Experimentelle Hardwareprojekte
Lehrstuhl für Advanced Computing
Institut für Informatik

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Versuch A3

Aufbau und Funktionsweise eines RISC-Prozessors

Online-Version

Andreas Reinsch

Inhaltsverzeichnis

Der Versuch führt am Beispiel des DLXJ-Prozessors [2], einer sehr vereinfachten Variante des RISC-Prozessors DLX [1] in den Aufbau und in die Funktionsweise von RISC-Prozessoren ein.

1. Versuchsvorbereitung

1. Informieren Sie sich über die Architektur, die Implementierung und die Testmöglichkeiten des DLXJ-Prozessorsystems [2].

2. Der DLXJ-Prozessorkern soll um die arithmetisch-logischen Befehle der Tabelle 3 in [2] erweitert werden. Der Maschinencode für drei aufeinanderfolgende neue Befehle lautet in Hexadezimaldarstellung:

   $$0x50010003$$
   $$0x0021100c$$
   $$0x00411809$$

   Wie lauten die entsprechenden Assemblerbefehle. Bestimmen Sie die Inhalte der drei Zielregister nachdem diese Befehle nacheinander ausgeführt worden sind. Verwenden Sie zur Lösung der Aufgabe die Tabelle 3 und die Abbildung 1 in [2]

3. Worin unterscheiden sich R-Typ-Befehle von I-Typ-Befehlen ein und derselben Operation (z.B. ADD und $ADD.I$)? Welche Baugruppe nimmt diese Unterscheidung generell vor. Um wieviele zusätzliche Zustände muß der Prozessor erweitert werden, wenn alle oben genannten Befehle implementiert werden sollen? Begründen Sie Ihre Entscheidung.

4. Erweitern Sie den Zustandsgraphen der Abbildung 8 in [2] des DLXJ-Prozessors um die arithmetisch-logischen Befehle der Tabelle 3 in [2] Die hierfür erforderlichen Zustände sind vordefiniert und lauten: add für die Addition, sub, für die Subtraktion, and_1, für die logische UND-Verknüpfung, or_1 für die logische ODER-Verknüpfung, sll_1 für die Operation „schiebe logisch links“ und srl_1 für die Operation „schiebe logisch rechts“ . Wie sind die Datenquellen und -senken im Datenpfad des Prozessors in den zusätzlichen Zuständen anzusteuern?

5. Wie sind die VHDL-Beschreibungen für die Zustandsüberführungsfunktion, für den Decoder 1 und für die Ausgabefunktion zu ergänzen? Schreiben Sie diese Ergänzungen auf. Die Konstanten für die Operationscodes und für die RRA-Funktionen der zusätzlichen Befehle sind vordefiniert (Anhang A).

6. Bereiten Sie ein einfaches Assemblerprogramm vor, das mindestens alle zusätzlich zu implementierenden Befehle enthält und das geeignet ist, die korrekte Funktion dieser Befehle nachzuweisen. Ein Programmbeispiel mit den bereits implementierten Befehlen finden Sie im Anhang B in [2]. Durch eine entsprechende Anweisung am Ende des Programms ist zu verhindern, daß der Prozessor abstürzen kann (die Anweisung „end“ im Programm Anhang B in [2]. ist lediglich eine Assembleranweisung, zusätzlicher Programmcode wird damit nicht generiert).

1.1. Vorbereitung des Arbeitsplatzes

Für den Versuch A3 benötigen Sie den Server `ipc764` und einen Arbeitsplatzrechner `ehpc[XX]`[1]. Die Kommunikation erfolgt jeweils über ein Terminal. Eine visuelle Verbindung über eine Webcam ist für die Versuche A3 und A4 gegenstandslos. Für die Arbeit am Server benötigen Sie eine X Window Umgebung (Option -X bzw. -Y). Datenkompression ist nicht zwingend erforderlich (Option -C), entlastet aber die Netzwerkverbindung.

Entweder stellen Sie eine VPN-Verbindung zum Netz der Fakultät für Mathematik und Informatik her oder Sie melden sich zunächst sowohl für den Zugang zum Server als auch für den Zugang zum Arbeitsplatzrechner in je einem Terminal mit dem Befehl `[Ihr_KSZ-Login]@login.fmi.uni-jena.de` an, für den Serverzugang mit den Optionen `-YC`.

Nun melden Sie sich an einem Arbeitsplatzrechner mit dem Befehl `[Ihr_KSZ-Login]@ehpc[XX].inf-ra.uni-jena.de` an. Sollte bereits ein Nutzer an diesem Arbeitsplatz angemeldet sein (Überprüfung mit `w` oder `who`), dann wechseln Sie bitte zu einem anderen Arbeitsplatzrechner.

In einem zweiten Terminalfenster melden Sie sich am Server mit dem Befehl `[Ihr_KSZ-Login]@ipc764.inf-ra.uni-jena.de -YC` an. Jetzt muß an jedem der beiden Terminalfenster mit dem Befehl

```
source /data/ehp/upgrade_a1-a4_2019/etc/bashrc
```

die Initialisierung der Shell erfolgen. Danach geben Sie an einem Terminal

```
a3_COPY
```

ein. Damit werden alle für den Versuch benötigten Dateien in Ihr `home`-Verzeichnis kopiert und für die aktuelle Shell die erforderlichen Umgebungsvariablen gesetzt. Bereits vorhandene Projektdateien werden **ohne** Nachfrage gelöscht. Der Befehl sollte demzufolge nur einmal zu Beginn Ihrer Arbeit ausgeführt werden. Um auch für das zweite Terminal die Umgebungsvariablen zu initialisieren, führen Sie hier den Befehl

```
a3
```

aus. Falls Sie im Verlauf des Versuchs am Arbeitsplatzrechner oder am Server ein weiteres Terminal öffnen, ist diese Anweisung zu wiederholen.

Für die weitere Arbeit beachten Sie bitte folgendes:

VHDL-Dateien dürfen nicht umbenannt und nicht in andere Verzeichnisse verschoben oder kopiert werden! Für alle Dateien, die im Verlauf des Versuchs eventuell modifiziert oder erstellt werden müssen, sollen die Verzeichnisse ∼/dlxj/v_todo (VHDL-Spezifikationen) und ∼/dlxj/asm (Assemblerprogramme) verwendet werden. Arbeiten Sie grundsätzlich nicht außerhalb des Verzeichnisbaums ∼/dlxj. Der Befehl `dlxj_editor` ruft den GUI-basierten Editor `gedit` auf. Bei geringer Datenrate des Netzwerks ist es günstiger, mit einem Editor ohne GUI zu arbeiten. Geeignet sind z.B. `nano` oder `vi`.

[1] siehe Moodle

Die Abbildung 1 zeigt die im Verlauf des Versuchs benötigten Programme, ihre gegenseitige Abhängigkeit und den Rechner, auf dem sie auszuführen sind.

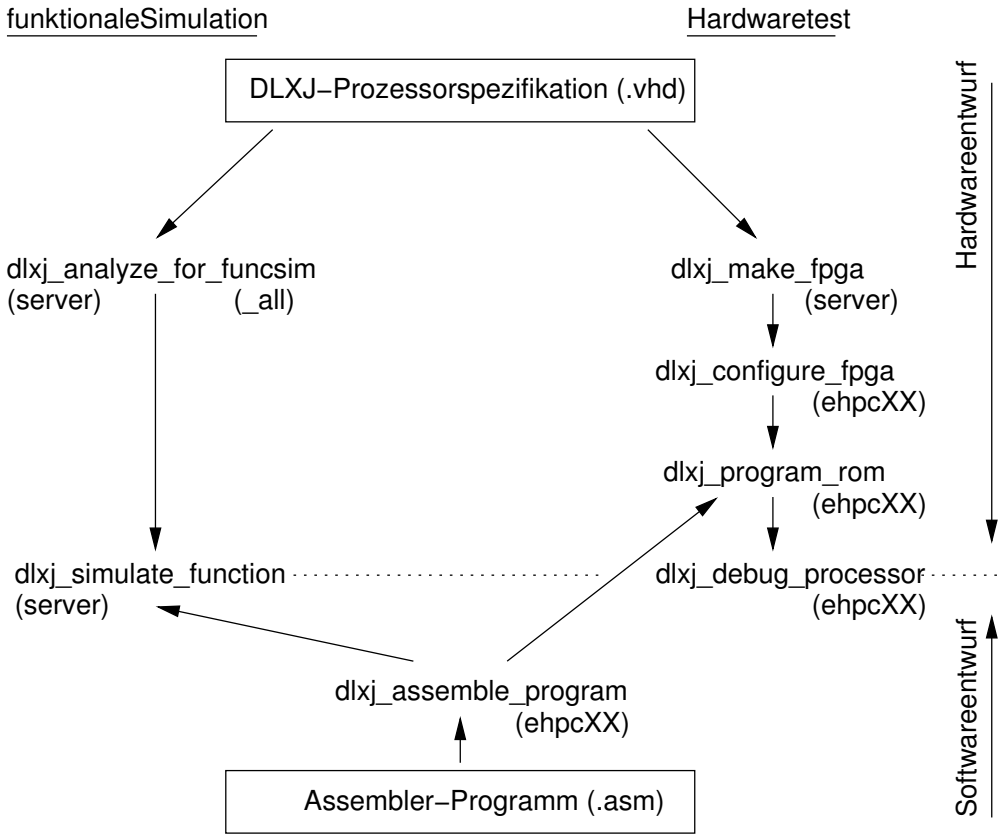

Abbildung 1: Der Entwurfsfluß

2. Versuchsdurchführung

2.1. Implementierung zusätzlicher Befehle

Es sind alle Befehle der Tabelle 3 in [2] zu implementieren:

1. Übersetzen Sie zunächst alle VHDL-Beschreibungen des Prozessors mit Hilfe des Scripts `dlxj_analyze_for_funcsim_all`. Fügen Sie den VHDL-Beschreibungen `ir_decode_1-behaviour.vhd` für den Decoder 1, `fsm_next-dataflow.vhd` für die Zustandsüberführungsfunktion und `fsm_output-dataflow.vhd` für die Ausgabefunktion die erforderlichen Anweisungen hinzu. Übersetzen Sie nun die geänderten Dateien (`dlxj_analyze_for_funcsim`) und korrigieren Sie evtl. vorhandene Syntaxfehler. Das Script `dlxj_analyze_for_funcsim` ist dann erneut auszuführen.

2. Editieren und assemblieren Sie das von Ihnen für den Test der zusätzlichen Befehle vorbereitete Assemblerprogramm. Achten Sie auf eventuelle Fehlerausgaben

des Assemblers. Arbeiten Sie hierzu im Verzeichnis ∼/dlxj/asm/ mit dem Befehl dlxj_assemble_program mit Ihrem Assemblerprogramm als Option.

Als Vorlage kann das Programm ∼/dlxj/asm/inc2_dec1.asm genutzt werden.

Hinweise für die Assemblerbenutzung : Immediate-Werte können dezimal (z.B.: 21), binär (z.B.: B"10101"), hexadezimal (z.B.: X"15"), oder oktal (z.B.: O"25") angegeben werden. Anstelle eines Immediate-Wertes kann ein statischer Ausdruck mit den Operationen $+$, $-$, $*$ stehen. Der Assembler erwartet als letztes Zeichen „Newline" - (Enter); Sonderzeichen sind generell nicht zugelassen. Der erste Befehl darf nicht mit einer Marke beginnen (evtl. einen NOP-Befehl einfügen).

3. Starten Sie die funktionale Simulation (dlxj_simulate_function) und untersuchen Sie nun, ob alle zusätzlich implementierten Befehle korrekt ausgeführt werden. Im DVE-Fenster stehen dafür zwei Schaltflächen zur Verfügung: mit „NI" (next instruction) wird jeweils der nächste Befehl abgearbeitet und mit „80ns" schreitet die Simulation um 80ns voran. Protokollieren Sie das Verhalten (die Zustandsfolge und die sich ändernden GPR). Die Ausgaben des Terminals „Instructions" werden gespeichert und stehen nach beendeter Simulation im Verzeichnis prn zur Verfügung (funcsim_results.pdf).
4. Vergleichen Sie die R-Typ-Befehle mit den I-Typ-Befehlen. Achten Sie dabei auf die Steuersignale $/b\_en\_out$ und $/immed\_en\_out2$ des Datenpfads.

2.2. Test der zusätzlichen Befehle und des Assemblerprogramms

Nach der erfolgreichen Simulation des DLXJ-Prozessors mit erweitertem Befehlssatz soll nun eine entsprechende FPGA-Konfiguration erzeugt und in Verbindung mit dem bereits bei der Simulation verwendeten Assemblerprogramm getestet werden:

1. Implementieren Sie die zusätzlichen Befehle in den DLXJ-Prozessor (dlxj_make_fpga; etwa 5 min Laufzeit).
2. Konfigurieren Sie das FPGA (dlxj_configure_fpga).
3. Übertragen Sie den Maschinencode Ihres Testprogramms in den Programmspeicher des DLXJ-Prozessors (dlxj_program_rom).
4. Starten Sie den Debugger (dlxj_debug_processor) und testen Sie Ihr Assemblerprogramm. Wählen Sie dazu selbständig geeignete Debuggerbefehle aus.
5. Beobachten Sie das Verhalten Ihres Programms. Protokollieren Sie den Testverlauf.
6. Nachdem das Programm vollständig abgearbeitet wurde sollten Sie den Inhalt der Register und des RAMs mit dem Debuggerbefehl: srr abspeichern. Nach Beendigung des Debuggers enthält die Datei debug_results.pdf im Verzeichnis prn dieses Ergebnis.

Dateien im pdf-Format können am Server mit dem Programm evince betrachtet werden.

3. Versuchsauswertung

Mit den im Verlauf des Versuchs entstandenen Aufzeichnungen, Änderungen an den VHDL-Dateien (nur die Änderungen in das Protokoll aufnehmen) und Ausdrucken ist der Gliederung der Versuchsdurchführung entsprechend ein Protokoll anzufertigen. Die Versuchsvorbereitung ist zusammmen mit dem Protokoll abzugeben.

Anhang A Kodierung des erweiterten DLXJ-Befehlssatzes

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE WORK.dlx_types.ALL;

PACKAGE dlx_instructions IS
  -- -------------------------
  -- encoding of Opcodes
  -- -------------------------
  CONSTANT op_rr_alu   : dlx_opcode := "000000";
  CONSTANT op_lw_i     : dlx_opcode := "000100";
  CONSTANT op_sw_i     : dlx_opcode := "001000";
  CONSTANT op_j        : dlx_opcode := "001100";
  CONSTANT op_beqz     : dlx_opcode := "010000";
  --
  -- opcodes for experimental implementations only,
  -- not part of basic DLXJ instruction set:
  --
  constant op_sub_i    : dlx_opcode := "010110";
  CONSTANT op_add_i    : dlx_opcode := "010100";
  constant op_and_i    : dlx_opcode := "011000";
  constant op_or_i     : dlx_opcode := "011001";
  constant op_sll_i    : dlx_opcode := "011100";
  constant op_srl_i    : dlx_opcode := "011110";
  --
  -- -----------------------------------------------
  -- encoding of register-register ALU functions
  -- -----------------------------------------------
  CONSTANT rr_func_nop : dlx_rr_func := "000000";
  CONSTANT rr_func_sub : dlx_rr_func := "000110";
  CONSTANT rr_func_slt : dlx_rr_func := "010100";
  --
  -- register-register ALU functions for experimental
  -- implementations only, not part of basic DLXJ
  -- instruction set:
  --
  CONSTANT rr_func_add : dlx_rr_func := "000100";
  constant rr_func_and : dlx_rr_func := "001000";
  constant rr_func_or  : dlx_rr_func := "001001";
  constant rr_func_sll : dlx_rr_func := "001100";
  constant rr_func_srl : dlx_rr_func := "001110";
  --
  -- -----------------------------------------------
```

```
END dlx_instructions;
```

Anhang B Testprogramm für die Additionsbefehle

```
;-------------------------------------------------------------;
;                                                             ;
; Datei         : add_test.asm                                ;
; Datum         : Nov. 2002                                   ;
; Beschreibung  : Testprogram fuer die Befehle                ;
;                 ADD Rd, Rs1, Rs2 und ADD.I Rd, Rs1, I       ;
;-------------------------------------------------------------;
        org   X"00000000"
        start init
init:
        nop
label   add.i   r1,  r0,  X"0005"
        add     r2,  r1,  r1
        j label
        end
```

Literatur

[1] Hennessy, J.L. and Patterson, D.A. *Rechnerarchitektur – Analyse, Entwurf Implementierung, Bewertung.* Friedr. Vieweg Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, 1994.

[2] A. Reinsch. *Der DLXJ RISC-Prozessor (Skript zur Lehrveranstaltung Experimentelle Hardwareprojekte).* FSU Jena, Institut für Informatik, 2020.