Formal Verification of Pipelined Microprocessors

Daniel Kröning

Universität des Saarlandes

Gegenstand der Dissertation ist die formale Verifikation von Mikroprozessoren mit Pipeline. Dies beinhaltet auch Prozessoren mit aktuellen Scheduling-Verfahren, wie den Tomasulo Scheduler und spekulativer Ausführung. Im Gegensatz zu weiten Teilen der bestehenden Literatur führen wir die Verifikation auf Gatter-Ebene durch. Des weitern beweisen wir sowohl Datenkonsistenz als auch eine obere Schranke für die Ausführungszeit. Die Beweise werden mit dem Theorem Beweissystem PVS verifiziert. Es werden sowohl in-order Maschinen als auch out-of-order Maschinen verifiziert. Zur Verifikation der in-order Maschinen erweitern wir die Stall Engine aus [MP00]. Wir entwickeln und implementieren ein Verfahren das die Transformation in die "pipelined machine" durchführt. Wir beschreiben eine generische Maschine, die die Spekulation auf beliebige Werte erlaubt. Wir verifizieren die Beweise für den Tomasulo Scheduler mit Reorder Buffer.

1 Einführung

Mikroprozessoren werden in vielen sicherheitskritischen Bereichen eingesetzt, wie beispielsweise in Automobilen oder Flugzeugen. Wir erachten daher die Korrektheit solcher Komponenten als lebenswichtig. Der Test von Prozessoren ist durch den extrem großen Zustandsraum moderner Prozessoren nur eingeschränkt möglich. Wir sind daher der Meinung, daß eine formale Verifikation die einzige Möglichkeit darstellt, eine Garantie zu erhalten.

Diese formale Verifikation sollte so durchgeführt werden, daß Dritten die Möglichkeit offen steht, die Korrektheit mit geringem Aufwand nachzuvollziehen. Wir wollen daher einen Beweis zur Verfügung stellen, der automatisiert überprüft werden kann. Insbesondere sollten alle kritischen Designs in Form von vier-Tupeln ausgeliefert werden: 1) das Design selbst, 2) eine Spezifikation, 3) ein manuell nachvollziehbarer Beweis und 4) ein maschinell verifizierbarer Beweis [KP01].

Gegenstand der Dissertation sind Korrektheitsbeweise für komplexe Mikroprozessoren. Das Design von Mirkoprozessoren gilt als fehleranfällig. Ein bekanntes Beispiel ist der Pentium FDIV Bug [Coe95].

In der Dissertation wird das Problem der Korrektheit von Hardware streng formal behandelt. Die Designs beinhalten Prozessoren mit aktuellen Scheduling Verfahren, wie beispielsweise dem Tomasulo Scheduler aus [Tom67] und spekulativer Ausführung. Im Gegensatz zu weiten Teilen der Literatur sind die Designs auf Gatter-Ebene spezifiziert.

Insbesondere werden einige der Designs für die XILINX FPGA Serie synthetisiert. Die Designs haben hohe Komplexität, was sich auf die Beweise auswirkt. Im Gegensatz zu [MP95, MP00] sind die Beweise mit dem Theorem-Beweissystem PVS [CRSS94] verifiziert. Wir geben in der Dissertation nicht den originalen PVS Beweis an, sondern versuchen, einen nachvollziehbaren Beweis in üblicher mathematischer Notation anzugeben.

Um sequentielle Maschinen zu verifizieren, erweitern wir die Datenkonsistenz-Invariante aus [MP00], indem wir einen "korrekten Wert" eines Implementation Registers wie beispielsweise $IR.2$ definieren. Ist die Korrektheit der funktionalen Komponenten, wie beispielsweise der ALU, gegeben, erlaubt uns dies den Beweis der Datenkonsistenz der präpariert sequentiellen Maschine in PVS fast völlig zu automatisieren. Wir argumentieren, daß die funktionellen Komponenten korrekte Ergebnisse liefern, wenn sie korrekte Eingaben erhalten.

Wir erweitern das Konzept der "Stall Engine" aus [MP00], indem wir eine vollständig generische Stall Engine angeben. Im Gegensatz zu der Stall Engine aus [MP00] erlaubt unsere Stall Engine eine beliebige Anzahl von Stufen und ermöglicht es, alle Stufen unabhängig voneinander anzuhalten. Des Weiteren unterstützt unsere Stall Engine das Entfernen von "Pipeline Bubbles". Das bedeutet, daß die Stufen immer dann in Betrieb sind, wenn dies die in-order Eigenschaft zuläßt. Das beinhaltet, daß "Pipeline Bubbles", wenn notwendig, aus der Pipeline entfernt werden. Wir verifizieren die Datenkonsistenz dieser Stall Engine und geben Eigenschaften an, die es erlauben, Laufzeitschranken zu beweisen.

Mit dieser erweiterten Stall Engine verbessern wir die Transformation der präpariert sequentiellen Maschine in die Maschine mit Pipeline, indem wir ein Programm implementieren, das diese Transformation automatisiert. Dies beinhaltet die Generierung von Forwarding und Interlock Schaltkreisen. Anschließend beweisen wir die Datenkonsistenz der Maschine mit Pipeline. Dies wird dadurch erreicht, daß wir beweisen, daß die Eingaben der Pipeline-Stufen korrekt sind. Damit können wir wie bei der präpariert sequentiellen Maschine argumentieren, daß die Ausgaben korrekt sind, da die funktionalen Einheiten identisch sind.

Wir geben einen generischen Ansatz zur Realisierung von spekulativer Ausführung an und stellen ein Datenkonsistenzkriterium dafür auf. Wir wenden diese Methode dann an, um DLX Pipelines mit Branch Prediction und präzisen Interrupts zu implementieren und zu verifizieren. Es ist allgemein bekannt, daß beide Techniken mit spekulativer Ausführung zu implementieren sind [SP88]. Nach unserem Wissen ist dies jedoch das erste Mal, daß beide Techniken als Instanz eines generischen Mechanismuses für die spekulative Ausführung implementiert werden.

Neben den in-order Pipelines verifizieren wir die Korrektheit des Tomasulo Scheduling Algorithmus' mit Reorder Buffer. Der Reorder Buffer bewirkt die in-order Terminierung, was es erlaubt, präzise Interrupts zu implementieren. Der Korrektheitsbeweis beinhaltet die Argumente, die notwendig sind, um die Eindeutigkeit der Tags zu beweisen.

Des Weiteren beweisen wir eine obere Schranke für die Ausführungszeit von Programmen auf allen Maschinen. Obwohl dies eine kritische Eigenschaft darstellt, wird dieses Thema in der offenen Literatur oft übergangen.

2 Hardware-Modellierung mit PVS

Sämtliche Designs und Beweise werden zweifach erstellt: Für den als Dissertation vorliegenden Text werden die Designs und Beweise im Stil des Buchs [MP00] erstellt. Zusätzlich werden die Designs und Beweise mit dem maschinellen Theorem-Beweissystem PVS [CRSS94] erfaßt.

Hardware wird auf Gatterebene als "clocked circuit" modelliert. Die Register des Designs (einschließlich der Speicherzellen) werden in PVS als Typ, also als Menge, formalisiert. Ein Zustand der Maschine c entspricht genau einem Element dieser Menge. Die Gatter werden durch eine Funktion $\delta(c)$ modelliert, die einer kombinatorischen (also Zyklusfreien) Schaltung entspricht. Tristate-Gatter werden nicht modelliert.

Dieses Modell erlaubt es, selbst komplexeste Strukturen von Mikroprozessoren hierarchisch bis auf die Ebene primitiver Gatter (AND, OR, NOT, ...) zu spezifizieren. Als Einführung zur Modellierung von Hardware mit PVS enthält die Dissertation Definitionen und Beweise für einfache Schaltkreise wie Addierer.

Notation Es sei R ein Register der Maschine. Es sei $\delta(c).R$ der Wert, der für R berechnet wird, wenn die Maschine im Zustand c ist.

3 In-Order Maschinen mit Pipeline

3.1 Die präpariert sequentielle Maschine

Der erste Schritt ist eine Maschine, die wie folgt arbeitet: Die Abarbeitung einer Instruktion wird in n beliebige Phasen aufgeteilt, die *Stufen* genannt werden. In jeder Phase wird ein Teil der Register berechnet. Die präpariert sequentielle Maschine [MP00] führt diese Schritte nacheinander durch und benötigt einen Takt für einen Schritt, d.h., es werden n Takte für eine Instruktion benötigt. Die Berechnung erfolgt also sequentiell, jedoch entspricht die Struktur der Maschine einer Maschine mit Pipeline.

Jedes Register R wird der Stufe $k \in \{0, \ldots, n-1\}$ zugewiesen, die den Wert von R berechnet. Wir verwenden $R \in out(k)$ als Notation dafür, daß R durch Stufe k geschrieben wird. Es wird auch als Ausgaberegister der Stufe k bezeichnet. Jedes Register hat einen Wertebereich $\mathcal{W}(R)$. Es seien $R_1, \ldots, R_j$ die Register, die Stufe k liest (Eingaberegister) und $R'_1, \ldots, R'_l$ die Register, die Stufe k schreibt (Ausgaberegister). Die (kombinatorischen) Datenpfade der Stufe k werden als Abbildung von Eingaberegisterwerten nach Ausgaberegisterwerten formalisiert:

$$f_k : \quad \mathcal{W}(R_1) \times \ldots \times \mathcal{W}(R_j) \longrightarrow \mathcal{W}(R'_1) \times \ldots \times \mathcal{W}(R'_l)$$

Zusätzlich sei

$$f_k Rwe : \quad \mathcal{W}(R_1) \times \ldots \times \mathcal{W}(R_j) \longrightarrow \{0,1\}$$

Takt	0	1	2	3	4	5	6...
ue_0	1	0	0	1	0	0	1
ue_1	0	1	0	0	1	0	0
ue_2	0	0	1	0	0	1	0

Tabelle 1: Sequentielle Ansteuerung der Stufen ohne Verzögerungen

das *write enable* Signal von Register $R \in out(k)$. Wenn dieses Signal aktiv ist, soll R beschrieben werden. Es sei ue_k das *update enable* Signal der Stufe k. Wenn ue_k aktiv ist, werden die Werte der Ausgaberegister der Stufe k aktualisiert.

Die Schaltkreise, die die Eingabewerte für die Funktionen $f_k R$ usw. berechnen, werden durch die Funktion g_k modelliert. Im Fall der präpariert sequentiellen Maschine ist dies einfach der Wert in dem gewünschten Register. Die Funktion wird für die Maschine mit Pipeline modifiziert, um die Forwarding Hardware einzufügen. Der Einfachheit halber wird der Parameter g_k im Folgenden weggelassen.

Der Wert, der in ein Register R geschrieben wird, hängt davon ab, ob eine Instanz von R in der vorhergehenden Stufe ist oder nicht.

- Wenn ja, dann ist der neue Wert durch $f_k R$ wenn $f_k Rwe$ aktiv ist, und ansonsten der Wert in der Instanz des Registers in der vorhergehenden Stufe. Das clock enable Signal eines solchen Registers ist ue_k.
- Wenn nein, dann ist der neue Wert immer $f_k R$. Das clock enable Signal ce des Registers ist aktiv wenn das write enable und das update enable signal aktiv sind:

  $$ce = f_k Rwe \wedge ue_k$$

Wenn R zu einem Register-File gehört, wird zusätzlich die Adresse benötigt. Die Schreibadresse sei $f_k Rwa$, die Adresse für Lesezugriffe sei $f_k Rra$. Die Signale $f_k Rwe$ und $f_k Rwa$ werden vorberechnet. $Rwe.j$ und $Rwa.j$ bezeichnen die vorberechneten Versionen dieser Signale in Stufe j.

Durch Reihum-Aktivierung der update enable Signale ue_k (Tabelle 1) erhält man eine sequentielle Maschine. Aufgrund der sequentiellen Natur der Berechnung ist die Korrektheit dieser Maschine sehr leicht nachzuweisen. Die Dissertation führt dies anhand eines 32 Bit RISC Prozessors aus.

3.2 Maschine mit Pipeline

Es sei $I_0, I_1, \ldots$ eine Instruktionsfolge. Die präpariert sequentielle Maschine arbeitet eine Instruktion in n Schritten ab. Die Maschine mit Pipeline bearbeitet bis zu n Instruktionen der Instruktionsfolge gleichzeitig (Abbildung 1).

Um eine Maschine mit Pipeline zu erhalten, transformieren wir die präpariert sequentielle Maschine. Aufgrund von Datenabhängigkeiten oder Verzögerungen in der Speicher Hier-

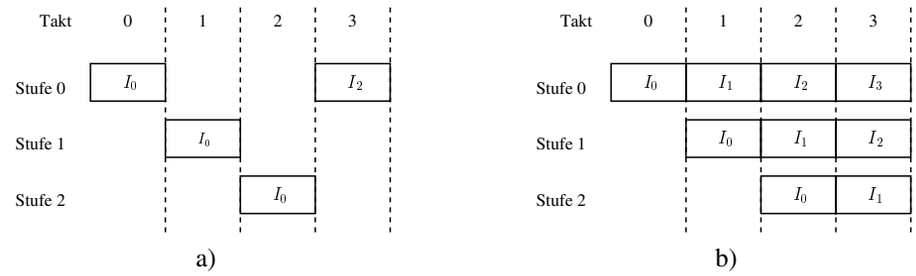

Abbildung 1: Ansteuerung der Stufen in der präpariert sequentiellen Maschine (a) und in der Maschine mit Pipeline (b).

archie ist es oft notwendig, die Ausführung einer Instruktion in einer bestimmten Stufe anzuhalten, während die anderen Stufen weiterarbeiten. Als ersten Schritt der Transformation fügen wir eine *Stall Engine* zur sequentiellen Maschine hinzu. Das Konzept ist von [MP00] übernommen und wird im Rahmen der Dissertation wesentlich erweitert.

Für jede Stufe $s \in \{1, \ldots, n-1\}$ wird ein ein-Bit Register $full.s$ hinzugefügt. Zusätzlich definieren wir ein Signal $full_k$ wie folgt:

$$\begin{array}{lrcl} & full_0 & = & 1 \\ k \in \{1, \ldots, n-1\}: & full_k & = & full.k \end{array}$$

Das Signal zeigt an, ob sich eine Instruktion in der Stufe befindet, die Stufe also "voll" ist.

Das Signal $stall_k$ zeigt an, daß die Bearbeitung der Instruktion in Stufe k angehalten (verzögert) werden soll. Mit der Hilfe dieses und des full Signals wird das update enable Signal definiert. Die Werte in den Ausgaberegistern einer Stufe werden aktualisiert, wenn eine Instruktion in der Stufe ist und die Stufe nicht angehalten ist:

$$ue_k = full_k \wedge \overline{stall_k}$$

Die full Bit Register werden mit Nullen initialisiert und wie folgt aktualisiert: Eine Stufe ist voll, wenn sie aktualisiert wird oder angehalten ist:

$$s \in \{1, \ldots, n-1\}: \quad \delta(c).full.s = ue_{s-1} \vee stall_s$$

Um das Signal $stall_k$ zu definieren, wird das Signal $dhaz_k$, das einen *Data Hazard* in Stufe k anzeigt und das Signal ext_k, das eine andere, externe Verzögerung anzeigt, verwendet. Stufe k wird angehalten, wenn eine Datenabhängigkeit oder eine externe Verzögerung vorliegt, oder wenn Stufe $k+1$ angehalten ist:

$$\begin{array}{lrcl} k \in \{1, \ldots, n-1\}: & stall_k & = & (dhaz_k \vee ext_k \vee stall_{k+1}) \wedge full_k \\ & stall_{n-1} & = & (dhaz_{n-1} \vee ext_{n-1}) \wedge full_k \end{array}$$

Mit dieser Stall Engine bleibt die Maschine wann immer möglich in Betrieb. Dies ist eine Verbesserung gegenüber Designs, die die Maschine im Fall von externen Verzögerungen

Takt	0	1	2	3	4	5	6...
ue_0	1	1	1	1	1	1	1
ue_1	0	1	1	1	1	1	1
ue_2	0	0	1	1	1	1	1

Tabelle 2: Parallele Ansteuerung der Stufen ohne Verzögerungen

vollständig anhalten. Tabelle 2 zeigt die Ansteuerung der Stufen für den Fall, daß keine Verzögerungen auftreten.

Forwarding Der letzte Schritt der Transformation der sequentiellen Maschine in eine Maschine mit Pipelining ist das Hinzufügen von Forwarding Logik. Forwarding Logik stellt sicher, daß im Fall von Datenabhängigkeiten der korrekte Wert als Eingabe für die Stufe verwendet wird oder - falls der Wert noch nicht verfügbar ist - die Stufe angehalten wird.

Die Forwarding Logik wird wie folgt hinzugefügt: R sei ein Eingaberegister der Stufe k. Wenn eine Instanz von R entweder Ausgaberegister der Stufe $k-1$ oder der Stufe k ist, braucht nichts geändert zu werden. In allen anderen Fällen sei w die Stufe, die das Register R beschreibt, d.h., $R \in out(w)$. Die fünfstufige DLX aus [HP96] liest einen GPR Operanden in Stufe $k = 1$ (Decode), der durch Stufe $w = 4$ (Write Back) geschrieben wird. In diesem Fall muß Forwarding Logik hinzugefügt werden.

Das Ergebnis vieler Instruktionen ist jedoch bereits lange vor der Stufe, die das Register beschreibt, verfügbar. So ist beispielsweise in der fünfstufigen DLX das Ergebnis von ALU Instruktionen bereits in Stufe 2 (Execute) verfügbar. Diese Ergebnisse werden in einem Register zwischengespeichert. Dieses Register wird als *write alias* Register bezeichnet. Das Programm, das die Transformation durchführt, erwartet die Namen dieser Register als Eingabe. Für den Fall der DLX mit fünf Stufen werden zwei dieser Register benötigt, eines in der Execute-Stufe und eines in der Memory-Stufe. Ein Wert, der in dieses Register geschrieben wird, muß dem entgültigen Ergebnis entsprechen. Es sei Q das write alias Register für Register R.

Mit der Hilfe des write enable Signals von Q wird ein *valid* Signal wie folgt definiert: Die Eingabe ist gültig (valid), wenn das Register Q in Stufe k ($f_k Qwe$ aktiv) oder in einer früheren Stufe beschrieben wird. Um festzustellen, ob es in einer früheren Stufe beschrieben wurde, wird das Signal in den Registern $Qv.k$ mitgeführt. Das valid Signal in Stufe k ist daher:

$$Q_k valid \quad = \quad Qv.k \vee f_k Qwe$$

Das Register $Qv.k$ wird wie folgt aktualisiert:

$$\delta(c).Qv.k \quad = \quad Q_{k-1} valid$$

Dies erlaubt es, *Hit* Signale $R_k hit[j]$ zu definieren, die die Stufe anzeigen, in der die

Instruktion ist, die den gewünschten Wert berechnet. Das Hit Signal einer Stufe ist aktiv, wenn die Stufe voll ist, die Instruktion in der Stufe das gewünschte Register beschreibt und die Adressen übereinstimmen. Zum Vergleich werden die vorherberechneten Versionen des write enable signals und der Adresse von R, d.h. $Rwe.j$ und $Rwa.j$, verwendet.

$$\begin{aligned}&\forall j \in \{k+1, \ldots, w-1\}: \\ &\quad R_k hit[j] \;=\; full_j \wedge Rwe.j \wedge (f_k Rra = Rwa.j)\end{aligned}$$

Der Adressenvergleich entfällt, wenn R ein einzelnes Register ist. Wenn eines der Hit Signale aktiv ist, sei top die erste Stufe mit aktiven Hit Signal:

$$top \;=\; \min\{j \in \{k+1, \ldots, w\} \mid R_k hit[j]\}$$

Für das Forwarding wird der Wert aus der Stufe top benutzt. Es sei $g_k R$ der Wert, den die Forwarding Logik in Stufe k für Eingaberegister R berechnet. Wenn top die Stufe ist, in der R geschrieben wird, d.h. $top = w$, wird der Wert am Eingang des Registers verwendet:

$$top = w \quad \Longrightarrow \quad g_k R = f_w R$$

Wenn der Wert aus einer anderen Stufe genommen wird, wird der Wert, der in Q geschrieben wird, verwendet. Wenn kein Hit Signal aktiv ist, wird der Wert in dem Register R genommen. Wenn R zu einem Register-File gehört, sei $a = f_k Rra$ die Adresse des Lesezugriffs.

$$top \text{ undefiniert} \quad \Longrightarrow \quad g_k R = R.(w+1)[a]$$

Data Hazards Dieses Verfahren schlägt fehl, wenn ein Hit vorliegt und der Wert, der zum Forwarding verwendet wird, noch nicht gültig ist. In diesem Fall wird das Signal $dhaz_k$ aktiviert. Des Weiteren aktivieren wir $dhaz_k$, wenn das Data Hazard Signal von Stage top aktiv ist, da in diesem Fall die Werte, die zum Forwarding verwendet werden, nicht gültig sind.

Forwarding wird nicht nur für Operanden von Instruktionen verwendet. Viele Prozessoren verwenden einen oder mehrere Delay Slots für Sprungbefehle. Diese Technik wird Delayed Branch genannt. Ist eine sequentielle Maschine gegeben, die Delayed Branch implementiert, erzeugt das Transformationsprogramm automatisch eine Maschine mit Pipeline mit einem oder mehreren Delay Slots.

3.3 Spekulative Ausführung

Die spekulative Ausführung ist eine Technik die das Anhalten der Pipeline durch Datenabhängigkeiten verhindert. Dies geschieht, wenn kein Forwarding eines benötigten Wertes möglich ist. Anstatt die Ausführung anzuhalten, wird die Ausführung einfach mit einem

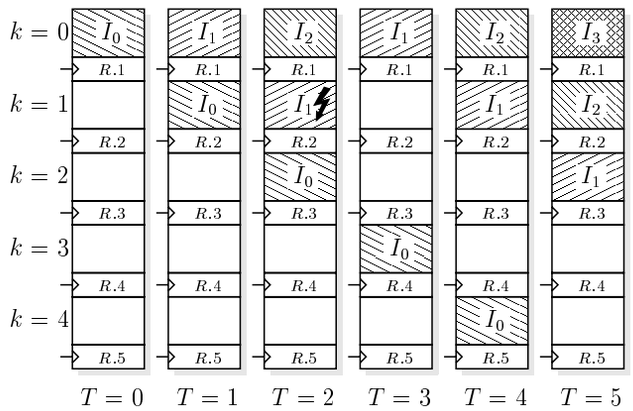

Abbildung 2: Ausführung der Instruktionen I_0 bis I_3 in einer Maschine mit Pipeline und Spekulation. Der PC von I_1 sei falsch spekuliert. Dies wird in Stufe 2 erkannt (angedeutet durch den Blitz). Volle Stufen sind schraffiert.

Wert, der geschätzt wird, fortgesetzt. Sobald der korrekte Wert verfügbar ist, wird der korrekte Wert mit dem geschätzten Wert verglichen. Wenn beide gleich sind, sind auch die Berechnungen, die mit dem geschätzten Wert gemacht worden sind, korrekt.

Wenn die beiden Werte verschieden sind, sind für gewöhnlich auch alle Berechnungen, die auf dem geschätzten Wert basieren, fehlerhaft. Dieser Fall wird Fehlspekulation genannt und die Berechnung muß in der Stufe, in der falsch geschätzt wurde, erneut begonnen werden. Dieser Vorgang wird als *Rollback* bezeichnet. Alle Änderungen am Zustand der Maschine, die durch die falsche Berechnung verursacht wurden, müssen rückgängig gemacht werden.

Die Dissertation führt ein neues Verfahren ein, das es erlaubt, auf beliebige Eingabewerte zu spekulieren. Ein Programm erzeugt automatisiert Hardware, die sicherstellt, daß bei Fehlspekulation die Berechnung mit dem korrekten Wert erneut durchgeführt wird. Der Hardwaredesigner gibt lediglich die Werte an, die spekuliert werden sollen und einen Schaltkreis, der die Schätzung durchführt. Die Dissertation führt zwei Beispiele für die Verwendung dieser Technik an:

- Durch Spekulation, ob ein bedingter Sprung durchgeführt wird oder nicht, wird eine Sprungvorhersage implementiert.
- Präzise Interrupts werden für IEEE konforme Fließkommazahlenarithmetik und virtuellen Speicher benötigt. Die Dissertation führt aus, wie präzise Interrupts als Sonderfall von spekulativer Ausführung implementiert werden können. In üblichen Designs wird dabei immer angenommen, daß kein Interrupt auftritt [SP88]. Wir schlagen statt dessen vor, auch auf das Auftreten von Interrupts zu spekulieren und somit die Leistung zu verbessern.

4 Out-of-Order Maschinen

Die bisher beschriebenen Maschinen bearbeiten die Instruktionen in Programmreihenfolge. Die Leistung solcher Prozessoren bricht ein, sobald Instruktionen mit langer Latenz wie beispielsweise Speicherzugriffe ausgeführt werden. Durch Verlassen der Programmreihenfolge kann die Leistung bei der Ausführung solcher Instruktionen in der Regel erhöht werden.

Diese Technik ist als *Out-of-Order Execution* bekannt. Der beliebteste Out-of-Order Execution Algorithmus ist der Tomasulo Scheduling Algorithmus [Tom67]. Er ist einer der leistungsstärksten Scheduling Algorithmen. Der Algorithmus wird vielfach eingestzt, beispielsweise durch den IBM PowerPC, den Intel Pentium-Pro oder den AMD K5. Der originale Tomasulo Scheduler verwendet Out-of-Order Terminierung und ist daher ohne zusätzliche Hardware nicht für präzise Interrupts geeignet. Wir implementieren präzise Interrupts mit Hilfe eines *Reorder Buffers*[SP88]. Der Reorder Buffer sortiert die Instruktionen wieder zurück in Programmreihenfolge, bevor die Ergebnisse in die Register geschrieben werden.

Der Tomasulo Scheduling Algorithmus implementiert Forwarding durch Verwendung von *Tags*. Ein Tag erlaubt es, eine Instruktion in der Maschine zu identifizieren. Der Algorithmus pflegt eine Tabelle, die für jedes Register das Tag der jeweils letzten Instruktion enthält, die das Register beschreibt. Um den Wert dieses Registers zu erhalten, wird dieses Tag verwendet. Die Herausforderung beim Korrektheitsbeweis liegt in der Wiederverwendung dieser Tags nach der vollständigen Abarbeitung einer Instruktion. Das bedeutet, daß sich ein Tag nicht eindeutig einer bestimmten Instruktion zuordnen läßt. Die Arbeit formalisiert die Argumente, die notwendig sind, um die Eindeutigkeit der Tags während der Ausführung der zugehörigen Instruktion zu zeigen.

5 Zusammenfassung und Ausblick

Die Arbeit beschreibt Algorithmen, die automatisiert Interlock und Forwarding Hardware erzeugen und somit den Entwicklungsprozeß entscheident erleichtern. Neben der Hardware werden auch Korrektheitsbeweise erzeugt. Es wird ein generisches Verfahren für spekulative Ausführung vorgestellt und auf Sprungvorhersage und präzise Interrupts angewendet. Das Tomasulo Scheduling Verfahren wird formalisiert und dessen Korrektheit wird gezeigt.

Gegenstand eines aktuellen Forschungsvorhabens ist die Implementierung eines Theorembeweisers der Hardwaredesignsprachen wie Verilog als Eingabesprache akzeptiert und der auch für große Designs schnell genug ist und automatisiert werden kann. So soll die Akzeptanz formaler Methoden in der Industrie verbessert werden.

Literaturverzeichnis

[Coe95] Tim Coe. Inside the Pentium FDIV Bug. *Dr. Dobb's Journal of Software Tools*, 20(4), Apr 1995.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective Theorem Proving for Hardware Verification. In *2nd International Conference on Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1994.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[KP01] Daniel Kröning and Wolfgang Paul. Automated Pipeline Design. In *Proc. of 38th ACM/IEEE Design Automation Conference (DAC 2001)*, pages 810–815. ACM Press, 2001.

[MP95] Silvia M. Müller and Wolfgang J. Paul. *The Complexity of Simple Computer Architectures*. Lecture Notes in Computer Science 995. Springer-Verlag, 1995.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer-Verlag, 2000.

[SP88] James E. Smith and Andrew R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[Tom67] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

Daniel Kröning: Ich wurde am 6. November 1975 in Mainz geboren und habe von 1986 bis 1990 das Marie-Therese-Gymnasium, Erlangen und von 1990 an das Gymnasiums am Rotenbühl, Saarbrücken besucht. Die Schulausbildung wurde 1995 mit dem Abitur abgeschlossen. Von 1995 bis 1996 habe ich Zivildienst in Saarbrücken geleistet. Seit 1995 bin ich als Geschäftsführer des Handshake e.V., Saarbrücken tätig.

Im Wintersemester 1996 habe ich das Studium der Informatik mit Nebenfach Wirtschaftswissenschaften an der Universität des Saarlandes aufgenommen. Das Studium wurde im Februar 1999 mit dem Diplom in Informatik innerhalb von 5 Semestern mit der Note "sehr gut" abgeschlossen.

Im Februar 1999 erhielt ich ein Stipendium des DFG Graduiertenkollegs "Effizienz und Komplexität von Algorithmen und Rechenanlagen", im Juli 1999 den Preis der Kühborth-Stiftung. Im November 2000 wurde ich in das DFG Graduiertenkolleg "Leistungsgarantien für Computersysteme" aufgenommen. Im Juli 2001 erhielt ich die Promotion in Informatik mit der Gesamtnote "ausgezeichnet".

Seit März 2001 bin ich Gastwissenschaftler an der Carnegie Mellon Universität in Pittsburgh, USA bei Prof. Edmund Clarke.