Digitaltechnik
Wintersemester 2021/2022
13. Übung

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Dr.-Ing. Thomas Schneider, M.Sc. Daniel Günther, M.Sc. Amos Treiber

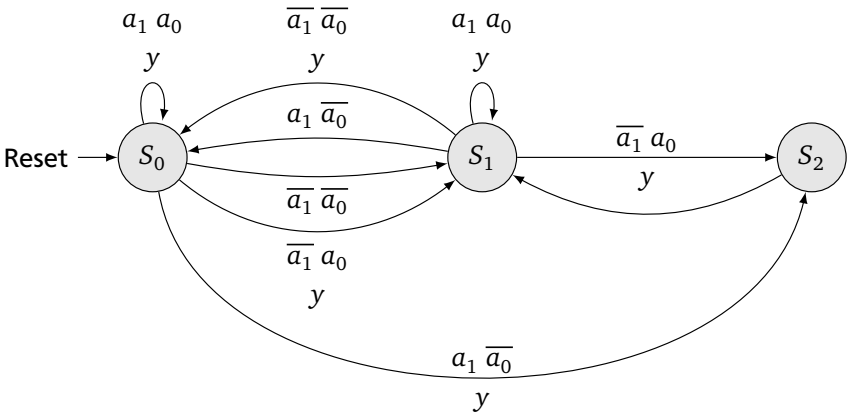
LÖSUNGSVORSCHLAG **KW06**

Bitte bearbeiten Sie die Übungsblätter bereits im Voraus, sodass Sie Ihre Lösungen zusammen mit Ihren Kommilitonen und Tutoren während der wöchentlichen Übungsstunde diskutieren können.
Mit der angegebenen Bearbeitungszeit für die einzelnen Aufgaben können Sie Ihren Leistungsstand besser einschätzen. Die mit **"Zusatzaufgabe"** gekennzeichneten Aufgaben sind zur zusätzlichen Vertiefung für interessierte Studierende gedacht und daher nicht im Zeitumfang von 90 Minuten einkalkuliert.

Übung 13.1 Robuste Endliche Automaten [20 min]

Implementieren Sie folgende endliche Automaten in SystemVerilog. Wenn eines der Eingangsbits `1'bz` oder `1'bx` ist, soll der Automat in den Startzustand wechseln und dabei kein Ausgangsbit auf 1 setzen. Verwenden Sie den `===` Operator zum Vergleich zwischen Ausdrücken vierwertiger Logik.
Die ungültigen Eingaben kann man direkt im Zustandsregister abfangen und muss somit diese Fälle nicht mehr bei der Bestimmung des nächsten Zustandes beachten. Da `~1'bz===~1'bx===1'bx` gilt (siehe Resolutionstabellen für mehrwertige Logik in Vorlesung 6), genügt dafür ein Vergleich pro Eingabebit.
Für Mealy-Automaten muss der `===` Operator auch in der Ausgabelogik verwendet werden.

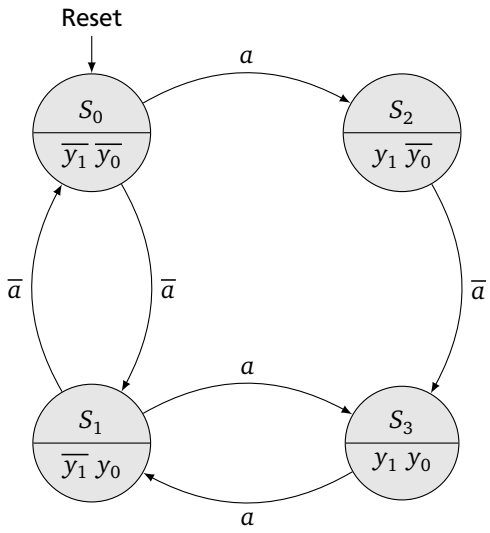
a)



fsm/robust/mealy.sv

```
module mealy(input logic CLK, RST, input logic [1:0] A, output logic Y);
  logic [1:0] state, nextstate;

  always_ff @(posedge CLK)
    state <= RST || ~A[0]===1'bx || ~A[1]===1'bx ? 0 : nextstate;

  always_comb case (state)
          0: nextstate = ~A[1] ? 1 : ~A[0] ? 2 : 0;
          1: nextstate = ~A[0] ? 0 : ~A[1] ? 2 : 1;
    default: nextstate = 1;
  endcase

  assign Y = (state==0 && (A===2'b11 || A===2'b01 || A===2'b10))
           || (state==1 && (A===2'b11 || A===2'b01 || A===2'b00));
endmodule
```

b)

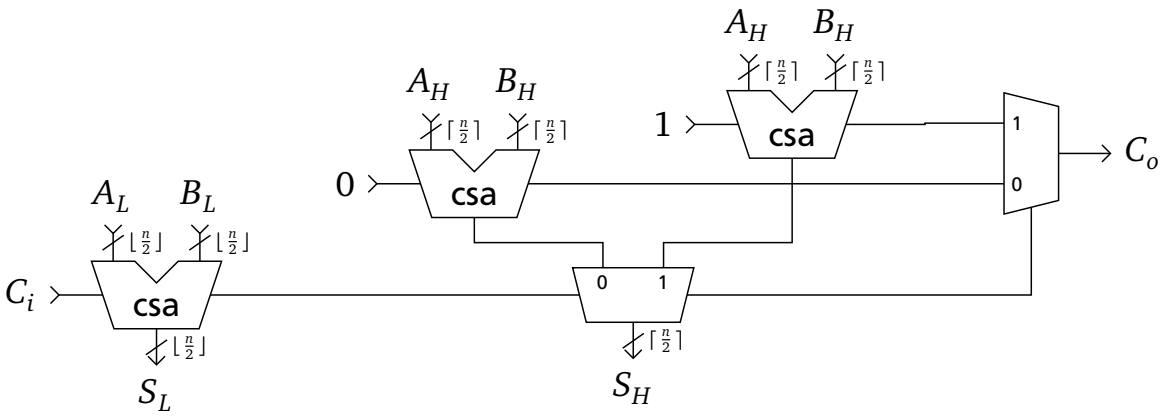


fsm/robust/moore.sv

```
1  module moore(input  logic CLK, RST, A,
2               output logic [1:0] Y);
3
4    logic [1:0] state, nextstate;
5
6    always_ff @(posedge CLK)
7      state <= RST || ~A===1'bx ? 0 : nextstate;
8
9    always_comb case (state)
10           0: nextstate = A ? 2 : 1;
11           1: nextstate = A ? 3 : 0;
12           2: nextstate = A ? 2 : 3;
13     default: nextstate = A ? 1 : 3;
14   endcase
15
16   assign Y = state;
17
18 endmodule
```

Übung 13.2 Conditional Sum Adder (CSA) [25 min]

Ein Nachteil des Ripple-Carry-Adders ist dessen lineare Übertragskette vom LSB bis zum MSB, wodurch der kritische Pfad linear mit der Bitbreite wächst. Ein n-Bit CSA bricht diese Übertragskette auf, indem für die oberen $\lceil \frac{n}{2} \rceil$ Eingabebits sowohl die einfache Summe ($A_H + B_H$), als auch dessen Inkrement ($A_H + B_H + 1$) *gleichzeitig* berechnet werden. Sobald der Übertrag des unteren Halbworts ($A_L + B_L + C_i$) verfügbar ist, muss nur noch das korrekte Ergebnis (Summe und Übertrag) aus den beiden Berechnungen für das obere Halbwort ausgewählt werden.



Übung 13.2.1 Rekursive Implementierung

Implementieren Sie den CSA in SystemVerilog als rekursives Modul mit Übertragsein- und ausgang:

arith/adder/csa.sv

```
2  module csa #(parameter      WIDTH=4)
3              (input logic  [WIDTH-1:0] A, B, input  logic CI,
4               output logic [WIDTH-1:0] S,    output logic CO);
```

Ein 1 bit CSA entspricht einem Volladdierer. Beachten Sie, dass `WIDTH` nicht immer ohne Rest durch zwei teilbar ist. Verwenden Sie die Module für Halb- und den Volladdierer (aus Übung 10), die in Moodle unter `SystemVerilog/src/arith/adder` zur Verfügung stehen. Die Verzögerungszeit der Multiplexer soll 4 ns betragen.

```
  generate
    if (WIDTH > 1) begin        // Rekursion fortsetzen
      localparam WL = WIDTH/2;  // floor(WIDTH/2)
      localparam WH = WIDTH-WL; // ceil (WIDTH/2)
      logic [WH-1:0] sh0,sh1;
      logic      cl, ch0,ch1;
      csa #(WL) csal  (A[ 0 +: WL], B[ 0 +: WL], CI,   S[0 +: WL], cl);
      csa #(WH) csah0 (A[WL +: WH], B[WL +: WH], 1'b0, sh0,        ch0);
      csa #(WH) csah1 (A[WL +: WH], B[WL +: WH], 1'b1, sh1,        ch1);
      assign #4 {CO,S[WL +: WH]} = cl ? {ch1,sh1} : {ch0,sh0}; // verzögerter MUX
    end else Volladdierer fa (CI, A, B, CO, S); // Rekursionsende
  endgenerate
endmodule
```

Zum vereinfachten Zugriff auf Teile eines Vektors bietet es sich an, eine abgekürzte Schreibweise zu verwenden (wie z.B. bei `A[WL +: WH]`). Der erste Teil gibt dabei den Startindex bzw. das Offset an und der zweite Teil die gewünschte Breite. Der Ausdruck `A[WL +: WH]` ist somit äquivalent zu `A[WL+WH-1 : WL]`.

Übung 13.2.2 Modul-Kapselung

Verpacken Sie den CSA in ein Modul mit der folgender allgemeiner Addierer-Schnittstelle:

```
module add #(parameter    WIDTH=4)
            (input logic [WIDTH-1:0] A, B, output logic [WIDTH:0] S);

  csa #(WIDTH) inst (A, B, 1'b0, S[WIDTH-1:0], S[WIDTH]);
endmodule
```

Übung 13.2.3 Verifikation

Schreiben Sie eine Testbench, die den CSA mit einer per **`localparam`** konfigurierbaren Bitbreite erschöpfend funktional validiert. Bestimmen Sie dabei auch die maximale Verzögerungszeit des CSA und ergänzen Sie dazu folgende Tabelle:

WIDTH	2	4	6	8	10
$t_{pd,CSA}$	4 ns	8 ns	12 ns	12 ns	16 ns

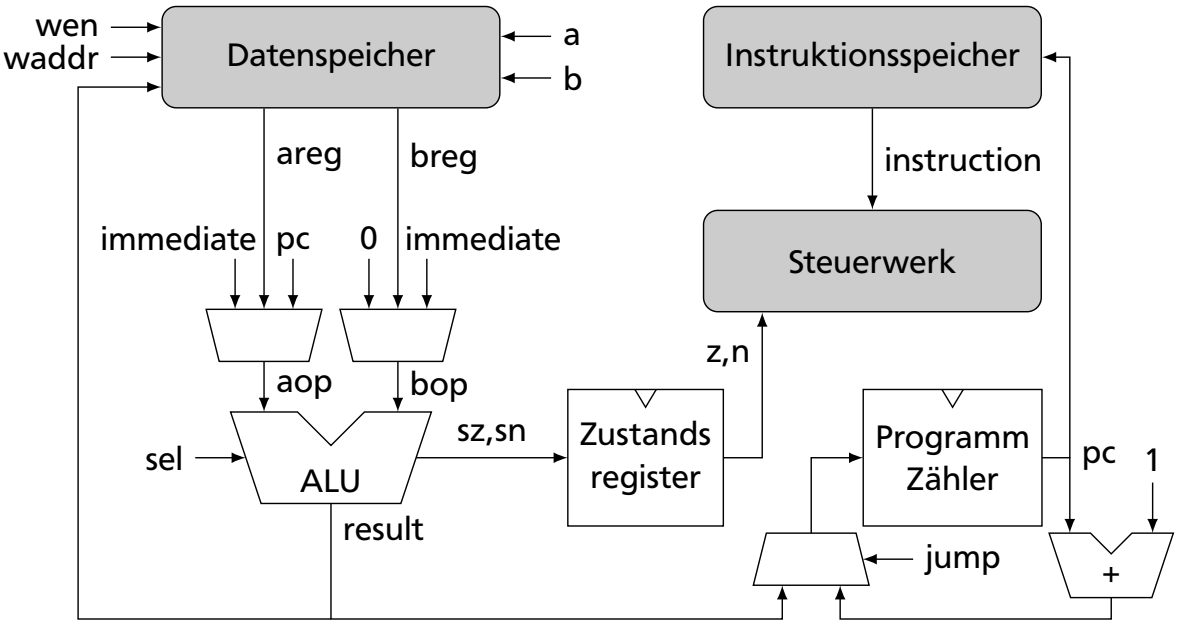
arith/adder/add_tb.sv

```
`timescale 1 ns / 10 ps
module add_tb;

  localparam W = 10;

  logic [W-1:0] a,b;
  logic [W  :0] s;

  add #(W) uut(a,b,s);

  real inputEvent;  // letzte Änderung an a oder b
  real delay;       // Verzögerungszeit
  real maxDelay = 0;// maximale Verzögerungszeit

  always @s delay = $realtime-inputEvent;

  initial begin
    $dumpfile("add_tb.vcd");
    $timeformat(-9, 0, " ns", 8);
    $dumpvars;

    // erschöpfender Test: alle Eingabekombinationen prüfen
    for (int i=0; i<(1<<2*W); i++) begin
```

```
      {a,b} = i;
      inputEvent = $realtime;
      #(10*W); // warten, bis s sicher stabil ist
      if (s != a+b) $display("%t: %0d+%0d=%0d but got %0d", $time, a, b, a+b, s);
      if (delay > maxDelay) maxDelay = delay;
    end

    $display("FINISHED add_tb for W=%0d, max delay=%f", W, maxDelay);
    $finish;
  end
endmodule
```

Übung 13.3 Modellprozessor [40 min]

In dieser Aufgabe wird ein einfacher Prozessor in SystemVerilog beschrieben und ein arithmetischer Algorithmus auf Basis des realisierten Instruktions-Satzes implementiert. Folgende Grafik zeigt die Architektur des Prozessors:



Einige für den Prozessor benötigte Quelldateien stehen in Moodle unter `SystemVerilog/src/processor` zur Verfügung.

Übung 13.3.1 Instruktionspeicher

Der Instruktionsspeicher benötigt lediglich einen asynchronen Leseport. Seine Initialisierung mit den Instruktionen des auszuführenden Programms erfolgt später in der Testbench des Prozessors. Implementieren Sie den Instruktionsspeicher mit folgender Schnittstelle:

processor/imem.sv

```
module imem #(parameter     WIDTH = 8,                  // Bitbreite der Instruktionen
              parameter     DEPTH = 16)                 // Anzahl der Instruktionen
             (input  logic [$clog2(DEPTH)-1:0] ADDR,   // Leseadresse
              output logic [WIDTH-1:0] D);             // Lesedaten
```

processor/imem.sv

```
  logic [WIDTH-1:0] m [0:DEPTH-1];
  assign D = m[ADDR];                                 // asynchrones Lesen
endmodule
```

Übung 13.3.2 Datenspeicher (Register)

Der Datenspeicher wird auch als Register-Satz bezeichnet und benötigt neben zwei asynchronen Leseports einen synchronen Schreibport. Dieser Speicher hat keinen Reset-Eingang und wird bei Bedarf durch das Ausführen bestimmter Instruktionen initialisiert. Implementieren Sie den Datenspeicher mit folgender generischer Schnittstelle:

processor/dmem.sv

```
module dmem
  #(parameter    WIDTH = 8,                                  // Bitbreite der Register
    parameter    DEPTH = 16)                                 // Anzahl der Register
   (input  logic CLK,                                        // Takt
    input  logic [$clog2(DEPTH)-1:0] AADDR,BADDR,WADDR,      // Schreib/Lese Adressen
    input  logic [WIDTH-1:0] WDATA,                          // Schreibdaten
    input  logic WEN,                                        // Schreibzugriff aktivieren
    output logic [WIDTH-1:0] ADATA, BDATA);                  // Lesedaten
```

processor/dmem.sv

```
  logic [WIDTH-1:0] m [0:DEPTH-1];

  assign ADATA = m[AADDR];                                   // asynchrones Lesen
  assign BDATA = m[BADDR];
  always_ff @(posedge CLK) if (WEN) m[WADDR] <= WDATA;       // synchrones Schreiben
endmodule
```

Übung 13.3.3 Arithmetisch-Logische Einheit (ALU)

Die ALU soll folgende Operationen umsetzen:

SEL	0	1	2	3	4	5	6	7	8	9	10
R	A+B	A-B	A&B	A|B	A^B	A<<B	A>>B	A<<<B	A>>>B	&A	|A

Dafür können die entsprechenden SystemVerilog Operatoren verwendet werden. Für alle anderen (ungenutzten) Werte des Selektionssignals (`SEL`) soll das Ergebnis der Addition ausgegeben werden. Neben dem Operationsergebnis sollen zwei Statusausgänge `Z` und `N` anzeigen, ob das Ergebnis 0 bzw. negativ ist. Implementieren Sie die kombinatorische ALU mit folgender generischer Schnittstelle:

processor/alu.sv

```
module alu #(parameter     WIDTH = 8)       // Bitbreite der Ein-/Ausgänge
            (input  logic [WIDTH-1:0] A,B, // Operanden
             input  logic [3:0] SEL,       // Auswahlsignal
             output logic [WIDTH-1:0] R,   // Ergebnis
             output logic Z,N);            // Statussignale
```

processor/alu.sv

```
  logic [WIDTH-1:0] r [0:10];
  assign r[0]  = A  +  B;
  assign r[1]  = A  -  B;
  assign r[2]  = A  &  B;
  assign r[3]  = A  |  B;
  assign r[4]  = A  ^  B;
  assign r[5]  = A << B;
  assign r[6]  = A >> B;
  assign r[7]  = A <<< B;
  assign r[8]  = A >>> B;
  assign r[9]  = & A;
  assign r[10] = | A;

  assign R = r[SEL > 10 ? 0 : SEL];
```

```
  assign Z = R == 0;
  assign N = R[WIDTH-1];
endmodule
```

Übung 13.3.4 Steuerwerk und Gesamtmodell

ALU, Instruktions- und Datenspeicher müssen im Modul des Prozessors instanziiert und mit dem Steuerwerk verknüpft werden. Die Bitbreiten der Daten und Adressleitungen werden durch den Instruktionssatz bestimmt und in `processor/isa.svh` als Präprozessor-Makros (beginnend mit backtick: `) definiert. Folgender Teil des Moduls ist bereits vorgegeben:

processor/core_stub.sv

```
`include "isa.svh"

module core (input logic CLK, RESET);

  localparam ZERO = `DATA_WIDTH'd0;

  logic signed [  `DATA_WIDTH-1:0] areg,breg,aop,bop,result,immediate;
  logic        [ `DADDR_WIDTH-1:0] a,b,r,waddr;
  logic        [ `INSTR_WIDTH-1:0] instruction;
  logic        [ `IADDR_WIDTH-1:0] pc;
  logic        [`OPCODE_WIDTH-1:0] opcode;
  logic        [              3:0] sel;
  logic                            wen,z,n,sz,sn,jump;

  // Datenspeicher (Register)
  dmem #(`DATA_WIDTH, `DATA_DEPTH) i_dmem
   (.CLK(CLK),   .WEN(wen),
    .AADDR(a),   .BADDR(b),   .WADDR(waddr),
    .ADATA(areg),.BDATA(breg),.WDATA(result));

  // Instruktionsspeicher
  imem #(`INSTR_WIDTH, `INSTR_DEPTH) i_imem (pc, instruction);

  // Arithmetisch-Logische Einheit
  alu #(`DATA_WIDTH) i_alu (aop,bop,sel,result,sz,sn);

  // Steuerwerk hier einfügen

endmodule
```

Das Steuerwerk soll als kombinatorische Logik im Modul des Prozessors realisiert werden. Es erzeugt aus der aktuellen Instruktion die Signale zum Ansteuern aller anderen Komponenten und realisiert so den Instruktionssatz des Prozessors:

Befehl	kodierte Instruktion	Registeränderung	nächster Programmzähler
ADD(r,a,b)	{4'b0000,7'bx,r,a,b}	R[r] = R[a] + R[b]	pc+1
SUB(r,a,b)	{4'b0001,7'bx,r,a,b}	R[r] = R[a] - R[b]	pc+1
AND(r,a,b)	{4'b0010,7'bx,r,a,b}	R[r] = R[a] & R[b]	pc+1
OR(r,a,b)	{4'b0011,7'bx,r,a,b}	R[r] = R[a] | R[b]	pc+1
XOR(r,a,b)	{4'b0100,7'bx,r,a,b}	R[r] = R[a] ^ R[b]	pc+1
SHL(r,a,b)	{4'b0101,7'bx,r,a,b}	R[r] = R[a]<< R[b]	pc+1
SHR(r,a,b)	{4'b0110,7'bx,r,a,b}	R[r] = R[a]>> R[b]	pc+1
ASHL(r,a,b)	{4'b0111,7'bx,r,a,b}	R[r] = R[a]<<<R[b]	pc+1
ASHR(r,a,b)	{4'b1000,7'bx,r,a,b}	R[r] = R[a]>>>R[b]	pc+1
ARED(r,a,b)	{4'b1001,7'bx,r,a,b}	R[r] = & R[a]	pc+1
ORED(r,a,b)	{4'b1010,7'bx,r,a,b}	R[r] = | R[a]	pc+1
MOV(r,a)	{4'b1011,7'bx,r,a,0}	R[r] = R[a]	pc+1
LDI(immediate)	{4'b1100,immediate}	R[0] = immediate	pc+1
JMP(immediate)	{4'b1101,immediate}		pc+ immediate
JN(immediate)	{4'b1110,immediate}		pc+(n ? immediate : 1)
JZ(immediate)	{4'b1111,immediate}		pc+(z ? immediate : 1)

Dabei sind a,b und r Registeradressen der Breite `DADDR_WIDTH und die immediate Einträge sind vorzeichenbehaftete Konstanten der Breite `DATA_WIDTH. n und z sind die Statussignale der ALU für die unmittelbar zuvor ausgeführten Instruktion. Sie müssen in einem Statusregister gepuffert werden, um in Abhängigkeit vom Ergebnis einer Berechnung einen Sprung im Programmfluss auszuführen. Wie im Schaltbild des Prozessors angedeutet, sollte das Sprungziel (pc+immediate) durch die ALU berechnet werden.
Die Befehle MOV ("move", für das Kopieren von Registern) und LDI ("load immediate", für das Laden von Konstanten) führen eigentlich keine Berechnung aus, lassen sich als Addition mit Null aber auch über die ALU realisieren.
Ergänzen Sie das Prozessormodul um Steuerwerk, Statusregister und Programmzähler.

processor/core.sv

```
27  // Instruktion in Bestandteile zerlegen
28  assign opcode    = instruction[`INSTR_WIDTH-1 -: `OPCODE_WIDTH];
29  assign immediate = instruction[0              +:  `DATA_WIDTH];
30  assign b         = instruction[0*`DADDR_WIDTH +:  `DADDR_WIDTH];
31  assign a         = instruction[1*`DADDR_WIDTH +:  `DADDR_WIDTH];
32  assign r         = instruction[2*`DADDR_WIDTH +:  `DADDR_WIDTH];
33
34  // Steuersignale ableiten
35  assign {aop,bop} = opcode <= `ORED ? {areg,     breg}      // ADD,SUB,...,ORED
36                   : opcode >= `JMP  ? {pc,       immediate} // JMP,JN,JZ
37                   : opcode == `MOV  ? {areg,     ZERO}      // MOV
38                   :                   {immediate,ZERO};     // LDI
39  assign sel       = opcode;  // ADD für opcode > `ORED laut ALU Spezifikation
40  assign waddr     = opcode == `LDI  ? 0 : r;
41  assign wen       = opcode <  `JMP;
42  assign jump      = opcode == `JMP
43                  || (opcode == `JN) && n
44                  || (opcode == `JZ) && z;
45
46  // Programmzähler
47  always_ff @(posedge CLK) pc    <= RESET ? 0 : jump ? result : pc+1;
48
49  // Statusregister
50  always_ff @(posedge CLK) {z,n} <= RESET ? 0 : {sz,sn};
```

Übung 13.3.5 Assembler-Programm – **Zusatzaufgabe**

Um die Funktionalität der Prozessor-Implementierung zu überprüfen, muss ein konkretes Programm in den Instruktionsspeicher geladen werden, dessen Abarbeitung dann beobachtet werden kann. Dazu wird folgende Testbench zur Verfügung gestellt:

processor/tb.sv

```
1  `default_nettype none
2  `timescale 1 ns / 10 ps
3  `include "isa.svh"
4
5  `define PROGRAM "simple.asm"
6
7  module tb;
8
9    // Prozessor takten
10   logic   clk=0, reset=1;
11   always  #0.5           clk   <= ~clk;
12   initial @(posedge clk) reset <= 0;
13   core uut (clk, reset);
14
15   // simuierte Signale (Speicher müssen explizit hinzugefügt werden)
16   initial begin
17     $dumpfile("tb.vcd");
18     $dumpvars;
19     for (int i=0; i<`INSTR_DEPTH; i++) $dumpvars(1, uut.i_imem.m[i]);
20     for (int i=0; i<`DATA_DEPTH;  i++) $dumpvars(1, uut.i_dmem.m[i]);
21   end
22
23   // Programm in Instruktionsspeicher laden
24   `include "asm.svh"
25   initial begin
26     clear_instructions;
27     `include   `PROGRAM
28     $readmemb({`PROGRAM,".bin"}, uut.i_imem.m);
29   end
30
31   // Simulation bei Endlosschleife abbrechen
32   always @(posedge clk) if (uut.opcode == `JMP && uut.immediate == 0) begin
33     $display("FINISHED tb");
34     $finish;
35   end
36 endmodule
```

Dabei wird das zu ladende Programm in Zeile 5 spezifiziert, welches neben SystemVerilog Kommentaren ausschließlich die oben angegebenen Assembler Befehle verwenden darf. Ein einfaches Beispiel für ein solches Assembler-Programm sieht wie folgt aus:

processor/simple.asm

```
1  /*PC*/
2  /* 0*/ LDI(1);       // R[0] = 1
3  /* 1*/ MOV(1,0);     // R[1] = R[0] = 1
4  /* 2*/ LDI(2);       // R[0] = 2
5  /* 3*/ MOV(2,0);     // R[2] = R[0] = 2
6  /* 4*/ ADD(3,1,2);   // R[3] = R[1] + R[2] = 3
7  /* 5*/ JMP(0);       // Endlosschleife
```

Die erste Kommentarspalte gibt dabei die Adresse des Befehls im Instruktionsspeicher an. Dies ist hilfreich bei der Verwendung von Sprüngen, da hier (im Gegensatz zu vollwertigen Assembler-Programmen) keine Sprungmarken verwendet werden können. Stattdessen muss der relative Abstand zum Sprungziel als `immediate` des Sprungbefehls angegeben werden. Daher realisiert der unbedingte Sprung um Null Instruktionen (`JMP(0)`) eine Endlosschleife. Diese Endlosschleife wird zum Abbruch der Simulation verwendet und sollte daher der letzte Befehl eines jeden Programms sein.
Die Testbench nimmt an, dass die lokalen Arrays im Instruktions- und Datenspeicher mit `m` bezeichnet werden (Zeile 19, 20, 28). Passen Sie Ihre Implementierungen entsprechend an, da sonst auch GTKWave nach der Simulation nicht die richtigen Signale anzeigt.

Zum Starten der Simulation genügt der Aufruf der Testbench, das Assembler-Programm muss also nicht als Teil der Quelldateien spezifiziert werden.
Realisieren Sie eine sequentielle Multiplikation von zwei vorzeichenlosen 8 bit Operanden. In Java würde dieser Algorithmus wie folgt implementiert:

processor/mul.java

```
int a = 42;
int b = 37;
int p = 0;
for (int n=8; n!=0; n--) {
  if (b & 1 == 1) p += a;
  a = a << 1;
  b = b >> 1;
}
```

Dabei werden in den ersten beiden Zeilen die miteinander zu multiplizierenden Operanden `a` und `b` spezifiziert. Nach Abbruch der Schleife enthält `p` das Produkt `a * b`. Setzen Sie diesen Algorithmus mit den Assembler-Befehlen des Modellprozessors um. Dabei sollen die Variable `a`, `b` und `p` in den Registern 1, 2 und 3 abgelegt werden. Evaluieren Sie Ihre Implementierung für verschiedene Operanden.

processor/mul.asm

```
/*PC*/
/* 0*/ LDI(42); MOV(1,0);              // R[1]  = 42 (a)
/* 2*/ LDI(37); MOV(2,0);              // R[2]  = 37 (b)
/* 4*/ LDI( 0); MOV(3,0);              // R[3]  =  0 (p)
/* 6*/ LDI( 8); MOV(4,0);              // R[4]  =  8 (n)
/* 8*/ LDI( 1); MOV(5,0);              // R[5]  =  1

//loop:
/*10*/ AND(0,2,5); JZ(2); ADD(3,3,1); // if (b & 1) p += a
/*13*/ SHL(1,1,5);                    // a <<= 1
/*14*/ SHR(2,2,5);                    // b >>= 1
/*15*/ SUB(4,4,5); JZ(2); JMP(-7);    // if (--n) goto loop
/*18*/ JMP(0);                        // Endlosschleife
```