Einführung in die Rechnerarchitektur Praktikum

ENTWICKLERDOKUMENTATION

Projekt:

VHDL - AM2901

Projektleiter	Martin Zinnecker
Dokumentation	Sandra Grujovic
Formaler Vortrag	Ivan Chimeno

1 Einleitung

Mit diesem Programm werden das Speicherwerk des AM2901, bestehend aus Q-Register und den 16 adressierbaren Registern (RAM), und die ALU-Quelloperandenauswahl mithilfe von VHDL simuliert. Ziel ist die Modularisierung dieser Elemente des AM2901. Es besteht aus 3 Einzelprogrammen, welche jeweils das Q-Register, die RAM und die ALU Quelloperandenauswahl realisieren.

2 Beschreibung der Programme

2.1 RAM – Funktionsweise

Das RAM-Programm realisiert die 16 adressierbaren Register des AM2901. Zu den Grundfunktionen dieses Moduls gehören das Schreiben eines Wertes in die Speichereinheit, und das Lesen daraus. Die Funktionsweise der Registereinheit wird mithilfe eines Signal Arrays realisiert, welcher intern als `mem` deklariert ist. Wird also in das Register 4 geschrieben, wird an `mem(4)` das entsprechende Datenwort eingelagert. Entsprechend werden die Datenworte, die in dem `mem`-Array gespeichert werden, ausgegeben, wenn der Lesevorgang eingeleitet wird. Dieses Programm wird primär dazu benutzt, die Speicherstruktur des AM2901 mithilfe einer anderen Sprache zu realisieren, um sie leichter verständlich und später synthetisierbar zu machen, um (vielleicht) mithilfe der VHDL Realisierung einen Bau der entsprechenden Schaltungen möglich zu machen.

Vorhandene Eingänge und ihre Bedeutung

Zu den vorhandenen Eingängen des Moduls gehören die folgenden:

- `d : 16 Bit langer std_logic_vector`
  Dieser Eingang beschreibt unser eigentliches Datenwort, welches wir in die Register schreiben möchten. Bei einem Lesevorgang ist das Datenwort zu vernachlässigen. Wichtig ist hierbei, dass es sich um ein 16-Bit langen std_logic_vector handeln muss.
- `a_address : 16 Bit langer std_logic_vector`
  Dies ist das binär kodierte Register, welches die Adresse enthält, an der man das gesuchte gespeicherte Wort findet. Hierbei ist zu beachten, dass die niederwertigsten 4 Bit der a_address binär kodiert die richtige Speicherzelle addressiert. Beispiel: wir möchten das a_address das Register 5 adressiert, das heißt

unsere niederwertigsten 4 Bit sind 0101. Der Rest der Bits der a_address ist wieder zu vernachlässigen und in diesem Programm nicht relevant.

- `b_address : 16 Bit langer std_logic_vector`
  Dies ist das binär kodierte Register, welches die Adresse enthält, an der man das gesuchte gespeicherte Wort findet. Hierbei ist zu beachten, dass die niederwertigsten 4 Bit der b_address binär kodiert die richtige Speicherzelle adressiert. Beispiel: wir möchten das b_address das Register 5 adressiert, das heißt unsere niederwertigsten 4 Bit sind 0101. Der Rest der Bits der b_address ist wieder zu vernachlässigen und in diesem Programm nicht relevant.
- `ce_write & ce_read : std_logic`
  Wenn ce_write/ce_read gesetzt ist, wird der Schreib/Lesevorgang aktiviert. Ist ce_write/ce_read nicht gesetzt, werden die Module und ihre eigentliche Funktionalität nicht aktiviert.
  Darüber hinaus besitzt das Modul auch einen `rst` Eingang, der, wenn gesetzt, alle Register auf undefined zurücksetzt. Außerdem ist dieses Modul getaktet, folglich besitzt es einen `clk` Eingang, der eine Periode von 20 ns besitzt. Diese Frequenz war der Aufgabenstellung zu entnehmen. Jedwede Art von Vorgang wird nur bei `rising_edge(clk)` ausgeführt.

Vorhandene Ausgänge und ihre Bedeutung

Zu den vorhandenen Ausgängen des Moduls gehören die folgenden:

- `a_out & b_out : 16 Bit langer std_logic_vector`
  Diese beiden Ausgänge geben nach einem Lesevorgang die Inhalte des Registers an den Adressen a_address und b_address aus. Wenn der Schreibvorgang eingeleitet wird, gibt b_out das gespeicherte Datenwort d aus, während a_out mit einem default Wert gefüllt wird, da es bei einem Schreibvorgang keine konkrete Bedeutung besitzt.

Vorhandene interne Signale und ihre Bedeutung

Zu den auftretenden internen Signalen des Moduls gehört:

- `signal mem : array von 16 std_logic_vector(15 downto 0)`
  Dieses Signal Array simuliert die 16 adressierbaren Register des RAM. Wenn der Array leer ist, sind alle Werte auf undefined gesetzt. Bei einem Schreibvorgang werden die entsprechenden Stellen des Arrays gefüllt, bei einem Lesevorgang wird

aus den Stellen des Speicherarrays gelesen. Ein Reset Vorgang setzt den mem Array in seinen unbeschriebenen Anfangszustand zurück. Es kann natürlich auch über existierende Werte im Array geschrieben werden. Um auf die korrekte Stelle des Arrays mithilfe der a_address und b_address zugreifen zu können, werden diese zu einem Unsigned Integer gecasted und im Arrayzugriff verwendet.

Was ist zu beachten?

- ce_write und ce_read können nicht gleichzeitig gesetzt werden, auch nicht direkt aufeinanderfolgend im nächsten Takt. Es muss ein Takt Pause dazwischen sein.
- Es ist darauf hinzuweisen, dass beim Lesevorgang tatsächlich etwas an den a_address und b_address spezifizierten Registern steht. Ein Tipp wäre, das Speicherarray erst mit mindestens zwei Werten zu beschreiben bevor man einen Lesevorgang einleitet.
- Das CLK Signal ist momentan auf 20ns gesetzt, kann aber beliebig verändert werden.
- Wie immer gilt es, auf die Datentypen und –längen der einzelnen Eingänge zu achten, bevor man das Modul benutzt.

Hinweise zum Einsatz

Funktionsweise – Schreiben

Der Schreibvorhang des RAM-Programm Moduls funktioniert wie folgt:

- Lege das zu speichernde Datenwort an den Eingang d.
- Spezifiziere b_address und a_address, achte hierbei auf die Richtlinien, die in Vorhandene Eingänge und ihre Bedeutung genannt worden sind
- Setze ce_write = 1
- Nun wird das Datenwort in das Register in b_address gespeichert. (z.B sei b_address das binär kodierte 4, dann wird das Datenwort in mem(4) geschrieben.
- In b_out wird nun das zuvor gespeicherte Datenwort ausgegeben, a_out besitzt einen default Wert.
- Setze ce_write wieder auf 0.

Dieser Vorgang lässt sich so oft wiederholen wie man möchte. Ist der Array voll und man möchte wieder einen leeren Array zum Beschreiben haben, reicht es, einen rst Vorgang einzuleiten.

Funktionsweise – Lesen

Der Lesevorgang des RAM-Programm Moduls wird wie folgt realisiert:

- a_address und b_address genau spezifizieren, nach den oben genannten Richtlinien.
- ce_read auf 1 setzen.
- Die entsprechenden Datenworte werden aus unserem mem-Array gelesen und jeweils auf a_out und b_out gelegt. (z.B sei b_address das binär kodierte 4, dann wird das Datenwort in mem(4) in b_out ausgegeben.)

Der Anwenderdokumentation lassen sich genauere, visuellere Beispiele der Vorgänge entnehmen. An dieser Stelle sei auf diese hingewiesen.

Bekannte Beschränkungen

- Bei einem Schreibvorgang wird das Datenwort immer auf b_out geleitet. Dies entspricht der korrekten Funktion des RAM Moduls des AM2901, allerdings würde sich hier, vor allem in Hinblick auf die Tests, anbieten, b_out mit einem default Zustand zu beschreiben, da per se das Datenwort bei einem Schreibvorgang nicht unbedingt in der Ausgabe sein müsste. Zusätzlich dazu würden die Tests und das damit verbundene Simulationsbild etwas leichter zu verstehen und zu lesen sein.
- Momentan gibt es keine Fehlerbehandlung falls inkorrekte Eingaben erfolgen.

Erweiterungsoptionen

- Die Speicherkapazität des Arrays kann beliebig angepasst werden.
- Beim Hinzufügen eines zusätzlichen Dateneinganges könnte der Schreibvorgang zusätzlich ausgebaut werden. Hierbei könnten beide Datenworte gleichzeitig in die simulierten Register geschrieben werden, jeweils an den Adressen a_address und b_address.
- Generell lässt sich sagen, dass durch das Hinzufügen von zusätzlichen Eingängen und Ausgängen mehrere Elemente auf einmal gespeichert, bzw. gelesen werden können.

Inwiefern dies sinnvoll wäre und ob sich diese Erweiterung anbieten würde, ist dem Entwickler selbst überlassen.

Grundsätzlich gilt bei allen Erweiterungen, dass sie zwar durchaus möglich sind, unser Modul aber nicht mehr hundertprozentig dem Grundmodell des AM2901 entsprechen würde.

2.2 QREG - Funktionsweise

Das QREG – Programm simuliert das tatsächliche Q-Register des AM2901. Analog zur RAM bestehen die Grundfunktionen des Moduls aus dem Lese- und Schreibvorgängen. Allerdings besitzt das QREG im Gegensatz zur RAM keinen Speicherarray, sondern lediglich ein einzelnes Signal welches den Speicher des QREG imitiert. Auch hier ist das Ziel des Programms, eine einfachere und simplere Art aufzuzeigen, um das AM2901 QREG zu synthetisieren und es später in eine tatsächliche Schaltung umbauen zu können. Das eigentliche Speichern wird durch ein einzelnes Signal, das q_mem, realisiert. Wird ein Schreibvorgang eingeleitet, wird das Datenwort in q_mem gespeichert, wird ein Lesevorgang eingeleitet, wird das der q_out Ausgang mit q_mem beschrieben.

Vorhandene Eingänge und ihre Bedeutung

Zu den vorhandenen Eingängen des Moduls gehören die folgenden:

- `f : 16 Bit langer std_logic_vector`
  Dieser Eingang beschreibt unser eigentliches Datenwort, welches wir in das Register schreiben möchten. Bei einem Lesevorgang ist das Datenwort zu vernachlässigen. Wichtig ist hierbei, dass es sich um ein 16-Bit langen std_logic_vector handeln muss.
- `ce_write & ce_read : std_logic`
  Wenn ce_write/ce_read gesetzt ist, wird der Schreib/Lesevorgang aktiviert. Ist ce_write/ce_read nicht gesetzt, wird das Modul und seine eigentliche Funktionalität nicht aktiviert.
  Darüber hinaus besitzt das Modul auch einen rst Eingang, der, wenn gesetzt, das Speichersignal auf undefined zurücksetzt. Außerdem ist dieses Modul getaktet, folglich besitzt es einen clk Eingang, der eine Periode von 20ns besitzt. Diese Frequenz war der Aufgabenstellung zu entnehmen. Jedwede Art von Vorgang wird nur bei rising_edge(clk) ausgeführt.

Vorhandene Ausgänge und ihre Bedeutung

Zu den vorhandenen Ausgängen des Moduls gehören die folgenden:

- `q_out – 16 Bit langer std_logic_vector`
  Dieser Ausgang gibt nach einem Lesevorgang die Inhalte des Speichersignals q_mem aus. Wenn der Schreibvorgang eingeleitet wird, gibt q_out das gespeicherte Datenwort f aus.

Vorhandene interne Signale und ihre Bedeutung

Zu den auftretenden internen Signalen des Moduls gehört:

- `signal q_mem : std_logic_vector(15 downto 0)`

  Dieses Signal simuliert das QREG, es ist also die eigentliche Einheit, die das Speichern realisiert. Wenn das Register leer ist, ist q_mem auf undefined gesetzt. Bei einem Schreibvorgang wird q_mem entsprechend gefüllt, bei einem Lesevorgang wird von q_mem auf q_out geschrieben.

Was ist zu beachten?

- ce_write und ce_read können nicht gleichzeitig gesetzt werden, auch nicht direkt aufeinanderfolgend im nächsten Takt. Es muss ein Takt Pause dazwischen sein.
- Es ist darauf hinzuweisen, dass beim Lesevorgang tatsächlich etwas in dem q_mem des QREG steht. Ein Tipp wäre, das QREG erst mit mindestens einem Wert zu beschreiben bevor man einen Lesevorgang einleitet.
- Das CLK Signal ist momentan auf 20ns gesetzt, kann aber beliebig verändert werden.
- Wie immer gilt es, auf die Datentypen und –längen der einzelnen Eingänge zu achten, bevor man das Modul benutzt.

Hinweise zum Einsatz

Funktionsweise – Schreiben

Der Schreibvorhang des RAM-Programm Moduls funktioniert wie folgt:

- Lege das zu speichernde Datenwort an den Eingang d.
- Spezifiziere b_address und a_address, achte hierbei auf die Richtlinien, die in Vorhandene Eingänge und ihre Bedeutung genannt worden sind

- Setze ce_write = 1
- Nun wird das Datenwort in das Register in b_address gespeichert. (z.B sei b_address das binär kodierte 4, dann wird das Datenwort in mem(4) geschrieben.
- In b_out wird nun das zuvor gespeicherte Datenwort ausgegeben, a_out besitzt einen default Wert.
- Setze ce_write wieder auf 0.

Dieser Vorgang lässt sich so oft wiederholen wie man möchte. Ist der Array voll und man möchte wieder einen leeren Array zum Beschreiben haben, reicht es, einen rst Vorgang einzuleiten.

Funktionsweise – Lesen

Der Lesevorgang des RAM-Programm Moduls wird wie folgt realisiert:

- a_address und b_address genau spezifizieren, nach den oben genannten Richtlinien.
- ce_read auf 1 setzen.
- Die entsprechenden Datenworte werden aus unserem mem-Array gelesen und jeweils auf a_out und b_out gelegt. (z.B sei b_address das binär kodierte 4, dann wird das Datenwort in mem(4) in b_out ausgegeben.)
- ce_read auf 0 setzen

Der Anwenderdokumentation lassen sich genauere, visuellere Beispiele der Vorgänge entnehmen. An dieser Stelle sei auf diese hingewiesen.

Bekannte Beschränkungen

- Bei einem Schreibvorgang wird das Datenwort f immer auf q_out geleitet. Dies entspricht der korrekten Funktion des QREG Moduls des AM2901, allerdings würde sich hier, vor allem in Hinblick auf die Tests, anbieten, q_out mit einem default Zustand zu beschreiben, da per se das Datenwort bei einem Schreibvorgang nicht unbedingt in der Ausgabe sein müsste. Zusätzlich dazu würde das Testen und das damit verbundene Simulationsbild etwas leichter zu verstehen und zu lesen sein. Dies ist allerdings nur eine Formalität die einem das Testen erleichtert, sie schränkt die Funktionalität des Moduls nicht ein.

- Momentan gibt es keine Fehlerbehandlung falls inkorrekte Eingaben erfolgen.

Erweiterungsoptionen

- Auch hier kann die Speicherkapazität des Arrays erweitert werden. Allerdings würde man dann letztendlich ein RAM-ähnliches Modul erhalten.

Grundsätzlich gilt bei allen Erweiterungen, dass sie zwar durchaus möglich sind, unser Modul aber nicht mehr hundertprozentig dem Grundmodell des AM2901 entsprechen würde.

2.3 ALU – Operandenauswahl – Grobe Funktionsweise

Das ALU-Operandenauswahl-Programm realisiert die Operandenauswahl des AM2901. Hierbei wird die Logik und Auswahl der richtigen Operanden mithilfe Concurrent Statements und Bedingungsabfragen realisiert. Auch hierbei wird angestrebt, durch die Übersetzung in VHDL das Modul leichter verständlich und letztendlich synthetisierbar und in Schaltung umsetzbar zu machen.

Vorhandene Eingänge und ihre Bedeutung

Zu den vorhandenen Eingängen des Moduls gehören die folgenden:

- `i : 9 Bit langer std_logic_vector`

  Dieser Eingang beschreibt unsere Instruktionswort. Hierbei sind die niederwertigsten 3 Bit am wichtigsten, da sie die Logik der Operandenauswahl beschreiben. Diese Bit werden anhand der unten angeführten Tabelle gesetzt.

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

- `d,a,b,q,zero : 16 Bit langer std_logic_vector`

  Dies sind alles Eingänge der ALU-Quelloperandenauswahl, die der AM2901 besitzt und wir modellieren müssen.

- `ce: std_logic`
  Wenn ce gesetzt ist, wird das Modul aktiviert. Ist ce nicht gesetzt, wird das Modul und eine eigentliche Funktionalität nicht aktiviert.

Vorhandene Ausgänge und ihre Bedeutung

Zu den vorhandenen Ausgängen des Moduls gehören die folgenden:

- `r_out & s_out : 16 Bit langer std_logic_vector`
  Die Ausgänge der AM 2901 Alu-Quelloperandenauswahl. Diese werden weiter an die Arithmetisch-logische Einheit weitergeleitet.

Vorhandene Ausgänge und ihre Bedeutung

Dieses Modul besitzt keine Signale.

Was ist zu beachten?

- Wie immer gilt es, auf die Datentypen und –längen der einzelnen Eingänge zu achten, bevor man das Modul benutzt.
- Der Zero Eingang wird in diesem Programm nur der Vollständigkeit wegen aufgeführt. Intern wird dies mit einer Zuweisung von 0 realisiert.

Hinweise zum Einsatz

Das ALU-Quelloperandenauswahl Modul funktioniert wie folgt:

- die niederwertigsten 0-2 Bit des Instruktionswortes i nach oben genannter Tabelle spezifizieren
- alle sonstigen Eingängen im Rahmen der Richtlinien beliebig initiieren
- setze ce = 1
- r_out und s_out geben nun die entsprechenden korrekt ausgewählten zwei Eingänge aus.

Der Anwenderdokumentation lassen sich genauere, visuellere Beispiele der Vorgänge entnehmen. An dieser Stelle sei auf diese hingewiesen.

Bekannte Beschränkungen

- Das gesamte Modul wird mithilfe von Concurrent Statements realisiert, entsprechend gibt es keine Form von Taktung.

- Einen Reset Vorgang gibt es auch nicht, da automatisch undefined auf die beiden Ausgänge r_out und s_out gelegt wird falls keines der oben in der Tabelle genannten Muster auf die niederwertigsten 3 Bit des Instruktionswortes passt.
- Momentan gibt es keine Fehlerbehandlung falls inkorrekte Eingaben erfolgen.

Erweiterungsoptionen

- Erweiterungen im Zuge von Hinzufügen von Mustern in der Tabelle sind möglich, die Anzahl der betrachteten Instruktionsbits muss dann aber entsprechend erhöht werden, um mehr als nur 8 Fälle decken zu können. Dies würde allerdings das gesamte Format des Instruktionswortes verändern.

Grundsätzlich gilt bei allen Erweiterungen, dass sie zwar durchaus möglich sind, unser Modul aber nicht mehr hundertprozentig dem Grundmodell des AM2901 entsprechen würde.