Graz University of Technology

IAIK Institute

Institute for Applied Information Processing and Communications

Rechnerorganisation Übungen 2010

Toy CPU

Assignment 4

composed by	Robin Ankele	(0931951)
	Christoph Bauernhofer	(0931145)
Groupname	ausserlechner15	
Tutor	Simon Ausserlechner	
Term	Summerterm 2010	
Date	11 06 2010	

Inhaltsverzeichnis

Kapitel 1
Aufgabenstellung

Allgemeines

Bei dem Assignment 4 in Rechnerorganisation Übungen 2010 soll eine 16 bit CPU namens TOY entwickelt werden. Die verwendete Programmiersprache ist Verilog.

Toy ist eine JAVA Simulation von der University Princeton. Bei diesem Assignment soll eine CPU entwickelt werden, welcher den Befehlssatz der Toy JAVA Simulation versteht und anwenden kann.

Es sollen 5 Testprogramme (ausserlechner4t 1-5.asm) in Toy Assembler entworfen und abgegeben werden. Hierbei ist darauf zu achten das die Testprogramme mit der TOY Java Simulation kompatibel sind.

Des weiteren sind ein:

Funktionales Modell der TOY-CPU samt MEMory und IO-Einheit in einem Testbett. TOY soll dabei den gesamten Instruktionssatz beherrschen. Der Speicher soll zu Simulationsbeginn mit dem Boot-Loader geladen sein. Damit sollte dieses Modell in der Lage sein, sogenannte EXE-Dateien über std_in zu laden und diese danach zu exekutieren. Dieses soll als (ausserlechner154f.v) abgegeben werden.

Gemischtes Modell des Systems aus Punkt 1. Lediglich TOY ist als gemischtes Modell auszuführen. Die darin vorkommende ALU und die Registerbank (16 16-Bit-Register) müssen nicht in Einzelteile aufgelöst werden. MEM und IO-Modul können als funktionales Model bleiben. Dieses soll als (ausserlechner154m.v) abgegeben werden.

zu entwickeln.

Kapitel 2
Toy CPU Aufbau

Die Toy CPU besteht aus:

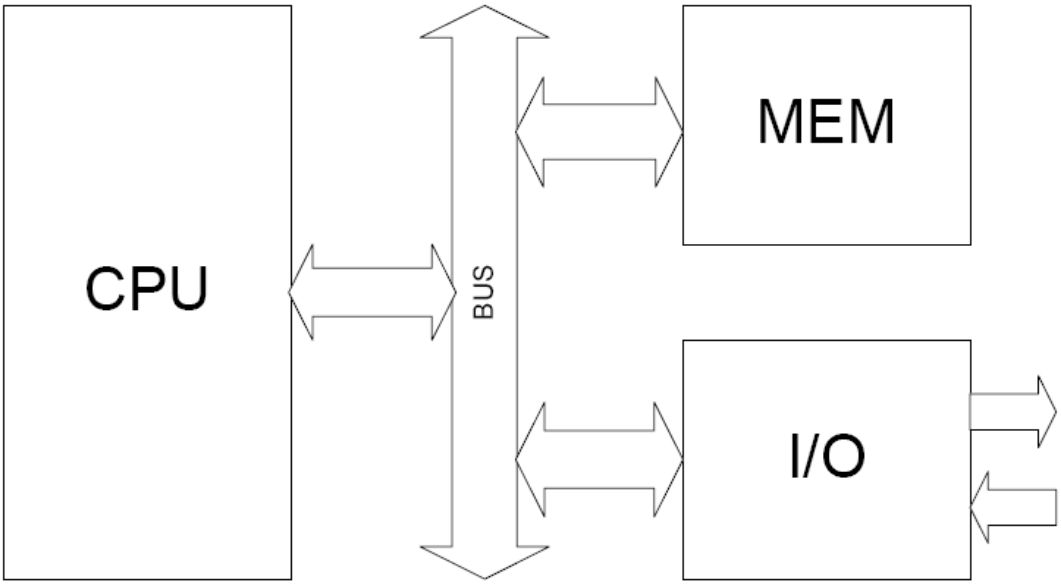


Abbildung 1. Toy CPU Aufbau

Der Toy CPU Module
Einem MEM Module
Einem I/O Module

Spezifikationen:

TOY CPU	16bit CPU
Speicherwort	16bit
	256 Speicherworte
Adressen	0-254 MEM
	255 I/O
Register	16 16bit Register (intern)
	(R1-RF), R0 liefert immer die Konstante 0 zurück

Befehlssatz 16 Befehle
Befehlssatz:

Op-Code	Mnemonic	Bedeutung	Assembler
0	halt	halt	hlt
1	add	R[d] ← R[s] + R[t]	add RD, RS, RT
2	subtract	R[d] ← R[s] - R[t]	sub RD, RS, RT
3	and	R[d] ← R[s] & R[t]	and RD, RS, RT
4	xor	R[d] ← R[s] ^ R[t]	xor RD, RS, RT
5	shift left	R[d] ← R[s] << R[t]	shl RD, RS, RT
6	shift right	R[d] ← R[s] >> R[t]	shr RD, RS, RT
7	load immediate	R[d] ← imm	lda RD, imm
8	load	R[d] ← mem[addr]	ld RD, addr
9	store	mem[addr] ← R[d]	st RD, addr
A	load indirect	R[d] ← mem[R[t]]	ldi RD, RT
B	store indirect	mem[R[t]] ← R[d]	sti RD, ST
C	branch zero	if (R[d] == 0] pc ← addr	bz RD, addr
D	branch positive	if (R[d] > 0) pc ← addr	bp RD, addr
E	jump register	pc ← R[d]	jr RD
F	jump and link	R[d] ← pc; pc ← addr	jl RD, addr

Weiters besitzt die Toy CPU:

Programmcounter
In diesem wird die Startadresse geladen und er wird pro Zustand um 1 erhöht.

Memory Buffer
Jedes Speicherwort auf dem Weg zwischen Speicher und CPU muss durch das Register MB hindurch.

Memory Address Register
Das Register MA liefert die Adresse für Speicher und I/O.

Instruktionsregister
Hier werden alle Instruktionen gespeichert

Wenn TOY auf die Adresse 255 (= hex FF) schreibt, so nennen wir diesen Vorgang „Schreiben auf Standard-Output“ (stdout). In ähnlicher Weise nennen wir das Lesen von der Adresse (hex) FF „Lesen von stdin“.

Kapitel 3
Testprogramme

Die Testprogramme wurden entwickelt um die in Verilog entwickelte TOY CPU zu testen:

test1

Sourcecodeauszug	ausserlechner154t1.asm

```
    ld      RA, A                   ;RA = 3
    ld      RB, B                   ;RB = 2
    ld  RC, C                       ;RC = 0
    add RA, RA, RB              ;RA = 5
    sub RB, RA, RB              ;RB = 3
    shl RC, RA, RB              ;RC = 40 od 0x28
    st  RC, 0xFF        ;print value of RC
    hlt
A           DW    3
B           DW    2
C           DW    0
```

test2

Sourcecodeauszug	ausserlechner154t2.asm

```
            ld          RA, A                   ;RA = 3
            lda         RB, 0x04                ;RB = 4
            xor R1, RA, RB                      ;R1 = 7
            lda RC, 0x0A                        ;RC = 10
            and R2, R1, RC                      ;R2 = 2
            lda RD, 0x01                        ;RD = 1
            shr R3, R2, RD                      ;R3 = 1
            shl R3, R3, RD                      ;R3 = 2
            ld          RE, E                   ;RE = 7
            sub R2, R2, RD                      ;R2 = 1
loop        sub RE, RE, RD                      ;decrement RE
            shl R3, R3, R2                      ;shift bits of R3 1 to the left
            bp          RE, loop                ;if RE is positiv goto label loop
            sub R3, R3, RD                      ;R3 = 255
            sti R3, R3          ;print mem(0xFF) = 255
            hlt
A           DW    3
E     DW 7
```

test3

Sourcecodeauszug	ausserlechner154t3.asm
ld RA, A	;RA = 16
ld RB, B	;RB = 2
ldi RC, RA	;RA = 0x8A00
lda RC, loopi	
loopi jl, RD, func	;jump to label func
bz RA, exit	;if RA is zero goto exit
jr RC	;jump to label loopi
func shr RA, RA, RB	;shift bits of RA, RB times to the right
st RA, 0xFF	;print val of RA
jr RD	;jump to label loopi + 1
exit hlt	
A DW 16	
B DW 2	

test4

Sourcecodeauszug	ausserlechner154t3.asm
ld R1, A	;8100 // Reg(1) <- mem(00)
ld R2, B	;8201 // Reg(2) <- mem(01)
shl R2, R2, R1	;5221 // Reg(2) <- Reg(2) << Reg(1)
st R2, 0x03	;9203 // mem(03) <- Reg(2)
ld R3, C	;8302 // Reg(3) <- mem(02)
shr R2, R2, R3	;6223 // Reg(2) <- Reg(2) >> Reg(3)
st R2, 0x04	;9204 // mem(04) <- Reg(2)
hlt	;0000 // halt
A DW 0x0001	;00
B DW 0x0002	;01
C DW 0x0001	;02

test5

Sourcecodeauszug	ausserlechner154t3.asm

```
        lda R1, 0xFF                ;71ff   // Reg(1) <- imm(ff)        load imm
        ld  R2, A                   ;8200   // Reg(2) <- mem(00)        load
        ld  R3, A                   ;8302   // Reg(3) <- mem(02)        load indirect
        ldi R4, R3                  ;A403   // Reg(4) <- mem(R(3))
        add R2, R1, R2              ;1212   // Reg(2) <- Reg(1) + Reg(2)   add
        st R2, 0xFF                 ;9203   // mem(3) <- Reg(2)         store
        sub R2, R2, R1              ;2221   // Reg(2) <- Reg(2) - Reg(1)   subtract
        ld  R4,C                    ;8402   // Reg(4) <- mem(02)        store indirect
        sti R2, R4                  ;B204   // mem(R(4)) <- Reg(2)
call_test5_2    lda RF, test5_2     ;7F25   // Reg(F) <- imm(25)        jmp test5_2
jr RF                               ;EF00   // pc <- Reg(F)

call_test5_3    lda RF, test5_3     ;7F40   // Reg(F) <- imm(40)        jmp test5_3
jr RF                               ;EF00   // pc <- Reg(F)

call_halt       jl RF, halt         ;FF50   // Reg(F) <- pc; pc <- halt     jmp halt

test5_2         ld R1, D            ;8105   // Reg(1) <- mem(05)        ld AAAA (1010 1010 1010)
ld  R2, A                           ;8200   // Reg(2) <- mem(00)        ld 0001
ld  R6, E                           ;8606   // Reg(6) <- mem(06)        ld 5555 (0101 0101 0101)

xor R7, R6, R1                      ;4761   // Reg(7) <- Reg(6) ^ Reg(1)   xor
shr  R1, R1, R2                     ;5112   // Reg(1) <- Reg(1) << Reg(2)  shift left
and R6, R6, R1                      ;3661   // Reg(6) <- Reg(6) & Reg(1)   and
lda RF, call_test5_3                ;7F21   // Reg(F) <- imm(21)        ret test5_2
jr  RF                              ;EF00   // pc <- Reg(F)

test5_3         ld R1, F            ;8107   // Reg(1) <- mem(07)        ld 0005
ld  R2, G                           ;8208   // Reg(2) <- mem(08)        ld 0001
subtract  sub R1, R1, R2            ;2112   // Reg(1) <- Reg(1) - Reg(2)   sub Reg(1) - Reg(2)
bz  R1,call_halt                    ;C123   // if (Reg(1) == 0) pc <- 40   if (Reg(1) == 0) jmp halt
bp  R1, subtract                    ;D142   // if (Reg(1) > 0) pc <- 28    if (Reg(1) > 0) jmp sub

halt hlt                    ;0000   // halt
A    DW 0x0001              ;00
B    DW 0x000F              ;01
C    DW 0x000F              ;02
D    DW 0xAAAA              ;05
E    DW 0x5555              ;06
F    DW 0x0005              ;07
G    DW 0x0001              ;08
H    DW 0xFFFF              ;0f
```

Kapitel 4
Funktionales Modell

Das funktionale Modell besteht aus folgenden Modulen:

- top module
- toy module
- io module
- mem module
- toy_cpu module

4.1 top module

Im top module steht der Bootloader, welcher die vom stdin.dat eingelesenen Daten, überprüft und dann auf das MEMory speichert.

Sourcecodeauszug 1	ausserlechner154f.v
toy_i.memory.m['h0000] = 'hCAFE;	//Hier wird auf die MEM address 0000 CAFÉ geschrieben
toy_i.memory.m['h0010] = 'hFFE0;	// jl RF, start
toy_i.memory.m['h00E0] = 'h7101;	// In Reg 1 wird Konstante 1 geschrieben
toy_i.memory.m['h00E1] = 'h8200;	// Wert von MEM address 0000 wir auf Reg 2 geladen
toy_i.memory.m['h00E2] = 'h83FF;	// Auf Reg 3 wird 1 Wort von stdin.dat eingelesen
toy_i.memory.m['h00E3] = 'h2332;	// Es wird auf CAFÉ überprüft
toy_i.memory.m['h00E4] = 'hC3E6;	// Wenn CAFE einlesen starten
toy_i.memory.m['h00E5] = 'h0000;	// Beenden
toy_i.memory.m['h00E6] = 'h84FF;	// start addr in R4
toy_i.memory.m['h00E7] = 'h1540;	// temp copy of start addr
toy_i.memory.m['h00E8] = 'h86FF;	// amount of words
toy_i.memory.m['h00E9] = 'hC6EF;	// goto exit if finished
toy_i.memory.m['h00EA] = 'h2661;	// decrement R6
toy_i.memory.m['h00EB] = 'h87FF;	// read a word
toy_i.memory.m['h00EC] = 'hB705;	
toy_i.memory.m['h00ED] = 'h1551;	// increment code address
toy_i.memory.m['h00EE] = 'hFFE9;	
toy_i.memory.m['h00EF] = 'hE400;	// jump to program's start addr
toy_i.memory.m['h00F0] = 'h0000;	// Beenden (sollte nicht erreicht werden)

Des Weiteren wird hier ein TOY Gesamtmodell mit MEM, I/O und CPU instanziert.
Dies wird über folgende Syntax realisiert:

```
toy toy_i(clk, continue);
```

In dem module wird ein Clock generiert:

```
initial   clk = 0;
always  #50 clk = ~clk;
```

(Nach 50 Zeiteinheiten wird der clk invertiert)

Ein Continue Signal startet die CPU bzw. führt fort wenn in den `IDLE Zustand gesprungen wurde (Aufgrund eines halt).

Des Weiteren werden noch für das debugging die Registerdatenbank bzw. das MEMory ausgegeben.

4.2 toy module

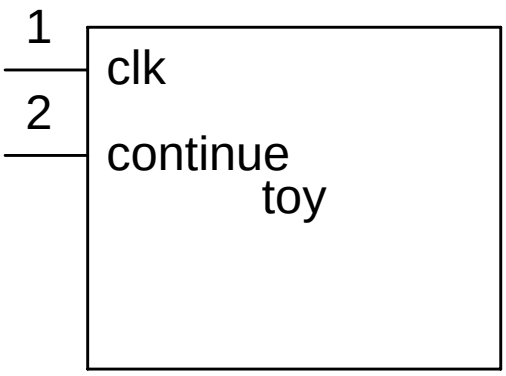


Abbildung 2. toy Blockschaltbild

Eingänge:

clk	Takt
continue	Dieses Signal wird für das toy_cpu module übergeben

Im toy module werden folgende module instanziert:

toy_cpu:

toy_cpu cpu(clk, continue, read, write, addr, toy_din, toy_dout);

mem:

mem memory(clk, write_mem, addr, toy_dout, mem_dout);

io:

io in_out(clk, read_io, write_io, toy_dout, io_dout);

Des Weiteren werden folgende Operationen durchgeführt:

Sourcecodeauszug 2	ausserlechner154f.v

```
assign write_io  = write & (addr == 16'hFF);
assign write_mem = write & (addr != 16'hFF);
assign read_io   = read  & (addr == 16'hFF);

assign toy_din = (addr == 16'hFF) ? io_dout : mem_dout;
```

Hier wird das flag write_io gesetzt wenn write high ist und die address 16'hff übergeben wird.
Es wird auf den das io module rausgeschrieben.
Bei write_mem funktioniert dies gleich nur diesmal wird auf das MEMory geschrieben, da die address ungleich 16'hff ist.
Read_io ist genau das Gegenstück zu write_io. Hier wird vom io module gelesen, dies aber nur wenn die address 16'hff ist.

Die letze Anweisung sind dann die Daten die weitergeschrieben werden. Bei address 16'hff wird auf io_dout geschrieben, bei einer anderen address wird auf das MEMory mit der übergebenen address geschrieben.

Der assign Operator schreibt bei jeder Änderung der rechts neben dem = Zeichen stehenden Signalen sofort die Änderung auf die links stehende Variable.

4.3 io module

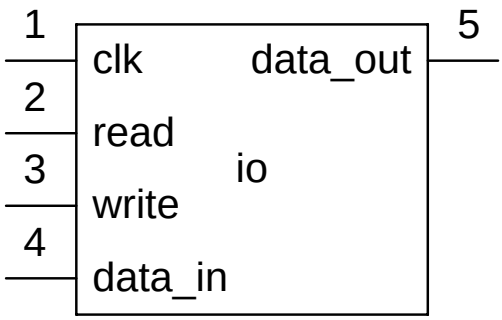


Abbildung 3. io module Blockschaltbild

Eingänge:

clk:	Takt
read:	flag, wenn gesetzt dann wird ein Wert von stdin.dat eingelesen.
write:	flag, wenn gesetzt, dann werden die Daten die an data_in anliegen auf die Datei stdout.dat hinausgeschrieben.
data_in [16bit]:	Dateneingang, Daten die auf stdout rausgeschrieben werden

Ausgänge:

data_out[16bit]:	Datenausgang, Daten die von stdin eingelesen wurden und nun an die CPU für eine Weiterverwendung übergeben werden.

Einlesen der Daten über ein std_in.dat File in das std_in Array

```
$readmemh("std_in.dat", std_in);
```

In dem module ist eine Ausgabe des std_in Arrays für ein besseres debugging vorhanden.

Sourcecodeauszug 3	ausserlechner154f.v

```
assign data_out = std_in[std_in_pointer];
std_in_pointer = 0;

always @(posedge clk)
    read_delayed = read;

assign read_strobe = (read == 0) && (read_delayed == 1);
```

```
always @(posedge clk)
    if (read_strobe == 1)
        std_in_pointer = std_in_pointer + 1;

always @(posedge clk)
    if (write == 1)
        begin
            $fdisplay(std_out_handle, "%h", data_in);
        end
```

Bei jeder Taktflanke wird read auf read_delayed gespeichert.
Danach wird es so geregelt, dass immer der std_in_pointer um 1 erhöht wird, wenn ein Speicherwort von der Datei std_in.dat eingelesen wurde.

Immer wenn write high wird wird der Dateneingang data_in auf das Outputfile std_out.dat geschrieben.

4.4 mem module

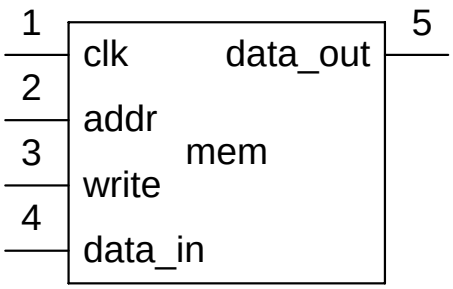


Abbildung 4. mem module Blockschaltbild

Eingänge:

clk:	Takt
addr[8bit]:	Addresse, gibt an an welcher Stelle im MEMory der gewünschte Wert geschrieben oder gelesen werden soll.
write:	flag, welches wenn gesetzt Daten auf eine Speicherstelle im MEMory schreibt.
data_in[16bit]:	Dateneingang, hier werden die Daten übergeben welche auf das MEMory geschrieben werden sollten.

Ausgänge:

data_out[16bit]: Datenausgang, hier werden die Daten ausgegeben welche auf einer Speicherstelle stehen, welche über die Addresse gefunden werden.

Sourcecodeauszug 4	ausserlechner154f.v

```
always @(addr)
begin
 data_out = m[addr];
end

always @(posedge clk)
 if (write == 1)
    begin
     m[addr] = data_in;
     data_out = data_in;
    end
```

Bei jeder übergebenen Addresseänderung wird auf den Datenausgang data_out der Wert der Speicherstelle von der address ausgegeben.

Wenn das flag write auf 1 gesetzt wurde, werden die Daten vom Dateneingang data_in auf die Speicherstelle der übergebenen Addresse gespeichert. Zusätzlich wird der Dateneingang am Datenausgang ausgegeben.

4.5 toy_cpu module

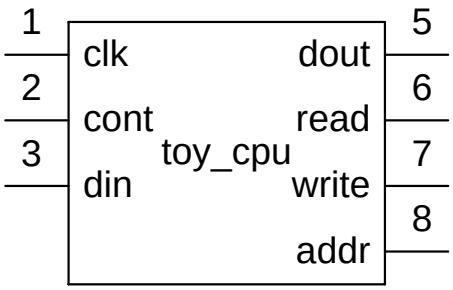


Abbildung 5. toy_cpu module Blockschaltbild

Eingänge:

clk: Takt

cont: flag, welches von top module gesetz wird um die CPU nach einem halt fortzusetzen.

din[16bit]: Dateneingang, hier werden die Daten zum Berechnen eingelesen(vom IO oder vom MEMory).

Ausgänge:

dout[16bit]:	Datenausgang, die berechneten Daten werden wieder gespeichert (IO oder MEMory).
read:	flag, dass gesetzt wird wenn Leseoperationen anstehen.
write:	flag, dass gesetzt wird wenn Schreibeoperationen anstehen.
addr[8bit]:	Adresse für Speicherzugriff oder wenn FF dann IO Zugriff.

Wichtige Variablen:

reg [15:0] register [15:0];	Registerdatenbank (16 * 16bit)
reg [7:0] pc;	Programmcounter (8bit]
reg [7:0] ma;	Memory Address Register (8bit)
reg [15:0] mb;	Memory Buffer Register (16bit)

Zustände:

Sourcecodeauszug 5 — ausserlechner154f.v

```
@(posedge clk) enter_new_state(`INIT);
    pc <= @(posedge clk) 8'h10;
    halt <= @(posedge clk) 1;
    while (1)
     begin
      @(posedge clk) enter_new_state(`FETCH1);
       ma <= @(posedge clk) pc;
       if (halt)
         begin
          while (~cont)
            begin
              @(posedge clk) enter_new_state(`IDLE);
              halt <= @(posedge clk) 0;
            end
         end
       else
         begin
           @(posedge clk) enter_new_state(`FETCH2);
           mb <= @(posedge clk) din;
           @(posedge clk) enter_new_state(`FETCH3);
           ir <= @(posedge clk) mb;
           @(posedge clk) enter_new_state(`DECODE);
           pc <= @(posedge clk) pc + 1;
```

Zu Beginn startet das toy_cpu module im ´INIT Zustand wobei der Programm Counter auf 8'h10 gesetzt wird. Das flag halt wird auch auf 1 gesetzt, woduch die CPU nach der Initialisierung in den Zustand ´IDLE springt.
Im Zustand ´FETCH1 wird der pc auf das Memory Address Register ma geschrieben.

Danach wird auf das halt flag geprüft, wenn high wird in den Zustand ´IDLE gesprungen, wo halt wieder 0 gesetzt wird und auf continue gewartet wird.
Wenn continue high wird, dann werden die Daten vom Dateneingang din in das Memory Buffer Register mb geladen. Dieses wird dann in das Instruction Register ir geladen.

Danach beginnt die eigentliche Arbeit der CPU. Der pc wird um 1 erhöht und es wird begonnen das zu exekutierende Programm abzuarbeiten

Sourcecodeauszug 6	ausserlechner154f.v

```
opcode <= @(posedge clk) ir[15:12];
d <= @(posedge clk)     ir[11:8];
s <= @(posedge clk)     ir[7:4];
t<= @(posedge clk)     ir[3:0];
addr_ <= @(posedge clk)  ir[7:0];
```

Hier wird der eingelesene Befehl, welcher im ir steht in 4 Abschnitte aufgeteilt:

opcode	sagt was die CPU tun soll
d	Register in das geschrieben werden soll
s	Register welches ALU Funktion ausführen soll
t	Register welches ALU Funktion ausführen soll
addr_	Addresse

Nun wird in einer Case Schleife nach den verschiedenen opcodes abgefragt:

Sourcecodeauszug 7	ausserlechner154f.v

```
`ADD:
begin
  @(posedge clk) enter_new_state(`ADD);
   register[d] <= @(posedge clk) register[s] + register[t];
end
```

Die ALU Funktionen sind alle ziemlich ähnlich, des wegen wir hier nur add erklärt:

In das Register des übergebenen Wertes d wird bei der nächsten positiven Taktflanke die Werte der Register mit den übergebenen Werten s und t gespeichert.

Bei den anderen ALU Funktionen sieht dies gleich aus, nur die Operationen sind

- Subtrahieren
- AND verknüpfen
- XOR verknüpfen
- Nach rechts schieben
- Nach links schieben

Weiters gibt es noch die MEMory bezogenen Operationen wie:

´LDA

Sourcecodeauszug 8	ausserlechner154f.v
register[d] <= @(posedge clk) addr_;	

Hier wird ein hexadezimaler 8 bit Wert auf ein Register mit dem übergebenen Wert d geschrieben.

´LD

Sourcecodeauszug 9	ausserlechner154f.v

@(posedge clk) enter_new_state(`LD);
ma <= @(posedge clk) addr_;
@(posedge clk)
read = 1;
@(posedge clk)
mb <= @(posedge clk) din;
@(posedge clk)
register[d] <= @(posedge clk) mb;	

Die Adresse addr_ wird in das Memory Address Register ma geschrieben. Das flag read wird gesetzt. Die Daten vom Dateneingang din werden in den Memorybuffer geladen und danach auf das gewählter Register gepeichert.

´ST

Sourcecodeauszug 10	ausserlechner154f.v

@(posedge clk) enter_new_state(`ST);
mb <= @(posedge clk) register[d];
ma <= @(posedge clk) addr_;
@(posedge clk)
write = 1;	

Store ist beinahe gleich wie ´LD. Hier wird zuerst der Wert vom register [d] in den mb und die addr_in den ma gespeichert und danach wird das flag write auf high gesetzt, sodass auf das MEMory geschrieben werden kann.

´LDI

Load indirect sieht fast gleich wie `LD aus. Hier wird nur register [t] anstatt von addr_ als Addresse in das ma geladen.

´STI

Store indirect sieht fast gleich wie ´ST aus. Hier wird nur register [t] anstatt von addr_ als Addresse in das ma geladen.

´BZ

Sourcecodeauszug 11	ausserlechner154f.v

@(posedge clk) enter_new_state(`BZ);
if(register[d] == 0)
pc <= @(posedge clk) addr_;	

Hier wird der Inhalt des registers[d] auf null überprüft. Wenn false dann wird nichts unternommen, wenn true, dann wird der pc auf die übergebene Addresse gesetzt.

´BP

´BP ist wieder ähnlich zu ´BZ. Hier wird überprüft ob der Inhalt von register[d] >0 ist. Ansonsten verhält es sich wie `BZ.

´JR

Sourcecodeauszug 12	ausserlechner154f.v
@(posedge clk) enter_new_state(`JR); pc <= @(posedge clk) register[d];	

Hier wird auf denpc die vom register[d] übergebenen Addresse gesprungen

´JL

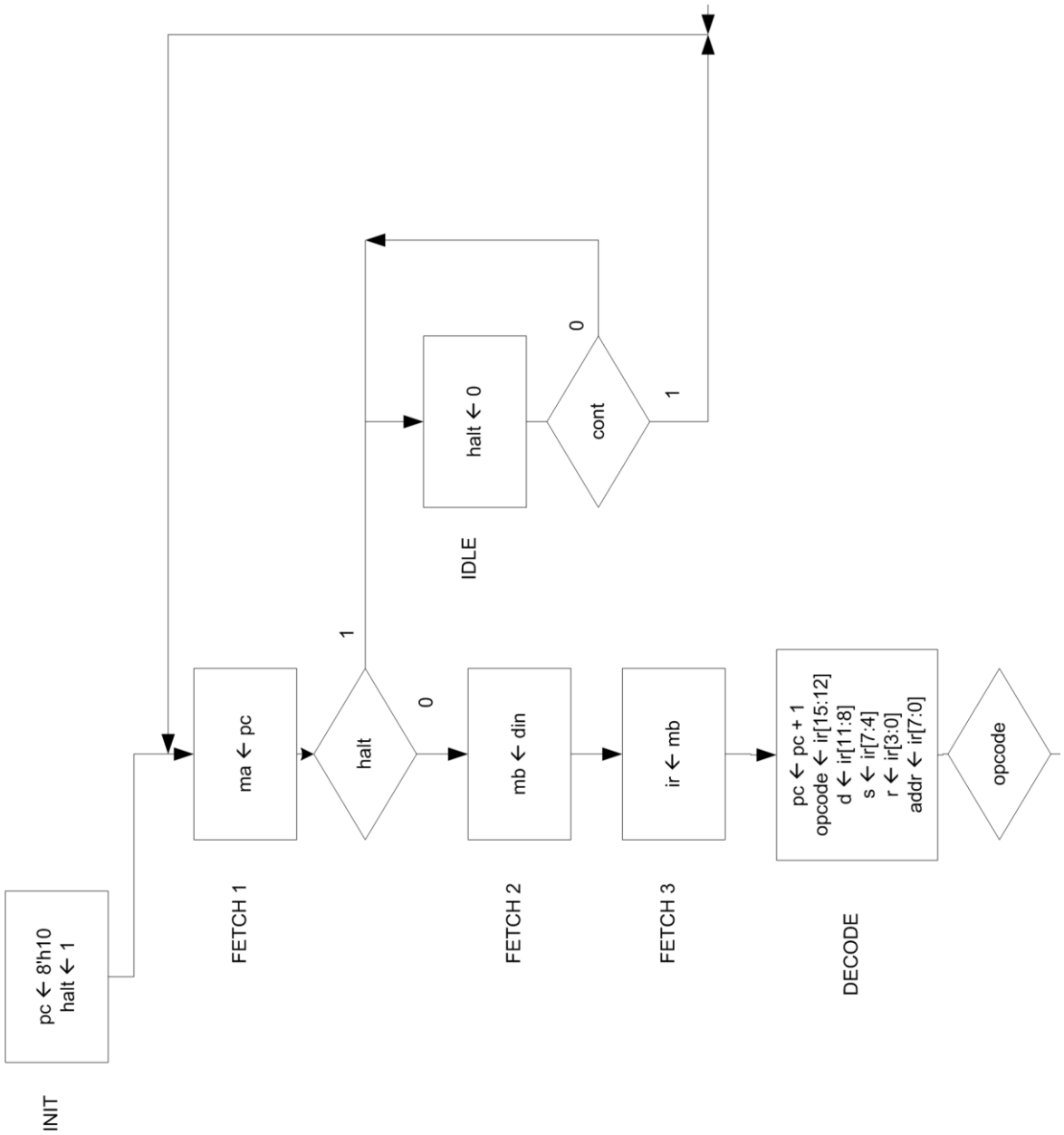
Sourcecodeauszug 13	ausserlechner154f.v

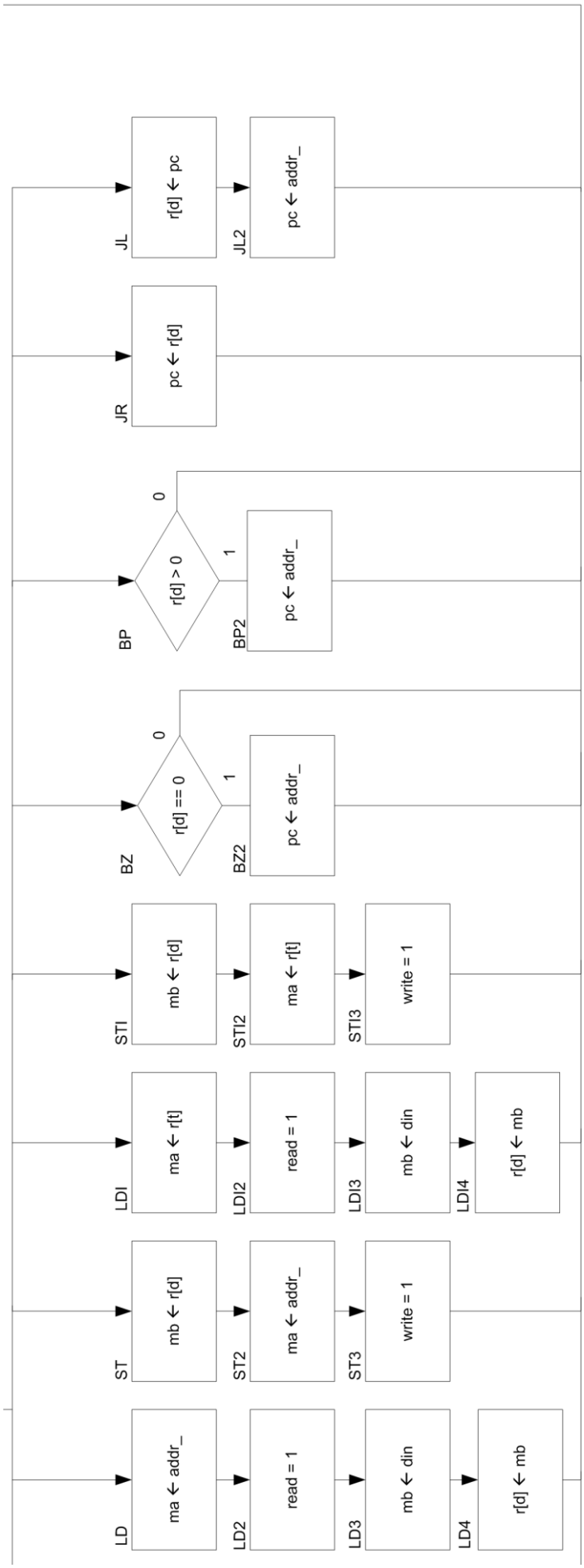
@(posedge clk) enter_new_state(`JL);
register[d] <= @(posedge clk) pc;
@(posedge clk)
pc <= @(posedge clk) addr_;	

Hier wird der pc auf register[d] gesichert. Danach wird der pc auf den Wert der Addresse addr_ gesetzt.

Bei jedem Task enter_new_state werden die flags read und write auf null zurückgesetzt und zusätzlich das register[0] auf null gesetzt.

Flussdiagramm des toy_cpu modules:


pc ← 8'h10
halt ← 1
INIT
ma ← pc
FETCH 1
halt
1
0
mb ← din
FETCH 2
ir ← mb
FETCH 3
pc ← pc + 1
opcode ← ir[15:12]
d ← ir[11:8]
s ← ir[7:4]
r ← ir[3:0]
addr ← ir[7:0]
DECODE
opcode
halt ← 0
IDLE
cont
0
1


LD
ma ← addr_
LD2
read = 1
LD3
mb ← din
LD4
r[d] ← mb
ST
mb ← r[d]
ST2
ma ← addr_
ST3
write = 1
LDI
ma ← r[t]
LDI2
read = 1
LDI3
mb ← din
LDI4
r[d] ← mb
STI
mb ← r[d]
STI2
ma ← r[t]
STI3
write = 1
BZ
r[d] == 0
0
1
BZ2
pc ← addr_
BP
r[d] > 0
0
1
BP2
pc ← addr_
JR
pc ← r[d]
JL
r[d] ← pc
JL2
pc ← addr_

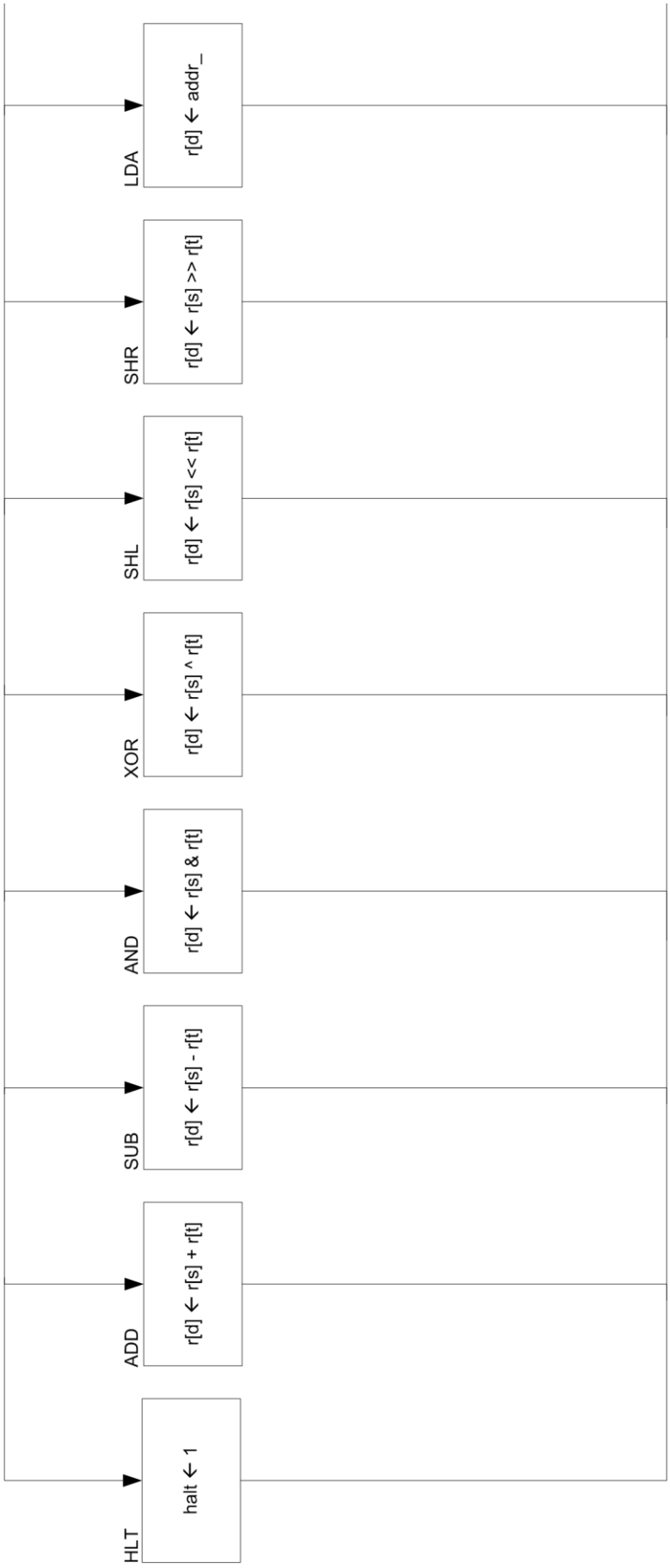


Abbildung 6. Flussdiagramm Funktionales Modell

Kapitel 5
Mixed Modell

Das Mixed Modell besteht aus folgenden Modulen:

- alu module
- register16 module
- register8 module
- register4 module
- register1 module
- register0 module
- register_dout module
- register_addr_out module
- mux16 module
- mux8 module
- muxsel module
- muxs_t module
- muxd module
- mux2to1_16 module
- mux2to1_8 module
- toy_cpu_datapath module
- toy_cpu_controller module
- mem module
- io module
- toy module
- top module

Auf die module top, toy, mem und io wird hier nicht näher eingegangen, da sie gleich bzw sehr ähnlich den modulen des Funktionalen Modells sind.

5.1 alu module

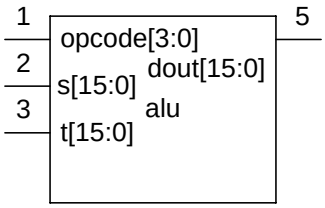


Abbildung 7. alu module Blockschaltbild

Eingänge:

opcode[4bit]	Der Befehl, mit welchem die ALU auswählt welche Operationen exekutiert werden.
s[16bit]	1. Dateneingang
t[16bit]	2. Dateneingang

Ausgänge:

dout[16bit]	Datenausgang

Die ALU ist die Recheneinheit der CPU und führt alle arithmetischen und logischen Befehle aus.

Sourcecodeauszug 8	ausserlechner154m.v

```
always @(opcode or s or t)
begin
  case(opcode)
    `ADD:
        dout = s + t;
```

Hier wird bei jeder Änderung des Signales opcode, s oder t mithilfe einer Case Abfrage der opcode überprüft und dementsprechend folgende Operationen ausgeführt:

- add
- sub
- and
- xor
- shl
- shr

Die Dateneingänge s und t werden dem Operationen entsprechend modifiziert und auf den Datenausgang dout gespeichert.

5.2 register16 module

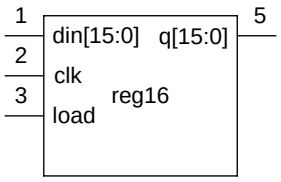


Abbildung 8. register16 module Blockschaltbild

Eingänge:

din[16bit]	Dateneingang
clk	Takt
load	flag, das angibt ob neu eingelesen oder der alte Wert ausgeben werden soll.

Ausgänge:

q[16bit]	Datenausgang

Die module register16 sind die 16 16bit Registerbank [register1-registerf]

Sourcecodeauszug 8	ausserlechner154m.v

```
always @(posedge clk)
  if(load == 1)
  begin
    q = din;
  end
```

Hier wird bei jeder positiven Taktflanke, wenn das load flag high ist, der Dateneingang auf den Datenausgang geschrieben, wenn load low ist, dann wird der zuletzt gespeicherte Wert hinaus geschrieben.

5.3 register8 module

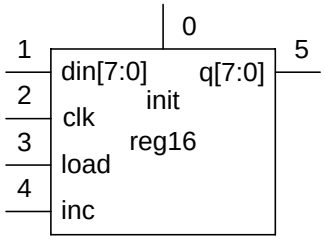


Abbildung 9. register8 module Blockschaltbild

Eingänge:

din[8bit]	Dateneingang
clk	Takt
load	flag, das angibt ob neu eingelesen oder der alte Wert ausgeben werden soll.

init	flag, das angibt ob ein fixer Zustand ausgegeben werden soll.
inc	flag, das angibt ob der Wert incrementiert werden soll.

Ausgänge:

q[8bit]	Datenausgang

Das register8 wird für den pc verwendet:

Sourcecodeauszug 8	ausserlechner154m.v

```
always @(posedge clk)
  begin
    if(load == 1)
      begin
        if(init == 1)
          q = 'h0010;
        else if(inc == 1)
          q = q + 1;
        else
          q = din;
```

Bei jeder positiven Taktflanke und wenn das flag load high ist, wird überprüft ob eines der flags(init, inc) high ist:

- init:
  Hierbei wird auf den Datenausgang der hex wert 10 geschrieben. (Startadresse)
- inc:
  Hierbei wird der pc um eins erhöht (Datenausgang +1)

5.4 register4 module

Das module register4 ist den module register16 sehr ähnlich. Daher wird hier nur die Änderung besprochen und nicht das ganze module:

Gleich wie register16 nur Dateneingang und ausgang sind 4 bit groß.

5.5 register1 module

Das module register1 ist den module register16 sehr ähnlich. Daher wird hier nur die Änderung besprochen und nicht das ganze module:

Gleich wie register16 nur Dateneingang und ausgang sind 1 bit groß. Der Datenausgang kann entweder low (0) oder high(1) werden.

5.6 register0 module

Das module register0 ist den module register16 sehr ähnlich. Daher wird hier nur die Änderung besprochen und nicht das ganze module:

Es entspricht wie das register16 der Registerdatenbank und zwar genau dem Register 0 der Registerdatenbank, welches immer 0 am Datenausgang zurückliefert.

5.7 register_dout module

Das module register_dout ist den module register16 sehr ähnlich. Daher wird hier nur die Änderung besprochen und nicht das ganze module:

Es verhält sich genau gleich wie das module register16.

5.8 register_addr_out module

Das module register_addr_out ist den module register16 sehr ähnlich. Daher wird hier nur die Änderung besprochen und nicht das ganze module:

Gleich wie register16 nur Dateneingang und ausgang sind 8 bit groß. Das Register wird als addr Register benutzt.

5.9 mux16 module

Eingänge:

in[0-15][16bit]	Dateneingang Register 0 -F
mb[16bit]	Memory Buffer
addr[8bit]	Addresse
pc[8bit]	Programm Counter
alu[16bit]	Daten vom ALU Ausgang
sel	select Eingang

Ausgänge:

data16[16bit]	Datenausgang

Bei dem module mux 16 werden die Register über den select Eingang angesprochen und je nachdem was von dem select Eingang geliefert wird (R0 –RF, MB_REG, ALU_OUT,ADDR_REG, PC_REG) werden die Werte der dementsprechenden Register auf den Datenausgang von dem Multiplexer ausgegeben.

5.10 mux8 module

Eingänge:

t[16bit]	Dateneingang Register 0 -F
addr[8bit]	Addresse
pc[8bit]	Programm Counter
sel	select Eingang

Ausgänge:

q[8bit]	Datenausgang

Bei dem module mux8 wird auf select abgeprüft und dementsprechend der Ausgang auf den pc, der addr oder dem Register [t] gelegt.

5.11 muxsel module

Eingänge:

d[4bit]	Dateneingang Register 0 -F
ld_d	flag, selectiert Registerstring
ld_mb_reg	flag, selectiert String MB_REG
ld_alu	flag, selectiert String ALU_OUT
ld_addr_reg	flag, selectiert String ADDR_REG
ld_pc_reg	flag, selectiert String PC_REG

Ausgänge:

sel_reg[8bit]	Datenausgang, selectiert die Register

Bei diesem module wird ein String an den Multiplexer m16 gesendet. Bei setzen eines der Flags (ld_d, ld_mb_reg, ld_alu, ld_addr_reg, ld_pc_reg) wird jeweils ein eigener String an m16 gesendet, welcher dann die Register auswählt.

5.12 muxs_t module

Eingänge:

r[0-F][16bit]	Dateneingang Register 0 -F
s[4bit]	Dateneingang, Wert gibt an welches Register1 selectiert werden soll.
t[4bit]	Dateneingang, Wert gibt an welches Register2 selectiert werden soll.

Ausgänge:

mux_out1[16bit]	Datenausgang, gibt gewähltes Register1 aus
mux_out2[16bit]	Datenausgang, gibt gewähltes Register2 aus

Bei diesem Multiplexer wird das Register ausgewählt welches über die Werte s und t übergeben werden. An dem jeweiligen Datenausgang steht nun der Wert des ausgewählten Registers.

5.13 muxd module

Eingänge:

d[4bit]	Dateneingang, Wert gibt an welches Register selectiert werden soll.
ld_d_reg	flag, multiplexer aktiviert

Ausgänge:

ld_[0-F][16bit]	Datenausgang, gibt gewähltes Register1 aus

Wenn das flag ld_d_reg high ist und über den Dateneingang d ein Wert hereinkommt wird auf den Datenausgang ld_[0-F] das jeweilige Register ausgewählt in welches dann die Werte aus den ALU Operationen bzw den Speicherzugriffen gespeichert wird.

5.14 mux2to1_16 module

Eingänge:

din[16bit]	Dateneingang von der CPU
data16[16bit]	Dateneingang, Rückgekoppeltes Signal
ld_d_reg	flag,gibt an ob neue Daten geladen werden oder rückgekoppelt wird.

Ausgänge:

mux2_1_out[16bit]	Datenausgang, gibt dementsprechende Daten weiter.

Bei diesem Multiplexer wird ausgewählt, ob die Daten vom Dateneingang din der CPU oder ein Rückgekoppeltes Signal weitergeleitet werden. Die Selektierung erfolgt über das flag ld_d_reg.

5.15 mux2to1_8 module

Eingänge:

addr[8bit]	Addresse
data16[16bit]	Dateneingang, Rückgekoppeltes Signal
sel_pc[8bit]	select Eingang

Ausgänge:

mux_addr[8bit]	Datenausgang, gibt dementsprechende Daten weiter.

Bei diesem Multiplexer wird von außen der Selectwert abgerufen. Wenn sel_pc high dann wird die addr auf den Augang geschrieben. Wenn low, dann werden die Daten rausgeschrieben.

5.16 toy_cpu_datapath module

Die Eingänge und Ausgänge des Datenpfades werden hier nicht aufgelistet, da dies nicht der Übersicht dient. Vielmehr wollen wir hier die Funktion des Datenpfades erläutern:

Im Datenpfad werden die ganzen Module welche im Schaltplan Mixed Modell ersichtlich sind instanziert. Hier ein kleines Codebsp:

Sourcecodeauszug 8	ausserlechner154m.v

```
register16 r_mb(clk, ld_mb, mux2_1_out, mb);
register16 r_ir(clk, ld_ir, mb , ir);
```

Hie wird der Memory Buffer Register und das Instruktionsregister instanziert.

Des Weiteren werden die ganzen module mit wires verdrahtet.

5.17 toy_cpu_controller

Die Eingänge und Ausgänge des Controllers werden hier nicht aufgelistet, da dies nicht der Übersicht dient. Vielmehr wollen wir hier die Funktion des Controllers erläutern:

Der Controller sieht dem Funktionalen Modell des modules toy_cpu sehr ähnlich. Beim Mixed Modell werden alle Register Transfer Anweisungen durch normale Anweisungen ersetzt.
Codebeispiel ADD:

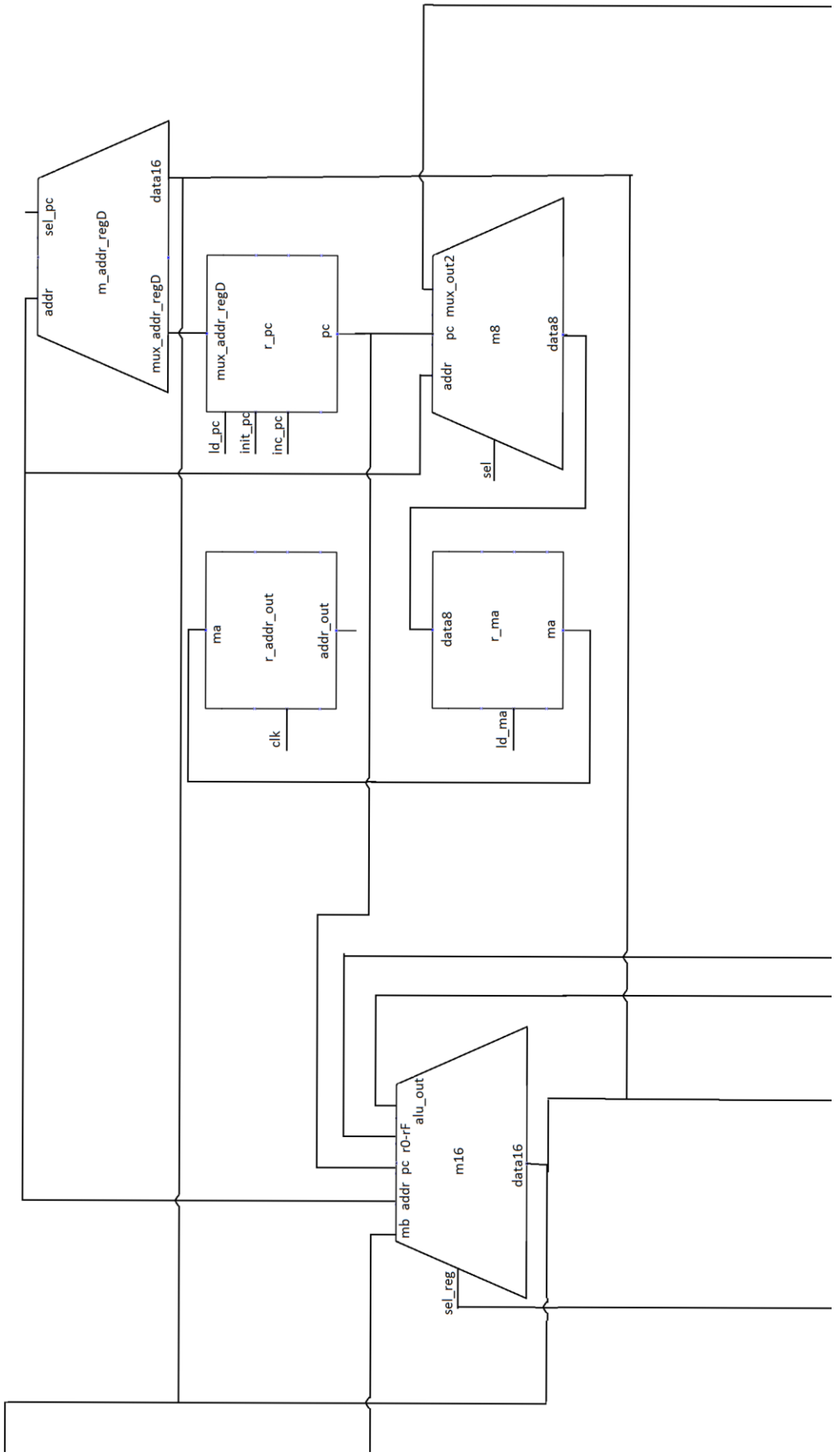
Sourcecodeauszug 8	ausserlechner154m.v

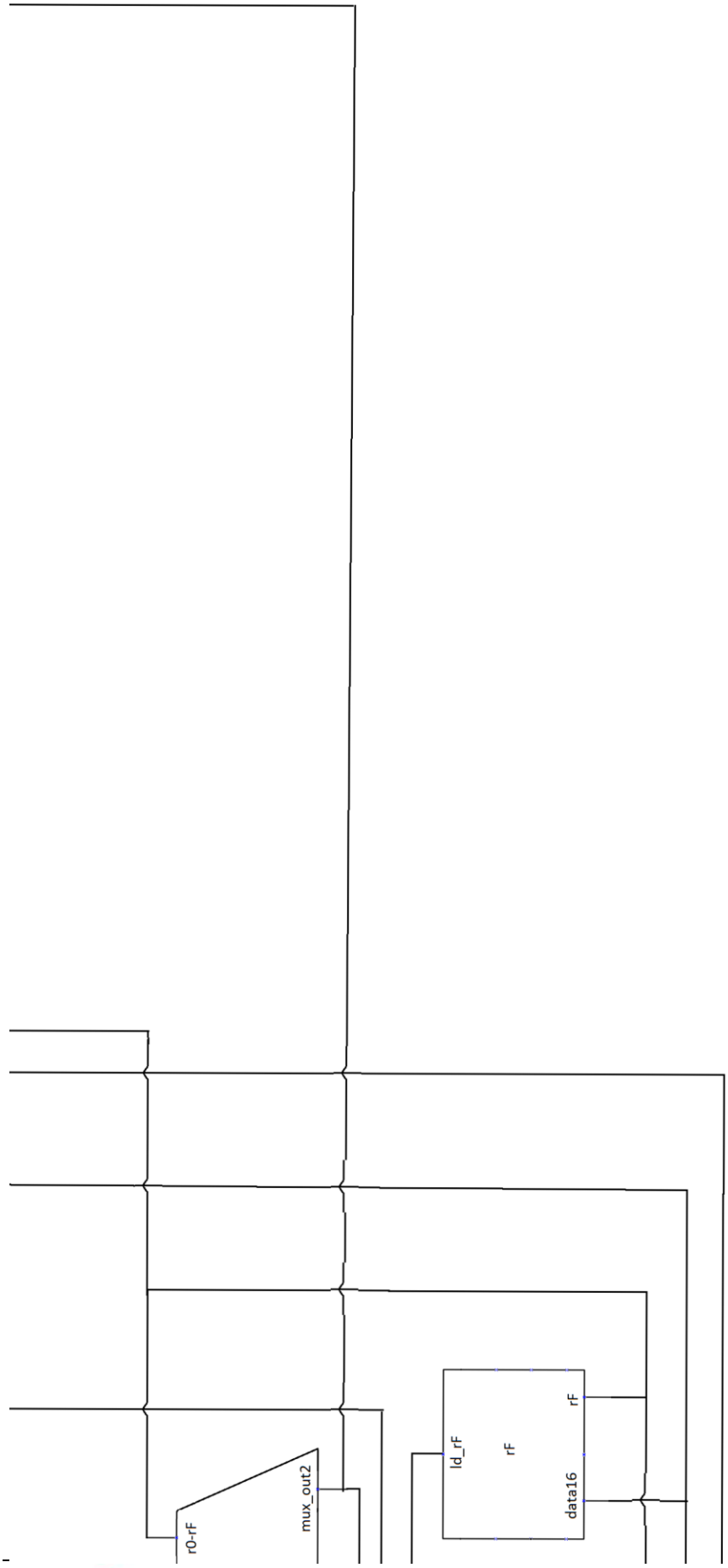
```
`ADD:
begin
   @(posedge clk) enter_new_state(`ADD);
    $write("add...\n");
    ld_d_reg = 1;
    ld_alu = 1;
end
```

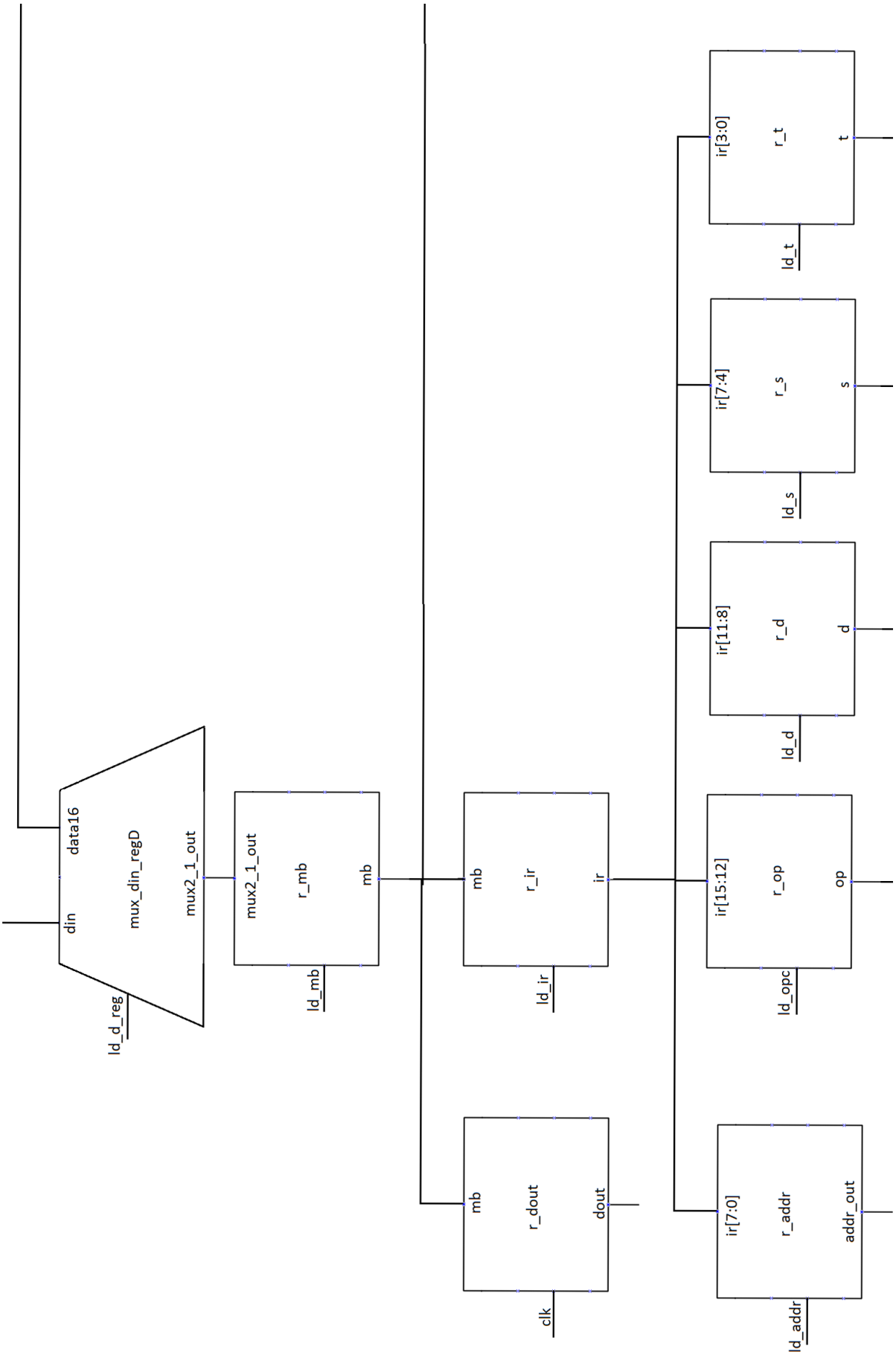
Hier wurden alle RTL Anweisungen ersetzt.

Der Controller steuert die Datensignale für den Datenpfad. Die 2 module Datenpfad und Controller sind in dem module toy über wires verkabelt. So werden dann alle Signale vom Controller auf den Datenpfad weitergeleitet und können dort die Register bzw die Multiplexer setzen oder rücksetzen.

Blockschaltbild des Gemischten Modells:


sel_pc
addr
m_addr_regD
data16
mux_addr_regD
mux_addr_regD
r_pc
pc
ld_pc
init_pc
inc_pc
pc
mux_out2
addr
m8
data8
sel
ma
r_addr_out
addr_out
clk
data8
r_ma
ma
ld_ma
alu_out
r0-rF
mb
addr
pc
m16
data16
sel_reg


r0-rF
mux_out2
ld_rF
rF
rF
data16


data16
din
mux_din_regD
mux2_1_out
ld_d_reg
mux2_1_out
r_mb
mb
ld_mb
mb
r_ir
ir
ld_ir
mb
r_dout
dout
clk
ir[15:12]
r_op
op
ld_opc
ir[11:8]
r_d
d
ld_d
ir[7:4]
r_s
s
ld_s
ir[3:0]
r_t
t
ld_t
ir[7:0]
r_addr
addr_out
ld_addr

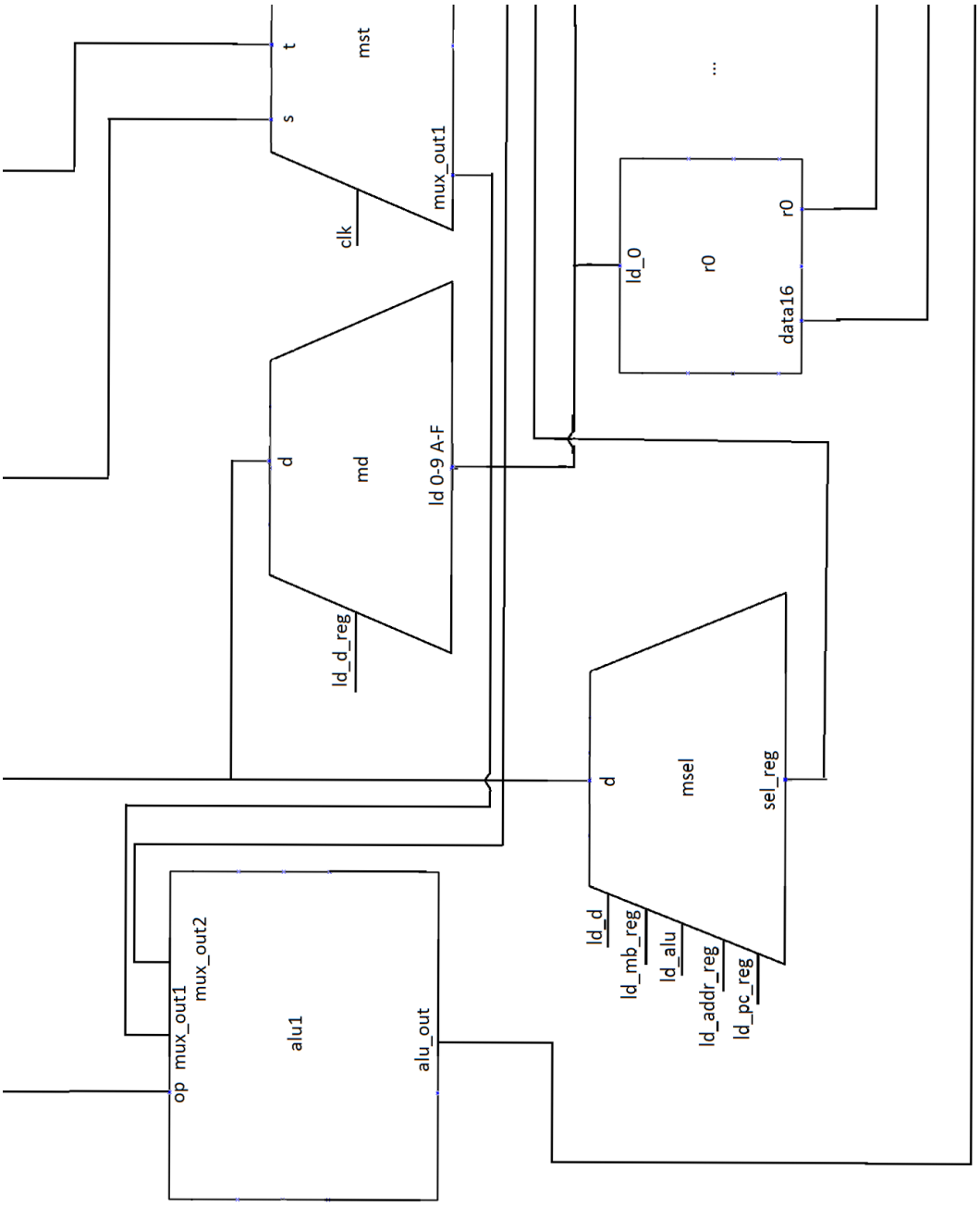


Abbildung 7. Schaltplan Mixed Modell

Kapitel 6
Beurteilung

1. Arbeitaufwand

   Arbeitsaufwand nach Aufzeichnungen laut Journal:

durchschnittliche Arbeit:	~51,5h
Functional Model:	~16h
Mixed Model:	~17h
Dokumentation:	~9h
Testprogramme:	~3h
Sontiges:	~6,5h

2. Schwierigkeiten

   Es traten vor allem vor dem Beginn der Programmierung einige Verständnisprobleme bezüglich was genau zu tun ist und wie man die Probleme lösen sollte.
   Aus der Aufgabenstellung ist beim erstmaligen durchlesen nicht die ganze Aufgabenstellung ersichtlich und man muss sich die Aufgabenstellung mindestens 2 mal und wenn Fragen auftreten noch einmal nachsehen was man lösen muss.

   Des weiteren traten Probleme beim Einbinden des Bootloaders auf. Wenig bis gar nicht erklärt wurde wie die Daten im std_in File schlussendlich auszusehen zu haben. Deswegen wurde wertvolle Zeit verbraucht um alle Möglichkeiten durchzutesten wie die Daten nun schlussendlich im std_in. dat zu stehen zu haben, sodass Verilog keine Probleme mehr hat und die Daten richtig in das MEMory eingelesen werden können.

   Unter anderem hat es Probleme mit der Programmierumgebung gegeben:
   Zum einen ist die Evaluation Version bei der Mixed Model Programmierung abgelaufen und es musst wieder für Windows und Linux Lösungen gefunden werden.
   Unter Windows traten dann immer wieder Probleme mit Verilogger auf so wie wenn in einer $write anweisung kein Zeilenumbruch definiert war der Verilogger sich aufgehängt hat.
   Unter Linux hatten wir auch ein Problem beim Mixed Model.
   Unter Windows konnte nicht auf das Outputfile stdout .dat geschrieben werden, unter Linux schon.

3. Verbesserungsvorschläge

   Verbesserungsvorschläge wären zb. nicht die ersten 3 Assignments so leicht zu Beginnen und immer alle Dateien anzubieten und zu letzt dann ins kalte Wasser schmeisen und dann als letzes dann ein Beispiel geben das in der Komplexität die andere 3 Bsp so stark in den Schatten stellt.
   Meiner Meinung ist es besser zum Schluss nicht so ein schweres Bsp zu geben sondern durchgehend Bsp die herausfordernd sind, aber mit ein wenig Fleiß lösbar.

   Was die Aufgabenstellung betrifft hätte ich ein wenig mehr Informationen bezüglich des einlesen der Daten aus den std_in.dat benötigt. Es war nicht so ersichtlich wie die Daten nun im stdin.dat gespeichert werden sollten, daher wäre ein Lösungsvorschlag von mir ein Testprogramm auch von den Tutoren zur Verfügung stellen an den man sich anhalten bzw. mit dem man auch testen kann.

   Zu den Programmen wäre vllcht ein zuverlässiges Programm bzw Bugfixes und eine Lösung zu den exeeded Lines auf den RO Web nicht schlecht. Bzw eine empfohlenes Betriebssystem wobei es dann zu keinen Problemen kommen kann.

Anhang
Entwicklungsumgebung

- ***Betriebssystem –***
  Windows 7
  Linux Ubuntu
- ***Textbearbeitung –***
  Microsoft Word, 2007 Enterprise Edition
  Adobe Acrobat, 8.0
- ***Bildbearbeitung –***
  Microsoft Paint
- ***Bildererstellung –***
  Microsoft Visio, 2007 Professional Edition
- ***Programmierumgebung –***
  Verilogger 8.1
  Notepad++

Anhang
Quellenverzeichnis

Aufgabenstellung:

RO_KU_2010.pdf Seite 103 – 107 Autor:: Karl C Posch
URL:http://www.iaik.tugraz.at/content/teaching/bachelor_courses/rechnerorganisation/practicals/downloads/files/RO_KU_2010.pdf

Quellcode:

Bootloader copy_in_to_out.v Autor:: Karl C Posch
mem module
io module

Anhang

Abbildungsverzeichnis

Anhang

Abkürzungsverzeichnis

CPU	*… Central Prozessing Unit*
PC	*… Program Counter*
MB	*… Memory Buffer*
MA	*… Memory Address Register*
IR	*… Instruction Register*
ALU	*… Arithmetic Logic Unit*