# *Übung 5*

## Zusammensetzen der 16-Bit ALU

Mit dem verifizierten Baustein aus Übung 4 erstellen wir eine 16-Bit ALU. Die Schnittstelle nach außen ist in folgender Tabelle festgelegt:

| ***Port*** | ***Richtung*** | ***Typ*** |
|---|---|---|
| S(2:0) | IN | STD_LOGIC_VECTOR |
| A(15:0), B(15:0) | IN | STD_LOGIC_VECTOR |
| Q(15:0) | OUT | STD_LOGIC_VECTOR |
| Z_OUT | OUT | STD_LOGIC |

**Tabelle 6: Portbeschreibung ALU16**

Der Entwicklungsprozess soll folgendermaßen ablaufen:

1. Erstellen Sie eine Spezifikation für die ALU, d.h. erzeugen Sie in VHDL eine **entity ALU16** und **architecture spec**, die Verhaltensbeschreibung der 16-Bit ALU
2. Entwerfen Sie eine Gatternetzliste (*alu16_impl*) aus Grundbausteinen und der bereits entwickelten **ALU4**
3. Geben Sie die Netzliste als Blockschaltbild in ActiveHDL ein.
4. Eingangssignale der einzelnen Komponenten, welche immer das Signal 1 bzw. 0 führen (z.B. **Z_IN** der ALU4 mit den niederwertigsten Bits), können über die Symbole **VCC** und **GND** (*Diagram->…*) auf Betriebsspannung(1) bzw. Masse(0) gesetzt werden.
5. Verifizieren bzw. validieren Sie Ihren Entwurf.

## Verifikation – Validierung

Da die ALU insgesamt über 35 Eingabebits verfügt, müssten zur Verifikation $2^{35}$ verschiedene Testmuster generiert und simuliert werden. Aufgrund der großen Anzahl wird die ALU nicht vollständig verifiziert. Zur Validierung können einerseits speziell ausgewählte Eingangsbelegungen simuliert werden, welche bestimmte Fehler sichtbar machen (z.B. fehlerhafte Carry-Propagierung). Andererseits besteht auch die Möglichkeit, eine Simulation mit Zufallsmustern durchzuführen.

Zur Erzeugung von Zufallsmustern steht die Bibliotheksfunktionen `RAND_INT` im Package `xilinxcorelib.ip_utils_math` zur Verfügung.

```
FUNCTION rand_int (seed : integer; maxval : integer)
    RETURN integer;


-------------------------------------------------------------------------
-- rand_int
--
-- Purpose: To randomly generate an integer between 0 and maxval.
--          (including 0 and maxval)
--
-- Inputs:
-- seed : Any integer value. Given the same seed, this function will
-- always generate the same output. It is recommended to always
-- use the output of this function as the seed for the next call.
-- maxval : The maximum value allowed. The randomly generated value will
-- not exceed this value.
```

```
--
-- Return:
-- pseudo-random integer from 0 to maxval.
--
-- Details:
-- This function is based on Linear Feedback.
------------------------------------------------------------------------
```

Eine weitere Möglichkeit zur Erzeugung von Zufallszahlen bietet die Prozedur `uniform` aus dem Package `IEEE.MATH_REAL`, welche gleichverteilte Zufallszahlen zwischen 0.0 und 1.0 liefert. Diese werden zur Testmustererzeugung mit dem größtmöglichen Testmusterwert multipliziert und auf ihren ganzzahligen Anteil beschränkt.

```
procedure uniform (variable seed1,seed2:inout positive; variable x: out
real);
```

seed1 und seed2 werden mit einer Zahl von 1 ... 2147483562 bzw. 1 … 2147438398 initialisiert und bei jedem Aufruf wieder übergeben, x enthält die erzeugte Zufallszahl.

Beispiel:

```
process is
    variable seed1,seed:positive:=1;
    variable ar:real;
    variable ai:integer;
begin
    uniform (seed1,seed2,ar);
    ai:=integer(ar*real(2**16)); --entsprechend casten
...
end process;
```

- Erstellen Sie für eine Testbench ein neues Blockschaltbild *alu16_tb.bde*.
- Wählen Sie spezielle Testmuster aus, die mögliche Fehler im Design aufdecken können, und dokumentieren Sie dazu ihre Testabsichten
- Testen Sie das Design mit ihren Testmustern sowie mit einer Reihe von Zufallsmustern, wobei alle Testmuster zusammen mit ihren Testantworten in einer Log-Datei mitprotokolliert werden müssen.
- Schreiben Sie in einer Programmiersprache ihrer Wahl (C++, Java,…) ein Programm zur maschinellen Auswertung der Log-Datei. Dazu werden aus den in der Log-Datei dokumentierten Testmustern die zu erwartenden Testantworten berechnet und mit den tatsächlichen Antworten verglichen.
- Beheben Sie alle Fehler im Design.