(Daten zu „stc“)

add %r31,0,%r31

(Daten zu „ld“)

add %r31,0,%r31

sub %r31,0,%r31

stc %c1,[%r5+1]

add %r31,0,%r31

(BHOLD gesetzt)

(BHOLD gesetzt)

(BHOLD gesetzt)

(Daten zu „stc“)

```
crst %c1

add %r31,0,%r31

sub %r31,0,%r31

sub ...

sub ...

sub ...

ld [%r31],%r1

cldc -73,%c0

cldc 6,%c1
```

stc ...

stc ...

add %r31,0,%r31

add ...

add ...

add ...

sub %r31,0,%r31

(Daten zu „stc“)

(Daten zu „stc“)

cldc -73,%c1

csel RD+1

add %r31,0,%r31

crst %c0

cmove %c0,%c0

stc %c1,[%r5+1]

stc ...

stc ...

stc ...

add %r31,0,%r31

stc %c0,[%r5+1]

add %r31,0,%r31

sub%r31,0,%r31

(Daten zu „stc“)

(Daten zu „stc“)

add %r31,0,%r31

csel WR+RESET+1

cmove %c0,%c0

```
add %r31,0,%r31

sub %r31,0,%r31

csel WR+RD+R+0

cldc 170,%c0

ldc [%r5],%c1

add %r31,0,%r31

sub %r31,0,%r31

(Daten für „ldc“)
```

D. Ergebnisse der Simulation

Auf den folgenden Seiten ist das aus der Simulation ermittelte Zeitverhalten der SFU für das in Abschnitt 3.5.1 vorgestellte Testprogramm abgebildet. Zur besseren Lesbarkeit sind neben den Diagrammen die Befehle abgedruckt, welche auf dem Datenbus D(31:0) liegen.

Die Zustände der aus mehreren Bits bestehenden Signale werden in sedezimaler Schreibweise angegeben. Sind Teile des Signals undefiniert, wird vom Simulator das gesamte Signal als Grauraster dargestellt. Daher sind Teile des SFU-Datenbus („RD“) als undefiniert gekennzeichnet, obwohl sie gültige Daten enthalten. Das kommt daher, daß sich die Breite des gültigen Datenwortes nach der Breite des SFHW-Ausgangs richtet, welche auch kleiner als 32 Bit sein kann. Bei der Darstellung auf dem Bildschirm werden jedoch die gültigen Daten angezeigt und konnten somit überprüft werden.

```
wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';

wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';

wait;

end process;

end behavior;
```

SFU_config

```
-------------------------------------------------------------------------------
--
-- Name:  SFU_config
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- Configurations of the SFU descriptions for simulating it with the
-- Synopsys tool „vhdldbx".
--
-------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 17-6-94
--
-------------------------------------------------------------------------------


configuration SFU_config of SFU is

for structure

 for control: SFU_control
                   use entity WORK.SFU_Control(structure);
 end for;

end for;

end SFU_config;



configuration SFU_test_config of SFU_test is

for behavior

 for test_SFU: SFU
                   use entity WORK.SFU(structure);
 end for;

end for;
```

```
:
:
: -- STC rd[1],... with HOLD signal set
:
: INST      <= '1';
: D         <= fmt_Load & „00001“ & op_STC & „00101“ & „10000000000001“;
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
:
: CINS1     <= '1';
: D         <= „10111110000000111111000000000000“;-- ADD instruction
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
:
: D         <= „00000000000000000000000000000000“;
: CINS1     <= '0';
: nBHOLD    <= '0';
: INST      <= '0';
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
:
: D         <= „10111110001001111110000000000000“;-- SUB instruction
:
: nBHOLD    <= '1';
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
:
: INST      <= '0';
: D         <= „ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ“;-- SFU writes to data bus
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
:
: INST      <= '1';
: D         <= „10111110000000111111000000000000“;-- ADD instruction
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
: wait for 40 ns;
: CLK       <= '1';
: wait for 40 ns;
: CLK       <= '0';
:
```

```

while nCHOLD='0' loop
 wait for 40 ns;
 CLK      <= '1';
 wait for 40 ns;
 CLK      <= '0';
end loop;



-- test nHOLD signal

nBHOLD    <= '0';

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';

nBHOLD    <= '1';


-- use CINS1 and CINS2

D         <= „11000010000001111110000000000000";-- LD instruction

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';


D         <= fmt_Arith & „00000" & op_CLDC & „1111111111110110111"; -- CLDC

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';


D         <= fmt_Arith & „00001" & op_CLDC & „0000000000000000110"; -- CLDC

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';


D         <= „01010101010101010101010101010101";-- data for LD

CINS2     <= '1';
INST      <= '0';

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';


D         <= „10111110000001111110000000000000";-- ADD instruction

CINS2     <= '0';
CINS1     <= '1';
INST      <= '1';

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';


D         <= „10111110001001111110000000000000";-- SUB instruction

CINS1     <= '0';

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';

```

```

wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';

CINS1    <= '0';

while nCHOLD='0' loop
 wait for 40 ns;
 CLK     <= '1';
 wait for 40 ns;
 CLK     <= '0';
end loop;


D        <= „10111110001001111110000000000000“;-- SUB instruction

wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';

while nCHOLD='0' loop
 wait for 40 ns;
 CLK     <= '1';
 wait for 40 ns;
 CLK     <= '0';
end loop;


D        <= Z_vector;-- data cycles of STC instruction
INST     <= '0';

wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';

wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';


-- CRST rd[1]

D        <= fmt_Arith & „11111“ & op_Cxx & „11100“ & cop_CRST & „00001“;
INST     <= '1';

wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';

while nCHOLD='0' loop
 wait for 40 ns;
 CLK     <= '1';
 wait for 40 ns;
 CLK     <= '0';
end loop;


-- two IU instructions

D        <= „10111110000001111110000000000000“;-- ADD instruction
INST     <= '1';
CINS1    <= '1';

wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';

while nCHOLD='0' loop
 wait for 40 ns;
 CLK     <= '1';
 wait for 40 ns;
 CLK     <= '0';
end loop;


D        <= „10111110001001111110000000000000“;-- SUB instruction

CINS1    <= '0';

wait for 40 ns;
CLK      <= '1';
wait for 40 ns;
CLK      <= '0';
```

```

-- CLDC #-73, rd[1]

D         <= fmt_Arith & „00001“ & op_CLDC & „1111111111110110111“;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;



-- CSEL RDEN+1

D         <= fmt_Arith & „11111“ & op_CLDC & „0100000000000000001“;

CINS1     <= ’1’;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;



-- one IU instruction

D         <= „10111110000001111110000000000000“;-- ADD instruction

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;



-- CRST rs[0]

D         <= fmt_Arith & „11111“ & op_Cxx & „11100“ & cop_CRST & „00000“;

CINS1     <= ’0’;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;



-- CMOVE rs[0],rd[0]

D         <= fmt_Arith & „00000“ & op_Cxx & „00000“ & cop_CMOVE & „00000“;

CINS1     <= ’1’;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;

while nCHOLD=’0’ loop
 wait for 40 ns;
 CLK      <= ’1’;
 wait for 40 ns;
 CLK      <= ’0’;
end loop;



-- STC rd[1],...

D         <= fmt_Load & „00001“ & op_STC & „00101“ & „10000000000001“;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;

while nCHOLD=’0’ loop
 wait for 40 ns;
 CLK      <= ’1’;
 wait for 40 ns;
 CLK      <= ’0’;
end loop;


-- two IU instructions

INST      <= ’1’;
D         <= „10111110000001111110000000000000“;-- ADD instruction
```

```
CLK       <= ’0’;

INST      <= ’1’;
D         <= „10111110000000111111000000000000000“;-- ADD instruction

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;



-- STC rd[0],...

INST      <= ’1’;
D         <= fmt_Load & „00000“ & op_STC & „00101“ & „10000000000001“;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;



-- three IU instructions

CINS1     <= ’1’;
D         <= „10111110000000111111000000000000000“;-- ADD instruction

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;

CINS1     <= ’0’;
D         <= „10111110001001111111000000000000000“;-- SUB instruction

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;

INST      <= ’0’;
D         <= „ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ“;-- SFU writes to data bus

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;


INST      <= ’1’;
D         <= „10111110000000111111000000000000000“;-- ADD instruction

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;



-- CSEL WREN+RESET+2

INST      <= ’1’;
D         <= fmt_Arith & „11111“ & op_CLDC & „1010000000000000001“;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;



-- CMOVE rs[0], rd[0]

D         <= fmt_Arith & „00000“ & op_Cxx & „00000“ & cop_CMOVE & „00000“;

CINS1     <= ’1’;

wait for 40 ns;
CLK       <= ’1’;
wait for 40 ns;
CLK       <= ’0’;


```

```
wait for 40 ns;
CLK       <= '0';

nRESET    <= '1';



-- two IU instructions (not for SFU) to start...

INST      <= '1';
D         <= „10111110000001111110000000000000";-- ADD instruction

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';

D         <= „10111110001001111110000000000000";-- SUB instruction

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';



-- CSEL WREN+RDEN+RESET+1 instruction

INST      <= '1';
D         <= fmt_Arith & „11111" & op_CLDC & „1110000000000000000";

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';



-- CLDC #170, rd[0]and complete previous instruction (CINS1 signal)

INST      <= '1';
D         <= fmt_Arith & „00000" & op_CLDC & „0000000000010101010";

CINS1     <= '1';

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';



-- LDC ...,rd[1]

INST      <= '1';
D         <= fmt_Load & „00001" & op_LDC & „00101" & „10000000000000";

CINS1     <= '1';

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';



-- three IU instructions: (and completion of LDC)

CINS1     <= '1';
D         <= „10111110000001111110000000000000";-- ADD instruction

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';

CINS1     <= '0';
D         <= „10111110001001111110000000000000";-- SUB instruction

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
CLK       <= '0';

INST      <= '0';
D         <= „00000000000000001000110011110000";-- data for LDC instruction

wait for 40 ns;
CLK       <= '1';
wait for 40 ns;
```

```
end behavior;
```

SFU_test

```
-----------------------------------------------------------------------------
--
-- Name:  SFU_test
--
-----------------------------------------------------------------------------
--
-- Function:
--
-- This entity supplies an environment to simulate and debug the functions
-- of the SFU. A sequence of instructions is simulated and all relevant
-- signals are generated to pretend a real Integer Unit.
-- Should be used with the special test-SFHW.
--
-----------------------------------------------------------------------------
--
-- Parameters: [none]
--
-----------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 1-7-94
--
-----------------------------------------------------------------------------


-----------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use       IEEE.std_logic_1164.all;

use       WORK.SFU_Types.all;
use       WORK.SFU_components.SFU;


-----------------------------------------------------------------------------
--
-- Interface
--

entity SFU_test is end;


-----------------------------------------------------------------------------
--
-- Architecture
--

architecture behavior of SFU_test is

signal D : bit_32;
signal INST, CNULL, CINS1, CINS2, CCCV, CLK, nRESET : bit_1;
signal nCHOLD, nBHOLD, nFHOLD, nMHOLDA, nMHOLDB, nCP : bit_1;
signal CCC : bit_2;

begin


test_SFU: SFU
          port map (D, INST, CNULL, CINS1, CINS2, CCC, CCCV, CLK, nRESET,
                              nCHOLD, nBHOLD, nFHOLD, nMHOLDA, nMHOLDB, nCP);


process begin

INST      <= '0';
CINS1     <= '0';
CINS2     <= '0';
CLK       <= '0';
nBHOLD    <= '1';
nFHOLD    <= '1';
nMHOLDA   <= '1';
nMHOLDB   <= '1';


-- Perform a reset

nRESET    <= '0';

wait for 40 ns;
CLK       <= '1';
```

```

-------------------------------------------------------------------------------
--
-- Interface
--

entity SFHW is

          generic ( adr : Natural );

          port (   input    : in     arr_bit_32(0 to n_inputs(adr)-1);
                   valid    : in     arr_bit_1 (0 to n_inputs(adr)-1);
                   output   : out    arr_bit_32(0 to n_outputs(adr)-1);
                   write    : out    arr_bit_1 (0 to n_outputs(adr)-1);

                   CLK      : in     bit_1;
                   reset    : in     bit_1
          );

end SFHW;


-------------------------------------------------------------------------------
--
-- Architecture
--

architecture behavior of SFHW is

begin

-- just a dummy

process

variable delay_1, delay_2 : natural;


begin

 case adr is

-- NOTICE: the following code is only an example of SFHW implementation
--        This example is used with the simulation test program

 when 0 =>if reset='1' then
                    write <= „00“;
                   end if;

                   if valid(0)='1' and valid(1)='1' then
                    output(0)(23 downto 0) <= input(0)(15 downto 0) & input(1)(7 downto 0);
                    write(0) <= '1';
                    write(1) <= '1';
                   end if;


 when 1 =>if reset='1' then
                    write <= „00“;
                    delay_1 := 6;
                    delay_2 := 10;
                   end if;

                   if valid(0)='1' and valid(1)='1' then

                    if delay_1=0 then
                    output(0) <= input(0) and (input(1)(15 downto 0) & input(1)(15 downto 0));
                    write(0) <= '1';
                    else
                    delay_1 := delay_1-1;
                    end if;

                    if delay_2=0 then
                    output(1) <= input(1)(15 downto 0) & input(1)(15 downto 0);
                    write(1) <= '1';
                    else
                    delay_2 := delay_2-1;
                    end if;

                   end if;

                   wait until CLK'event and CLK='1';

 when others => null;

 end case;

wait for 1 ns;

end process;

```

```

   -- look if reset is active (highest priority)

   if reset='1' then
    valid_state<= '0';

   elsif WR='1' and valid_state='0' then-- connections to the SFHW
    data <= D(width-1 downto 0);-- latch data
    valid_state <= '1';

   end if;

   if EN='1' and A=reg_adr then
                                     -- connections to the SFU bus
    delay(0)<= '1';
    Rvalid_en<= '1';         -- assign valid bit to bus
   else
    delay(0)<= '0';
    Rvalid_en<= '0';
   end if;

  end if;

 end process;


 process
 begin

  if delay(1)='1' and RD='1' and EN='1' then
   Q(width-1 downto 0) <= data;                 -- put data to bus
   for i in 31 downto width loop      --expandsignbit
    Q(i)<=data(width-1);
   endloop;
  else
   Q <= Z_vector;
  end if;

  if Rvalid_en='1' then-- assign valid bit to bus
   Rvalid <= valid_state;
  else
   Rvalid <= 'Z';
  end if;

  wait for 10 ns;-- simulate combinational logic

 end process;

 end behavior;
```

SFHW

```
 -------------------------------------------------------------------------------
 --
 -- Name:  SFHW
 --
 -------------------------------------------------------------------------------
 --
 -- Function:
 -- This is a description of all SFHWs. It provides a standard interface
 -- and therefore allows an easy integration into the SFU implementation.
 -- For a database of all SFHW specific data (e.g. number of ports) see
 -- the file „SFHW_descriptions.vhdl".
 --
 -------------------------------------------------------------------------------
 --
 -- Parameters:
 -- · the number (address) of the SFHW to be built
 --
 -------------------------------------------------------------------------------
 --
 -- Author: Andreas Günther
 --
 -- Date: 16-6-94
 --
 -------------------------------------------------------------------------------


 -------------------------------------------------------------------------------
 --
 -- Libraries:
 --

 library IEEE;
 use      IEEE.std_logic_1164.all;

 use      WORK.SFU_Types.all;
 use      WORK.SFHW_descriptions.all;
```

```
-- · Address of the register
-- · Width of the latched data (optional, default is 32 bit)
-- The width of the input and output port is always 32 bit.
--
-------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 27-6-94
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.conv_std_logic_vector;

use WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Interface
--

entity ResReg is

         generic ( adr : Natural; width : Natural := 32 );

         port (   -- Interface to the SFU bus:

                   Q        : out    bit_32;
                   A        : in     bit_5;
                   RD       : in     bit_1;
                   Rvalid   : out    bit_1;

                   -- input from the SFHW:

                   D        : in     bit_32;
                   WR       : in     bit_1;

                   -- others:

                   valid    : out    bit_1;
                   CLK      : in     bit_1;
                   reset    : in     bit_1;
                   EN       : in     bit_1
         );

end ResReg;


-------------------------------------------------------------------------------
--
-- Architecture
--

architecture behavior of ResReg is


signal data : std_logic_vector(width-1 downto 0);
                                     -- internal data latch
signal delay : bit_2;-- shift register for delayed
                                     -- signal assignments to the
                                     -- SFU bus
signal valid_state : bit_1;-- internal valid latch
signal reg_adr : bit_5;-- address of the register as bit_vector
signal Rvalid_en : bit_1;-- enable signal for Rvalid bus driver


begin


valid    <= valid_state;
reg_adr  <= conv_std_logic_vector(adr, 5);


process (CLK)
begin

 if CLK'event and CLK='1' then-- rising edge

  -- shift the delay register

  delay(1) <= delay(0);
```

```

use      WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Interface
--

entity OpReg is

         generic ( adr : Natural; width : Natural := 32);

         port (   -- Interface to the SFU bus:

                  D        : in     bit_32;
                  A        : in     bit_5;
                  WR       : in     bit_1;

                  -- output to the SFHW:

                  Q        : out    bit_32;
                  valid    : out    bit_1;

                  -- others:

                  CLK      : in     bit_1;
                  reset    : in     bit_1;
                  EN       : in     bit_1
         );

end OpReg;


-------------------------------------------------------------------------------
--
-- Architecture
--


architecture behavior of OpReg is

signal reg_adr : bit_5;

begin

reg_adr <= conv_std_logic_vector(adr, 5);


process (CLK)
begin

 if CLK'event and CLK='1' then

  -- first look if reset is active (highest priority)

  if reset='1' then
   valid <= '0';

  elsif EN='1' and WR='1' and A=reg_adr then
   Q(width-1 downto 0) <= D(width-1 downto 0);
   valid <= '1';

  end if;

 end if;

end process;

end behavior;
```

ResReg

```
-------------------------------------------------------------------------------
--
-- Name:  ResReg
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This register (Result Register) latches the output of the SFHW and supplies
-- this data on the SFU bus upon request. The status of the data can be
-- read via the status output.
--
-------------------------------------------------------------------------------
--
-- Parameters:
```

```
begin

reset <= int_reset;-- same as external signals
ENOP <= int_ENOP;
ENRES <= int_ENRES;


process (D,WR,A)
begin

 if WR='1' and A="11111" then

  if D(15 downto 0)=conv_std_logic_vector(adr, 16) then
   int_reset <= D(16);               -- this SFSU is selected
   int_ENRES <= D(17);
   int_ENOP <= D(18);

  else
   if D(17)='1' then                 -- another SFSU is selected
    int_ENRES <= '0';                -- so deselect the selection in this SFSU
   end if;
   if D(18)='1' then
    int_ENOP <= '0';
   end if;

  end if;

 else

  int_reset <= '0';

 end if;

end process;


process(int_ENRES, SFHW_status)
begin

 if int_ENRES='1' then               -- put SFSU status to bus while the
  STATUS <= SFHW_status;             -- result registers are selected
 else
  STATUS <= 'Z';
 end if;

end process;


end behavior;
```

OpReg

```
------------------------------------------------------------------------------
--
-- Name:  OpReg
--
------------------------------------------------------------------------------
--
-- Function:
--
-- Operand register. When it is selected by its address and enable line,
-- the input value is latched and the valid bit is set. The reset input
-- resets the valid bit.
--
------------------------------------------------------------------------------
--
-- Parameters:
-- · Address of this register
-- · Width of data in bits (optional, default is 32)
-- The input and output is always 32 bit width.
--
------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 27-6-94
--
------------------------------------------------------------------------------


------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.conv_std_logic_vector;
```

```

end process;


end structure;
```

SFSU_control

```
--------------------------------------------------------------------------------
--
-- Name:  SFSU_control
--
--------------------------------------------------------------------------------
--
-- Function:
--
-- This module controls all registers which belong to the specified SFSU.
-- In addition, the operations concerning the entire SFSU are performed
-- here. (SFSU reset and status)
--
--------------------------------------------------------------------------------
--
-- Parameters:
-- · Address of the SFSU
--
--------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 16-6-94
--
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.conv_std_logic_vector;

use WORK.SFU_Types.all;


--------------------------------------------------------------------------------
--
-- Interface
--

entity SFSU_control is

         generic ( adr : Natural);

         port (   -- Interface to the SFU bus:

                   D        : in     bit_19;
                   A        : in     bit_5;
                   WR       : in     bit_1;
                   STATUS   : out    bit_1;

                   -- Interface to the registers:

                   reset    : out    bit_1;
                   ENOP     : out    bit_1;
                   ENRES    : out    bit_1;
                   SFHW_status:
                              in     bit_1;

                   -- others:

                   CLK      : in     bit_1
         );

end SFSU_control;


--------------------------------------------------------------------------------
--
-- Architecture
--

architecture behavior of SFSU_control is

signal int_reset : bit_1;           -- internal signals
signal int_ENOP, int_ENRES : bit_1;

```

```
-------------------------------------------------------------------------------
--
-- Interface
--

entity SFSU is

          generic ( adr : Natural );

          port (    -- Interface to the SFU bus:

                   RD       : inout  bit_32;
                   RWA      : in     bit_5;
                   RRA      : in     bit_5;
                   RWR      : in     bit_1;
                   RRD      : in     bit_1;
                   RVALID   : out    bit_1;
                   RSTATUS  : out    bit_1;

                   -- other

                   CLK      : in     bit_1
          );

end SFSU;


-------------------------------------------------------------------------------
--
-- Architecture
--

architecture structure of SFSU is


-- signals from operand registers to SFHW:
signal op_data     : arr_bit_32(0 to n_inputs(adr)-1);
signal op_valid    : arr_bit_1 (0 to n_inputs(adr)-1);

-- signals from SFHW to result registers:
signal res_data    : arr_bit_32(0 to n_outputs(adr)-1);
signal write       : arr_bit_1 (0 to n_outputs(adr)-1);

-- other signals:
signal res_valid: arr_bit_1 (0 to n_outputs(adr)-1);
signal reset, status: bit_1;
signal en_op, en_res: bit_1;


begin

control: SFSU_control
          generic map (adr)
          port map (RD(18 downto 0), RWA, RWR, RSTATUS, reset, en_op, en_res, status, CLK);

OpRegisters:
 for i in 0 to n_inputs(adr)-1 generate
   OpRegister: OpReg
          generic map (i, op_width(adr)(i))
          port map (RD, RWA, RWR, op_data(i), op_valid(i), CLK, reset, en_op);
 end generate;

ResRegisters:
 for i in 0 to n_outputs(adr)-1 generate
    ResRegister: ResReg
          generic map (i, res_width(adr)(i))
          port map (RD, RRA, RRD, RVALID, res_data(i), write(i), res_valid(i), CLK, reset, en_res);
 end generate;

this_SFHW: SFHW
          generic map (adr)
          port map (op_data, op_valid, res_data, write, CLK, reset);



-- the following process generates an AND-gate to compute the SFHW status

process (res_valid)

variable tmp_status : bit_1;

begin

 tmp_status := '1';

 for i in 0 to n_outputs(adr)-1 loop
   tmp_status := tmp_status and res_valid(i);
 end loop;

 status <= tmp_status;
```

```
    case Instr(13 downto 5) is

     when cop_CMOVE =>      -- CMOVE instruction
                            RWA <= Instr(29 downto 25);
                            RWR <= ’1’;
                            RRD <= ’1’;

     when cop_CRST =>       -- CRST instruction
                            RWR <= ’0’;
                            RRD <= ’1’;

     when cop_CNOP =>       -- CNOP instruction
                            RWR <= ’0’;
                            RRD <= ’0’;

     when others =>
                            null;

    end case;

   end if;

  end if;


  if Instr(31 downto 30)=fmt_Load and Instr(24 downto 19)=op_STC then
   if stc_flag=’1’ then             -- set CNULL signal during second STC cycle
    CNULL <= ’1’;
    stc_flag <= ’0’;
   else
    stc_flag <= ’1’;
   end if;
  else
   CNULL <= ’0’;
  end if;


 end if;

end process;


end behavior;
```

SFSU

```
-------------------------------------------------------------------------------
--
-- Name:  SFSU
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This is the description of one SFSU consisting of one SFHW, one
-- SFSU control unit and the appropriate number of operand- and result
-- registers.
-- Data about the SFHW is contained in the file „SFHW_descriptions.vhdl“.
--
-------------------------------------------------------------------------------
--
-- Parameters:
-- · Address of the SFSU
--
-------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 16-6-94
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;

use WORK.SFU_Types.all;
use WORK.SFU_components.OpReg;
use WORK.SFU_components.ResReg;
use WORK.SFU_components.SFSU_control;
use WORK.SFU_components.SFHW;
use WORK.SFHW_descriptions.all;

```

```
                     RD       : inout  bit_32;
                     RWA      : out    bit_5;
                     RRD      : out    bit_1;
                     RWR      : out    bit_1;

                     -- Interface to the execute stage:

                     Instruction:
                               in      bit_32;

                     -- others:

                     HOLD     : in     bit_1;
                     CLK      : in     bit_1;
                     RESET    : in     bit_1

          );

end write_stage;


-----------------------------------------------------------------------------
--
-- Architecture
--

architecture behavior of write_stage is

signal Instr : bit_32;               -- instruction to be executed
signal stc_flag : bit_1;             -- is '1' when in second cycle of STC instruction

begin


process (CLK)
begin

 if CLK'event and CLK='1' then

  if HOLD='0' then
   Instr <= Instruction;
  end if;

  if INST='0' and Instruction(31 downto 30)=fmt_Load and Instruction(24 downto 19)=op_STC then
   D <= RD;
  else
   D <= Z_vector;

  end if;

 end if;

end process;


process (CLK)
begin

 if CLK'event and CLK='0' then

  if Instr(31 downto 30)=fmt_Load then

   if Instr(24 downto 19)=op_LDC then
    RD <= D;                                     -- LDC instruction
    RWA <= Instr(29 downto 25);
    RWR <= '1';
    RRD <= '0';
   end if;

   if Instr(24 downto 19)=op_STC then
    RD <= Z_vector;
    RWR <= '0';                                  -- STC instruction
    RRD <= '1';
   end if;

  elsif Instr(31 downto 30)=fmt_Arith then

   if Instr(24 downto 19)=op_CLDC then           -- CLDC instruction
    for i in 31 downto 19 loop
     RD(i) <= Instr(18);                         -- sign extension
    end loop;
    RD(18 downto 0) <= Instr(18 downto 0);
    RWA <= Instr(29 downto 25);
    RWR <= '1';
    RRD <= '0';
   else
    RD <= Z_vector;
   end if;

   if Instr(24 downto 19)=op_Cxx then
```

```

process(CLK)
begin

 if CLK'event and CLK='0' then       -- falling edge of CLK

  if Inst(31 downto 30)=fmt_Arith and Inst(24 downto 19)=op_Cxx and Inst(13 downto 5)=cop_CRST then
   CCC(1) <= RVALID;
  end if;

 end if;


end process;


process
begin

 -- hold pipeline when STC or CMOVE with invalid data:

 if (Inst(31 downto 30)=fmt_Load and Inst(24 downto 19)=op_STC) or
 (Inst(31 downto 30)=fmt_Arith and Inst(24 downto 19)=op_Cxx and
                                               Inst(13 downto 5)=cop_CMOVE) then
  CHOLD <= not RVALID;
 elsif RESET='1' then
  CHOLD <= '0';
 end if;

 wait for 10 ns;            -- to simulate combinational logic

end process;


end behavior;
```

write_stage

```
-------------------------------------------------------------------------------
--
-- Name:  write_stage
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This stage reads the data from the memory and writes them to the SFU bus
-- or vice versa.
--
-------------------------------------------------------------------------------
--
-- Parameters:[none]
--
-------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 18-6-94
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use      IEEE.std_logic_1164.all;

use      WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Interface
--

entity write_stage is

         port (    -- Interface to the IU:

                   D        : inout  bit_32;
                   INST     : in     bit_1;
                   CNULL    : out    bit_1;

                   -- Interface to the SFU bus:

```

```
-- The pipeline is freezed if there is a read instruction (STC or CMOVE)
-- for which the source value is not valid yet.
--
-------------------------------------------------------------------------------
--
-- Parameters:[none]
--
-------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 16-6-94
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use       IEEE.std_logic_1164.all;

use       WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Interface
--

entity execute_stage is

          port (    -- Interface to the IU:

                    CCC      : out    bit_2;
                    CHOLD    : out    bit_1;

                    -- Interface to the SFU bus:

                    RVALID   : in     bit_1;
                    RSTATUS  : in     bit_1;

                    -- Interface to other stages:

                    Instruction_in:
                              in      bit_32;
                    Instruction_out:
                              out     bit_32;

                    -- others:

                    HOLD     : in     bit_1;
                    CLK      : in     bit_1;
                    RESET    : in     bit_1
          );

end execute_stage;


-------------------------------------------------------------------------------
--
-- Architecture
--

architecture behavior of execute_stage is


signal Inst : bit_32;                -- instruction to be executed

begin

CCC(0) <= RSTATUS;


process (CLK)
begin

 if CLK'event and CLK='1' then       -- rising edge of CLK

  if HOLD='0' then
   Inst <= Instruction_in;
   Instruction_out <= Instruction_in;
  end if;

 end if;

end process;

```

```
begin

 if CLK'event and CLK='1' then       -- rising edge of CLK

  if INST='1' and HOLD='0' then
   Inst_1 <= D;
   Inst_2 <= Inst_1;
  end if;

  if HOLD='0' then
   IOP(1) <= IOP(0);
  end if;

 end if;

end process;


process(CLK)
begin

 if CLK'event and CLK='0' then       -- falling edge of CLK

  -- assign address of source register to SFU bus if HOLD is not asserted

  if HOLD='0' then
   if Inst_sel(24 downto 19)=op_STC then
    RRA <= Inst_sel(29 downto 25);             -- the source address is located
   else                                        -- at different bits in STC
    RRA <= Inst_sel(4 downto 0);
   end if;
  end if;

 end if;

end process;


process
begin

 -- when CINS1 or CINS2 is asserted, we know which instruction to execute

 if IOP(1)='0' and HOLD='0' then

  if CINS1='1' then                  -- decode instruction in buffer 1
   Inst_sel <= Inst_1;
  elsif CINS2='1' then               -- decode instruction in buffer 2
   Inst_sel <= Inst_2;
  else                       -- if no SFU instruction then
   Inst_sel(31 downto 30) <= fmt_Arith;-- generate CNOP instruction
   Inst_sel(24 downto 19) <= op_Cxx;
   Inst_sel(13 downto 5) <= cop_CNOP;
  end if;

 end if;

  -- find out, if an IOP is to be generated (only with STC instruction)

  if Inst_sel(24 downto 19)=op_STC and IOP(1)='0' then
   IOP(0) <= '1';
  else
   IOP(0) <= '0';
  end if;

 -- propagate the instruction to the execute stage

 Instruction <= Inst_sel;

 wait for 10 ns; -- this simulates combinational logic

end process;


end behavior;
```

execute_stage

```
-------------------------------------------------------------------------------
--
-- Name:  execute_stage
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This stage reads the status bits from the SFU bus and sets the condition
-- codes according to them if a CRST or CSST instruction enters this stage.
```

decode_stage

```
------------------------------------------------------------------------------
--
-- Name:  decode_stage
--
------------------------------------------------------------------------------
--
-- Function:
--
-- Reads all instructions from the data bus and latches the last two ones.
-- Decodes the instruction which is selected by the CINS input and supplies
-- the read address RRA on the SFU bus. The instruction is the forwarded to
-- the execute stage.
-- The IOPs which are necessary for multi-cycle instructions are generated
-- in this stage.
--
------------------------------------------------------------------------------
--
-- Parameters:[none]
--
------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 16-6-94
--
------------------------------------------------------------------------------


------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use       IEEE.std_logic_1164.all;

use       WORK.SFU_Types.all;


------------------------------------------------------------------------------
--
-- Interface
--

entity decode_stage is

          port (   -- Interface to the IU:

                   D        : in     bit_32;
                   INST     : in     bit_1;
                   CINS1    : in     bit_1;
                   CINS2    : in     bit_1;

                   -- Interface to the SFU bus:

                   RRA      : out    bit_5;

                   -- Interface to the execute stage:

                   Instruction:
                             out     bit_32;

                   -- others:

                   HOLD     : in     bit_1;
                   CLK      : in     bit_1;
                   RESET    : in     bit_1
          );

end decode_stage;


------------------------------------------------------------------------------
--
-- Architecture
--

architecture behavior of decode_stage is

signal Inst_1, Inst_2: bit_32;       -- last two instructions
signal Inst_sel    : bit_32;         -- selected instruction which is decoded
signal IOP         : bit_2;          -- shift register to remember STC instruction


begin

process (CLK)
```

```
--------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 16-6-94
--
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use      IEEE.std_logic_1164.all;

use      WORK.SFU_Types.all;
use      WORK.SFU_components.decode_stage;
use      WORK.SFU_components.execute_stage;
use      WORK.SFU_components.write_stage;


--------------------------------------------------------------------------------
--
-- Interface
--

entity SFU_control is

         port (   -- Interface to the processor:

                  D        : inout  bit_32;
                  INST     : in     bit_1;
                  CINS1    : in     bit_1;
                  CINS2    : in     bit_1;
                  CCC      : out    bit_2;
                  CNULL    : out    bit_1;
                  CHOLD    : out    bit_1;
                  HOLD     : in     bit_1;

                  -- Interface to the SFU bus:

                  RD       : inout  bit_32;
                  RWA      : out    bit_5;
                  RRA      : out    bit_5;
                  RWR      : out    bit_1;
                  RRD      : out    bit_1;
                  RVALID   : in     bit_1;
                  RSTATUS  : in     bit_1;

                  -- others:

                  CLK      : in     bit_1;
                  RESET    : in     bit_1
         );

end SFU_control;


--------------------------------------------------------------------------------
--
-- Architecture
--

architecture structure of SFU_control is


signal Inst_D_E, Inst_E_W : bit_32;            -- instruction pipeline


begin

-- the pipeline stages with the IU connection:

dec: decode_stage
         port map (D, INST, CINS1, CINS2, RRA, Inst_D_E, HOLD, CLK, RESET);

exe: execute_stage
         port map (CCC, CHOLD, RVALID, RSTATUS, Inst_D_E, Inst_E_W, HOLD, CLK, RESET);

wri: write_stage
         port map (D, INST, CNULL, RD, RWA, RRD, RWR, Inst_E_W, HOLD, CLK, RESET);


end structure;
```

```

                    D        : inout  bit_32;
                    INST     : in     bit_1;
                    CNULL    : out    bit_1;
                    CINS1    : in     bit_1;
                    CINS2    : in     bit_1;
                    CCC      : out    bit_2;
                    CCCV     : out    bit_1;
                    CLK      : in     bit_1;
                    nRESET   : in     bit_1;
                    nCHOLD   : out    bit_1;
                    nBHOLD   : in     bit_1;
                    nFHOLD   : in     bit_1;
                    nMHOLDA  : in     bit_1;
                    nMHOLDB  : in     bit_1;
                    nCP      : out    bit_1
          );

end SFU;


----------------------------------------------------------------------------
--
-- Architecture
--

architecture structure of SFU is


signal RESET, HOLD, CHOLD : bit_1;

signal RD : bit_32;
signal RWA, RRA : bit_5;
signal RRD, RWR, RVALID, RSTATUS : bit_1;


begin    -- (architecture)


-- connections to the IU:

RESET     <= not nRESET;
HOLD      <= not nBHOLD or not nFHOLD or not nMHOLDA or not nMHOLDB or CHOLD;
nCHOLD    <= not CHOLD;
nCP       <= '0';
CCCV      <= '1';           -- condition codes are always valid


-- SFU control:

control: SFU_control
         port map (D, INST, CINS1, CINS2, CCC, CNULL, CHOLD, HOLD, RD, RWA, RRA,
                            RWR, RRD, RVALID, RSTATUS, CLK, RESET);


-- all SFSUs:

all_SFSU:

for i in 0 to n_SFHW-1 generate

 SFSubUnit : SFSU
         generic map (i)
         port map (RD, RWA, RRA, RWR, RRD, RVALID, RSTATUS, CLK);

end generate;


end structure;
```

SFU_control

```
----------------------------------------------------------------------------
--
-- Name:  SFU_control
--
----------------------------------------------------------------------------
--
-- Function:
--
-- This module contains the instruction pipeline to decode and execute the
-- special SFU instructions. It serves as an interface between the
-- processor bus and the SFU bus.
--
----------------------------------------------------------------------------
--
-- Parameters: [none]
--
```

```

                    D        : in     bit_19;
                    A        : in     bit_5;
                    WR       : in     bit_1;
                    STATUS   : out    bit_1;

                    -- Interface to the registers:

                    reset    : out    bit_1;
                    ENOP     : out    bit_1;
                    ENRES    : out    bit_1;
                    SFHW_status:
                               in     bit_1;

                    -- others:

                    CLK      : in     bit_1
          );
 end component;


 component SFHW

          generic ( adr : Natural );

          port (
                    input    : in     arr_bit_32(0 to n_inputs(adr)-1);
                    valid    : in     arr_bit_1 (0 to n_inputs(adr)-1);
                    output   : out    arr_bit_32(0 to n_outputs(adr)-1);
                    write    : out    arr_bit_1 (0 to n_outputs(adr)-1);

                    CLK      : in     bit_1;
                    reset    : in     bit_1
          );
 end component;


end SFU_components;
```

SFU

```
-------------------------------------------------------------------------------
--
-- Name:  SFU
--
-------------------------------------------------------------------------------
--
-- Function:
--
-- This is the top level entity of the SFU-Implementation.
-- It contains the complete SFU consisting of all SFSUs and the instruction
-- pipeline.
--
-------------------------------------------------------------------------------
--
-- Parameters:[none]
--
-------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 27-6-94
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use       IEEE.std_logic_1164.all;

use       WORK.SFU_Types.all;
use       WORK.SFHW_descriptions.all;
use       WORK.SFU_components.SFU_control;
use       WORK.SFU_components.SFSU;


-------------------------------------------------------------------------------
--
-- Interface
--

entity SFU is

          port (   -- Interface to the IU:
```

```

                    Instruction:
                              in      bit_32;

                    -- others:

                    HOLD     : in     bit_1;
                    CLK      : in     bit_1;
                    RESET    : in     bit_1

          );
 end component;


 component SFSU

          generic ( adr : Natural );

          port (    -- Interface to the SFU bus:

                    RD       : inout  bit_32;
                    RWA      : in     bit_5;
                    RRA      : in     bit_5;
                    RWR      : in     bit_1;
                    RRD      : in     bit_1;
                    RVALID   : out    bit_1;
                    RSTATUS  : out    bit_1;

                    -- other

                    CLK      : in     bit_1
          );
 end component;


 component OpReg

          generic ( adr : Natural; width : Natural := 32);

          port (    -- Interface to the SFU bus:

                    D        : in     bit_32;
                    A        : in     bit_5;
                    WR       : in     bit_1;

                    -- output to the SFHW:

                    Q        : out    bit_32;
                    valid    : out    bit_1;

                    -- others:

                    CLK      : in     bit_1;
                    reset    : in     bit_1;
                    EN       : in     bit_1
          );
 end component;


 component ResReg

          generic ( adr : Natural; width : Natural := 32 );

          port (    -- Interface to the SFU bus:

                    Q        : out    bit_32;
                    A        : in     bit_5;
                    RD       : in     bit_1;
                    Rvalid   : out    bit_1;

                    -- input from the SFHW:

                    D        : in     bit_32;
                    WR       : in     bit_1;

                    -- others:

                    valid    : out    bit_1;
                    CLK      : in     bit_1;
                    reset    : in     bit_1;
                    EN       : in     bit_1
          );
 end component;


 component SFSU_control

          generic ( adr : Natural);

          port (    -- Interface to the SFU bus:
```

```

                  -- Interface to the SFU bus:

                  RD       : inout  bit_32;
                  RWA      : out    bit_5;
                  RRA      : out    bit_5;
                  RWR      : out    bit_1;
                  RRD      : out    bit_1;
                  RVALID   : in     bit_1;
                  RSTATUS  : in     bit_1;

                  -- others:

                  CLK      : in     bit_1;
                  RESET    : in     bit_1
        );
end component;


component decode_stage

        port (    -- Interface to the IU:

                  D        : in     bit_32;
                  INST     : in     bit_1;
                  CINS1    : in     bit_1;
                  CINS2    : in     bit_1;

                  -- Interface to the SFU bus:

                  RRA      : out    bit_5;

                  -- Interface to the execute stage:

                  Instruction:
                            out     bit_32;

                  -- others:

                  HOLD     : in     bit_1;
                  CLK      : in     bit_1;
                  RESET    : in     bit_1
        );
end component;


component execute_stage

        port (    -- Interface to the IU:

                  CCC      : out    bit_2;
                  CHOLD    : out    bit_1;

                  -- Interface to the SFU bus:

                  RVALID   : in     bit_1;
                  RSTATUS  : in     bit_1;

                  -- Interface to other stages:

                  Instruction_in:
                            in      bit_32;
                  Instruction_out:
                            out     bit_32;

                  -- others:

                  HOLD     : in     bit_1;
                  CLK      : in     bit_1;
                  RESET    : in     bit_1
        );
end component;


component write_stage

        port (    -- Interface to the IU:

                  D        : inout  bit_32;
                  INST     : in     bit_1;
                  CNULL    : out    bit_1;

                  -- Interface to the SFU bus:

                  RD       : inout  bit_32;
                  RWA      : out    bit_5;
                  RRD      : out    bit_1;
                  RWR      : out    bit_1;

                  -- Interface to the execute stage:
```

```
constant res_width: Width_arr:= ((32,16, 0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0),
                                 (32,32, 0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0));


end SFHW_descriptions;

package body SFHW_descriptions is end;
```

SFU_components

```
-------------------------------------------------------------------------------
--
-- Name:  SFU components
--
-------------------------------------------------------------------------------
--
-- Function:
-- Entity descriptions of all components.
--
-------------------------------------------------------------------------------
--
-- Parameters:[none]
--
-------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 16-6-94
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use      IEEE.std_logic_1164.all;

use      WORK.SFU_Types.all;
use      WORK.SFHW_descriptions.all;


-------------------------------------------------------------------------------
--
-- Package
--

package SFU_components is

 component SFU

         port (   -- Interface to the IU:

                  D        : inout  bit_32;
                  INST     : in     bit_1;
                  CNULL    : out    bit_1;
                  CINS1    : in     bit_1;
                  CINS2    : in     bit_1;
                  CCC      : out    bit_2;
                  CCCV     : out    bit_1;
                  CLK      : in     bit_1;
                  nRESET   : in     bit_1;
                  nCHOLD   : out    bit_1;
                  nBHOLD   : in     bit_1;
                  nFHOLD   : in     bit_1;
                  nMHOLDA  : in     bit_1;
                  nMHOLDB  : in     bit_1;
                  nCP      : out    bit_1
         );
 end component;


 component SFU_control

         port (   -- Interface to the processor:

                  D        : inout  bit_32;
                  INST     : in     bit_1;
                  CINS1    : in     bit_1;
                  CINS2    : in     bit_1;
                  CCC      : out    bit_2;
                  CNULL    : out    bit_1;
                  CHOLD    : out    bit_1;
                  HOLD     : in     bit_1;
```

```

constant op_CLDC: bit_6 := „110110“;
constant op_Cxx    : bit_6 := „110111“;-- specified by cop_...

-- coprocessor specific opcodes:

constant cop_CMOVE: bit_9 := „000000001“;
constant cop_CRST : bit_9 := „000000010“;
constant cop_CNOP : bit_9 := „000000000“;


end SFU_Types;

package body SFU_Types is end;
```

SFHW_descriptions

```
-------------------------------------------------------------------------------
--
-- Name:  SFHW descriptions
--
-------------------------------------------------------------------------------
--
-- Function:
-- This is a package with contains the descriptions of all SFHW interfaces.
-- The ordinal number (address) of each SFHW (equal to the SFSU address) and
-- the number of input and output ports is defined here.
-- The descriptions of the SFHWs can be found in the file „SFHW.vhdl“.
--
-------------------------------------------------------------------------------
--
-- Parameters:[none]
--
-------------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 16-6-94
--
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use      IEEE.std_logic_1164.all;

use      WORK.SFU_Types.all;


-------------------------------------------------------------------------------
--
-- Package
--

package SFHW_descriptions is


-------------------------------------------------------------------------------
-- total number of SFHW units:

constant n_SFHW    : Natural := 2;


-------------------------------------------------------------------------------
-- number of input and output ports of each SFHW:

type SFHW_arr is array(0 to n_SFHW-1) of Natural;

constant n_inputs: SFHW_arr:= (2,2);

constant n_outputs: SFHW_arr:= (2,2);


-------------------------------------------------------------------------------
-- bitwidth of each input and output port:

type Register_Width is array(0 to 31) of Natural;
type Width_arr is array(0 to n_SFHW-1) of Register_Width;

constant op_width: Width_arr:= ((16,8, 0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0),
                                                (24,1, 0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0,0,0,0, 0,0));

```

C. VHDL-Beschreibungen

In diesem Anhang sind sämtliche VHDL-Beschreibungen der SFU abgedruckt. Für Erläuterungen zu diesen Quelltexten siehe Abschnitt 3.4.

SFU_Types

```
-----------------------------------------------------------------------------
--
-- Name:  SFU Types
--
-----------------------------------------------------------------------------
--
-- Function:
-- Type declarations for all global used data.
--
-----------------------------------------------------------------------------
--
-- Parameters:[none]
--
-----------------------------------------------------------------------------
--
-- Author: Andreas Günther
--
-- Date: 16-6-94
--
-----------------------------------------------------------------------------


-----------------------------------------------------------------------------
--
-- Libraries:
--

library IEEE;
use IEEE.std_logic_1164.all;


-----------------------------------------------------------------------------
--
-- Package
--

package SFU_Types is


-----------------------------------------------------------------------------
-- bit vectors

subtype bit_32 is std_logic_vector(31 downto 0);
subtype bit_19 is std_logic_vector(18 downto 0);
subtype bit_9  is std_logic_vector(8 downto 0);
subtype bit_6  is std_logic_vector(5 downto 0);
subtype bit_5  is std_logic_vector(4 downto 0);
subtype bit_2  is std_logic_vector(1 downto 0);
subtype bit_1  is std_logic;


-----------------------------------------------------------------------------
-- arrays of the above bit_vectors

type arr_bit_32 is array( natural range <> ) of bit_32;
type arr_bit_1  is array( natural range <> ) of bit_1;


-----------------------------------------------------------------------------
-- constant bit vectors

constant Z_vector: bit_32 := „ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ“;


-----------------------------------------------------------------------------
-- opcodes

-- formats:

constant fmt_Load: bit_2 := „11“;-- for LDC and STC
constant fmt_Arith: bit_2 := „10“; -- for other instructions

-- load and store instructions:

constant op_LDC    : bit_6 := „110000“;
constant op_STC    : bit_6 := „110100“;

-- other instructions:
```

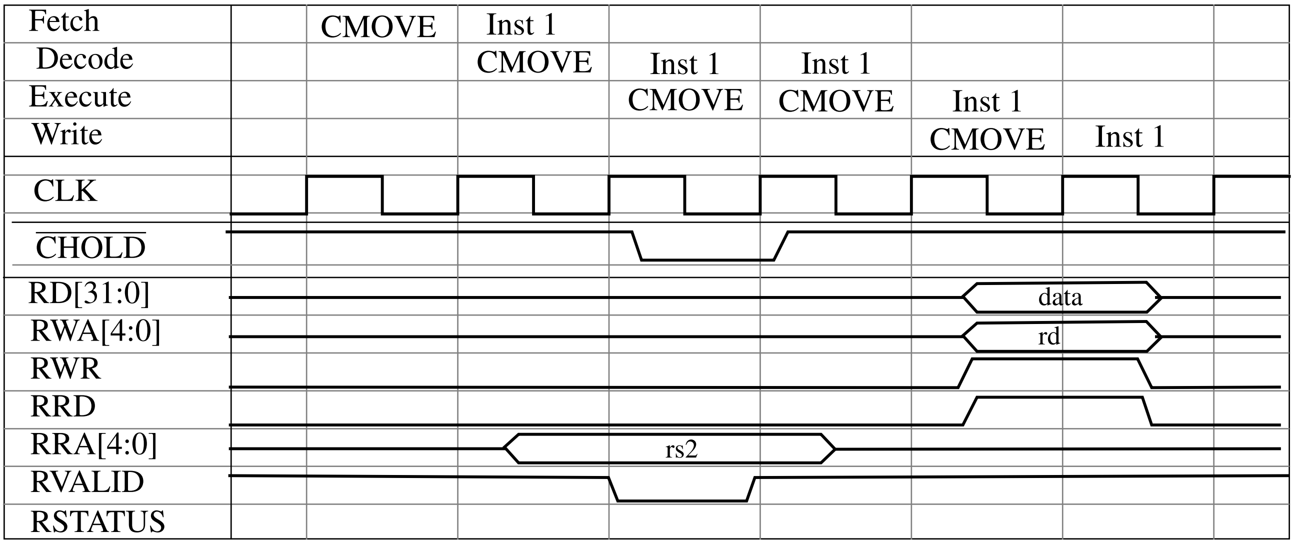

Abb. D.7: Timing des „CMOVE“-Befehls bei ungültigen Daten

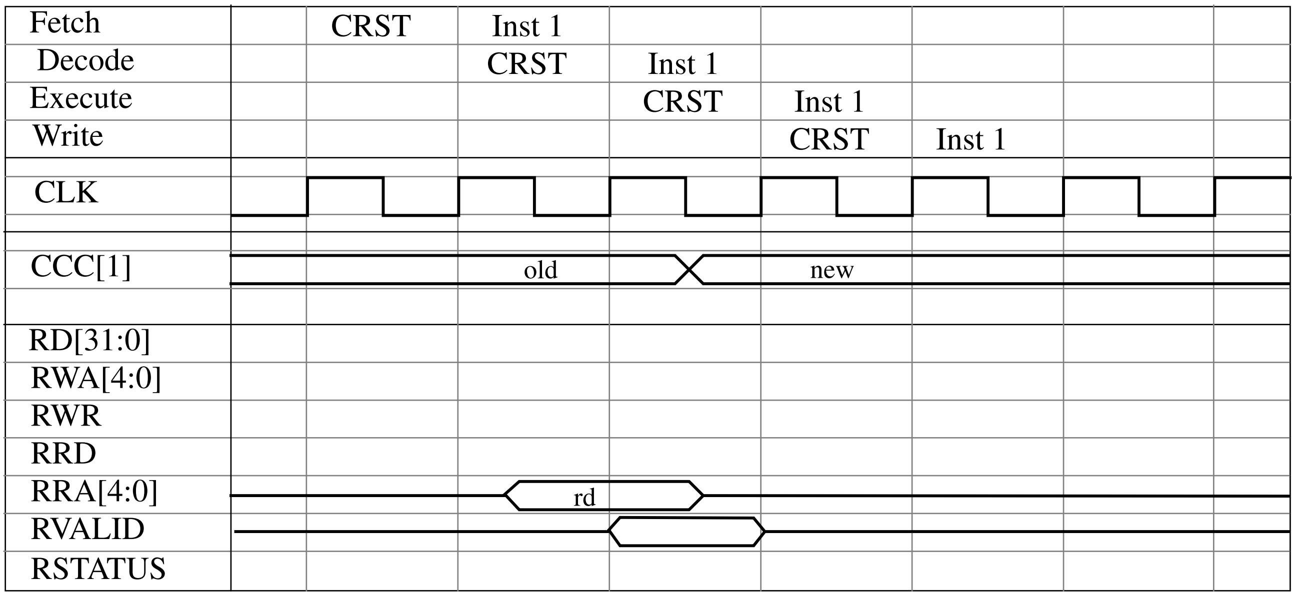

Abb. D.5: Timing des „CRST“-Befehls

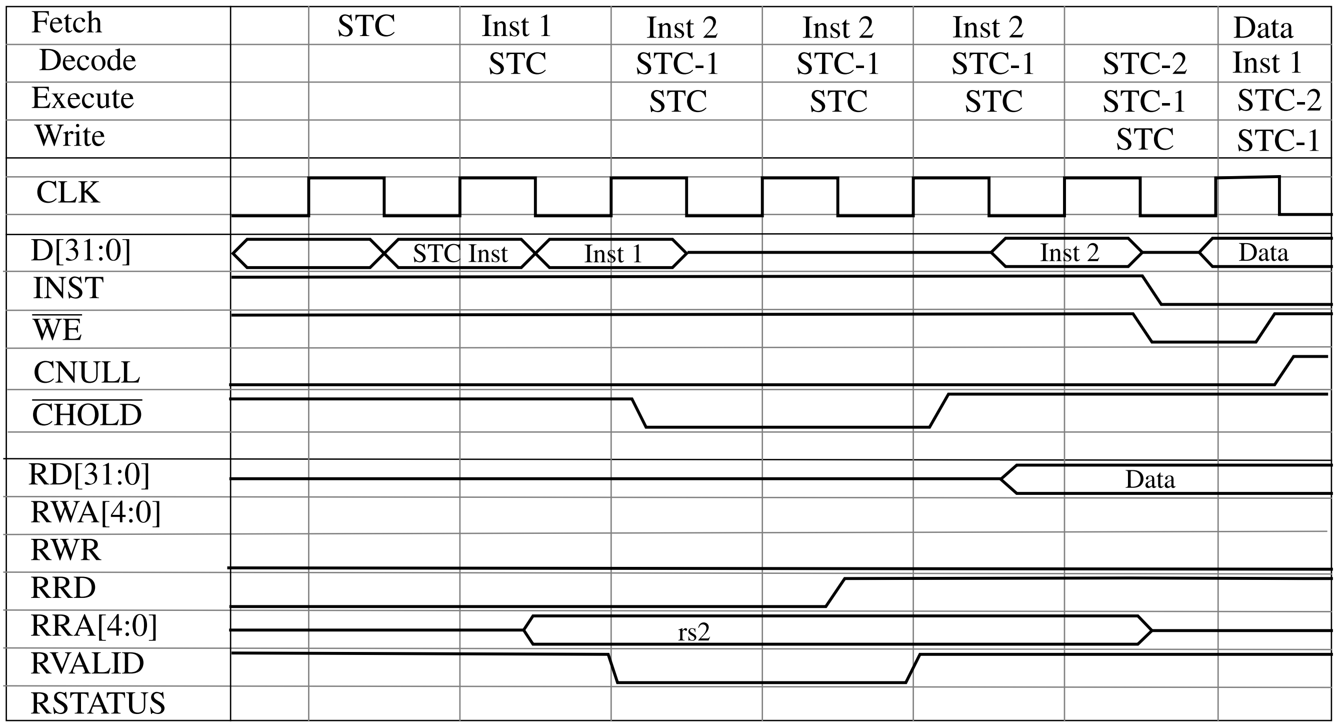

Abb. D.6: Timing des „STC“-Befehls bei ungültigen Daten

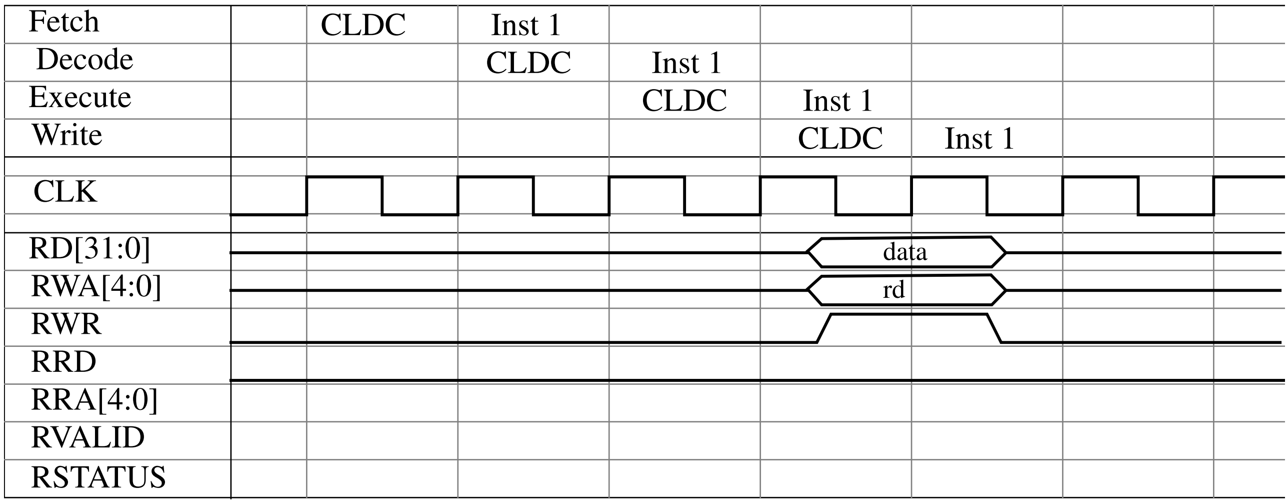

Abb. D.3: Timing des „CLDC“-Befehls

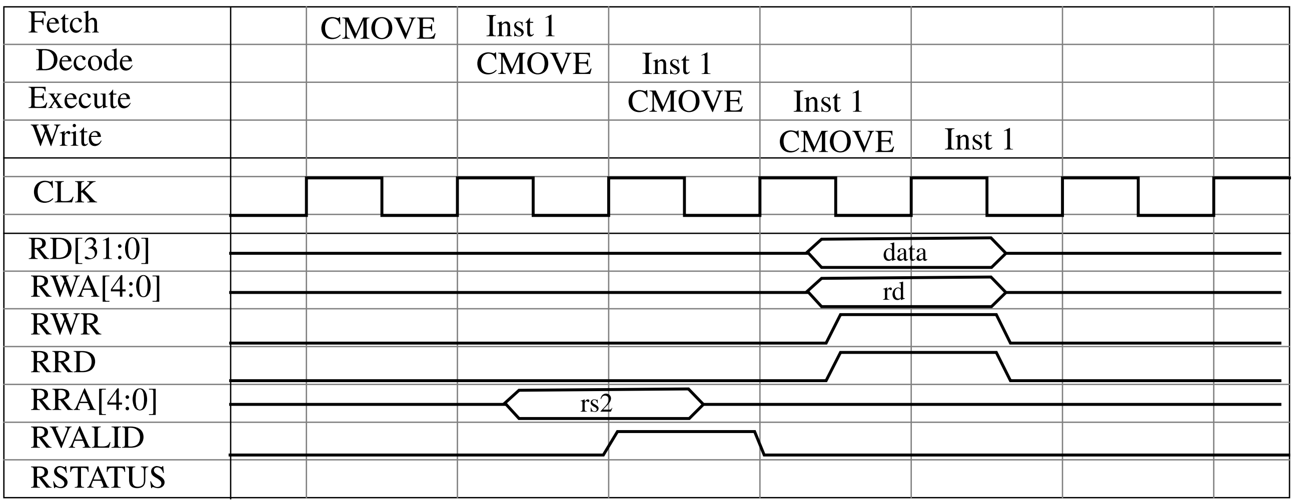

Abb. D.4: Timing des „CMOVE“-Befehls

B. Timing-Diagramme

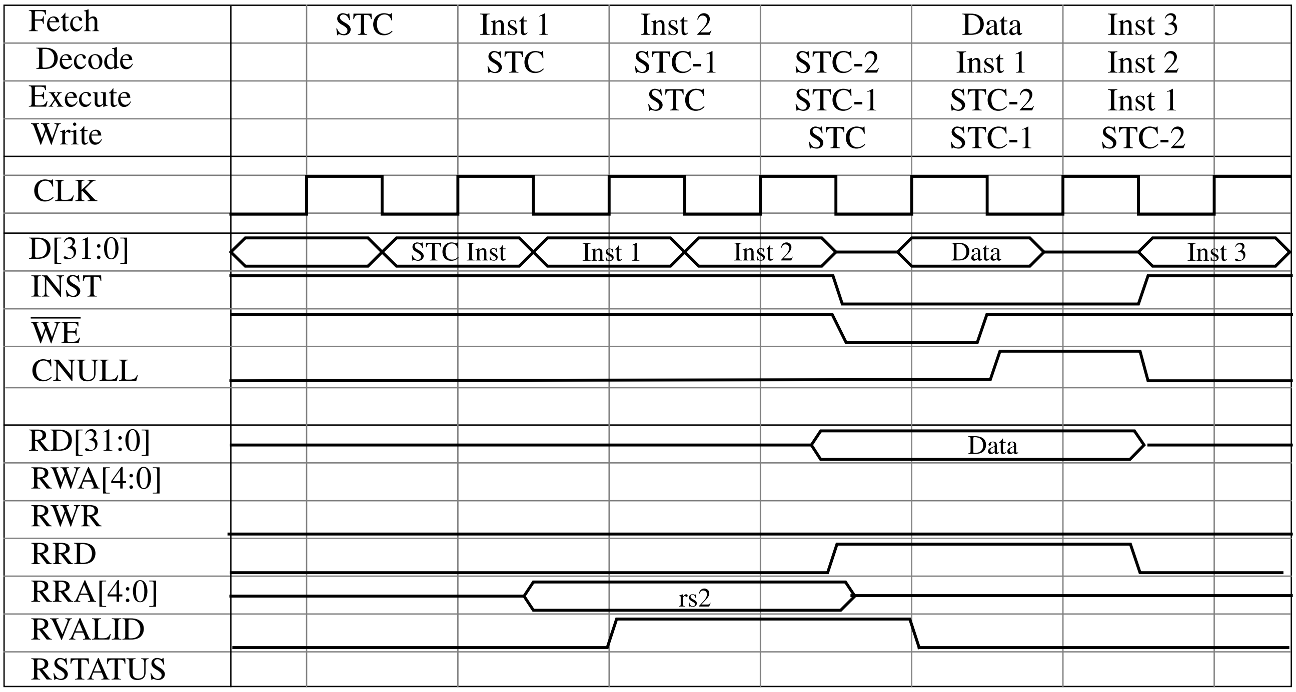

Abb. D.1: Timing des „STC“-Befehls

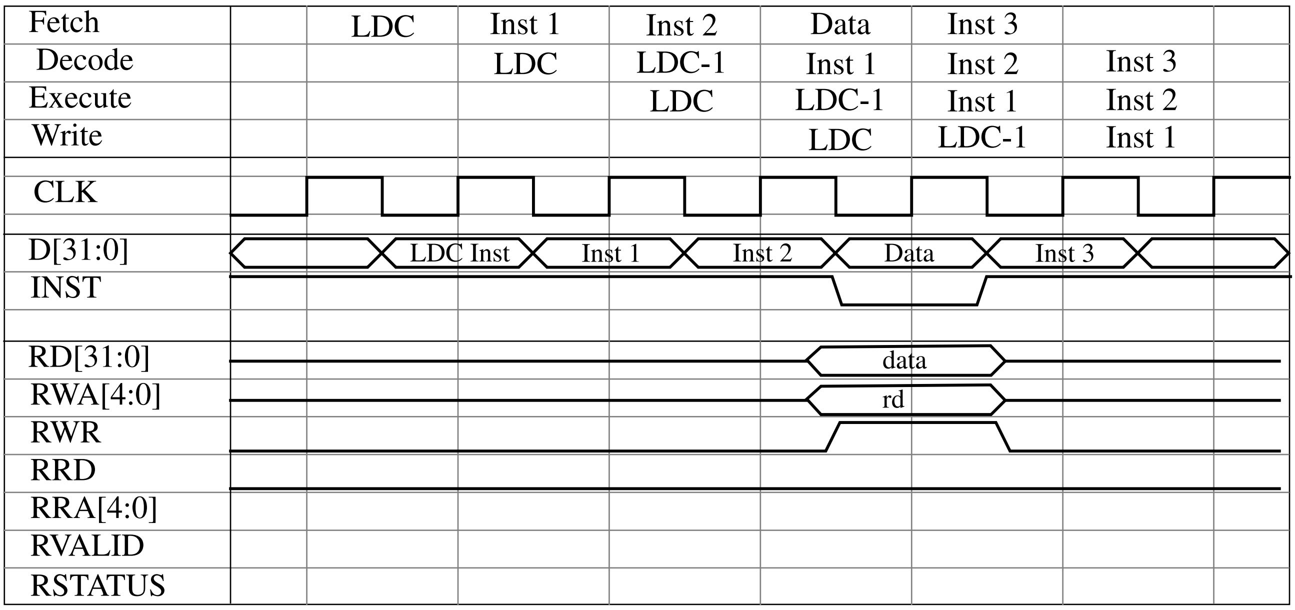

Abb. D.2: Timing des „LDC“-Befehls

Anhang

A. Literatur

[Bell92] F. Belli. „Programmiertechnik für nebenläufige und verteilte Systeme” *Universität-GH Paderborn FB 14*, 1992.

[Berg92] Jean-Michel-Bergé et al. „VHDL Designer’s Reference” *Kluwer*, 1992.

[Camp91] Raul Camposano, Robert A. Walker. „A Survey of high-level synthesis systems“ *Kluwer*, 1991.

[Camp94] Raul Camposano, Reiner Genevriere. „Abschlußbericht der Projektgruppe RISC-Design” *Universität-GH-Paderborn FB 17*, 1994.

[Cypr90] Cypress Semiconductor. „SPARC RISC User’s Guide.” Ross Technology Subsidiary, 3901 North First Street, San Jose, CA 95134. 1990.

[DeMi92] G. De Micheli, R.K. Gupta. „System-level Synthesis using Reprogrammable Components” *Proceedings of the European Design Automation Conference*, 1992.

[Erns92] R. Enst, J. Henkel. „Hardware-Software Co-Design of Embedded Controllers based on Hardware Extraction” *Proceedings of the ACM Workshop on Hardware-Software-CoDesign*, Oktober 1992.

[Hard93] Wolfram Hardt, Raul Camposano. „Trade-Offs in Hardware/Software Codesign” *Workshop Handouts of the ACM Workshop on Hardware-Software Codesign*, 1993.

[Hard94] Wolfram Hardt, Raul Camposano. „Specification Analysis for HW/SW-Partitioning” *Technical Report: SFB-358-B2-5/94, TU Dresden, 01062 Dresden*, 1994.

[Gupt92] R. K. Gupta, D. De Micheli. „System Synthesis via Hardware-Software Co-Design” *CSL TR-93-548*, 1992.

[Ku92] David C. Ku. „High level synthesis of ASICs under timing and synchronization constraints“ *Kluwer*, 1992.

[Lips89] R. Lipset, C. Schaefer, C. Ussery. „VHDL: Hardware Description and Design” *Kluwer*, 1989.

[Mano93] M. Morris Mano. „Computer System Architecture” *Prentice Hall*, 1993.

[Rose89] W. Rosenstiel, R.Camposano. „Rechnergestützter Entwurf hochintegrierter Schaltungen“ *Springer-Verlag*, 1989.

[Spar92] „The SPARC Architecture Manual, Version 8” *Prentice Hall*, 1992.

[Syno92] Synopsys Inc. „Design Compiler Reference Manual” *Synopsys Inc.*, 1993.

Die in dieser Arbeit gewonnenen Erkenntnisse über das Zeitverhalten sowie über die benötigte Fläche einer realen Schnittstelle zwischen Hardware- und Softwaremodulen erlauben eine genauere Bewertung bei der Auswahl von Softwaremodulen, die in Hardware implementiert werden sollen (etwa mit der in [Hard94] beschriebenen Kostenfunktion), da die durch das Interface verursachten Kosten bisher nur geschätzt werden konnten. Weiterhin ist geplant, die SFU mit Hilfe eines ASIC-Emulators („InCA“-System) in einem SPARC-System einzusetzen und dadurch Erfahrungen beim Einsatz von HW/SW-Codesign unter Echtzeitbedingungen zu sammeln.

5. Zusammenfassung und Ausblick

Es wurde für eine SPARC-Architektur ein Coprozessor zur effizienten Anbindung von „Special Function Hardware“ entworfen und implementiert. Besonderer Wert wurde dabei auf hohen Datendurchsatz, geringe Größe und weitgehende Unabhängigkeit von den einzusetzenden Hardwaremodulen gelegt.

Der hohe Datendurchsatz wird durch die in der SPARC-Definition bereits vorgesehene Einbindung des Coprozessors erreicht. Dies ermöglicht ein einfaches Übertragungsprotokoll und die problemlose Mitbenutzung von Cache-RAM und MMU. Da i. a. kein Zugriff auf den M-Bus notwendig ist, erfolgt der Datenaustausch mit der Integer-Unit in der Regel ohne Verzögerung durch ungültige Daten im Cache-RAM. Durch den SFU-internen Datentransfer zwischen den SFHWs ist in einigen Fällen kein externer Speicherzugriff nötig. Weiterhin wurde mit den Registermonitor in Abschnitt 2.4 ein Mechanismus vorgestellt, der einen möglichst frühzeitigen Start der SFHW ermöglicht und außerdem den Zeitaufwand für die Steuerung minimiert.

Ein geringer Bedarf an Chipfläche wird dadurch erreicht, daß der Befehlssatz klein gehalten und die bereits durch die SPARC-Definition vorhandene Coprozessorschnittstelle ausgenutzt wird. Da für mehrere SFHWs nur eine Steuerung (Befehlspipeline) benötigt wird und die für eine SFSU notwendigen Komponenten individuell nach Bedarf erzeugt werden (z. B. nur die benötigte Registerbreite), wird der Platzbedarf auf ein Minimum beschränkt. Wie in Abschnitt 4.4 gezeigt, ist die gesamte Steuerung tatsächlich bedeutend kleiner als typische SFHWs.

Die notwendige Flexibilität wird durch den modularen Aufbau erreicht. Dadurch ist die Zahl der SFHWs sowie deren Ein- und Ausgänge (in Grenzen) frei wählbar. Es besteht außerdem die Möglichkeit, die SFSUs an andere Prozessoren anzuschließen, da die SFSUs nur über den SFU-Bus mit der SFU-Steuerung verbunden ist. Lediglich die SFU-Steuerung muß an die andere Prozessorarchitektur angepaßt werden.

Es sind weitere Verbesserungen denkbar, deren Realisierung jedoch den zeitlichen Rahmen dieser Studienarbeit sprengen würden. Um ohne großen zusätzlichen Aufwand Funktionen mit mehr als 31 Operanden zu benutzen zu können, sind verschiedene Methoden möglich. Zum einen könnte ein Operandenregister durch beliebig viele, als „Schlange“ angeordnete Register (FIFO) erweitert werden, wobei durch sukzessives Beschreiben des Operandenregisters sämtliche Operanden in die Register gelangen. Eine andere Möglichkeit ist die Erweiterung des Registermonitors, so daß ein Operandenregister wiederbeschreiben werden kann, nachdem sein Wert von der SFHW ausgelesen wurde.

Bei der SFU-Steuerung fehlen Mechanismen, welche bei einem Kontextwechsel (z. B. in einer Multitasking-Umgebung) benötigt werden, daher ist es bisher nur möglich, eine SFHW von einem Prozeß zu benutzen. Dieses Problem läßt sich aber durch Einsatz von Semaphoren für die einzelnen SFHWs auf der Ebene des Betriebssystems lösen.

Schaltung	LSI 10k	ecpd 10
gcd	1808	0,84
diffeq	42279	17,23
ellipticF	36661	15,27
SFU[a]	5319	1,63

Abb. 4.3: Flächenbedarf verschiedener Schaltungen

a. ohne SFHW, siehe Text

der SFU-Steuerung gegenüber den SFHWs nochmals. Die Forderung nach dem geringen Bedarf an Chipfläche ist somit erfüllt.

$$A_{SFU} = A_{SFU\_control} + \sum_{s=0}^{n_{SFHW}-1} A_{SFSU,s}$$

Dabei ist

$$A_{SFSU,s} = A_{SFSU\_control} + A_{SFHW,s} + n_{OpReg,s} \cdot A_{OpReg} + n_{ResReg,s} \cdot A_{ResReg}$$

die Fläche der SFSU mit der Nummer s.

Die Werte n_{SFHW}, $n_{OpReg,s}$, $n_{ResReg,s}$ sind durch die Beschreibung der SFHWs bekannt und in der Datei „SFHW_descriptions.vhdl" definiert. Die Größen der einzelnen Komponenten lassen sich mit dem „Design Analyzer" ermitteln und sind in Abb. 4.2 angegeben. Die Werte können etwas variieren, da z. B. bei den Registern je nach Adresse eine verschiedene Dekodierlogik benutzt wird. Die Abweichungen sind jedoch minimal. Unbedingt beachtet werden sollte jedoch, daß die für die Verdrahtung benötigte Fläche nicht berücksichtigt wird. Diese kann erst bestimmt werden, nachdem aus der synthetisierten Schaltung das Layout erstellt wurde.

	LSI 10k	**ecpd 10**
$A_{SFU\_control}$	3456	1,392
$A_{SFSU\_control}$	46	0,023
A_{OpReg}	$10b + 15$ [a]	$(4{,}25b + 6)\ 10^{-3}$
A_{ResReg}	477	0,192
Einheit	Gatter	mm^2

Abb. 4.2: Größe der SFU-Komponenten

a. b ist die Wortbreite des Registers

Um den Platzbedarf der SFU-Steuerung gegenüber der SFHW zu beurteilen, werden die in [Hard93] aufgeführten Beispiele betrachtet. Dort werden verschiedene Algorithmen angegeben, die als SFHW eingesetzt werden könnten (Größter gemeinsamer Teiler (ggt), Lösen einer Differentialgleichung (diffeq), elliptischer Filter (ellipticF)). Die Beispiele wurden an die SFU angepasst (Steuerung durch Registermonitor) und mit den Bibliotheken „LSI 10k" und „ecpd 10" synthetisiert, um vergleichbare Daten zu erhalten. Die ermittelten Größen der Hardwaremodule werden in Abb. 4.3 angegeben. Als Beispiel für die SFU wird eine SFU mit einer SFHW, vier Operandenregistern (je 32 Bit breit) und einem Ergebnisregister gewählt.. Für die SFU (ohne SFHW) ergibt sich eine Fläche von 5319 Gattern bzw. 1,63 mm^2.

Der Vergleich mit den Werten aus Abb. 4.2 zeigt, daß die Fläche der SFU ohne SFHW im allgemeinen deutlich kleiner ist als die der SFHW. Dabei ist zu beachten, daß hier nur eine SFHW eingebaut wurde. Normalerweise sind mehrere SFHWs in einer SFU untergebracht, wobei die SFU-Steuerung nur einmal benötigt wird. Dadurch sinkt der relative Platzbedarf

werden als „Pad“ gekennzeichnet, damit bei der Synthese ein entsprechendes I/O-Pad (z. B. mit Treibern) eingefügt wird.

Der Entwurf wird nun nach den Vorgaben optimiert und auf die gewählte Zellen-Bibliothek abgebildet. Das geschieht mit dem „compile“-Befehl. Die Option „-boundary_optimization“ erlaubt das Optimieren über die Grenzen der Komponenten hinaus. Dadurch wird z. B. überflüssige Logik bei einem unbeschalteten Ausgang erkannt. Vor und nach der Optimierung wird das Design abgespeichert.

Mit Hilfe des Syntheseprogramms kann das fertige Design untersucht werden. Dazu lassen sich z. B. Informationen über das Zeitverhalten oder die benötigte Fläche ausgeben und die Einhaltung der Vorgaben überprüfen. Zur Weiterverarbeitung mit CAD-Programmen läßt sich der Entwurf in verschiedenen Formaten speichern.

4.4 Ergebnisse

In diesem Abschnitt soll auf das Zeitverhalten und die Fläche der synthetisierten SFU eingegangen werden.

Zunächst zum Zeitverhalten. Hier interessiert vor allem die Verzögerung des kritischen Pfades. Mit dem „Design Analyzer“ läßt sich diese Verzögerung bestimmen. Dadurch erhält man die in Abb. 4.1 angegebenen Werte. Die „setup-time“ ist dabei die Zeit, welche das Signal vor der Taktflanke bereits stabil an den Registern anliegen muß. Dieser Wert steht in der entsprechenden Bibliothek. Das durch das Taktsignal von 25 Mhz vorgegebene Zeitlimit wurde nicht überschritten. Es ist sogar möglich, die mit der „ecpd 10“-Bibliothek synthetisierte SFU mit einem Takt von 45 MHz zu betreiben, was einen Einsatz mit der 40 Mhz-Version der Integer Unit zuläßt (sofern die eingesetzten SFHWs für diese Frequenz geeignet sind).

Bibliothek	**kritischer Pfad**	**setup time**	**max. Taktfrequenz**
LSI 10k	11,04 ns	0,8 ns	40,9 MHz
ecpd 10	8,86 ns	2,25 ns	45,0 MHz

Abb. 4.1: kritischer Pfad

Die Größe der SFU A_{SFU} läßt sich nicht als absoluter Zahlenwert angeben, da sie von den eingesetzten SFHWs abhängt. Es läßt sich jedoch zur Berechnung der Größe folgende Formel ansetzen, wobei

A_x die Größe der Komponente x,

n_{SFHW} die Anzahl der SFHWs,

$n_{OpReg,\,s}$ die Anzahl der Operandenregister der SFSU s und

$n_{ResReg,\,s}$ die Anzahl der Ergebnisregister der SFSU s angibt :

4.3 Durchführung

Der Synthesevorgang besteht aus mehreren Schritten. Zunächst werden die VHDL-Beschreibungen eingelesen und in technologieunabhängige und nicht optimierte Netzlisten übersetzt. Dann werden Vorgaben für die Optimierung definiert. Schließlich wird die Schaltung nach den Vorgaben optimiert und auf Elemente einer Bibliothek abgebildet.

Für die Durchführung dieser Schritte wurde ein Skript erstellt. Damit entsteht aus den VHDL-Beschreibungen der SFU die entsprechende Schaltung in einer bestimmten Technologie.

Es werden für die SFU zwei verschiedenen Technologien verwendet, wobei für die Synthese mit jeder Technologie jeweils ein eigenes Skript verwendet wird.

- lsi_10k

  Dies ist eine Bibliothek der Firma LSI für ein Gate-Array, welches 10.000 Gatter (10k) umfaßt. Um die SFU mit dem „InCA"-System, einem ASIC-Emulator (siehe Kapitel 5) zu emulieren, muß diese Bibliothek verwendet werden. Selbstverständlich ist damit auch eine Fertigung der SFU als Gate-Array möglich.

- ecpd_10

  Diese Bibliothek wird für die Herstellung der SFU benutzt. Hierfür stehen weitere CAD-Tools zur Verfügung, mit denen das Layout erstellt werden kann. Anschließend ist eine Fertigung möglich, z. B. im Rahmen des „Eurochip"-Projektes .

Im folgenden sollen auf die einzelnen Schritte der Synthese erläutert werden.

Zunächst werden die VHDL-Beschreibungen eingelesen. Dabei muß die in Abb. 3.1 angegebene Hierarchie berücksichtigt werden. Anschließend werden durch den Befehl „uniquify" die generischen Beschreibungen (z. B. der Register) an den Stellen eingesetzt, wo sie benötigt werden. Die Komponenten werden dabei entsprechend oft kopiert und mit jeweils individuellen Namen versehen.

Bevor die Optimierung stattfindet, werden nun verschiedene Parameter gesetzt, welche die Vorgaben für die Ziele der Optimierung darstellen. Der „Design Compiler" versucht, die im Rahmen der Vorgaben kleinstmögliche Schaltung zu erzeugen.

Zuerst wird die maximale Taktfrequenz und der Takteingang der SFU definiert. Dabei wird von einer Frequenz von 25 Mhz ausgegangen. Das entspricht der 25 MHz-Version der Integer Unit (CY7C601-25). Man erhält eine Taktperiode von 40 ns, welche laut Datenbuch ein Impuls-Pause-Verhältnis von 1:1 besitzten soll. Bei der Optimierung der Schaltung wird nun versucht, die Verzögerung des kritischen Pfades ausreichend klein zu halten.

Anschließend wird Ausgangslastfaktor (Fanout) und der Eingangslastfaktor (Fanin) definiert. Es wird ein Fanout von 3 und ein Fanin von 1 gewählt. Die externen Anschlüsse der SFU

4. Synthese

4.1 Einführung

Der Ablauf von der Spezifikation zum fertigen Chip sieht etwa folgendermaßen aus:

1. Entwurf
2. Beschreibung (z.B. in VHDL)
3. Simulation der Beschreibung
4. Synthese
5. Abbilden des Entwurfs auf Zellen einer bestimmten Technologie („Mapping“)
6. Layout-Generierung
7. Chip-Fertigung

Nachdem die ersten drei Schritte bereits in den Kapiteln 1 und 2 behandelt wurden, wird in diesem Kapitel die Durchführung der Schritte 4 und 5 für die entworfene SFU beschrieben.

Die Synthese selbst besteht aus zwei wesentlichen Prozessen: Übersetzung und Optimierung. Zunächst wird die Beschreibung des Entwurfs in Netzlisten auf Gatterebene übersetzt und anschließend nach bestimmten Vorgaben optimiert [Camp91, Syno92]. Die Erstellung des Layouts sowie die Fertigung der integrierten Schaltung werden in dieser Arbeit nicht behandelt, ausführliche Informationen dazu finden sich z. B. in [Rose89].

4.2 Verwendete Software

Zur Synthese wird das Programmpaket „Design Compiler“ der Firma „Synopsys Inc.“ benutzt.

Mit diesem Tool ist es möglich, Hardware-Entwürfe in verschiedenen Beschreibungssprachen einzulesen und in Netzlisten auf Gatterebene zu übersetzen. Diese Netzlisten können anschließend bezüglich Zeitverhalten, Fläche oder Leistungsaufnahme optimiert werden und auf eine Zielbibliothek, welche Elemente (Zellen) einer bestimmten Technologie enthält, abgebildet werden. Dieser optimierte Entwurf kann abschließend in verschiedenen Formaten wieder gespeichert werden und mit anderen (z. B. CAD/CAE-) Tools weiterverarbeitet werden. Es werden die zur Realisierung benötigten Zellen gesucht und der „Schaltplan“ erstellt (Netzliste). Mit weiteren Tool wie z. B. „Cadence“ oder „Mentor“ wird aus der Netzliste das Layout generiert, welches zur Herstellung der integrierten Schaltung verwendet wird.

Das Programm „Design Analyzer“ bietet eine grafische Oberfläche zum „Design-Compiler“ und ermöglicht die interaktive Durchführung des Syntheseablaufs. Das Programm kann auch über eine Kommandozeilen-Schnittstelle („dc_shell“) bedient werden. Über Skripten ist eine Automatisierung des Syntheseablaufs möglich.

Das aus der Simulation gewonnene Zeitverhalten kann nun mit den Vorgaben aus dem Entwurf (Diagramme in Anhang B) verglichen werden. Dabei stellt sich heraus, daß die SFU in allen Testfällen korrekt arbeitet. Die Testfälle decken alle kritischen Bereiche ab, so daß nun die Synthese der SFU-Beschreibung vorgenommen werden kann.

```
cmove %c0,%c0           ! CMOVE-Befehl mit ungültigen Daten (Pipeline
                        ! anhalten)
stc   %c1,[%r5+1]       ! SFU-Store-Befehl mit ungültigen Daten

add   %r31,0,%r31       ! zwei IU-Befehle, um die vorhergehenden Befehle
                        ! abzuschließen
sub   %r31,0,%r31

crst  %c1               ! CRST-Befehl mit gültigen Daten

add   %r31,0,%r31       ! zwei IU-Befehle, um die vorhergehenden Befehle
                        ! abzuschließen
sub   %r31,0,%r31

ld    [%r31],%r1        ! Load-Befehl für Integer-Unit, danach zwei SFU
                        ! Befehle, um Behandlung der CINS1- und CINS2-
                        ! Signale zu testen (vgl. [Cypr90], Fig. 3-7)
cldc -73,%c0            ! erster SFU-Befehl (CINS2 gesetzt)
cldc  6,%c1             ! zweiter SFU-Befehl (CINS1 gesetzt)

add   %r31,0,%r31       ! zwei IU-Befehle
sub   %r31,0,%r31

stc   %c1,[%r5+1]       ! STC-Befehl, um Verhalten bei externem HOLD-
                        ! Signal zu testen

add   %r31,0,%r31       ! drei IU-Befehle zum Abschließen des
                        ! Testprogramms
sub   %r31,0,%r31
add   %r31,0,%r31
```

3.5.2 Ergebnisse

Mit dem VHDL-Simulator läßt sich der zeitliche Verlauf jedes Signals und jeder Variable im VHDL-Programm verfolgen. Dabei kann die Ausgabe grafisch erfolgen, sodaß eine übersichtliche Auswertung möglich ist. Im Anhang D ist das Ergebnis der Simulation für das vorgestellte Testprogramm beigefügt. Da in der SFU-Beschreibung zu viele verschiedene Signale vorkommen, um sie in ein Diagramm darzustellen, sind nur die wichtigsten aufgeführt. Das sind zunächst die Signale, welche vom Testprogramm erzeugt werden und an den externen Anschlüssen der SFU angelegt werden (CLK, D, INST, CINS1, CINS2, BHOLD). Weiterhin werden die Ausgänge der SFU angezeigt (D, CNULL, CHOLD, CCC). Um den Befehlsfluß in der Pipeline zu überprüfen, werden die Befehlsregister der drei Pipeline-Stufen gezeigt (INST_SEL für die Decode-Stufe, INST für die Execute-Stufe und INSTR für die Write-Stufe). Außerdem sind alle Signale des SFU-Bus aufgeführt.

wie in [Cypr90] beschrieben. Diese Beschreibung reagiert an den kritischen Stellen korrekt auf das „CHOLD“-Signal der SFU und stoppt den Befehlsfluß auf dem Datenbus. Am Ende des Programms wird zusätzlich die Funktion der CINS1- und CINS2-Signale sowie das Anhalten der Pipeline durch ein externes „HOLD“-Signal simuliert.

Es werden zwei SFHWs benutzt, welche jeweils zwei Operanden benötigen und zwei Ergebnisse liefern. Beide SFHWs führen lediglich einfache Funktionen aus (Datentransfer von den Operanden zu den Ergebnisregistern). Die SFHW Nr. 0 berechnet ihr Ergebnis sofort während die SFHW Nr. 1 die Ergebnisse nach 6 bzw. 10 Taktzyklen bereitstellt, um das Verhalten der SFU bei ungültigen Daten zu simulieren.

```
add   %r31,0,%r31          ! zwei IU-Befehle
sub   %r31,0,%r31

csel  WREN+RDEN+RESET+0    ! Konstante nach Adresse 31 schreiben, um SFSU
                           ! Nr.0 auszuwählen.
cldc  170,%c0              ! Konstante 170 in das SFU-Register 0 schreiben
ldc   [%r5],%c1            ! Wert aus dem Speicher (vom Datenbus in SFU-
                           ! Register 1 schreiben

add   %r31,0,%r31          ! drei IU-Befehle, um die Ausführung des LDC-
                           ! Befehls beobachten zu können
sub   %r31,0,%r31
add   %r31,0,%r31

stc   %c0,[%r5+1]          ! SFU-Store-Befehl mit gültigen Daten

add   %r31,0,%r31          ! drei IU-Befehle, um die Ausführung des STC-
                           ! Befehls beobachten zu können
sub   %r31,0,%r31
add   %r31,0,%r31

csel  WREN+RESET+1         ! Unterschiedliche SFSUs zum Lesen und Schreiben
                           ! auswählen (lesen von SFSU 0 und schreiben in
                           ! SFSU 1)
cmove %c0,%c0              ! Wert aus Ergebnisregister 0 in SFSU 0 nach
                           ! Operandenregister 0 in SFSU 1 kopieren (mit
                           ! gültigen Daten)
cldc  -73,%c1              ! Vorzeichenerweiterung beim CLDC-Befehl testen
                           ! und Register 1 laden
csel  RDEN+1               ! SFSU 1 nun auch zum Lesen selektieren

add   %r31,0,%r31          ! IU-Befehl einschieben

crst  %c0                  ! Status von Ergebnisregister 0 abfragen (Datum
                           ! ist nicht gültig)
```

```
case adr is
    when 0 => [Implementierung der SFHW Nr. 0]
    when 1 => [Implementierung der SFHW Nr. 1]
     [...]
    when others => null;
end case;
```

Hier wird für jede SFHW die entsprechende Beschreibung eingefügt. Als Parameter werden dazu folgende Daten übergeben:

input: Ein Vektor mit allen Operanden der SFHW. Jeder Operand ist 32 Bit breit, wovon jedoch je nach Schnittstellendefinition nicht alle benutzt werden.

valid: Der Registermonitor für die Operandenregister (RM_{op}).

output: Ein Vektor mit allen Ausgängen der SFHW. Wenn hier ein gültiges Ergebnis anliegt, muß das entsprechende Bit des Vektors „write“ auf ’1’ gesetzt werden.

write: Das i-te Bit dieses Vektors signalisiert durch eine ’1’, daß am i-ten Ausgang der SFHW gültige Daten anliegen. Das Datum am Ausgang wird bei der nächsten steigenden Taktflanke übernommen, alle darauffolgenden Änderungen werden ignoriert.

CLK: Der Systemtakt.

reset: Dieses Signal liegt für einen Taktzyklus an und dient dazu, die SFHW in den Anfangszustand zurückzusetzen.

3.5 Simulation

3.5.1 Testprogramm

Mit dem VHDL-Simulationsprogramm „vhdldbx“ der Firma „Synopsys“ wird nun die Funktion der SFU und die Einhaltung des vorgegebenen Zeitverhaltens überprüft. Zu diesem Zweck wurde ein Testprogramm erstellt, welches sämtliche SFU-Befehle beinhaltet. Bei den STC- und CMOVE-Befehlen wird sowohl der Fall berücksichtigt, daß die Daten von der SFHW bereits vorliegen als auch der Fall, daß die SFHW noch nicht terminiert ist und die Pipeline angehalten wird (siehe Abschnitt 1.2). Das Programm ist weiter unten unter Verwendung der in [Spar92] vorgeschlagenen Assemblersyntax abgedruckt. Dort werden die einzelnen Befehle kommentiert. Neben den SFU-Befehlen werden einige Befehle für die Integer-Unit benötigt. Diese dienen zur Überprüfung der Decode-Stufe (Generierung von CNOP-Befehlen) und erlauben eine einfachere Überprüfung der Simulationsergebnisse durch zeitliche Abstände zwischen den SFU-Befehlen. Allerdings werden ebenfalls mehrere SFU-Befehle direkt nacheinander ausgeführt, um das korrekte Verhalten bei aufeinanderfolgenden SFU-Befehlen zu überprüfen.

Das Programm wurde in eine VHDL-Beschreibung umgesetzt („SFU_test.vhdl“), welche sämtliche externen Signale der SFU steuert. Dabei wurde ein Zeitverhalten implementiert,

Um die als Datentyp „Integer“ übergebene Adresse mit dem Adreßeingang vergleichen zu können, welcher vom Datentyp „bit_5“ ist, wird die Registeradresse zunächst in einen Bit-Vektor konvertiert. Das geschieht in Zeile 83 mit Hilfe einer Funktion aus dem „IEEE.std_logic_1164“-Package. Das direkte Einsetzen des Funktionsaufrufes in den Vergleich (Zeile 96) führt bei der Simulation zu Problemen (der Ausdruck ist immer „FALSE“).

Da alle Aktionen des Registers auf positive Taktflanken getriggert sind, besteht die Beschreibung aus einem einzelnen, einfachen Prozeß.

3.4.4.3 Ergebnisregister

Die Beschreibung besteht aus zwei Prozessen. Der erste ist auf steigende Taktflanken getriggert. Dort werden die Daten vom Eingang in den internen Speicher geschrieben, wenn der „write“-Eingang aktiv ist und außerdem die Leseadresse vom Bus um zwei Taktzyklen verzögert wird, wie auch in den Timing-Diagrammen angegeben. Dies geschieht durch ein 2-Bit-Schieberegister („delay“). In Zeile 105 wird das Register in jedem Taktzyklus verschoben, während in den Zeilen 118-125 je nach anliegender Adresse dem niederwertigsten Bit eine ’0’ oder ’1’ zugewiesen.

Im zweiten Prozess, welcher kombinatorische Logik darstellt, wird das höherwertige Bit des Schieberegisters ausgewertet (Zeile 135) und ggf. das gespeicherte Datum auf den SFU-Bus gelegt. Durch die beiden „if then else“-Konstrukte in diesem Prozeß werden Tri-State-Treiber nachgebildet. Wenn die Wortlänge des Registers kleiner als 32 ist, so werden die oberen Bits vorzeichenrichtig erweitert.

3.4.5 SFHW

Eine Vorgabe beim Entwurf der SFU war die Unabhängigkeit von den benutzten SFHWs. Es können daher SFHWs mit einer (innerhalb des zulässigen Bereiches) beliebigen Zahl von Eingängen und Ausgängen eingesetzt werden. Die Breite der Ein- und Ausgänge kann von 1 bis 31 Bit individuell für jeden Anschluß bestimmt werden. Dadurch werden für die Operanden- und Ergebnisregister nicht mehr Flip-Flops benutzt als notwendig sind.

Die Definition der Schnittstellen für die verschiedenen SFHWs werden in der Datei „SFHW_descriptions.vhdl“ vorgenommen. Dort wird zunächst die Konstante „n_SFHW“ definiert, welche die Anzahl der SFHWs (und somit auch der SFSUs) vorgibt. Anschließend werden in den Feldern „n_inputs“ und „n_outputs“ die Zahl der Eingänge bzw. Ausgänge der SFHWs eingetragen. Die Breite jedes Eingangs wird in dem Feld „op_width“ für jede SFHW eingetragen. Dabei müssen aufgrund der VHDL-Syntax für jede SFHW 32 Werte angegeben werden, wobei die nicht benötigten Angaben ignoriert werden. Das Feld „res_width“ wird entsprechend für die Definition der Bitbreiten aller Ausgänge benutzt.

Die Implementierung der SFHWs selbst wird in der Datei „SFHW.vhdl“ vorgenommen. Hier kann auch die Einbindung der SFHWs als weitere „Entities“ erfolgen. Der Rahmen dieses Programmteils bildet eine Fallunterscheidung:

wird der Ausgang zum Datenbus hochohmig geschaltet („D <= Z_vector"). „Z_vector" ist in „SFU_Types" definiert und besteht aus 32 Bits mit dem Wert „Z" (hochohmig).

In dem auf fallenden Taktflanken synchronisierten Prozeß in Zeile 106 wird der Befehl dekodiert und die Signale des SFU-Bus entsprechend gesetzt. Tritt nach einem STC-Opcode im darauffolgenden Taktzyklus ein weiterer STC-Opcode auf, so handelt es sich um den „Internal Opcode" STC-1 und es wird das CNULL-Signal gesetzt.

3.4.4 SFSU

Diese Beschreibung erzeugt die erforderliche Anzahl von Operanden- und Ergebnisregistern sowie die SFSU-Steuerung und verbindet diese Komponenten untereinander und mit der entsprechenden SFHW. Da in einer SFU i. d. R. mehrere SFSUs vorhanden sind, wird die SFSU über ihre Adresse „adr" identifiziert.

Die Adresse wird zusammen mit den Anschlüssen (port) als Parameter übergeben (Zeile 48). Für die Verbindungen zwischen den Registern und der SFHW werden Felder von Bit-Vektoren benutzt. Da die Anzahl der Register von der Zahl der Ein- und Ausgänge der SFHW abhängt, wird die jeweilige Anzahl aus der SFHW-Beschreibung in der Datei „SFHW_descriptions.vhdl" benutzt. Die Signale sind immer 32 Bit breit, obwohl die Breite der SFHW-Anschlüsse individuell festgelegt werden kann. Das ist notwendig, um ein Feld von Bit-Vektoren benutzen zu können, da die Elemente eines Feldes immer gleichen Typs sein müssen. Bei der Synthese der Schaltung entfallen die überflüssigen Leitungen jedoch durch automatische Optimierung.

Den Registern wird neben ihrer individuellen Adresse auch die notwendige Breite (in Bits) als Parameter übergeben (Zeilen 101 und 108). Dadurch werden die Register in der notwendigen Wortlänge generiert.

In der SFSU-Beschreibung wird weiterhin ein UND-Gatter benötigt, um die „valid"-Bits der Ergebnisregister zu verknüpfen (siehe Abb. 2.8). Dieses Gatter, dessen Zahl der Eingänge gleich der Zahl der Ergebnisregister sein muß, wird in dem Prozeß ab Zeile 120 durch eine Verhaltensbeschreibung erzeugt.

3.4.4.1 SFSU-Steuerung

Hier wird in zwei Prozessen ausschließlich kombinatorische Logik beschrieben.

3.4.4.2 Operandenregister

Da jedes Register über eine eigene Adresse angesprochen werden kann und eine individuelle Wortlänge der zu speichernden Daten hat, werden die Adresse und die Breite des Datenwortes als Parameter erwartet (Zeile 48).

3.4.3.1 Decode-Stufe

Bei dieser Beschreibung handelt es sich um eine Verhaltensbeschreibung, die aus drei parallel laufenden Prozessen besteht.

Der erste Prozess (Zeilen 86-102) ist auf steigende Taktflanken getriggert. Das heißt, die Inhalte von „Inst_1“ und „Inst_2“ ändern sich nur bei steigender Taktflanke (und wenn gleichzeitig die Bedingung „INST='1' and HOLD='0'“ erfüllt ist).

Im zweiten Prozeß wird bei fallender Taktflanke die Adresse des zu lesenden Registers auf den SFU-Bus gelegt, sofern das HOLD-Signal nicht gesetzt ist (Zeilen 105-122). Dabei liegt die Adresse je nach Opcode an verschiedenen Stellen im Befehlswort (siehe Abb. 2.15).

Der dritte Prozess (ab Zeile 125) beschreibt hingegen eine kombinatorische Logik, die sofort auf Signaländerungen reagieren soll. In der Simulation wird dieser Prozess wie eine Endlosschleife ausgeführt. In Zeile 156 steht daher ein „wait“-Befehl; wird dieser nicht eingefügt, so würde der Simulator keine anderen Prozesse mehr abarbeiten. Es könnte daher keine Simulation durchgeführt werden. In der Synthese wird diese Beschreibung in reine kombinatorische Logik umgesetzt.

In den Zeilen 146-150 wird für den STC-Befehl der IOP erzeugt. Falls kein SFU-Befehl ausgeführt werden soll, wird die Pipeline mit CNOP-Operationen gefüllt (Zeilen 137-139), damit in den folgenden Stufen nur eine kleine Anzahl von definierten Befehlen dekodiert werden muß.

Die symbolischen Bezeichnungen für die Opcodes werden in der Datei „SFU_Types.vhdl“ definiert.

3.4.3.2 Execute-Stufe

Diese Beschreibung besteht aus drei Prozessen: jeweils einen auf steigender (Zeile 87) bzw. fallender (Zeile 102) Taktflanke getriggert sowie die Beschreibung von kombinatorischer Logik (Zeile 117).

Der erste Prozeß dient lediglich zur Weiterleitung des Befehls an die Write-Stufe.

Im zweiten Prozeß wird bei einem CRST-Befehl das Bit 1 der „Condition Codes“ auf den richtigen Wert gesetzt.

Der dritte Prozeß setzt das CHOLD-Signal, falls bei einem STC- oder CMOVE-Befehl die Daten noch nicht bereitstehen.

3.4.3.3 Write-Stufe

Hier sind alle Aktionen mit dem Takt (CLK) synchronisiert. Bei steigender Taktflanke (Zeile 85) wird das Befehlswort von der Execute-Stufe übernommen und bei einem STC-Befehl die Daten vom SFU-Bus auf dem externen Datenbus ausgegeben. Bei allen anderen Befehlen

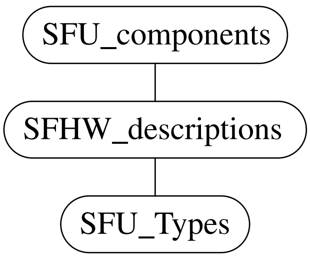

Abb. 3.2: Verwendete Typ-Definitionen

Beschreibungen, welche die hierarchische darunterliegenden Komponenten verbinden und eine Schnittstelle zur höheren Ebene bereitstellen (siehe Abschnitt 3.3). Die „Blätter“ des „Hierarchiebaumes“ in Abb. 3.1 sind Verhaltensbeschreibungen der verschiedenen Komponenten.

In den VHDL-Texten werden die Signale in den Schnittstellen nach den Entwürfen in Kapitel 2 benannt. Da in VHDL nur alphanumerische Zeichenketten als Bezeichner zulässig sind, werden „low-aktive“ Signale durch ein vorangestelltes „n“ gekennzeichnet. Das Signal „nCHOLD“ ist also gleich „$\overline{\text{CHOLD}}$“. Durch diese Konvention läßt sich im Quelltext sofort feststellen, ob ein Signal low- oder high-aktiv ist.

Sämtliche VHDL-Beschreibungen sind im Anhang C abgedruckt.

3.4.2 SFU

Die Schnittstelle dieser Beschreibung repräsentiert die externen Anschlüsse der SFU (Zeilen 47-64).

In dieser Beschreibung wird die SFU-Steuerung mit der Befehlspipeline (SFU_control) sowie die benötigte Anzahl von SFSUs mit den externen Anschlüssen verbunden. Jeder erzeugten SFSU wird in der Schleife eine individuelle Adresse (i) zugewiesen, anhand der sie unterschieden werden kann.

Da die Verbindung der SFU-Steuerung mit den SFSUs über interne Signale vorgenommen wird (u. a. über den SFU-Bus), werden die Signale deklariert (Zeilen 77-81). Diese Signale repräsentieren die physikalischen Leitungen.

3.4.3 SFU-Steuerung

Die SFU-Steuerung faßt die drei Stufen der Pipeline zusammen. Das in der Schnittstelle als Ausgang definierte Signal „CHOLD“ kann nicht gelesen werden, wird aber von der Decode-Stufe als Eingang benötigt. Daher wird das Signal „int_CHOLD“ zusätzlich als internes Signal deklariert, um Decode- und Execute-Stufe sowie die Schnittstelle zu verbinden.

„vhdldbx“ ermöglicht es, den Quellcode während der Abarbeitung zu beobachten, Eingangssignale zu setzen und Signalverläufe grafisch darzustellen.

3.3 Hierarchische Struktur

In Abb. 3.1 ist die Aufteilung der SFU-Beschreibung in einzelne Komponenten gezeigt. Für jede Komponente („entity“) existiert eine eigene VHDL-Beschreibung. Die weiter unten stehenden Komponenten werden von den darüberliegenden Modulen benutzt und müssen daher vor diesen übersetzt werden.

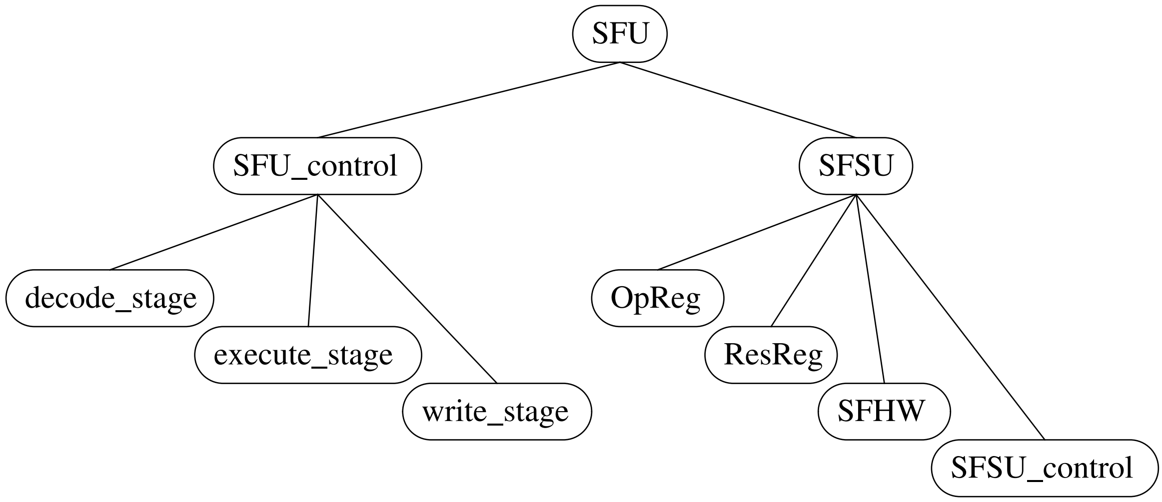

Abb. 3.1: Hierarchische Gliederung der SFU

Die in Abb. 3.2 abgebildeten Typ-Definitionen werden von allen Komponenten verwendet und müssen daher bei der Übersetzung der übrigen Befehle bereits zur Verfügung stehen. In „SFU_Types“ werden die benötigten Datentypen definiert. „SFHW_descriptions“ enthält Informationen über die verwendeten SFHWs. Die Implementierung soll unabhängig von den eingesetzten SFHWs sein, daher müssen die Informationen darüber in einer Datei zusammengefaßt werden (siehe Abschnitt 3.4.5). Das sind die Zahl und Bitbreiten der Ein- und Ausgänge. Die Beschreibungen der SFHWs werden in der Datei „SFHW.vhdl“ vorgenommen. In „SFU_components“ werden die Schnittstellen aller Komponenten aufgelistet.

3.4 VHDL-Beschreibungen

3.4.1 Allgemeines

Im folgenden soll die Implementierung der SFU in VHDL erläutert werden. Wie bereits in Abschnitt 3.1 erwähnt, ermöglicht VHDL die Beschreibung eines Designs auf verschiedenen Abstraktionsebenen. Die Komponenten „SFU“, „SFU_control“ und „SFSU“ sind strukturelle

stück ersichtlich:
(„<=“ ist eine Signalzuweisung, „:=“ eine Variablenzuweisung)

```
a <= b;
b <= a;
```

Hier werden die Werte von a und b vertauscht, da beide Zuweisungen gleichzeitig ausgeführt werden.

```
a := b;
b := a;
```

In diesem Fall werden die beiden Zuweisungen sequentiell ausgeführt, dadurch enthalten a und b nach der Ausführung den gleichen Wert.

Eine in VHDL erstellte Beschreibung kann zunächst simuliert werden, um die Funktion zu überprüfen. Die so geprüfte Beschreibung der Schaltung kann synthetisiert werden. Das Ergebnis ist eine Schaltung auf Gatterebene, welche dann mit entsprechender CAD-Software weiterverarbeitet werden kann. In diesem Kapitel wird auf die Implementierung und Simulation der einzelnen SFU-Komponenten eingegangen, während sich das nächste Kapitel mit der Synthese befaßt.

Als Datentyp zur Beschreibung logischer Zustände ist in VHDL der Datentyp „Bit“ als Standard vorhanden. Da dieser jedoch nur zwei Zustände annehmen kann ('0' bzw. '1'), benötigt man für die Beschreibung realer Entwürfe eine Erweiterung dieses Typs. Dazu wurde ein Logiksystem mit 9 Zuständen entwickelt und von der IEEE normiert. In der VHDL-Package „STD_LOGIC_1164“ sind neben der Definition der Datentypen auch entsprechende Funtionen, z. B. zur Konvertierung, vorhanden.

Neben den Zuständen '0' und '1' ist z. B. auch 'U' für undefiniert und 'Z' für hochohmig definiert. Da die Datentypen („std_logic“) normiert sind, werden sie von den meisten VHDL-Simulatoren und Sytheseprogrammen unterstützt.

3.2 Verwendete Software

Zur Simulation und Synthese werden Programme von verschiedenen Herstellern angeboten. Eine der führenden Firmen ist „Synopsys Inc.“, deren Software zur Simulation und Synthese der SFU verwendet wird. Diese Tools wurden aufgrund ihrer Leistungsfähigkeit und Verfügbarkeit ausgewählt. Für die Simulation der VHDL-Beschreibungen wird der Übersetzer „vhdlan“ sowie das eigentliche Simulationsprogramm „vhdldbx“ benutzt. Die Programme werden unter dem Betriebssystem „UNIX“ auf einer SPARC-Workstation eingesetzt.

1. Eine Ausnahme bilden hier bidirektionale Signale, z. B. für Busse. Für diesen Fall können Tri-State-Ausgänge nachgebildet werden.

3. Implementierung

3.1 Die Sprache VHDL

Die Sprache VHDL ist eine der am weitesten verbreiteten Hardware-Beschreibungssprachen. Das resultiert u. a. daraus, daß die Sprache von der IEEE standardisiert wurde und dadurch der Austausch von VHDL-Beschreibungen zwischen Software-Produkten verschiedener Hersteller vereinfacht. Aus diesem Grund wird die SFU in VHDL beschrieben.

Hier soll zunächst eine kurze Einführung in VHDL gegeben werden. Dabei werden besonders die Unterschiede zu anderen Programmiersprachen erläutert. Für eine weitergehnde Sprachbeschreibung sei auf entsprechende Literatur verwiesen, wie z. B. [Berg92, Lips89].

VHDL unterscheidet sich von anderen Programmiersprachen wie Pascal oder C dadurch, daß es möglich ist, eine Beschreibung eines Hardwarebausteins zu erstellen, wobei die Besonderheiten von Hardware-Komponenten berücksichtigt werden können. In VHDL ist etwa die genaue Spezifizierung des zeitlichen Verhaltens einer Schaltung möglich, beispielsweise kann ein Signal für genau 2 ns auf den Wert '1' gesetzt werden oder eine Verzögerung (Signallaufzeit) bestimmt werden. Das ist insbesondere für die Nachbildung realer Schaltungen wichtig, mit denen z. B. neue Entwürfe simuliert werden können.

Die Beschreibung ist auf verschiedenen Abstraktionsebenen möglich. Die strukturelle Ebene ist z. B. für die Beschreibung von Netzlisten nützlich, während eine Verhaltensbeschreibung die Nachbildung einer Schaltung mit Konstrukten der herkömmlichen Programmiersprachen ermöglicht.

Eine VHDL-Beschreibung einer Komponente (entity) besteht aus der Beschreibung der „Anschlüsse" des Bausteins (port) und dem Innenleben (architecture). Eine Beschreibung kann wiederum in mehrere Komponenten unterteilt werden. Dadurch ist ein hierarchischer Aufbau möglich. Ebenso kann ein Baustein (z. B. ein Register) mehrfach verwendet werden. In der Schaltung wird es dann entsprechend oft dupliziert.

Um eine Beschreibung flexibel zu gestalten, können Parameter übergeben werden (generic entities). Diese Möglichkeit erlaubt es z. B., ein Register mit variabler Datenlänge zu beschreiben. Erst, wenn das Register eingesetzt wird, muß die tatsächliche Datenlänge festgelegt werden.

Für synchrone Entwürfe ist es wichtig, Aktionen durch steigende oder fallende Signalflanken zu triggern. Auch dafür stellt VHDL entsprechende Konstrukte zur Verfügung.

In VHDL-Programmen muß zwischen Signalen und Variablen unterschieden werden. Während Variablen nur lokal verwendet werden dürfen (in Verhaltensbeschreibungen), repräsentieren Signale physikalische Leitungen. Ein Signal ist i. a. gerichtet und besitzt nur eine Quelle sowie beliebig viele Senken[1]. Dafür kann aber mehreren Signalen gleichzeitig ein Wert zugewiesen werden. Der Unterschied zu Variablen ist aus dem folgenden Programm-

tor definiert. Es wurde gezeigt, daß damit eine Steigerung der Performance möglich ist, ohne eine Verlangsamung zu riskieren. Daher wurde der Registermonitor bei dem Entwurf verwendet.

Im folgenden Kapitel wird die Implementierung der Komponenten sowie eine Simulation der SFU beschrieben.

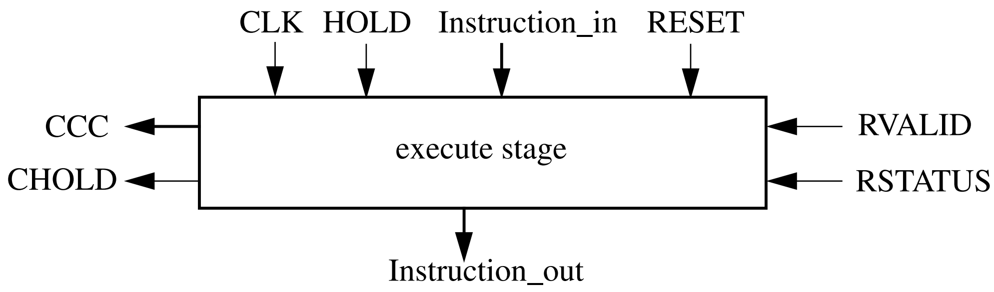

Abb. 2.20: Anschlüsse der Execute-Stufe

Mit dem HOLD-Eingang wird die Abarbeitung der Pipeline angehalten, über den „Instruction_out"-Ausgang wird der Befehl an die Write-Stufe weitergegeben, wo er im nächsten Taktzyklus weiter abgearbeitet wird.

2.7.7 Write-Stufe

In Abb. 2.21 sind die Anschlüsse der Write-Stufe dargestellt.

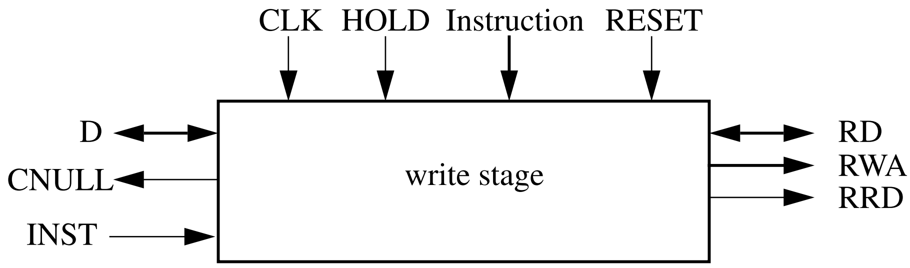

Abb. 2.21: Anschlüsse der Write-Stufe

Dies ist die letzte Stufe der Befehlspipeline. Sie ist dafür zuständig, die Daten vom Prozessor-Datenbus auf den internen SFU-Datenbus zu schreiben (beim „LDC"-Befehl) oder umgekehrt die Daten vom internen SFU-Datenbus auf den Prozessor-Datenbus zu legen (beim „STC"-Befehl). Beim „CLDC"-Befehl wird die im Befehlswort („Instruction") angegebene Konstante auf den SFU-Datenbus gelegt.

Weiterhin wird zusammen mit den Daten die Adresse des zu beschreibenden Operandenregisters („RWA") auf den SFU-Bus gelegt sowie das Signal „RRD" gesteuert.

2.8 Zusammenfassung

In diesem Kapitel wurde eine Aufteilung des Entwurfs in Steuerung und Sub-Units vorgenommen. Zur Steuerung der SFU wird eine Befehlspipeline verwendet, welche mit der Integer-Unit zusammenarbeitet. Die Anbindung der SFSUs erfolgt durch den internen SFU-Bus, wodurch die SFU-Steuerung unabhängig von der Zahl der tatsächlich verwendeten SFSUs und deren Register ist. Um den Datendurchsatz zu steigern, wurde ein Registermoni-

2.7.5 Decode-Stufe

In Abb. 2.19 sind die Anschlüsse der Decode-Stufe dargestellt..

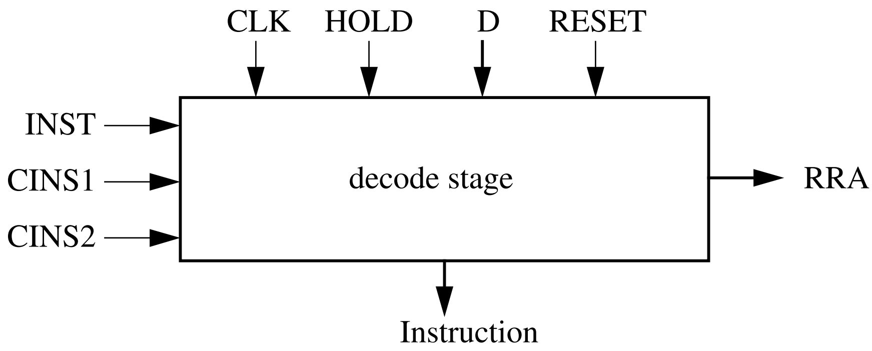

Abb. 2.19: Anschlüsse der Decode-Stufe

Diese erste Stufe der Befehls-Pipeline hat die Aufgabe, über den Eingang „D“ alle Befehle vom Datenbus mitzulesen und jeweils die letzten beiden gelesenen zu speichern. Auf dem Datenbus befindliche Befehle können durch das „INST“-Signal von anliegenden Daten unterschieden werden. Über die Signale „CINS1“ und „CINS2“ signalisiert die IU, ob es sich um einen Coprozessor-Befehl handelt und welcher der gepufferten Befehle abgearbeitet werden soll. Die Decode-Stufe sorgt nun dafür, daß die Adresse des Operandenregisters auf dem internen SFU-Bus bereitgestellt wird. Außerdem wird der abzuarbeitende Befehl an die Execute-Stufe weitergegeben, welche im nächsten Taktzyklus mit der Ausführung fortfährt.

Diese Stufe ist weiterhin dafür zuständig, daß bei Befehlen, welche mehrere Taktzyklen benötigen, die entsprechenden IOPs (internal opcodes) erzeugt werden (siehe Abb. 2.16). Falls kein Coprozessor-Befehl zur Abarbeitung ansteht, so wird intern der Befehl „CNOP“ erzeugt.

Der Eingang „HOLD“ dient dazu, die Befehlspipeline vorübergehend anzuhalten. Er wird benötigt, um festzustellen, ob die Adresse auf dem SFU-Bus (RRA) weiterhin anliegen muß (siehe Abb. D.6 und Abb. D.7).

2.7.6 Execute-Stufe

In Abb. 2.20 sind die Anschlüsse der Execute-Stufe dargestellt.

Aufgabe der Execute-Stufe ist es, die Statusbits „RVALID“ und „RSTATUS“ vom SFU-Bus zu lesen und als Coprozessor-Condition-Codes (CCC) der IU zur Verfügung zu stellen. Die Belegung der CCC-Bits ist in Abb. 2.18 angegeben. Falls der Inhalt des zu lesenden Ergebnisregisters beim „STC“ oder „CMOVE“-Befehl noch nicht gültig ist, wird über den „CHOLD“-Ausgang die Pipeline solange angehalten, bis das Ergebnis bereitsteht.

Signal	**Beschreibung**
D[31:0]	(*Eingang/Ausgang*) Datenbus, dieser Anschluß wird nur während eines SFU-Store Befehls als Ausgang geschaltet.
INST	(*Eingang*) Dieses Signal wird vom Prozessor gesteuert und zeigt an, daß ein Befehlswort auf dem Datenbus anliegt. Es wird benutzt, um die Befehle vom Bus mitzulesen und die jeweils letzten beiden Befehle zu speichern.
CNULL	(*Ausgang*) Dieses Signal signalisiert dem Speicher, daß die SFU die Befehls-Pipeline des Systems anhält. Dies geschieht, wenn $\overline{\text{CHOLD}}$ aktiviert wird (siehe dort).
CINS1 CINS2	(*Eingang*) Diese Signale werden von der IU während der Decode-Phase erzeugt. Es wird signalisiert, in welchem der zwei Befehls-Buffer der abzuarbeitende Befehl steht. Es wird maximal eines dieser Signale aktiviert.
CCC[1:0]	(*Ausgang*) Diese Bits beinhalten den Zustand einer SFSU oder eines Ergebnisregisters nach einem CRST-Befehl. Der CBccc-Befehl benutzt diese Bits. **Bit** — **Bedeutung** CCC[0] — Eine ‘1’ bedeutet, daß die zuletzt zum Lesen selektierte SFSU fertig ist und das komplette Ergebnis bereitsteht. CCC[1] — Das Ergebnis des letzten CRST-Befehls, eine ‘1’ bedeutet, daß der Wert im angegebenen Register gültig ist.
CCCV	(*Ausgang*) Das Signal zeigt, ob die CCC[1:0]-Bits gültig sind. In dieser Implementierung ist CCCV immer auf ‘1’, da die Condition Codes ohne Verzögerung ermittelt werden können.
CLK	(*Eingang*) Systemtakt, die steigende Flanke signalisiert den Beginn eines Prozessorzyklus.
$\overline{\text{RESET}}$	(*Eingang*) Durch dieses Signal wird die Befehls-Pipeline zurückgesetzt und die Ausführung aller Befehle, die sich in der Pipeline befanden, abgebrochen.
$\overline{\text{CHOLD}}$	(*Ausgang*) Dieses Signal wird gesetzt, wenn die SFU nicht mit der Ausführung eines Befehls fortfahren kann, weil ein Ergebnis der SFHW fehlt. Daraufhin wird die Befehlspipeline im gesamten System angehalten, bis $\overline{\text{CHOLD}}$ wieder zurückgenommen wird.
$\overline{\text{BHOLD}}$ $\overline{\text{FHOLD}}$ $\overline{\text{MHOLDA}}$ $\overline{\text{MHOLDB}}$	(*Eingänge*) Durch Aktivierung eines dieser Signale wird die Befehls-Pipeline solange angehalten, bis alle Signale wieder zurückgenommen werden.
$\overline{\text{CP}}$	(*Ausgang*) Hiermit wird angezeigt, ob ein Coprozessor im System vorhanden ist. Dieser Anschluß liegt dauerhaft auf ‘low’. Wenn kein Coprozessor vorhanden ist, wird dieses Signal durch einen Pullup-Widerstand auf ‘high’ gehalten.

Abb. 2.18: Externe Anschlüsse der SFU

verläufen läßt sich erkennen, daß sich die IOPs durch bereits vorhandene Opcodes darstellen lassen. Diese Zuordnung ist in Abb. 2.16 zusammengefaßt.

IOP	dargestellt durch
STC-1	STC
STC-2	CNOP
LDC-1	CNOP

Abb. 2.16: Darstellung der IOPs durch vorhandene Opcodes

2.7.4 Pipeline

Der Aufbau der Pipeline aus den drei Stufen ist in Abb. 2.17 dargestellt. Es werden im Gegensatz zur IU nur drei Stufen benötigt, da alle Aufgaben der Fetch-Stufe von der IU ausgeführt werden. Die IU erzeugt also die Adressen für die auszuführenden Befehle und legt sie auf den Adreßbus. Die SFU muß sämtliche Instruktionen vom Datenbus mitlesen und sobald eine Coprozessor-Anweisung kommt, diese ausführen. Die Signale zur Kommunikation mit der IU und dem externen Bus sind in Abb. 2.18 aufgelistet.

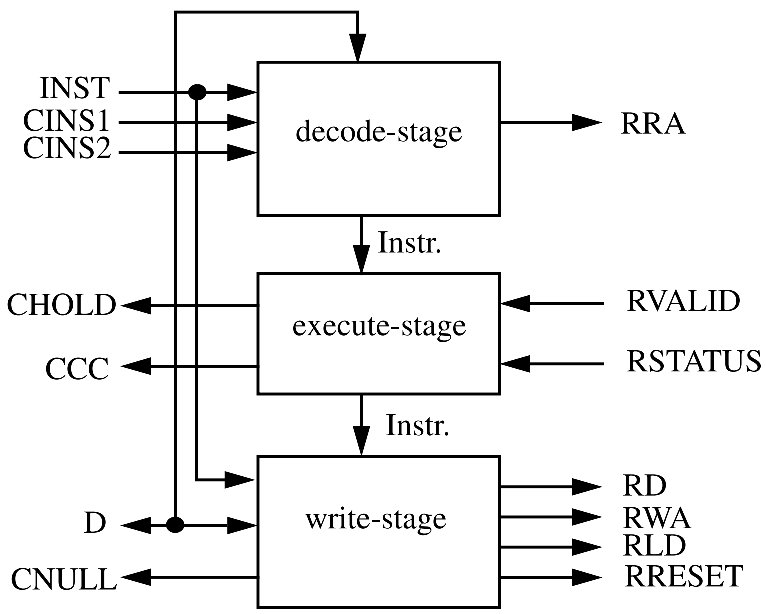

Abb. 2.17: Struktur der Pipeline

	31 30	29 25	24 19	18 14	13	12 5	4 0
LDC	1 1	rd	1 1 0 0 0 0	rs1	i=0	ignored	rs2
					i=1	simm 13	
STC	1 1	rd	1 1 0 1 0 0	rs1	i=0	ignored	rs2
					i=1	simm 13	
CLDC	1 0	rd	1 1 0 1 1 0	simm 19			
CSEL	1 0	1 1 1 1 1	1 1 0 1 1 0	WREN RDEN RESET 15	sfsu		
CMOVE	1 0	rd	1 1 0 1 1 1	ignored	0 0 0 0 0 0 0 0 1		rs2
CRST	1 0	ignored	1 1 0 1 1 1	ignored	0 0 0 0 0 0 0 1 0		rs2
CNOP	1 0	ignored	1 1 0 1 1 1	ignored	0 0 0 0 0 0 0 0 0		ignored

Abb. 2.15: Befehlsformate

Ergebnis bereitsteht. Das Anhalten der Pipeline wirkt sich auch in der Integer Unit und FPU aus und geschieht durch das „$\overline{\text{CHOLD}}$“-Signal. Da dieses Signal bereits während der Execute-Phase bei fallender Taktflanke bereitstehen soll, muß frühzeitig die Gültigkeit der Daten geprüft werden. Das geschieht in der Execute-Stufe, daher muß die Adresse des Ergebnisregisters auch schon während der Decode-Phase auf den SFU-Bus gelegt werden. Die entsprechenden Timing-Diagramme für die betroffenen Befehle (STC und CMOVE) werden in Abb. D.6 und Abb. D.7 gezeigt.

2.7.3 Internal Opcodes

Bei Befehlen, die zur Abarbeitung mehr als einen Zyklus benötigen, muß die Pipeline durch intern erzeugte, zusätzliche Opcodes gefüllt werden („Internal Opcodes“, IOP). Das betrifft die beiden Befehle „LDC“ und „STC“. Diese Befehle benötigen den Datenbus für den Datenaustausch mit dem Speicher, sodaß der Bus in diesem Zyklus nicht für das Laden eines Befehls aus dem Speicher zur Verfügung steht. Das Zeitverhalten ist durch die Integer Unit vorgegeben [Cypr90].

Die Erzeugung der IOPs geschieht in der Decode-Stufe. In den Timing-Diagrammen werden diese Opcodes allgemein durch <Befehl>-1 und <Befehl>-2 gekennzeichnet. An den Signal-

Signal	Beschreibung
A[4:0]	An diesem Eingang wird die Adresse des zu beschreibenden Registers angelegt. Die SFSU-Steuerung reagiert ausschließlich auf Zugriffe an der Adresse 31. Um einen Schreibzugriff auszuführen, muß der Eingang „WR“ gesetzt sein.
WR	Wenn WR=’0’ ist und gleichzeitig die Adresse 31 an A anliegt wird, so wird über den Eingang D eine Steueranweisung an alle SFSUs geschrieben.
STATUS	Dieser Ausgang ist an den „SFU-bus“ angeschlossen. Hier wird der Zustand des „SFHW status“ Eingangs ausgegeben, wenn die SFSU zum Lesen selektiert ist. Sonst befindet sich der Ausgang im hochohmigen Zustand.
SFHW status	Eingang, welcher den Status der SFHW repräsentiert. Wenn die SFSU selektiert ist, wird der Zustand dieses Eingangs über den „STATUS“-Ausgang ausgegeben.
reset	Wenn die SFHW zurückgesetzt werden soll, wird dieser Ausgang aktiviert (auf ‘1’ gesetzt).
ENOP	Dieser Ausgang wird auf ‘1’ gesetzt, wenn die SFSU zum Schreiben aktiviert ist.
ENRES	Dieser Ausgang wird auf ‘1’ gesetzt, wenn die SFSU zum Lesen aktiviert ist.
D	Dateneingang, hier wird die Adresse der anzusprechenden SFSU sowie die auszuführenden Aktionen übermittelt. Das Format ist in Abb. 2.15 angegeben.
CLK	Takteingang, alle Aktionen sind auf positive Taktflanken getriggert.

Abb. 2.14: Anschlüsse der SFSU-Steuerung

2.7.1 Befehlsformate

Die Formate der Befehle „LDC“ und „STC“ (Load Coprocessor bzw. Store Coprocessor) sind durch die SPARC-Definition festgelegt, die übrigen Befehle lassen sich durch die beiden SPARC-Befehle „CPop1“ und „CPop2“ (CPop = Coprocessor Operate) festlegen. Der „CPop1“-Befehl wird dabei ausschließlich zur Definition des „CLDC“-Befehls (Laden einer Konstanten) verwendet, um die größtmögliche Bitlänge der Konstanten zu erreichen.

Der „CNOP“-Befehl wird für die Implementierung der „Internal Opcodes“ (IOPs) benötigt (siehe Abschnitt 2.7.3).

Die Formate der Befehle sind in Abb. 2.15 angegeben.

2.7.2 Zeitverhalten

Um Überschneidungen mit vorhergehenden oder nachfolgenden Coprozessor-Befehlen zu vermeiden, wird für jeden Befehl grundsätzlich das in Abb. 2.7 gezeigte Timing-Schema verwendet. Im Anhang B wird für jeden Befehl das Diagramm mit dem jeweiligen Zeitverhalten gezeigt. Auf die „Internal Opcodes“, wie z. B. STC-1 und STC-2 wird in Abschnitt 2.7.3 eingegangen.

In Abb. D.1 bis Abb. D.5 ist das zeitliche Verhalten aller Befehle für den normalen Fall angegeben. Das heißt insbesondere, daß die Daten bereits in den Ergebnisregistern bereitstehen und gültig sind. Wenn dies nicht der Fall ist, wird die Pipeline solange angehalten, bis das

2.6.3 SFSU-Steuerung

Die Aufgabe dieses Moduls ist die Steuerung der SFSU. Hier wird die Adresse der SFSU dekodiert und die Reset- und Enable-Signale für die Register erzeugt. Eine SFSU wird über die Registeradresse 31 gesteuert. Die Nummer der SFSU wird dafür zusammen mit drei Steuerbits an diese Adresse geschrieben (siehe Format des „CSEL“-Befehls in Abb. 2.15). Dabei ist die Auswahl der Operanden- oder Ergebnisregister oder ein Reset der SFSU möglich, wobei durch das Setzen mehrerer Bits verschiedene Aktionen zusammen erfolgen können, z. B. die Auswahl aller Register und ein gleichzeitiger Reset. Die Registerauswahl bleibt solange gültig, bis die Register einer andere SFSU selektiert werden.

Die SFSU-Steuerung muß also ständig die Schreibvorgänge auf Adresse 31 mitlesen und ggf. reagieren:

- Wenn diese SFSU angesprochen wird, muß die entsprechende Aktion ausgeführt werden. Das kann die Freigabe der Register sein (über die „ENOP“ bzw. „ENRES“-Ausgänge) oder der Reset der SFSU (durch den „reset“-Ausgang).
- Wenn eine andere SFSU adressiert wird, so muß ggf. die Registerauswahl zurückgenommen werden.

Der Status der SFSU ist die logische UND-Verknüpfung aller Bits des Registermonitors ihrer Ergebnisregister. Er wird auf den Bus gelegt, wenn die Ergebnisregister der SFSU selektiert sind.

Die Anschlüsse der SFSU-Steuerung sind in Abb. 2.13 dargestellt und in Abb. 2.12 erläutert.

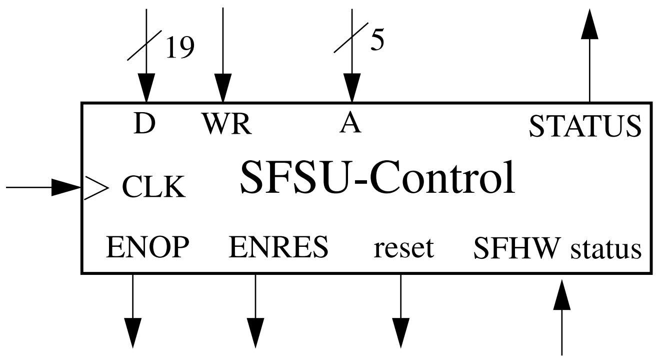

Abb. 2.13: SFSU-Steuerung

2.7 SFU-Steuerung

Die SFU-Steuerung besteht aus einer dreistufigen Befehlspipeline, welche die speziellen SFU-Befehle dekodiert und über den SFU-Bus die SFSUs entsprechend steuert. Für den Entwurf der Pipeline muß das genaue Format und das Zeitverhalten der Befehle bekannt sein. Daher wird in den nächsten Abschnitten zunächst darauf eingegangen.

Alle Aktionen des Registers sind auf die positive Taktflanke getriggert. Die Anschlüsse sind in Abb. 2.11 dargestellt und in Abb. 2.12 erläutert.

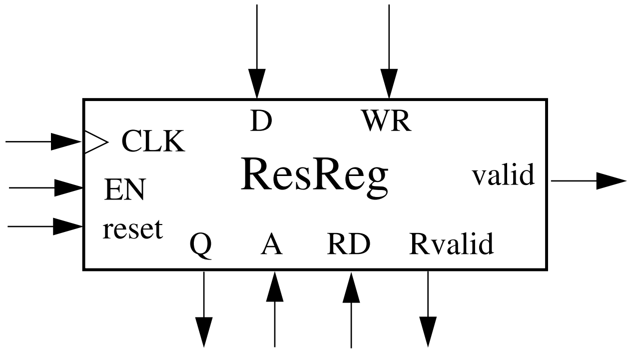

Abb. 2.11: Ergebnisregister

Signal	Beschreibung
D[31:0]	Dateneingang
WR	Eine '1' an diesem Eingang bedeutet, daß die Daten an D in das Register übernommen werden sollen, wenn das „valid bit“ nicht gesetzt ist (='0'). Gleichzeitig wird das „valid bit“ auf '1' gesetzt.
reset	Bei einer '1' an diesem Eingang wird das „valid bit“ auf '0' gesetzt.
EN	Nur, wenn dieser Eingang auf '1' liegt, kann das Register über den Adreßeingang A angesprochen werden.
valid	Dieser Ausgang gibt den Zustand des „valid bit“s wieder. Das „valid bit“ wird durch „reset“ zurückgesetzt und durch das Beschreiben des Registes gesetzt.
Rvalid	Dieser Ausgang befindet sich im hochohmigen Zustand, wenn das Register nicht angesprochen ist. Sobald die richtige Adresse an A anliegt und RD='1' ist, wird der Zustand des „valid bit“ ausgegeben.(Timingdiagramme beachten !)
RD	Eine '1' an diesem Eingang bedeutet, daß die gespeicherten Daten über den Ausgang Q ausgegeben werden, wenn zwei Taktzyklen zuvor die Adresse des Registers an A anlag..
A[4:0]	Der Adreßeingang des Registers. Das Register wird durch eine individuelle Adresse angesprochen, wenn der EN-Eingang auf '1' liegt.
Q[31:0]	Der Datenausgang. Wenn das Register nicht angesprochen wird, ist dieser Ausgang hochohmig.
CLK	Takteingang, alle Aktionen außer dem Anlegen von „Rvalid“ sind auf positive Taktflanken getriggert.

Abb. 2.12: Anschlüsse des Ergebnisregisters

Das Zeitverhalten der Ausgänge „RVALID“ und „Q“ läßt sich dem Diagramm in Abb. 2.7 entnehmen.

Alle Zustandsänderungen finden bei positiver Taktflanke statt. Die Anschlüsse des Operandenregisters sind in Abb. 2.9 dargestellt und in Abb. 2.10 erläutert.

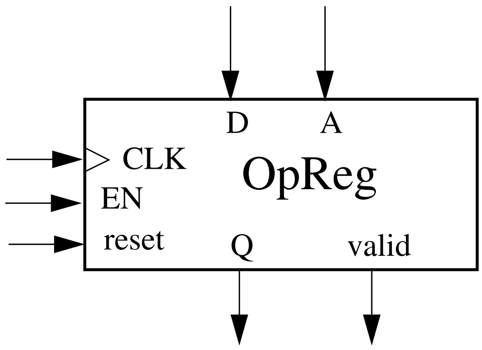

Abb. 2.9: Operandenregister

Signal	Beschreibung
D[31:0]	Dies ist der Dateneingang des Registers. Die Daten werden gespeichert, wenn am Adreßeingang A die richtige Adresse anliegt und der Eingang EN auf ‘1’ liegt.
A[4:0]	Durch diese Adreßleitung wird das Register ausgewählt.
EN	Ein ‘1’-Signal an diesem Eingang gibt das Register zum Schreiben frei.
reset	Rücksetzeingang, eine ‘1’ an diesem Eingang setzt den „valid“-Ausgang zurück.
Q	Ausgangsdaten
valid	Eine ‘1’ an diesem Ausgang signalisiert, daß die Daten am Ausgang „Q“ gültig sind. Dieser Ausgang wird durch den „reset“-Eingang auf ‘0’ gesetzt und durch Beschreiben des Registers auf ‘1’ gesetzt.
CLK	Takteingang, alle Aktionen sind auf positive Taktflanken getriggert.

Abb. 2.10: Anschlüsse des Operandenregisters

2.6.2 Ergebnisregister

Diese Register speichern die bis zu 32 Bit breiten Ergebnisse der SFHW und können die Daten auf den SFU-Bus legen. Jedes Register wird über eine 5 Bit breite Adresse zusammen mit dem Enable-Eingang „EN“ selektiert. Weiterhin kann festgestellt werden, ob die Daten gültig sind, d. h. ob die SFHW das Ergebnis bereits in ein Register geschrieben hat. Dazu enthält jedes Register ein Bit, welches die Gültigkeit der Daten anzeigt. Der Zustand dieses Bits liegt am „valid“ Ausgang an. Das Bit wird durch den Rücksetzeingang „reset“ zurückgesetzt und beim Beschreiben des Registers durch die SFHW gesetzt.

Die Ausgänge „RVALID“ und „Q“ werden mit dem SFU-Bus verbunden und sind Tristate-Ausgänge. Sie bleiben im hochohmigen Zustand, solange das Register nicht durch Adresse und Enable-Eingang ausgewählt wird.

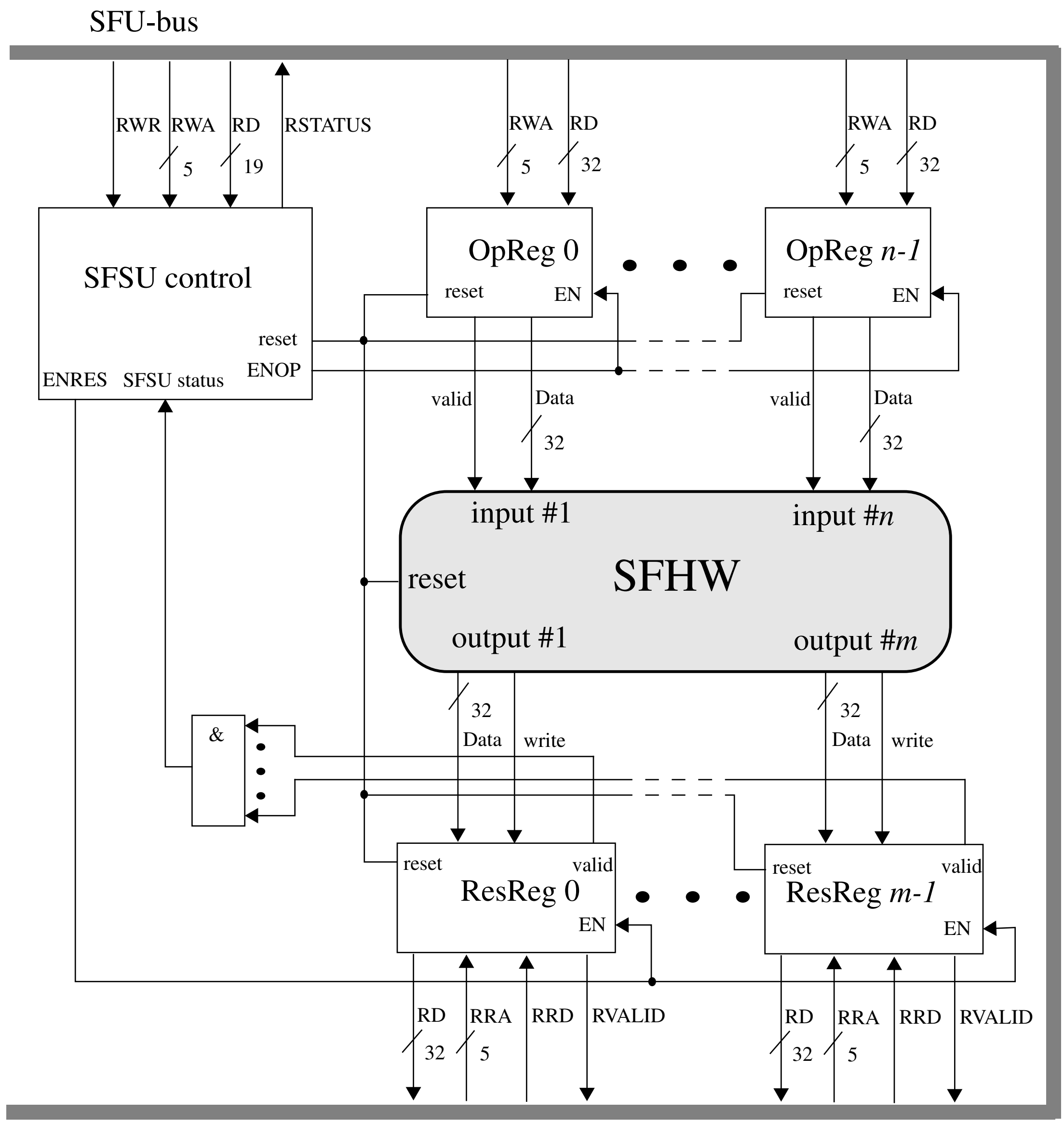

Abb. 2.8: Blockschltbild einer SFSU

zusammen mit dem Enable-Eingang „EN“ zur Selektierung dient. Weiterhin enthält jedes Register ein Bit, welches die Gültigkeit der Daten anzeigt (Registermonitor). Der Zustand dieses Bits ist über den Ausgang „valid“ extern lesbar. Mit einem Rücksetzeingang („reset“) wird dieses Bit zurückgesetzt. Ein Schreibvorgang in das Register setzt das Bit.

gültig sein muß. Dies ist nötig, um ggf. die Befehls-Pipeline anzuhalten, wie in Abschnitt 2.7.6 gezeigt wird.

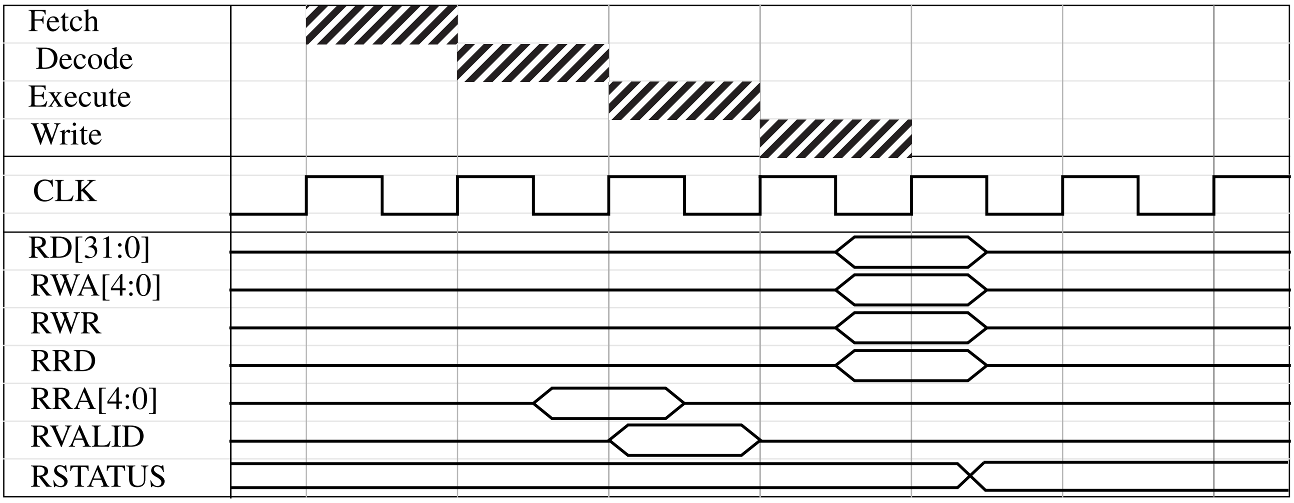

Abb. 2.7: zeitliche Belegung des „SFU-bus“

2.6 Special Function Sub Unit

Eine SFSU hat die Aufgabe, eine einheitliche Schnittstelle einer beliebigen SFHW („Special Function Hardware“) zum SFU-Bus bereitzustellen. Das Blockschaltbild der SFSU ist in Abb. 2.8 dargestellt. Eine SFSU besteht aus einer Steuerung („SFSU-Control“), bis zu n Operandenregister und bis zu m Ergebnisregister, wobei $n \leq 31$ und $m \leq 32$.

Die SFHW besitzt n Eingänge für Operanden (jeweils bis zu 32 Bit breit), m Ausgänge für Ergebnisse (jeweils bis zu 32 Bit breit) sowie einen „Reset“-Eingang zum Initialisieren der SFHW. Zu jedem Operandeneingang gibt es das entsprechende Bit des Registermonitors (siehe Kapitel 2.4), welches anzeigt, ob die Daten im jeweiligen Operandenregister gültig sind. Ebenfalls wird für die Ausgänge der SFHW jeweils eine Strobe-Leitung („write“) benötigt. Wenn dieses Signal gesetzt wird, so übernimmt das zugehörige Ergebnisregister bei der nächsten positiven Taktflanke das Datum am Ausgang der SFHW und ignoriert bis zu einem Reset weitere Änderungen des Datums oder des „write“-Signals. Gleichzeitig bedeutet dies, daß ein gültiges Ergebnis vorhanden ist und das entsprechende Bit des Registermonitors wird im Ergebnisregister gesetzt.

Die drei neben der SFHW wesentlichen Teile der SFSU, die Operanden- und Ergebnisregister sowie die SFSU-Steuerung werden in den folgenden Abschnitten beschrieben.

2.6.1 Operandenregister

Diese Register haben die Aufgabe, die bis zu 32 Bit breiten Operanden für die SFHW zu speichern. Jedes Register besitzt eine 5 Bit breite Adresse, welche im Register dekodiert wird und

Signal	Beschreibung
RD[31:0]	bidirektionale Datenleitung
RWA[4:0]	Auswahl des zu beschreibenden Operandenregisters (wenn RLD='1') oder Auswahl der SFHW (zum Zurücksetzen, d.h. wenn RWR='0' und RRD='0')
RRA[4:0]	Auswahl des zu lesenden Ergebnisregisters (sobald RRD='1' ist, wird der Inhalt auf den Bus gelegt), in jedem Fall liegt das „valid bit“ des adressierten Registers bei der nächsten steigenden Taktflanke auf der RVALID-Leitung und der Status der adressierten SFHW auf der RSTATUS-Leitung.
RWR	Wenn diese Leitung aktiv ist (RWR='1'), wird das Datum auf RD in das durch RWA adressierte Operandenregister der zuvor selektierten SFSU geschrieben. Das „valid bit“ des Registers wird gesetzt.
RRD	Wenn RRD='1' ist, wird: • der Wert des vorher durch RWA adressierten Ergebnisregisters auf RD gelegt. (sonst ist der Registerausgang hochohmig) • der Zustand (valid bit) des durch RWA adressierten Ergebnisregisters im nächsten Taktzyklus auf die RVALID-Leitung gelegt.
RVALID	Hier wird das „valid bit“ des durch RRA adressierten Ergebnisregisters angelegt. Dies muß im Gegensatz zu den anderen Signalen bereits bei der nächsten fallenden Taktflanke gültig sein.
RSTATUS	Der Status der selektierten SFSU. Der Status ergibt sich aus der logischen „UND“-Verknüpfung aller „valid bits“ der zugehörigen Ergebnisregister.

Abb. 2.6: Signale des „SFU-bus“

- Fetch:

  In dieser Stufe werden die Adressen der Befehle generiert und auf den Bus gelegt. Die Befehle stehen einen Takt später zur Verarbeitung in der Decode-Stufe zur Verfügung. Der Befehlszähler befindet sich in der Fetch-Stufe.

- Decode:

  Hier werden die Befehle dekodiert und die Quelloperanden bereitgestellt. Das können entweder Registerinhalte oder ein Registerinhalt und eine Konstante sein.

- Execute:

  Arithmetische oder logische Operationen werden in dieser Stufe ausgeführt. Dazu werden die in der Decode-Stufe ermittelten Operanden benutzt. Das Ergebnis wird an die Write-Stufe weitergegeben.

- Write:

  Die letzte Stufe ist für das Schreiben der Daten aus der Execute-Stufe in das Zielregister oder in den Speicher zuständig. Außerdem können hier Daten vom Datenbus in ein Register transferiert werden (bei „LD“-Befehlen).

Die Signale sind bis auf eine Ausnahme bei positiver Taktflanke gültig. Die Ausnahme bildet das „RVALID“-Signal, welches bereits während der Execute-Phase (bei fallender Taktflanke)

Mit der Annahme $t_{LD} = 2$ ergeben sich hier folgende Werte für $t_{y1} - t_{y2}$:

Fall	$\min(t_{y1} - t_{y2})$	$\max(t_{y1} - t_{y2})$
1	4	4
2	0	4

Für $t_{LD} = 2$ ist wie bei der sequenziellen Struktur ein Zeitgewinn von maximal 4 Taktzyklen möglich.

Es wurde gezeigt, daß durch Einsatz des Registermonitors eine Steigerung des Datendurchsatzes erreicht werden kann, ohne daß eine Verschlechterung eintritt. Die Quantität der Steigerung hängt von der Struktur der SFHW ab. Bei der Realisierung ohne RM wurde in den vorgestellten Beispielen die Zeit zum expliziten Starten der SFHW nicht berücksichtigt, welche bei Implementierung als eigener Befehl mindestens einen weiteren Taktzyklus benötigt. Wird dieser mit einbezogen, so ist die Realisierung ohne RM immer langsamer als mit RM.

Die Benutzung des Registermonitors ermöglicht also in den meisten Fällen eine Steigerung des Durchsatzes und bremst die Verarbeitung in keinem Fall, daher ist der Einsatz sinnvoll.

2.5 SFU-Bus

Um eine einheitliche Schnittstelle zwischen der SFU-Steuerung und den SFSUs zu erhalten, wird der SFU-Bus verwendet. Über diesen SFU-internen Bus tauscht die SFU-Steuerung mit den einzelnen SFSUs sowie die SFSUs untereinander Daten aus. Das Zeitverhalten hängt von der Abarbeitung der Befehle in der Befehls-Pipeline und dem Protokoll des externen Busses ab.

Der SFU-Bus ist ein synchroner 46 Bit breiter Bus mit getrennten Daten- und Adreßleitungen. Der Bus besteht aus 32 bidirektionalen Datenleitungen, zwei Adreßbussen (je 5 Bit breit) für Schreib- bzw. Leseadressen sowie diversen Steuersignalen und Statusleitungen. Die Adressen werden ausschließlich durch die SFU-Steuerung erzeugt und auf den Bus gelegt. Die genaue Funktion der Signale ist in Abb. 2.6 erläutert.

In Abb. 2.7 ist die zeitliche Belegung des SFU-Bus gezeigt. Da die schnellsten SFU-Befehle nur einen Taktzyklus benötigen, darf jede Bus-Leitung pro Befehl nur während eines Zyklus belegt werden. Aufgrund der Verarbeitung der Befehle in einer Pipeline gibt es zeitliche Verschiebungen zwischen verschiedenen Signalen auf dem Bus. In den oberen Zeilen des Diagramms ist die Belegung der Befehlspipeline für einen Befehl angegeben, der einen Taktzyklus benötigt. Bei jeder positiven Taktflanke wird der Befehl um eine Stufe weitergeschoben. Die Stufen haben in der Integer-Unit folgende Aufgaben:

Zweites Beispiel

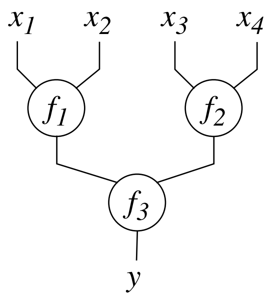

Abb. 2.5: Symmetrische Struktur einer SFHW

- ohne Registermonitor:

$$t_{y1} = 4t_{LD} + \max(t(f_1), t(f_2)) + t(f_3)$$

Hier werden zwei Fälle unterschieden:

$$t(f_1) \geq t(f_2) : \quad t_{y1} = 4t_{LD} + t(f_1) + t(f_3)$$
$$t(f_1) \leq t(f_2) : \quad t_{y1} = 4t_{LD} + t(f_2) + t(f_3)$$

- mit Registermonitor:

$$t_{y2} = 2t_{LD} + \max(t(f_1), 2t_{LD} + t(f_2)) + t(f_3)$$

Auch hier sind zwei Fälle zu unterschieden:

$$t(f_1) \geq 2t_{LD} + t(f_2) : \quad t_{y2} = 2t_{LD} + t(f_1) + t(f_3)$$
$$t(f_1) \leq 2t_{LD} + t(f_2) : \quad t_{y2} = 4t_{LD} + t(f_2) + t(f_3)$$

Es wird nun vorausgesetzt, daß $t(f_1) \geq t(f_2)$ ist. Aufgrund der symmetrischen Struktur läßt sich diese Voraussetzung immer durch entsprechende Anordnung der x_1 bis x_4 erfüllen (Weiter oben wurde festgelegt, daß die Operanden in der Reihenfolge x_1 bis x_4 geladen werden.). Daraus resultiert, daß für die folgende Berechnung nur dieser Fall betrachtet werden muß.

Als Zeitgewinn ergibt sich hier:

Fall		$t_{y1} - t_{y2}$
1	$t(f_1) \geq 2t_{LD} + t(f_2)$	$2t_{LD}$
2	$t(f_1) \leq 2t_{LD} + t(f_2)$	$t(f_1) - t(f_2)$

Bei der symmetrischen Struktur gilt also : $t_{y1} - t_{y2} \geq 0 \Leftrightarrow t_{y2} \leq t_{y1}$. Es ist also hier im Gegensatz zum ersten Beispiel auch der Fall möglich, daß keine Steigerung des Durchsatzes durch den Einsatz des Registermonitors erzielt wird.

Es soll nun der Zeitgewinn berechnet werden, also die Differenz $t_{y1} - t_{y2}$. Dazu müssen 4 Fälle betrachtet werden:

Fall	Berechnung von t_{y2}		$t_{y1} - t_{y2}$
1	$t(f_1) \geq t_{LD}, t(f_2) \geq t_{LD}$:	$t_{y2} = 2t_{LD} + t(f_1) + t(f_2) + t(f_3)$	$2t_{LD}$
2	$t(f_1) \leq t_{LD}, t(f_2) \geq t_{LD}$:	$t_{y2} = 3t_{LD} + t(f_2) + t(f_3)$	$t_{LD} + t(f_1)$
3	$t(f_1) \geq t_{LD}, t(f_2) \leq t_{LD}$:	$t_{y2} = 3t_{LD} + t(f_1) + t(f_3)$	$t_{LD} + t(f_2)$
4	$t(f_1) \leq t_{LD}, t(f_2) \leq t_{LD}$:	$t_{y2} = 4t_{LD} + t(f_3)$	$t(f_1) + t(f_2)$

Offensichtlich gilt $t_{y1} - t_{y2} > 0$ und damit $t_{y2} < t_{y1}$. Also ist eine Realisierung mit Registermonitor für die betrachtete Struktur immer schneller als ohne Registermonitor. Dies läßt sich auch nach der Definition von t_{y1} und t_{y2} beweisen:

Es gilt immer: $\max(a, b) < a + b$ für $a, b \in \mathbb{N}$
Durch Anwendung dieser Ungleichung auf t_{y2} erhält man:

$$\begin{aligned} t_{y2} &= 2t_{LD} + \max(t(f_1), t_{LD}) + \max(t(f_2), t_{LD}) + t(f_3) \\ &< 2t_{LD} + t(f_1) + t_{LD} + t(f_2) + t_{LD} + t(f_3) = 4t_{LD} + t(f_1) + t(f_2) + t(f_3) = t_{y1} \end{aligned}$$

q.e.d.

Um eine realistische Vorstellung zu erhalten, wird $t_{LD} = 2$ angenommen. Dies ist die Zahl der Taktzyklen, die der Befehl „LDC“ benötigt. Dieser lädt einen Wert aus dem Speicher in ein Operandenregister. Aus dieser Annahme lassen sich Werte für $t_{y1} - t_{y2}$ berechnen. Dabei ist jeweils der kleinste und der größte Wert in der folgenden Tabelle aufgelistet:

Fall	$\min(t_{y1} - t_{y2})$	$\max(t_{y1} - t_{y2})$
1	4	4
2	3	4
3	3	4
4	2	4

Es ist für $t_{LD} = 2$ also ein Zeitgewinn von 2 bis 4 Taktzyklen möglich.

t_{LD} ist die Zeit, welche benötigt wird, um einen Operanden x_i in das Operandenregister i zu schreiben. Vereinfachend wird diese Zeit als konstant angesehen, um vergleichbare Ergebnisse zu erhalten. In der Praxis können die Zeiten zum Laden der Operanden je nach Quelle unterschiedlich sein.

$t(f_i)$ ist die Zeit, nach der f_i das Ergebnis bereitstellt. Es gilt: $t(f_i) \geq 0$.

anschließend die SFHW gestartet wird. Bei der Berechnung der Zeit mit Verwendung des Registermonitors werden nur die Operanden geladen (in der Reihenfolge x_1 bis x_4).

Erstes Beispiel

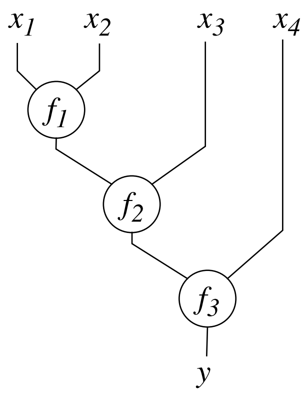

Abb. 2.4: Sequenzielle Struktur einer SFHW

- ohne Registermonitor:

$$t_{y1} = 4t_{LD} + t(f_1) + t(f_2) + t(f_3)$$

- mit Registermonitor:

$$t_{y2} = 2t_{LD} + \max(t(f_1), t_{LD}) + \max(t(f_2), t_{LD}) + t(f_3)$$

Die Komponenten von RM_s sind binäre Variablen, also $V_i, Q_j, R_k \in \{0, 1\}$. Dabei bedeutet der Wert 0, daß der Inhalt des entsprechenden Registers ungültig ist, während der Wert 1 anzeigt, daß das Register gültige Daten enthält.

Bei der Initialisierung der SFSU wird allen Komponenten der Wert 0 zugewiesen.

V_i wird beim Beschreiben des Operandenregisters i auf 1 gesetzt, ensprechend wird mit Q_j beim Beschreiben eines internen Registers und mit R_k beim Beschreiben eines Ergebnisregisters verfahren. Der Vektor RM_{res} kann von der Software ausgewertet werden.

Es soll nun die SFHW betrachtet werden. Eine SFHW besteht aus einer Menge von N Operationen f_i, wobei die Operation f_i genau $L(i)$ Operanden benötigt und $K(i)$ Ergebnisse liefert:

$$SFHW = \{\underline{f}_0(\underline{x}_0), \underline{f}_1(\underline{x}_1), \ldots, \underline{f}_{N-1}(\underline{x}_{N-1})\}$$

$$\underline{f}_i(\underline{x}_i) = (f_{i,0}(\underline{x}_i), f_{i,1}(\underline{x}_i), \ldots, f_{i,K(i)-1}(\underline{x}_i))$$

$$\underline{x}_i = (x_{i,0}, x_{i,1}, \ldots, x_{i,L(i)-1})$$

Eine Operation f_i kann ausgeführt werden, wenn alle Operanden vorhanden sind. Das ist genau dann der Fall, wenn gilt: $\forall j \in [0, L(i)-1] : RM_{x_{i,j}} = 1$. $RM_{x_{i,j}}$ ist das Bit des Registermonitors, welches zu dem Register gehört, das die Variable $x_{i,j}$ speichert.

2.4.2 Bewertung

Um die Steigerung des Datendurchsatzes, welche durch den Registermonitor erreicht werden soll, zu beurteilen, soll nun an zwei repräsentativen Beispielen ein Vergleich zur Implementierung ohne Registermonitor durchgeführt werden.

Es werden zwei unterschiedliche SFHW-Strukturen untersucht. Jede dieser SFHWs besteht aus drei Operationen (f_1 bis f_3) mit je zwei Eingängen, vier Operanden (x_1 bis x_4) sowie einem Ausgang y. Im ersten Beispiel sind die Operationen wie in Abb. 2.4 angeordnet und repräsentieren eine sequenzielle Struktur. Das zweite Beispiel untersucht eine die symmetrische Anordnung der Operationen wie in Abb. 2.5 gezeigt. Jede SFHW-Struktur läßt sich i. a. auf die gezeigten Strukturen reduzieren. Dazu müssen ggf. mehrere Operationen zusammengefaßt und Operanden vertauscht werden.

Zunächst sollen die verwendeten Größen definiert werden:

Es gilt $t \in \mathbb{N}$, da durch das synchrone Design die Betrachtung von diskreten Zeiteinheiten möglich ist (Taktzyklen).

In den nun folgenden Beispielen wird jeweils die Zeit berechnet, die vom Schreiben des ersten Operanden bis zum Erhalt des Ergebnisses y vergeht. Die Berechnung ohne Verwendung des Registermonitors geht davon aus, daß zunächst die Operanden geladen werden und

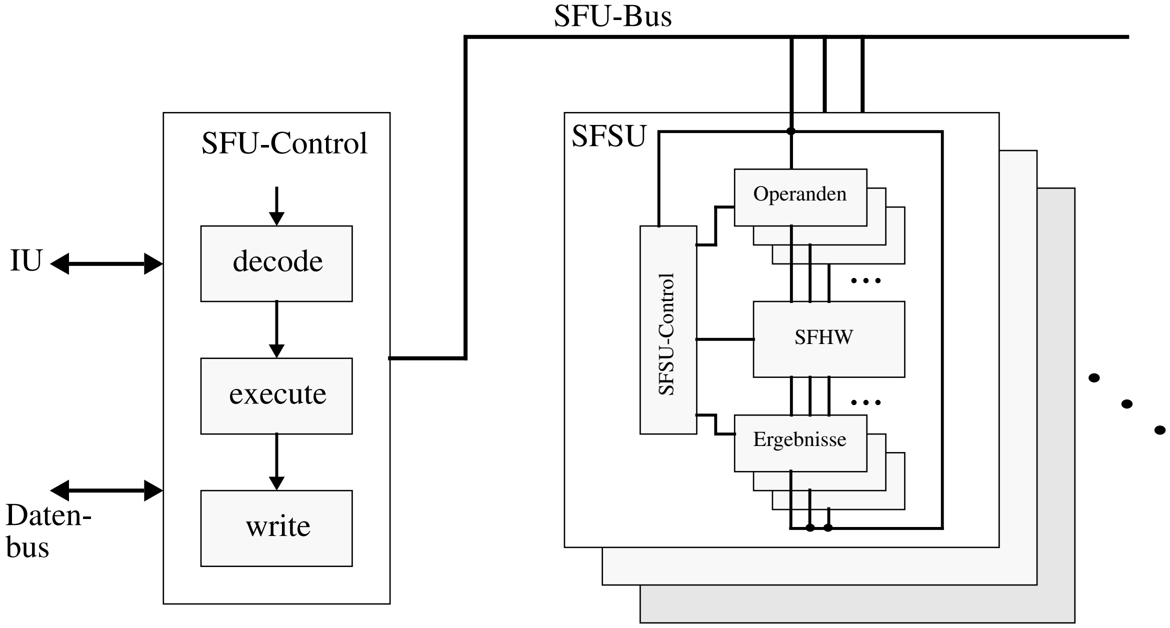

Abb. 2.3: Struktur der SFU

eine Operation vorhanden sind, wird diese selbständig gestartet. Dadurch entfällt eine explizite Steuerung, welche durch die Software vorgenommen werden müßte.

2.4.1 Definition

Zur Definition des Registermonitors RM wird jetzt eine SFSU s betrachtet. Der Registermonitor wird als Vektor definiert:

$RM_s = (RM_{op,s}, RM_{int,s}, RM_{res,s})$, wobei

$RM_{op,s} = (V_0, V_1, \ldots, V_{l(s)-1})$ der RM der Operandenregister,

$RM_{int,s} = (Q_0, Q_1, \ldots, Q_{m(s)-1})$ der RM der internen Regster der SFHW und

$RM_{res,s} = (R_0, R_1, \ldots, R_{n(s)-1})$ der RM der Ergebnisregister ist.

Die Variablen haben folgende Bedeutungen:

s ist die Nummer der betrachteten SFSU; $0 \le s < 2^{16}$

$l(s)$ ist die Anzahl der Operandenregister der SFSU s; $0 < l(s) \le 31$

$m(s)$ ist die Anzahl der internen Register der entsprechenden SFHW ; $m(s) > 0$

$n(s)$ ist die Anzahl der Ergebnisregister der SFSU s; $0 < n(s) \le 32$

```
...
CSEL     WREN+0           ; nicht nötig, wenn vorher mit SFSU #0
                          ; gearbeitet wurde
CSEL     RDEN+RESET+1     ; Auswahl und Zurücksetzen der SFSU #1
CMOVE    cs[0],cr[0]      ; Ergebnis aus Register #0 der SFSU #0
                          ; als Operand #0 der SFSU #1 benutzen
CMOVE    cs[1],cr[1]      ; Ergebnis aus Register #1 der SFSU #0
                          ; als Operand #1 der SFSU #1 benutzen
...
```

Mit der Verwendung eines CMOVE-Befehls werden gegenüber der Verwendung von STC und LDC-Befehlen vier Taktzyklen eingespart. Selbst wenn das Ergebnis zusätzlich im Speicher abgelegt werden soll, weil es vom Programm benötigt wird, sollte der CMOVE-Befehl verwendet werden. Dies spart einen Taktzyklus. Das obige Beispiel müßte dazu durch die folgenden Zeilen ergänzt werden:

```
STC      cr[0], Ergebnis_1    ; Ergebnis aus Register #0 der SFSU #0
                              ; im Speicher ablegen
STC      cr[1], Ergebnis_2    ; das gleiche mit Register #1
```

2.3 SFU-Struktur

Der Aufbau und die Modularisierung der SFU ist in Abb. 2.3 gezeigt. Die Steuerung „SFU-Control“ (siehe Kap. 2.7) besteht aus einer dreistufigen Befehlspipeline, welche parallel zur IU arbeitet. In der Steuerung werden die Befehle mit Unterstützung der IU geholt und dekodiert. Die Befehle, welche die SFU betreffen, werden über den SFU-Bus an die jeweiligen SFSUs weitergeleitet. Der SFU-Bus (siehe Kap. 2.5) besitzt ein eigenes Protokoll, welches das Zeitverhalten der Befehlspipeline berücksichtigt. Der Zugriff auf den Datenbus und die Übermittlung von Statusinformationen an die IU wird ebenfalls von der „SFU-Control“ ausgeführt. Es findet also eine völlige Trennung zwischen dem externen Anschluß an die IU und dem internen SFU-Bus statt.

Am SFU-Bus können max. 2^{16} verschiedenene SFSUs („Special Function Sub Unit“, siehe Kap. 2.6) angeschlossen werden. Jede SFSU kann einzeln adressiert werden und besitzt jeweils bis zu 31 Operanden- und 32 Ergebnisregister. Die Schnittstelle zwischen SFU-Bus und SFSU ist unabhängig von der jeweiligen SFHW („Special Function Hardware“). Die SFHW ist die eigentliche Implementierung der Funktion und besitzt eine individuelle Anzahl von Ein- und Ausgängen, nach welcher sich die Zahl der Register in einer SFSU richtet.

In den folgenden Abschnitten werden der Registermonitor und die einzelnen Komponenten spezifiziert, um auf dieser Grundlage die SFU zu implementieren.

2.4 Registermonitor

Um den Datendurchsatz zu steigern und weitere Steuermechanismen einzusparen, wird ein „Registermonitor“ verwendet. Darin gibt es für jedes Register einer SFSU ein Bit, welches anzeigt, ob die Daten in diesem Register gültig sind. Sobald alle notwendigen Operanden für

2.2.4 Statusabfrage der SFHW

Ein Nachteil der einfachen Methode aus dem letzten Abschnitt wird deutlich, wenn man bedenkt, daß die Verarbeitungszeit einer SFHW nicht konstant zu sein braucht und für verschiedene Eingaben unterschiedlich lange dauern kann. Diese Zeit kann bei der Erstellung der Software nicht vorausgesehen werden. Es können zwei Fälle auftreten:

1. Die SFHW benötigt für eine Operation sehr lange. Dann kann es sein, daß die Software unnötig lange angehalten wird, bis das Ergebnis bereitsteht. In dieser Zeit könnten evtl. andere Dinge erledigt werden.
2. Die SFHW hat eine Operation schnell beendet. Dann könnte sie bereits für neue Berechnungen eingesetzt werden.

Um in einem solchen Fall Programm und SFHW optimal zu koordinieren, kann der Status einer SFSU abgefragt werden.

Mit dem CRST-Befehl wird der Status eines Ergebnisregisters an die IU übermittelt. Der Status kann daraufhin mit einem CBccc-Befehl ausgewertet werden. Dabei ist zu beachten, daß aufgrund der SPARC-Architektur zwischen dem CRST-Befehl und dem CBccc-Befehl mindestens eine nicht-Coprozessor-Anweisung stehen muß (siehe [Cypr90], S. 2-85). Ein solches Programm kann z. B. folgendermaßen aussehen:

```
...
CSEL     WREN+RDEN+RESET+0   ; SFSU #0 auswählen
CMOVE    ...                 ; Berechnung starten
...
CRST     cr[0]               ; testen, ob Ergebnisregister #0 gültige
                             ; Daten enthält
NOP                          ; ein beliebiger nicht-Coprozessor-
                             ; Befehl
CB13     data_ready          ; wenn Ergebnis vorliegt, dann springen
...
```

Der Status einer SFSU kann direkt abgefragt werden. In diesem Fall ist das Bit 1 des Coprozessor-Condition-Code gesetzt, wenn alle Ergebnisregister der zuvor zum Lesen selektierten SFSU gültige Daten enthalten. Die Abfrage erfolgt mit dem bedingten Sprungbefehl „CB23“.

2.2.5 Aufeinanderfolgende SFHW-Operationen

Soll das Ergebnis einer SFHW-Operation als Operand einer anderen Operation dienen, so sollte der CMOVE-Befehl benutzt werden. Da dieser Befehl nur einen Taktzyklus braucht, ist er schneller, als wenn das Ergebnis mit aufeinanderfolgenden STC und LDC-Befehlen über den Umweg des Speichers transferiert wird. Folgendes Programmfragment verwendet z.B. die Ergebnisse aus den Ergebnisregistern 0 und 1 der SFSU Nr. 0 als Operanden für die SFSU Nr. 1 (Operandenregister 0 und 1):

a. Der Befehl ist identisch mit „CLDC“: es wird eine Konstante in das (Steuer-) Register Nr. 31 geschrieben (siehe dazu Abschnitt 2.6.3).

verwendete Notation:

opreg[*n*]: Operandenregister *n*
resreg[*n*]: Ergebnisregister *n*
[*adr*]: Speicherinhalt an der Adresse *adr*

Zum Transfer von Daten innerhalb der SFU kann der Befehl „CMOVE“ verwendet werden. Die Registerangaben beziehen sich dabei auf die zuletzt mit dem Befehl „CSEL“ selektierten SFSUs. Es ist also möglich, Daten zwischen verschiedenen SFSUs auszutauschen, wenn je eine SFSU zum Schreiben und eine andere SFSU zum Lesen ausgewählt wurde.

Konstanten können mit dem Befehl „CLDC“ in ein SFU-Register geschrieben werden. Die Konstante kann eine Länge von maximal 19 Bit besitzen, wobei die oberen 13 Bits des Datenwortes vorzeichenrichtig erweitert werden.

Zur Statusabfrage wird nur ein Befehl benötigt: „CRST“. Dieser Befehl erlaubt es, den Status (Registermonitor) eines Ergebnisregisters abzufragen. Die Auswertung erfolgt mit den „CBccc“-Befehlen aus dem SPARC-Befehlssatz, da das Ergebnis an einer Coprozessor-Statusleitung anliegt. Um den Status einer SFSU auszuwerten, wird kein expliziter Befehl benötigt, da der Status der zum Lesen selektierten SFSU ständig an der zweiten Coprozessor-Statusleitung anliegt und somit unmittelbar durch einen „CBccc“-Befehl abgefragt werden kann.

2.2.3 Einfaches Beispiel

Ein Programm, welches die SFHW benutzt, kann also beispielsweise folgendermaßen aussehen: (es wird hier die SFHW Nr. 0 benutzt, welche drei Operanden- und ein Ergebnisregister besitzt)

```
...
CSEL    WREN+RDEN+RESET+0   ; SFSU #0 auswählen und alle
                              Registerinhalte für ungültig erklären
LDC     Operand_1, cs[0]    ; ersten Operand laden
LDC     Operand_2, cs[1]    ; zweiten Operand laden
LDC     Operand_3, cs[2]]   ; dritten Operand laden

...  [ hier arbeitet Software und SFHW parallel]

STC     cd[0], Ergebnis     ; Ergebnis auslesen und im Speicher ablegen,
                            ; das Programm wartet ggf., bis die SFHW das
...                         ; Ergebnis bereitstellt
```

Dies ist die einfachste Methode, die SFHW zu benutzen. Diese Methode ist in vielen Fällen ausreichend. Wenn z.B. die Verarbeitungszeit der SFHW konstant und vorher bekannt ist, so kann das Programm darauf abgestimmt werden. Wird das Ergebnis der SFHW vom Programm unbedingt an einer bestimmten Stelle benötigt, so gibt es keine Alternative zum Anhalten der Programmausführung.

Mnemonic	Funktion
LDC [adr],opreg[rd]	*Load Coprocessor register* - Ein Wort wird aus dem Speicher in das Operandenregister rd geladen. Der Registerinhalt dieses Registers wird anschließend für gültig erklärt, sodaß die SFU mit einer Operation beginnen kann, welche von diesem Register abhängt.
STC resreg[rs], [adr]	*Store Coprocessor register* - Der Inhalt des Ergebnisregisters rs wird an der Adresse im Speicher abgelegt. Wenn das Ergebnis noch nicht im Register bereitsteht, wird die Ausführung des Programms in der IU und FPU ggf. angehalten, bis die SFHW die Berechnung des Ergebnisses abgeschlossen hat.
CMOVE resreg[rs], opreg[rd]	*Move coprocessor register* - Der Inhalt des Ergebnisregisters rs wird in das Operandenregister rd kopiert. Es werden dabei die SFSUs benutzt, welche zuvor mit dem CSEL-Befehl ausgewählt wurden. Wenn das Ergebnis noch nicht im Register rs bereitsteht, wird die Ausführung des Programms in der IU und FPU ggf. angehalten, bis die SFHW das Ergebnis berechnet hat. Dieser Befehl ist bei Verwendung mehrerer SFSUs sinnvoll, um nicht den Umweg über den Speicher gehen zu müssen, falls das Ergebnis einer Operation als Operand einer weiteren SFHW verwendet werden soll.
CSEL flags+sfsu[a]	*Coprocessor Select* - Die angegebene SFSU wird für den in den Flags angegebenen Operationen ausgewählt. Folgende Flags sind möglich: WREN: *write enable* - Die Operandenregister der angegebenen SFSU werden ausgewählt und können anschließend beschrieben werden. Eine vorherige Auswahl wird aufgehoben. RDEN: *read enable* - Die Ergebnisregister der angegebenen SFSU werden ausgewählt und können gelesen werden. Eine vorherige Auswahl wird aufgehoben. RESET: Die angegebene SFSU wird zurückgesetzt. Dazu werden die Operanden- und Ergebnisregister für ungültig erklärt sowie ein Reset der entsprechenden SFHW durchgeführt. Durch Addition der Flags können mehrere Operationen in einem Befehl kombiniert werden.
CRST resreg[rs]	*Coprocessor Register Status* - Der Status des angegebenen Ergebnisregisters wird als Bit 0 des Coprozessor Condition Code (ccc) ausgegeben und kann mit dem CB13-Befehl ausgewertet werden.
CLDC *imm*, opreg[rd]	*Coprocessor Load Constant* - Die 19-Bit Konstante *imm* wird in das Operandenregister rd geschrieben.

Abb. 2.2: Implementierte Befehle

1. Die absolute Adressierung ist nur in den unteren 4 kByte des Adressraums möglich.

2.2 Softwaresicht

2.2.1 Allgemeines

Aus Sicht der Software besteht die SFU aus bis zu 64 Registern, die beschrieben oder gelesen werden können. Folgende Punkte sollten jedoch beachtet werden:

- Die Operandenregister können *nur* beschrieben werden
- Die Ergebnisregister können *nur* gelesen werden
- Durch das Beschreiben der Operandenregister wird die SFHW gestartet
- Wenn ein ungültiges Ergebnisregister gelesen werden soll, wird gewartet, bis es einen gültigen Wert enthält

Da die Aktionen der SFHW nicht explizit gestartet werden müssen, braucht das Programm lediglich die Operanden in die Operandenregister zu schreiben und das Ergebnis auszulesen. Allerdings muß vor dem Beschreiben der Operandenregister die SFSU ausgewählt und deren Registerinhalte für ungültig erklärt werden. Dies geschieht mit einem speziellen Befehl, dem CSEL-Befehl.

2.2.2 Befehlssatz

Die von der SFU zur Verfügung gestellten Befehle lassen sich in die Bereiche Datenübertragung, Steuerung und Statusabfrage unterteilen (siehe Abb. 1.2). Die größte Gruppe bilden dabei die Befehle zum Datentransfer. Sämtliche Befehle sind in Abb. 2.2 aufgelistet und erläutert.

Der einzige Befehl zur Steuerung der SFU ist der Befehl „CSEL“. Mit diesem Befehl wird eine SFSU selektiert, auf welche sich die darauffolgenden Befehle beziehen. Dabei kann gleichzeitig die SFSU zurückgesetzt werden. Es ist möglich, in die Operandenregister einer SFSU zu schreiben und weiterhin die Ergebnisregister einer anderen SFSU zu lesen. Dadurch können Daten schnell zwischen zwei SFSUs übertragen werden.

Für den Transfer eines Datenwortes aus dem Speicher in ein Operandenregister der ausgewählten SFSU wird der Befehl „LDC“ benutzt. Durch das Beschreiben des Registers werden die Daten für gültig erklärt, sodaß eine Berechnung evtl. sofort gestartet werden kann (siehe Registermonitor, Abschnitt 2.4).

Um den Inhalt eines Ergebnisregisters in den Speicher zu schreiben, existiert der Befehl „STC“. Wird dieser Befehl ausgeführt, wenn die Daten noch nicht zur Verfügung stehen (d.h. die SFHW ihre Operation noch nicht beendet hat), so wird die weitere Ausführung des Programms gestoppt, bis die Daten bereitstehen. Es stehen die üblichen Adressierungsarten (Register-indirekt oder absolut[1]) zur Verfügung, da die Befehlsformate durch die SPARC-Definition vorgegeben sind (siehe [Cypr90]).

2. Entwurf

2.1 Zielarchitektur

Es soll nun erläutert werden, wie die zu entwerfende SFU in die Zielarchitektur eingebunden wird. In Abb. 2.1 ist ein Blockschaltbild mit allen wichtigen Komponenten des Prozessorsy-

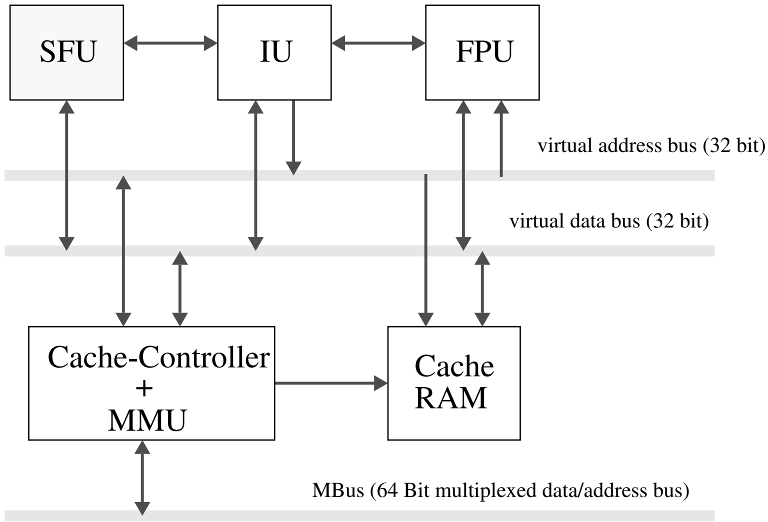

Abb. 2.1: Architektur des zu verwendenden Prozessorsystems [Cypr90]

stems gezeigt. Bei der Interger-Unit (IU) handelt es sich dabei um den SPARC-Prozessor CY-7C601 der Firma „Cypress Semiconductor Corp.“. Die Fließkommaeinheit (FPU), die Memory-Management-Unit (MMU) und der Cache-Speicher werden vom gleichen Hersteller angeboten. Die SPARC-Architektur erlaubt den Anschluß eines zweiten, benutzerdefinierten Coprozessors, der bereits mit eigenen Befehlsmasken im Befehlssatz unterstützt wird. Die SFU wird daher neben der FPU als zweiter Coprozessor angeschlossen.

In der Integer-Unit durchläuft jeder Befehl eine vierstufige Befehlspipeline. Dort werden auch die FPU- und Coprozessor-Befehle dekodiert und für alle Speicherzugriffe von IU, FPU und SFU die Adressen berechnet. Bei der Adreßberechnung werden interne Register der IU benutzt, daher ist eine Berechnung in der FPU oder SFU nicht möglich und auch nicht notwendig. Aus diesem Grunde benötigt die SFU keinen Zugriff auf den Adreßbus. Weiterhin wird dadurch der Speicherzugriff über die MMU auf den „MBus“ und den Cache ohne zusätzlichen Aufwand mitbenutzt.

die SFHW die Gültigkeit der Daten und kann bei geeigneter Reihenfolge des Schreibens bereits mit der Verabeitung anfangen, bevor sämtliche Operanden geschrieben wurden. Außerdem entfällt ein expliziter Startbefehl für die SFSU.

1.3 Gewählte Lösung

Wie bereits erwähnt, wird die SFU als Coprozessor an die Integer-Unit angeschlossen. Durch die Vorgaben der SPARC-Definition ergeben sich einige Besonderheiten, die beachtet werden müssen.

Eine Übersicht über den Aufbau des Coprozessors (SFU) gibt das Blockschaltbild in Abb. 1.7.

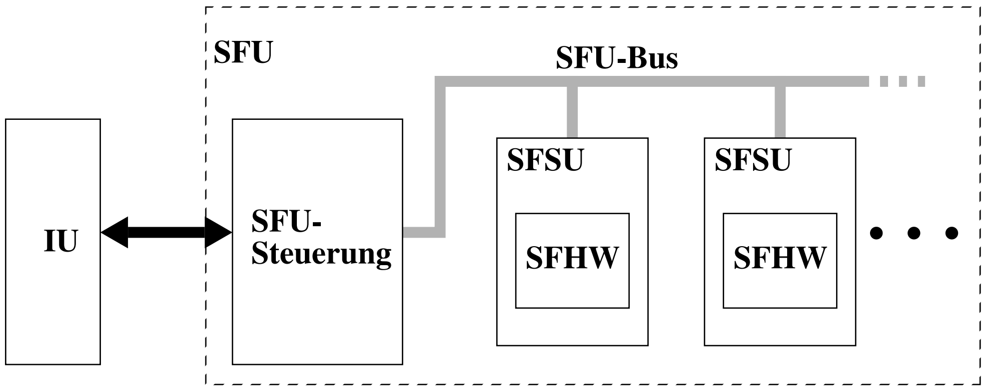

Abb. 1.7: Blockschaltbild der SFU

Es ist möglich, mehrere verschiedene Hardware-Funktionen im Coprozessor unterzubringen. Jede dieser Funktionen wird durch eine „Special Function Sub Unit“ (SFSU) repräsentiert. Diese besteht wiederum aus Operanden- und Ergebnisregistern, einer Steuerung sowie der eigentlichen in Hardware implementierten Funktion („Special Function Hardware“, SFHW). Die SFSUs werden über einen internen Bus mit der SFU-Steuereinheit verbunden. Dadurch ist es möglich, eine nahezu beliebige Anzahl von SFSUs in einem Coprozessor zu implementieren. Die SFU-Steuereinheit stellt die Schnittstelle zwischen der durch die Integer-Unit festgelegten externen Anschlüsse und dem internen SFU-Bus dar.

Durch den modularen Aufbau ist die Implementierung weitgehend unabhängig von der SFHW und erweiterbar. Durch Austauschen der SFU-Steuerung lassen sich die SFSUs auch an andere Architekturen anbinden.

Es können 32 Coprozessor-Register direkt angesprochen werden. Diese Anzahl ist durch das festliegende Format der Coprozessor-Befehle im SPARC-Befehlssatz begrenzt. Daher wurde eine Begrenzung auf maximal 31 Operanden und maximal 32 Ergebnisse vorgenommen (siehe auch Kap. 5). Das übrige Register wird zur Auswahl der SFSU benutzt. Dabei ist eine maximale Anzahl von 2^{16} SFSUs möglich, was in der Praxis völlig ausreicht.

Eine deutliche Steigerung des Durchsatzes kann mit dem „Registermonitor“ (Kap. 2.4) erreicht werden. Bei diesem Mechanismus gibt es zu jedem Operandenregister ein Kontrollbit, welches anzeigt, ob die Daten im Register gültig sind. Alle diese Kontrollbits werden vor Benutzung der SFSU gemeinsam zurückgesetzt und durch das Schreiben eines Operanden in das jeweilige Register gesetzt. Werden nun die Operanden in die Register geschrieben, so erkennt

1. Fall ($\tau \geq t_2$): Der Interrupt kommt, bevor die Daten benötigt werden. Die SW muß trotzdem zum Zeitpunkt $t = \tau$ abfragen, ob die Daten bereits vorliegen, d. h. ob ein Interrupt auftrat. Der Aufwand ist also wie bei „Programmed I/O“ plus dem Interrupt-Overhead.
2. Fall($\tau < t_2$): ähnlich wie beim 1. Fall kommt auch hier der Zeitbedarf für die Interrupt-Behandlung dazu (Abb. 1.3).

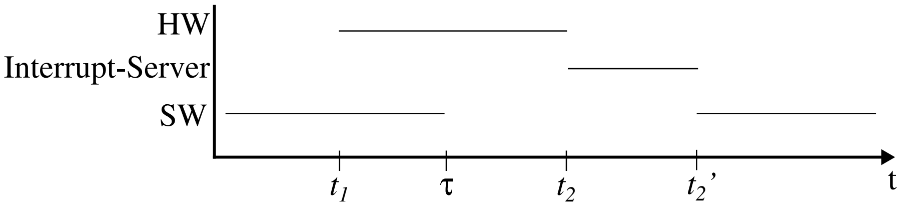

Abb. 1.6: zusätzlicher Zeitbedarf t_2'-t_2 bei Verwendung von Interrupts

Zusätzlich ist ein größerer Hardware-Aufwand zur Interrupt-Generierung notwendig als bei „Programmed I/O“.

DMA

Bei DMA (Direct Memory Access) greift die SFU selbständig direkt auf den Speicher zu, um Daten zu holen und Ergebnisse zurückzuschreiben. Daher ist diese Methode besonders gut für große Datenblöcke geeignet. DMA benötigt zusätzlich eine Initialisierung und Statusabfrage (evtl. über Interrupts). Da die SFU über DMA die Daten direkt aus dem Hauptspeicher liest und dorthin zurückschreibt, wird der Cache-Speicher nicht mitbenutzt. Weiterhin wird i. a. mit den benötigten Daten gerade gearbeitet, daher muß zunächst der Cache zurückgeschrieben und nach der Berechnung wieder aktualisiert werden. Es läßt sich also keine Steigerung der Geschwindigkeit erreichen, die Abarbeitung der Software wird dadurch verlangsamt. Weiterhin ist ein wesentlich höherer Hardwareaufwand als bei den anderen beiden Verfahren nötig.

1.2.3 Ergebnis

Da die Gesamt-Performance erster Gesichtspunkt ist,, der Overhead für Steuerung und Statusabfrage klein und der HW-Aufwand (Chipfläche) auch möglichst gering sein soll, wird die Implementierung als Coprozessor gewählt.

Als Methode zur Datenübertragung wird - wie bei der FPU (Floating Point Unit) - „Programmed I/O“ benutzt. Diese Methode konnte in der vorliegenden Arbeit jedoch aufgrund der Besonderheiten der Coprozessor-Einbindung in das SPARC-System bezüglich der Geschwindigkeit noch verbessert werden.

weils internen Registern der IU und SFU wird mehr Zeit benötigt, weil in diesem Fall das Datum temporär im Speicher abgelegt werden muß. Die Häufigkeit, mit der dieser Fall auftritt, kann bei Erstellung der Software minimiert werden. Für die Kommunikation bei „Memory Mapped I/O" wird bei den meisten Befehlsfolgen zusätzlich ein temporäres Register der IU benötigt, welches evtl. vorher gesichert werden muß.

Wird weiterhin den Aufwand an HW-Komponenten betrachtet, so stellt man keinen großen Unterschied fest. Bei „Memory Mapped I/O" wird eine Adreßdekodierung sowie eine Steuerung des Busprotokolls benötigt, im Coprozessor eine Befehlspipeline. Diese Pipeline ist nicht sehr aufwendig, da sie von der IU gesteuert wird. Es ist keine Adreßgenerierung notwendig, da diese auch von der IU durchgeführt wird. Bei beiden Lösungen ist etwa die gleiche Anzahl an Registern notwendig (hauptsächlich, um Operanden zu speichern).

1.2.2 Bewertung der Übertragungsverfahren

Bei den Überlegungen zur Effizienz der verschiedenen I/O-Verfahren sollten folgende Dinge beachtet werden:

- Erst wenn die Daten von der SFU benötigt werden, wird geprüft, ob sie bereits zur Verfügung stehen.
- Die Ergebnisse der SFU werden i. a. unbedingt zur Fortsetzung des Programms benötigt.
- Ein kritischer Zeitpunkt ist das Ende der HW-Phase (t_2). Hier soll der Zeitverlust gering gehalten werden.

Bei der Bewertung wird auf die auf Seite 6 beschriebenen Fälle eingegangen.

Programmed I/O

Bei dieser Methode wird die gesamte Datenübertragung und Kommunikation mit der SFU durch ein Programm gesteuert (Polling).

1. Fall ($\tau \geq t_2$): Hier gibt es keine Verzögerung, da nur eine Statusabfrage nötig ist.
2. Fall ($\tau < t_2$): Die HW wird später fertig als die SW das Ergebnis benötigt, also wird die SW durch „busy wait" darauf warten. Da die SW während dieser Zeit keine anderen Aufgaben hat, stört das „busy waiting" nicht.

Interrupt

Sobald die HW mit ihrer Berechnung fertig ist, wird ein Interrupt ausgelöst, um der SW dies mitzuteilen. Ein Interrupt-Behandlungsprogramm kann nun die Ergebnisse von der SFU holen.

2. Fall: $\tau < t_2$, d. h. SW muß warten

   Die SW benötigt das Ergebnis der HW, bevor diese fertig ist. Da die SW auf die Daten angewiesen ist, muß bis t_2 gewartet werden. (Abb. 1.3)

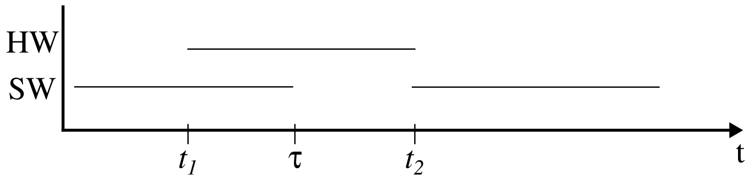

Abb. 1.4: SW muß auf Ergebnis der HW warten

Analog dazu läßt sich die FPU betrachten, bei der für die Ausführung von arithmetischen Operationen die beiden dargestellten Fälle auftreten können.

1.2.1 Bewertung der Schnittstellen

In der in Abb. 1.5 gezeigten Tabelle ist das Zeitverhalten der unterschiedlichen Anschlußmög-

Funktion	**Datenrichtung**	**MM-I/O Befehle**	**MM-I/O Taktzyklen**	**Coproz. Befehle**	**Coproz. Taktzyklen**
Datenübertragung	IU → SFU	ST	3	ST, LDC	5
	SFU → IU	LD	2	STC, LD	5
	Speicher → SFU	LD, ST	5	LDC	2
	SFU → Speicher	LD, ST	5	STC	3
	SFU → SFU	LD, ST	5	CPop	1
Steuerung		ADD, ST[a]	4	CPop	1
Statusabfrage		LD, CMP	3	CPop (-)[b]	1 (0)

Abb. 1.5: Zeitbedarf bei „Memory Mapped I/O“ und bei Anschluß als Coprozessor

a. Schreiben einer Konstanten in ein Steuerregister

b. Ergebnis wird über Statusleitungen an die IU übermittelt; in einigen Fällen kann der CPop-Befehl entfallen, es wird dann kein Taktzyklus benötigt

lichkeiten für verschiedene Aktionen aufgelistet. Dabei wird die Datenübertragung, die Steuerung und Statusabfrage betrachtet. Bei der Anbindung als Coprozessor ist der Zeitaufwand besonders für Steuerung und Statusabfrage kleiner als bei „Memory Mapped I/O“, was daran liegt, daß der Coprozessor stärker in die Architektur der Integer Unit[1] (IU) integriert ist und durch spezielle Befehle unterstützt wird. Lediglich bei der Datenübertragung zwischen den je-

1. so wird in der SPARC-Architektur die zentrale Steuereinheit (CPU) benannt.

(Abb. 1.1). Es lassen sich also die von I/O-Geräten bekannten Methoden anwenden. Zunächst

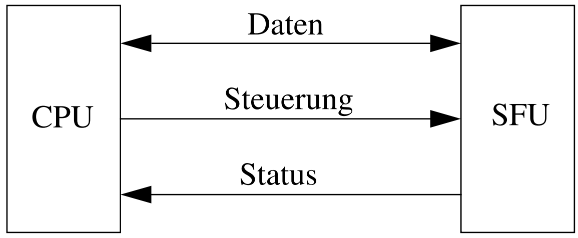

Abb. 1.1: Daten-/Steuerfluß zwischen CPU und SFU

wird der Anschluß der Schnittstellen betrachtet. Dabei sind beim verwendeten SPARC-Prozessor lediglich „Memory Mapped I/O" sowie der Anschluß als Coprozessor möglich [Spar92]. Weiterhin gibt es verschiedenen Verfahren zur Datenübertragung und Steuerung (Abb. 1.2). Eine davon ist „Programmed I/O", dabei wird die gesamte Steuerung und Statusabfrage programmgesteuert durchgeführt („Polling"). Bei der Verwendung von Interrupts erfolgen Statusmeldungen der SFU über Signalleitungen. Das DMA-Verfahren erlaubt die Steuerung der gesamten Datenübertragung durch die SFU.

Schnittstellen	**Datenübertragung/Steuerung**
Memory Mapped I/O	programmed I/O
Coprozessor	Interrupt
	DMA

Abb. 1.2: verschiedene Anschlußmöglichkeiten

Zur Beurteilung der verschiedenen I/O-Verfahren wird nun die Problemstellung formalisiert. Dabei werden die Resourcen Hardware (HW) und Software (SW) betrachtet, welche parallel arbeiten können. Der zeitliche Ablauf der Abarbeitung wird als Gantt-Diagramm dargestellt [Bell92]. t_1 bezeichnet den Zeitpunkt, an dem die HW durch das SW-Programm gestartet wird und mit der Berechnung beginnt. Zum Zeitpunkt t_2 terminiert die HW-Operation. Die SW benötigt die Ergebnisse der HW jeweils zum Zeitpunkt τ, wobei gilt: $\tau > t_1$. Aufgrund möglicher Datenabhängigkeiten müssen zwei Fälle unterschieden werden:

1. Fall: $\tau \geq t_2$, d. h. HW-Operation terminiert rechtzeitig

   Hier muß die SW nicht auf die Ergebnisse der HW warten, sondern kann sofort mit den Ergebnissen weiterarbeiten. (Abb. 1.3)

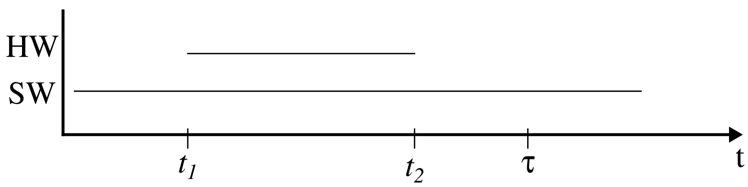

Abb. 1.3: HW terminiert, bevor das Ergebnis benötigt wird

Beim HW/SW-Codesign gibt es verschiedene Ansätze, so z. B.:

1. Hardware-orientiert

   Das System wird zunächst komplett in Hardware entworfen. Anschließend werden Module extrahiert, für welche eine Realisierung in Software vorgenommen wird [Gupt92].

2. Software-orientiert

   Hier wird von einer reinen Software-Lösung ausgegangen und aus dem (bestehenden) Programm Module ausgewählt und in Hardware realisiert werden [Erns92, Hard94].

In der Arbeitsgruppe von Prof. Camposano wird ein Software-orientierter Ansatz verfolgt. Ausgehend von einem Programm werden nach bestimmten Kriterien (u.a. Größe, Zeitverhalten, Interface-Kosten) Programmteile gesucht, welche effizienter in Hardware als in Software implementiert werden können. Dabei werden reine Software-Funktionen betrachtet, jedoch keine „typischen" Hardware-Komponenten wie z. B. I/O.

Nach der Partitionierung (Aufteilung in Hard- und Software) erhält man getrennte HW- und SW-Partitionen. Für den Datentransfer wird ein effizientes Interface benötigt, um den Vorteil einer effizienten Implementierung in Hardware nicht durch hohe Interface-Kosten zunichte zu machen. Die Kosten für das Interface wurden zwar bisher (z. B. bei der Partitionierung) bereits berücksichtigt, allerdings war nur eine grobe Schätzung möglich, da sie sehr vom verwendeten Interface-Design abhängen [Hard94].

Um eine realistische Kostenabschätzung zu ermöglichen, soll nun ein Interface für eine konkrete Zielarchitektur entworfen und implementiert werden. Weiterhin ist diese Implementierung ein wichtiger Schritt, ein durch HW/SW-Codesign entworfenes System in der Praxis zu realisieren. Der Entwurf sowie die Implementierung eines solchen Interfaces ist das Thema dieser Arbeit.

Folgende wichtige Vorgaben sollen dabei u.a. berücksichtigt werden:

- Zielarchitektur: SPARC (Cypress CY 7C-601)
- möglichst hoher Datendurchsatz
- Bedarf an Chip-Fläche soll gering sein
- flexibel (weitgehend unabhängig von der HW-Funktion oder der Anzahl ihrer Parameter)
- Steuerung erfolgt durch die SW

1.2 Verschiedene Lösungsansätze

Es sollen jetzt verschiedene Möglichkeiten der Realisierung beschrieben und bewertet werden. Dazu wird die „Special Function Unit" (SFU), das sind die in Hardware implementierten Funktionen mitsamt Interface, als I/O-Gerät betrachtet [Mano93]. Die Kommunikation mit der CPU läßt sich in die Bereiche Datenübertragung, Steuerung und Statusmeldungen unterteilen

1. Einleitung

1.1 Hardware/Software-Codesign

Unter Hardware/Software-Codesign (kurz: HW/SW-Codesign) wird die Aufteilung eines digitalen Systems in Hard- und Software verstanden. Dabei geschieht die Aufteilung während des Designprozesses. Im Gegensatz dazu wurde ein System bisher vor dem Entwurf in Komponenten aufgeteilt, die in Hard- oder in Software realisiert und anschließend getrennt voneinander entwickelt wurden.

Die Motivation für die Verwendung von HW/SW-Codesign liegt darin, daß sowohl die Realisierung als Software-Lösung als auch eine Hardware-Lösung jeweils andere Vor- und Nachteile bringen, welche je nach Modul gegeneinander abgeschätzt werden müssen. Die Realisierung eines Entwurfs als reine Software-Lösung ist zwar in vielen Fällen kostengünstig, aber langsam; eine reine Hardware-Realisierung ist hingegen schneller, aber auch teuerer.

Die Entwicklung von Software (für bestehende Prozessorsysteme) ist einfacher und kostengünstiger, weil Programme einfach geändert werden können, was sich in kürzeren „Turn-Around-Zeiten“ bei der Entwicklung niederschlägt. Weiterhin ist der Bedarf an Chipfläche geringer als bei einer reinen Hardware-Realisierung, da heutige Halbleiterspeicher eine sehr hohe Speicherdichte besitzen [Hard93]. Durch Einsatz von Standardbauteilen für Prozessor und Speicher werden keine oder weniger ASICs[1] benötigt, was ebenfalls die Kosten senkt. Allerdings ist die Geschwindigkeit für einige Anwendungen selbst beim Einsatz von schnellen Prozessoren (welche wiederum teurer sind) zu gering, da Berechnungen aus elementaren Operationen zusammengesetzt sind, welche i.a. sequentiell ausgeführt werden.

Der Vorteil beim Einsatz von Hardware ist die hohe Geschwindigkeit, welche bei der Realisierung komplexer Funktionen nur durch die Gatterlaufzeit begrenzt wird. Bei synchronen Schaltungen kann der Durchsatz durch Anwendung von Pipeline- oder anderen Techniken erhöht werden. Vergleicht man die Ausführungszeiten von gleichen Funktionen bei Implementierung in Hard- und Software, so ergibt sich besonders bei Steuerungsanweisungen (Verzweigungen, Iterationen) eine deutliche Steigerung der Geschwindigkeit bei der Hardware-Realisierung [Hard93]. Diese Vorteile werden allerdings durch einen größeren Bedarf an Chipfläche und höheren Entwicklungsaufwand erkauft, was höhere Kosten als bei der Software-Entwicklung verursacht. Hier kann der Einsatz von Synthesewerkzeugen oder Codesign-Systemen Vorteile bringen.

Da sich die Hardware-Entwicklung durch den Einsatz von Hochsprachen wie z.B. Verilog oder VHDL[2] immer weiter dem Entwicklungsprozeß von Software annähert, liegt es nahe, die Aufteilung auf Hardware und Software erst während des Entwurfs vorzunehmen.

---

1. **A**pplication **S**pecific **I**ntegrated **C**ircuit
2. **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

Inhaltsverzeichnis

Kurzfassung

In dieser Arbeit wird ein Coprozessor als effiziente Schnittstelle zu Hardware-Komponenten vorgestellt, welche durch HW/SW-Codesign entstehen. Der auf einer SPARC-Architektur basierende Coprozessor erlaubt durch modularen Aufbau den Einsatz mehrerer unabhängiger Hardware-Komponenten. Es werden Methoden vorgestellt, um einen hohen Datendurchsatz zu erhalten. Darauf aufbauend wird der Coprozessor in der Sprache „VHDL“ implementiert sowie eine Simulation und eine Synthese des Entwurfs durchgeführt.

Abstract

This paper describes a coprocessor which serves as an efficient interface to hardware components that are derived from HW/SW-codesign. The coprocessor is based on a SPARC architecture and allows the handling of multiple hardware components by a modular structure. Methods are introduced to obtain high throughput of data. Based on this, the coprocessor is implemented in the language „VHDL“. Finally, a simulation and synthesis of the design is performed.

Ich versichere, daß ich diese Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Paderborn, den 27.7.1994

UNIVERSITÄT-GH-PADERBORN

Entwicklung einer Hardware-Schnittstelle zur effizienten Anbindung von Coprozessoren

Andreas Günther

Juli 1994

Studienarbeit

Betreuer:

Prof. Dr.-Ing. Erik Maehle
Dipl.-Ing. Thomas Kosch
Dipl.-Inf. Wolfram Hardt

Bearbeitungszeitraum:
27.4.1994 - 27.7.1994

Fachgebiet Datentechnik (Prof. Dr.-Ing. Erik Maehle)

FB 14 (Elektrotechnik) • Pohlweg 47-49 • D-33095 Paderborn • Tel.: 05251/602210