Seminar Formal Methods for Fun and Profit

Verifikation der Fliesskomma-Arithmetik bei Intel

Dozent Jun.-Prof. Dr. Bernhard Beckert et al.
FB Informatik

Dennis Willkomm
SS 05

Universität Koblenz-Landau

Zusammenfassung. In dieser Arbeit geht es um die Verifikation der Fliesskommaarithmetik bei Intel. Nach einer kurzen Einführung in die jüngere Vergangenheit Intels und dem Paradebeispiel des FDIV-Bugs wird Intels Arbeit und die Werkzeuge vorgestellt. Die Darstellung von Fliesskommazahlen nach IEEE-Standard wird eingeführt, und im Anschluss daran die Vorgehensweise einer Spezifikation und anschliessenden Verifikation am einfachen Beispiel von FMUL erklärt. Im Anschluss wird am Beispiel von FDIV die Verifikation in der Praxis verdeutlicht.

1 Einleitung

In dieser Seminararbeit im Rahmen des Seminars „Formal Methods for Fun and Profit“ geht es um die Verifikation der Fliesskommaarithmetik bei Intel. Dies ist ein sehr spannendes Thema, da gerade die Firma Intel in der Vergangenheit bittere Erfahrungen mit einem fehlerhaft ausgeliefertem Prozessor machen musste. Daher ist es interessant zu sehen, wie man bei Intel aus vergangenen Fehlern gelernt hat und diese Erfahrungen heute in der Entwicklung von neuen Prozessoren gewinnbringend einsetzt.
Zu Beginn der Arbeit wird kurz auf Intels Pentium Bug eingegangen, der exemplarisch zeigt, was passieren kann, wenn man unverifizierte Produkte auf den Markt bringt. Danach werden die Mittel der Verifikation vorgestellt, die die Mitarbeiter von Intel verwenden, um die Korrektheit ihrer Prozessoren zu testen. Nachdem die grundlegenden Werkzeuge eingeführt sind, wird kurz auf die Repräsentation von Fliesskommazahlen nach dem IEEE-Standard und die damit verbundenen Probleme eingegangen. Der letzte und umfangreichste Teil der Arbeit wird dann die vorgestellten Werkzeuge, Methoden und das Wissen über die Fliesskommarepräsentation zusammenführen und exemplarisch zeigen, wie die Mitarbeiter von Intel Fliesskommaoperationen formal verifizieren. Als Beispiel dafür dienen die Funktionen FMUL, die eine Multiplikation berechnet, und FDIV, die Division von Fliesskommazahlen.

2 Der Pentium-Bug

2.1 Prominente Fehler

Mit der zunehmenden Komplexität der Technik wird auch die Entwicklung immer komplexer. Es schleichen sich viel schneller Fehler, sogenannte Bugs, ein. Die Hersteller haben dann nur noch begrenzt die Möglichkeit diese Fehler nach der Auslieferung zu beseitigen, zum Beispiel durch Workaraounds, Änderungen der Dokumentationen oder auch in einer Bugfix-Version. Es besteht jedoch immer auch die Möglichkeit, dass größere Konsequenzen auftreten könnten.
So gibt es einige prominente Beispiele, wo fehlerhafte Hardware oder Software Menschenleben in Gefahr gebracht hat. Als Beispiel sei nur einmal der Vorfall mit der Ariane Rakete genannt, die aufgrund eines Fehlers explodierte.
Nicht jeder Fehler muss gleich so lebensgefährlich sein wie der Ariane Fehler. Aber für ein Unternehmen kann auch ein kleiner Fehler sehr gefährlich sein, wie das Beispiel Intel zeigt.

2.2 Der FDIV-Bug

Bekannt wurde der FDIV-Bug, benannt nach dem Assemblerbefehl für Fliesskommadivisionen FDIV, 1994. Es wurden Beispiele gefunden bei denen das Ergebnis der Division nach dem Runden um bis zu 61ppm(*parts per million*) von dem genauen Ergebnis abwich. Die Einheit ppm steht hier, ähnlich wie Prozent, für den Millionsten Teil eines Ganzen. Verwendet wird diese Einheit in der

Physik, der Chemie oder auch, wie in diesem Fall, im Qualitätsmanagement um Fehlerraten zu beschreiben.
Intel behauptete, den Fehler schon vor seiner Veröffentlichung in der Presse entdeckt zu haben. Es entstand eine Diskussion, ob der Fehler für den Normalanwender überhaupt von Bedeutung sei, da die meisten Programme, die 1994 auf den Heimcomputern liefen, die Möglichkeit der Fliesskommaberechnung aufgrund fehlender Fliesskommaeinheiten gar nicht nutzten.
So gab es verschiedene Ansichten: Intel behauptete der Fehler würde statistisch nur alle 27.000 Jahre auftreten während IBM behauptete, der Fehler könne alle sechs Stunden auftreten.
Die Reaktion auf diesen Bug war, dass Intel erst ankündigte alle CPUs von Anwendern, die darlegen konnten, dass sie von dem Fehler betroffen waren, auszutauschen. Dies stiess jedoch in der Öffentlichkeit auf Empörung und man forderte Intel auf, alle fehlerhaften CPUs zu tauschen. Intel beugte sich dem Druck der Öffentlichkeit und tauschte nun also alle fehlerhaften Produkte aus. Dadurch entstand Intel nicht nur ein nicht zu verachtender Imageschaden, sondern es gab auch immense Kosten, die sich auf 500 Millionen Dollar beliefen.
Aber Intel zog seine Lehren aus dieser kleinen Katastrophe. Man begann schonungslos mit der Veröffentlichung von Fehlern in den eigenen CPU's. Um diese Informationen einzusehen musste man zuvor Verschwiegenheitserklärungen unterzeichnen, während die Informationen nun jedem zugänglich gemacht wurden.

3 Formale Methoden bei Intel

3.1 Möglichkeiten und Grenzen der Verifikation

Prozessoren und Computerchips werden immer leistungsfähiger und komplexer. Durch diese steigende Komplexität werden sie jedoch auch sehr viel fehleranfälliger.
In [Harrison 2003] ist die Rede von fast 8000 Bugs beim Design des Pentiums 4, die erkannt wurden. Zugute kommt den Mitarbeitern dabei, dass die Erkennungsrate in der Phase vor der Herstellung des ersten Prototypen bei nahezu 100 Prozent liegt.
Beim Auffinden der Bugs gibt es zwei Möglichkeiten. Die eine besteht aus umfangreichen Tests, während die andere Methode die formale Verifikation liefert.
Die Grenzen der Tests sind schnell erreicht. Zum einen dauert es sehr lange, bis alle relevanten Tests durchgeführt wurden, da man sich ja noch in einer *Pre-Silicon*-Phase befindet. Zum anderen gibt es aber auch einfach zu viele Möglichkeiten, die alle explizit getestet werden müssten, um auf Nummer sicher zu gehen. Theoretisch müsste man, zum Beispiel, um einen Addierer für Fliesskommazahlen umfassend zu testen, 2^{160} mögliche Paare testen.
Daher setzt man zunehmend auf die Möglichkeiten, die die formale Verifikation bietet.
Der Vorteil einer formalen Verifikation besteht darin, dass sie einen mathematischen Beweis für die Korrektheit eines Designs darstellt, basierend auf einer formalen Spezifikation.

In der Mathematik werden nur formal gehaltene Beweise anerkannt, ein „Beweis“ durch umfangreiche Tests hingegen wird nicht anerkannt. In der Commputertechnik sah das bis vor einigen Jahren noch genau umgekehrt aus. Hier wurden umfangreiche Tests als Standardverfahren für den Beweis der Korrektheit verwendet, die auch offiziell anerkannt wurden. Im Gegensatz zur Mathematik galten Beweise durch formale Methoden als sehr exotisch [Harrison 2003].
Jedoch sollte der FDIV-Bug im Pentium gezeigt haben, dass Tests nicht ausreichend sind um Korrektheit lückenlos nachzuweisen. Schließlich wurden auch in diesem speziellen Falle unzählige Tests durchgeführt, wovon keiner negativ verlaufen war.

3.2 Formale Methoden in der Industrie

In der Industrie werden die formalen Methoden immer mehr zum Standard für die Hardwareentwicklung. Der Hauptgrund dafür sind die Konsequenzen, die für ein Unternehmen entstehen, wenn es, wie im Beispiel Intel, Hardware produziert, die fehlerhaft ist. Aber auch der modulare Aufbau der Hardware und der größere Bereich für die Automatisierung sind als Gründe für diese Entwicklung zu nennen.
Zu den in der Industrie, und in dem speziellen Beispiel Intel, hauptsächlich verwendeten formalen Methoden gehören laut [Harrison 2003]

Model Checking stellt eine algorithmische Methode dar, um finite-state Systeme formal zu verifizieren. Dabei wird getestet, ob das Design einer ‚häufig in temporal-logik aufgestellten, Spezifikation entspricht. Dabei wird das System häufig als gerichteter Graph dargestellt.

Theorem Proving prüft mathematische Theoreme. Dabei wird ein gegebener Satz beispielsweisse aus vielen gegebenen Regeln hergeleitet, oder, im umgekehrten Fall, in viele Regeln zerlegt, bis dass bewiesen ist, dass der Satz korrekt ist.

Auf die detaillierte Funktionsweise der einzelnen Methoden wird in einer speziellen Arbeit im Rahmen des Seminars gesondert eingegangen.

3.3 Die Arbeit bei Intel

Bei Intel wird nach den Verlusten durch den Pentium Bug sehr stark auf die Möglichkeiten der formalen Verifikation gesetzt. Das Resultat der Arbeit ist, dass viele Bugs aufgedeckt wurden und die formale Verifikation mittlerweile Standard bei Intel ist, im Bereich der Fliesskomma-Einheiten.
Bei der Verifikation der Fliesskomma-Einheit des Itanium Prozessors wurde unter anderem die Division, die Quadratwurzelfunktion sowie Sinus und Cosinus durch *Theorem Proving* geprüft.
Wie bereits oben erwähnt wird ein Satz beim Theorem Proving auf eine Reihe von gegebenen Regeln zurückgeführt. Dies kann sehr viele Schritte umfassen, bis

aus den primitiven Regeln ein komplexer Satz entstehen kann. Da dieser Vorgang von Hand ausgeführt sehr aufwändig und langwierig ist, gibt es spezielle Theorem Prover auf Softwarebasis, die die Arbeit erleichtern. Als Beispiel sei hier ACL2 genannt, der von AMD zur Verifikation von Fliesskomma-Einheiten verwendet wird, sowie HOL Light, das von Intel verwendet wird.

4 Die Verifikation einer Fliesskomma-Einheit

Nachdem nun die grundlegenden Handwerkzeuge in der Industrie kurz beleuchtet wurden, beschäftigt sich dieses Kapitel mit der Verifikation der Fliesskommaeinheit bei Intel im speziellen.
Dabei wird zuerst einmal auf den IEEE-Standard für Fliesskommazahlen eingegangen, um den grundlegenden Aufbau von Fliesskommazahlen zu erläutern. Mit diesem Wissen wird erst die recht einfache Funktion FMUL spezifiziert und verifiziert und anschliessend die etwas komplexere FDIV Operation.

4.1 Repräsentation von Fliesskommzahlen

Es gibt viele unterschiedliche Möglichkeiten um Fliesskommazahlen darzustellen, die sich je nach Prozessortyp unterscheiden können. Um die formale Verifikation bei Intel genauer betrachten zu können ist es nötig, erst einmal die intern verwendete Repräsentation von Fliesskommazahlen zu betrachten. Dabei wird hauptsächlich auf [O'Leary u. a.] zurückgegriffen, der die IEEE-Spezifikation verwendet.
Eine Fliesskommazahl wird aus einem Vorzeichen $s \in \{-1, 1\}$, einer Mantisse $m \geq 0$, und einem Exponenten $e \geq 0$. Dabei repräsentiert jede Fliesskommazahl die Rationale Zahl

$$R = \frac{s \cdot m \cdot 2^e}{2^P \cdot 2^{bias}}$$

wobei P die Anzahl von Bits im fraktalen Anteil der Mantisse darstellt.
Für ein solches R werden nun der jeweils nächstgrößere und nächstkleinere darstellbare Wert ermittelt,

$$ulp^+ = \frac{B^+}{2^P \cdot 2^{bias}} \qquad ulp^- = \frac{B^-}{2^P \cdot 2^{bias}} \qquad ulp = \frac{2^n \cdot 2^e}{2^P \cdot 2^{bias}}$$

wobei ulp^+ der nächstgrößere Wert wäre, ulp^- der nächstkleinere und ulp allgemein der nächste repräsentierbare Wert ist. In den meisten Fällen gilt $B^+ = B^- = 2^n \cdot 2^e$. In Abbildung 1 sieht man den Aufbau einer Fliesskommazahl nach IEEE-Standard schematisch dargestellt. Dabei stellt die Anzahl der Bits, die für den Exponent stehen die Charakteristik der Zahl dar. Die Gesamtlänge der Fliesskommazahl beschreibt die Genauigkeit (32bit für single, 64bit für double). Weiterhin legt der IEEE-Standard fest, dass die Zahlen berechnet werden, als würde man mit einer unendlichen Genauigkeit arbeiten können. Nach der Berechnung werden die Zahlen dann in ein darstellbares Format konvertiert. Dabei gibt es vier Rundungsmethoden:

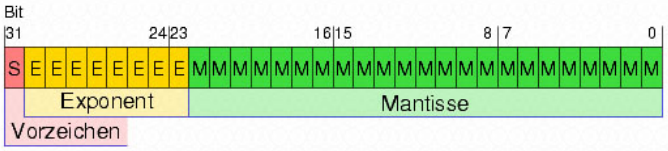

Abb. 1. Repräsentation von Fliesskommazahlen nach IEEE 754(Quelle: Wikipedia)

- $round(toZero, R, V) = (|R| \leq |V|) \wedge (|V| < |R| + ulp)$
- $round(toNegInf, R, V) = (R \leq V) \wedge (V < R + ulp^{+})$
- $round(toPosInf, R, V) = (R - ulp^{-} < V) \wedge (V \leq R)$
- $round(toNearest, R, V) = (R - \frac{1}{2}ulp^{-} \leq V) \wedge (V \leq R + \frac{1}{2}ulp^{+}) \wedge ((R - \frac{1}{2}ulp^{-} = V) \vee (V = R + \frac{1}{2}ulp^{+}) \Rightarrow even(R))$

wobei R das gerundete Ergebnis ist und V das Ergebnis mit unendlicher Genauigkeit.

4.2 Spezifikation von Fliesskommamultiplikation

Mit dem Vorwissen über die Repräsentation von Fliesskommazahlen kann man nun den Multiplikationsvorgang $FMUL$ spezifizieren.

$$V_{FMUL} = \frac{s_1 \cdot m_1 \cdot 2^{e_1}}{2^P \cdot 2^{bias}} \cdot \frac{s_2 \cdot m_2 \cdot 2^{e_2}}{2^P \cdot 2^{bias}}$$

Gegeben ist eine Zahl $R_{FMUL} = \frac{s \cdot m \cdot 2^e}{2^P \cdot 2^{bias}}$, die aus der oben gezeigten Fliesskommamultiplikation entstanden ist, und mit einer der vier oben angegebenen Methoden gerundet wurde.
Da die Verifikationsumgebung bei Intel nur mit Integeroperationen umgehen kann, muss die Fliesskommazahl so umgewandelt werden, dass nur noch Integeroperationen verwendet werden. Durch Erweitern und Umformen erhält man aus obiger Zahl

$$round(toNegInf, R_{FMUL}, V_{FMUL}) = ((s \cdot m \cdot 2^e) \cdot 2^{P|bias} \leq V) \wedge (V' < (s \cdot m \cdot 2^e + B^{+}) \cdot 2^{P+bias})$$

wobei $V' = (s_1 \cdot m_1 \cdot 2^{e_1}) \cdot (s_2 \cdot m_2 \cdot 2^{e_2})$ und nur noch Integeroperationen verwendet werden.
Exemplarisch wurde hier der Rundungsmodus $toNegInf$ verwendet. Nachdem man nun die Spezifikation erstellt hat muss die Implementation anhand dieser Spezifikation getestet werden. Zu überprüfen ist, ob die verwendete Implementation auch das tut, was die Spezifikation vorgibt.

4.3 Die Verifikation der Fliesskommamultiplikation

Die Spezifikation der Multiplikation ist immer noch zu komplex um sie direkt zur Verifikation zu verwenden. Daher wird die gesamte Operation konkretisiert

und in kleinere Teilprobleme zerlegt.
Die erste Phase besteht also darin eine Zerlegung durchzuführen. Dabei wird so lange zerlegt, bis ein Model-checking möglich ist. In dem hier gezeigten Fall wäre das die Zerlegung in ein Multiplikationsmodul und in ein Rundungsmodul (siehe auch Abb.2). Der zweite Schritt besteht dann in der Verifikation dieser Lower-Level-Spezifikation mit dem Model Checker. Ist gezeigt, dass die Implementation in diesem unteren Level korrekt ist, so kann man nun mit einem Theorem Prover im dritten Schritt zeigen, dass die gesamte Spezifikation von der Implementation erfüllt wird.
In Abbildung 2 sieht man ein Blockdiagramm, das den grundlegenden Algorith-

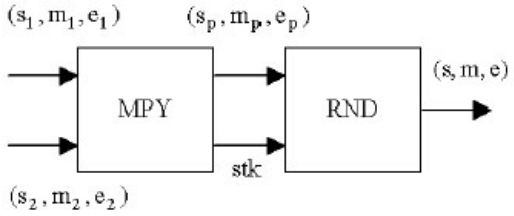

Abb. 2. Blockdiagramm: Multiplikation von Fliesskommazahlen (Quelle:O'Leary)

mus der Fliesskommamultiplikation verdeutlicht.
Zwei Fliesskommazahlen werden im ersten Teilprozess MPY zu einer approximierten Fliesskommalösung multipliziert, wobei noch nicht gerundet wird. Dabei werden die Mantissen multipliziert, die Exponenten addiert und auf die Vorzeichen wird ein XOR angewendet. Dieses Ergebnis wird in den Nachkommastellen einfach abgeschnitten von der Lösung mit unendlicher Genauigkeit. Ausserdem wird ein Sticky-Bit gesetzt, für den Fall, dass während des Abschneidevorgangs wichtige Informationen verloren gehen.
Die ungerundete Zahl und das Sticky-Bit werden dann an das RND Modul gesendet und nach dem gewünschten Schema gerundet.
Aufgrund dieser Zerlegung wird die Verifikation der $FMUL$-Operation aufgeteilt, in die Verifikation von MPY und RND. MPY wird dabei wieder aufgesplittet in Operationen auf der Mantisse, dem Exponenten und dem Vorzeichen. Durch die Verifikation dieser einzelnen Operationen mit Hilfe von Modelchecking und anschliessendem Kombinieren der Ergebnisse kann man die $FMUL$-Operation in Gänze verifizieren.
Konkreter bedeutet dies für die MPY-Einheit, dass man die folgende Eigenschaften von Mantisse, Exponent und Vorzeichen zu verifizieren hat:

- $m_P \leq m_1 \cdot m_2$

- $m_P + 1 > m_1 \cdot m_2$
- $e_P = e_1 + e_2 + P + bias$
- $s_P = s_1 \cdot s_2$

Ausserdem muss verifiziert werden, dass das Sticky-Bit den Präzisionsverlust in der Multiplikation korrekt angibt. Dafür verifiziert man

- $stk = (m_P < m_1 \cdot m_2)$

Die oben gewonnenen Resultate werden dann mit Hilfe des Theorem Provers wieder kombiniert.

Nach gleichem Schema wird nun auch beim RND-Modul vorgegangen. Man verifiziert die Bestandteile des Moduls und setzt diese nachher wieder zusammen um das gesamte Modul mit Theorem Proving zu verifizieren.

4.4 Die Verifikation von FDIV

Auch die Spezifikation von FDIV muss in Integeroperationen umgewandelt werden, da das Framework ja nur mit diesen umgehen kann. Diese Umwandlung stellt aber noch kein größeres Problem dar.
Eine Fliesskommadivision stellt sich folgendermaßen dar:

$$V_{FDIV} = \frac{\frac{s_2 \cdot m_2 \cdot 2^{e_2}}{2^P \cdot 2^{bias}}}{\frac{s_1 \cdot m_1 \cdot 2^{e_1}}{2^P \cdot 2^{bias}}}$$

$$R_{FDIV} = \frac{s \cdot m \cdot 2^e}{2^P \cdot 2^{bias}}$$

Durch Umformung erhält man

$$V_{FDIV} = \frac{s_2 \cdot m_2 \cdot 2^{e_2}}{s_1 \cdot m_1 \cdot 2^{e_1}}$$

Da $s1, s2 \in \{-1, 1\} \rightarrow \frac{s_2}{s_1} = s_1 \cdot s_2$ gilt, kann man eine weitere Umformung durchführen.

$$V_{FDIV} = \frac{s_1 \cdot s_2 \cdot m_2 \cdot 2^{e_2}}{m_1 \cdot 2^{e_1}}$$

Nun erhält man, für eine Rundung gegen $-\infty$ folgende Spezifikation für die $FDIV$ Operation:

$$round(toNegInf, R_{FDIV}, V_{FDIV}) = (s \cdot m \cdot 2^e \cdot m_1 \cdot 2^{e_1} \leq s_1 \cdot s_2 \cdot m_2 \cdot 2^{e_2} \cdot 2^{P+bias})$$

$$\wedge (s_1 \cdot s_2 \cdot m_2 \cdot 2^{e_2} \cdot 2^{P|bias} < (s \cdot m \cdot 2^e - B) \cdot m_1 \cdot 2^e)$$

Die $FDIV$ Operation ist in Intels Pentium-Prozessor durch iterative Algorithmen implementiert. Die Komplexität dieser verwendeten Algorithmen erschweren es, die Operation zu verifizieren. Man muss die Iterationen als Loop ansehen, sowie einen Anfangs- und Endzustand annehmen.

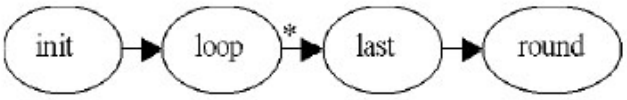

Abb. 3. Ablaufdiagramm für FDIV (Quelle: Harrison)

Der erste Schritt besteht nun darin, den Ablauf der Operation hinsichtlich seiner Korrektheit zu kontrollieren. In Abbildung 3 sieht man ein Ablaufdiagramm für $FDIV$.
Ausgehend von diesem Diagramm müssen nun die Kontrollstrukturen getestet werden.
Man geht nun von der Annahme aus, die Berechnung brauche n Iterationen, dann muss bewiesen werden, dass, wenn die Berechnung im Ausgangszustand ist, man sich auch im Zyklus 0 befindet. Während der Schleife muss man sich im Zyklus $1 \ldots n$ befinden. Im finalen Zustand muss sichergestellt sein, dass man sich im Zyklus $n+1$ befindet, und der Rundungsvorgang muss im Zyklus $n+2$ stattfinden.

Dies wird verifziert über einen Beweis durch Induktion. Die Induktionsannahme geht davon aus, dass im Zyklus 0 der initiale Zustand ist, der Schleifenzähler n ist und der nächste Zustand ist der Schleifenzustand (loop).
Der Induktionsschritt sagt, wenn der aktuelle Zustand der Schleifenzustand ist und der Schleifenzähler größer als 0 ist, dann ist der nächste Zustand wieder der Schleifenzustand und der Schleifenzähler wird um eins verringert. Wenn der Zustand der Schleifenzustand ist, der Schleifenzähler jedoch auf 0 steht, so ist der nächste Zustand der finale Zustand. Ferner wird gesagt, dass wenn der finale Zustand erreicht wurde, der nächste Schritt das Runden des errechneten Ergebnisses sein wird.
Sowohl die Annahme als auch die Schritte werden mit Model-Checking getestet und mit Theorem Proving wieder kombiniert um die Korrektheit dieses Kontrollflusses zu verifizieren.
Der zweite Schritt zur Verifikation ist die Dateninvariante zu untersuchen.
Für die iterative Abfolge der Operation sind daher genau zwei Invarianten zu beachten. Einmal hat man den Raum aus dem die Werte für Zähler und Nenner zu wählen sind, zum anderen ist da die Invariante der Werte von Zähler, Nenner und Quotient. Diese Invarianten werden wieder durch eine Induktion bewiesen. Wieder wird erst per Model Checking die Annahme und die Schritte verifiziert, bevor wieder alles mit Theorem Proving kombiniert wird um auf höchster Ebene die Verifikation abzuschliessen. Dabei wird auch das Rundungsmodul mit in die Verifikation einbezogen.

4.5 Ergebnisse bei Intels Pentium Pro

In den vorangegangenen Abschnitten wurden die Werkzeuge und die Methoden der Arbeiter bei Intel vorgestellt. Diese Herangehensweise und Durchführung der Verifikation wurde auch beim Pentium Pro Prozessor angewendet. Dabei wurden wichtige Erkenntnisse gewonnen.
Die Ziele der Verifikation der Fliesskommaeinheit für Intel waren

- das Finden und Konstruieren von Spezfikationen für alle von der Floating-Point Execution Unit (FEU) durchzuführenden Operationen
- die Verifikation der korrekten Datenpfade für alle spezifizierten Operationen
- die Verifikation von Flags und Fehlermeldungen
- die Abdeckung aller Varianten von Operationen
- die Abdeckung aller Präzisionen und Rundungsarten

Dabei teilte Intel die Aufgaben in fünf Arbeitsbereiche auf. Die ersten beiden Bereiche beschäftigten sich hauptsächlich mit den Arbeitswerkzeugen und der Entwicklung von Methoden zur Verifikation. Hier wurden hauptsächlich die Software des Model Checker und des Theorem Provers verfeinert und auf Probleme während der Verifikation hingewiesen. Die letzten drei Bereiche führten die Verifikation an sich durch. Dabei wurden vor allem die sechs Hauptoperationen verifiziert (FADD, FSUB, FMUL, FDIV, FSQRT) und eine Menge an kleineren Operationen, die zwar alle für sich recht einfach zu verifizieren waren, aber aufgrund ihrer Anzahl sehr viel Arbeitsaufwand gekostet haben.
Jeder Unterbereich beinhaltete dabei auch die Entwicklung einer Spezifikation, das Verstehen des Designs der FEU, des Model Checkers und des Theorem Provers.
Bei dieser Arbeit legte man Wert auf die Verifikation der numerischen Korrektheit. Viele andere Punkte, die für das fehlerlose Arbeiten eines Prozessors von Bedeutung sein können, wurden an dieser Stelle noch nicht von Intel geprüft.
Ausserdem legte man bei der Entwicklung der Methoden Wert darauf, dass man sie auch bei anderen Prozessormodellen wiederverwenden konnte.

5 Fazit und Ausblick

Aus der Sicht von Intel gab es bei der Verifikation der Fliesskommaeinheit drei wichtige Erkentnisse.
Die erste Erkenntnis ist, dass durch die Verifikation der Fliesskommaeinheit und der Operationen im Pre-Silicon Zustand schon Fehler vermieden werden können. Diese könnten zu späteren Entwicklungszeitpunkten vielleicht nicht mehr so schnell behoben werden. Oder im schlimmsten Fall, wie in Intels Vergangenheit mit dem FDIV-Bug, erst nach der Auslieferung erkannt werden. In dem Fall würde es sehr teuer für die Hersteller werden den Fehler zu beseitigen. Daher ist es sinnvoll eine formale Verifikation durchzuführen.
Der zweite Punkt, den Intel erkannte ist, dass eine Verifikation sich nicht von alleine durchführt, sondern dass man qualifizierte Mitarbeiter haben muss, die

das nötige Wissen und den Fachverstand besitzen um die Verifikation durchzuführen. Dabei ist es auch zeitaufwändig, so dass diese Mitarbeiter lange mit der Aufgabe beschäftigt sind. Die Ausgaben, die Intel dadurch entstehen, rentieren sich aber wieder, aus den selben Gründen wie bereits oben angesprochen: der Nutzen rechtfertigt die Investition.
Die dritte Erkenntnis ist, dass man für die Durchführung besonders detaillierte Kenntnisse besitzen muss. Und obwohl diese besonderen Kenntnisse ohne weiteres gelernt werden können sind sie bis heute noch nicht fester Bestandteil in der Computer- und Elektrotechnik.
Abschliessend bleibt zu sagen, dass die formale Verifikation in der Industrie sehr sinnvoll ist, wie das Beispiel Intel zeigt. Lange Zeit wurde diese Beweismethode sehr stiefmütterlich behandelt und galt als sehr exotisch in der Computertechnik. Doch nach Intel scheinen sich auch immer mehr andere Firmen der Bedeutung und der Möglichkeit dieser Methoden bewusst zu werden. Bei der Recherche für ein ähnliches Thema wurden von einer Firma keine Informationen über ihre Verifikationsstrategien an die Öffentlichkeit herausgegeben, was darauf schliessen lässt, dass die Methodik der formalen Verifikation in der kommerziellen Welt angekommen ist.
Intel zumindest fährt auf seiner Schiene weiter und setzt auch in Zukunft auf die Verifikation in der Entwicklung.

Literatur

Gordon 1985. Gordon, Mike: HOL: A Machine Oriented Formulation of Higher Order Logic. 1985 (68). – Forschungsbericht

Harrison 2000. Harrison, John: *The HOL Light Manual*. Univerity of Cambridge Computer Laboratory, www.cl.cam.ac.uk/users/jrh, 2000

Harrison 2003. Harrison, John: *Intel's Successes with Formal Methods*. World Forestry Center, Portland OR : Software, Science and Society, 2003

O'Leary u. a. . O'Leary, John ; Zhao, Xudong ; Gerth, Rob ; Seger, Carl-Johan H.: *Formally Verifying IEEE Compliance of Floating-Point Hardware*. Intel Corporation, Hillsboro OR : Strategic CAD Labs