Darstellung eines 1-Bit seriellen Addierwerks mit VHDL

Tom Nagengast, Mathias Herbst
IAV 07/09
Rudolf-Diesel-Fachschule
für Techniker

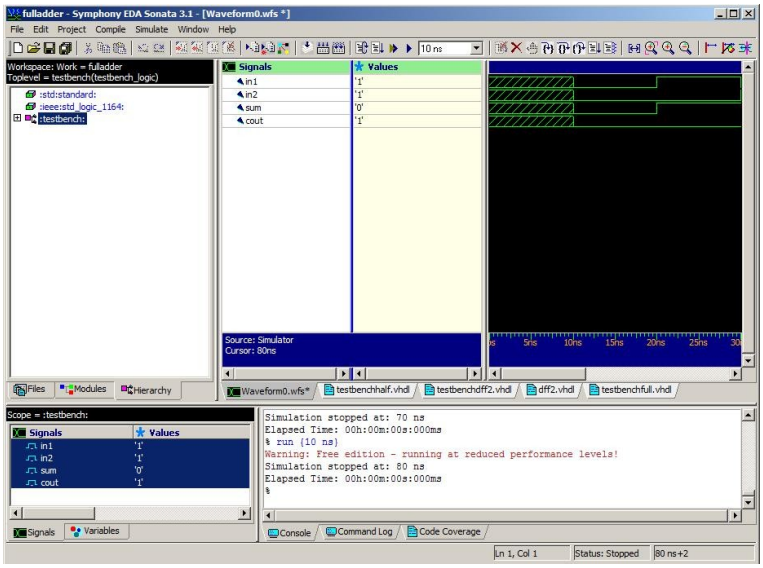
Inhalt:

1. Verwendete Tools
   1.1 Simili 3.1
   1.2 Tina
2. Vorgehensweise
   2.1 Halbaddierer
   2.2 Volladdierer
   2.3 Signallaufzeiten
   2.4 Volladdierer mit Latch
   2.5 Serielles Addierwerk mit Ausgaberegister

1. Verwendete Tools

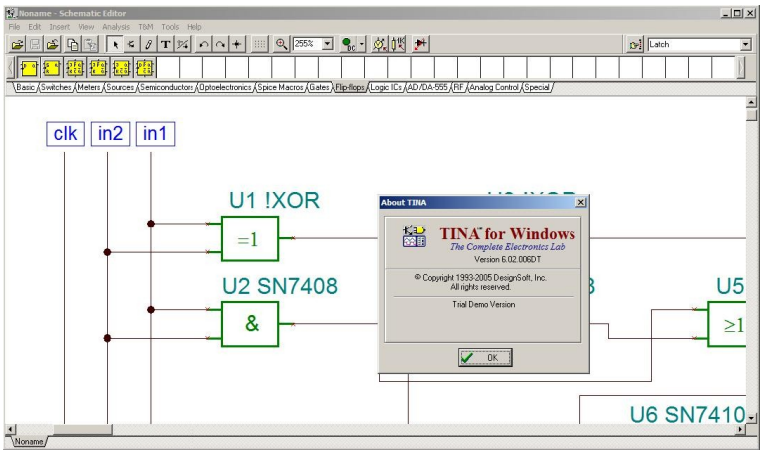
1.1 Simili 3.1

Zur Durchführung unserer Projektarbeit verwenden wir die Software Simili der Firma Symphony EDA in der Version 3.1.
Wir haben diese Software ausgewählt weil sie in einer freien Version mit leichten Einschränkungen verfügbar ist und Simulationen der VHDL Sprache leicht ermöglicht.



1.2 Tina

Die Abbildung der Schaltungen in diesem Dokument erfolgten mit der Trial-Demo der Software Tina, die auch zur Schaltungs-simulation genutzt werden kann.



2.Vorgehensweise

Die folgenden Punkte zeigen unser schrittweises Vorgehen um das Addierwerk mit der Hardware Beschreibungssprache VHDL darzustellen.
Orientiert haben wir uns hierbei an Unterrichtsmaterial aus dem Fach Datenverarbeitungstechnik und zum anderen an, als PDF erhältlichen, Anleitungen zu VHDL.

2.1 Halbaddierer

Da sich eine ALU aus Halbaddierern zusammensetzt ist der erste Schritt einen Halbaddierer mit VHDL darzustellen.

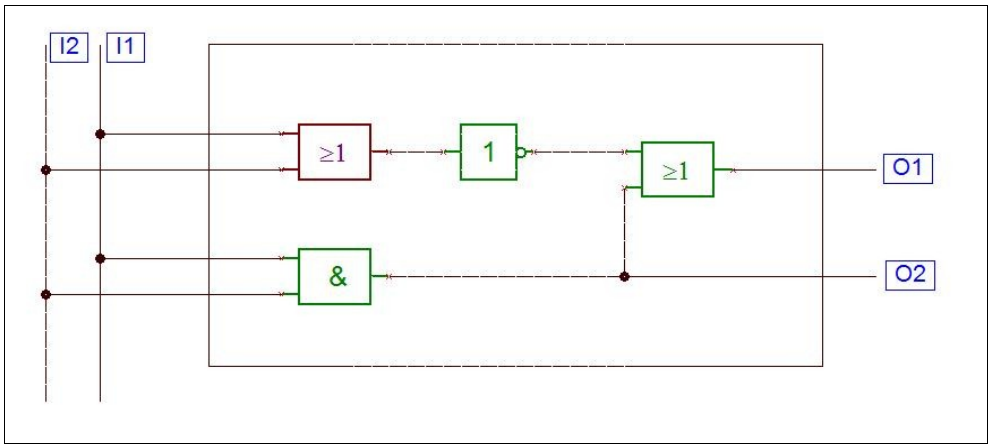


Bild 1.1 Schematischer Aufbau eines Halbaddierers

```
1   LIBRARY ieee;
2   use ieee.std_logic_1164.all;
3
4    entity halfadder_block is
5
6    port ( i1: in bit;
7           i2: in bit;
8           o1: out bit;
9           o2: out bit);
10
11   end halfadder_block;
12
13   architecture halfadder_logic of halfadder_block is
14
15   begin
16          o1 <= not((not(i1 or i2))or(i1 and i2));
17          o2 <= i1 and i2;
18
19   end halfadder_logic;
20
21
```

Bild 1.2 Umsetzung des Halbaddierers mit VHDL

In VHDL werden die Eingänge und Ausgänge einer Schaltung im ENTITY-Bereich festgelegt. Unsere erste Schaltung beinhaltet zwei Ein- und zwei Ausgänge (I1, I2, O1, O2).
Die eigentliche Funktionsfestlegung der Schaltung erfolgt im ARCHITECTURE-Bereich.
Hier werden die Eingänge mit ihren logischen Funktionen den Ausgängen zugeordnet.

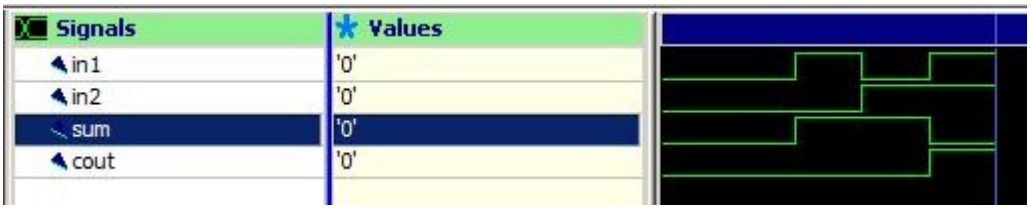
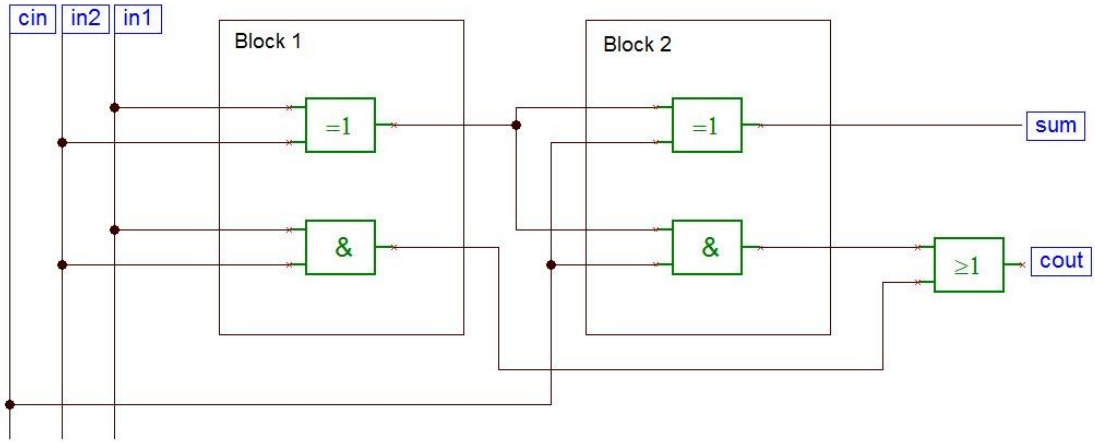


Bild 1.3 Testbench des Halbaddierers mit Simili

Das Testen erfolgt in der Simili Software mithilfe einer so genannten Testbench.
Diese ist ein eigenständiges VHDL File, das der zu testenden Schaltung Eingangssignale vorgibt.

2.2 Volladdierer

Da sich ein Volladdierer aus zwei Halbaddierern und einem ODER-Glied für den Übertrag zusammensetzt wird die bestehende Komponente Halbaddierer in unserem Volladdierer eingesetzt.
Anhand der Wertetabelle kann ein Großteil der Schaltung des Halbaddierers durch XOR Bausteine ersetzt werden.

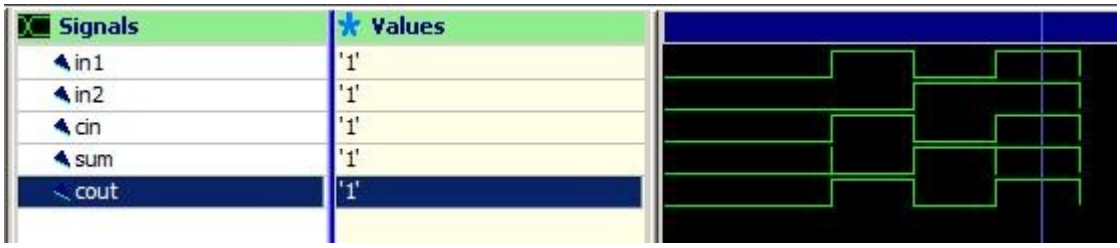


2.1 Schematischer Aufbau eines Volladdierers

Bestehende Komponenten werden mit COMPONENT in das VHDL-File eingeführt. Über die sogenannte PORT MAP können diesen Komponenten entsprechende Signale bzw. PINS übergeben werden.

```
1   LIBRARY ieee;
2   use ieee.std_logic_1164.all;
3
4    entity fulladder_block is
5
6    port ( pin1: in bit;
7           pin2: in bit;
8           pin3: in bit;
9           pin4: out bit;
10          pin5: out bit);
11
12   end fulladder_block;
13
14   architecture fulladder_logic of fulladder_block is
15
16   component halfadder_block
17          port(   i1: in bit;
18                  i2: in bit;
19                  o1: out bit;
20                  o2: out bit);
21   end component;
22
23   signal connect1 : bit;
24   signal connect2 : bit;
25   signal connect3 : bit;
26
27   begin
28          b1: halfadder_block port map(i1=>pin1, i2=>pin2, o1=>connect1, o2=>connect2);
29          b2: halfadder_block port map(i1=>connect1, i2=>pin3, o1=>pin4, o2=>connect3);
30
31          pin5 <= connect2 or connect3;
32
33   end fulladder_logic;
34
35
36
```

2.2 Umsetzung des Volladdierers mit VHDL



2.3 Testbench des Volladdierers mit Simili

2.3 Signallaufzeiten

Um die serielle Addition der Daten zu Testen wird nun der bestehenden Volladdierer im nächsten Punkt mit einem Latch, also einem D-FlipFlop erweitert.
Da das D-FF ein zeitabhängiger Baustein ist muss ab diesem Punkt auf die Signallaufzeiten bzw. die Signalanliegezeiten geachtet werden.
Dies wurde in der Testbench eingearbeitet, muss jedoch bei einer späteren Hardwarelösung auf die Plattform abgestimmt werden.

```
15
16    signal in1,in2,clk,sum : bit;
17
18    begin
19    DUT: serfull_block PORT MAP(in1,in2,clk,sum);
20    DUT_proc : process
21            begin
22             clk<='1';
23             in1<='0';
24             in2<='0';
25             wait for 10 ns;
26             clk<='0';
27             in1<='0';
28             in2<='0';
29             wait for 10 ns;
30             clk<='1';
31             in1<='1';
32             in2<='0';
33             wait for 10 ns;
34             clk<='0';
35             in1<='1';
36             in2<='0';
37             wait for 10 ns;
38             clk<='1';
39             in1<='0';
40             in2<='1';
41             wait for 10 ns;
42             clk<='0';
43             in1<='0';
44             in2<='1';
45             wait for 10 ns;
46             clk<='1';
47             in1<='1';
48             in2<='1';
49             wait for 10 ns;
50             clk<='0';
51             in1<='1';
52             in2<='1';
53             wait for 10 ns;
54
55            end process DUT_proc;
56    end testbench logic;
```

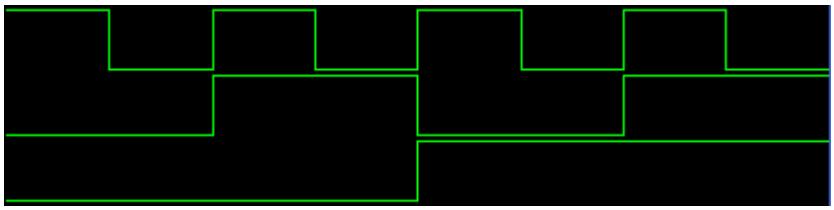


Bild 3.1 Problemdarstellung der Laufzeiten anhand der Testbench

Würden die Signallaufzeiten ab diesem Punkt nicht beachtet werden wäre keine saubere Übergabe des Carry-Bit möglich.
Somit würde die ALU nicht richtig addieren.

2.4 Volladdierer mit Latch

Die Latch (realisiert mit einem D-FlipFlop) wird seriell an den Carry Out des Volladdierers gelegt und dann über den Clock getaktet zurück an den Carry Eingang des Volladdierers gegeben.

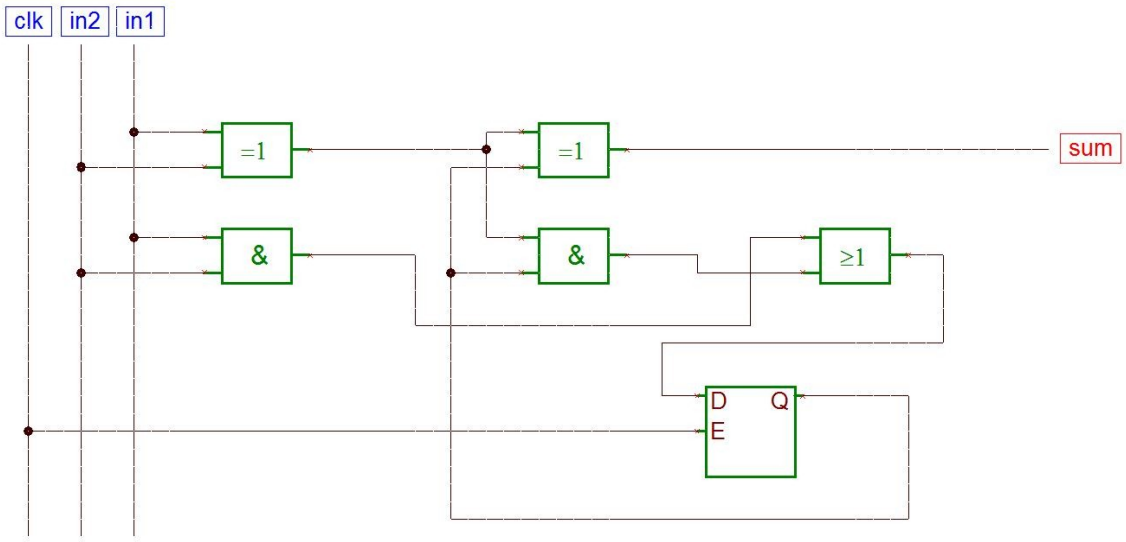


Bild 4.1 Schematischer Aufbau des seriellen Addierers

Zu Darstellungszwecken wird im VHDL Quellcode ein Carry Ausgang mitgeführt. Dieser wurde jedoch nur von uns eingeführt damit die Signallaufzeiten nachvollziehbar bleiben. Die Funktion der Schaltung würde ohne diesen Ausgang unverändert bleiben.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4    entity serfull_block is
5
6   port ( in1: in bit;
7          in2: in bit;
8          clk: in bit;
9          carry: out bit;
10         sum: out bit);
11
12  end serfull_block;
13
14  architecture serfull_logic of serfull_block is
15
16  component fulladder_block
17        port (  pin1: in bit;
18                pin2: in bit;
19                pin3: in bit;
20                pin4: out bit;
21                pin5: out bit);
22  end component;
23
24  signal connect1 : bit;
25  signal connect2 : bit;
26
27  begin
28        b1: fulladder_block port map(pin1=>in1, pin2=>in2, pin3=>connect1, pin4=>sum, pin5=>connect2);
29
30        process(clk)
31        begin
32                if(clk'event and clk='1')then
33                carry <= connect2;
34                connect1 <= connect2;
35                end if;
36        end process;
37
38  end serfull_logic;
39
```

Bild 4.2 Aufbau des seriellen Addierers mit VHDL

Das Carry-Bit liegt über den rot markierten Bereich an, auch wenn dies darstellungstechnisch nicht von uns realisiert wurde.

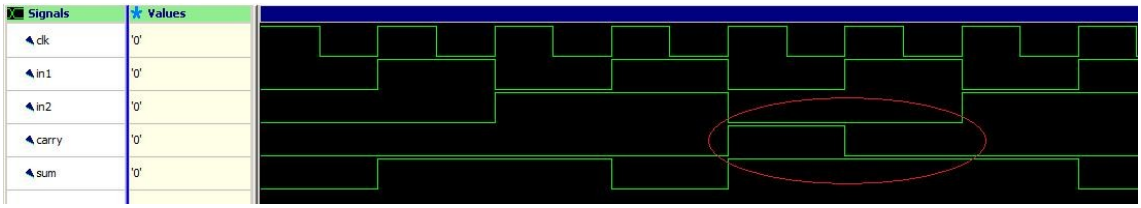


Bild 4.3 Testbench des seriellen Addierers

2.5 Serielles Addierwerk mit Ausgaberegister

Um den Quellcode für eine Hardwarelösung umzusetzen ist der Einsatz eines Schieberegisters vorteilhaft.
Der folgende Quellcode zeigt ein Beispiel für ein 4-Bit SREG.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4
5   entity srg_block is
6           port (  clk: in std_logic;
7                   reset: in std_logic;
8                   seriell: in std_logic;
9                   q: buffer std_logic_vector (0 to 3));
10  end srg_block;
11
12  architecture srg_logic of srg_block is
13
14  begin
15    schieben: process (clk,reset)
16      begin
17            if reset = '1' then q <= "0000";
18          elsif (clk'event and clk = '1') then
19             q(0) <= seriell;
20             q(1) <= q(0);
21             q(2) <= q(1);
22             q(3) <= q(2);
23            end if;
24     end process schieben;
25  end srg_logic;
26
27
```

Bild 5.1 4-Bit Schieberegister

Danach wird der Quellcode des 1-Bit Addierwerks um die Komponente 4-Bit Schieberegister erweitert.

```
1   LIBRARY ieee;
2   use ieee.std_logic_1164.all;
3
4    entity adderreg_block is
5
6   port ( i1: in std_logic;
7          i2: in std_logic;
8          res: in std_logic;
9          clock: in std_logic;
10         summ: out std_logic;
11         sout:buffer std_logic_vector(0 to 3));
12
13
14  end adderreg_block;
15
16  architecture adderreg_logic of adderreg_block is
17
18  component serfull_block
19  port ( in1: in std_logic;
20         in2: in std_logic;
21         clk: in std_logic;
22         sum: out std_logic);
23  end component;
24
25   component srg_block
26         port (  clk: in std_logic;
27                 reset: in std_logic;
28                 seriell: in std_logic;
29                 q: buffer std_logic_vector (0 to 3));
30  end component;
31
32  signal connectl : std_logic;
33
34  begin
35         b1: serfull_block port map(in1=>i1, in2=>i2, clk=>clock, sum=>connectl);
36         summ<=connectl;
37         b2: srg_block port map(reset=>res, clk=>clock, seriell=>connectl,q(0)=>sout(0),q(1)=>sout(1),q(2)=>sout(2),q(3)=>sout(3));
38
39  end adderreg_logic;
40
41
```

Bild 5.2 VHDL Code des Addierwerks mit Register

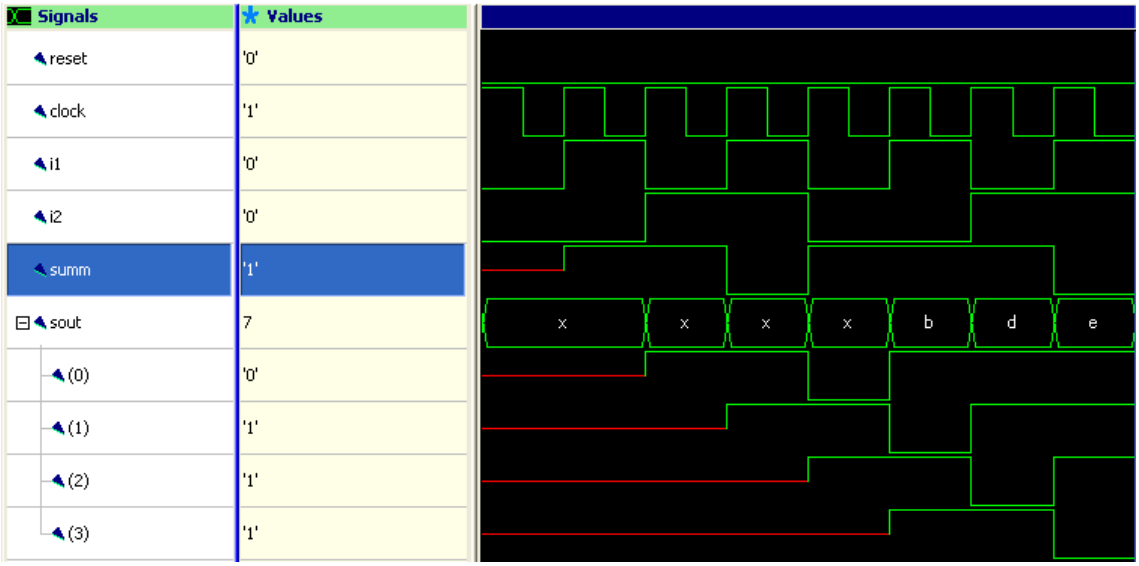


Bild 5.3 Signalverlauf des Addierwerks mit Register