TECHNISCHE UNIVERSITÄT CHEMNITZ 175 Jahre 1836-2011

Rechnerorganisation

Praktikum

Prof. Dr.-Ing. W. Rehm

Praktikum 3

Bearbeiter: Ralph Schlosser

Letzte Änderung: René Oertel, 17.10.2011

RECHNER ARCHITEKTUR

1 Themen

Hier zunächst ein Überblick über die Themen des 3. Praktikums:

- ALUs
- Speicher
- Daten- und Steuerpfad

2 ALUs

2.1 Grundlagen

ALUs (engl. *Arithmetic Logic Units*) gehören neben den im letzten Praktikum behandelten Registern und dem Steuerwerk, mit dem wir uns im nächsten Praktikum noch befassen werden, zu den zentralen Bestandteilen einer CPU. Die Hauptaufgabe einer ALU ist – wie der Name schon andeutet – die Bereitstellung von Funktionseinheiten zur Berechnung von arithmetischen und logischen Verknüpfungen (Addition, Subtraktion, AND, OR, XOR usw.) in einer einzigen, komplexen Einheit.

Abbildung 1 gibt einen Überblick über die Beschaltung einer einfachen ALU, die wir in VHDL realisieren wollen.

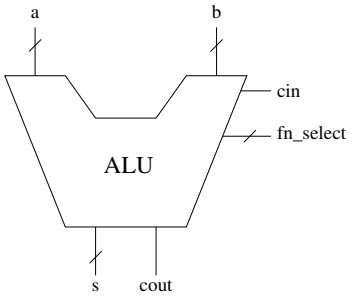

Abbildung 1: Schnittstellen einer ALU

Die ALU bekommt hier 2 Eingabevektoren `a` und `b` und erzeugt einen Ausgabevektor `s` sowie ein auslaufenden Übertrag (engl. *Carry*) `cout`. Daneben existiert noch das `cin`-Signal, das für den einlaufenden Übertrag benutzt wird. Die Auswahl der jeweiligen ALU-Funktion wird über das Steuersignal `fn_select` getroffen. Dieses Steuersignal ist auch ein Bitvektor, da eine ALU üblicherweise mehr als 2 arithmetische/logische Funktionen besitzt.

2.2 Realisierung in VHDL

Eine zu entwerfende 4 Bit-Alu soll die folgenden drei arithmetisch-logischen Operationen auf den Eingabevektoren `a` und `b` unterstützen:

- `ALU_ADD` – Addition der Eingänge `a` und `b` mit eingehenden und ausgehenden Übertrag.

- `ALU_AND` – Logisches UND auf `a` und `b`.
- `ALU_SLL` – *Shift Left Logical*: Alle Bits von `a` werden um eine Bitposition nach links geschoben. Die dabei „frei“ werdende niederwertigste Bitstelle wird mit `0` aufgefüllt; das höchstwertige Bit wird an das Übertrag-Bit `cout` herausgegeben.

Ausgehend von Abbildung 1 und unserer Spezifikation könnte man eine einfache Realisierungsvariante als Verhaltensbeschreibung in VHDL, etwa wie in Listing 1 dargestellt, angeben.

Listing 1: Verhaltensbeschreibung einer einfachen ALU

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  entity f_bit_alu is
6      Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
7             b : in  STD_LOGIC_VECTOR (3 downto 0);
8             cin : in  STD_LOGIC;
9             fn_select : in  STD_LOGIC_VECTOR (2 downto 0);
10            s : out  STD_LOGIC_VECTOR (3 downto 0);
11            cout : out  STD_LOGIC);
12 end f_bit_alu;
13
14 architecture a of f_bit_alu is
15
16 constant ALU_ADD: std_logic_vector(2 downto 0) := "000";
17 constant ALU_AND: std_logic_vector(2 downto 0) := "001";
18 constant ALU_SLL: std_logic_vector(2 downto 0) := "010";
19
20 signal temp_s, temp_a, temp_b, temp_cin: std_logic_vector(4 downto 0);
21
22 begin
23
24     temp_a <= '0' & a;  -- Erweitern auf 5 Bit und 5. Bit auf 0 setzen
25     temp_b <= '0' & b;   -- (z.B. aus "1011" wird "01011")
26     temp_cin <= "0000" & cin;
27
28 p0: process(temp_a, temp_b, temp_cin, fn_select)
29         begin
30             case fn_select is
31                 when ALU_ADD =>
32                     temp_s <= temp_a + temp_b + temp_cin;
33                 when ALU_AND =>
34                     temp_s <= temp_a and temp_b;
35                 when ALU_SLL =>
36                     temp_s(4) <= temp_a(3);
37                     temp_s(3) <= temp_a(2);
38                     temp_s(2) <= temp_a(1);
39                     temp_s(1) <= temp_a(0);
40                     temp_s(0) <= '0';
41                 when others =>
42                     temp_s <= "00000";
43             end case;
44         end process p0;
45
46 cout <= temp_s(4);
47 s <= temp_s(3 downto 0);
48 end a;
```

Wie man sieht, besteht die ALU im Wesentlichen aus einem `case`-Konstrukt, in dem für alle möglichen Steuersignale die Abbildung der Eingabevektoren auf den Ergebnisvektor beschrieben ist. Die Codes der oben spezifizierten Funktionen wurden aus Gründen der besseren Lesbarkeit als Konstanten definiert. Die vorgesehene Breite von `fn_select` (3 Bit) erlaubt die Implementation von bis zu 8 verschiedenen ALU-Funktionen.

Der Ergebnisvektor hat eine Breite von 4 Bit, entsprechend der Maßgabe, dass es sich um eine 4 Bit-ALU handeln soll.

Die eigentliche Berechnung der Funktionsergebnisse geschieht mit den 5 Bit breiten Vektoren `temp_a` und `temp_b`. Falls ein nicht definierter Funktionscode über `fn_select` an die ALU angelegt wird, erzeugt die ALU das Ergebnis `0000`.

3 Speicher

Wir wollen in diesem Abschnitt noch die VHDL-Realisierung einer weiteren zentralen Komponente von Rechnern betrachten: Den Speicher. Dabei beschränken wir uns auf die Modellierung eines RAM (engl. *Random Access Memory*); die Modellierung von ROM (engl. *Read Only Memory*) ergibt sich dann analog zum RAM-Design.

Halten wir zunächst eine beispielhafte prinzipielle Spezifikation eines solchen RAMs fest, den wir weiter unten in VHDL realisieren werden:

- Der RAM unterstützt (laut Definition) die Operationen Lesen und Schreiben.
- Zum Lesen bzw. Schreiben einer bestimmten Speicherzelle muss eine Adresse übermittelt werden. Diese Adresse wird in `address` angegeben (`m` Bits breit, je nach gewünschter Tiefe).
- Der RAM soll getrennte Datenleitungen für Lesen (`read_data`) und Schreiben (`write_data`) besitzen – jeweils mit einer Breite von `n` Bits.

  Hinweis: Die meisten RAMs die man als einzelne ICs findet besitzen einen Datenbus über den sowohl gelesen als auch geschrieben wird (natürlich nicht gleichzeitig). In FPGAs verwendet man der Einfachheit halber jedoch meist getrennte Ein- und Ausgänge.
- Zum Schreiben muss ein sog. *Write Enable*-Signal (`we`) auf `1` gesetzt werden.
- Das Schreiben geschieht synchron zu einem Takt, und der RAM übernimmt die zu schreibenden Daten mit der steigenden Taktflanke von `write_data`.
- Das Lesen soll komplett asynchron (also ohne jegliche Taktung) stattfinden. Dass heißt, `read_data` folgt unmittelbar `address`.

Auf weitere Eingänge, wie sie bei „realen Chips“ vorhanden sind (etwa *Chip Select* zur Aktivierung des jeweiligen RAMs) verzichten wir hier der Einfachheit halber. In der Abbildung 2 ist die Beschaltung des eben beschriebenen Speichers zu sehen.

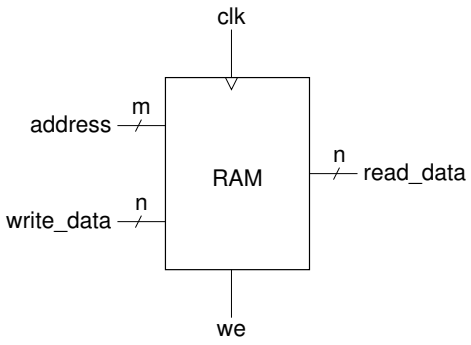

Abbildung 2: Schnittstellen des RAM

Hinweis:

Hierbei handelt es sich um einen relativ einfachen statischen RAM (SRAM), der an sehr vielen Stellen in digitalen Systemen eingesetzt wird. Wird eine sehr hohe Speicherdichte benötigt, wie etwa für den Hauptspeicher „normaler“ Computer, greift man auf sog. dynamische RAMs (DRAMs) zurück. Moderne DRAMs erfordern aber relativ komplizierte Kommunikationsprotokolle, welche mit der Ansteuerung einfacher SRAMs kaum Ähnlichkeit besitzen.

In den nächsten beiden Abschnitten werden wir zwei Varianten kennenlernen, um einen SRAM zu realisieren.

3.1 Variante 1

Der VHDL-Quellcode in Listing 2 gibt eine Verhaltensbeschreibung des oben beschriebenen RAMs an. Dabei wird in Hochsprachen-Manier ein Signal eines entsprechenden Typs angelegt. Dieses Signal stellt nichts weiter als ein Feld dar, auf dessen Elemente man mittels eines Index (die Adresse) zugreifen kann. Die Organisation des RAMs beträgt dabei 8×8 Bits, d. h. es wird eine 3 Bit-Adresse benötigt.

Listing 2: Verhaltensbeschreibung eines RAM

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use IEEE.STD_LOGIC_ARITH.ALL;
5
6  entity ram_1 is
7      Port ( clk : in  STD_LOGIC;
8             we : in  STD_LOGIC;
9             address : in  STD_LOGIC_VECTOR (2 downto 0);
10            write_data : in  STD_LOGIC_VECTOR (7 downto 0);
11            read_data : out  STD_LOGIC_VECTOR (7 downto 0));
12 end ram_1;
13
14 architecture a_behav of ram_1 is
15
16        -- Einen entsprechenden Typ fuer den Speicher anlegen
17        type memory_t is array(0 to 7) of std_logic_vector(7 downto 0);
18        -- Ein Signal mit diesem Typ anlegen (der eigentliche Speicher)
19        signal memory: memory_t;
20
21 begin
22     -- Auslesen des RAMs
23     -- read_data folgt address ohne jeglichen Takt
24     -- --> Lesen ist asynchron
25     read_data <= memory(CONV_INTEGER(address));
26
27 p0: process(clk)
28     begin
29         if clk'event and clk = '1' then
30             -- geschrieben wird mit der steigenden Taktflanke
31             -- --> Schreiben ist synchron
32             if we = '1' then
33                 memory(CONV_INTEGER(address)) <= write_data;
34             end if;
35         end if;
36     end process p0;
37 end a_behav;
```

Soll von einem Speicherelement gelesen, bzw. auf ein Speicherelement geschrieben werden, so wird der jeweilige Index mittels der Funktion `CONV_INTEGER` ermittelt. Diese wandelt einen `std_logic_vector` in eine Ganzzahl um, und man kann ohne Probleme auf die einzelnen Elemente des Arrays zugreifen.

In der Praxis wird man allerdings eher davon absehen, Speicher auf die eben geschilderte Weise für Synthesezwecke zu modellieren, da bei der Synthese für eine konkrete Hardware für jedes Bit des RAMs eine ganze Menge Ressourcen verschwendet werden. Das liegt daran, dass die Synthesesoftware üblicherweise für jede einzelne Speicherzelle ein D-Flipflop anlegt (in unserem Fall wären das schon $8 \times 8 = 64$ Flipflops). In obigen VHDL-Code finden wir ja auch deutlich das bereits bekannte Konstrukt zum Erzeugen von D-Flipflops. Hinzu kommt noch je nach Größe eine recht aufwendige kombinatorische Logik zur Dekodierung der Adresse sowie zur Ausgabe der gelesenen Daten (Multiplexer).

Hinweis: Als weiterführende Übung kann einmal versucht werden den oben beschriebenen RAM zu synthetisieren. Wenn man einen Blick in den FPGA-Editor (unter *Implement* → *Place&Route* → *View/Edit Routed Design*) wirft, kann man genau nachvollziehen, wie das Synthesetool den RAM umgesetzt hat.

Da viele programmierbare Logikbausteine schon von Haus aus Dinge wie RAMs als spezielle Komponenten enthalten, verwendet man deswegen die vom Hersteller gelieferten *Vendor Libraries* (oft auch *Hard-Macros* genannt). Dies trifft natürlich nicht nur für RAMs, sondern für sämtliche Spezialkomponenten wie z. B. spezielle Rechenwerke oder Ähnliches zu. Zwar erzeugt man damit eine gewisse Abhängigkeit, aber meist lässt sich dies einfach anpassen.

3.2 Variante 2

In dieser Variante (Listing 3) benutzen wir den Block RAM des Spartan-3E. Da dort direkt Speicher auf dem Chip implementiert ist, der nicht erst aus Logikzellen und Flipflops zusammengebaut werden muss, ist dieser Speicher wesentlich effizienter. Allerdings ist er auch hardwareabhängig, das heißt, dass diese Module nur auf Xilinx-FPGAs funktionieren (siehe auch *Chapter 4, Using Block RAM*, [3]). Das erfordert Anpassungen wenn das Design portiert werden soll, allerdings bieten alle Hersteller ähnliche Speicherbausteine an, so dass dies kein großes Problem ist.

Listing 3: Verhaltensbeschreibung eines Block RAM

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  library UNISIM;
5  use UNISIM.VComponents.all;
6
7  entity ram_2 is
8      Port ( clk : in  STD_LOGIC;
9             we : in  STD_LOGIC;
10            address : in  STD_LOGIC_VECTOR (3 downto 0);
11            write_data : in  STD_LOGIC_VECTOR (7 downto 0);
12            read_data : out  STD_LOGIC_VECTOR (7 downto 0));
13 end ram_2;
14
15 architecture b_behav of ram_2 is
16
17 signal addr: std_logic_vector(10 downto 0);
18
19 begin
20     addr <= "0000000" & address; -- Erweiterung auf 11-bit
21
22    -- RAMB16_S9: 2k x 8 + 1 Parity bit Single-Port RAM
23    --            Spartan-3E
24    -- Xilinx HDL Language Template, version 13.2
25
26    RAMB16_S9_inst : RAMB16_S9
27    generic map (
28       INIT => X"000", --  Value of output RAM registers at startup
29       SRVAL => X"000", --  Ouput value upon SSR assertion
30       WRITE_MODE => "WRITE_FIRST" --  WRITE_FIRST, READ_FIRST or NO_CHANGE
31         -- Zeilen
32         -- INIT_00 bis INIT_3F, sowie
33         -- INITP_00 bis INITP_3F entfernt, weil default '0'
34    )
35    port map (
36       DO => read_data,      -- 8-bit Data Output
37       DOP => open,          -- 1-bit parity Output, nicht verbunden
38       ADDR => addr,         -- 11-bit Address Input
39       CLK => clk,           -- Clock
40       DI => write_data,     -- 8-bit Data Input
41       DIP => "0",           -- 1-bit parity Input
42       EN => '1',            -- RAM Enable Input
43       SSR => '0',           -- Synchronous Set/Reset Input
44       WE => we              -- Write Enable Input
45    );
46
47    -- End of RAMB16_S9_inst instantiation
48 end b_behav;
```

Die Größe des RAM lässt sich nicht verändern, man muss immer ein komplettes Modul benutzen. Allerdings kann man einen RAM mit geringerer Bitbreite simulieren, indem man einen Adressvektor der gewünschten Breite mit dem vordefinierten Adressvektor des Block RAMs konkateniert bzw. koppelt (siehe Zeile 21). Die Daten werden dann entsprechend der variablen Adressbits im RAM verteilt. Die Initialbelegung des RAMs wurde hier entfernt, weil sie laut Dokumentation standardmäßig mit `0` konfiguriert wird. Die VHDL-Vorlage für die Block RAMs dieser Art erhält man in der ISE Design Suite über *Edit → Language Templates → VHDL → Device Primitive Instantiation → Spartan-3E → RAM/ROM → Block RAM → Single-Port*.

4 Versuchsvorbereitung

Hinweis:
In Vorbereitung auf den Versuch ist es empfehlenswert, dass die folgenden Vorarbeiten von jeder Versuchsgruppe durchgeführt und die schriftlich ausgearbeiteten stichpunktartigen Antworten zu diesen Fragen und ggf. vorbereitete VHDL-Dateien zur praktischen Übung mitgebracht werden!

1. Warum werden die Eingänge in der ALU aus Abschnitt 2.2 intern auf 5 Bit erweitert? Eigentlich handelt es sich doch um eine 4 Bit ALU, oder?
2. Erläutern Sie, wie diese ALU den auslaufenden Übertrag ermittelt!
3. Wie lässt sich die in Aufgabe 1 zu implementierende arithmetische Rechtsschiebeoperation prinzipiell realisieren?
4. Versuchen Sie den VHDL-Code der folgenden Aufgaben weitestgehend vorzubereiten!

5 Aufgaben

1. Erweitern Sie die in Abschnitt 2.2 vorgestellte 4 Bit-ALU um folgende Funktionen:
   (a) `ALU_SRA` – *Shift Right Arithmetical*: Der 4 Bit-Vektor `a` wird diesmal nach rechts geschoben. Allerdings handelt es sich nicht um eine logische Schiebeoperation, sondern um eine arithmetische. D. h. für den Fall, dass es sich bei `a` um eine negative Zahl in 2er-Komplementdarstellung handelt, muss als Ergebnis auch eine negative Zahl herauskommen (also z. B. aus -4 wird -2).
   (b) `ALU_XOR` – Logisches, bitweises XOR auf `a` und `b`.
   (c) `ALU_ROL` – *Rotate Left*: Jede Bitposition von `a` wird um eine Stelle nach links geschoben. Das höchstwertige Bit (MSB) „fällt heraus“ und wird als niederwertigstes Bit (LSB) eingefügt.
   (d) ? – Eine selbst gewählte Funktion, die `a` und `b` verknüpft oder auf `a` oder `b` operiert, soll implementiert werden.
2. Nun sollen die in Abschnitt 2.2 angegebenen und die neu implementierten Funktionen auf der Hardware getestet werden. Schreiben Sie dazu ein VHDL-Design, das folgende Funktionalität besitzt:
   - Die zwei 4 Bit-Eingabevektoren `a` und `b` können über die Schiebeschalter eingestellt werden.
   - Durch Drücken eines Tasters sollen alle ALU-Funktionen nacheinander ausgewählt werden können. Dass heißt, zu Beginn wird die erste Operation durchgeführt, drückt man einen Taster, so wird auf die zweite ALU-Funktion umgeschaltet usw. Sind alle Funktionen einmal dran gewesen, fängt man wieder von vorne an.
     Das Single-Step-Modul aus dem zweiten Praktikum muss hier entsprechend einbezogen werden!
   - Das Ergebnis der gerade ausgeführten Funktion wird jedesmal in geeigneter Weise auf den LEDs oder 7-Segment-Anzeigen angezeigt.

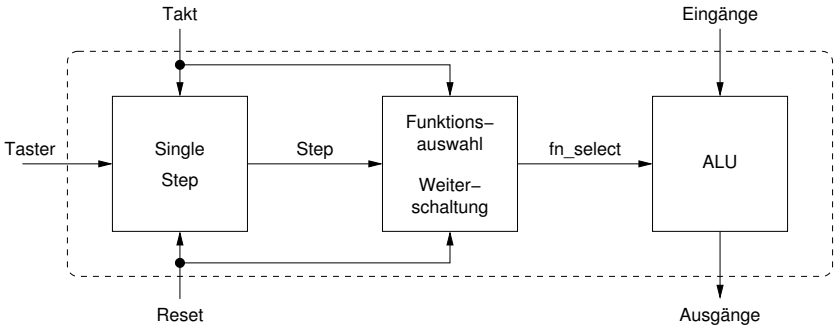

Abbildung 3: Vereinfachtes hierarchiches Strukturmodell für den Test der ALU

Das Test-Design soll dabei hierarchisch aufgebaut werden und kann z. B. wie in Abbildung 3 gestaltet sein.

3. Unter Benutzung des Block RAMs soll ein 16×4 Bit-Speicher implementiert werden. Als Taktquelle ist wieder der über Pin B8 zur Verfügung gestellte 50 MHz-Takt zu verwenden.

   Die Speicheradresse soll an den Schiebeschaltern *SW3–0* und das zu schreibende Datum soll an den Schiebeschaltern *SW7–4* eingestellt werden können.

Das eingestellte Datum soll jedoch nur dann geschrieben werden, wenn ein entsprechender Taster gedrückt wird.

Die vom RAM ausgegebenen Lesedaten sollen auf LEDs *LD3–0* ausgegeben werden. Dabei soll dies durch einen weiteren Taster gesteuert werden. Ist dieser gedrückt, so sollen die Daten vom RAM direkt ausgegeben werden. Ist er nicht gedrückt, so soll immer eine `0` ausgegeben werden.

Weiterhin sind geeignete Tests durchzuführen um festzustellen, ob der RAM wirklich etwas speichert.

Hinweise, Berichtigungen und Kritik zu den Übungsunterlagen bitte an:

- René Oertel <`rene.oertel@cs.tu-chemnitz.de`>

Literatur und wichtige Links

[1] *The VHDL Cookbook, First Edition*
Peter J. Ashenden
`http://www.vhdl.org/misc/VHDL-Cookbook.ps.tar.Z`
`http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf`

[2] *Schaltungsdesign mit VHDL*
Gunther Lehmann, Bernhard Wunder, Manfred Selz
Eine ausführliche Einführung in VHDL und nebenbei ein gutes Referenzhandbuch. Ein „must-have“ ;-) Das Buch, bestehend aus mehreren `.pdf`-Dateien und Beispielen, ist kostenlos im Internet verfügbar:
`http://www.itiv.kit.edu/653.php`

[3] *Spartan-3 Generation FPGA User Guide*
Xilinx
`http://www.xilinx.com/support/documentation/user_guides/ug331.pdf`

17. Oktober 2011