Einführung in die Rechnerarchitektur Praktikum

SPEZIFIKATION

Projekt:

VHDL : AM2901

Projektleiter	Martin Zinnecker
Dokumentation	Sandra Grujovic
Formaler Vortrag	Ivan Chimeno

1 Aufgabenkurzbeschreibung

Für die Registerbank gelten folgende Rahmenbedingungen: es gibt 3 Eingänge für die RAM, bestehend aus dem Dateneingang, der A-Adresse und der B-Adresse. Zusätzlich gehören dazu auch die Eingänge des Q Registers, Eingang F und Q‘. Die vorhandenen Ausgänge sind der Datenausgang A und der Datenausgang B, und der Ausgang Q gehört zum Q Register. Dieses Modul ist getaktet.

Das ALU-Quelloperandenauswahl Modul hingegen besitzt die Eingänge D, A, B, 0 (welches eine 0 enthält) und Q. Die zugehörigen Ausgänge sind R und S. Dieses Modul ist ungetaktet. Dies liegt primär daran, dass man bei physischen Schaltungen nicht eindeutig davon ausgehen kann, dass beide Eingänge zum richtigen Zeitpunkt zu Beginn des Taktes vorliegen, weil grundsätzlich Verzögerungen möglich sind. Dies würde die Funktionsweise beeinträchtigen, deswegen muss das ALU-Quelloperanden Modul ungetaktet sein.

Darüber hinaus gibt es zusätzliche Vorgaben für beide Module. Jeder Baustein muss seine Funktion innerhalb eines Basistakts á 50 MHz erfüllen können. Sowohl die Registerbank als auch die ALU-Quelloperandenauswahl werden mit einem oder mehreren CE Signalen versorgt. Sobald dieses Signal gesetzt ist, ist das Modul aktiv. Die Registerbank besitzt zusätzlich zu diesem CE Signal auch noch die CE_Read und CE_Write Signale, die, wenn sie gesetzt sind, festlegen, ob das Modul einen Wert aus einem Register lesen soll oder einen Wert in ein Register schreiben soll. Beide Signale dürfen nicht gleichzeitig oder im nachfolgenden Takt nacheinander belegt werden. Laden, die eigentliche Berechnung und das Speichern in die Register dauert zusammen einen Effektivtakt, welcher 4 Basistakten entspricht. Allerdings darf das CE Signal nicht zur Flankensteuerung eingesetzt werden. Außerdem bietet die Registerbank einen synchronen Reset an, falls notwendig.

Alle Eingänge und Ausgänge sind 16 Bit lang, da der kaskadierbare AM2901 in der MI-Maschine vierfach vorhanden ist. Die zu implementierende ALU-Quelloperandenauswahl hingegen ist anders aufgebaut – Je nachdem, welche Eingänge ausgewählt werden, werden diese beiden auf die Ausgänge R und S gelegt. Mithilfe des Instruktionsbits I_0 bis I_2 wird genau spezifiziert, welche dieser Eingänge auf den R, bzw. S Ausgang gelegt werden. Zusätzlich gibt es hier den Eingang D, der Daten aus externen Quellen einlesen kann, und das 0-Register, welches lediglich eine Null enthält und diese weiterleitet, falls ausgewählt. Die Eingänge A, B, und Q kommen allesamt über die Registerbank.

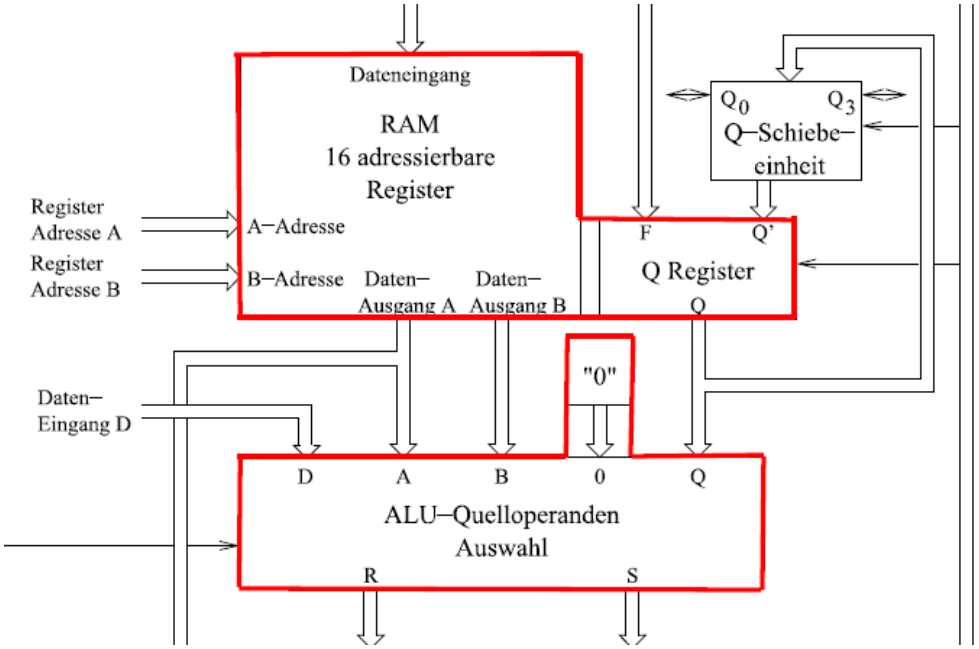

Abbildung 1: Ausschnitt der zu implementierenden Module

2 Lösungsansätze

2.1 Lösungsansatz A

Für den Lösungsansatz A wird die Registerbank als ein gemeinsames Modul implementiert. Dies bedeutet, dass sowohl das Q-Register, als auch die RAM (16-addressierbare Register) in einem Modul realisiert werden. Im Folgenden ist die Entity deklariert, die die entsprechenden Eingänge und Ausgänge definiert.

```
entity registerbank is
port(
--Eingänge
clk: in std_logic; -- clk signal á 50 MHz
d: in std_logic_vector(15 downto 0); -- Dateneingang
a_address: in std_logic_vector(15 downto 0); -- Eingang A-Adresse
b_address: in std_logic_vector(15 downto 0); --Eingang B-Adresse
q: in std_logic_vector(15 downto 0); -- Q Dateneinang für QReg
--Ausgänge
a_out: out std_logic_vector(15 downto 0); --Ausgang A
b_out: out std_logic_vector(15 downto 0); --Ausgang B
q_out: out std_logic_vector(15 downto 0) --Ausgang für QReg
);
end registerbank;
```

Die 16 Register und Q-Register werden hierbei mithilfe eines Signal-Arrays (mit Indizes 0-16) modelliert, der anfangs noch unbelegt ist. Wenn ein Wert gespeichert werden muss, wird er somit in die i-te Stelle des Arrays gesetzt, wobei i die Nummer des Registers darstellt. Hierbei muss grundsätzlich zwischen den beiden Operationen ‚Schreiben' und ‚Lesen' unterschieden werden. Man nehme an, dass das entsprechende CE Signal in diesem Falle gesetzt ist, sodass das richtige Modul aktiv ist.

Das CE_read Signal ist gesetzt: Beim Lesen eines Wertes wird die **a_address** und die **b_address** genauer betrachtet - Je nachdem, welche Register diese addressieren, werden die entsprechenden Register aus unserer modellierten Speicherbank ausgewählt. (Beispiel: sei **a_address** das binär kodierte Register 5, durch diese Spezifikation wissen wir, dass wir den Wert des deklarierten Signal-Arrays an Stelle 5 auf unseren Ausgang **a_out** legen müssen. Dasselbe Prinzip gilt analog auch für **b_address** und **b_out**.) Das Q-Register wird in diesem Lösungsansatz als zusätzliches Register im Signal Array geführt, an Index 16. Soll also aus diesem Register gelesen werden, wird der Wert an Index 16 der Wert herausgenommen und auf **q_out** gelegt. Die Indexe 0 bis 15 sind bereits für das RAM reserviert. Aus den Indexen 0 bis 15 darf nur gelesen werden, falls das CE Signal des RAM Moduls gesetzt ist, aus dem Index 16 (also dem Q Register) darf nur gelesen werden, wenn das CE Signal des Q-Registers gesetzt ist.

Das Schreiben eines Wertes funktioniert ähnlich. Hier wiederum ist das CE_write Signal gesetzt, und signalisiert uns den Schreibvorgang. Falls das CE Signal das RAM gesetzt ist, wird nun das Ergebnis, welches aus dem Dateneingang d kommt, in den Index unseres Signal-Arrays gespeichert, welcher durch den in **b_address** spezifizierte, binär kodierten Wert gegeben ist. Analog zum Lesevorgang ist Schreiben an die Indexe 0 bis 15 unseres internen Signal Arrays nur erlaubt, wenn das RAM CE Signal gesetzt ist, Index 16 ist beschreibbar, wenn das CE Signal des Q-Registers gesetzt ist. Hierbei wird der **q** Eingang in das Register am Index 16 geschrieben.

Das ganze Modul wird mit einem **clk** Eingang genährt, und die atomaren Operation Lesen bzw Schreiben sind in einem Takt in dieser Implementierung durchfürbar. Der synchrone Reset würde sich genauso an diesem Takt orientieren.

2.2 Lösungsansatz B

Im zweiten Lösungsansatz für die Speichereinheit der MI Maschine liegt vor allem der Aspekt der Trennung im Vordergrund. Hierbei wird das das RAM Modul mit seinen 16 adressierbaren Registern und das Q-Register parallel betrachtet und als autonome, einzelne VHDL Module implementiert. Die Entity des RAM Moduls sieht wie folgt aus:

```
entity RAM is
port(
--Eingänge
clk: in std_logic; -- clk signal á 50 MHz
d: in std_logic_vector(15 downto 0); -- Dateneingang
a_address: in std_logic_vector(15 downto 0); -- Eingang A-Adresse
b_address: in std_logic_vector(15 downto 0); --Eingang B-Adresse
--Ausgänge
a_out: out std_logic_vector(15 downto 0); --Ausgang A
b_out: out std_logic_vector(15 downto 0); --Ausgang B
);
end RAM;
```

Ähnlich wie im Lösungsansatz A wird der eigentliche Speichervorgang mithilfe eines Signal-Arrays implementiert. Allerdings besitzt in dieser Implementierung das RAM Modul 16 Register, was bedeutet, dass unser Signal-Array eine Länge von 16 Stellen hat. (0-15) Hierbei gibt es keine Restriktionen hinsichtlich der Zugriffsberechtigung der einzelnen Stellen des Arrays. Ist das CE Signal des RAM Moduls, gesetzt, so sind alle 16 Stellen des Arrays veränderbar und addressierbar. Im Gegensatz zum Lösungsansatz A muss hier also nicht kompliziert geprüft werden, ob das korrekte CE Signal von einem der beiden Module gesetzt worden ist, sondern es wird einfach nur auf die vorhandenen modellierten Speicherzellen zugegriffen. Sowohl für den Schreib- als auch Lesevorgang muss das CE Signal gesetzt werden. (dies gilt für die RAM Entity, als auch die Q-Register Entity.)

Die Register der RAM wie oben genannt, durch einen Signal Array realisiert, der 16 Stellen besitzt. Wird das CE Write Signal gesetzt, wird das über **d** Eingang eintretende Datenwort in die Stelle des Signal-Arrays gespeichert, welches im Eingang **b_address** spezifiziert ist. Liefert uns als z.B **b_address** das binär kodierte ‚0001' (Bit $_0$- Bit $_3$), also 1, so wissen wir, das wir unser Datenwort an signal_array[1] schreiben müssen.

Wird das CE Read Signal gesetzt, müssen die korrekten Stellen des Signal-Arrays ausgewählt und auf den Ausgang **a_out** und **b_out** gelegt werden. Dies funktioniert ähnlich wie der obige erklärte Schreibvorgang. Die Eingänge **a_address** und **b_address** spezifizieren genau durch die Bit 0-3, welches Register ausgewählt wird, und entsprechend dieser binären Codierung müssen die richtigen Stellen aus dem Signal-Array gelesen und auf die Ausgänge **a_out** und **b_out** geleitet werden.

Im Grunde genommen funktioniert hier die Speicherlogik genauso wie im Lösungsansatz A, mit dem wichtigen Unterschied, das nicht mehr geprüft werden muss, ob die Zugriffsrechte vorhanden sind um in eine ganz bestimmte Stelle des Signal Arrays speichern bzw. aus ihm laden zu können.

Diese Modul besitzt einen synchronen Reset, beide Module (RAM, Q Reg) funktionieren mit einem **clk** Eingang á 50 MHz.

Analog dazu folgt nun die Entity-Beschreibung unseres Q-Register Bausteins.

```
entity qreg is
port(
--Eingänge
clk: in std_logic; -- clk signal á 50 MHz
f: in std_logic_vector(15 downto 0); -- Dateneingang für QReg
q: in std_logic_vector(15 downto 0); -- Q' Dateneinang für QReg
--Ausgänge
q_out: out std_logic_vector(15 downto 0) --Ausgang für QReg
);
end qreg;
```

In diesem Lösungsansatz wird der Speicher des Q-Registers durch ein einzelnes Signal, nennen wir es *q_speicherzelle* innerhalb des Q-Registers modelliert. Falls das CE Signal gesetzt ist, kann auf dieses Modul zugegriffen werden. Ist das CE_Write Signal gesetzt, wird das Datenwort, welches über den Eingang **q** in das Modul eintritt, in *q_speicherzelle* geschrieben. Falls das CE_Read Signal gesetzt ist, wird der entsprechende, an *q_speicherzelle* stehende Wert ausgegeben, und über den **q_out** Ausgang weiter an die ALU-Quelloperandenauswahl geleitet.

2.3 Gemeinsame Lösung für ALU-Quelloperanden Auswahl

Die ALU-Quelloperandenauswahl wird wie folgt in eine VHDL Entity übersetzt:

```
entity alu_operandenauswahl is
port(
--Eingänge
i: in std_logic_vector(8 downto 0); -- übermitteltes Instruktionswort
d: in std_logic_vector(15 downto 0); -- Dateneingang D
a in std_logic_vector(15 downto 0); -- Dateneingang A
b in std_logic_vector(15 downto 0); -- Dateneingang B
q: in std_logic_vector(15 downto 0); -- Dateneingang für Q
zero: in std_logic_vector(15 downto 0); -- Dateineingang 0
--Ausgänge
r_out: out std_logic_vector(15 downto 0); --Ausgang R
s_out: out std_logic_vector(15 downto 0) --Ausgang S
);
end alu_operandenauswahl;
```

Dieses Modul wird als einzelnes Modul aufgefasst und implementiert. Hierbei empfangen wir durch den **i** Eingang das übermittelte Instruktionswort, welches uns genau spezifiziert, welche beiden der Eingänge **d, a, b, zero, q** ausgewählt werden. (**zero** wird hierbei als ein zusätzliches Signal realisiert, welches immer ‚0' ist). Dies lässt sich an den Bits 0 bis 2 von i ablesen. Die zwei spezifizierten Eingänge werden dann auf die Ausgänge **r_out** bzw. **s_out** geleitet. Im Folgenden ist eine tabellarische Darstellung aller Funktionen, die die ALU-Quelloperanden Auswahl erfüllen muss.

I_0	I_1	I_2	r_out	s_out	Beschreibung
0	0	0	a	q	**a** wird auf **r_out** gelegt, **q** wird auf **s_out** gelegt
0	0	1	a	b	**a** wird auf **r_out** gelegt, **b** wird auf **s_out** gelegt
0	1	0	zero	q	**zero** wird auf **r_out** gelegt, **q** wird auf **s_out** gelegt
0	1	1	zero	b	**zero** wird auf **r_out** gelegt, **b** wird auf **s_out** gelegt
1	0	0	zero	a	**zero** wird auf **r_out** gelegt, **a** wird auf **s_out** gelegt
1	0	1	d	a	**d** wird auf **r_out** gelegt, **a** wird auf **s_out** gelegt
1	1	0	d	q	**d** wird auf **r_out** gelegt, **q** wird auf **s_out** gelegt
1	1	1	d	zero	**d** wird auf **r_out** gelegt, **zero** wird auf **s_out** gelegt

Dieses Modul besteht, recht plakativ erläutert, aus einem einzigen switch-case Statement. Bei der Aktivierung des Moduls durch das gesetzte CE-Signal wird der Eingang **i** genauer betrachtet: Die Bit 0 bis 2 sind für die Funktionalität interessant, da sie genau spezifizieren, wie die Selektion der Operanden codiert wird und funktioniert. Wären die Bit 0-2 des Instruktionswortes also zum Beispiel ‚001', so würden der Eingang **a** und der Eingang **q** auf den Ausgang **r_out** und **s_out** gelegt werden. Dasselbe gilt für die restlichen sieben möglichen Kombinationen I_0-I_2. Somit ist das gesamte Modul durch ein entsprechend komplexes Switch-Case Statement prüfbar, und hieraus ergibt sich auch die einzige Möglichkeit, es zu implementieren. Darüber hinaus ist in der Aufgabenstellung explizit gegeben, dass das Modul ungetaktet werden muss (weiter oben in der Aufgabenbeschreibung befindet sich die Begründung warum dies so sein muss). Somit kann dieses Modul nicht ungetaktet umgesetzt werden. Aus diesen Gründen folgt die Entscheidung, das ALU-Quelloperanden Modul sowohl in Lösungsansatz A als auch in Lösungsansatz B auf dieses Art und Weise zu implementieren.

3 Bewertung der Lösungsansätze

Lösungsansatz A		**Lösungsansatz B**	
Vorteile	Nachteile	Vorteile	Nachteile
- Zugriffsrechte müssen genauestens geprüft werden bevor auf eine Speicherzelle zugegriffen werden kann - Hohe Kohäsion - Speicherbank in einem Signal Array modelliert	- schwieriger zu Implementieren - unübersichtlicher - Tests müssen ausführlicher geschrieben werden - Flexibilität eingeschränkt	- Geringere Kopplung und Hohe Kohäsion - bessere Testbarkeit - Sauberer und getrennter Code - Keine unnötigen zusätzlichen Zugriffsrechte Checks bei Zugriffen auf den Signal Array - nicht überladen - Erweiterbarkeit des Codes leichter, da Module getrennt	- Speichereinheit ist getrennt, 2 Module statt einem großen - für die zwei Module müssen mehr unabhängige Tests geschrieben werden -> höhere Anzahl von Tests

4 Entscheidungswahl

Im Folgenden wird erläutert, weshalb für eine der beiden Lösungen entschieden wurde. Der Lösungsansatz B erscheint vielversprechender, da er einige signifikante Vorteile mit sich bringt. Vor allem die Zweiteilung des RAM-Moduls und des Q-Registers bringt Vorteile, die besser erscheinen, als die Implementierung als einzelnes Modul für die Speichereinheit.

Zum einen ist eine Trennung der Module code- und aufbautechnisch simpler und effizienter – Bei dem Zugriffen auf die modellierten Register des Moduls muss nicht zusätzlich untersucht werden, ob die entsprechenden Signale gesetzt werden, damit man auf die richtigen Array-Slots zugreifen kann. Auch wirkt der Code durch die zwei verschiedenen Entitäten ordentlicher, außerdem ist die Implementierung so aufgeteilt, wird also nicht übermäßig lang und überladen wie im Lösungsansatz A. Auch ist es einfacher und flexibler die Module nach Bedarf zu erweitern. Nach der Implementierung bringt die Aufteilung der Speichereinheit den Vorteil, dass Tests leichter zu schreiben und zu überprüfen sind, was zusätzlichen Aufwand minimiert. Generell erscheint dieser Ansatz persönlich effektiver, da wir eine geringere Kopplung aber eine höhere Kohäsion erreichen würden. Aus diesen Gründen haben wir uns dafür entschieden, den Lösungsansatz B für unsere Implementierung zu realisieren.