ALU des Rechenwerkbausteins Am2901

Dokumentation VHDL

Praktikum: Einführung in die Rechnerarchitektur
Sommersemester 2018
Gruppe 33

Gruppenmitglieder	Franziska Steinle: Projektleitung (03702739)
	Roland Würsching: Dokumentation (03648533)
	Ruben Bachmann: Vortrag (03693902)
Eingereicht am	08.07.2018

Inhaltsverzeichnis

1. Einleitung

Der mikroprogrammierbare Rechner („MI-Maschine“) entspricht dem Von-Neumann-Modell, d.h. er besteht aus einem **Leitwerk** zur Steuerung der anderen Komponenten, einem **Rechenwerk** zur logischen und arithmetischen Verarbeitung von 16-Bit Ganzzahlen, einem **Hauptspeicher** mit 16-Bit breiten Adressen und einem **Ein-/Ausgabewerk**.

Dieses Projekt behandelt ausschließlich das Rechenwerk. Im Mittelpunkt steht dabei der Rechenwerkbaustein Am2901. Dieser erhält 16-Bit breite Eingabewörter aus verschiedenen Quellen: 16 Register über die Adressfelder A und B, das Hilfsregister Q, der Datenbus und ein Konstantenfeld. Aus der Eingabe werden durch eine Quellsteuerung zwei Wörter ausgewählt und durch die arithmetisch-logische Einheit (ALU) verarbeitet. Das Ergebnis kann durch eine Zielsteuerung in ein Register oder auf den Daten-/Adressbus geschrieben werden, oder einfach verworfen werden. **Ziel dieses Projekts ist die Entwicklung der ALU in VHDL.**
Folgende Abbildung zeigt das Schema der MI-Maschine mit gekennzeichnetem Rechenwerk.

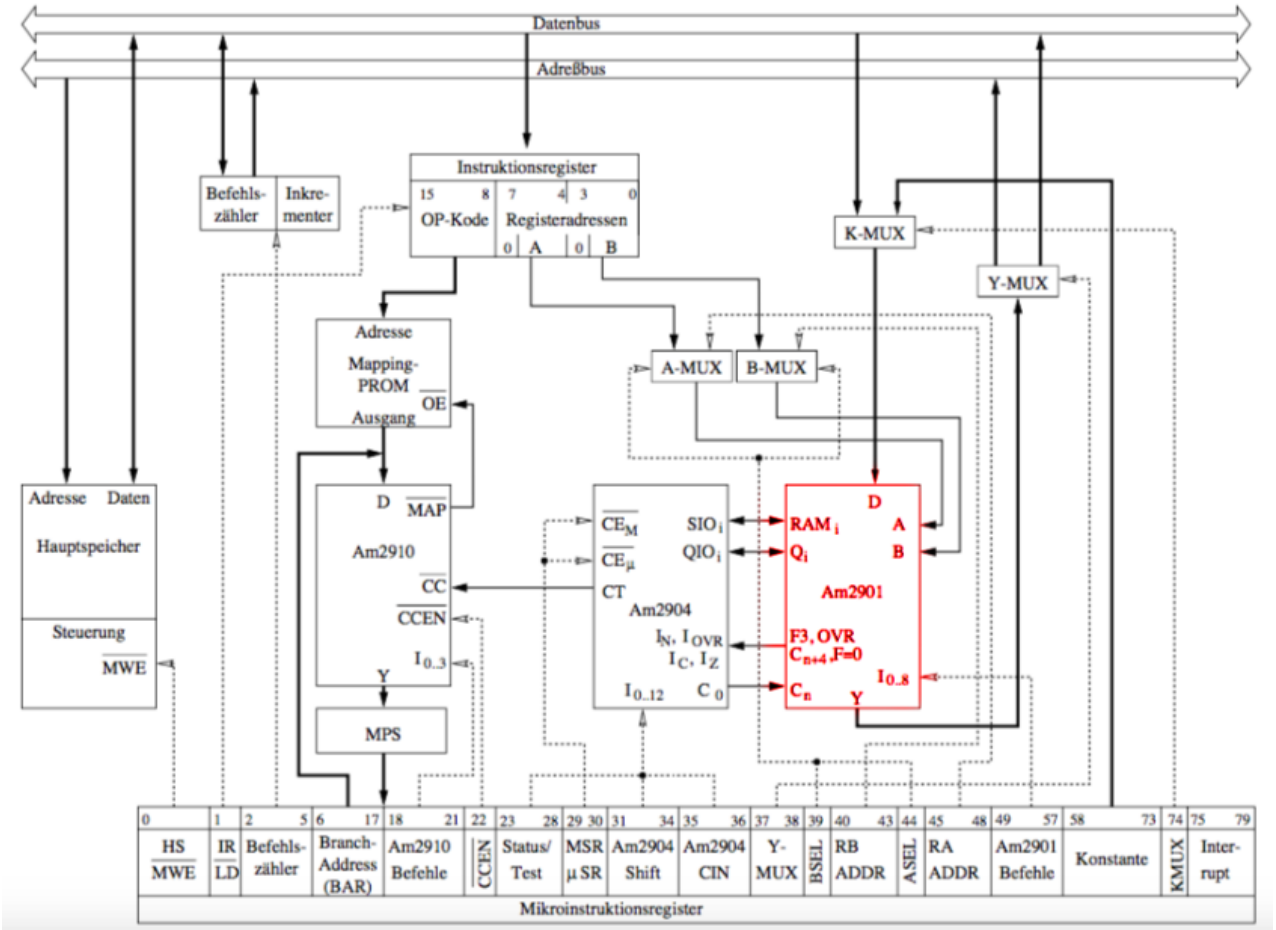

Abbildung 1 MI-Maschine (rot: Am2901 Rechenwerk)

2. Lösungsansatz

2.1. Beschreibung

Der ALU-Baustein des Rechenwerks wird mit einer Entity und einer Architecture modelliert. In der Entity werden die folgenden Ports definiert.

Eingänge:

- `r_in`, `s_in`: Datenwörter an den Eingängen R und S (unsigned 16-Bit Vektoren)
- `carry_in`: Carry-Eingang (1 Bit)
- `op`: Funktionssteuerung (3-Bit Vektor)
- `clk`: Basistakt (1 Bit)
- `ce`: CE-Eingang (1 Bit)

Ausgänge:

- `f_out`: Ergebnis der Rechenoperation am F-Ausgang (16-Bit Vektor)
- `carry_out`: Carry-Ausgang (1 Bit)
- `sign`: Vorzeichen-Flag (1 Bit)
- `zero`: Null-Flag (1 Bit)

In der Architecture wird die Rechnung durchgeführt.

Zunächst wird ein Signal `temp` als 17-Bit Vektor deklariert. Aus diesem werden nach der Rechnung die Statusflags und das Ergebnis gelesen.

Dann werden alle Eingabewerte auf Gültigkeit geprüft, um undefinierte Ergebnisse zu vermeiden. Gegebenenfalls werden Default-Werte zugewiesen.

Der Prozess reagiert auf den Takt `clk` und das CE-Signal `ce`. Bei einer steigender Flanke und gesetztem CE-Signal soll die ALU aktiv werden.

Nun wird mit einem case-Konstrukt die gewünschte Rechenoperation in `op` abgefragt. Diese wird dann folgendermaßen durchgeführt.

ADD: R + S
Die Wörter `r_in` und `s_in` werden addiert und in `temp` gespeichert. Falls `carry_in` gesetzt ist, wird

zusätzlich 1 addiert. Um die Längen anzugleichen müssen `r_in` und `s_in` um ein höchstes 0-Bit erweitert werden.

SUBR: S - R

Das Wort `r_in` wird von `s_in` abgezogen und in `temp` gespeichert. Ist `carry_in` nicht gesetzt, wird zusätzlich 1 abgezogen. Auch hier müssen die Wörter um ein Bit erweitert werden.

SUBS: R – S

Wie SUBR, mit vertauschten Operanden.

OR, AND, NOTRS, EXOR, EXNOR:

Das Ergebnis der logischen Operationen wird in den unteren 16 Bit von `temp` gespeichert, das 17. Bit (Übertrag) bleibt immer 0.

Außerhalb des Prozesses wird das Zwischenergebnis `temp` verarbeitet. Das 17. Bit von `temp` stellt den Übertrag dar und wird in `carry_out` geschrieben. Die unteren 16 Bit von `temp` sind das Ergebnis und werden nach `f_out` geleitet. Sind diese 16 Bit alle 0, so wird das Null-Flag `zero` gesetzt. Ist das höchste Bit des Ergebnisses, also das 16. Bit von `temp`, gesetzt, so ist das Ergebnis negativ und es wird das Vorzeichen-Flag `sign` gesetzt.

2.2. Abweichungen von der Spezifikation

2.2.1. Sensitivitätsliste

Anders als in der Spezifikation angegeben, reagiert die Sensitivitätsliste des ALU-Prozesses nicht nur auf den Takt, sondern auch auf das CE-Signal. Dies hat zur Folge, dass bei gesetztem CE-Signal die Eingabewerte immer geprüft werden. Die Synchronisation ist dadurch nicht gefährdet, da die ALU nur bei einer steigenden Taktflanke rechnet.

2.2.2. Eingabekontrolle

Der implementierte Prozess prüft vor dem Rechnen die Eingabewerte. Dabei sollen die beiden Datenwörter im Bereich 0 bis $2^{16} - 1$ liegen, und es sollen keine undefinierten Eingaben vorkommen.

2.3. Bewertung

Die Verwendung vom Typ *unsigned* für die beiden Eingabewörter R und S ist besser geeignet als der Typ *std_logic_vector*, da bei der Berechnung kein Cast nötig ist.
Die Verwendung eines einzigen `temp`-Vektors für das Ergebnis samt allen Statusflags ist übersichtlicher und einfacher als eine explizite Berechnung des Carry-Ausgangs.
Durch die Kontrolle der Eingabewerte sind alle Ausgaben fest definiert. Dies ist erwünscht, da die ALU Bestandteil einer größeren Komponente ist und undefiniertes Verhalten evtl. zu „silent bugs“ führen könnte.

Im Rückblick wurde eine gute Wahl des Ansatzes getroffen.

3. Anwenderdokumentation

3.1. Beschreibung

Für dieses Projekt wurden folgende zusätzliche Vorgaben gestellt:

- Der eigentliche Rechenwerkbaustein, der aus vier 4 Bit Komponenten besteht, wird zu einem 16 Bit Baustein zusammengefasst.
- Der Basistakt beträgt 50 MHz.
- Ein Effektivtakt (4 Basistakte) umfasst das Laden der Daten aus den Registern, die Rechenoperation und das Speichern in die Register.
- Alle Bausteine werden mit dem Basistakt und einem/mehrere CE Signal versorgt. Ist mindestens einer der CE Eingänge gesetzt, so soll der Baustein aktiv sein.
- Jeder Baustein muss seine Funktion innerhalb eines Basistaktes erfüllen können.

Die von der Quellsteuerung ausgewählten Daten liegen an den Eingängen R und S der ALU an. Außerdem besitzt die ALU einen Carry-Eingang (C_n) und einen CE-Eingang. Eine Funktionssteuerung wählt eine der acht möglichen Operationen aus:

- ADD: $R + S$
- SUBR: $S - R$
- SUBS: $R - S$
- OR: $R \vee S$
- AND: $R \wedge S$
- NOTRS: $\overline{R} \wedge S$
- EXOR: $R \oplus S$
- EXNOR: $\overline{R \oplus S}$

Die ausgewählte Operation wird ausgeführt und das Ergebnis auf den F-Ausgang der ALU geleitet. Schließlich müssen die folgenden Flags des Statusregisters korrekt gesetzt werden:

- Zero-Flag ($F = 0$)
- Sign-Flag (F_3)
- Carry-Ausgang (C_{n+4})

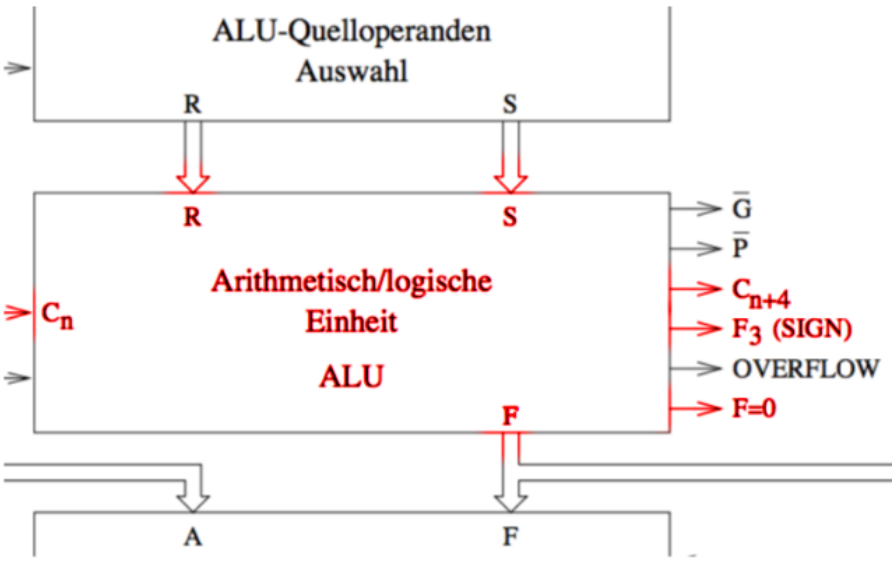

Abbildung 2 ALU (rot: relevante Komponenten)

3.2. Voraussetzungen

Die Implementierung wurde auf **Ubuntu 16.04** (64-bit) getestet. Ebenfalls benötigt werden der VHDL-Simulator **GHDL** (0.33) und **GTKWave Analyzer** (v3.3.66).

3.3. Kompilieren

Die Dateien `ALU.vhdl`, `ALU_tb.vhdl`, `ALU_config.gtkw` und `Makefile` müssen im selben Verzeichnis liegen. Mit

```
$ make alu
```

wird die Testbench kompiliert und ausgeführt; das Ergebnis wird im Terminal angezeigt und als Wellendatei `ALU.vcd` gespeichert. Diese Datei kann mit

```
$ make gtk
```

in GTKWave dargestellt werden. Um alle erzeugten Dateien zu entfernen, kann

```
$ make clean
```

aufgerufen werden.

3.4. Bedienung

Die Testbench kann nach dem Kompilieren mit `./alu_tb` ausgeführt werden. Eine fehlerhafte Operation wird durch einen *assertion error* angezeigt, Undefinierte Werte (z.B. X, U) können *assertion warnings* erzeugen. Eine korrekte Implementierung sollte keine errors aufweisen. Für eine bessere Visualisierung der ALU bietet sich GTKWave an: Mit `make gtk` wird im Fenster **Waves** der gesamte Wellenverlauf der Testbench dargestellt.

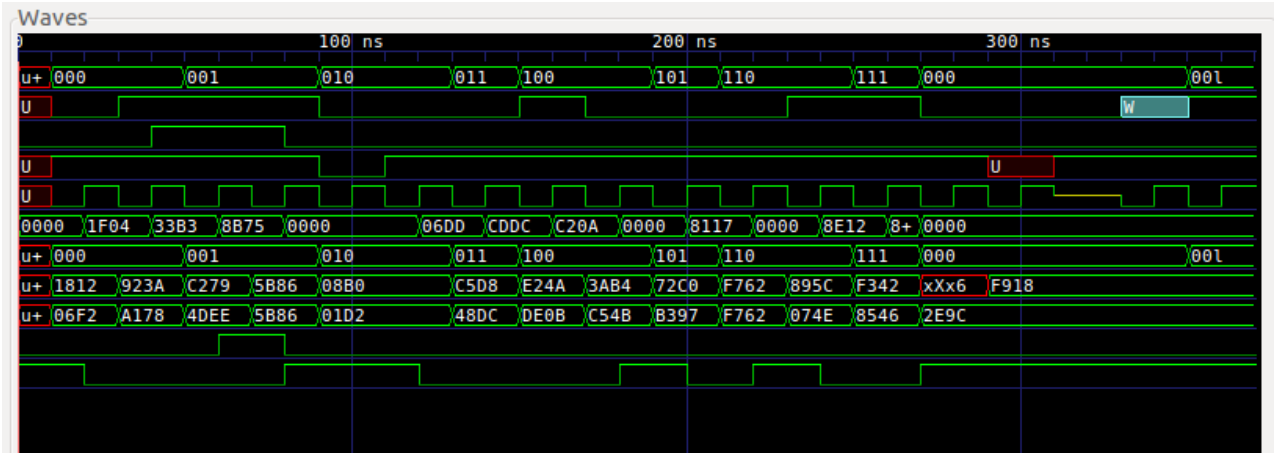

Abbildung 3 GTKWave - Waves

Die angezeigten Signale sind im Fenster **Signals** aufgelistet; sie können vom linken unteren Fenster mit der Maus hineingezogen und per Rechtsklick wieder entfernt werden.

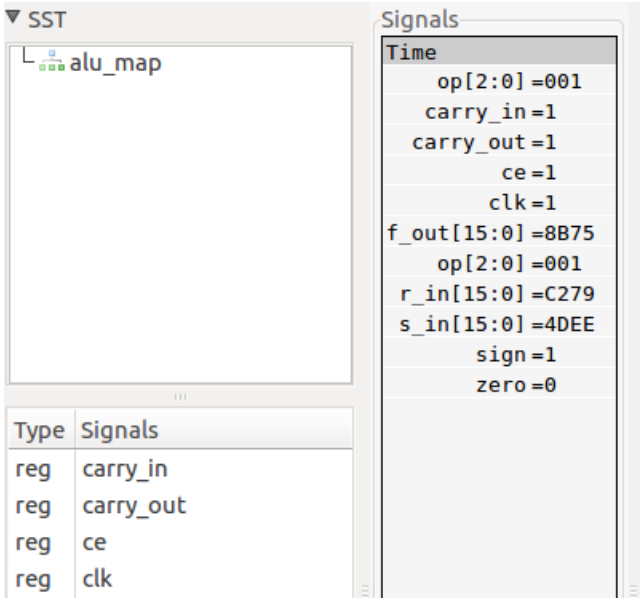

Abbildung 4 GTKWave - Signals

In der Menüleiste links oben kann mit **Zoom Fit** der gesamte verwendete Zeitbereich betrachtet werden. Mit **Zoom In** und **Zoom Out** kann der Bereich verkleinert und vergrößert werden.

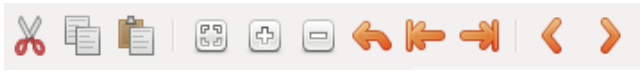

Abbildung 5 GTKWave - Toolbar

Über **File > Write Save File As** kann die aktuelle Konfiguration als `ALU_config.gtkw` gespeichert werden. Sie wird beim nächsten Aufruf von `make gtk` wieder geladen. So kann z.B. der Anwender sich nur die gewünschten Signale im Fenster anzeigen lassen, und diese beim nächsten Laden wieder zur Verfügung haben.

4. Entwicklerdokumentation

4.1. Implementierung

4.1.1. ALU.vhdl

Folgende Standardbibliotheken werden verwendet:

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
```

Zu Beginn werden die Ports der ALU-Entity definiert. Die an den Eingängen R und S anliegenden Datenwörter `r_in` und `s_in` werden mit vorzeichenlosen 16-Bit Vektoren modelliert. Mit der 3-Bit breiten Funktionssteuerung wird die Rechenoperation `op` gewählt. Zusätzlich gibt es das eingehende Carry-Bit `carry_in`, den Basistakt `clk` und das „Clock-Enable“-Bit `ce`, mit dem die ALU aktiviert werden kann.

```
entity ALU_entity is
        port
        (
        r_in: in unsigned (15 downto 0);
        s_in: in unsigned (15 downto 0);
        op: in std_logic_vector (2 downto 0);

        carry_in: in std_logic;
        clk: in std_logic;
        ce: in std_logic;
```

Der F-Ausgang `f_out` muss ebenfalls vorzeichenlos und 16-Bit breit sein. Implementiert wurden hier nur das Carry-Flag `carry_out`, das Vorzeichenflag `sign` und das Zero-Flag `zero`.

```
        f_out: out unsigned (15 downto 0);

        carry_out: out std_logic;
        sign: out std_logic;
        zero: out std_logic
        );
end ALU_entity;
```

Danach wird die Architecture erstellt. Dazu wird zunächst ein internes Hilfssignal `temp` angelegt. Dieses ist 17-Bit breit, um den Übertrag bei der Addition und Subtraktion abzufangen, da dieser sonst verloren gehen würde. Das Signal wird mit 0 initialisiert.

```
architecture ALU_architecture of ALU_entity is
        signal temp: unsigned (16 downto 0) := "00000000000000000";
```

Mit `begin` werden die *concurrent statements* (nebenläufige Anweisungen) eingeleitet. Die Ausgänge werden aus dem Signal `temp` ausgelesen: Das Ergebnis `f_out` der Rechnung liegt in den unteren 16 Bit. Der Carry-Ausgang `carry_out` liegt im höchsten Bit und wird nur bei der Addition und Subtraktion gesetzt (Opcodes $000, 001, 010$). Das Vorzeichen-Flag `sign` wird ebenfalls nur bei den arithmetischen Operationen gesetzt; es liegt im obersten Bit des Ergebnisses, also im zweithöchsten Bit von `temp`. Schließlich wird das Zero-Flag `zero` gesetzt, wenn das Ergebnis gleich 0 ist (unabhängig von `carry_out`).

```
begin
f_out <= temp(15 downto 0);
carry_out <= '1' when ((op = "000" or op = "001" or op = "010")
                        and temp (16) = '1') else '0';
sign <= '1' when ((op = "000" or op = "001" or op = "010")
                        and temp (15) = '1') else '0';
zero <= '1' when temp (15 downto 0) = "0000000000000000" else '0';
```

Nun wird der eigentliche Prozess gestartet. Die Sensitivitätsliste reagiert auf den Takt `clk` und (taktasynchron) auf das CE-Signal `ce`. Ist mindestens einer der beiden gesetzt, wird der Prozess ausgeführt.
Vor der Berechnung werden die Eingangswerte auf Gültigkeit überprüft, d.h. die Eingänge sollen im erwarteten Wertebereich liegen. Die 16-Bit Datenwörter `r_in` und `s_in` können Zahlen von 0 bis $2^{16} - 1 = 65535$ darstellen. Alle 1-Bit Signale sollen entweder 0 oder 1 sein. Diese Einschränkung ist notwendig, da Signale auch nicht-numerische Werte annehmen können, z.B. `U` (nicht initialisiert) oder `X` (unbekannt).

```
process(clk, ce) is
        begin
        if((r_in >= 0 and r_in <= 65535)
                and (s_in >= 0 and s_in <= 65535)
                and (carry_in = '0' or carry_in = '1')
                and (clk = '0' or clk = '1') and (ce = '0' or ce = '1')) then
```

Liegt eine steigende Flanke vor und ist das CE-Signal gesetzt, wird die gewählte Operation `op` abgefragt.

```
if rising_edge(clk) and (ce = '1') then

case op is
```

Bei der Addition muss der eingehende Übertrag `carry_in` auf das Ergebnis gerechnet werden. Die Vektoren `r_in` und `s_in` werden kurzzeitig um jeweils ein oberes Bit erweitert, um den entstehenden Übertrag in `temp` zu speichern. Beispiel:

$$\begin{array}{rr} & \mathbf{0}\ 1000100010001000 \\ + & \mathbf{0}\ 1001000100010001 \\ \hline & \mathbf{1}\ 0001100110011001 \end{array}$$

```
when "000" =>  -- ADD
if(carry_in = '1') then
        temp <= (("0" & r_in) + ("0" & s_in) + 1);
else
        temp <= (("0" & r_in) + ("0" & s_in));
end if;
```

Bei gesetztem `carry_in` soll das Zweierkomplement, ansonsten das Einerkomplement (Subtraktion - 1) realisiert werden. Auch hier werden die Datenwörter auf 17 Bit erweitert und das Ergebnis in `temp` gespeichert. Mit SUBR kann R und mit SUBS kann S als Subtrahend gewählt werden.

```
when "001" =>  -- SUBR => S - R
if(carry_in = '1') then
        temp <= (("0" & s_in) - ("0" & r_in));
else
        temp <= (("0" & s_in) - ("0" & r_in) - 1);
end if;
when "010" =>  -- SUBS => R - S
if(carry_in = '1') then
        temp <= (("0" & r_in) - ("0" & s_in));
else
        temp <= (("0" & r_in) - ("0" & s_in) - 1);
end if;
```

Bei der Umsetzung der logischen Operationen besteht kein wesentlicher Unterschied. Die verfügbaren Operationen sind:

- OR: $R \vee S$
- AND: $R \wedge S$
- NOTRS: $\overline{R} \wedge S$
- EXOR: $R \oplus S$
- EXNOR: $\overline{R \oplus S}$

```
when "011" =>
temp <= (("0" & r_in) or ("0" & s_in));

when "100" =>
temp <= (("0" & r_in) and ("0" & s_in));

when "101" =>
temp <= (("0" & (not r_in)) and ("0" & s_in));

when "110" =>
temp <= (("0" & r_in) xor ("0" & s_in));

when "111" =>
temp <= (("0" & r_in) xnor ("0" & s_in));
```

Wurde keiner der acht Operationen gewählt oder liegt eine ungültige Eingabe vor, so wird `temp` auf 0 gesetzt. Die Zuweisungen im Prozess werden erst mit `end process` ausgeführt.

```
                                when others =>
                                temp <= "0000000000000000";
                                end case;
                        end if;
                else
                        temp <= "0000000000000000";
                end if;
        end process;
end ALU_architecture;
```

4.1.2. ALU_tb.vhdl

Es werden die selben Bibliotheken wie im ALU-Baustein verwendet. Die Testbench-Entity benötigt keine weiteren Ports.

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;

entity ALU_tb is
end ALU_tb;
```

Die oben beschriebene ALU-Entity wird als Komponente deklariert. Die Portdefinitionen werden dabei genau übernommen.

```
architecture ALU_tb_architecture of ALU_tb is
    component ALU_entity
        port
        (
        r_in: in unsigned (15 downto 0);
        s_in: in unsigned (15 downto 0);
        op: in std_logic_vector (2 downto 0);

        carry_in: in std_logic;
        clk: in std_logic;
        ce: in std_logic;

        f_out: out unsigned (15 downto 0);

        carry_out: out std_logic;
        sign: out std_logic;
        zero: out std_logic
        );
    end component;
```

Die ALU-Entity wird instanziiert und deren Ports auf interne Signale mittels `port map` abgebildet. Die Testbench started dann den Hauptprozess und steht zur Verwendung bereit.

```
signal r_in, s_in, f_out: unsigned (15 downto 0);
signal op: std_logic_vector (2 downto 0);
signal carry_in, clk, ce, carry_out, sign, zero: std_logic;

begin
    ALU_map: ALU_entity port map (r_in => r_in, s_in => s_in, f_out => f_out,
                                  op => op, carry_in => carry_in,
                                  clk => clk, ce => ce, carry_out => carry_out,
                                  sign => sign, zero => zero);
    process begin
```

Ab dieser Stelle folgt eine Reihe von Testfällen, die alle im folgenden Format erzeugt werden:
Zuerst werden die Datenwörter, die Operation, der Carry-Eingang, der Takt und das CE-Signal auf die gewünschte Eingabe gesetzt. Nach einer halben Taktperiode wird mit `assert` geprüft, ob die Ausgabe dem erwarteten Ergebnis entspricht. Falls nicht, wird eine informative Fehlermeldung gedruckt.

```
r_in <= "1001001000111010";      -- R <= 37434
    s_in <= "1010000101111000";      -- S <= 41336
    op <= "000";                     -- Addition

    carry_in <= '1';
    clk <= '1';
    ce <= '1';

    wait for 10 ns;

    assert (f_out = "0011001110110011" and carry_out = '1'
            and sign = '0' and zero = '0')
    report "Something went wrong at the 5th assertion";
```

Der Takt `clk` muss dabei nach jeder halben Periode (10 ns) gewechselt werden, um den 50 MHz Takt zu simulieren. ($\frac{1}{50 \cdot 10^6 \text{Hz}} = 20 \cdot 10^{-9}$s)
Zum Schluss werden, im Falle nicht bestandener Tests, Fehlermeldungen (`assertion error`) ausgegeben. Außerdem erzeugen undefinierte Eingaben Warnungen (`assertion warning`). Es wird ein letztes mal `wait` aufgerufen und der Prozess beendet.

```
assert false report "Reached end the of the testbench." & lf
& "In case exactly 3 assertion warnings" & lf
& "(caused by setting entry values to 'X' and 'U' in the final test cases)" & lf
& "and 0 other assertion errors are displayed, all computations were correct."
severity note;
    wait;    -- Final wait command to assure the testbench stops
    end process;
end ALU_tb_architecture;
```

4.2. Bekannte Probleme und Erweiterungen

Im Folgenden werden Verbesserungsvorschläge und mögliche Erweiterungen der Implementierung vorgestellt.
Der Quellcode der Testbench kann verkürzt werden, indem die Testfälle durch Schleifen fast vollständig automatisiert werden. Dazu würde in einer äußeren Schleife der R-Wert, in der inneren der S-Wert fortlaufend inkrementiert werden, um alle Werte innerhalb eines Bereiches abzudecken.
Die Ausgabe am Ende der Testbench kann übersichtlicher gestaltet werden, indem die assertion warnings bei ungültigen Eingaben unterdrückt werden (da diese ja keinen Implementierungsfehler andeuten). Dies kann mittels `-ieee-asserts=disable` beim Erstellen der Testbench erreicht werden.
Über ein Rahmenprogramm können Tests dynamisch in die Testbenchdatei geschrieben und ausgeführt werden, damit ein Entwickler nicht immer den Code von Hand verändern muss. Dazu würde das Programm die gewünschten Eingaben abfragen, diese dann im korrekten Format (siehe 4.1.2) nach `ALU_tb.vhdl` schreiben und das Makefile aufrufen.