Praktische Übungen zu Computertechnik 2

Versuchsprotokoll

Versuch:

A3 – Befehlssatzerweiterung und Test eines RISC-Prozessors

Versuchsdatum und -zeit:

Donnerstag, 06. Mai 2010, 10-13 Uhr

Betreuer:

Andreas Reinsch

Name, Studiengang, Mat.-Nr.:

Ralf Wondratschek, B. Sc. Informatik, 112626

Email:

ralf.wondratschek@uni-jena.de

Name, Studiengang, Mat.-Nr.:

Kerstin Gößner, B. Sc. Informatik, 114656

Email:

kerstin.goessner@uni-jena.de

Vom Betreuer auszufüllen:

Vorbereitung/Kolloquium:

Durchführung:

Protokoll:

Gesamtbewertung:

Gliederung

Weiterhin sind noch die Quelltexte und das Assemblerprogramm aus dem Versuch angehangen

1. Vorbereitung

Für diesen Versuch ist ein DLXJ-Prozessor (eine RISC-Prozessorarchitektur von Hennessy und Patterson abgewandelt in der Jenaer Version von Dr. Reinsch) gegeben. Der Prozessorkern kennt bereits einige arithmetisch-logische Befehle, zum Beispiel die Subtraktion SUB. Ziel dieses Projektes ist es, elf weitere Befehle hinzuzufügen (ADD, ADD.I, SUB.I, AND, AND.I, OR, OR.I, SLL, SLL.I, SRL und SRL.I). Dabei sind die neuen Befehle in das R-Typ und I-Typ Format zu unterscheiden. Das J-Typ Format, welches ausschließlich von Sprungbefehlen genutzt wird, ist dabei nicht vertreten.

Der Maschinencode für drei der neuen Befehle in Hexadezimaldarstellung sieht wie folgt aus:

1. 0x50010003

Um diesen Code zu entschlüsseln, muss er erst in eine Binärdarstellung umgewandelt werden:

0x50010003 = 010100|00000|00001|0000000000000011

Am Opcode, dem ersten Teil von links der Binärdarstellung, erkennt man, dass es sich um den ADD.I Befehl handelt. In dem Fall steht im Rs1 Register eine Null (2. Teil der Binärdarstellung) und der Immediate beträgt Drei (letzter Teil). In das Rd Register, welches den zu speichernden Wert enthält und auf die Stelle 00001 adressiert ist, wird Drei geschrieben.

Rd ← Rs1 + I[a]

11 ← 0 + 11

2. 0x0021100c = 000000|00001|00001|00010|00000|001100

Da hier der Opcode gleich Null ist, handelt es sich um ein R-Typ Format. Um herauszubekommen, um welche Operation es sich genau handelt, muss die rr-func betrachtet werden (letzter Teil). Der 001100 ist eine logische Verschiebung nach links zu zuordnen (SLL). Das Rs1 Register wird dabei um so viel Stellen, wie im Rs2 Register stehen, nach links verschoben. In dem Fall wird die Drei aus der Adresse 00001 um 3 Stellen nach link geschoben. Aus der 11 wird eine 11000, welche in der Adresse 00010 gespeichert wird.

Rd ← Rs1 log. links geschoben mit Rs2

11000 ← 11 log. links geschoben mit 11

3. 0x00411809 = 000000|00010|00001|00011|00000|001001

Es handelt sich wieder um ein R-Typ Format. Am rr_func Bitfelt erkennt man, dass es ein logical OR ist. Dabei wird der Wert von der Adresse 00010 bitweise mit logischem OR mit der Adresse 00001 verknüpft und anschließend in der Adresse 00011 abgelegt.

Rd ← Rs1 OR Rs2

11011 ← 11000 OR 00011

Allgemein kann man sagen: bei einem R-Typ Format werden zwei Quellregister Rs1 und Rs2 adressiert und ein Zielregister Rd. Am Opcode kann noch nicht festgestellt werden, um welche arithmetische Funktion es sich handelt, das geschieht erst mit dem rr_func Bitfeld (Register-Register-ALU Operation).

Rd ← Rs1 rr_func Rs2

Bei dem I-Typ Format handelt es sich um einen Datentransportbefehl. Am Opcode wird erkannt, dass es sich um einen I-Typ handelt und auch bereits um welche Funktion. Das Rd Register enthält das zu ladende oder speichernde Wort. Des Weiteren gibt es nur ein Quellregister Rs1. Der zweite Wert ist der 16 Bit lange Immediate-Wert, der im Befehl enthalten ist. Über das IR kommt der Wert entweder auf den S1 oder S2 Bus.

Rd ← Rs1 op. I

Die Steuerung unterscheidet zwischen den Befehlstypen. Der Decoder 1 entschlüsselt den entsprechenden Befehl aus dem Opcode. Bei einem R-Typ Format entscheidet das rr_func Bitfeld, welche ALU-Operation an das Schaltwerk weitergeleitet wird. Wird im Decoder 1 ein I-Typ Format festgestellt, so decodiert der Decoder 2 die Datentransport-Operation weiter aus.

Implementiert man die elf neuen Befehle aus der Aufgabenstellung, müsste der Prozessor nur um fünf neue Zustände erweitert werden (siehe Seite A1). Das liegt daran, dass die Zustandsübergänge nur angeben, wie der Befehl abgearbeitet wird. An der ALU liegen der S1 und S2 Bus an, dabei spielt es keine Rolle, von wem das Signal auf den Bussen stammt. Somit braucht man für die I-Typ Befehle keine neuen Zustände.

Zwar müssen nur fünf weiter Zustände hinzugefügt werden, trotzdem kann ein und derselbe Zustand verschiedene Datenquellen haben, zum Beispiel bei ADD und bei ADD.I. Bei R-Typ Befehlen dienen als Quellen die Register A und B, welche mit den Steuersignalen /a_out_en bzw. /b_out_en die Daten auf den S1 Bus bzw. den S2 Bus legen. Wird im Decoder 3 ein I-Typ Befehl erkannt, so liegt nicht das Register B als Datenquelle am S2 Bus, sondern der Immediate des IR. Dabei nimmt das Signal rd_s2_addr_sel die Unterscheidung vor, welche Quelle nun anliegt. Als Datensenke dient das C

Register, in welches mit dem Signal c_latch_en das Datum eingeschrieben wird.

Die genaue Implementierung der elf neuen Befehle findet man im Anhang auf Seite A2, A3 und A4. Dabei handelt es sich allerdings nur um die Änderungen, die später bei der Durchführung an die richtige Stelle eingefügt werden müssen (die genauen Quelltexte findet man Notfalls nochmal am Ende des Anhangs).

Auch haben wir ein Assembler Programm vorbereitet (siehe Seite A5)(wir haben auch den Quelltext aus dem Versuch angefügt, allerdings druckten wir nur die falsche Version aus und vergaßen die korrekte auszudrucken). Dazu setzten wir die Vorlage mit den entsprechenden Ergebnissen fort. Die Ergebnisse, die dabei rauskommen sollen, wenn alles richtig funktioniert, stehen als Kommentare dahinter. Auch haben wir am Ende eine Endlosschleife gesetzt, damit der Prozessor nicht abstürzt. Er wird immer wieder erneut zu demselben Jump-Befehl springen.

2. Vorgehensweise

Um den DLXJ-Prozessor die elf Befehle hinzuzufügen, müssen zuerst die entsprechenden VHDL-Dateien geändert werden. Dazu werden die Befehle aus der Vorbereitung an die entsprechenden Stellen eingefügt. Mit dem passenden Skript dlxj_analyze_for_funcsim werden die Dateien übersetzt. Danach wird das Assemblerprogramm, welches bereits in der Vorbereitung angefertigt wurde, editiert und assembliert.

Anschließend kann die funktionale Simulation durchgeführt werden mit dem Skript dlxj_simulate_function. Hier wird unteranderem überprüft, ob alle zusätzlichen Befehle fehlerfrei funktionieren und ob die Ergebnisse aus dem Assemblerprogramm mit den vorher handschriftlich berechneten Werten übereinstimmen.

Verlief die funktionale Simulation ohne Unstimmigkeiten, so können die elf Befehle in den DLXJ-Prozessor implementiert werden. Nachdem der FPGA konfiguriert und das Testprogramm ebenfalls übertragen wurde, kann mit Hilfe des Debuggers das Testprogramm getestet werden.

3. Erprobung

Nachdem wir unseren Arbeitsplatz vorbereiteten und alle VHDL-Dateien mit dem Skript dlxj_analyze_for_funcsim_all übersetzten, fügten wir den VHDL-Beschreibungen ir_cecode_i-behaviour.vhd, fsm_next-dataflow.vhd und fsm_output-dataflow.vhd die zusätzlichen Befehle aus der Vorbereitung (siehe Anhang) hinzu. Nach einer erneuten Analyse mit dem Skript und korrigierten Syntaxfehlern, editierten und assemblierten wir unser vorbereitetes Assemblerprogramm (siehe Seite A5). Da dies problemlos funktionierte, setzten wir mit der funktionalen Simulation mit Hilfe des Skriptes dlxj_simulate_function fort. Die Ergebnisse der zusätzlich implementierten Befehle findet man im Anhang auf Seite A6 und A7. Auch hier wurden alle Ergebnisse korrekt ausgegeben.

Als die funktionale Simulation komplett abgeschlossen war, implementierten wir die zusätzlichen Befehle in den DLXJ-Prozessor mit dem Skript dlxj_make_fpga und konfigurierten ihn mit dlxj_configure_fpga. Das gleiche Testprogramm aus der Vorbereitung übertrugen wir in den Programmspeicher des Prozessors mit dlxj_program_rom. Danach starten wir den Debugger und testeten unser Programm. Den einzigen Befehl, den wir dazu brauchten, war der n1 Befehl. Er hat zur Folge, dass der nächste Befehl ausgeführt und disassembliert wird. So war es auch möglich, die einzelnen Zustände und Registerinhalten anzeigen zu lassen. Besonders wichtig war uns das R4 Register, wo die Ergebnisse eingetragen und ausgegeben wurden. Bei nicht jedem ausgeführten Befehl hat sich die Ausgabe geändert, die Ergebnisse wurden allerdings in folgender Reihenfolge ausgegeben: 2000.0000 | 0000.0002 | 0000.00001 | 0000.0001 | 0000.0002 | 0000.0004 | 0000.0001 | 0000.0007 | 0000.0005 | 0000.000A | 0000.0004 | 0000.0002 | 0000.0001

Vergleicht man diese Zahlen mit den Ausgaben des Terminals aus der funktionalen Simulation, so stimmen sie wieder überein. Im Debugger sieht man die Ergebnisse, die mit sw.i zero(r31), r4 im Assemblerprogramm ausgegeben werden. Damit eine neue Ausgabe erscheinte, musste man den n1 Befehl im Debugger auch genau so oft eingeben, wie die Anzahl der anderen Befehle zwischen den sw.i Befehlen. Deshalb sieht man zum Beispiel die Zahl 6 im Debugger auch nicht, die aber in der funktionalen Simulation angezeigt wird (siehe erster Befehl auf Seite A7). Gab man weitere n1 Befehle im Debugger ein, so erschien keine neue Ausgabe, da das Programm sich in der Endlosschleife am Ende befand. Das Ergebnis findet man auch nochmal auf Seite A8.

4. Schlussfolgerungen

Bei einer guten Vorbereitung und intensiven Auseinandersetzung mit der Aufgabenstellung, war die Durchführung des Projektes nicht mehr so schwierig. Die vorbereiteten Quelltexte mussten nur noch eingefügt werden. Kleine Fehler beim Einfügen behinderten den Fluss, da man lange nach ihnen suchen musste, konnten aber gelöst werden. Die neuen Befehle funktionierten auch genau so, wie es vorhergesehen war. Auch die Benutzung des Debuggers bereitete keine weiteren Probleme und verwunderliche Störungen traten im ganzen Projekt auch nicht auf.

Insgesamt erfüllte das Ergebnis unsere Erwartungen.