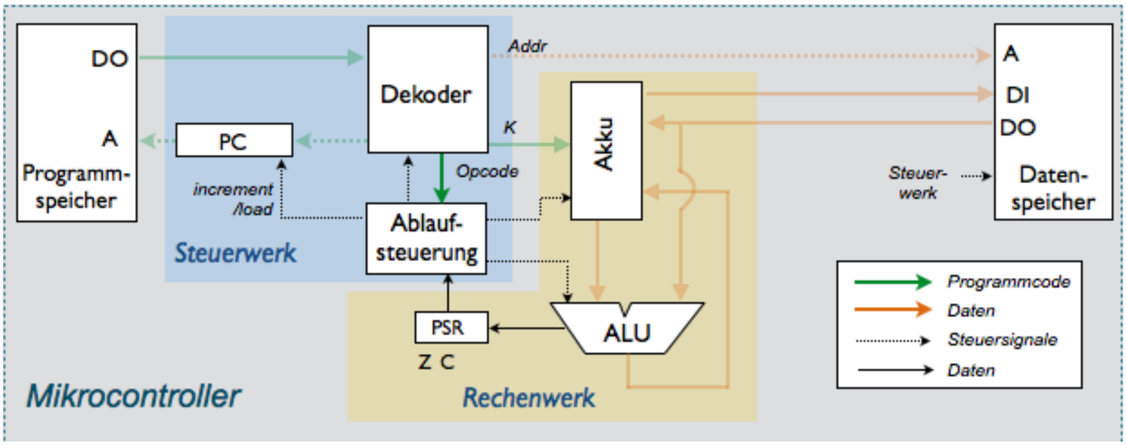




DO
A
Programm-
speicher
PC
increment
/load
Steuerwerk
Dekoder
Addr
K
Opcode
Ablauf-
steuerung
Akku
PSR
Z C
ALU
Rechenwerk
A
DI
DO
Steuer-
werk
Daten-
speicher
Programmcode
Daten
Steuersignale
Daten
Mikrocontroller



DO
A
Programm-
speicher
PC
increment
/load
Steuerwerk
Dekoder
Addr
K
Opcode
Ablauf-
steuerung
Akku
PSR
Z C
ALU
Rechenwerk
A
DI
DO
Steuer-
werk
Daten-
speicher
Programmcode
Daten
Steuersignale
Daten
Mikrocontroller



DO
A
Programm-
speicher
PC
increment
/load
Steuerwerk
Dekoder
Addr
K
Opcode
Ablauf-
steuerung
Akku
PSR
Z C
ALU
Rechenwerk
A
DI
DO
Steuer-
werk
Daten-
speicher
Programmcode
Daten
Steuersignale
Daten
Mikrocontroller

Anhang C - Implementierung des Mikrocontrollers in VHDL

Folgende HDL-Implementierung kann vom Dozenten und ggf. von den Studenten für Demonstrationen der internen Abläufe im Mikrocontroller verwendet werden. Der HDL-Text kann hierzu von der PDF-Vorlage in den HDL-Editor eines Simulationsprogramms kopiert werden (z.B. Modelsim). Die Implementierung enthält folgende Komponenten: Package mit Vereinbarungen für alle Komponenten, Programmspeicher mit Testprogramm für den Mikrocontroller, Datenspeicher, Instruktionsregister mit Programmzähler, Rechenwerk (ALU mit PSR und Akku), Steuerwerk mit Zustandsautomat, Testprogramm.

Package mit Vereinbarungen für alle Komponenten

```
--- Package for the DHBW MCT controller (VHDL)

library ieee;
use ieee.std_logic_1164.all;

package MCT_Pack_uP is
-- types representing opcodes
type OPTYPE is (NOP, LSL, LSR, LDI, LDA, STR, ADD,
  SUB, ANDA, EOR, ORA, JMP, BRBC, BRBS, BCLR, ERR);

-- data type and address type
subtype P_Type is std_logic_vector(15 downto 0);
-- 16-bit of programm code
subtype D_Type is std_logic_vector (7 downto 0);
--  8-bit of data
subtype DA_Type is std_logic_vector(9 downto 0);
-- 10-bit addresses of data memory
subtype PA_Type is std_logic_vector(7 downto 0);
--  8-bit addresses of program memory

-- constants
constant P_Width  : integer := 16;
constant D_Width  : integer := 8;
constant DA_Width : integer := 10;
constant PA_Width : integer := 8;

end MCT_Pack_uP;
```

Programmspeicher mit Testprogramm

```
--- Program_Memory for the DHBW MCT controller (VHDL)
```

```
use work.MCT_Pack_uP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity P_ROM is
       port ( Clk, Pipe_EN : in std_logic; -- Clock
              PADDR : in  PA_TYPE; PDOUT : out P_TYPE);
end P_ROM;

architecture RTL of P_ROM is

 -- array of 2**8 samples, each 16 bits wide (reduced for tests)
 type ROM_array is array(0 to 32) of P_TYPE;

       -- instantiate memory object with sample program
       signal pmemory : ROM_array   := (
         0 => "0001100000101000", -- LDI 5
         1 => "0010100000000000", -- STR $0
         2 => "0010100000010100", -- STR $10
         3 => "0001100000011000", -- LDI 3
         4 => "0011000000000000", -- ADD $0
         5 => "0010100000000010", -- STR $1
         others => "0000000000000000");

 begin
       -- read process
       process (Clk, Pipe_EN) begin
         if (Pipe_EN = '1') then
           if (rising_edge(Clk)) then
       PDOUT <= pmemory(to_integer(unsigned(PADDR)));
      end if;
    end if;
  end process;

end RTL;
```

Datenspeicher

```
--- Data Memory for the DHBW MCT controller (VHDL)

use work.MCT_Pack_uP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
```

```
entity D_RAM is
  port ( Clk, DS_EN, RnW : in std_logic;
         -- Clock, D_RAM Enable, ReadNotWrite
        RWADDR : in  DA_TYPE;
        DATIN  : in  D_TYPE; DTOUT  : out D_TYPE);
end D_RAM;

architecture RTL of D_RAM is

 -- array of 32 samples, each 8 bits wide (reduced for tests)
 type RAM_array is array(0 to 32) of D_TYPE;

       -- instantiate memory object of X_RAM
       signal dmemory : RAM_array  := (others => x"00");

 begin
  -- read & write process
  process (Clk) begin
        if (DS_EN = '1') then
            if (rising_edge(Clk)) then
              if (RnW = '0') then      -- write access
                    dmemory(to_integer(unsigned(RWADDR))) <= DATIN;
              elsif (RnW = '1') then   -- read access
                    DTOUT <= dmemory(to_integer(unsigned(RWADDR)));
              end if;
            end if;
        end if;
  end process;

end RTL;
```

Befehlsdekoder mit Programmzähler

```
-- Instruction Decoder for DHBW MCT controller (VHDL)
-- also includes Program Counter (PC)

use work.MCT_Pack_uP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity IR_DEC is
  port ( Clk: in std_logic; RS : in std_logic; DIN : in P_Type;
         PC_EN, Pipe_EN, PC_LOADA, PC_LOADR : in std_logic;
```

```
            A: out DA_Type; -- addresses of operands
            PSR_Bit : out std_logic_vector(4 downto 0);
            -- bits for branch operation
            PC : out PA_Type; OPCODE : out OPTYPE; K8 : out D_Type);
end IR_DEC;

architecture RTL of IR_DEC is
  signal PCINT : signed(8 downto 0); -- internal program counter;
  signal Offset: unsigned(10 downto 0);

  begin
  decoder : process (Clk, RS) begin
    if (RS = '1') then OPCODE <= NOP; PSR_Bit <= "00000";
    elsif (Pipe_EN = '1') then
        if rising_edge(Clk) then
        case DIN(15 downto 11) is
          when "00000" => OPCODE <= NOP;
          when "00001" => OPCODE <= LSL;
          when "00010" => OPCODE <= LSR;
          when "00011" => OPCODE <= LDI;  K8 <= DIN(10 downto 3);
          when "00100" => OPCODE <= LDA;  A  <= DIN(10 downto 1);
          when "00101" => OPCODE <= STR;  A  <= DIN(10 downto 1);
          when "00110" => OPCODE <= ADD;  A  <= DIN(10 downto 1);
          when "00111" => OPCODE <= SUB;  A  <= DIN(10 downto 1);
          when "01000" => OPCODE <= ANDA; A  <= DIN(10 downto 1);
          when "01001" => OPCODE <= EOR;  A  <= DIN(10 downto 1);
          when "01010" => OPCODE <= ORA;  A  <= DIN(10 downto 1);
          when "01011" => OPCODE <= JMP;
                          Offset<=unsigned(DIN(10 downto 0));
          when "01100" => OPCODE <= BRBC; PSR_Bit <= DIN(10 downto 6);
                          Offset<=unsigned(DIN(10 downto 0));
          when "01101" => OPCODE <= BRBS; PSR_Bit <= DIN(10 downto 6);
                          Offset<=unsigned(DIN(10 downto 0));
          when others  => null;
        end case;
      end if;
      end if;
  end process decoder;

  set_internal_PC : process (Clk, RS)
    begin
    if (RS = '1') then PCINT <= (others => '0');
    elsif rising_edge(Clk) then
      if (PC_EN = '1') then
        if (Pipe_EN = '1') then
            PCINT <= PCINT + 1;
```

```
          elsif (PC_LOADA = '1') then
              PCINT <= signed('0' & Offset(7 downto 0));
          elsif (PC_LOADR = '1') then
              if (Offset(5) = '1') then
                PCINT <= PCINT - signed("0000" & Offset(4 downto 0));
              elsif (Offset(5) = '0') then
                PCINT <= PCINT + signed("0000" & Offset(4 downto 0));
              end if;
          end if;
        end if;
      end if;
    end process set_internal_PC;

    PC <= std_logic_vector(PCINT(7 downto 0));

  end RTL;
```

Rechenwerk (ALU mit PSR und Akku)

```
  --- ALU, PSR and Akku for the DHBW MCT controller (VHDL)

  use work.MCT_Pack_uP.all;
  library ieee;
  use ieee.std_logic_1164.all;
  use IEEE.numeric_std.all;

  entity ALU_PSR_Akku is
    port ( OPCODE : in OPTYPE; K, DS_DO : in D_Type;
           Clk, ACC_EN : in std_logic;
           DS_DI: out D_Type; CFlag, ZFlag, NFlag: out std_logic :='0');
  end ALU_PSR_Akku;

  architecture RTL of ALU_PSR_Akku is
  signal Zero : D_Type := (others => '0');
  signal Akku, RdOut : D_Type := (others => '0');

  begin

    process (OPCODE, K, DS_DO, Akku)
      variable RsVar, RdVar, RdoutVar : D_Type;
      variable RVar : signed(D_Width downto 0); -- 9 bits wide
      begin
        RsVar := DS_DO; -- copy Rs from data memory
        RdVar := Akku;  -- copy Rd from Akku
```

```
          RdoutVar := Zero; -- avoids latches in synthesis

          case OPCODE is
             when LSL => CFlag <= RdVar(D_Width-1); -- msb
                RdOutVar(D_Width-1 downto 1) := RdVar(D_Width-2 downto 0);
                         RdOutVar(0):='0';          -- lsb
             when LSR => CFlag <= Akku(0);          -- lsb
                RdOutVar(D_Width-2 downto 0) := Akku(D_Width-1 downto 1);
                         RdOutVar(D_Width-1):='0';   -- msb
           when ADD => RVar := signed('0' & RsVar) +  signed('0' & RdVar);
                  RdOutVar := std_logic_vector(RVar(D_Width-1 downto 0));
                  NFlag <= RVar(D_Width);
           when SUB => RVar := signed('0' & RsVar) -  signed('0' & RdVar);
                  RdOutVar := std_logic_vector(RVar(D_Width-1 downto 0));
                  NFlag <= RVar(D_Width);
           when ANDA=> RdOutVar := RsVar AND RdVar;
           when EOR => RdOutVar := RsVar XOR RdVar;
           when ORA => RdOutVar := RsVar OR RdVar;
           when STR => DS_DI <= Akku;
           when LDA => RdOutVar := DS_DO;
           when LDI => RdOutVar := K;
           when  others => null; -- no action
        end case;
        if (RdOutVar = Zero) then ZFlag <='1'; end if;
        RdOut <= RdOutVar;
      end process;

      load_akku : process (Clk) begin
       if (ACC_EN = '1') then
          if rising_edge(Clk) then
              Akku <= RdOut;
          end if;
        end if;
      end process load_akku;

    end RTL;
```

Steuerwerk mit Zustandsautomat

```
    --- FSM and Top Module for the DHBW MCT controller (VHDL)

    use work.MCT_Pack_uP.all;
    library ieee;
    use ieee.std_logic_1164.all;
    use IEEE.numeric_std.all;
```

```
entity FSM_Top_Module is
        port ( Clk, RS: in std_logic);
end FSM_Top_Module;

architecture RTL of FSM_Top_Module  is

component P_ROM is
       port ( Clk, Pipe_EN  : in std_logic; -- Clock, Pipeline enable
              PADDR : in  PA_Type;
              PDOUT : out P_Type);
end component;

component D_RAM is
       port ( Clk, DS_EN, RnW  : in std_logic;
              -- Clock, D_RAM enable, Read not Write
              RWADDR : in  DA_Type;
              DATIN : in D_Type;
              DTOUT : out D_Type);
end component;

component IR_DEC is
       port ( Clk: in std_logic; RS : in std_logic; DIN : in P_Type;
              PC_EN, Pipe_EN, PC_LOADA, PC_LOADR : in std_logic;
              A : out DA_Type; -- addresses of operands
              PSR_Bit : out std_logic_vector(4 downto 0);
              PC : out PA_Type; OPCODE : out OPTYPE; K8 : out D_Type);
end component;

component ALU_PSR_Akku is
       port (OPCODE : in OPTYPE; K, DS_DO : in D_Type;
             Clk, ACC_EN : in std_logic;
             DS_DI  : out D_Type; CFlag, ZFlag, NFlag: out std_logic);
end component;


-- Top_Module internal signals

type state_type is (Z1, Z2);
-- Z1: operation executes in one clock cycle,
-- Z2: operation executes in two clock cycles

signal next_state, current_state: state_type :=Z1;

-- to interconnect modules incl. controls
signal T_OPCODE : OPTYPE;
```

```
    signal T_PC : PA_TYPE;
    signal T_A : DA_TYPE;
    signal T_DIN : P_TYPE;
    signal T_DI, T_DO, T_K8 : D_Type;
    signal T_Zero, T_Carry, T_Neg : std_logic;
    signal T_PSR : std_logic_vector (4 downto 0);
    signal T_Pipe_EN, T_PC_EN, T_DS_RnW : std_logic :='1';
    signal T_DS_EN, T_PC_LOADA, T_PC_LOADR, T_ACC_EN : std_logic :='0';

    begin

    -- instantiate and connect modules to Top Module
    PMem: P_ROM port map (Clk => Clk, Pipe_EN=> T_Pipe_EN,
                          PADDR => T_PC, PDOUT => T_DIN);

    DMem: D_RAM port map (Clk => Clk, DS_EN=> T_DS_EN, RnW => T_DS_RnW,
                          RWADDR => T_A, DATIN => T_DI, DTOUT => T_DO);

    IR_Decoder: IR_DEC port map (Clk => Clk, RS => RS, DIN => T_DIN,
                Pipe_EN => T_Pipe_EN, PC_EN => T_PC_EN,
                PC_LOADA => T_PC_LOADA, PC_LOADR => T_PC_LOADR,
                A => T_A, PSR_Bit => T_PSR, PC => T_PC,
                OPCODE => T_OPCODE, K8 => T_K8);

    Alu : ALU_PSR_Akku port map (OPCODE => T_OPCODE, K => T_K8,
                DS_DO => T_DO, Clk => Clk, ACC_EN => T_ACC_EN,
                DS_DI => T_DI, ZFlag => T_Zero, CFlag => T_Carry,
                NFlag => T_Neg);

    -- run processes
    update_state_register: process(Clk, RS)
      begin
           if (RS='1') then
             current_state <= Z1;
           elsif rising_edge(Clk) then
             current_state <= next_state;
           end if;

    end process update_state_register;

    -- state transitions & actions (single logic)
    logic_next_state_and_actions: process(current_state, T_OPCODE)

      begin

        case current_state is
```

```
        when Z1 => -- one clock cycle per operation

            case T_OPCODE is
               when NOP => T_PC_EN<='1'; T_Pipe_EN<='1'; T_DS_EN<='0';
                           T_PC_LOADA<='0'; T_PC_LOADR<='0';
                           T_ACC_EN<='0';

               when LSL | LSR | LDI => T_Pipe_EN<='1'; T_PC_EN<='1';
                                      T_ACC_EN<='1'; T_DS_EN<='0';

               when STR => T_Pipe_EN<='1'; T_PC_EN<='1';
                           T_DS_EN<='1'; T_DS_RnW<='0'; T_ACC_EN<='0';

               when ADD | SUB | ANDA | EOR | ORA | LDA  =>
                            T_Pipe_EN<='0'; T_ACC_EN<='0';
                            T_DS_EN<='1'; T_DS_RnW<='1';
                            next_state<=Z2;  -- 2nd cycle

               when JMP =>  T_Pipe_EN<='0'; T_PC_EN<='1'; T_PC_LOADA<='1';
                            next_state<=Z2;  -- 2nd cycle

               when BRBC =>
                    case T_PSR is
                    when "10000" =>
                         if (T_Zero = '0')  then
                             T_Pipe_EN<='0'; T_PC_LOADR<='1';
                              next_state<=Z2;
                         elsif (T_Zero ='1') then
                             T_Pipe_EN<='1'; T_PC_LOADR<='0';
                         end if;

                    when "01000" =>
                         if (T_Carry = '0') then
                             T_Pipe_EN<='0'; T_PC_LOADR<='1';
                             next_state<=Z2;
                         elsif (T_Zero ='1') then
                             T_Pipe_EN<='1'; T_PC_LOADR<='0';
                         end if;

                    when "00001" =>
                         if (T_Neg = '0') then
                             T_Pipe_EN<='0'; T_PC_LOADR<='1';
                             next_state<=Z2;
                         elsif (T_Neg ='1') then
                             T_Pipe_EN<='1'; T_PC_LOADR<='0';
```

```
                    end if;
                 when others => null;
                 end case;

            when BRBS =>
                 case T_PSR is
                 when "10000" =>
                      if (T_Zero = '1')  then
                          T_Pipe_EN<='0'; T_PC_LOADR<='1';
                          next_state<=Z2;
                      elsif (T_Zero ='0') then
                          T_Pipe_EN<='1'; T_PC_LOADR<='0';
                      end if;

                 when "01000" =>
                      if (T_Carry = '1') then
                          T_Pipe_EN<='0'; T_PC_LOADR<='1';
                          next_state<=Z2;
                      elsif (T_Zero ='0') then
                          T_Pipe_EN<='1'; T_PC_LOADR<='0';
                      end if;

                 when "00001" =>
                      if (T_Neg = '1') then
                          T_Pipe_EN<='0'; T_PC_LOADR<='1';
                          next_state<=Z2;
                      elsif (T_Neg ='0') then
                          T_Pipe_EN<='1'; T_PC_LOADR<='0';
                      end if;

                  when others => null;
                  end case;

             when others  => null;
           end case; -- opcode one cycle

        when Z2 => -- second clock cycle per operation

          case T_OPCODE is

            when BRBS | BRBC| JMP => -- re-animate pipeline
                 T_PC_EN<='1'; T_Pipe_EN<='1';
                 T_PC_LOADR<='0'; T_PC_LOADR<='0';
                 next_state<=Z1;

            when ADD | SUB | ANDA | EOR | ORA | LDA  =>
```

```
                    T_ACC_EN<='1'; T_DS_EN<='0'; -- enable akku for result
                    T_Pipe_EN<='1';     -- re-animate pipeline
                    next_state<=Z1;

              when others  => null;
            end case; -- opcode two cycles

        when others => null;
      end case; -- states

  end process logic_next_state_and_actions;

  end RTL;
```

Test Program

```
--- Testbench for FSM and Top Module of DHBW MCT controller (VHDL)

use work.MCT_Pack_uP.all;
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

entity test_SP is
end test_SP ;

architecture Behavioural of test_SP  is

component FSM_Top_Module is
   port(Clk, RS : in std_logic);
end component;

-- Test bench internal signals
signal T_Clk, T_RS : std_logic :='0';

begin
-- connect FSM_Top_Module to testbench
PCT1: FSM_Top_Module port map (Clk=>T_Clk, RS=>T_RS);

-- run tests
reset : process begin
   T_RS <= '1'; wait for 5 ns; T_RS <= '0'; wait;
end process reset;

clock : process begin
```

```
      T_Clk <= '0'; wait for 10 ns; T_Clk <= '1'; wait for 10 ns;
   end process clock;

   end Behavioural;
```