ALU des Rechenwerkbausteins Am2901

Spezifikation VHDL

Praktikum: Einführung in die Rechnerarchitektur
Sommersemester 2018
Gruppe 33

Autoren	Ruben Bachmann
	Franziska Steinle
	Roland Würsching
Eingereicht am	13.05.2018

Inhaltsverzeichnis

1. Aufgabenverteilung

Projektleiter: Franziska Steinle

Dokumentation: Roland Würsching

Abschlussvortrag: Ruben Bachmann

2. Aufgabenbeschreibung

Ziel dieser Aufgabe ist es, die Arithmetisch-Logische Einheit (ALU) des mikroprogrammierbaren Rechenwerks Am2901 in VHDL zu entwickeln. Als grundlegender Rahmen dient die Beschreibung der MI-Maschine:
http://wwwi10.lrr.in.tum.de/ eti/Vorlesung/WS1718/Informationsmaterial/mimaschine.pdf

Folgende Änderungen und Erweiterungen des Am2901 wurden festgelegt:

1. Der eigentliche Rechenwerkbaustein, der aus vier 4 Bit Komponenten besteht, wird zu einem 16 Bit Baustein zusammengefasst
2. Der Basistakt beträgt 50 MHz.
3. Ein Effektivtakt (4 Basistakte) umfasst das Laden der Daten aus den Registern, die Rechenoperation und das Speichern in die Register.
4. 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.
5. Jeder Baustein muss seine Funktion innerhalb eines Basistaktes erfüllen können.
6. Ein Steuerungsbaustein generiert aus dem Basistakt die CE Signale für die anderen Bausteine.
7. Die Registerbank (16 Register + Q-Register) umfasst zwei CE Eingänge:
   - einen CE_read: Daten aus Registern lesen
   - einen CE_write: Daten in Register speichern

   Beide CE-Eingänge können nicht gleichzeitig oder im direkt folgenden Takt nacheinander gesetzt werden (min. ein Takt Pause).
8. Die Bausteine bieten falls sinnvoll einen synchronen Reset an.

2.1. Ist-Analyse

Die Quelloperandenauswahl erhält Daten aus den folgenden Quellen:

- Extern (D)
- 16 Register, adressiert über die Adressfelder A und B
- Null
- Q-Register

Mittels der Quelloperandensteuerung (Bits I_0 bis I_2 der ALU-Steuerung) werden zwei Quellen

ausgewählt und deren Werte über die Ausgänge R und S weitergeleitet.

2.2. Soll-Analyse

Die von der Quelloperandenauswahl ausgegebenen Daten liegen an den Eingängen R und S der ALU an. Außerdem besitzt die ALU einen Carry-Eingang (C_n) und einen CE-Eingang. Die Funktionssteuerung (Bits I_3 bis I_5) 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})

2.3. Werkzeug und Hilfsmittel

- 64-Bit Ubuntu 16.04
- GHDL (Implementierung)
- GtkWave (Visualisierung, Testen)

2.4. Bemerkungen

Für den Nachweis einer erfolgreichen Implementierung wird eine VHDL-Testbench entwickelt. Außerdem wird zum Kompilieren ein Makefile bereitgestellt.

3. Lösungsansätze

Der ALU-Baustein 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.

3.1. Lösungsansatz A

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

Der Prozess soll nur auf den Takt `clk` reagieren. Zu Beginn wird auf eine steigende Flanke geprüft. Falls zusätzlich das CE-Signal gesetzt ist, 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.
SUBR: S - R
Das Wort `r_in` wird von `s_in` abgezogen und in `temp` gespeichert. Um die Längen anzugleichen müssen `r_in` und `s_in` um ein höchstes 0-Bit erweitert werden. Ist `carry_in` nicht

gesetzt, wird zusätzlich 1 abgezogen.
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.

3.2. Lösungsansatz B

Folgende (voneinander unabhängige) Änderungen können an Ansatz A vorgenommen werden:

3.2.1. Bestimmung des Carry-Ausgangs

Anstelle des erweiterten Vektors `temp` kann der Carry-Ausgang auch „manuell“ ermittelt werden.
Bei ADD werden die höchsten Bits der Eingänge und des Ergebnisses (r(15), s(15), f(15)) betrachtet. Dann wird `carry_out` in genau den folgenden Fällen gesetzt:

- r(15) = s(15) = 1
- (r(15) or s(15)) and (not f(15))

Bei SUBR wird `carry_out` gesetzt, falls `r_in` $>$ `s_in`.
Bei SUBS wird `carry_out` gesetzt, falls `s_in` $>$ `r_in`.

3.2.2. Subtraktion

Die Subtraktion kann alternativ als Addition mit dem Einer- oder Zweierkomplement formuliert werden. Dazu wird z.B. für R – S
`r_in` + (not `s_in`) + `carry_in`
gerechnet. Dabei muss beachtet werden, dass s_in mit einem 1-Bit erweitert wird, um das Komplement richtig darzustellen.

3.2.3. Datentyp der Eingänge

Die Eingangswerte `r_in` und `s_in` können als *std_logic_vector* statt *unsigned* deklariert werden. Bei den arithmetischen Operationen ADD, SUBR und SUBS müssen sie dann als *unsigned* angegeben werden.

3.3. Bewertung der Ansätze

Aufgrund der eindeutigen Vorgaben der mikroprogrammierbaren ALU ist der Spielraum für grundlegende Änderungen relativ gering.
Die alternative Bestimmung des Carry-Ausgangs spart zwar ein Signal ein, ist aber im Vergleich zu Ansatz A zu umständlich.
Die Subtraktion in Ansatz B macht zwar die Bildung des korrekten Komplements deutlich, kann aber weniger fehleranfällig mit dem Minus-Operator umgesetzt werden.
Die Eingangswerte als *std_logic_vector* sind zwar allgemeiner und flexibler, müssen jedoch für die arithmetischen Operationen auf *unsigned* konvertiert werden.

Der erste Lösungsansatz erfüllt die Aufgaben der ALU gemäß der Beschreibung und kann übersichtlich und leicht verständlich implementiert werden. Deshalb wird beschlossen, Ansatz A umzusetzen.

4. Zeitplanung

Termin	Aufgabe
18.04.	Erstes Treffen zur Klärung grundsätzlicher Fragen zur Aufgabenstellung. Festlegung der Rollen. Planung des weiteren Vorgehens. (ca. 1 Stunde)
18.04. - 02.05.	Selbstständiges informieren zu den wichtigsten Punkten der Aufgabenstellung. Gedanken zu möglichen Lösungsansätzen machen. (ca. 5 Stunden)
02.05.	Treffen zum Zusammentragen der in der vergangenen Woche selbstständig erarbeiteten Informationen. Beginn der Ausarbeitung der Spezifikation. (ca. 1,5 Stunden)
02.05. - 09.05.	Fertigstellung der Spezifikation. (ca. 10 – 15 Stunden)
09.05.	Treffen zum Besprechen letzter Unklarheiten bezüglich der Spezifikation. Planung der Implementierungsphase. (ca. 1,5 Stunden)
11.05. - 13.05.	Hochladen der finalen Version der Spezifikation in das SVN.
13.05. - 30.05.	Beginn der Implementierung. Hierbei übernimmt jedes Gruppenmitglied die Aufgaben, welche ihm beim Treffen am 09.05.2018 zugeteilt wurden. (ca. 20 – 30 Stunden)
30.05.	Treffen zum Austausch über bisherigen Verlauf der Implementierung. Diskussion von eventuell aufgetretenen Problemen. (ca. 1,5 Stunden)
30.05. - 13.06.	Fortsetzen der Implementierung. (ca. 20 – 30 Stunden)
13.06.	Treffen zum Austausch über bisherigen Verlauf der Implementierung. Ausarbeiten der Tests. (ca. 1,5 Stunden)
06.06. - 13.06.	Fertigstellung der Implementierung. Schreiben und testen der Tests. (ca. 10 Stunden)
13.06.	Treffen zum Besprechen letzter Unklarheiten bezüglich der Implementierung, sowie Planung der Ausarbeitungsphase. (ca. 1,5 Stunden)
15.06. - 17.06.	Hochladen der finalen Version der Implementierung in das SVN.
17.06. - 27.06.	Beginn der Ausarbeitung. Hierbei übernimmt jedes Gruppenmitglied die ihm/ihr zugeteilten Aufgaben. (ca. 10 – 15 Stunden)

27.06.	Treffen zum Austausch über bisherigen Verlauf der Ausarbeitung (schreiben der Dokumentationen). Diskussion von eventuell aufgetretenen Problemen. => in Kombination mit Assembler Projekt (ca. 1,5 Stunden)
27.06. - 06.07.	Fertigstellung der Ausarbeitung. Überprüfung der Dokumentationen auf Verständlichkeit. Planung der Vortragsphase. (ca. 15 – 20 Stunden)
06.07. - 08.07.	Hochladen der finalen Version der Dokumentationen bzw. des Lösungsansatzes in das SVN.
08.07. - Vortrag	Gruppenmitglieder bereiten ihren jeweiligen Teil der Präsentation vor und halten sich gegenseitig auf dem Laufenden (ca. 20 – 30 Stunden)