Digitaltechnik
Wintersemester 2017/2018
13. Übung

TECHNISCHE UNIVERSITÄT DARMSTADT

Andreas Engel, Raad Bahmani **LÖSUNGSVORSCHLAG** **KW05**

Die Präsenzübungen werden in Kleingruppen während der wöchentlichen Übungsstunde bearbeitet. Bei Fragen hilft Ihnen Ihr Tutor gerne weiter. 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 Carry Lookahead Compare [30 min]

In dieser Aufgabe soll ein „größer als“ Operator auf vorzeichenlosen Zahlen mit dem Generate und Propagate Konzept des Carry Lookahead Adders realisiert werden.

Übung 13.1.1 Generate / Propagate Block

Implementieren Sie ein k bit Carry Lookahead Block mit folgender generischer Schnittstelle:

arith/greater.sv

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

Dabei soll nur die Übertragslogik ohne eine tatsächliche Summenberechnung realisiert werden. `CO` soll also dem Bit der Summe `A+B+CI` mit dem Gewicht $2^{\texttt{WIDTH}}$ entsprechen. Ermitteln Sie dazu zunächst die Generate und Propagate Signale (aus `A` und `B`), so dass der kritische Pfad zwischen `CI` und `CO` möglichst kurz ist.

arith/greater.sv

```
5    logic [WIDTH-1:0] pc,gc;
6    assign pc = A | B; // spaltenweise propagate
7    assign gc = A & B; // spaltenweise generate
8
9    logic p,g;
10   assign p = &pc;    // propagate des ganzen Blocks
11
12   always_comb begin  // generate des ganzen Blocks
13     g = gc[0];
14     for (int i=1; i<WIDTH; i++) g = gc[i] | pc[i] & g;
15   end
16
17   assign CO = g | p & CI;   // kurzer Carry-Pfad
18
19 endmodule
```

Übung 13.1.2 Verkettete Blöcke

Verketten Sie 4 bit Carry Lookahead Blöcke zu einem kombinatorischen Operator, der `A > B` berechnet. Implementieren Sie dazu die folgende Schnittstelle:

arith/greater.sv

```
22 module greater #(parameter    WIDTH=8)
23                 (input logic [WIDTH-1:0] A,B, output logic Y);
```

Nutzen Sie dabei, dass $A > B \Leftrightarrow B - A < 0$. Um vorzeichenlose Zahlen richtig zu Negieren, muss `A` zunächst um eine Stelle (vorzeichenlos) expandiert werden. Die Breite der Operanden (`WIDTH`) muss dabei nicht unbedingt ein ganzzahliges Vielfaches der Blockbreite sein.

arith/greater.sv

```
25    localparam k = 4;                  // Breite der Carry-Blöcke
26    localparam blocks = WIDTH/k;       // Anzahl der benötigten vollen Blöcke
27    localparam r = WIDTH - blocks*k;   // Breite des letzten (zusätzlichen) Blocks
28
29    logic [blocks:0] c;                // Übertrag zwischen den Böcken
30    assign c[0] = 1'b1;                // Übertrag in ersten Block für Subtraktion
31
32    genvar i;
33    generate
34      // volle Blöcke
35      for (i=0; i<blocks; i=i+1) begin
36        block #(k) b (B[i*k     +: k], ~A[i*k     +: k], c[i], c[i+1]);
37      end
38
39      // letzter Block ...
40      if (r) begin
41        logic ny;
42        block #(r) b (B[WIDTH-1 -: r], ~A[WIDTH-1 -: r], c[blocks], ny);
43        assign Y = ~ny;
44
45      // ... oder direkte Zuweisung an Ausgang
46      end else begin
47        assign Y = ~c[blocks];
48      end
49    endgenerate
50  endmodule
```

Die vorzeichenlose Expansion von `A` vor der Komplementbildung resultiert nach der Komplementbildung in einer 1 in der höchsten Stelle, die zum `CO` des letzten Blocks addiert werden muss. Alternativ kann auch einfach auf die Vorzeichenexpansion verzichtet werden, und stattdessen das negierte `CO` als Ergebnis des Vergleichs verwendet werden.

Übung 13.1.3 Verifikation

Implementieren Sie eine Testbensh zur erschöpfenden Verifikation Ihrers Vergleichers für `WIDTH=9`.

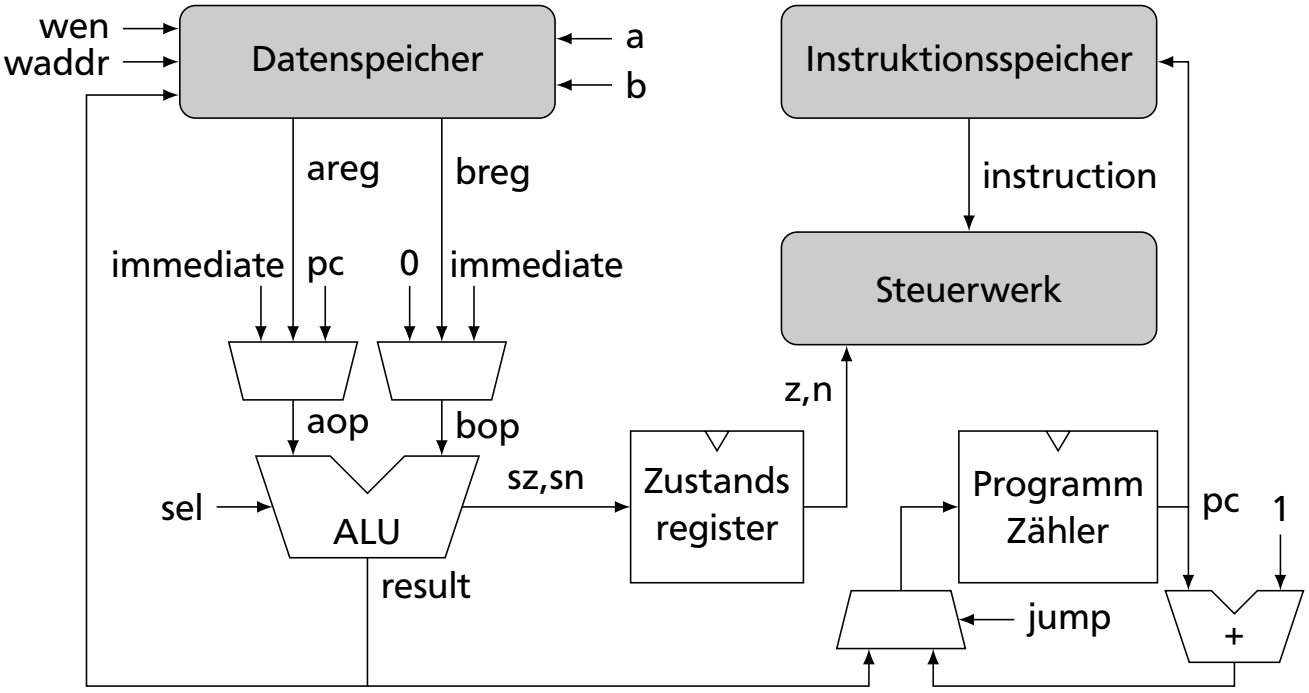
arith/greater_tb.sv

```
1   `timescale 1 ns / 10 ps
2   module greater_tb;
3
4     localparam W = 9;
5     logic [W-1:0] a,b;
6     logic y;
7     greater #(W) uut(a,b,y);
8
9     initial begin
10      $dumpfile("greater_tb.vcd");
11      $dumpvars;
12
13      for (int i=0; i<(1<<2*W); i++) begin
14        {a,b} = i;
15        #1;
16        if (y != (a > b))
17          $display("%t: %0d>%0d expected %b, but got %b ", $time, a, b, a > b, y);
18      end
```

```
19
20      $display("FINISHED greater_tb for W=%0d", W);
21      $finish;
22    end
23
24  endmodule
```

Übung 13.2 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 (Harvard-) Architektur des Prozessors:



Einige für den Prozessor benötigte Quelldateien sind im Moodle verfügbar (`examples/cpu`).

Übung 13.2.1 Instruktionspeicher

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

processor/imem.sv

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

processor/imem.sv

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

Übung 13.2.2 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 13.2.3 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 anzeigen, ob das Operationsergebnis Null (Z) oder negativ (N) 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 13.2.4 Steuerwerk und Gesamtmodell

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

  // hier Steuerwerk einfügen

endmodule
```

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));

```

```
21    // Instruktionsspeicher
22    imem #(`INSTR_WIDTH, `INSTR_DEPTH) i_imem (pc, instruction);
23
24    // Arithmetisch-Logische Einheit
25    alu #(`DATA_WIDTH) i_alu (aop,bop,sel,result,sz,sn);
26
27    // hier Steuerwerk einfügen
28
29  endmodule
```

Das Steuerwerk soll als kombinatorische Logik im Toplevel-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 Blockschaltbild des Prozessors angedeutet, sollte das Sprungziel (pc+immediate) durch die ALU berechnet werden.
Die Befehle MOV („move“, für 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 Prozessor Toplevel-Modul 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 13.2.5 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    // Rechenkern 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
```

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    // Rechenkern 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 jedes 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 ein `path/to/bin/sim tb`, das Assembler-Programm muss also nicht als Teil der Quelldateien spezifiziert werden.
Realisieren Sie nun eine sequentielle Multiplikation von zwei vorzeichenlosen 8 bit Operanden nach dem in Übung 11.2 verwendeten Algorithmus. In Java würde dieser Algorithmus wie folgt implementiert:

processor/mul.java

```
1  int a = 42;
2  int b = 37;
```

```
3 int p = 0;
4 for (int n=8; n!=0; n--) {
5   if (b & 1 == 1) p += a;
6   a = a << 1;
7   b = b >> 1;
8 }
```

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 Register 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
```