Einführung in

Computer Microsystems

Sommersemester 2010

Wolfgang Heenes

TECHNISCHE UNIVERSITÄT DARMSTADT

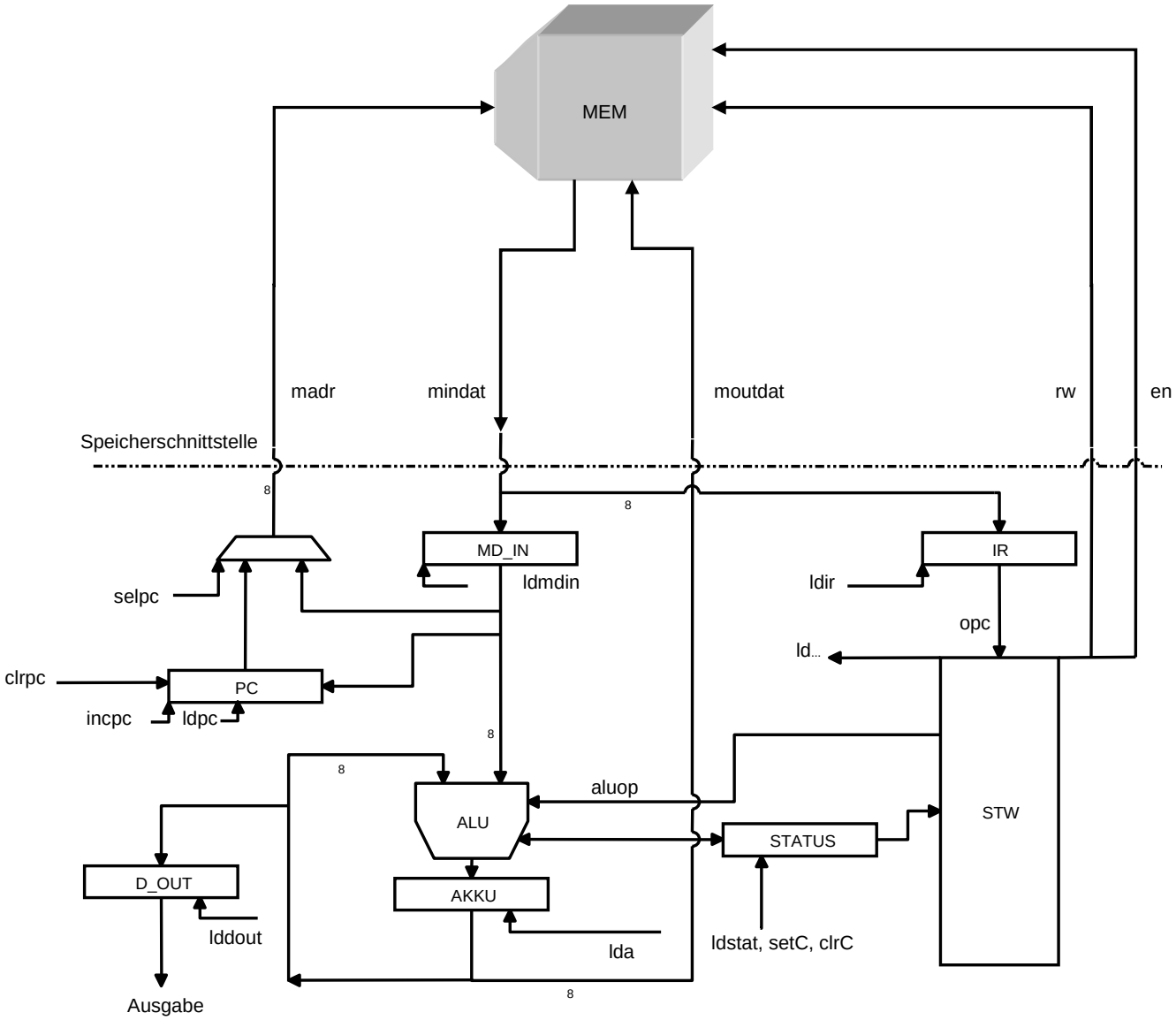
7. Aufgabenblatt mit Lösungsvorschlag

02.06.2010

Aufgabe 1: Realisierung der ALU für den Modellrechner

Die in den Folien der 7. Vorlesung spezifizierte ALU soll nun in Verilog HDL umgesetzt werden.

Die folgende Abbildung zeigt nochmals die Architektur/Struktur des Modellrechners.



Die Liste der Maschinenbefehle des Modellrechners sind in der folgenden Tabelle angegeben. Alle Befehle der ALU und die dadurch veränderten Statusbits sind der Liste zu entnehmen. Statusbits sind: O-Overflow, C-Carry, N-Negativ und Z-Zero.

Status: *Beim Statusbits bedeutet ! = wird verändert und + = wird nicht verändert.*
Der Eintrag – bedeutet, dass die entsprechenden Befehle keine Statusbits beinflussen.

OPC	*I*	*Befehl*	*MNEMONIC*	*Byte1*	*Byte2*	*Status OCNZ*
0	*0*	*lade AKKU mit Wert*	*LDA #C*	*0000 0000*	*WERT*	*00!!*
0	*1*	*lade AKKU mit dem Speicherinhalt von Adresse*	*LDA N*	*0000 0001*	*ADRESSE*	*00!!*
1	*X*	*AKKU bitweise negieren*	*NOT*	*0001 000x*		*00!!*
2	*0*	*AKKU mit Wert verunden*	*AND #C*	*0010 0000*	*WERT*	*00!!*
2	*1*	*AKKU mit dem Speicherinhalt von Adresse verunden*	*AND N*	*0010 0001*	*ADRESSE*	*00!!*
3	*0*	*AKKU mit Wert verodern*	*OR #C*	*0011 0000*	*WERT*	*00!!*
3	*1*	*AKKU mit dem Speicherinhalt von Adresse verodern*	*OR N*	*0011 0001*	*ADRESSE*	*00!!*
4	*0*	*Wert zum AKKU addieren*	*ADD #C*	*0100 0000*	*WERT*	*!!!!*
4	*1*	*Speicherinhalt von Adresse zum AKKU addieren*	*ADD N*	*0100 0001*	*ADRESSE*	*!!!!*
5	*X*	*AKKU nach rechts schieben*	*SHR*	*0101 000x*		*0!!!*
6	*X*	*AKKU nach rechts schieben Vorzeichen nachziehen*	*ASR*	*0110 000x*		*0!!!*
7	*X*	*AKKU nach links schieben*	*SHL*	*0111 000x*		*0!!!*
8	*X*	*Carrybit im Statusregister löschen*	*CLRC*	*1000 000x*		*+0++*
9	*X*	*Carrybit im Statusregister setzen*	*SETC*	*1001 000x*		*+1++*
10	*X*	*NOP, keine Operation*	*NOP*	*1010 000x*		*–*
11	*X*	*Sprung zur Adresse Label*	*BRA L*	*1011 0000*	*LABEL*	*–*
12	*X*	*Sprung zur Adresse Label falls Z-Statusbit gesetzt ist*	*BRZ L*	*1100 0000*	*LABEL*	*– –*
13	*X*	*Schreibe Inhalt des AKKUs an Adresse in den Speicher*	*STA N*	*1101 0000*	*ADRESSE*	*– –*
14	*X*	*Schreibe Inhalt des AKKUs in das Ausgaberegister*	*OUT*	*1110 000x*		*– –*

a) *Implementieren Sie die ALU als Schaltnetz in Verilog HDL. Die vorzeichenbehafteten Zahlen werden als 2K-Zahlen dargestellt. Testen Sie die korrekte Funktionsweise der Befehle.*

Lösungsvorschlag:

Überlegung zur Berechnung von Carry- und Overflow-Bit und der Vorzeichenbehandlung.

1. Fall: Operanden haben gleiches Vorzeichen.

```
  00000001
  00010010
-----------
  000010011
```

Ergebnis: Kein Überlauf, kein Carry.

2. Fall: Operanden haben gleiches Vorzeichen.

```
  0111 1111
  0111 1111
-------------
  01111 1110
```

Überlauf, kein Carry.

```
  1000 0000
  1000 0000
-------------
 10000 0000
```

Carry und Überlauf.

3. Fall: Operanden haben verschiedenes Vorzeichen.

```
  1111 1111
  0000 0001
-------------
 10000 0000
```

Carry, kein Überlauf.

Es reicht also nicht, nur Carry und oberstes Summenbit zu betrachten. Außerdem müssen die Vorzeichen der Operanden berücksichtigt werden.

Achtung: Carrys können bei jeder Rechnung mit 2K-Zahlen auftreten.

Beispiel:

```
  00010100 = 20
   11111011 = -5
---------------
 100001111 = 15
```

```
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
```

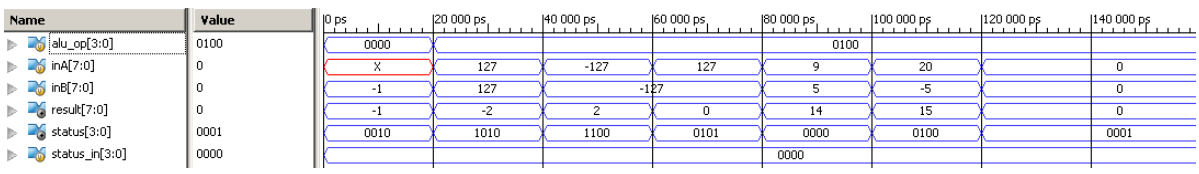
```
        opc_or   : begin // Bitweise ODER
                    result = inA | inB;
                    Z = result == 8'b0;
                    N = result[7];
                    {OV, C} = 2'b0;
                   end
        opc_add  : begin // 2K-addieren mit Carry-In
                    Z = result == 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
```

Folgende Testbench wird für die Simulation benutzt. Um auch die richtige Behandlung der 2K-Zahlen zu testen, werden auch alle Möglichkeiten getestet.

```
`timescale 1ns / 1ps
module testbed;

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,
```

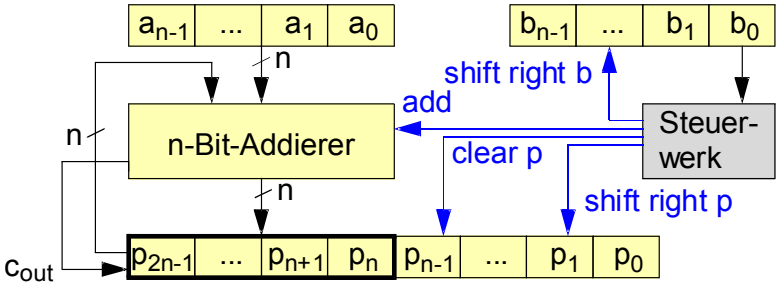
```
opc_clrc = 4'b1000,
opc_setc = 4'b1001;

    // Inputs
    reg [7:0] inA;
    reg [7:0] inB;
    reg [3:0] alu_op;
    reg [3:0] status_in;

    // Outputs
    wire [7:0] result;
    wire [3:0] status;

    // Instantiate the Unit Under Test (UUT)
    ALU uut (
        .inA(inA),
        .inB(inB),
        .alu_op(alu_op),
        .status_in(status_in),
        .result(result),
        .status(status)
    );

    initial begin
        // Initialize Inputs

        status_in = 0;

        alu_op = opc_load;
        inB = -1;

        #20
        alu_op = opc_add;

        inA = 127;
        inB = 127;

        #20
        inA = -127;
        inB = -127;

        #20
        inA = 127;
        inB = -127;

        #20
        inA = 9;
        inB = 5;

        #20
        inA = 20;
        inB = -5;

        #20
        inA = 0;
        inB = 0;

    end

endmodule
```

Die Verhaltenssimulation der Befehle LDA und ADD ergibt folgendes Bild. Die Simulation zeigt die korrekte Funktionsweise der Statusbits.



Aufgabe 2: Implementierung einer seriellen Multiplikation

Folgendes Schaltwerk realisiert eine serielle Multiplikation.



Der Algorithmus zur seriellen Multiplikation funktioniert wie folgt.
Die Befehle in Klammern geben die Mikrooperationen an (vgl. Implementierung in Verilog HDL).

- *Lösche p (clear p)*
- *Ermittle b_0*
- *Addiere a auf $(p_{2n-1}, ..., p_{n+1}, p_n)$ oder nicht, je nach b_0 (add)*
- *Verschiebe p einschließlich c_{out} der vorherigen Addition (shift right p)*
- *Verschiebe b (shift right b)*

a) *Berechnen Sie nach dem vorgestellten Algorithmus das Produkt $p = a \cdot b$ für $a = 0010$ und $b = 0011$.*

Lösungsvorschlag:

$[p_{2n-1}, p_n]$	$[p_{n-1}, p_0]$	b	Erklärung
xxxx	xxxx	0011	lösche p
0000	0000	0011	addiere a ($b_0 = 1$)
0010	0000	0011	shift p
0001	0000	0011	shift b
0001	0000	0001	addiere a ($b_0 = 1$)
0011	0000	0001	shift p
0001	1000	0001	shift b
0001	1000	0000	shift p
0000	1100	0000	shift b
0000	1100	0000	shift p
0000	0110	0000	shift b
0000	0110	0000	Ergebnis

$p = a \cdot b = 0000\,0110$

b) *Implementieren Sie die vorgestellte serielle Multiplikation in Verilog HDL. Die Register a und b sind jeweils 4 Bit breit. Das Ergebnisregister p ist 8 Bit breit. Verwenden Sie die vorgegebenen Mikrooperationen. Das Steuerwerk kann als Zustandsgraph beschrieben werden. Die Kodierung der Zustände kann frei gewählt werden.*

Lösungsvorschlag:

Toplevel-Modul des Multiplizierers:

```
`timescale 1ns / 1ps
module sMult(a_in, b_in, result, finish, clock, reset);

input [n-1:0] a_in;
input [n-1:0] b_in;
output [2*n-1:0] result;

output finish;
input clock;
input reset;

parameter n=1;

reg [2*n-1:0] p;
reg [n-1:0] a;
reg [n-1:0] b;
reg carry_reg;

wire [n-1:0] sum;
wire carry;

adder #(.n(n)) inst_adder(
.a(a),
.b(p[2*n-1:n]),
.sum(sum),
.carry(carry)
);

wire add, shift_b, clear_p, shift_p;

ctrl #(.n(n)) ctrl_inst(
.b0(b[0]),
.shift_b(shift_b),
.add(add),
.clear_p(clear_p),
.shift_p(shift_p),
.clock(clock),
.reset(reset)
);

always@(posedge clock or posedge reset)
begin

carry_reg<=carry & add;

if(reset)
begin
    p<={n{1'b0}};
    a<=a_in;
    b<=b_in;
end
else
begin
    case({add, clear_p, shift_p & ~finish})
        3'b100: p[2*n-1:n] <= sum;
        3'b010: p<={n{1'b0}};
        3'b001: p<={carry_reg,p[2*n-1:1]};
        default: p<=p;
    endcase

    if(shift_b) b<={1'b0,b[n-1:1]};

end
end

assign result=p;

/* Counter für Finish */
reg [n/2:0] count;
```

```
always@(posedge clock or posedge reset)
begin
    if(reset)
        count<=0;
    else
        count<=count+(((shift_p & shift_b) ^ (shift_p & ~shift_b)) & ~finish);
end

assign finish = count==n;

endmodule
```

Beschreibung des Addierers:

```
‘timescale 1ns / 1ps
module adder(a, b, sum, carry);

parameter n=1;

input [n-1:0] a;
input [n-1:0] b;
output [n-1:0] sum;
output carry;

assign {carry,sum} = {1’b0,a} + {1’b0,b};

endmodule
```

Beschreibung des Kontrollers:

```
‘timescale 1ns / 1ps
module ctrl(b0, shift_b, add, clear_p, shift_p, clock, reset);

parameter n=1;

input b0;
output shift_b;
output add;
output clear_p;
output shift_p;

input clock;
input reset;

reg add;
reg clear_p;
reg finish;
reg shift_p;
reg shift_b;

reg FSM;

always@(posedge clock or posedge reset)
begin

if(reset)
begin
    FSM<=0;
    add<=1’b0;
    clear_p<=1’b1;
    shift_p<=1’b0;
    shift_b<=1’b0;
end
else
    case(FSM)

    0:
    begin
        clear_p<=1’b0;
```

```
            if(b0)
            begin
                FSM<=1;
                add<=1'b1;
                shift_p<=1'b0;
                shift_b<=1'b1;
            end
            else
            begin
                FSM<=0;
                add<=1'b0;
                shift_p<=1'b1;
                shift_b<=1'b1;
            end
        end

        1:
        begin
            add<=1'b0;
            FSM<=0;
            shift_p<=1'b1;
            shift_b<=1'b0;
        end

        endcase
    end

    endmodule
```

Testbench:

```
    `timescale 1ns / 1ps
    module testbed;

    parameter n=4;

        // Inputs
        reg [n-1:0] a;
        reg [n-1:0] b;
        wire [2*n-1:0] result;
        reg clock;
        reg reset;

        // Outputs
        wire finish;

        // Instantiate the Unit Under Test (UUT)
        sMult #(.n(n)) uut (
            .a_in(a),
            .b_in(b),
            .result(result),
            .finish(finish),
            .clock(clock),
            .reset(reset)
        );

        initial begin
            // Initialize Inputs
            reset=1;
            a = 15;
            b = 3;
            #15
            reset = 0;

            #85
            reset=1;
            a = 15;
            b = 15;
            #15
```

```
            reset = 0;

            #100
            reset=1;
            a = 4'b0010;
            b = 4'b0011;
            #15
            reset =  0;

        end

        initial begin
            clock = 1;
            forever #5 clock = !clock;
        end
    endmodule
```

c) *Zeigen Sie die korrekte Funktionsweise durch eine Simulation.*

Lösungsvorschlag:

Die Verhaltenssimulation zeigt die korrekte Funktionsweise. Ein *finish* Signal zeigt an, wenn das Ergebnis fertig berechnet wurde.