KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur

Martin Giese, David Kempe
Arno Schönegge

Interner Bericht Nr. 16/97

1997

Universität Karlsruhe

Fakultät für Informatik

Institut für Logik, Komplexität und Deduktionssysteme

KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur*

Martin Giese, David Kempe, Arno Schönegge

Zusammenfassung

In der hier beschriebenen Fallstudie wurde das KIV-System (Karlsruhe Interactive Verifier) zur Verifikation von ASM-Spezifikationen (Abstract State Machines) eingesetzt. Diese Fallstudie behandelt die in [BM96, BM97] aufbereitete Verifikation der *DLX*-Pipelining-Architektur. Wir geben Details der formalen Spezifikation und Verifikation mit KIV, schätzen den damit verbundenen Arbeitsaufwand ab und skizzieren kleinere Unzulänglichkeiten der informellen Verifikation, welche durch die Formalisierung aufgedeckt werden konnten.

Zudem wird von einer Erweiterung des KIV-Systems um zwei neue Beweistaktiken berichtet, welche speziell auf die effiziente Verifikation von ASM-Spezifikationen zugeschnitten sind. Diese wurden im Rahmen der hier behandelten Fallstudie erarbeitet, implementiert und eingesetzt.

Inhaltsverzeichnis

*Diese Arbeit wurde von der Deutschen Forschungsgemeinschaft im Rahmen des Schwerpunktprogrammes "Deduktion" finanziert.

1 Einleitung

ASMs (*Abstract State Machines*) [Gur94], auch als *Evolving Algebras* bekannt, sind ein Formalismus zur zustandsbasierten Spezifikation. Daneben wird mit ihnen oftmals eine gewisse Verifikationsmethodik verbunden, die es erlaubt komplexe Beweisaufgaben in handhabbare Teilaufgaben zu zerlegen. Das Prinzip dabei ist, die „Differenz“ zweier Maschinenbeschreibungen, deren „Äquivalenz“ nachzuweisen ist, durch Konstruktion geeigneter Zwischenspezifikationen (schrittweise Verfeinerungen) zu überbrücken; [Bör95]:

> *“My guiding principle for breaking complex statements into simpler ones has been to stop only where the proofs become routine inductions and case distinctions which can be carried out by an automatic or interactive theorem proving system.”*

Dieses Verfahren wurde in verschiedenen Anwendungen erfolgreich eingesetzt, z. B. zur Compilerverifikation [BR95, BDR94] oder zur Protokollverifikation [Hug94]. Insbesondere wurden bereits erste Schritte dahin unternommen, die resultierenden natürlichsprachlichen Teilbeweise tatsächlich streng formal mit Hilfe von Deduktionssystemen durchzuführen. Etwa wurde die WAM-Verifikation sowohl mit KIV [SA97] als auch mit Isabelle [Pus96] angegangen.

In dieser Arbeit wird von einer Fallstudie berichtet, in der das KIV-System (Karlsruhe Interactive Verifier) [Rei92] zur Verifikation der *DLX*-Pipelining-Architektur eingesetzt wurde. Als Grundlage dieser Fallstudie wurden die von Börger & Mazzanti [BM96, BM97] gegebenen ASM-Spezifikationen benutzt (und wir werden hier Kenntnis dieses Artikels voraussetzen). Neben der formalen Verifikation an sich ist diese Arbeit durch folgende Fragestellungen motiviert:

- Ist der ASM-Ansatz als Ausgangspunkt für die formale (mechanische) Verifikation geeignet?

  Einerseits ist die bereits angesprochene Verfeinerungstechnik geeignet, komplexe Beweisprobleme in für Deduktionssysteme handhabbare Teilprobleme zu zerlegen. Andererseits sind die in den natürlichsprachlichen Beweisen verwendeten informellen Argumente oft nicht in einfacher Weise formalisierbar (aber gerade die Strenge des Formalismus und eines Deduktionssystems bieten die Sicherheit, die man sich in manchen Anwendungen wünscht).

- Ist das KIV-System für die Verifikation von ASM-Spezifikationen geeignet?

  Das KIV-System ist ein interaktives Deduktionswerkzeug, welches auf Dynamischer Logik [Har79, HRS89] basiert und somit zur Verifikation imperativer Programme einsetzbar ist. Es wurde dahingehend erweitert, daß als Programmkonstrukte (neben Zuweisung, bedingter Anweisung, Schleife und Prozeduren) auch die für ASMs typischen Konstrukte simultane Ausführung (`simultaneous`), indeterministische Ausführung (`alternative`) und Abänderung von Funktionswerten (*function update*) bereitgestellt sind [Sch95]. Dies erlaubt es, in KIV Aussagen über ASM-Spezifikationen zu formalisieren[1] und zu verifizieren.

  Die Fallstudie soll einerseits aufzeigen, daß KIV zur Verifikation von ASMs geeignet ist und andererseits der Identifizierung von für diese Anwendung typischen Beweistaktiken und -Heuristiken dienen. Tatsächlich konnte die Effizienz der Beweisführung mit KIV durch die Implementierung solcher ASM-typischen Beweistaktiken erheblich gesteigert werden.

Das Papier gliedert sich wie folgt. Abschnitt 2 beschreibt, wie die in [BM97] gegebenen ASM-Spezifikationen (inklusive der zugrundeliegenden Datentypen) und die zu zeigende Beweisverpflichtung (inklusive der dabei benutzten Prädikate) in KIV formalisiert wurden. (Die komplette Spezifikation findet sich in Anhang A.) In Abschnitt 3 wird dann erläutert wie der formale Beweis mit KIV geführt wurde. Die im Laufe dieser Fallstudie erarbeiteten Erweiterungen des KIV-Systems um ASM-spezifische Beweistaktiken und -Heuristiken wird in Abschnitt 4 behandelt. Schließlich diskutieren wir die gewonnenen Erkenntnisse in Abschnitt 5.

[1] In Dynamischer Logik lassen sich jedoch nicht beliebige Aussagen unmittelbar ausdrücken; z. B. nicht Eigenschaften wie Fairness oder Liveness, die etwa bei der Protokollverifikation eine Rolle spielen.

2 Formalisierung in KIV

In der ersten Phase des Fallstudie wurde die in Teilen natürlichsprachliche Spezifikation in [BM96] in eine zur Verifikation mit KIV geeignete formale Spezifikation übertragen. Diese Formalisierung umfaßte die grundlegenden Datentypen und Funktionen, die Arbeitsweise der Prozessoren (d. h. die ASM-Regeln) und die zu beweisende Korrektheitsaussage. Die Beweise erforderten darüberhinaus einige Hilfsprädikate. In KIV wurden Datentypen und Funktionen algebraisch spezifiziert, die ASM-Regeln als Programme beschrieben, und die Korrektheitsaussage in (erweiterter) Dynamischer Logik [Sch95] formuliert.

Wir versuchten, die Grundideen der Spezifikation in [BM96] beizubehalten, gleichzeitig verschiedene Modifikationen vorzunehmen, um die Stärken des Beweissystems KIV besonders auszunützen.

2.1 Grundlegenden Datentypen und Funktionen

Um die in [BM96] angestrebte Unabhängigkeit des Korrektheitsbeweises von der tatsächlichen Form der Datentypen noch zu verstärken, haben wir darauf verzichtet, die zugrundeliegenden Datentypen als Teilmengen voneinander zu deklarieren. Stattdessen haben wir, wo nötig, Konversionsfunktionen zwischen Datentypen bereitgestellt, die im man sich im konkreten Fall der Teilmengenbeziehung als Einbettungen vorstellen kann. Zum Beispiel haben wir auf die Forderung `word ⊆address` verzichtet und stattdessen eine Konversionsfunktion `word->address` bereitgestellt, die, wo nötig, Berechnungsresultate in Adressen umwandelt.

Bei der Unterscheidung der verschiedenen Befehlstypen haben wir auf die explizite Verwendung von Mengen verzichtet und stattdessen Prädikate auf den Operationen eingeführt, die auf Zugehörigkeit zu den jeweiligen Befehlsklassen testen. Um die ursprüngliche Notation möglichst zu erhalten, wurden die Prädikate mit `∈alu`, `∈load` usw. bezeichnet, siehe Anhang A.1.

Während wir für Speicher und allgemeine Register Funktionen verwendet haben, haben wir uns entschlossen, für interne Register einzelne Variablen zu verwenden. Hiermit konnte leicht ausgeschlossen werden, daß prozessorinterne Register als Zielregister von Operationen verwendet werden,[2] gleichzeitig konnte auf natürliche Weise die abkürzende Schreibweise aus [BM96] beibehalten werden.

Um die Arbeit mit undefinierten Werten zu vermeiden, haben wir die Befehlsklasse `NOP` hinzugefügt. Ein `NOP`-Befehl bewirkt auf der *DLX* eine sofortige Rückkehr in den Modus `fetch`, auf der DLX^p wird in keiner Pipeline-Stufe eine Aktion ausgeführt. Damit wird das für undefinierte Befehle verlangte Verhalten gewährleistet.[3]

Ein Punkt, in dem eine tiefgreifendere Veränderung gegenüber [BM96] nötig war, war die Vermeidung selbstmodifizierenden Codes. Börger & Mazzanti haben diese Forderung rein verbal gestellt, ohne eine formale Spezifikation der Eigenschaft Selbstmodifikation anzugeben. Konkret bedeutet deren Ausschluß, daß Teile des Speichers als nicht vom Programm beschreibbar bzw. als reiner Programmspeicher angesehen werden müssen. Läßt man nun die Bedingung fallen, daß der Programmspeicher Teil des Datenspeichers ist, so gelangt man zu einer Harvard-Architektur mit getrennten Speichern für Programm und Daten. Diese haben wir modelliert, indem wir zusätzlich zur Funktion `mem` eine (nie überschriebene) Funktion `mem_instr` eingeführt haben, über die der Zugriff auf den Programmspeicher vorgenommen wird.

Um das Einfügen von je zwei `NOP`s nach `BRANCH`- oder `JUMP`-Anweisungen zu erreichen, modifizierten wir die `next`-Funktion zu einer Funktion `next_p`, die auf der DLX^p anstelle der `next`-Funktion bei der Berechnung der neuen Adresse verwendet wird. Durch die folgenden Axiome wird auch festgelegt, daß für alle anderen Befehlsklassen beide Funktionen übereinstimmende Resultate liefern (siehe Anhang A.1).

---

[2] Dies wurde in [BM96] nicht spezifiziert, ist aber offenbar nötig!

[3] In [BM96] war allerdings weder spezifiziert, wie sich die *DLX* bei undefinierten Befehlen verhält noch war deren Auftreten ausgeschlossen.

```
      opcode (mem_instr (ins_add)) ∈jump
    ∨ opcode (mem_instr (ins_add)) ∈branch
 → (  opcode(mem_instr(next_p(ins_add))) ∈nop
    ∧ opcode(mem_instr(next_p(next_p(ins_add)))) ∈nop
    ∧ next_p(next_p(next_p(ins_add))) = next(ins_add) ),

      ¬ opcode (mem_instr (ins_add)) ∈jump
    ∧ ¬ opcode (mem_instr (ins_add)) ∈branch
 → next_p(ins_add) = next(ins_add)
```

2.2 Regeln der Prozessoren

Die Formulierung der Befehlsausführung konnte nahezu unverändert übernommen werden, abgesehen von den oben beschriebenen Modifikationen. Die simultane Ausführung von Regeln konnte leicht unter Verwendung des im Sprachschatz von KIV vorhandenen `simultaneous`-Konstrukts formuliert werden (siehe [Sch95]). Wir haben die Abarbeitung jeweils eines Taktzyklus als Prozeduren `DLX-step#` bzw. `DLXp-step#` formuliert, siehe Anhang A.4, A.5. Ein kompletter Befehl wird auf der *DLX* durch die Prozedur `DLX-cycle#` beschrieben — hier werden bis zur Rückkehr in den Modus fetch alle Stufen abgearbeitet.

Die simultane Abarbeitung von Programmen auf beiden Prozessoren wird durch die Prozedur `DLX-DLXp-loop#` beschrieben. Hier wird in einer Schleife mit indeterministischer Terminierung jeweils auf beiden Prozessoren ein Befehl abgearbeitet. Im Beweis wird dann gezeigt werden, daß für alle terminierenden Durchläufe die Resultate im Sinne von [BM96] übereinstimmen (vgl. 2.3). Um nur solche Durchläufe zu betrachten, in denen keine Datenabhängigkeiten auftreten, wird nach jedem Befehl auf Datenabhängigkeit getestet. Tritt eine solche auf, wird mit Hilfe des `abort`-Befehls eine Nichtterminierung erreicht, so daß für den entsprechenden Durchlauf keine Aussage bewiesen werden muß.

Um eine Synchronisation zwischen beiden Prozessoren zu erreichen, war eine Sonderbehandlung der Sprungbefehle notwendig. Die in [BM96] gemachte Annahme, daß die `NOP`-Operationen keinen eigenen Instruktionszyklus starten, konnte nur aufrechterhalten werden, indem die DLX^p nach jedem Sprungbefehl zwei weitere Befehle lädt, ohne daß auf der *DLX* gleichzeitig Befehle bearbeitet werden. Somit erhält man als Invariante, daß bei jeder Terminierung von `DLX-DLXp-loop#` die Befehlszähler beider Prozessoren gleiche Inhalte haben.

Bei allen verwendeten Prozeduren haben wir uns entschieden, die für den Prozessor zugänglichen Daten nicht als globale Zustandsvariable zu behandeln, sondern als Parameter an die Prozedur zu übergeben. Dadurch erreichen wir, daß sowohl für den menschlichen Betrachter als auch für das Beweissystem syntaktisch leicht festzustellen ist, welche Inhalte durch welche Prozeduren (entsprechend welchen Prozessor) verändert werden können.

Um die Funktionen `mem` und `reg` als Parameter behandeln zu können, mußten wir sie endlich modellieren, da KIV die Übergabe von Funktionen als Parameter nicht erlaubt. In Hinsicht auf den modellierten Sachverhalt stellt diese Abweichung keinen Verlust an Allgemeinheit dar, da der zur Verfügung stehende Speicher (bzw. die Register) stets endliche Gebilde sein werden.

2.3 Spezifikation der Prädikate

Um die in [BM96] beschriebene Übereinstimmung der Berechnungsergebnisse möglichst intuitiv zu formalisieren, haben wir uns entschlossen, eine Hilfsprozedur `equalresults#` zu definieren, siehe Anhang A.6. Diese liefert das Ergebnis `tt` (also „wahr“) genau dann, wenn die Berechnungsergebnisse des aktuellen Befehls (d. h. des Befehls, auf dessen Adresse der Befehlszähler zeigt) auf *DLX* und DLX^p übereinstimmen.

Hierzu wird der Befehl auf der *DLX* vollständig ausgeführt, während auf der DLX^p so viele Pipeline-Stufen, wie durch Tabelle 1 in [BM96] angegeben ist, abgearbeitet werden. Anschließend werden alle Inhalte verglichen, die in Tabelle 1 als Ergebnisse des Befehls bezeichnet werden, und das Ergebnis der Vergleiche wird konjunktiv verknüpft.

Um die Beweisstruktur an die in [BM96] anzulehnen, wählten wir eine ähnliche Unterteilung in zwei Hilfslemmata, von denen Teil a eine Aussage über die Eingangswerte für die Befehle einer Berechnung machte, während Teil b die Gleichheit der Ergebnisse eines einzelnen Befehls bei gleichen Eingangswerten besagt. Somit war nur für Teil a des Lemmas ein Induktionsbeweis nötig, während Teil b genauer Bezug nimmt auf die zugrundeliegenden Regeln.

Um die Trennung tatsächlich aufrecht erhalten zu können, erwies es sich allerdings als nötig, nicht nur die in [BM96] als relevant bezeichneten, sondern alle möglicherweise verwendeten Inhalte zu vergleichen. Denn im Beweis des b-Lemmas in [BM96] war implizit auf die Induktionsvoraussetzung im Teil a Bezug genommen worden — eine Technik, die maschineller Verifikation nicht standhalten konnte. Wir führten stattdessen ein Prädikat `equalrelevantusedlocations` ein, das die Gleichheit aller potentiell verwendeten Inhalte beschreibt.

Da die Gleichheit nicht im ersten Schritt, sondern erst zu einem späteren Zeitpunkt gefordert wird, definierten wir Hilfsfunktionen, die quasi die Abarbeitung in der Pipeline simulieren, um anhand der vorangehenden Befehle den Inhalt zum kritischen Zeitpunkt zu berechnen (siehe Anhang B.1). Die kritischen Zeitpunkte mußten wir gegenüber den Angaben in Tabelle 2 von [BM96] korrigieren, da dort nicht berücksichtigt worden war, daß die Inhalte der Register und des Programmzählers (im Fall eines Sprungbefehls) schon zum Zeitpunkt ID vorliegen müssen, da sie dann in Hilfsregister übertragen werden.

Über die oben beschriebenen Prädikate hinaus mußten wir noch ein Prädikat `datadependent` einführen, das auf Datenabhängigkeiten testet. Hier ließen sich die in [BM96] angegebenen Definitionen problemlos formalisieren (siehe Anhang A.3).

2.4 Correctness Theorem

Mit Hilfe der oben beschriebenen Formalisierungen ließ sich das Correctness Theorem, die zentrale Aussage, folgendermaßen formulieren (siehe Anhang A.7):

```
mode = fetch,
opcode(IR_p)∈nop, opcode(IR1_p)∈nop, opcode(IR2_p)∈nop, opcode(IR3_p)∈nop,
reg_p = reg, mem_p = mem, PC_p = PC, IAR = IAR_p,
⟨DLX-DLXp-loop# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                   reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                   A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)⟩
⟨equalresults# (mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p;
                equalresults)⟩ equalresults = ff
⊢
```

Hierdurch wird ausgedrückt, daß bei übereinstimmenden Werten auf den gemeinsamen Registern (bzw. dem Arbeitsspeicher) und einer leeren Pipeline der DLX^p sowie korrektem Anfangszustand der *DLX* für jeden terminierenden Durchlauf der indeterministischen Prozedur `DLX-DLXp-loop#` die gleichen Ergebnisse im Sinne von [BM96] berechnet werden.

Die dort getroffene Voraussetzung, daß die Register PC1 und C1 mit dem Wert *undef* initialisiert seien, war nicht notwendig und konnte daher fallengelassen werden.

3 Beweis in KIV

Entsprechend dem Vorgehen in [BM96] bewiesen wir das Correctness Theorem, indem wir zwei Hilfslemmata (**DLXp-lemma-a**, **DLXp-lemma-b**) fomulierten, die zusammen die Aussage implizieren. Im Teil a des DLX^p-Lemmas mußte gezeigt werden, daß unter synchronen Startbedingungen DLX und DLX^p stets in einem Zustand enden, in dem alle relevanten Register und Speicherzellen den gleichen Inhalt besitzen. Teil b besagt, daß unter eben diesen Voraussetzungen DLX und DLX^p identische Ergebnisse berechnen. Die Anwendung beider Lemmata impliziert dann offensichtlich die Behauptung.

So erforderte der formale Beweis des Correctness Theorem auch nur noch 21 Schritte, davon 6 Interaktionen, die im wesentlichen die Anwendung von **DLXp-lemma-a** und **DLXp-lemma-b** sind, wobei dazu noch zwei einfache Invariantenaussagen benötigt wurden, die die möglichen Anfangszustände für synchrone Läufe von DLX und DLX^p charakterisieren.

3.1 Beweis des Lemmas DLXp-lemma-a

Das Lemma **DLXp-lemma-a** stellt eine Verschärfung der ursprünglichen Aussage dar, da nicht nur die Gleichheit des zuletzt berechneten Ergebnisses, sondern auch die weiterer Daten behauptet wird. In der verschärften Version ist die Aussage stark genug, um als Induktionsvoraussetzung zu dienen.

Als Sequenz in dynamischer Logik sieht das Lemma folgendermaßen aus (vgl. Anhang B.3):

```
mode = fetch,
opcode(IR_p)∈nop, opcode(IR1_p)∈nop, opcode(IR2_p)∈nop, opcode(IR3_p)∈nop,
reg_p = reg, mem_p = mem, PC_p = PC, IAR = IAR_p,
⟨DLX-DLXp-loop# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                   reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                   A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)⟩
   ¬ equalrelevantusedlocations (
                  mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                  reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                  A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
⊢
```

In den ersten beiden Zeilen wird ausgesagt, daß sich DLX und DLX^p in synchronen Initialzuständen befinden. Hierzu ist erforderlich, daß die Inhalte der Befehlszähler, der Interruptadreßregister, der allgemeinen Register und der Speicher übereinstimmen. Außerdem muß gefordert werden, daß die Pipeline der DLX^p mit `NOP`-Befehlen angefüllt ist und sich die DLX im Zustand fetch befindet, also bereit ist, den nächsten Befehl zu lesen.

Liegt nach Beendigung der (nichtdeterministischen) Prozedur `DLX-DLXp-loop#` ein Zustand vor, in dem das Prädikat `equalrelevantusedlocations` nicht erfüllt ist, so muß ein Widerspruch folgen. Denn dies besagt ja gerade, daß mindestens eine der obengenannten relevanten Stellen auf beiden Maschinen abweichende Werte beinhaltet.

Welche Inhalte gerade relevant sind, hängt bei der DLX^p nicht nur davon ab, welcher Befehl als nächstes ausgeführt werden wird, sondern auch davon, welche Befehle noch in der Pipeline sind. Schließlich kann ein noch nicht beendeter Befehl den Inhalt eines Registers noch verändern, bevor der Inhalt relevant wird. Daher wird durch Hilfsfunktionen `fstopvalue`, `scdopvalue`, `iarvalue`, `alocvalue` beschrieben, welchen Inhalt ein Register bzw. eine Speicherstelle zum relevanten Zeitpunkt haben wird, abhängig von den derzeitigen Inhalten und den folgenden Befehlen. Die Beschreibung des Prädikats `equalrelevantusedlocations` und der Hilfsfunktionen mittels Axiomen erfolgt in den zugrundeliegenden Spezifikationen (siehe Anhang B.1, B.2).

In [BM96] wird der Beweis dieses Lemmas mit den Worten

> „[...] and they both compute the same result. Therefore, IC_{n+1} and IC^p_{n+1} are instruction cycles for the same instruction *instr* and start with the same values for the relevant locations used by that instruction.“

abgetan. Tatsächlich stecken hinter dem Induktionsschritt einige Überlegungen, die offensichtlich dem geneigten Leser überlassen werden, obwohl sie teilweise nicht gerade trivial sind.

Die hauptsächliche Erkenntnis ist hierbei, daß eine Stelle, die nicht explizit verändert wird, ihren Inhalt beibehält. Umgekehrt muß jede Veränderung, die der Inhalt einer Stelle gegenüber ihrem initialen Wert erfahren hat, aus einer expliziten Veränderung zu einem bestimmten Zeitpunkt der Abarbeitung resultieren.

Unterscheiden sich zu einem bestimmten Zeitpunkt die Inhalte der Stellen von *DLX* und DLX^p so muß es also einen letzten Zeitpunkt gegeben haben, zu dem die betreffende Stelle modifiziert wurde, und seit diesem Zeitpunkt müssen sich die Stellen stets unterschieden haben.

Hat man nun als Induktionsvoraussetzung, daß zu jedem vorangehenden Zeitpunkt alle relevanten Stellen über gleiche Inhalte verfügten, so folgt mit Hilfe von Lemma `DLXp-lemma-b`, daß auch das gleiche Ergebnis berechnet wurde, im Widerspruch zur Annahme, nach Ausführung des Befehls unterschieden sich die betreffenden Stellen.

Durch diese Vorüberlegungen ist die grobe Struktur des Beweises schon ersichtlich. Es wird eine vollständige Induktion über die Anzahl abgearbeiteter Befehle geführt. Unter der Annahme, es hätten nicht alle relevanten Stellen übereinstimmende Inhalte, werden vier Fälle unterschieden, je nachdem, ob der Unterschied im Befehlszähler, dem Interruptadreßregister, einem allgemeinen Register oder einer Speicherstelle auftritt.

Der Befehlszähler stellt insofern einen Sonderfall dar, als er bei jedem Befehl verändert wird. Hier wird also noch nicht wirklich die Induktionsvoraussetzung für alle vorangehenden Befehle benötigt, sondern lediglich die Gleichheit der relevanten Stellen vor Abarbeitung des *letzten* Befehls. Auf den letzten Befehl läßt sich Lemma `DLXp-lemma-b` anwenden. Es folgt, daß *DLX* und DLX^p das gleiche Ergebnis berechnen. Da auch der Inhalt des Befehlszählers Teil des Ergebnisses jedes Befehls ist, folgt hieraus die Gleichheit der Inhalte.

Allerdings erfordert auch dieser Teil des Beweises noch einige Arbeit im Detail. Denn das Ergebnis der Ausführung eines Befehls auf der DLX^p liegt noch nicht nach der Befehlshole-Phase vor, vielmehr müssen abhängig von der Art des Befehls noch weitere Schritte durchlaufen werden. Daher werden in der Prozedur `equalresults#`, deren Ergebnis die Gleichheit der Resultate beschreibt, die jeweils nötigen Einzelschritte abgearbeitet und das Ergebnis verglichen. Um sicherzustellen, daß ein einmal festgestellter Unterschied bis zum Prozedurende propagiert wird, müssen alle Fälle vollständig betrachtet werden.

Die Fälle, in denen die Unterschiede im Interruptadreßregister, in einem allgemeinen Register oder einer Speicherstelle auftreten, sind sich in der Struktur relativ ähnlich, wir haben daher exemplarisch nur den Fall der Unterschiede im Interruptadreßregister `IAR` betrachtet.

Während die nichtdeterministische Terminierung der Schleife zur möglichst genauen Modellierung des realen Verhaltens der Prozessoren am geeignetesten war, erwies es sich bei den anschließenden Korrektheitsbeweisen als günstig, die Anzahl abzuarbeitender Befehle explizit als Zähler an die Prozedur zu übergeben. Zu diesem Zweck haben wir die (ansonsten identische) Prozedur `DLX-DLXp-nloop#` definiert, und deren Äquivalenz zu `DLX-DLXp-loop#` bewiesen, in dem Sinne, daß eine Aussage genau dann für alle terminierenden Durchläufe von `DLX-DLXp-nloop#` gilt, wenn sie für alle terminierenden Durchläufe von `DLX-DLXp-loop#` gilt.

Der Beweis gliedert sich in zwei etwa gleich umfangreiche Teile. Ein Hilfslemma formalisiert die oben beschriebene Erkenntnis, daß die unterschiedlichen Werte des `IAR` ihren Ursprung in einem vorangehenden Befehl haben müssen. Genauer besagt das Lemma, daß bei übereinstimmenden Anfangszuständen und unterschiedlichen Werten von `IAR` nach Ausführung von n Befehlen ein $n_0 < n$ existieren muß, so daß der n_0-te Befehl `IAR` modifiziert und sich nach seiner Ausführung die Werte des `IAR` bei *DLX* und DLX^p unterscheiden. Formal sieht das Lemma wie folgt aus:

```
 mode = fetch,
 opcode(IR_p)∈nop, opcode(IR1_p)∈nop, opcode(IR2_p)∈nop, opcode(IR3_p)∈nop,
 reg_p = reg, mem_p = mem, PC_p = PC, IAR = IAR_p,
 ∀ n0.
  (  n0 < n
   → ⟨DLX-DLXp-nloop#(n0; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                         reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                         A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)⟩
         PC = PC_p ),
  ⟨DLX-DLXp-nloop# (n ; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                       reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                       A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)⟩
        (  ¬ opcode(IR_p) ∈movi2s ∧ ¬ opcode(IR1_p) ∈movi2s
         ∧ ¬ opcode(IR_p) ∈trap ∧ ¬ opcode(IR1_p) ∈trap ∧ IAR ≠ IAR_p )
⊢
 ∃ n0.
  (  n0 < n
   ∧ ⟨DLX-DLXp-nloop#(n0; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                         reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                         A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)⟩
          (opcode(mem_instr(PC_p)) ∈movi2s ∨ opcode(mem_instr(PC_p)) ∈trap)
   ∧ ⟨begin
       DLX-DLXp-nloop#(n0; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                         reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                         A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p);
       DLX-cycle# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR);
       DLXp-step# (; reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                     A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p);
       DLXp-step# (; reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                     A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p);
       DLXp-step# (; reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                     A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p);
      end⟩  IAR ≠ IAR_p  )
```

Durch die ersten drei Zeilen wird wiederum ein synchroner Startzustand der beiden Maschinen gefordert. Die zweite Voraussetzung besagt, daß die gelesenen Programme bis zum nb-ten Befehl auf beiden Maschinen übereinstimmen. Dies ist ja durch die Induktionsvoraussetzung sichergestellt. In der dritten Voraussetzung schließlich wird ausgesagt, daß nach Ausführen des n-ten Befehls die Interruptadreßregister der beiden Maschinen verschiedene Inhalte haben, und daß dies nicht daher rührt, daß sich der IAR modifizierende Befehl auf der DLX^p noch in der Pipeline befindet, es sich also nicht um einen temporären Zustand handelt.

Unter diesen Voraussetzungen folgt, daß ein früherer Zeitpunkt n_0 existiert, so daß nach Ausführung von n_0 Befehlen ein IAR modifizierender Befehl anliegt, also ein TRAP- oder MOVS2I-Befehl. Wird dieser Befehl bis zum Vorliegen des Ergebnisses abgearbeitet — also bei der *DLX* einen vollen Befehlszyklus lang, bei der DLX^p drei Pipelinestufen — so unterscheiden sich danach die Inhalte des IAR auf beiden Maschinen.

Der Beweis dieses Lemmas erfolgte durch Induktion über die Anzahl n der abgearbeiteten Befehle. Im Induktionsanfang folgt unmittelbar ein Widerspruch zur Voraussetzung, bei den am Anfang in der Pipeline befindlichen Befehlen handele es sich um NOP-Befehle.

Im Induktionsschritt muß unterschieden werden, ob unter den letzten Befehlen einer das IAR modifizierte. War dies der Fall, so ist dessen Index in der Befehlsfolge das gesuchte n_0, und es muß nur noch gezeigt werden, daß seine Ausführung auch tatsächlich das Resultat unterschiedlicher IAR liefert. Andernfalls kann das n_0 aus der Induktionsvoraussetzung verwendet werden, und es ist zu zeigen, daß die nachfolgenden Befehle keine weiteren Veränderungen am IAR mehr bewirken.

In beiden Fällen müssen rückwirkend mehrere Befehle betrachtet werden, um die Effekte der noch in der Pipeline befindlichen Befehle mitzubehandeln. Da Sprünge anders als sequentielle Befehle behandelt werden, ergeben sich jeweils verschiedene Fälle, so daß der Beweis ziemlich umfangreich wird (450 Beweisschritte).

Mit Hilfe des obigen Lemmas lassen sich Unterschiede im `IAR` auf einen früheren Befehl zurückführen, so daß für diesen die Induktionsvoraussetzung angewendet werden kann und einen Widerspruch liefert. Nicht betrachtet sind hierdurch jedoch die Fälle, in denen das relevante Register auf der DLX^p nicht `IAR` selbst ist. Dies tritt dann ein, wenn einer der beiden unmittelbar vorangehenden Befehle `IAR` modifiziert. Diese Fälle bilden die ähnlich umfangreiche zweite Hälfte des Beweises (ca. 330 Beweisschritte), die im folgenden kurz umrissen wird.

Zu dem Zeitpunkt, zu dem der `IAR` lesende Befehl geladen wird, ist die Modifikation noch nicht sichtbar geworden, wird es jedoch bis zu dem Zeitpunkt, zu dem der Inhalt relevant wird. In diesem Fall muß also die Gleichheit des `IAR` der *DLX* mit dem einer Speicherstelle oder eines allgemeinen Registers auf der DLX^p gezeigt werden, je nachdem, um was für einen Befehl es sich handelt. Auch hier sind bei der Verfolgung der verschiedenen Ausführungsphasen mehrere Fallunterscheidungen nach vorangehenden Befehlen nötig. Da der modifizierende Befehl unmittelbar vorangeht, wird die Induktionsvoraussetzung auch nur für einen direkten Vorgänger benötigt.

3.2 Beweis des Lemmas DLXp-lemma-b

Die Aussage von Teil b des DLX^p Lemmas wurde in **DLXp-lemma-b** formalisiert (siehe B.3):

```
  mode = fetch,
  ¬ opcode(IR1_p) ∈jump, ¬ opcode(IR1_p) ∈branch,
  ¬ opcode(IR_p)  ∈jump, ¬ opcode(IR_p)  ∈branch,
  equalrelevantusedlocations (
               mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
               reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
               A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p) ,
  ¬ datadependent (reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                   A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p),
 ⟨equalresults# (mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                 reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                 A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p;
                 equalresults)⟩ equalresults = ff
⊢
```

`equalrelevantusedlocations` und `equalresults#` beziehen sich immer auf den nächsten zu bearbeitenden Befehl, denjenigen also, auf den der `PC` zeigt. Die zusätzlichen Bedingungen am Anfang der Sequenz bedeuten, daß *DLX* und DLX^p in einem für den Anfang eines Instruktionszyklus legalen Zustand befinden. Das heißt, die *DLX* befindet sich im Modus `FETCH` und die beiden vorhergehenden Befehle in der DLX^p-Pipe sind keine Sprunganweisungen (sonst wäre der aktuelle Befehl einer der für die DLX^p eingefügten `NOP`s, und diese werden hier ausgeschlossen).

Struktur des Beweises

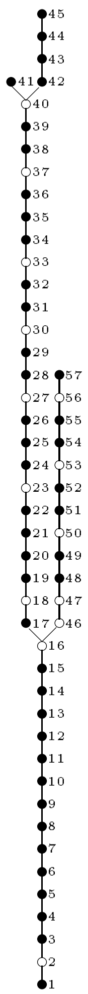
Für den Beweis des **DLXp-lemma-b** wurde zunächst der Aufruf von `DLX-cycle#` in `equalresults#` ausgeführt. Dadurch entstanden 14 Beweiszweige, die den verschiedenen Ablaufmöglichkeiten der unterschiedlichen Befehlstypen entsprechen. An jedem Blatt des entstandenen Beweisbaums ist nun zu zeigen, daß die DLX^p für diesen Fall das Resultat liefert, das auch von der *DLX* berechnet wurde.

Die Beweise dieser 14 Fälle wurden in 14 Lemmata ausgelagert, um den Beweis handhabbarer und übersichtlicher zu machen. Die Zuordnung zu den verschiedenen Fällen liefert Tabelle 1.

Hier soll exemplarisch der Beweis zu **branch-7-lemma**, also der Fall für `TRAP` beschrieben werden. Der Beweisbaum dazu ist in Abb. 1 zu sehen.

Die Struktur ist offensichtlich recht einfach. In den Schritten 1 bis 16 wurden in erster Linie „symbolic execution“ und die „split“-Heuristik verwendet, um aus der Definition von equalresults# die drei Aufrufe von DLX-step# zu isolieren, ohne sie jedoch auszuführen.

Abbildung 1: Beweisbaum für **branch-7-lemma**



Im Fall des TRAP-Befehls sind die „Resultate“ die Werte von PC und IAR nach dem dritten Zyklus. Es ist also zu zeigen, daß diese nach dem dritten Durchlauf durch DLX-cycle# den richtigen Wert haben. Dazu kann die „simulta-neous split“-Regel eingesetzt werden, siehe Abschnitt 4.2. Um jedoch durch die drei Zyklen hindurch die Berechnung des neuen Werts der beiden Register zurückzuverfolgen brauchen wir noch eine zusätzliche Information: daß nämlich der Wert von IR1 am Anfang des dritten Zyklus gerade der Befehl ist, auf den PC vor dem ersten gezeigt hat. Diese Information wird hier mittles der „cut“-Regel zur Sequenz hinzugefügt. Durch Anwendung der „simultaneous split“-Regel läßt sich der dritte Ast leicht schließen: man verfolgt, daß der Wert von IR1 vor dem dritten Zyklus aus dem Wert von IR vor dem zweiten Zyklus stammt, und diesem wiederum im ersten Zyklus mem_instr(PC) zugewiesen wird.

Im linken Ast kann diese Information dann benutzt werden, um den Wert von PC und IAR zu ermitteln. Die hinzugefügte Information über den Wert von IR erlaubt es jeweils, mittels „simultaneous simplify“ (siehe Abschnitt 4.1) die ASM-Regeln, d.h. Zweige des simultaneous Konstrukts zu entfernen, die die Behandlung anderer Befehlstypen in der aktuellen Pipeline-Stufe betreffen. Mit der „simultaneous split“-Regel können die ASM-Regeln für die anderen Pipeline-Stufen entfernt werden. Dadurch bleiben in jedem der drei Schritte nur noch die Zweige übrig, die einen Einfluß auf die Berechnung der beiden Register haben.

In Schritt 40 ist dann gezeigt, daß PC den richtigen Wert erhalten hat, nämlich die Adresse des Sprungziels. Weiterhin ist IAR, die Rücksprungadresse, auf next_p(next_p(next_p(PC))) gesetzt worden, was aufgrund der beiden folgenden NOP-Befehle die Adresse des nächsten eigentlichen Befehls nach dem TRAP ist.[4] Zu zeigen ist, daß dieser Wert gleich next(PC) ist, dem Wert, der von der *DLX* errechnet wurde. Aber dies ergibt sich aus einem Axiom der Spezifikation **static**, welches (unter anderem) besagt, daß

next_p(next_p(next_p(PC)))=next(PC),

falls PC auf einen JUMP- oder BRANCH-Befehl zeigt. Nach Einfügen dieses Axioms (Knoten 41) wird der Beweis geschlossen.

Die Beweise für die anderen Zweige laufen nach dem gleichen Prinzip. Etwas schwieriger waren jeweils die Befehle, die auf Registerinhalte zurückgreifen: Die gebrauchten Werte stehen eventuell erst im zweiten Zyklus (*ID*) in den Registern. Dies wird durch das Prädikat equalrelevantusedlocations sichergestellt, dessen Definition dann auch jeweils am Ende des Beweises benutzt wurde. Hier konnten simplifier-Regeln eingeführt werden, die einem den Umgang mit der vollständigen (recht umfangreichen) prädikatenlogischen Definition ersparen.

Eine andere Schwierigkeit stellten der LOAD- und der MOVS2I-Befehl dar, und zwar aufgrund der in [BM96] auf Seite 19 als irrelevant bezeichneten Fälle. Die Argumentation, aufgrund welcher diese Fälle „irrelevant“ sind, wäre nur schwer in unseren formalen Beweis einzubringen gewesen. Wir haben daher auf die Unterscheidung zwischen relevanten und irrelevanten benutzten Werten verzichtet und behaupten mit dem Prädikat equalrelevantusedlocations — welches also besser einfach „equalusedlocations“ heißen sollte — auch die Gleichheit der gelesenen Speicherstellen für LOAD bzw. des IAR für MOVS2I. Das macht die Definition des Prädikats natürlich erheblich komplizierter, da bis zu 3 Zyklen vorausgeschaut werden muß. Aber es war auch hierfür möglich, simplifier-Regeln zu erhalten. Damit verursachte das Lemma für MOVS2I dann keine zusätzlichen Probleme mehr. Für LOAD hingegen ist eine Reihe von Fallunterscheidungen nach den Typen und den beschriebenen Adressen der vorhergehenden Befehle nötig.

[4] next_p ist die Weiterzählfunktion für der Programmzähler der DLX^p

Tabelle 1: Größen der Zweige im Beweis von DLX^p-Lemma-b

Lemma-Name	Operation	Bew. Schritte	Interakt.	Zyklen
`branch-1-lemma`	`ALU` oder `SET` mit „immediate“ Operand	115	25	5
`branch-2-lemma`	`ALU` oder `SET` mit Register Operand	117	28	5
`branch-3-lemma`	`STORE`	116	24	4
`branch-4-lemma`	`LOAD`	199	52	5
`branch-5-lemma`	`MOVS2I`	118	23	5
`branch-6-lemma`	`MOVI2S`	80	16	3
`branch-7-lemma`	`TRAP`	65	14	3
`branch-8-lemma`	`BRANCH` mit erfüllter Bedingung	96	16	3
`branch-9-lemma`	`BRANCH` mit nicht erfüllter Bedingung	71	16	3
`branch-10-lemma`	`PLAINJ` mit „immediate“ Operand	52	14	3
`branch-11-lemma`	`JLINK` mit „immediate“ Operand	121	36	5
`branch-12-lemma`	`PLAINJ` mit Register Operand	68	15	3
`branch-13-lemma`	`JLINK` mit Register Operand	108	27	5
`branch-14-lemma`	`NOP`	22	4	1
`DLXp-lemma-b`	*DLX* Durchlauf & Lemmaanwendung	241	16	

Beweisgröße, Aufwand

In Tabelle 1 sind die Beweisgrößen der einzelnen Zweige angegeben.[5] Der Hauptbeweis erforderte neben dem Einfügen der 14 Lemmata nur noch zwei Interaktionen: einen „call“, der die Heuristik anfangs in Gang setzt und ein (nicht verzweigendes) „cut” im `NOP`-Zweig, das der simplifier aus technischen Gründen nicht selbst gefunden hatte.

Beim Vergleich mit der Anzahl von Zyklen, die jeder Befehl benötigt, stellt man fest, daß dies der Hauptfaktor in der Beweisgröße ist. Die beiden Zweige im Beispielbeweis werden dann entsprechend länger. Ein sekundärer Faktor ist die Anzahl der Operanden, die der Befehl aus Registern erhält, da durch diese Operanden im ersten Zyklus Fallunterscheidungen auftreten. Diese brauchen allerdings erst sehr spät im Beweis ausgeführt zu werden, was ihren Einfluß auf die Beweisgröße beschränkt.

Der längste der Beweise ist der für den `LOAD`-Fall, da dieser Befehl volle 5 Zyklen benötigt, seinen Operanden (die Adresse) aus einem Register bezieht und außerdem die Komplikationen mit sich bringt, auf die gegen Ende des letzten Abschnitts eingegangen wurde.

Insgesamt erforderte der Beweis für das Lemma `DLXp-lemma-b` also etwa 1.600 Beweisschritte, davon etwa 330 Interaktion. Ohne die Split-Regel wären laut der Abschätzungen in der Fußnote auf Seite 14 fast eine Million Beweisschritte nötig gewesen, wobei nicht zu garantieren ist, daß jeder der etwa 50.000 entstehenden Zweige automatisch geschlossen werden könnte!

Die Ersparnis durch die „simultaneous simplify“-Regel (siehe Ende von Abschnitt 4.1) hängt natürlich stark von der Größe der vorkommenden `simultaneous`-statements ab. Allerdings dürfte eine Verkleinerung auf ein fünftel auch für den gesamten Beweis eine realistische Schätzung darstellen.

Die Beweise der 14 Zweige erforderten zusammen weniger als 20 Stunden, wobei allerdings die Zeit für Korrekturen an der Formalisierung und Formulierung von Simplifierregeln für die Spezifikationen `datadependent` und `equalrelevant` nicht hinzugerechnet wurden.

[5] In der Projektstatistik des KIV-Systems stehen hier und bei `branch-2-lemma` doppelt so große Werte. Das liegt daran, daß aus beweistechnischen Gründen jeweils zu Beginn eine Aufspaltung nach `ALU` oder `SET` Operation vorgenommen wurde. Einer der Zweige wurde dann wie beschrieben geschlossen. Der andere Ast konnte durch ein Replay des ersten geschlossen werden, verursachte also keine zusätzliche Arbeit. KIV zählt diese durch Replay duplizierten Interaktionen jedoch mit.

4 Beweistaktiken für ASMs

Im Laufe der Fallstudie wurden neue Beweistaktiken zur effizienteren Verifikation von Aussagen über `simultaneous`-Konstrukte identifiziert, im KIV-System implementiert und eingesetzt.

4.1 Die „Simultaneous Simplify“-Regel

Zu Beginn der Fallstudie standen in KIV zur Behandlung von `simultaneous`Anweisungen lediglich zwei Typen von Inferenzregeln zur Verfügung (siehe [Sch95]):

- Die „simultaneous update“-Regel. Mit dieser kann eine `simultaneous`-Anweisung, deren Zweige Zuweisungen sind, symbolisch ausgeführt werden.
- Die „simultaneous push“-Regel, die auf den folgenden Äquivalenzen beruht:

$$\texttt{simultaneous } \alpha \texttt{ and abort} \equiv \texttt{abort}$$

$$\texttt{simultaneous } \alpha \texttt{ and skip} \equiv \alpha$$

$$\texttt{simultaneous } \alpha \texttt{ and if } e \texttt{ then } \beta_1 \texttt{ else } \beta_2$$
$$\equiv \texttt{if } e \texttt{ then simultaneous } \alpha \texttt{ and } \beta_1 \texttt{ else simultaneous } \alpha \texttt{ and } \beta_2$$

Das Lemma **`DLXp-lemma-b`** wäre unter alleiniger Verwendung dieser Regeln zu beweisen gewesen, allerdings nur mit unvertretbarem Aufwand. Der erste Versuch, dieses Lemma zu beweisen erfolgte durch naive symbolische Ausführung von `equalresults#`. In dieser Prozedur wird erst der komplette *DLX*-Zyklus ausgeführt. Dann werden entsprechend der verschiedenen Befehlstypen so viele DLX^p-Zyklen ausgeführt, bis die zu dem Befehl gehörigen „result locations“ geschrieben worden sind. Das sind je nach Befehlstyp bis zu 5 Durchläufe von `DLXp-cycle#`.

Man sehe sich etwa die Definition von `DLX-step#` an. Die ASM-Regeln sind die Zweige eines einzigen, großen `simultaneous`-Konstrukts. Wenn hier `mode=FETCH` bekannt ist, kann die der ersten Regel entsprechende `if`-Anweisung „gepusht“ werden, das heißt aus

```
mode = FETCH,
⟨simultaneous
   if mode = FETCH then
    IR ← mem(PC),
    PC ← next(PC),
    mode := OPERAND
  and
   if mode = OPERAND then
    ...
   and ... ⟩ ...
```

wird

```
mode = FETCH,
⟨if mode = FETCH then
   simultaneous
      IR ← mem(PC)
   and
      PC ← next(PC)
   and
      mode := OPERAND
   and
    if mode = OPERAND then
      ...
   and ...
  else
   simultaneous
    skip
   and
    if mode = OPERAND then
      ...
   and ... ⟩ ...
```

Hier kann freilich die Regel „if positive“ angewandt werden und es bleibt

$$\begin{array}{l} mode = FETCH, \\ \langle\texttt{simultaneous} \\ \quad\quad IR \leftarrow mem(PC) \\ \quad \texttt{and} \\ \quad\quad PC \leftarrow next(PC) \\ \quad \texttt{and} \\ \quad\quad mode := OPERAND \\ \quad \texttt{and} \\ \quad\; \texttt{if } mode = OPERAND \texttt{ then} \\ \quad\quad \ldots \\ \quad\; \texttt{and} \ldots\rangle\ldots \end{array}$$

Man bemerke, daß nun alle anderen Zweige trivialerweise nicht ausführbar sind, da sie einen anderen Modus als `FETCH` verlangen. Um die Zuweisungen auszuführen, muß man mit „simultaneous push“ jede einzelne `if`-Anweisung nach außen holen, mittels „if negative“ eliminieren und das verbleibende `skip` mit einem abermaligen „simultaneous push'“ entfernen. Das sind 3 Regelanwendungen für jede der 14 verbleibenden Regeln, also 42 Beweisschritte, um auf eine simulatane Zuweisung zu kommen. Man bemerke, daß diese Aufblähung überall auftritt, wo mehrere simultane bedingte Anweisungen auftreten, deren Bedingungen sich gegenseitig ausschließen. Das ist wiederum typisch für die Beschreibung eines größtenteils sequentiellen Algorithmus durch Regeln: ein für ASMs typischer Fall also.

Um dieses Problem zu lösen, haben wir uns entschieden, eine Makroregel zu implementieren, die in einem Schritt das Resultat der oben beschriebenen Operationen liefert. Konkret erledigt die Regel folgendes, bis nichts davon mehr möglich ist:[6]

- Tritt in einem `simultaneous`-Konstrukt der Sequenz ein `skip` auf, wird dieses entfernt (Ein „leeres“ `simultaneous` entspricht `skip`)
- Tritt in einem `simultaneous`-Konstrukt der Sequenz ein `abort` auf, wird das ganze Statement durch `abort` ersetzt.
- Tritt in einem `simultaneous`-Konstrukt der Sequenz ein `if` auf, dessen Bedingung aus den PL-Formeln der Sequenz abgeleitet werden kann, so wird das `if` durch dessen positiven (`then`) Zweig ersetzt.
- Tritt in einem `simultaneous`-Konstrukt der Sequenz ein `if` auf, dessen *negierte* Bedingung aus den PL-Formeln der Sequenz abgeleitet werden kann, so wird das `if` durch dessen negativen (`else`) Zweig ersetzt.

Die Tests für die `if`-Anweisungen werden genauso wie bei der „if positive“ und „if negative“ Regel durchgeführt. Hier ist zu bemerken, daß natürlich nicht immer alle `if`-Anweisungen anhand der vorhandenen PL-Formeln tatsächlich eliminiert werden können. Es werden aber je zwei „logic tests“ pro `if` aufgerufen, und zwar bereits beim Test, ob die Regel anwendbar ist, und bei der Anwendung zur Zeit nochmals.[7] Diese Tests sind leider ziemlich rechenaufwendig. Daher wäre es eventuell sinnvoll, die „simultaneous simplify“ Regel immer anzubieten, wenn ein `simultaneous`-statement in der Sequenz vorkommt — ohne zu testen ob es sich auch wirklich vereinfachen läßt. Das entspräche der Implementierung der PL-Simplifier-Regeln. Noch besser wäre eine Art simultaner „logic test“ für mehrere Formeln.

Trotz dieser Effizienzprobleme hat sich die Regel bewährt und wurde in die „symbolic execution“-Heuristik aufgenommen: Wenn keine `simultaneous`-Konstrukte vorhanden sind, braucht sie

[6] Betrachtet werden alle diamond- und box-Formeln der Sequenz, deren übergeordnetes Programmkonstrukt ein `simultaneous`-statement ist. Tiefer geschachtelte `simultaneous`-Konstrukte werden nicht betrachtet.

[7] Dies liegt an Details in der Datenverwaltung in KIV. Es wäre programmiertechnisch aufwendig, die ersten Testergebnisse wiederzuverwenden.

keine zusätzliche Zeit, und wenn welche vorhanden sind, zeigt die Erfahrung, daß das Warten auf die Simplifikation sich meistens lohnt. Wenn die Simplifikation von `if`s nicht nötig ist, kann man diese mit der Heuristik-Option „No positive/negative tests" ausschalten.

Zusätzlich wurden zwei einfache Heuristiken eingebaut. Die eine, „sim. update" führt simultane Updates aus, falls solche in der Sequenz vorkommen. Diese Heuristik ist normalerweise ebenso unproblematisch wie die gewöhnliche „assign"-Regel. Die andere Heuristik beruht auf der Einsicht, daß diejenigen `if` Anweisungen, die nicht durch die „simultaneous simplify"-Regel eliminiert werden können, durch push behandelt werden müssen (siehe jedoch den folgenden Abschnitt), und daß die Reihenfolge, in der verschiedene `if`s gepusht werden, oft irrelevant ist. Die „sim. push"-Heuristik wendet also irgendeinen push auf irgendeine `simultaneous`-Anweisung an, falls dies möglich ist, wobei `abort` und `skip` aus Effizienzgründen bevorzugt werden.

Nach Einrichtung dieser Heuristiken konnte vollautomatisch ein vollständiger Zyklus der *DLX* symbolisch ausgeführt werden, und zwar mit Aufspaltung nach den 14 möglichen Kontrollflüssen, die den Befehlstypen entsprechen. Dieser Teil des Beweises benötigt etwa 220 Beweisschritte. Ohne die „simultaneous simplify"-Regel wären dazu etwa 1000 Schritte[8] mehr nötig gewesen. Die neue Regel konnte also die Größe des Beweises um etwa 80% reduzieren.

Programmierung, Einbau ins KIV-System und Testen der „simultaneous simplify"-Regel erforderten etwa 12 Arbeitsstunden.

4.2 Die „Simultaneous Split"-Regel

Nachdem ein Zyklus der *DLX* nun symbolisch ausgeführt werden konnte, ergaben sich bei der Ausführung der drei bis fünf DLX^p-Schritte allerdings erneut Probleme.

Beim Betrachten der Prozedur `equalresults#` bemerkt man, daß bei weitem nicht alle Zustandsvariablen, die durch die DLX^p-Schritte geändert werden, relevant sind. Es kommt für das Resultat der Prozedur nur auf wenige Werte an, eben die Ergebnisse des ausgeführten Befehls.

Werden Durchläufe von `DLXp-step#` einfach symbolisch ausgeführt, so entstehen Beweisverzweigungen für alle möglichen Kontrollflüsse der *gesamten* DLX^p. Es werden also auch Verzweigungen erzeugt für verschiedene Ablaufvarianten von Befehlen, die schon vor dem betrachteten Befehl in die Pipe kamen oder später folgen, deren Ablauf also den aktuellen Befehl gar nicht beeinflussen kann. Das ist im Prinzip legitim, denn es soll ja gerade *bewiesen* werden, daß die Bearbeitung der anderen Befehle in der Pipe auf den betrachteten keinen Einfluß hat.

Allerdings sieht man schnell, daß die Anzahl der Verzweigungen enorm wird. Ein Versuch, den Ast für ein TRAP — die Resultate sind der veränderte `PC` sowie das `IAR` und nach drei Zyklen vorhanden — durch naive symbolische Ausführung zu schließen, mußte abgebrochen werden. Zwar konnten die ersten Zweige automatisch geschlossen werden, aber es entstanden so viele Verzweigungen, daß eine *optimistische* Schätzung anhand der Struktur des Teilbeweises eine Gesamtgröße von etlichen tausend Knoten erwarten ließ.[9] Dabei sind die anderen 13 Äste zum Teil noch deutlich aufwendiger, da hier fünf Zyklen durchgerechnet werden müssen.

Unser Ansatz zur Lösung des Problems besteht darin, einfachere, *syntaktische* Kriterien zu finden, die sicherstellen, daß Teile einer Berechnung den eigentlich interessanten Teil nicht beeinflussen. Solche irrelevanten Teile können dann abgespalten werden und brauchen nicht weiter betrachtet werden.

[8] Etwa 40 Schritte gespart für jede Anwendung der Regel, die dem beschriebenen Fall entspricht. Etwa 25 solche Anwendungen im Beweis.

[9] Wir haben angefangen, das **`branch-7-lemma`** zu beweisen. Dabei haben wir die 14 am weitesten links liegenden Zweige automatisch schließen lassen. Es stellte sich heraus, daß hier bei jeder Verzweigung der rechts folgende Teilbeweis mindestens so groß wie der links folgende ist. Um den Zweig zu schließen, von dem die besagten 14 ausgingen, waren von der Wurzel dieses Teilbeweises ausgehend etwa 220 Schritte nötig. Unterhalb dieses Teilbeweises waren allerdings 4 Verzweigungen. Unter der Annahme, daß auch im restlichen Teilbeweis die Teilbeweisgrößen „rechtslastig" sind ergeben sich mindestens $2^4 \cdot 220 \approx 3.500$ Schritte. Für einen Zweig, in dem 5 Zyklen durchlaufen werden ergab ein ähnlicher Test $2^{11} \cdot 80 \approx 160.000$ Schritte.

Beispiel:

Sei n eine Variable über den natürlichen Zahlen.

$$\begin{array}{l} \langle \texttt{begin} \\ \quad A; \quad \text{— komplexe Berechnung} \\ \quad n := n + 1 \\ \; \texttt{end} \rangle \, n = 0 \\ \vdash \end{array}$$

Es ist klar, daß dies eine gültige Sequenz ist. Nach jedem terminierenden Lauf durch den Programmteil — was auch immer in A geschehen mag — wird $n > 0$ gelten. Symbolische Ausführung wird mit der möglicherweise sehr aufwendigen Analyse von A beginnen. Erst danach wird die Zuweisung ausgeführt und der Ast geschlossen.

Aber in KIV existiert die split-Regel. Wenden wir sie auf die Sequenz an, ergibt sich:

$$\begin{array}{l} \langle A \rangle \, u_0 = v_0 \wedge u_1 = v_1 \wedge \ldots \wedge n = m, \\ \langle m := m + 1 \rangle \, m = 0 \\ \vdash \end{array}$$

Dabei werden also neue Variablen eingeführt für die Werte der in A zugewiesenen Variablen nach der Ausführung von A. Im allgemeinen kann auch n dazugehören. Der Trick besteht nun darin, sofort die Zuweisung auszuführen und damit den Ast zu schließen.

Was ist hier geschehen? Das Programm wurde gewissermaßen „von hinten“ ausgeführt. In der Formel hinter dem diamond wird nur n verwendet. Also können wir das Programm zerlegen und nach der Stelle suchen, an der der am Ende aktuelle Wert von n gesetzt wird. Daraus ergeben sich dann neue Bedingungen. Solch eine „Rückwärtsausführung“ ist nicht immer möglich, es kann ja mehrere Stellen im Programm geben, etwa in verschiedenen Zweigen einer `if`-Anweisung, an denen die letzte Zuweisung an die Variable geschieht. Aber wenn sie möglich ist, hat sie den Vorteil, daß nur die Teile des Programms betrachtet werden, die für das Ergebnis interessant sind.

Für das `simultaneous`-Konstrukt ergibt sich nun eine ähnliche Möglichkeit. Man betrachte folgende Sequenz:

$$\begin{array}{l} \langle \texttt{simultaneous} \\ \quad A \quad \text{— komplexe Berechnung} \\ \; \texttt{and} \\ \quad n := n + 1 \\ \rangle \, n = 0 \\ \vdash \end{array}$$

Dabei soll die Variable n im Programmteil A nicht zugewiesen werden. Man wünscht sich, eine Aufspaltung des `simultaneous`-Konstrukts vornehmen zu können, um

$$\begin{array}{l} \langle A \rangle \, \mathit{true}, \\ \langle n := n + 1 \rangle \, n = 0 \\ \vdash \end{array}$$

zu erhalten. Zu diesem Zweck wurde die „simultaneous split“-Regel eingeführt. Sie kann nur auf Diamond-Formeln auf der linken Seite der Sequenz angewandt werden. Auf Diamond-Formeln rechts muß zunächst die gewöhnliche „split right“-Regel angewandt werden, die allerdings im Fall nicht-deterministischer Programme zu Informationsverlusten führen kann. Eine Variante für Box-Formeln ließe sich durch Dualität erarbeiten, wurde jedoch nicht benötigt.

Die „simultaneous split“-Regel soll hier zuerst für den einfachen Fall eines `simultaneous`-Konstrukts mit nur zwei Zweigen erläutert werden. Gegeben sei also eine Sequenz

$$\langle \texttt{simultaneous} \; \alpha \; \texttt{and} \; \beta \rangle \, \phi, \Gamma \vdash \Delta$$

Falls die in α und β (syntaktisch!) zugewiesenen Variablen disjunkt sind, also

$$asgvars(\alpha) \cap asgvars(\beta) = \emptyset,$$

ist die folgende Regel zulässig:

$$\frac{\langle\alpha\rangle\, \underline{x} = \underline{u},\ \langle\beta\rangle\, \underline{y} = \underline{v},\ \phi[\underline{x} \leftarrow \underline{u},\ \underline{y} \leftarrow \underline{v}],\ \Gamma \vdash \Delta}{\langle \texttt{simultaneous}\ \alpha\ \texttt{and}\ \beta\rangle\, \phi,\ \Gamma \vdash \Delta},$$

wobei

$$\begin{aligned}
&\underline{x} = asgvars(\alpha) \cap vars(\phi),\\
&\underline{y} = asgvars(\beta) \cap vars(\phi),\\
&\underline{u} \text{ bzw. } \underline{v} \text{ neue Variablen für } \underline{x} \text{ bzw. } \underline{y}
\end{aligned}$$

Die tatsächliche Implementierung unterscheidet sich von dieser einfachsten Form in einigen wichtigen Punkten:

- In der Praxis tauchen `simultaneous`-Konstrukte mit vielen Zweigen auf. Wegen Assoziativität und Kommutativität des `simultaneous` können die Zweige zu minimalen Blöcken zusammengefasst werden, so daß keine Variable in mehr als einem Block zugewiesen wird. Dazu wird der ungerichtete Graph betrachtet, der entsteht, wenn man jeden Programmzweig des `simultaneous` als Knoten des Graphen betrachtet und eine Kante zwischen zwei Knoten zieht, falls die zugehörigen Programmzweige eine gemeinsame zugewiesene Variable besitzen. Die minimalen abspaltbaren Blöcke entsprechen dann gerade den Zusammenhangskomponenten dieses Graphen.

- Im oben angegebenen Beispiel wird die Formel $\langle A\rangle\, true$ für den Beweis nicht mehr gebraucht. Diese Erzeugung von überflüssigen Formeln, die, um die Übersichtlichkeit zu wahren, im nächsten Beweisschritt aus der Sequenz entfernt werden sollten, hat sich in unseren ersten Versuchen mit der Regel als recht lästig herausgestellt. Typischerweise kam es — wie im Beispiel — nur auf einen oder zwei der Zweige an.

Die derzeitige Implementierung der Regel berechnet zu einem `simultaneous`-Statement also zunächst die besagten minimalen Blöcke. Dann können interaktiv diejenigen Blöcke ausgewählt werden (mindestens einer), die im Beweis beibehalten werden sollen. Insgesamt sieht die Regel also wie folgt aus, wenn α_1 bis α_n die ausgewählten Blöcke sind und α_0 die Zusammenfassung der restliche Blöcke:

$$\frac{\langle\alpha_1\rangle\, \underline{x}_1 = \underline{u}_1,\ \ldots\ \langle\alpha_n\rangle\, \underline{x}_n = \underline{u}_n,\ \phi[\underline{x}_0 \leftarrow \underline{u}_0, \ldots \underline{x}_n \leftarrow \underline{u}_n],\ \Gamma \vdash \Delta}{\langle \texttt{simultaneous}\ \alpha_0\ \texttt{and}\ \ldots\ \texttt{and}\ \alpha_n\rangle\, \phi,\ \Gamma \vdash \Delta},$$

wobei

$$\begin{aligned}
&\underline{x}_i = asgvars(\alpha_i) \cap vars(\phi) \quad \text{für } 0 \le i \le n,\\
&\underline{u}_i \text{ neue Variablen für } \underline{x}_i,\\
&asgvars(\alpha_i) \cap asgvars(\alpha_j) = \emptyset \quad \text{für } 0 \le i < j \le n,
\end{aligned}$$

Mit Hilfe dieser Regel konnte das Lemma **`DLXp-lemma-b`** vollständig bewiesen werden.

Programmierung, Einbau ins KIV-System und Testen der „simultaneous split"-Regel brauchten etwa 15 Arbeitsstunden.

5 Diskussion

In der hier dokumentierten Fallstudie haben wir das KIV-System zur Verifikation der DLX-Pipelining-Architektur eingesetzt. Als Grundlage wurden dazu die von Börger & Mazzanti [BM96, BM97] gegebenen ASM-Spezifikationen benutzt. Genauer wurde die erste Stufe der dort beschriebenen sukzessiven Verfeinerung formalisiert und verifiziert. Diese beschreibt den Übergang von einem sequentiellen Maschinenmodell zu einer Pipelining-Architektur unter der Annahme, daß keine Datenabhängigkeiten auftreten und durch einen Compiler Kontrollabhängigkeiten vermieden werden. Sie kann als repräsentativ auch für die weiteren Stufen der in [BM97] beschriebenen Verifikation gelten, da die Grundstruktur der Beweise in allen Stufen übereinstimmt, und der Anteil einzelner Regeln im Vergleich zum Gesamtbeweis relativ gering ist.

Neben der WAM-Fallstudie [SA97] ist dies ein weiterer Beleg dafür, daß sich das KIV-System prinzipiell für die Formalisierung und Verifikation von ASM-Spezifikationen eignet. Dies soll jedoch nicht darüber hinwegtäuschen, daß die Formalisierung der nachzuweisenden Aussagen und der Beweise mit nicht unerheblichem Aufwand verbunden ist (wie sich auch bereits in der WAM-Fallstudie zeigte). Einerseits sind gerade die strukturierte Aufbereitung der Beweisaufgaben und die in der ASM-Methodik propagierte Verfeinerungstechnik sehr wertvoll und können bei der Formalisierung beibehalten werden. Andererseits sind manche Beweisargumente derart abstrakt und informell, daß sie einer maschinellen Verifikation nicht direkt zugänglich sind, so daß die Beweise gleichsam neu geführt werden müssen.

Je nach Anwendung sollte man also abwägen, ob die Formalisierung den Aufwand lohnt. Formalisierung bringt zum einen den Vorteil, Spezifikationen wie auch Beweise maschinell verarbeiten zu können (z. B. Visualisierung, Analyse, automatische Dokumentenerstellung und Wiederverwendung). Ein anderer wesentlicher Punkt ist, daß die mechanische Verifikation größtmögliche Sicherheit vor Fehlern in Beweisen bietet — Deduktionssysteme sind kritische aber objektive Kollegen, die es zu überzeugen gilt.

Tatsächlich wurden bei der hier beschriebenen Fallstudie folgende — wenn auch kleinere — Unzulänglichkeiten in der Spezifikation und informellen Beweisführung in [BM96] aufgedeckt:

- Die Tabelle 2 der kritischen Zeitpunkte für Eingangsgrößen gibt falsche Pipeline-Stufen an.
- Im Beweis desselben Lemmas wurde implizit auf die Induktionsvoraussetzung von Teil a zurückgegriffen.
- Der Ausschluß selbstmodifizierenden Codes wurde nicht formalisiert.
- Das Verhalten der *DLX* bei Auftreten von `NOP`-Befehlen war nicht spezifiziert.
- In der formalen Spezifikation war nicht ausgeschlossen, daß das Ziel von Befehlen ein internes Register des Prozessors sei.
- Im Beweis des DLX^p-Lemmas, Teil b, wurde ein nie auftretender Fall betrachtet.[10]
- Die verlangten Initialbedingungen $PC1 = undef$ und $C1 = undef$ sind unnötig.

Schließlich wurden im Laufe der Fallstudie spezielle Beweistaktiken und -heuristiken identifiziert und im KIV-System implementiert, welche die Verifikation von ASM-Spezifikationen deutlich effizienter machen (vgl. Abschnitt 4).

[10] Genauer wurde bei **Irrelev 1** in [BM96, BM97] nicht berücksichtigt, daß die nächsten zwei nach `TRAP` folgenden Befehle `NOP`s sein müssen.

Literatur

[BDR94] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and compiler correctness. In U. Montanari and E.-R. Olderog, editors, *Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*. North Holland, 1994.

[BM96] E. Börger and S. Mazzanti. A correctness proof for pipelining in RISC architectures. Technical Report 96-22, DIMACS, 1996.

[BM97] E. Börger and S. Mazzanti. A practical method for rigourously controllable hardware design. In J.P. Bowen, M.G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*. Springer, Berlin, 1997.

[Bör95] E. Börger. Why use evolving algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiedermann, editors, *SOFSEM '95: Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271, 1995.

[BR95] E. Börger and D. Rosenzweig. The WAM — definition and compiler correctness. In L.C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Series in Computer Science and Artificial Intelligence. North–Holland, 1995.

[Gur94] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[Har79] D. Harel. *First Order Dynamic Logic*. LNCS. Springer, Berlin, 1979.

[HRS89] M. Heisel, W. Reif, and W. Stephan. A dynamic logic for program verification. In *Logical Foundations of Computer Science*, volume 363 of *LNCS*, pages 134–145. Springer, Berlin, 1989.

[Hug94] J. Huggins. Kermit: Specification and verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[Pus96] C. Pusch. Verification of compiler correctness for the WAM. In J. Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOL'96)*, volume 1125 of *LNCS*. Springer, Berlin, 1996.

[Rei92] W. Reif. The KIV-system: Systematic construction of verified software. In *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *LNCS*. Springer, Berlin, 1992.

[SA97] G. Schellhorn and W. Ahrendt. Reasoning about abstract state machines: The WAM case study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[Sch95] A. Schönegge. Extending dynamic logic for reasoning about evolving algebras. Technical Report 49/95, Universität Karlsruhe, Fakultät für Informatik, 1995.

A Formalisierung der Hauptbeweisverpflichtung

In diesem Abschnitt ist die komplette KIV-Formalisierung mit Spezifikationen, Implementierung der Maschinen und der Hauptbeweisverpflichtung (Correctness Theorem) aufgeführt.

Abschnitt B zeigt dann diejenigen Teile, die zur Formulierung der beiden Lemmata gebraucht wurden, auf denen der Beweis des Correctness Theorem beruht.

Es sind also nur die in diesem Abschnitt aufgeführten Teile relevant für die Aussage des bewiesenen Theorems. Dagegen stellt Abschnitt B die nur zum Beweisen verwendeten Zusatzstrukturen dar.

A.1 Spezifikation `static`

```
static=
specification

sorts
        word,
        address,
        instruction,
        instr_addr,
        modes,
        register,
        operation;

constants
        fetch, operand, alu, alu', mem_addr, iarmove, jumps,
        mem_acc, write_back, pass_b_to_mdr, subword : modes;

        r0, r31 : register;
        word0 : word;

(: static :) functions
        next      ( instr_addr )  : instr_addr ;
        next_p    ( instr_addr )  : instr_addr ;
        opcode    ( instruction ) : operation ;
        dest      ( instruction ) : register ;
        fstop     ( instruction ) : register ;
        scdop     ( instruction ) : register ;
        ival      ( instruction ) : word ;
        mem_instr ( instr_addr )  : instruction ;

        alu-op  ( operation , word , word ) : word ;    (: alu operation :)
        alu-sw  ( operation , word ) : word ;           (: subword :)

        word->instr_addr ( word ) : instr_addr ;
        instr_addr->word ( instr_addr ) : word ;

        . +word . ( word , word ) : word ;
        . +pc . ( instr_addr, word ) : instr_addr ;

        next-reg ( register ) : register ;
        next-word ( word ) : word ;
        word->instruction ( word ) : instruction ;
        word->address ( word ) : address ;
```

```
predicates
        iop(operation);
        condition(word);

        . ∈alu          (operation) ;
        . ∈set          (operation) ;
        . ∈load         (operation) ;
            ...

        . ∈nop          (operation) ;


variables
        wrd : word;  add : address;
        ins : instruction; ins_add : instr_addr;
        mode : modes ;
        ri : register ; op : operation ;

axioms
        (: for modes :)
        modes freely generated by
                fetch, operand, alu, alu', mem_addr, iarmove, jumps,
                mem_acc, write_back, pass_b_to_mdr, subword;

        (: generation axioms :)
        register generated by r0, next-reg;
        word generated by word0, next-word;
        instruction generated by word->instruction;
        address generated by word->address;
        instr_addr generated by word->instr_addr;
        operation generated by opcode;

        next-reg(r31) = r0,
        opcode(ins) ∈jlink → dest(ins) = r31,

        (: correspondence between next and next_p functions :)
             opcode (mem_instr (ins_add)) ∈jump
           ∨ opcode (mem_instr (ins_add)) ∈branch
        → (  opcode(mem_instr(next_p(ins_add))) ∈nop
           ∧ opcode(mem_instr(next_p(next_p(ins_add)))) ∈nop
           ∧ next_p(next_p(next_p(ins_add))) = next(ins_add) ),

             ¬ opcode (mem_instr (ins_add)) ∈jump
           ∧ ¬ opcode (mem_instr (ins_add)) ∈branch
        → next_p(ins_add) = next(ins_add),


        (: operation types :)
        op ∈jump ↔ op ∈plainj ∨ op ∈jlink ∨ op ∈trap,

        (: any operation has one of the types :)
        op ∈alu ∨ op ∈set ∨ op ∈load ∨ op ∈store ∨ op ∈movs2i ∨
        op ∈movi2s ∨ op ∈jump ∨ op ∈branch ∨ op ∈nop,
```

```
        (: various operation types are disjoint :)
        op ∈alu → ¬ (              op ∈set    ∨ op ∈load   ∨ op ∈store ∨
                        op ∈movs2i ∨ op ∈movi2s ∨ op ∈plainj ∨ op ∈jlink ∨
                        op ∈trap   ∨ op ∈branch ∨ op ∈nop ),
        op ∈set → ¬ (   op ∈alu    ∨              op ∈load   ∨ op ∈store ∨
                        op ∈movs2i ∨ op ∈movi2s ∨ op ∈plainj ∨ op ∈jlink ∨
                        op ∈trap   ∨ op ∈branch ∨ op ∈nop ),

        ...

        op ∈nop → ¬ (   op ∈alu    ∨ op ∈set    ∨ op ∈load   ∨ op ∈store ∨
                        op ∈movs2i ∨ op ∈movi2s ∨ op ∈plainj ∨ op ∈jlink ∨
                        op ∈trap   ∨ op ∈branch           )

end specification
```

A.2 Spezifikation `dynamic` und endliche Funktionen

```
dom+cod =
specification
      sorts
         domain ,
         codomain ;
      variables
         x : domain ;
         y : codomain ;
end specification

finite-fun =
generic specification
   parameter dom+cod
   target    sorts
                fun ;
             functions
                constfun(domain) : fun ;
                apply(fun, domain) : codomain ;
                update(fun, domain, codomain) : fun ;
             variables
                func : fun ;
                x0 : domain ;
         axioms
            fun generated by constfun, update ;
         apply(update(func, x, y), x) = y ,
         x0 ≠ x → apply(update(func, x, y), x0) = apply(func, x0)
end generic specification

register->word =
actualize finite-fun
with static
by morphism domain -> register ,  codomain -> word ,  x -> ri ,
            x0 -> rj ,  y -> wrd ,  constfun -> const-reg ,
            apply -> apply-reg ,  update -> update-reg ,
            fun -> register->word ,  func -> reg ,
end actualize
```

```
address->word =
actualize finite-fun
with static
by morphism domain -> address ,  codomain -> word ,  x -> add ,
            x0 -> add0 ,  y -> wrd ,  constfun -> const-mem ,
            apply -> apply-mem ,  update -> update-mem ,
            fun -> address->word ,  func -> mem ,
end actualize

dynamic =
enrich static, register->word, address->word with
variables

        mem :  address->word ;
        reg :  register->word ;
end enrich
```

A.3 Spezifikation `datadependent` (Prädikat datadependent)

Abkürzungen

```
*opcode  := opcode(mem_instr(PC_p))
*fstop   := fstop(mem_instr(PC_p))
*scdop   := fstop(mem_instr(PC_p))
*opc     := opcode(IR_p)
*opc1    := opcode(IR1_p)
*opc2    := opcode(IR2_p)
*dst     := dest(IR_p)
*dst1    := dest(IR1_p)
*dst2    := dest(IR2_p)
```

Spezifikation

```
datadependent =
enrich dynamic with

  (: ... siehe Anhang B.1 ... :)

  predicates datadependent (register->word,     (: reg_p          :)
                            address->word,      (: mem_p          :)
                            instruction,        (: IR_p           :)
                            instruction,        (: IR1_p          :)
                            instruction,        (: IR2_p          :)
                            instruction,        (: IR3_p          :)
                            instr_addr,         (: PC_p           :)
                            instr_addr,         (: PC1_p          :)
                            word, word,         (: A_p,    B_p    :)
                            word, word,         (: C_p,    C1_p   :)
                            word, word,         (: SMDR_p, LMDR_p :)
                            address, word);     (: MAR_p,  IAR_p  :)


  variables     IR_p, IR1_p, IR2_p, IR3_p       : instruction;
                PC_p, PC1_p                     : instr_addr;
                A_p, B_p, C_p, C1_p             : word;
```

```
                IAR_p, SMDR_p, LMDR_p          : word;
                MAR_p                          : address;
                reg_p                          : register->word;
                mem_p                          : address->word;

  axioms

    (: ... siehe Anhang B.1 ... :)

    datadependent (reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                   A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
    ↔ (      (*opcode ∈jump ∨ *opcode ∈branch)
          ∧ (   (  *fstop = *dst2
                ∧ (*opc2 ∈alu ∨ *opc2 ∈set ∨ *opc2 ∈movs2i ∨ *opc2 ∈jlink
                   ∨ *opc2 ∈load))
             ∨ (  *fstop = *dst1
                ∧ (*opc1 ∈alu ∨ *opc1 ∈set ∨ *opc1 ∈movs2i ∨ *opc1 ∈jlink
                   ∨ *opc1 ∈load))
             ∨ (  *fstop = *dst)
                ∧ (*opc ∈alu ∨ *opc ∈set ∨ *opc ∈movs2i ∨ *opc ∈jlink
                   ∨ *opc ∈load))
       ∨  ¬ *opcode ∈jump ∧ ¬ *opcode ∈branch
          ∧ (   (  (*fstop = *dst2 ∨ *scdop = *dst2)
                ∧ (*opc2 ∈alu ∨ *opc2 ∈set ∨ *opc2 ∈movs2i ∨ *opc2 ∈jlink
                   ∨ *opc2 ∈load))
             ∨ (  (*fstop = *dst1 ∨ *scdop = *dst1)
                ∧ (*opc1 ∈alu ∨ *opc1 ∈set ∨ *opc1 ∈movs2i ∨ *opc1 ∈jlink
                   ∨ *opc1 ∈load))
             ∨ (  (*fstop = *dst ∨ *scdop = *dst))
                ∧ (*opc ∈alu ∨ *opc ∈set ∨ *opc ∈movs2i ∨ *opc ∈jlink
                   ∨ *opc ∈load))
     )
end enrich
```

A.4 Implementierung der *DLX*

Abkürzungen

```
*opcode := opcode(IR)
*fstop  := apply-reg(reg,fstop(IR))
*scdop  := apply-reg(reg,scdop(IR))
*dest   := apply-reg(reg,dest(IR))
*ival   := ival(IR)
```

Regeln

```
*DLX-FETCH-rule :=
if mode = fetch then
  simultaneous
    IR := mem_instr(PC)
  and
    PC := next(PC)
  and
    mode := operand
```

```
*DLX-OPERAND-rule :=
if mode = operand then
  simultaneous
    A := *fstop
  and
    B := *scdop
  and
    if *opcode ∈alu ∨ *opcode ∈set then
      mode := alu
  and
    if *opcode ∈load ∨ *opcode ∈store then
      mode := mem_addr
  and
    if *opcode ∈movs2i ∨ *opcode ∈movi2s then
      mode := iarmove
  and
    if *opcode ∈jump ∨ *opcode ∈branch then
      mode := jumps
  and
    if *opcode ∈nop then
      mode := fetch

*DLX-ALU-rule :=
if mode = alu then
  simultaneous
    if iop(*opcode) then
      TEMP := *ival
    else
      TEMP := B
  and
    mode := alu'

*DLX-ALU1-rule :=
     (: ALU' doesn't work, as quotes are disallowed in identifiers :)
if mode = alu' then
  simultaneous
    C := alu-op(*opcode,A,TEMP)
  and
    mode := write_back

*DLX-WRITE_BACK-rule :=
if mode = write_back then
  simultaneous
    (: reg(dest(IR)) := C :)
    reg := update-reg(reg,dest(IR),C)
  and
    mode := fetch


*DLX-MEM_ADDR-rule :=
if mode = mem_addr then
  simultaneous
    MAR := word->address(A +word *ival)
  and
```

```
    if *opcode ∈store then
      mode := pass_b_to_mdr
    else
      mode := mem_acc

*DLX-Pass_B_to_MDR-rule :=
if mode = pass_b_to_mdr then
  simultaneous
    MDR := B
  and
    mode := mem_acc

*DLX-STORE-rule :=
if mode = mem_acc ∧ *opcode ∈store then
  simultaneous
    (: mem(MAR) := MDR :)
    mem := update-mem(mem,MAR,MDR)
  and
    mode := fetch

*DLX-LOAD-rule :=
if mode = mem_acc ∧ *opcode ∈load then
  simultaneous
    (: MDR := mem(MAR) :)
    MDR := apply-mem(mem,MAR)
  and
    mode := subword

*DLX-SUBWORD-rule :=
if mode = subword then
  simultaneous
    C := alu-sw(*opcode, MDR)
  and
    mode := write_back

*DLX-TRAP-rule :=
if mode = jumps ∧ *opcode ∈trap then
  simultaneous
    IAR := instr_addr->word(PC)
  and
    PC := word->instr_addr(*ival)
  and
    mode := fetch

*DLX-BRANCH-rule :=
if mode = jumps ∧ *opcode ∈branch then
  simultaneous
    if condition(A) then
      PC := PC +pc *ival
  and
    mode := fetch

*DLX-JUMP-rule :=
if mode = jumps ∧ (*opcode ∈plainj ∨ *opcode ∈jlink) then
```

```
  simultaneous
    if iop(*opcode) then
      PC := PC +pc *ival
    else
      PC := word->instr_addr(A)
  and
    if *opcode ∈plainj then
      mode := fetch
    else
      simultaneous
        C := instr_addr->word(PC)
      and
        mode := write_back

*DLX-MOVS2I-rule :=
if mode = iarmove ∧ *opcode ∈movs2i then
  simultaneous
    C := IAR
  and
    mode := write_back

*DLX-MOVI2S-rule :=
if mode = iarmove ∧ *opcode ∈movi2s then
  simultaneous
    IAR := A
  and
    mode := fetch
```

Prozedurdeklaration

```
DLX-step#(var mode,reg,mem,IR,PC,A,B,C,TEMP,MDR,MAR,IAR)
BEGIN
    simultaneous
        *DLX-FETCH-rule
    and *DLX-OPERAND-rule
    and *DLX-ALU-rule
    and *DLX-ALU1-rule
    and *DLX-WRITE_BACK-rule
    and *DLX-MEM_ADDR-rule
    and *DLX-Pass_B_to_MDR-rule
    and *DLX-STORE-rule
    and *DLX-LOAD-rule
    and *DLX-SUBWORD-rule
    and *DLX-TRAP-rule
    and *DLX-BRANCH-rule
    and *DLX-JUMP-rule
    and *DLX-MOVS2I-rule
    and *DLX-MOVI2S-rule
END;

DLX-cycle# (var mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR)
BEGIN
  DLX-step# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR);
  IF mode ≠ fetch THEN
```

```
  BEGIN
    DLX-cycle# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR)
  END
END
```

A.5 Implementierung der DLX^p

Regeln

IF

```
*DLXp-FETCH-rule :=
  simultaneous
    IR_p := mem_instr (PC_p)
  and
    if ¬ (opcode (IR1_p) ∈jump ∨ (opcode (IR1_p) ∈branch ∧ condition (A_p)))
       then PC_p := next_p (PC_p)
```

ID

```
*DLXp-PreserveIR-rule :=
IR1_p := IR_p

*DLXp-PreservePC-rule :=
PC1_p := next_p (next_p (PC_p))

*DLXp-OPERAND-rule :=
  simultaneous
    A_p := apply-reg (reg_p, fstop (IR_p))
  and
    B_p := apply-reg (reg_p, scdop (IR_p))
```

EX

```
*DLXp-ALU-rule :=
  if opcode (IR1_p) ∈alu ∨ opcode (IR1_p) ∈set
     then if iop (opcode (IR1_p))
             then C_p := alu-op (opcode (IR1_p), A_p, ival (IR1_p))
             else C_p := alu-op (opcode (IR1_p), A_p, B_p)

*DLXp-PreserveIR1-rule :=
IR2_p := IR1_p

*DLXp-MEM_ADDR-rule :=
  if opcode (IR1_p) ∈load ∨ opcode (IR1_p) ∈store
     then MAR_p := word->address (A_p +word ival(IR1_p))

*DLXp-Pass_B_to_MDR-rule :=
  if opcode (IR1_p) ∈store
     then SMDR_p := B_p

*DLXp-MOVS2I-rule :=
  if opcode (IR1_p) ∈movs2i
     then C_p := IAR_p
```

```
*DLXp-MOVI2S-rule :=
  if opcode (IR1_p) ∈movi2s
     then IAR_p := A_p

*DLXp-JUMP-rule :=
  if opcode (IR1_p) ∈plainj ∨ opcode (IR1_p) ∈jlink
     then if iop (opcode (IR1_p))
             then PC_p := PC1_p +pc ival (IR1_p)
             else PC_p := word->instr_addr (A_p)

*DLXp-TRAP-rule :=
  if opcode (IR1_p) ∈trap
     then simultaneous
            IAR_p := instr_addr->word (PC1_p)
          and
            PC_p := word->instr_addr (ival (IR1_p))

*DLXp-BRANCH-rule :=
  if opcode (IR1_p) ∈branch
     then if condition (A_p)
             then PC_p := PC1_p +pc ival (IR1_p)

*DLXp-LINK-rule :=
  if opcode (IR1_p) ∈jlink
     then C_p := instr_addr->word (PC1_p)
```

MEM

```
*DLXp-STORE-rule :=
  if opcode (IR2_p) ∈store
     then (: mem_p (MAR_p) := SMDR_p :)
          mem_p := update-mem (mem_p, MAR_p, SMDR_p)

*DLXp-LOAD-rule :=
  if opcode (IR2_p) ∈load
     then (: LMDR_p := mem_p (MAR_p) :)
          LMDR_p := apply-mem (mem_p, MAR_p)

*DLXp-PreserveC-rule :=
C1_p := C_p

*DLXp-PreserveIR2-rule :=
IR3_p := IR2_p
```

WB

```
*DLXp-WRITE_BACK-rule :=
  simultaneous
    if opcode (IR3_p) ∈alu    ∨ opcode (IR3_p) ∈set ∨
       opcode (IR3_p) ∈movs2i ∨ opcode (IR3_p) ∈jlink then
      (: reg_p (dest (IR3_p)) := C1_p :)
      reg_p := update-reg (reg_p, dest (IR3_p), C1_p)
  and
    if opcode (IR3_p) ∈load then
```

```
    (: reg_p (dest (IR3_p)) := alu-sw (opcode (IR3_p), LMDR_p) :)
    reg_p := update-reg (reg_p, dest (IR3_p), alu-sw (opcode (IR3_p), LMDR_p))
```

Prozedurdeklaration

```
DLXp-step# (var reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p,
            PC_p, PC1_p, A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
BEGIN
  simultaneous
        *DLXp-FETCH-rule
    and *DLXp-PreserveIR-rule
    and *DLXp-PreservePC-rule
    and *DLXp-OPERAND-rule
    and *DLXp-ALU-rule
    and *DLXp-PreserveIR1-rule
    and *DLXp-MEM_ADDR-rule
    and *DLXp-Pass_B_to_MDR-rule
    and *DLXp-MOVS2I-rule
    and *DLXp-MOVI2S-rule
    and *DLXp-JUMP-rule
    and *DLXp-TRAP-rule
    and *DLXp-BRANCH-rule
    and *DLXp-LINK-rule
    and *DLXp-STORE-rule
    and *DLXp-LOAD-rule
    and *DLXp-PreserveC-rule
    and *DLXp-PreserveIR2-rule
    and *DLXp-WRITE_BACK-rule
END;
```

A.6 Implementierung von `equalresults#`

Abkürzungen

```
*one-DLXp-call :=
DLXp-step# (; reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
              A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
```

Prozedurdeklaration

```
equalresults# (mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
               reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
               A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p;
               var equalresults)
BEGIN
  DLX-cycle# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR);

  equalresults := tt;
  *one-DLXp-call ; (: IF :)

  IF opcode(IR) ∈jump ∨ opcode(IR) ∈branch THEN
    *one-DLXp-call ; (: OP :)
    *one-DLXp-call ; (: EX :)

    IF PC_p ≠ PC THEN equalresults := ff ;
```

```
    IF opcode(IR) ∈trap THEN
      IF IAR ≠ IAR_p then equalresults := ff END
    ELSE
      IF opcode(IR) ∈jlink THEN
        *one-DLXp-call ; (: MEM :)
        *one-DLXp-call ; (: WB :)

        IF apply-reg (reg, dest (IR)) ≠ apply-reg (reg_p, dest (IR)) THEN
          equalresults := ff
  ELSE
    IF PC_p ≠ PC THEN equalresults := ff;

    IF opcode(IR) ∈alu ∨ opcode(IR) ∈set ∨
       opcode(IR) ∈load ∨ opcode(IR) ∈movs2i THEN
      *one-DLXp-call ; (: OP :)
      *one-DLXp-call ; (: EX :)
      *one-DLXp-call ; (: MEM :)
      *one-DLXp-call ; (: WB :)

      IF apply-reg (reg, dest (IR)) ≠ apply-reg (reg_p, dest (IR)) THEN
        equalresults := ff

    ELSE IF opcode(IR) ∈movi2s THEN
      *one-DLXp-call ; (: OP :)
      *one-DLXp-call ; (: EX :)

      IF IAR ≠ IAR_p then equalresults := ff

    ELSE IF opcode(IR) ∈store THEN
      *one-DLXp-call ; (: OP :)
      *one-DLXp-call ; (: EX :)
      *one-DLXp-call ; (: MEM :)

      IF apply-mem (mem, MAR) ≠ apply-mem (mem_p, MAR) THEN
        equalresults := ff
END
```

A.7 Formulierung des Correctness Theorem

Abkürzungen

```
*initial-state :=
mode = fetch ∧
opcode(IR_p)∈nop ∧ opcode(IR1_p)∈nop ∧ opcode(IR2_p)∈nop ∧ opcode(IR3_p)∈nop ∧
reg_p = reg ∧ mem_p = mem ∧ PC_p = PC ∧ IAR = IAR_p
```

```
*sync-state :=
  ¬ opcode(IR1_p) ∈jump ∧ ¬ opcode(IR1_p) ∈branch
∧ ¬ opcode(IR_p)  ∈jump ∧ ¬ opcode(IR_p)  ∈branch
```

Hilfsprozedur

```
DLX-DLXp-loop# (var mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
```

```
               A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
BEGIN
  ALTERNATIVE
    DLX-DLXp-loop#(; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                     reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                     A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p);

    IF opcode(mem_instr(PC_p)) ∈jump ∨ opcode(mem_instr(PC_p)) ∈branch THEN
      (: in DLXp 2 nops ueberspringen bei jump/branch Befehlen :)
      DLX-cycle# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR);
      DLXp-step# (; reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                    A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p);
      DLXp-step# (; reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                    A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p);
      DLXp-step# (; reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                    A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
    ELSE
      DLX-cycle# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR);
      DLXp-step# (; reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                    A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
    END
  OR
    SKIP;

  (: Datenabhaengigkeiten ausschliessen :)
  IF datadependent (reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                    A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p) THEN
     ABORT
END;
```

Correctness Theorem

```
 *initial-state,
 ⟨DLX-DLXp-loop# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                    reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                    A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)⟩
 ⟨equalresults# (mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                 reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                 A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p;
                 equalresults)⟩ equalresults = ff
⊢
```

B Formalisierung der Lemmata

Hier werden diejenigen Teile der Formalisierung angegeben, die nicht direkter Bestandteil der Hauptbeweisverpflichtung sind, aber zur Formulierung der beiden Teile des DLX^p-Lemmas gebraucht wurden, also insbesondere die Spezifikation des Prädikats `equalrelevantusedlocations`.

B.1 Spezifikation `datadependent` (Vorausschau-Funktionen)

Dies sind Vorausschau-Funktionen, die vom Prädikat `equalrelevantusedlocations` benötigt werden und in der Spezifikation datadependent untergebracht wurden. Sie sind hier, und nicht in A.3 aufgeführt, da sie nicht in die Hauptbeweisverpflichtung einfließen.

Abkürzungen

```
*opc  := opcode(IR_p)
*opc1 := opcode(IR1_p)
*opc2 := opcode(IR2_p)
*opc3 := opcode(IR3_p)
*dst3 := dest(IR3_p)

*aloc := word->address(     fstopvalue(reg_p, mem_p, PC_p, IR3_p, C1_p, LMDR_p)
                      +word ival(mem_instr(PC_p)))
*alocv  := alocvalue  (reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                       A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
*alocv1 := alocvalue' (reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                       A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
*alocv2 := alocvalue''(reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                       A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)

(: Wert von Registern nach einem Zyklus :)
*MAR1  := word->address(A_p +word ival(IR1_p))
*SMDR1 := B_p
*A1    := apply-reg(reg_p,fstop(IR_p))
*B1    := apply-reg(reg_p,scdop(IR_p))

(: Wert von Registern nach zwei Zyklen :)
*MAR2  := word->address(*A1 +word ival(IR_p))
*SMDR2 := *B1
```

Spezifikation

```
datadependent =
enrich dynamic with
  functions addressedlocation( register->word,  (: reg_p        :)
                               address->word,   (: mem_p        :)
                               instr_addr,      (: PC_p         :)
                               instruction,     (: IR3_p        :)
                               word,            (: C1_p         :)
                               word)            (: LMDR_p       :)
                               : address;
            (: Adresse, auf die ein LOAD/SAVE Befehl in mem_instr(PC_p)
               zugreifen wird, unter Beruecksichtigung moeglicher
               Aenderungen von Registern durch den Befehl in IR3_p :)
```

```
fstopvalue( ... )  (: Parameter wie addressedlocation :)
                   : word;
(: Wie addressedlocation, liefert aber den Wert des ersten
   Operanden, den der Befehl in mem_instr(PC_p) benutzen
   wird :)

scdopvalue( ... )  (: Parameter wie addressedlocation :)
                   : word;
(: Wie fstopvalue, fuer den zweiten Operanden :)

iarvalue(          register->word,  (: reg_p        :)
                   instruction,     (: IR_p         :)
                   instruction,     (: IR1_p        :)
                   instr_addr,      (: PC_p         :)
                   instr_addr,      (: PC1_p        :)
                   word,            (: A_p          :)
                   word)            (: IAR_p        :)
                   : word;
(: Wert, den das IAR_p Register zwei Zyklen spaeter haben
   wird :)

iarvalue'( ... )   (: Parameter wie iarvalue :)
                   : word;
(: Wert, den das IAR_p Register einen Zyklus spaeter haben
   wird :)


alocvalue(         register->word,  (: reg_p          :)
                   address->word,   (: mem_p          :)
                   instruction,     (: IR_p           :)
                   instruction,     (: IR1_p          :)
                   instruction,     (: IR2_p          :)
                   instruction,     (: IR3_p          :)
                   instr_addr,      (: PC_p           :)
                   instr_addr,      (: PC1_p          :)
                   word, word,      (: A_p,    B_p    :)
                   word, word,      (: C_p,    C1_p   :)
                   word, word,      (: SMDR_p, LMDR_p :)
                   address, word)   (: MAR_p,  IAR_p  :)
                   :word
(: Wert, den die von einem LOAD Befehl in mem_instr(PC_p)
   addressierte Speicherstelle nach 3 Zyklen (wenn sie
   gebraucht wird) haben wird :)

alocvalue'( ... )  (: Parameter wie alocvalue :)
(: Wert, den die von einem LOAD Befehl in mem_instr(PC_p)
   addressierte Speicherstelle nach 1 Zyklus haben wird :)

alocvalue''( ... )  (: Parameter wie alocvalue :)
(: Wert, den die von einem LOAD Befehl in mem_instr(PC_p)
   addressierte Speicherstelle nach 2 Zyklen haben wird :)

axioms
```

```
   (: Axiome fuer fstopvalue schauen einen Zyklus voraus :)

   (  *dst3 ≠ fstop(mem_instr(PC_p))
    ∨ ( ¬ *opc3 ∈alu ∧ ¬ *opc3 ∈set ∧ ¬ *opc3 ∈movs2i ∧
        ¬ *opc3 ∈jlink ∧ ¬ *opc3 ∈load ))
 →   fstopvalue (reg_p, mem_p, PC_p, IR3_p, C1_p, LMDR_p)
   = apply-reg(reg_p, fstop(mem_instr(PC_p))),

   *dst3 = fstop(mem_instr(PC_p)) ∧ *opc3 ∈load
 →   fstopvalue (reg_p, mem_p, PC_p, IR3_p, C1_p, LMDR_p)
   = alu-sw(*opc3, LMDR_p),

     *dst3 = fstop(mem_instr(PC_p))
   ∧ ( *opc3 ∈alu ∨ *opc3 ∈set ∨ *opc3 ∈movs2i ∨ *opc3 ∈jlink)
 →   fstopvalue (reg_p, mem_p, PC_p, IR3_p, C1_p, LMDR_p)
   = C1_p,

... es folgen die entsprechenden Axiome fuer scdopvalue ...

   (: Axiome fuer iarvalue schauen zwei Zyklen voraus,
      und benutzen dazu ggf. iarvalue' :)

    iarvalue (reg_p, IR_p, IR1_p, PC_p, PC1_p, A_p, IAR_p)
  = iarvalue'(reg_p, IR_p, IR1_p, PC_p, PC1_p, A_p, IAR_p)
∨ *opc ∈movi2s ∨ *opc ∈trap,

    iarvalue(reg_p, IR_p, IR1_p, PC_p, PC1_p, A_p, IAR_p)
  = apply-reg(reg_p, fstop(IR_p))
∨ ¬ *opc ∈movi2s,

    iarvalue(reg_p, IR_p, IR1_p, PC_p, PC1_p, A_p, IAR_p)
  = instr_addr->word(next_p(next_p(PC_p)))
∨ ¬ *opc ∈trap,

   (: Axiome fuer iarvalue' schauen einen Zyklus voraus :)

    iarvalue'(reg_p, IR_p, IR1_p, PC_p, PC1_p, A_p, IAR_p)
  = IAR_p
∨ *opc1 ∈movi2s ∨ *opc1 ∈trap,

    iarvalue'(reg_p, IR_p, IR1_p, PC_p, PC1_p, A_p, IAR_p)
  = A_p
∨ ¬ *opc1 ∈movi2s,

    iarvalue'(reg_p, IR_p, IR1_p, PC_p, PC1_p, A_p, IAR_p)
  = instr_addr->word(PC1_p)
∨ ¬ *opc1 ∈trap,

   (: Axiome fuer alocvalue.
      alocvalue wird auf alocvalue'' reduziert und
      dieses auf alocvalue'. :)

*alocv = *alocv2 ∨ *opc ∈store,
```

```
   *alocv = *alocv2 ∨ *aloc = *mar2,
   *alocv = *smdr2  ∨ ¬ *opc ∈store ∨ ¬ *aloc = *mar2,

   *alocv2 = *alocv1 ∨ *opc1 ∈store,
   *alocv2 = *alocv1 ∨ *aloc = *mar1,
   *alocv2 = *smdr1 ∨ ¬ *opc1 ∈store ∨ ¬ *aloc = *mar1,

   *alocv1 = apply-mem(mem_p,*aloc) ∨ *opc2 ∈store,
   *alocv1 = apply-mem(mem_p,*aloc) ∨ *aloc = MAR_p,
   *alocv1 = SMDR_p ∨ ¬ *opc2 ∈store ∨ ¬ *aloc = MAR_p,

      (: Axiom fuer addressedlocation :)

   addressedlocation (reg_p, mem_p, PC_p, IR3_p, C1_p, LMDR_p) = *aloc

end enrich
```

B.2 Spezifikation `equalrelevant`

Abkürzungen

```
*opcode := opcode(mem_instr(PC))
*fstop  := apply-reg(reg,fstop(mem_instr(PC)))
*scdop  := apply-reg(reg,scdop(mem_instr(PC)))
*iop    := iop(opcode(mem_instr(PC)))
```

Spezifikation

```
equalrelevant =
enrich datadependent with

  predicates equalrelevantusedlocations(
              modes,              (: mode           :)
              register->word,     (: reg            :)
              address->word,      (: mem            :)
              instruction,        (: IR             :)
              instr_addr,         (: PC             :)
              word, word, word    (: A, B, C        :)
              word, word,         (: TEMP, MDR      :)
              address, word       (: MAR, IAR       :)
              register->word,     (: reg_p          :)
              address->word,      (: mem_p          :)
              instruction,        (: IR_p           :)
              instruction,        (: IR1_p          :)
              instruction,        (: IR2_p          :)
              instruction,        (: IR3_p          :)
              instr_addr,         (: PC_p           :)
              instr_addr,         (: PC1_p          :)
              word, word,         (: A_p,    B_p    :)
              word, word,         (: C_p,    C1_p   :)
              word, word,         (: SMDR_p, LMDR_p :)
              address, word);     (: MAR_p,  IAR_p  :)


  axioms
```

```
  equalrelevantusedlocations
            (mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
             reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
             A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
  ↔  PC_p = PC

    ∧ ((  *opcode ∈alu ∨ *opcode ∈set ∨ *opcode ∈branch
        ∨ *opcode ∈movi2s ∨ *opcode ∈jump ∨ *opcode ∈trap
        ∨ *opcode ∈load ∨ *opcode ∈store)
       → *fstop = fstopvalue (reg_p, mem_p, PC_p, IR3_p, C1_p, LMDR_p))

    ∧ ((  (*opcode ∈alu ∨ *opcode ∈set) ∧ ¬ *iop
        ∨ *opcode ∈store)
       → *scdop = scdopvalue (reg_p, mem_p, PC_p, IR3_p, C1_p, LMDR_p))

    ∧ (  *opcode ∈movs2i
       → IAR = iarvalue(reg_p, IR_p, IR1_p, PC_p, PC1_p, A_p, IAR_p) )

    ∧ (  *opcode ∈load
       →  apply-mem (mem,
             addressedlocation (reg_p, mem_p, PC_p, IR3_p, C1_p, LMDR_p))
         = alocvalue(reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                     A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p) )
end enrich
```

B.3 Formulierung der DLX^p-Lemmata

Abkürzungen

```
*initial-state,
*sync-state
```

siehe A.7

Hilfsprozedur

```
DLX-DLXp-loop# (var mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
```

siehe A.7

DLX^p-Lemma-a

```
 *initial-state,

 ⟨DLX-DLXp-loop# (; mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                    reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                    A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)⟩

  ¬ equalrelevantusedlocations (
                   mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                   reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                   A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p)
⊢
```

DLX^p-Lemma-b

```
  mode = fetch, *sync-state,

  equalrelevantusedlocations (
              mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
              reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
              A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p) ,

  ¬ datadependent (reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                  A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p),

 ⟨equalresults# (mode, reg, mem, IR, PC, A, B, C, TEMP, MDR, MAR, IAR,
                reg_p, mem_p, IR_p, IR1_p, IR2_p, IR3_p, PC_p, PC1_p,
                A_p, B_p, C_p, C1_p, SMDR_p, LMDR_p, MAR_p, IAR_p;
                equalresults)⟩ equalresults = ff
⊢
```