

Digitaltechnik

Wintersemester 2021/2022

13. Übung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

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.

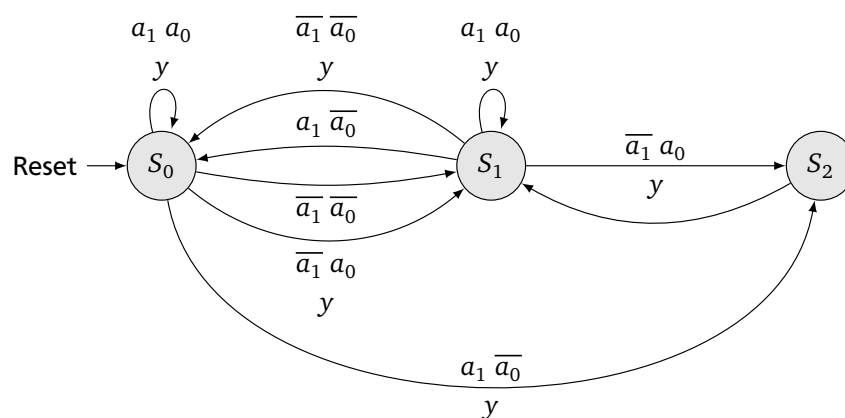
Mit der angegebenen Bearbeitungszeit für die einzelnen Aufgaben können Sie Ihren Leistungsstand besser einschätzen. Die mit **“Zusatzaufgabe”** gekennzeichneten Aufgaben sind zur zusätzlichen Vertiefung für interessierte Studierende gedacht und daher nicht im Zeitumfang von 90 Minuten einkalkuliert.

Übung 13.1 Robuste Endliche Automaten

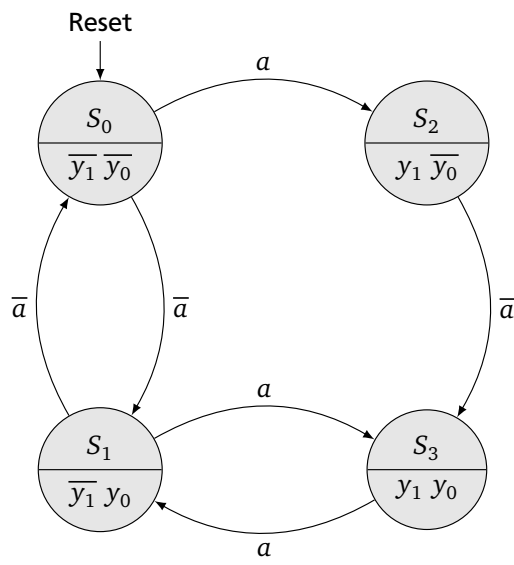
[20 min]

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.

a)



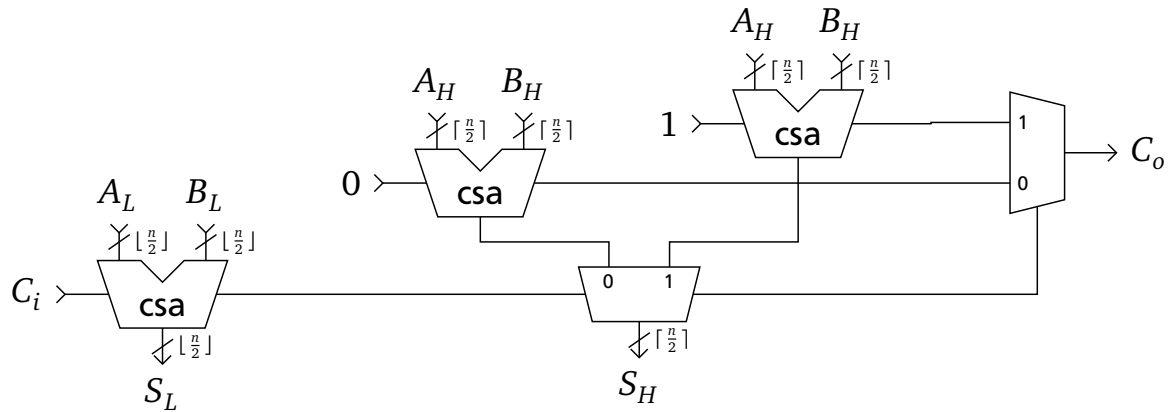
b)



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 Halbwords ($A_L + B_L + C_i$) 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.

Übung 13.2.2 Modul-Kapselung

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

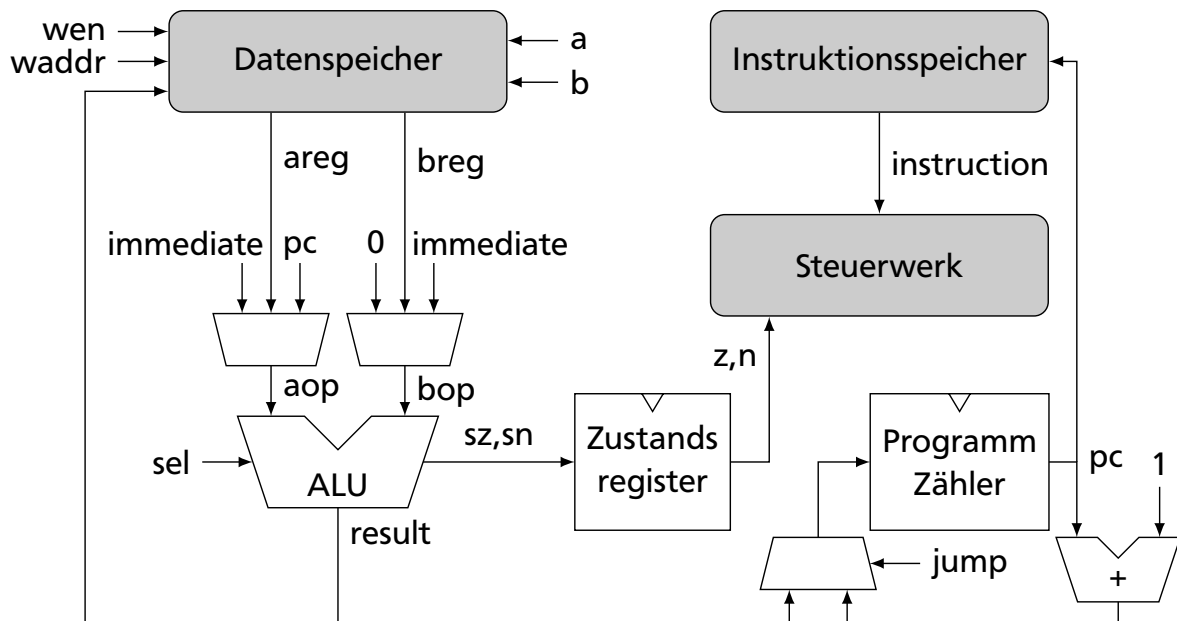
```
21 module add #(parameter    WIDTH=4)
22     (input logic  [WIDTH-1:0] A, B, output logic [WIDTH:0] S);
```

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
$t_{pd,CSA}$					

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 Instruktionsspeicher

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
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
```

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

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

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


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
1 `include "isa.svh"
2
3 module core (input logic CLK, RESET);
4
5     localparam ZERO = `DATA_WIDTH'd0;
6
7     logic signed [ `DATA_WIDTH-1:0] areg,breg,aop,bop,result,immediate;
8     logic [ `DADDR_WIDTH-1:0] a,b,r,waddr;
9     logic [ `INSTR_WIDTH-1:0] instruction;
10    logic [ `IADDR_WIDTH-1:0] pc;
11    logic [ `OPCODE_WIDTH-1:0] opcode;
12    logic [ 3:0] sel;
13    logic wen,z,n,sz,sn,jump;
14
15    // Datenspeicher (Register)
16    dmem #(`DATA_WIDTH, `DATA_DEPTH) i_dmem
17    (.CLK(CLK), .WEN(wen),
18     .AADDR(a), .BADDR(b), .WADDR(waddr),
19     .ADATA(areg),.BDATA(breg),.WDATA(result));
20
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    // Steuerwerk hier einfügen
28
29 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 aus-

geführten 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.

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    // simulierte 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 wer-

den. 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

```
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 Registern 1, 2 und 3 abgelegt werden. Evaluieren Sie Ihre Implementierung für verschiedene Operanden.