Einführung in

Computer Microsystems

Sommersemester 2010

Wolfgang Heenes

TECHNISCHE
UNIVERSITÄT
DARMSTADT

8. Aufgabenblatt mit Lösungsvorschlag

09.06.2010

Aufgabe 1: Realisierung des Modellrechners WKP

Der in der Vorlesung vorgestellt Modellrechner soll jetzt vollständig in Verilog HDL beschrieben, synthetisiert und simuliert werden.

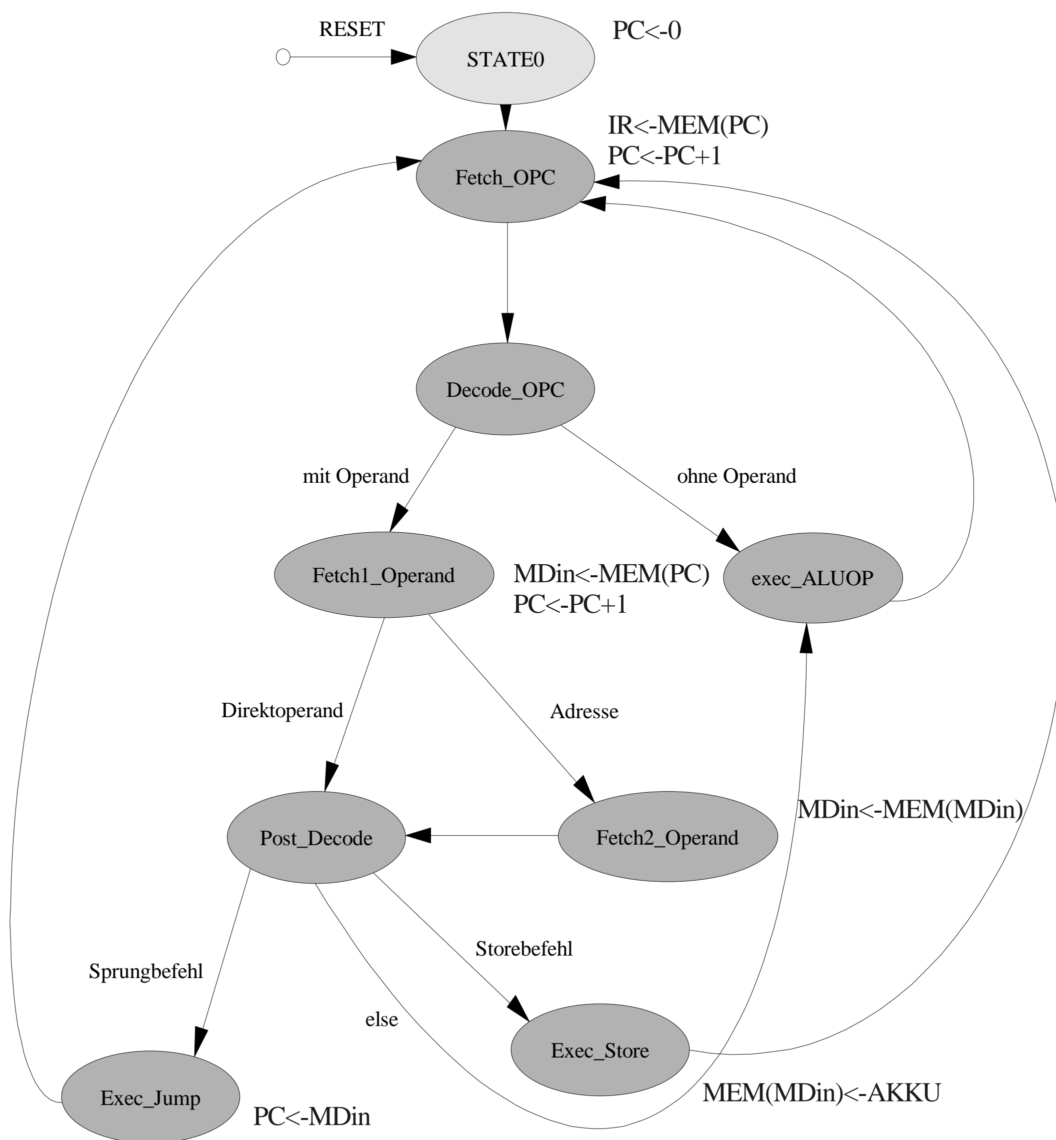
Die folgende Abbildung zeigt die Architektur/Struktur des Modellrechners.

Die wesentlichen Komponenten sind:

- *Speicher (Generierung über IP-Core Generator (vgl. Vorlesung 8))*
- *ALU (Realisierung als Schaltnetz)*
- *Steuerwerk*

Die Register (PC, IR, AKKU) sind 8 Bit breit. Der Speicher wird mit einer 8 Bit breiten Adresse angesprochen. Der Dateneingang/Datenausgang ist jeweils 8 Bit breit. Die Beschreibung der Befehle sind auf dem Hilfsblatt zum Prozessor (`http://www.ra.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_RA/cms/hilfsblatt_modellprozessor.pdf`*) zu finden.*

Der Zustandsgraph des Steuerwerks ist in folgender Abbildung zu sehen.



a) *Beschreiben Sie den Modellrechner in Verilog HDL.*

Lösungsvorschlag:

Der Prozessor wird gemäß der Spezifikation implementiert. Im *Top-Level-Module* `prozessor` werden die Register `PC`, `IR`, `AKKU`, `MD_IN`, `D_OUT` und `STATUS` definiert, die Module für den Speicher (`MEM`), das Operationswerk (`ALU`) und das Steuerwerk (`STW`) eingebunden und geeignete Verbindungen geschaffen. Der Verilog-Code ist im folgenden Listing zu sehen:

```
module prozessor(clkin, reset, D_OUT);

input clkin, reset;
output [7:0] D_OUT;

wire clk;           // Takt
divider ownDivider (clkin, clk);    // Taktteiler, da die Frequenz von clkin zu hoch für die menschliche Wahrnehmung ist.

reg  [7:0] IR;      // Instruction Register
reg  [7:0] PC;      // Program Counter
reg  [7:0] MD_IN;   // MD_IN - Register
reg  [7:0] D_OUT;   // Ausgaberegister
reg  [7:0] AKKU;    // Akkumulator
reg  [3:0] STATUS;  // Statusbits

wire [7:0] douta;   // Ausgang des Speichers
wire selpc, ibit;
wire [3:0] alu_op;  // ALU-Operation
wire [7:0] madr;    // Speicheradresse
wire [3:0] status_out;  // Statusbits von der ALU
wire [7:0] alu_out;     // Ergebnis der ALU
wire [3:0] opc;         // Befehlscode

//  Ports: (clka, dina, addra, ena, wea, douta)
MEM ownMEM (clk,  AKKU, madr,  en,  rw,  douta);

//  Ports: (clk, reset, opc, ibit, STATUS, rw, en, ldir, ldmdin, selpc, clrpc, incpc, ldpc, lddout, lda, ldstat, alu_op)
STW ownSTW (clk, reset, opc, ibit, STATUS, rw, en, ldir, ldmdin, selpc, clrpc, incpc, ldpc, lddout, lda, ldstat, alu_op);

//  Ports: (inA,  inB,   alu_op, status_in, result,  status)
ALU ownALU (AKKU, MD_IN, alu_op, STATUS,    alu_out, status_out);

assign madr = selpc ? MD_IN : PC; // Multiplexer Speicher

assign ibit = IR[0];
assign opc  = IR [7:4];

always @(posedge clk) begin
    if (reset) begin
        IR     <= 8'b0;
        PC     <= 8'b0;
        MD_IN  <= 8'b0;
        D_OUT  <= 8'b0;
        AKKU   <= 8'b0;
        STATUS <= 3'b0;
    end else begin
        if (ldir)   IR     <= douta;
        if (clrpc)  PC     <= 0;
        if (incpc)  PC     <= PC+1;
        if (ldpc)   PC     <= MD_IN;
        if (ldmdin) MD_IN   <= douta;
        if (lddout) D_OUT  <= AKKU;
        if (lda)    AKKU   <= alu_out;
        if (ldstat) STATUS <= status_out;
    end
end
endmodule
```

Das Modul für den Speicher (`MEM`) wird vom *CORE Generator* erstellt und hier nicht aufgeführt. Das Modul kann wie in obigem Listing zu sehen eingebunden werden.

Das Operationswerk (ALU) besteht aus kombinatorischer Logik, ohne Register oder Latches (vgl. Übung 7). Hier werden die Werte für den AKKU und die STATUS-Bits berechnet. Wie es realisiert werden kann zeigt folgendes Listing.

```
module ALU(inA, inB, alu_op, status_in, result, status);

parameter
opc_load    = 4'b0000,
opc_not     = 4'b0001,
opc_and     = 4'b0010,
opc_or      = 4'b0011,
opc_add     = 4'b0100,
opc_lshr    = 4'b0101,
opc_ashr    = 4'b0110,
opc_shl     = 4'b0111,
opc_clrc    = 4'b1000,
opc_setc    = 4'b1001;

input  [7:0] inA, inB;
input  [3:0] alu_op;
input  [3:0] status_in;
output [3:0] status; // Statusbits: OV C N Z
output reg [7:0] result;

reg OV, C, N, Z;
assign status = {OV, C, N, Z};

// ALU als kombinatorischer always-Block
// Operation ausführen und Statusbits berechnen
always @(*) begin
    case (alu_op)
        opc_load : begin // AKKU laden
                    result = inB;
                    Z = result == 8'b0;
                    N = result[7];
                    {OV, C} = 2'b0;
                   end
        opc_not  : begin // Bitweises invertieren
                    result =  ~(inA);
                    Z = result == 8'b0;
                    N = result[7];
                    {OV, C} = 2'b0;
                   end
        opc_and  : begin // Bitweise UND
                    result = inA & inB;
                    Z = result == 8'b0;
                    N = result[7];
                    {OV, C} = 2'b0;
                   end
        opc_or   : begin // Bitweise ODER
                    result = inA | inB;
                    Z = result == 8'b0;
                    N = result[7];
                    {OV, C} = 2'b0;
                   end
        opc_add  : begin // 2K-addieren mit Carry-In
                    Z = result == 8'b0;
                    N = result[7];
                   {C, result} = inA + inB + status_in[2];      // Addition und Carry-Flag
                    OV = (inA[7] ~^ inB[7]) && (result[7] ^ C); // Overflow-Flag
                   end
        opc_lshr : begin // Rechts shiften
                    result = {1'b0, inA[7:1]};
                    Z = result == 8'b0;
                    N = result[7];
                    C = inA[0];
                    OV = 1'b0;
                   end
        opc_ashr : begin // Arithmetisch (vorzeichenerhaltend) rechts shiften
                    result = {inA[7], inA[7:1]};
                    Z = result == 8'b0;
```

```
                    N = result[7];
                    C = inA[0];
                    OV = 1'b0;
                   end
        opc_shl  : begin // Links shiften
                    result = {inA[6:0], 1'b0};
                    Z = result == 8'b0;
                    N = result[7];
                    C = inA[7];
                    OV = 1'b0;
                   end
        opc_clrc : begin // C löschen, andere Statusbits und AKKU unverändert lassen
                    result = inA;
                    Z = status_in[0];
                    N = status_in[1];
                    C = 1'b0;
                    OV = status_in[3];
                   end
        opc_setc : begin // C setzen, andere Statusbits und AKKU unverändert lassen
                    result = inA;
                    Z = status_in[0];
                    N = status_in[1];
                    C = 1'b1;
                    OV = status_in[3];
                   end
        default  : begin // Ungültige/Keine Operation
                    result  = inA;
                    {OV, C, N, Z} = status_in;
                   end
   endcase
end
endmodule
```

Das Steuerwerk (STW) wird als Automat beschrieben, bei dem kombinatorische Logik und Speicherelemente getrennt sind (Übersichtlichkeit und Erweiterbarkeit). Es generiert die Steuersignale der Register und des Speichers als einzelne Signale und hat entsprechend viele Ports. Einige der im Zustandsdiagramm dargestellten Zustände werden an das Verhalten des Speichers angepasst und in zwei Zustände aufgeteilt, so dass das Steuerwerk 13 Zustände hat.

```
module STW(clk, reset, opc, ibit, status, rw, en, ldir, ldmdin, selpc, clrpc, incpc, ldpc,
           lddout, lda, ldstat, alu_op);

parameter // Zustände
STATE0          = 4'b0000,
Fetch_OPC1      = 4'b0001,
Fetch_OPC2      = 4'b0010,
Decode_OPC      = 4'b0011,
Fetch1_Operand1 = 4'b0100,
Fetch1_Operand2 = 4'b0101,
Fetch2_Operand1 = 4'b0110,
Fetch2_Operand2 = 4'b0111,
Post_Decode     = 4'b1000,
Exec_Jump       = 4'b1001,
Exec_Store1     = 4'b1010,
Exec_ALUOP      = 4'b1011,
DOUT            = 4'b1100;

parameter // Befehle
LDA             = 4'b0000,
NOT             = 4'b0001,
AND             = 4'b0010,
OR              = 4'b0011,
ADD             = 4'b0100,
SHR             = 4'b0101,
ASR             = 4'b0110,
SHL             = 4'b0111,
CLRC            = 4'b1000,
SETC            = 4'b1001,
NOP             = 4'b1010,
BRA_L           = 4'b1011,
```

```
BRZ_L          = 4'b1100,
STA_N          = 4'b1101,
OUT            = 4'b1110;

input clk, reset, ibit;
input [3:0] opc;
input [3:0] status; // Statusbits: OV, C, N, Z
output rw, en, ldir, ldmdin, selpc, clrpc, incpc, ldpc, lddout, lda, ldstat;
output [3:0] alu_op;

reg [3:0] STATE, nextstate; // Zustandsregister

//Ausgangssignale generieren

assign clrpc  = (STATE == STATE0);
assign ldir   = (STATE == Fetch_OPC2);

assign incpc  = (STATE == Fetch_OPC2) || (STATE == Fetch1_Operand2);
assign en     = (STATE == STATE0) || (STATE ==Exec_ALUOP) || (STATE == Fetch_OPC1)
                ||(STATE == Fetch_OPC2) || (STATE == Fetch1_Operand1) || (STATE == Fetch2_Operand1)
                ||(STATE == Exec_Store1);
assign rw     = (STATE == Exec_Store1);
assign ldmdin = (STATE == Fetch1_Operand2) || (STATE == Fetch2_Operand2);
assign alu_op = opc;

assign lda   = (STATE == Exec_ALUOP);
assign ldstat = (STATE == Exec_ALUOP);

// Sprungbefehl realisieren
assign ldpc   = (STATE == Exec_Jump);
assign lddout = (STATE == DOUT);
assign selpc  = (STATE == Fetch2_Operand1) || (STATE == Fetch2_Operand2)
                 || (STATE == Exec_Store1);

always @(STATE, status, opc, ibit) begin
    case (STATE)
        STATE0 :
            begin
                nextstate = Fetch_OPC1;
            end
        Fetch_OPC1 :
            begin
                nextstate = Fetch_OPC2;
            end
        Fetch_OPC2 :
            begin
                nextstate = Decode_OPC;
            end
        Decode_OPC :
            begin
                if ((opc == NOT) || (opc == SHR) || (opc == ASR) || (opc == SHL) || (opc == CLRC) || (opc == SETC))
                    nextstate = Exec_ALUOP;
                else if ((opc == OUT)) nextstate = DOUT;
                else
                    nextstate = Fetch1_Operand1;
            end

        Fetch1_Operand1 :
            begin
             nextstate = Fetch1_Operand2;
            end
        Fetch1_Operand2 :
            begin
                if (ibit) //I-Bit gesetzt?
                    begin
                        nextstate = Fetch2_Operand1;
                    end
                else
                    nextstate = Post_Decode;
            end
```

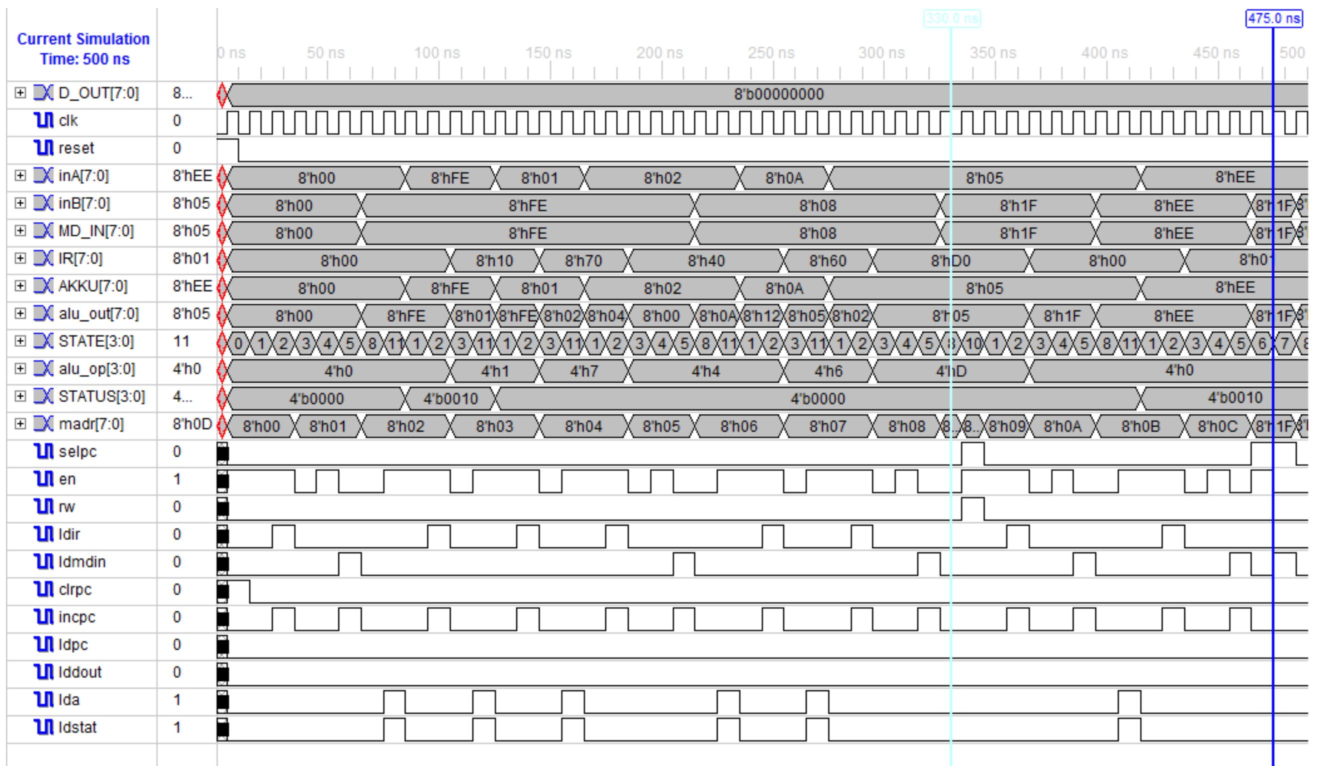
```
            Fetch2_Operand1 :
                begin
                    nextstate = Fetch2_Operand2;
                end
            Fetch2_Operand2 :
                begin
                    nextstate = Post_Decode;
                end
            Post_Decode :
                begin
                    if ((opc == BRA_L) || ((opc == BRZ_L) && status[0]))
                        begin
                            nextstate = Exec_Jump;
                        end
                    else if (opc == STA_N)
                        begin
                            nextstate = Exec_Store1;
                        end
                    else if ((opc == LDA) || (opc == AND) || (opc== OR) || (opc == ADD))
                    begin
                        nextstate = Exec_ALUOP;
                    end
                    else nextstate = Fetch_OPC1;
                end
            Exec_Jump :
                begin
                    nextstate = Fetch_OPC1;
                end
            Exec_Store1:
               begin
                    nextstate = Fetch_OPC1;
                end
            Exec_ALUOP :
                begin
                    nextstate = Fetch_OPC1;
                end
            DOUT :
                begin
                    nextstate = Fetch_OPC1;
                end
            default: nextstate = STATE0;
        endcase

    end //always

    always @(posedge clk) begin
        if (reset)
            STATE <= STATE0;
        else
            STATE <= nextstate;
    end //always
    endmodule
```

Ein kleines Programm testet mehrere Befehle, wie z. B. das Rückschreiben in den Speicher.

```
00 fe 10 70 40 08 60 d0 1f 00 ee 01 1f e0;
```

Nach dem Laden der Zahl `fe` wird der Akku negiert. Damit steht im LSB eine 1. Die folgenden Befehle (shl - OPC: 70, add - OPC: 40, asr - OPC: 60) testen die ALU. Danach sollte der Wert 5 im AKKU stehen. Dieser wird an Stelle `1f` im Speicher abgelegt (d0 1f). Nach dem Laden des AKKUs mit `ee` wird die Speicherstelle `1f` in den AKKU geladen. Das Ergebnis 5 wird in das Ausgaberegister geschrieben. Die Ausgabe ist allerdings nicht mehr im Waveform dargestellt.

Die folgende Abbildung zeigt das Waveform. Bei Betrachtung des Ausgangs der ALU stellt man fest, dass „ungültige“ Zwischenergebnisse auftreten können. Die Steuerung der Register sorgt dafür, dass diese Werte nicht übernommen werden.



Da der im FPGA erzeugte Takt so schnell ist, dass ein Mensch Änderungen am Ausgang des Prozessors nur schwer verfolgen kann, wird für die Realisierung auf einem FPGA im *Top-Level-Module* ein Taktteiler (`divider`) eingebaut, der den Takt auf $\frac{1}{2^{18}}$ reduziert.

```
module divider(clkin, clkout);
input clkin;
output clkout;
reg [17:0] count;
assign clkout = count[17];
initial count = 0;
always @(posedge clkin) begin
    count <= count + 1;
end
endmodule
```

b) *Analysieren Sie die Ergebnisse der Synthese. Wieviele CLBs (LUTs und Register) benötigt Ihre Implementierung. Mit welcher Taktfrequenz können Sie Ihre Implementierung maximal betreiben.*

Lösungsvorschlag:

Die Synthese des Prozessors mit Xilinx ISE 12.1 ergibt folgende Ergebnisse:

- LUTs: 163
  - als Logik: 144
  - zum Durchleiten: 19
- Register (Flip-Flops): 72
- Taktfrequenz: ≈ 136 MHz

Wenn als Design-Constraints eine Optimierung für die Fläche gewählt wird, ergeben sich folgende Werte:

- LUTs: 146

- als Logik: 125
  - zum Durchleiten: 21
- Register (Flip-Flops): 62
- Taktfrequenz: ≈ 103 MHz

Aufgabe 2: Test für den Modellprozessor WKP

Nach erfolgreicher Implementierung des Modellprozessors soll folgendes Testprogramm abgearbeitet werden.

```
0  : 00;
1  : 01;
2  : e0;
3  : 70;
4  : e0;
5  : c0;
6  : 09;
7  : b0;
8  : 03;
9  : 00;
a  : 80;
b  : e0;
c  : 50;
d  : e0;
e  : c0;
f  : 00;
10 : b0;
11 : 0c;
```

a) *Analysieren und kommentieren Sie das Programm.*

Die MIF-Datei, ein Simulator und ein Assembler stehen auf der Webseite zur Veranstaltung zum Download bereit.

Lösungsvorschlag:

Die Bedeutung der Befehle des Testprogramms sind in folgender Tabelle aufgeführt:

Adr.	:	Inhalt	Befehl	Wirkung
00	:	00	LDA 1	Lade Akku mit Wert
01	:	01		Wert = 1
02	:	e0	OUT	Gib Inhalt des Akkus aus
03	:	70	SHL	Shift nach links
04	:	e0	OUT	Gib Inhalt des Akkus aus
05	:	c0	BRZ 9	Sprung nach Label, falls Z-Bit gesetzt
06	:	09		Label = 9
07	:	b0	BRA 3	Sprung nach Label
08	:	03		Label = 3
09	:	00	LDA 128	Lade Akku mit Wert
0a	:	80		Wert = 128
0b	:	e0	OUT	Gib Inhalt des Akkus aus
0c	:	50	SHR	Shift nach rechts (mit 0 auffüllen)
0d	:	e0	OUT	Gib Inhalt des Akkus aus
0e	:	c0	BRZ 0	Sprung nach Label, falls Z-Bit gesetzt
0f	:	00		Lable = 0 → Anfang des Programms
10	:	b0	BRA 12	Sprung nach Label
11	:	0c		Label = 12 (Achtung: 12d = 0Ch)

Das Programm schiebt eine 1 vom LSB zum MSB und wieder zum LSB zurück und beginnt dann von neuem.

Bei der Generierung eines Speichers mit dem *CORE Generator* kann eine .coe-Datei zur Initialisierung des Speichers verwendet werden. Für dieses Programm kann die Datei wie folgt aussehen:

```
; Initialisierungsdatei für das Shift-Programm
; 8-Bit Breite, 17 Tiefe
memory_initialization_radix = 16;
memory_initialization_vector =
00 01 e0 70 e0 c0 09 b0 03 00 80 e0 50 e0 c0 00 b0 0c;
```

Aufgabe 3: Test des Modellprozessors auf einem FPGA

Testen Sie Ihre Implementierung mit dem Testprogramm aus Aufgabe 2 auf einem FPGA. Dazu stehen Ihnen die beiden Virtual FPGA Lab Server zur Verfügung.

Für die Ausführung auf dem FPGA ist ein User-Constraints-File nötig. Für den Spartan3E XC3S500E FPGA im Package FG320 auf einem SPARTAN-3E-Board sieht diese Datei wie folgt aus. Hierbei wird der Port `D_OUT` *(entspricht dem Ausgaberegister) auf die acht LEDs gelegt, so dass man das ausgegebene Byte betrachten kann.*

```
NET "D_OUT<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "D_OUT<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "D_OUT<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "D_OUT<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "D_OUT<4>" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "D_OUT<5>" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "D_OUT<6>" LOC = "E9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;
NET "D_OUT<7>" LOC = "F9"  | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8;

NET "clkin" LOC = "C9" | IOSTANDARD = LVCMOS33;

NET "reset" LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
```

Diese Datei steht auf der Webseite zur Veranstaltung zum Download zur Verfügung.

Aufgabe 4: Zusatzaufgabe

Implementieren Sie das Steuerwerk des Modellprozessors als Mikroprogrammsteuerwerk und testen Sie Ihre Implementierung mit dem Testprogramm aus Aufgabe 2.