

# Methoden zur Unterstützung tracebasierter Simulation für die **PowerPC**-Architektur

```
||OPTIONS: fff0ff0 | TRACE: sntest1_trace.stf.gz | VERSION: SNOOP 2.4.1 (C) 2003 C. Braun|
R00 = 000253b0 R01 = 7ffff1f0 R02 = 00000000 R03 = 00000000 R04 = 00000000 R05 = 00000000 R06 = 30000000
F00 = 30020a4000000004 F01 = 3ff0000000000000 F02 = 3fd1758000000000 F03 = 0000000000000000 F04 = 00000000
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033c8 OPC = 7c0b5040 MNC = cmpl
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033cc OPC = 7c1c0214 MNC = add
R00 = 300253b0
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033d0 OPC = 7c1c492e MNC = stwk
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033d4 OPC = 4180ffe8 MNC = bc
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033bc OPC = 800b0000 MNC = lwz
R00 = 000253c8
XER = 00000000 MSR = 5000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033c0 OPC = 812b0000 MNC = lwz
R09 = 00027610
XER = 00000000 MSR = 5000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033c4 OPC = 395b000c MNC = addi
R11 = 30001d48
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033c8 OPC = 7c0b5040 MNC = cmpl
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033cc OPC = 7c1c0214 MNC = add
R00 = 300253c8
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033d0 OPC = 7c1c492e MNC = stwk
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033d4 OPC = 4180ffe8 MNC = bc
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033bc OPC = 800b0000 MNC = lwz
R00 = 000253bc
XER = 00000000 MSR = 5000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033c0 OPC = 812b0000 MNC = lwz
R09 = 00027614
XER = 00000000 MSR = 5000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033c4 OPC = 395b000c MNC = addi
R11 = 30001d54
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033c8 OPC = 7c0b5040 MNC = cmpl
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 00000000 HQ = 00000000 FPSCR = 00000000
PCN = 300033cc OPC = 7c1c0214 MNC = add
```

Studienarbeit im Fach Informatik  
an der Eberhard-Karls-Universität Tübingen,  
Wilhelm-Schickard-Institut  
Claus Braun, Reutlingen



# Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe und dabei keine anderen als die angegebenen Hilfsmittel verwendet habe.

Reutlingen, den 24. Juli 2003

Claus Braun



# Dank

An erster Stelle möchte ich Herrn Prof. Dr. Wolfgang Rosenstiel dafür danken, dass er mir diese Arbeit an seinem Lehrstuhl ermöglicht hat. Mein weiterer Dank gilt meinem Betreuer Herrn Dipl. Inform. Gerald Heim. Er hat es mir einerseits ermöglicht, selbständig zu Arbeiten und eigene Vorstellungen zu realisieren, andererseits hatte er für alle Fragen und Probleme immer und uneingeschränkt Zeit und Rat.



# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
<b>2</b>	<b>Die <i>PowerPC</i>-Architektur</b>	<b>3</b>
2.1	Von POWER zu <i>PowerPC</i> . . . . .	3
2.2	Die Architektur . . . . .	5
2.3	Der Befehlssatz . . . . .	6
2.4	Die Register . . . . .	9
<b>3</b>	<b>Erzeugung von Traces</b>	<b>15</b>
3.1	Anforderungen und Grundlagen . . . . .	16
3.2	Die Struktur des Tracers . . . . .	19
3.3	<i>snoop</i> im Einsatz . . . . .	25
3.4	Plausibilität der Ergebnisse . . . . .	27
<b>4</b>	<b>Integration der <i>PowerPC</i>-Architektur</b>	<b>29</b>
4.1	Verarbeitung von <i>snoop</i> -Traces . . . . .	29
4.2	Instruktionsobjekte und ihre Erzeugung . . . . .	34
4.3	Performance . . . . .	37
<b>5</b>	<b>Zusammenfassung</b>	<b>38</b>
<b>A</b>	<b>PowerPC-Architekturebenen</b>	<b>39</b>



# Kapitel 1

## Einführung

Stetig sinkende Produktlaufzeiten und die damit verbundene Verkürzung der Entwicklungszyklen einer neuen Prozessorgeneration, stellen immer höhere Anforderungen an den Entwicklungsprozess. Der damit einhergehende, steigende Kostendruck sorgt außerdem dafür, dass Entwicklungs- und Produktionsabläufe kontinuierlich optimiert und beschleunigt werden müssen. Um die wachsende Komplexität moderner Mikroprozessoren beherrschen zu können, werden verstärkt Systembeschreibungssprachen eingesetzt, die die Entwicklung vereinfachen und eine bessere Verifikation ermöglichen. Auf der Produktionsseite werden durch neue Herstellungsverfahren immer komplexere und schnellere Produkte ermöglicht. Da neue Technologien und Verfahren nicht mehr in beliebig vielen Prototypen getestet werden können, tritt die Simulation immer stärker in den Vordergrund. Sowohl die Simulation der Hardware mittels Systembeschreibungssprachen wie *SystemC*, als auch die direkte Simulation spezieller Verfahren, wie zum Beispiel für Sprung- und Wertevorhersage, gewinnen daher zunehmend an Bedeutung. Im Gegensatz zu den Systembeschreibungssprachen, bei denen eine Standardisierung wie im Falle von *SystemC* die Entwicklung beschleunigt, gibt es für die Verfahrenssimulation keine einheitliche Basis. Industrie und Forschungseinrichtungen entwickeln eigene Systeme, die jeweils auf die eigenen Interessengebiete zugeschnitten und nicht kompatibel sind. Es existieren keine verbindlichen Standards für Simulationsumgebungen in diesem Bereich der Forschung.

Den Ausgangspunkt für die vorliegende Studienarbeit bildet ein Simulationssystem, das am Lehrstuhl *Technische Informatik* der Universität Tübingen entwickelt wird. Im Gegensatz zu anderen, geschlossenen Systemen handelt es sich bei dieser Simulationsumgebung um ein Java-basiertes Framework, das dem Benutzer eine offene Plattform bietet. So wird zum Beispiel der Umgang mit Trace-Dateien und die interne Kommunikation, die für einen reibungslosen Ablauf der Simulation notwendig sind, vom System komplett übernommen. Das gewünschte Verfahren hingegen kann in Form einer gewöhnlichen Java-Klasse implementiert werden. Die Vorgaben des Systems an den Nutzer sind dabei sehr gering und ermöglichen so einen hohen Freiheitsgrad bei der Implementierung. Ein weiterer großer Vorteil gegenüber anderen Systemen ist die Unabhängigkeit von einer bestimmten Plattform, die durch Java ermöglicht wird. So stehen alle modernen Betriebssysteme als Simulationsplattform offen. Durch den modularen Aufbau des Systems und durch die Trennung zwischen Grundfunktionen und Architektur, sind Erweiterungen kaum Grenzen gesetzt. Dies gilt sowohl im Bezug auf

die eigentlichen Verfahren, als auch auf die Unterstützung anderer Prozessorplattformen. Bisher stellt die Simulationsumgebung standardmäßig die Unterstützung für die S/390-Architektur von IBM zur Verfügung. Dies bedeutet, dass der Befehlssatz dieser Architektur auf eine Klassenhierarchie abgebildet wurde und somit alle Instruktionen als Java-Objekte für die Simulation bereitstehen. Die Erweiterung des Systems auf die PowerPC-, SPARC- oder Intel-Architektur ist durch die Abbildung deren Befehlssätze und die Anpassung der Trace-Verarbeitung möglich.

Im Rahmen der Studienarbeit wurde zum Einen eine Methode zur software-basierten Erzeugung von Traces auf der PowerPC-Plattform entwickelt und zum Anderen die Integration der PowerPC-Architektur in das bestehende Simulationssystem vorgenommen. Die Unterschiedlichkeit der beiden Aufgabenteile machte die Verwendung verschiedener Programmiersprachen notwendig. Da es sich bei der Erzeugung der PowerPC-Traces, wie sie in Kapitel 3 vorgestellt wird, um eine sehr betriebsystemnahe Aufgabe handelt, wurde hier die Sprache *C* unter Linux für PowerPC eingesetzt. Die Integration der PowerPC-Architektur in das Simulationssystem wurde im Gegensatz dazu in Java unter MacOS X realisiert und wird in Kapitel 4 beschrieben. Als Einstieg wird im folgenden Kapitel 2 ein kurzer Überblick über die PowerPC-Architektur gegeben. Dabei wird vor allem auf Entwicklungen der letzten Jahre und die Grundzüge der Architektur eingegangen.

## Kapitel 2

# Die *PowerPC*-Architektur

### 2.1 Von POWER zu *PowerPC*

Die Wurzeln der *PowerPC*-Architektur gehen zurück auf die POWER<sup>1</sup>-Familie von IBM. Seit Beginn der neunziger Jahre werden die RISC<sup>2</sup>-Prozessoren der POWER-Architektur von IBM in Servern und Workstations eingesetzt. Anfangs bestanden diese POWER-Systeme immer aus mehreren Bausteinen, die auf speziellen Prozessorplatinen gekoppelt waren. Erst bei den darauf folgenden POWER-Generationen wurden diese Einzelkomponenten auf einem Mikrochip zusammengeführt. Im Jahre 1991 wurde von Apple, IBM und Motorola die sogenannte *PowerPC*-Allianz ins Leben gerufen. Ihr Ziel war und ist die Entwicklung neuer Generationen von Prozessoren auf Basis der POWER-Plattform. Die erste Spezifikation der neuen Architektur wurde Ende 1991 verabschiedet und enthielt sowohl eine 32-Bit- als auch eine 64-Bit-Variante des Prozessors. Als erster Ableger der neuen Familie kam 1993 der *PowerPC* 601 auf den Markt. Aus heutiger Sicht unterscheidet sich der PPC601 in einigen Punkten deutlich von seinen Nachfolgern und kann daher als eine Art Übergangsmodell angesehen werden. Die Entwicklung erfolgte zwar bereits in weiten Teilen gemeinsam von IBM und Motorola, das Ergebnis war jedoch ein Prozessor, dessen Kern nahezu komplett von IBM stammte und dessen Bussystem von Motorola beigesteuert wurde. Ein weiteres Unterscheidungsmerkmal des PPC601 gegenüber seinen Nachfolgern war die Fähigkeit POWER-Instruktionen verarbeiten zu können. Im gemeinsamen Entwicklungszentrum in Somerset entstand mit dem *PowerPC* 603 und dem *PowerPC* 604 in den folgenden Jahren die zweite Generation von *PowerPC*-Prozessoren. Bei ihrer Entwicklung wurde das Hauptaugenmerk auf zwei unterschiedliche Bereiche gelegt. Der *PowerPC* 603 sollte ungefähr die Leistung des PPC601 erreichen, dabei jedoch energiesparender und billiger zu produzieren sein. Erreicht wurden diese Ziele durch eine deutliche Reduktion der Chipfläche und der Pins. Mit seinem geringen Energieverbrauch eignete sich der PPC603 als erster *PowerPC*-Prozessor für den Einsatz in mobilen Computern. Bei der Entwicklung des *PowerPC* 604 hingegen stand die Steigerung der Performance im Vordergrund. Im Gegensatz zum 603 erhielt der 604 ein neues Cachesystem, das ihn mehr als doppelt so leistungsfähig machte. Eine weitere Neuerung des PPC604 war seine Eignung für den Einsatz in Multiprozessorsystemen. Nach

---

<sup>1</sup>POWER ist eine Abkürzung und steht für **P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC

<sup>2</sup>**R**educed **I**nstruction **S**et **C**omputer

einer ersten Produktionsphase erhielt sowohl der PPC603 als auch der PPC604 einen verbesserten Nachfolger (PowerPC 603e und PowerPC 604e) mit nochmals gesteigerter Leistung. Die Ära der 60x PowerPC-Prozessoren ging ab der Mitte des Jahres 1997 langsam zu Ende, als mit der PowerPC 7xx-Reihe die dritte Generation (PowerPC G3) eingeführt wurde.

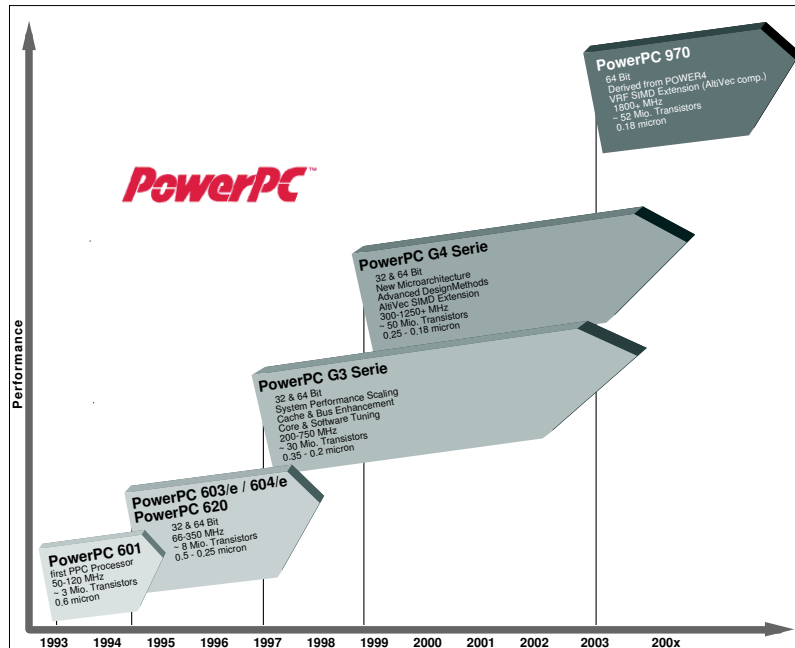


Abbildung 2.1: Entwicklung der PowerPC-Architektur

Verfügen die ersten beiden PowerPC-Generationen noch über knapp 3 bzw. 8 Millionen Transistoren, so kam es nun bei der dritten Generation zu einem Sprung auf 30 Millionen Transistoren. Weitere Verbesserungen an Cache- und Bussystem führten außerdem zu einer deutlichen Steigerung der Performance. Bei seiner Einführung wurde der PowerPC G3 mit 233MHz getaktet, seit Anfang 2003 sind Versionen mit über 1GHz erhältlich. Im Jahre 1999 bekam die PowerPC-Plattform weiteren Zuwachs, als mit der PowerPC 7xxx-Baureihe die vierte Generation (PowerPC G4) auf den Markt kam. Die ersten Unterschiede in der Entwicklung des PowerPC zwischen IBM und Motorola, die bereits beim PowerPC G3 sichtbar wurden, traten bei der vierten Generation noch deutlicher hervor. Die G4-Baureihe wurde von Motorola entwickelt und von IBM nicht übernommen, da dort die Entwicklung des PowerPC G3 vorangetrieben wurde. Die größte Neuerung des PowerPC G4 war die neue SIMD-Einheit AltiVec<sup>3</sup>. Vergleichbar mit den SIMD-Erweiterungen der Intel Pentium-Prozessoren erbrachte auch sie eine deutliche Steigerung der Performance. Besonders Anwendungen aus dem Bereich Multimedia profitieren von dieser Erweiterung. Die aktuellen Taktraten des PowerPC G4 liegen zwischen 1.2GHz und 1.5GHz. Der nächste große Leistungssprung wird Mitte 2003 mit der Einführung der fünften Generation erfolgen. Bei dem von IBM entwickelten PowerPC 970 handelt es sich um einen 64-Bit-Prozessor, der von der POWER4-Architektur abgeleitet wurde. Mit VRF wird erstmals ein PowerPC-

<sup>3</sup>auch als VelocityEngine bezeichnet

Prozessor von IBM über eine AltiVec-kompatible SIMD-Einheit verfügen und außerdem 32-Bit-kompatibel sein. Der Einstieg ist mit Taktraten von 1.8GHz bis 2GHz geplant. Anfang des Jahres 2004 soll der PowerPC 970 in einer Version mit 3GHz auf den Markt kommen.

Nicht unerwähnt bleiben soll an dieser Stelle eine weitere Besonderheit der PowerPC-Familie: Im Unterschied zu vielen anderen Prozessor-Architekturen, die sich hauptsächlich in Desktop- und Serversystemen etabliert haben, hat sich bei der PowerPC-Plattform ein eigenständiger Microcontrollerzweig entwickelt. Mit den Baureihen 4xx, 5xx und 8xx ist der PowerPC im Embedded-Markt stark vertreten. Die Einsatzgebiete reichen dabei vom Automotive-Bereich über Multimedia-Anwendungen bis hin zu Lösungen für Mobile Computing.

## 2.2 Die Architektur

Die PowerPC-Architektur ist äußerst flexibel und skalierbar angelegt, dies spiegelt sich im extrem breiten Anwendungsfeld, das von Microcontrollern in eingebetteten Systemen über mobile Computer und Desktop-Systeme bis hin zu Mehrprozessorservern und Mainframes reicht, wider. Grundsätzlich handelt es sich bei PowerPC um eine 64-Bit-Architektur, die ein 32-Bit-Subset enthält. Man unterscheidet daher zwischen reinen 64-Bit-Implementierungen<sup>4</sup>, die alle Vorteile einer 64-Bit-Architektur ausschöpfen (*64 bit mode*) und zwischen 32-Bit-Implementierungen, die nur das 32-Bit-Subset umfassen (*32 bit mode*). Um eine möglichst gute Kompatibilität zwischen beiden Implementierungen zu gewährleisten, kann auch ein 64-Bit PPC-Prozessor im 32-Bit-Modus betrieben werden. Die stärkste Verbreitung haben momentan die 32-Bit-Vertreter der PowerPC-Familie.

Die Architektur selbst ist hierarchisch gegliedert und lässt sich in drei verschiedene Ebenen unterteilen:

- **UISA - PowerPC User Instruction Set Architecture.** Die UISA definiert die Architekturebene, die für Softwareanwendungen die größte Bedeutung hat und zu der diese konform sein müssen. In der User Instruction Set Architecture ist das sogenannte *Base User-Level Instruction Set*, die *User-Level Register*, *Datentypen*, *Floating-Point-Speicherkonventionen* und Teile des *Exception-Modells* definiert.
- **VEA - PowerPC Virtual Environment Architecture.** Die VEA definiert zusätzliche Funktionalitäten auf Benutzerebene, die aus den typischen Anforderungen von Softwareanwendungen herausfallen. Die VEA beschreibt das Speichermodell für eine Umgebung, in der mehrere Geräte auf den Speicher Zugriff haben. Außerdem werden in der Virtual Environment Architecture das *Cache-Modell* und die *Cache Control Instructions* festgelegt.
- **OEA - PowerPC Operating Environment Architecture.** Die OEA definiert die sogenannte Supervisor-Ebene (*privileged state*), die typischerweise vom Betriebssystem benötigt wird. Diese Ebene enthält das *Speichermanagement*, die

---

<sup>4</sup>zum Beispiel PowerPC 970

*Supervisor-Level Register, Synchronisationsmechanismen und das Exception-Modell.*

Abbildung 2.2 verdeutlicht die Struktur und Anordnung der verschiedenen Architekturebenen. Eine detaillierte Darstellung der Architekturebenen sowie der dazugehörigen Register findet sich im Anhang A.

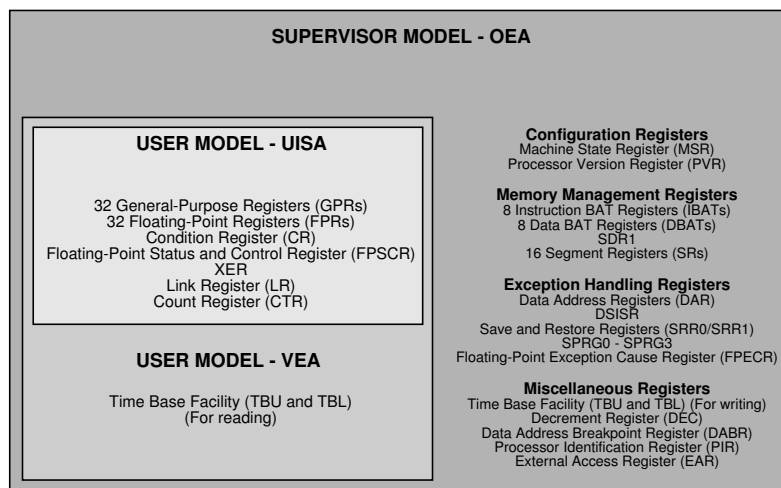


Abbildung 2.2: Architekturebenen

## 2.3 Der Befehlssatz

PowerPC-Prozessoren gehören zur Familie der RISC<sup>5</sup>-Prozessoren. Diese Klasse von Prozessoren geht in ihrer Entwicklung auf Untersuchungen existenter (komplexer) Befehlssätze zurück. Die sogenannte 80/20-Regel steht stellvertretend für die Ergebnisse dieser Untersuchungen und besagt, dass nur rund 20% eines Befehlssatzes für die Ausführung von rund 80% aller Programme verwendet werden. Im Gegensatz zu einem CISC<sup>6</sup>-Prozessor, verfügt ein RISC-Prozessor daher über deutlich weniger und einfachere Befehle. Diese Veränderung der Befehlssätze führte auch zur Prägung des Begriffs der Load/Store-Architektur, da nun eine strikte Trennung zwischen Lade-/Speicherbefehlen und Recheninstruktionen herrscht. Neben dem PowerPC gehören die SPARC- und die MIPS-Architektur zu den bekanntesten Vertretern der RISC-Familie. In jüngerer Zeit verschwimmen die Grenzen zwischen RISC- und CISC-Prozessoren immer häufiger und es werden in beiden Bereichen Konzepte und Ideen des jeweils anderen übernommen.

Von ihrer grundlegenden Struktur her gehört die PowerPC-Architektur zwar eindeutig dem RISC-Lager an, ungewöhnlich ist jedoch der relativ umfangreiche Befehlssatz (über 290 Befehle). Viele dieser Befehle gehören allerdings zu einer Familie und sind

<sup>5</sup>Reduced Instruction Set Computer

<sup>6</sup>Complex Instruction Set Computer

daher eher als Varianten ein und desselben Befehls zu betrachten. Abbildung 2.3 zeigt beispielhaft eine solche Instruktionsfamilie.

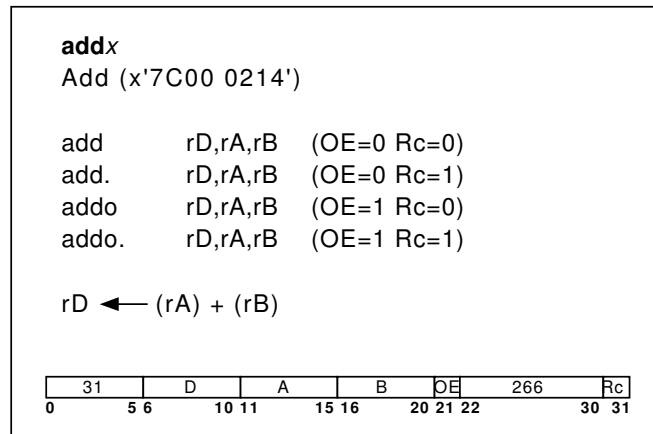


Abbildung 2.3: Beispiel eines PowerPC Befehls

Hinter der Gruppenkennung 31 im abgebildeten Beispiel, verbergen sich über 90 verschiedene Instruktionen, die jeweils über weitere Varianten verfügen. Die genaue Identifizierung des entsprechenden Additionsbefehls erfolgt in diesem Fall über die Bits 21 bis 31. Der erweiterte Opcode teil ist bei allen vier Instruktionen gleich (266), die einzelnen Varianten werden durch das OE- und das Rc-Bit kodiert.

Der PowerPC-Befehlssatz gliedert sich in die folgenden Gruppen:

- **Integer Instructions**

- Integer Arithmetic Instructions
- Logical Instructions
- Integer Rotate And Shift Instructions

- **Floating-Point Instructions**

- Floating-Point Arithmetic Instructions
- Floating-Point Multiply/Add Instructions
- Floating-Point Compare Instructions
- Floating-Point Status And Control Instructions
- Floating-Point Move Instructions
- Optional Floating-Point Instructions

- **Load/Store Instructions**

- Integer Load And Store Instructions
- Integer Load And Store With Byte Reverse Instructions
- Integer Load And Store Multiple Instructions

Integer Load And Store String Instructions

Floating-Point Load And Store Instructions

- **Load/Store With Reservation Instructions**
- **Synchronization Instructions**
- **Flow Control Instructions**
- **Processor Control Instructions**
- **Memory/Cache Control Instructions**
- **External Control Instructions**

Der Befehlssatz heutiger PowerPC-Prozessoren enthält im Unterschied zu dem des PPC601 keine POWER-Befehle mehr und ist damit zu dieser alten Version auch nicht mehr kompatibel. Die mit der vierten Generation (PowerPC G4) eingeführte SIMD-Erweiterung AltiVec umfasst rund 162 zusätzliche Befehle, auf die an dieser Stelle nicht näher eingegangen wird.

PowerPC-Prozessoren verfügen über einen Befehlssatz mit konstanter Instruktionslänge, ein Befehl wird daher grundsätzlich immer in einem 32-Bit-Wort kodiert. Die ersten sechs Bits (Bit 0 bis Bit 5) werden dabei immer für die Zuordnung zu einer Befehlsgruppe verwendet. Mit dieser Zuordnung alleine ist eine wirkliche Unterscheidung der Instruktionen jedoch noch nicht möglich<sup>7</sup>, deshalb enthalten die meisten PowerPC-Befehle zusätzlich noch den sogenannten *Extended Opcode* in den Bits 21 bis 31. Abbildung 2.4 zeigt einige der häufigsten Befehlsformate der PowerPC-Architektur im Vergleich.

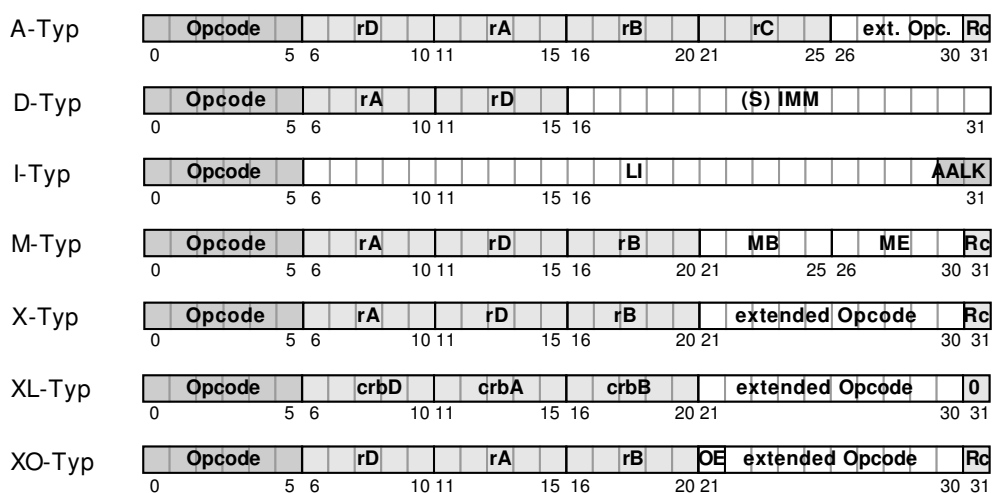


Abbildung 2.4: Befehlsformate des PowerPC

<sup>7</sup>mit 6 Bit ließen sich nur 64 Befehle insgesamt kodieren.

Dem A-Typ gehören überwiegend Floating-Point-Befehle wie zum Beispiel **fadd**<sup>8</sup> an. Befehle deren Kodierung im sogenannten D-Format vorliegt, lassen sich eindeutig anhand ihrer ersten sechs Bits identifizieren, was sie von den anderen Formaten unterscheidet. Der hintere Bit-Block (Bits 21 bis 31) wird hier für den Immediate-Teil des Befehls verwendet. Viele Load-/Store-Befehle liegen im D-Format vor. Das M-Format kommt besonders bei Rotationsbefehlen zum Einsatz, wobei ME und MB entsprechende Bitmasken beinhalten. Das XL- sowie das XO-Format können als Unterformate des X-Formats angesehen werden. Alle Spezialbefehle zur Manipulation von Special-Purpose-Registern, wie zum Beispiel **mcrfs**<sup>9</sup>, gehören dem X-Typ an. Wie bereits erwähnt, lassen sich viele Befehle in unterschiedlichen Versionen verwenden. Einer der Unterschiede wird zum Beispiel durch das Rc-Bit (Bit 31) beschrieben, mit dessen Hilfe entschieden wird, ob das Condition-Register (CR) verändert werden soll oder nicht. Neben den hier erwähnten Befehlsformaten gibt es noch weitere wie zum Beispiel das SC-Format<sup>10</sup>. Im Vergleich zu den dargestellten Formaten gehören diesen jedoch deutlich weniger Instruktionen an.

## 2.4 Die Register

PowerPC-Prozessoren verfügen über 32 General-Purpose- (GPR) und 32 Floating-Point-Register (FPR). Die General-Purpose-Register haben eine Breite von 32 Bit, die Floating-Point-Register sind mit 64 Bit doppelt so groß. Die General-Purpose- und die Floating-Point-Register gehören zur Gruppe der *User Register*. Neben diesen Hauptregistern gibt es noch zahlreiche weitere Register, die den unterschiedlichen Architekturebenen zugeordnet sind.

In den folgenden Abschnitten werden einige PowerPC-Register näher vorgestellt, die für die Trace-Erzeugung interessant sind und auf die in Kapitel 3 Bezug genommen wird. Die überwiegende Zahl dieser Register gehört zur Gruppe der *Special-Purpose Register*.

- **Das Integer-Exception-Register XER**

Das Integer-Exception-Register XER, ist eng mit den General-Purpose Registern verbunden und enthält Informationen über den Ausgang verschiedenster Integer-Operationen. Abbildung 2.5 zeigt den genauen Aufbau des Registers.

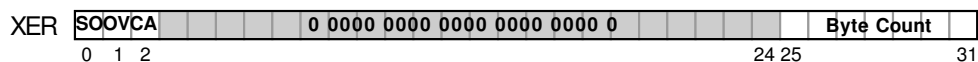


Abbildung 2.5: Das Integer-Exception-Register des PowerPC

Besonders wichtig ist der Inhalt der ersten drei Bits des XER<sup>11</sup>:

- **SO - Summary Overflow.** Das SO-Bit wird immer dann gesetzt, wenn eine Instruktion das OV-Bit für den allgemeinen Überlauf setzt. Sobald das

<sup>8</sup>fadd = Floating-Point Add

<sup>9</sup>mcrfs = Move To Condition Register From FPSCR

<sup>10</sup>Für den *System-Call*-Befehl

<sup>11</sup>entsprechend der *XER Bit Definitions* des PowerPC-Manuals [1]

Bit einmal gesetzt ist, bleibt es aktiv bis es durch eine **mcrxr**-Instruktion<sup>12</sup> oder durch den **mtspr**-Befehl<sup>13</sup> gelöscht wird. Falls auf XER der **mtspr**-Befehl ausgeführt wird und dabei SO=0 und OV=1 ist, wird SO gelöscht und OV gesetzt.

- **OV - Overflow.** Das OV-Bit zeigt an, dass während der Ausführung einer Instruktion ein Überlauf aufgetreten ist. Bestimmte Additions- und Subtraktionsbefehle, deren OE-Bit gesetzt ist, setzen OV=1, falls das *carry out* des MSB<sup>14</sup> nicht mit dem *carry in* übereinstimmt. In allen anderen Fällen wird das OV-Bit gelöscht. Neben den genannten Befehlen kann das OV-Bit auch durch Multiplikations- und Divisionsbefehle gesetzt werden, falls deren OE-Bit gesetzt ist und das Ergebnis des Befehls nicht mit 32 Bit darstellbar ist.
- **CA - Carry.** Das CA-Bit kann bei der Ausführung der folgenden Befehle gesetzt werden, falls es ein *carry out* des MSB gibt: *Add Carrying*, *Subtract From Carrying*, *Add Extended* und *Subtract From Extended*. Algebraische Schiebeoperationen können das CA-Bit ebenfalls setzen.
- **Byte Count.** Die Bits 25 bis 31 bilden das sogenannte Byte-Count-Feld. Hier wird die Anzahl der Bytes angegeben, die von **lswx**<sup>15</sup> oder **stswx**<sup>16</sup> gelesen, bzw. geschrieben werden sollen.

#### • Das Condition-Register CR

Das Condition-Register CR verfügt über acht Felder (CR[0] bis CR[7]), die jeweils aus vier Bits bestehen. CR spiegelt die Ergebnisse vieler Instruktionen wider und dient als Ausgangspunkt für bestimmte Sprungbefehle (*conditional branches*). Abbildung 2.6 zeigt den Aufbau des Condition-Registers.

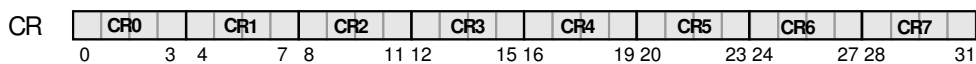


Abbildung 2.6: Das Condition-Register des PowerPC

Die einzelnen Felder des Condition-Register können mit Werten aus den General-Purpose-Registern belegt werden, hierfür steht der **mtrcf**-Befehl<sup>17</sup> zur Verfügung. Mittels **mcrf**<sup>18</sup> lassen sich auch die ersten drei Bit des XER in einem der Felder ablegen. Für die Ausführung von logischen Operationen auf dem Condition-Register werden verschiedene Spezialbefehle bereitgestellt (z.B. **crand** und **cror**). Einen besonderen Status nehmen die ersten beiden Bitfelder CR[0] und CR[1] ein, sie können die impliziten Ergebnisse von Integer- bzw Floating-Point-Operationen enthalten. Dieser Fall tritt immer genau dann ein, wenn bei der entsprechenden Integer- oder Floating-Point-Instruktion das sogenannte Record Bit

<sup>12</sup>mcrxr = Move To Condition Register From XER

<sup>13</sup>mtspr = Move To Special-Purpose Register

<sup>14</sup>Most Significant Bit

<sup>15</sup>lswx = Load String Word Indexed

<sup>16</sup>stswx = Store String Word Indexed

<sup>17</sup>mtrcf = Mover To Condition Register Field

<sup>18</sup>mcrf = Move Condition Register Field

$R_c$  gesetzt ist. Gekennzeichnet wird dies durch einen Punkt am Ende des Befehls (Vgl. **addc**<sup>19</sup> und **addc.**). Für die CR-Felder gelten die folgenden Bit-Belegungen:

CR[0] Bit	Beschreibung
0	Negative (LT) – wird gesetzt, wenn das Ergebnis negativ ist.
1	Positive (GT) – wird gesetzt, wenn das Ergebnis positiv und nicht 0 ist.
2	Zero (EQ) – wird gesetzt, wenn das Ergebnis 0 ist.
3	Summary Overflow (SO) – entspricht dem SO-Bit des XER.

Tabelle 2.1: Bit-Belegungen für CR[0]

CR[1] Bit	Beschreibung
4	Floating-Point Exception (FX) – entspricht dem FX-Bit des FPSCR.
5	Floating-Point Enabled Exception (FEX) – entspricht dem FEX-Bit des FPSCR.
6	Floating-Point Invalid Exception (VX) – entspricht dem VX-Bit des FPSCR.
7	Floating-Point Overflow Exception (OX) – entspricht dem OX-Bit des FPSCR.

Tabelle 2.2: Bit-Belegungen für CR[1]

CR[n] Bit	Beschreibung
0	Less Than bzw. Floating-Point Less Than (LT, FL) – Kleiner als...
1	Greater Than bzw. Floating-Point Greater Than (GT, FG) – Grösser als...
2	Equal bzw. Floating-Point Equal (EQ, FE) – Gleich...
3	Summary Overflow bzw. Floating-Point Unordered (SO, FU) – entspricht dem SO-Bit des XER bzw. NaN bei FP-Operationen

Tabelle 2.3: Bit-Belegungen für CR[n] bei Vergleichsoperationen

- **Das Count-Register *CTR***

Das Count-Register des PowerPC besitzt mehrere Aufgaben. Einerseits dient es als 32-Bit-Zählregister für Schleifendurchgänge, andererseits kann es auch Adressen enthalten. Diese Adressen dienen als Zieladresse für bestimmte Sprungbefehle, wie zum Beispiel **bcctrx**<sup>20</sup>. Abbildung 2.7 zeigt den Aufbau des Count-Registers.

Das Count-Register kann von entsprechender Hardware auch für die Sprungvorhersage und das *Instruction Prefetching* genutzt werden.

<sup>19</sup>addc = Add Carrying

<sup>20</sup>bcctrx = Branch Conditional To Count Register

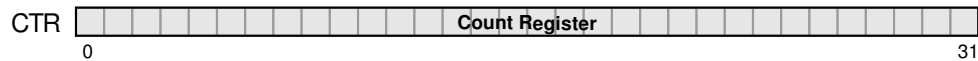


Abbildung 2.7: Das Count-Register des PowerPC

- **Das Link-Register *LR***

Das Link-Register ist 32 Bit breit und besonders wichtig für Sprungbefehle wie **bclrx**<sup>21</sup>. Abbildung 2.8 zeigt den Aufbau des Link-Registers.

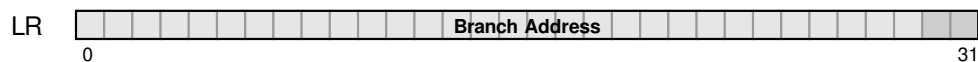


Abbildung 2.8: Das Link-Register des PowerPC

LR enthält die Zieladresse des jeweiligen Sprungbefehls, zu beachten ist an dieser Stelle allerdings, dass die Bits 30 und 31 hierfür nicht genutzt werden. Da es sich beim Link-Register wie bei den obigen Registern um ein Special-Purpose-Register handelt, kann es mittels **mtspr**<sup>22</sup> und **mfspr**<sup>23</sup> verändert werden. Das Besondere am Link-Register besteht darin, dass alle Branch-Befehle mit einer Option ausgestattet werden können, beim Sprung die auf den Branch-Befehl folgende Adresse im Link-Register abzulegen. Dies ermöglicht den Ablauf von Unterprogrammen, da sich im Link-Register die Rücksprungadresse befindet. Wenn das Unterprogramm seinerseits nicht wieder Unterprogramme aufrufen will, kann die Rücksprungadresse im Link-Register verbleiben, bis sie durch einen entsprechenden Sprungbefehl genutzt wird.

- **Das Floating-Point-Status-and-Control-Register *FPSCR***

Das Floating-Point-Status-and-Control-Register FPSCR ähnelt in manchen Bereichen dem XER, da es ebenfalls implizite Ergebnisse von Operationen enthält. Die Hauptaufgaben des FPSCR lassen sich anhand der folgenden vier Punkte beschreiben:

- Darstellung von Exceptions, die von Floating-Point-Operationen erzeugt werden.
- Darstellung des Ergebnistyps von Floating-Point-Operationen.
- Kontrolle des Rundungsmodus bei Floating-Point-Operationen.
- Aktivierung und Deaktivierung des Exception-Handlers.

<sup>21</sup>bclrx = Branch Conditional To Link Register

<sup>22</sup>mtspr = Move To Special-Purpose-Register

<sup>23</sup>mfspr = Move From Special-Purpose-Register

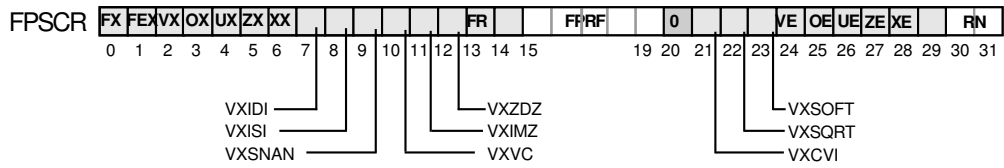


Abbildung 2.9: Das Floating-Point-Status-and-Control-Register des PowerPC

Eine detaillierte Erklärung der einzelnen FPSCR-Bits findet sich in [1] Kapitel 2.1.4.

• **Das Machine-State-Register *MSR***

Das Machine-State-Register MSR ist ein 32-Bit-Register, dessen Hauptaufgabe in der Darstellung des aktuellen Prozessorstatus liegt. Sobald es zu einer Exception während der Ausführung eines Programms kommt, wird der gesamte Inhalt des MSR im Save-and-Restore-Register SRR1 gesichert und durch neue Bits, entsprechend der Exception ersetzt. Neben den Zugriffsbefehlen *mtmsr*<sup>24</sup> und *mfmsr*<sup>25</sup> kann der Inhalt des MSR auch durch Systemaufrufe und Interrupts verändert werden. Abbildung 2.10 zeigt den Aufbau des Maschinen-Status-Registers.

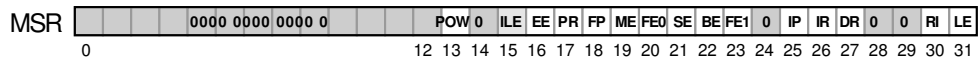


Abbildung 2.10: Das Machine-State-Register des PowerPC

Eine Übersicht über die einzelnen Status-Bits des MSR gibt Tabelle 2.4

Neben den hier vorgestellten Registern existiert noch eine große Zahl weiterer Special-Purpose-Register, wie zum Beispiel die *Memory Management Register* oder die *Exception Handling Register*, die im Anhang A.1 dargestellt sind. Diese Register gehören jedoch der VEA, bzw. der OEA an.

<sup>24</sup>mtmsr = Move To Machine State Register  
<sup>25</sup>mfmsr = Move From Machine State Register

<b>Bit</b>	<b>Name</b>	<b>Beschreibung</b>
0 - 12	-	Reserviert.
13	POW	Power management enable
14	-	Reserviert.
15	ILE	Exception little-endian mode.
16	EE	External interrupt enable.
17	PR	Privilege level.
18	FP	Floating-Point available.
19	ME	Machine check enable.
20	FE0	Floating-Point exception mode 0.
21	SE	Single-step trace enable (Optional).
22	BE	Branch trace enable (Optional).
23	FE1	Floating-Point exception mode 1.
24	-	Reserviert.
25	IP	Exception prefix.
26	IR	Instruction address translation.
27	DR	Data address translation.
28-29	-	Reserviert.
30	RI	Recoverable exception.
31	LE	Little-endian mode enable.

Tabelle 2.4: Status-Bits des Machine-State-Registers

## Kapitel 3

# Erzeugung von Traces

Unter dem Begriff Trace<sup>1</sup> findet man in der Informatik eine Reihe durchaus verschiedener Dinge aus unterschiedlichen Anwendungsbereichen. Grundsätzlich kann man einen Trace aber als eine Art Ablaufprotokoll ansehen, in dem ein Vorgang exakt dokumentiert wird. So gibt es zum Beispiel Memory-, Cache- und Instruction-Traces, die den Zustand des jeweiligen Trace-Ziels während der Ausführung einer Anwendung oder des Betriebssystems dokumentieren. Auch Netzwerke und Datenbanken können das Ziel von Tracing-Anwendungen sein. Hilfreich bei der Entwicklung von Software sind auch sogenannte Stack-Traces, die im Problemfall die letzten ausgeführten Instruktionen wiedergeben. Im weitesten Sinne kann man daher auch das Debugging einer Software, bei dem jede ausgeführte Instruktion und die entsprechenden Daten sichtbar werden, als einen Trace-Vorgang ansehen. Besonders interessant sind Instruction-Traces, da sie zum Beispiel für die Performanceanalyse und die Simulation von Werte- und Sprungvorhersageverfahren genutzt werden können. Auch die statistische Auswertung dieser Art von Traces kann sehr aufschlussreich sein, da sie Auskunft über die verwendeten Instruktionen und deren Häufigkeit gibt. Man kann somit jede Software nach ihrer Ausnutzung des Befehlssatzes charakterisieren. Je nach Art der Anwendung unterscheiden sich Traces sehr stark in ihrer Struktur und Form, es kann sich sowohl um reine Textausgaben auf dem Bildschirm als auch um Dateien unterschiedlichster Formate handeln. Für die Erzeugung von Traces bieten sich grundsätzlich, je nach Anwendungsziel, zwei Möglichkeiten an. Die hardwarebasierte Trace-Erzeugung ermöglicht sehr genaue Ergebnisse, ist jedoch gegenüber softwarebasierten Lösungen deutlich kostenintensiver und aufwändiger. Ein sehr interessanter Ansatz zur hardwarebasierten Trace-Erzeugung wurde 1997 P. A. Sandon et al. [2] beschrieben. Sandon und seinen Kollegen bei IBM entwickelten mit *NTrace* eine Methode, mit der sich aus Bus-Daten mittels eines Simulators<sup>2</sup> Instruction-Traces erzeugen lassen. Ausgangspunkt für die Trace-Erzeugung mit *NTrace* ist die Aufnahme von Bus-Transaktionen mit Hilfe eines Logic-Analyzers, die durch den Aufruf eines eigens dafür entwickelten Systemprogramms gestartet wird. Der große Vorteil gegenüber anderen Ansätzen liegt in der extrem geringen Beeinflussung des Systems und der damit verbundenen geringen Verfälschung der Ergebnisse. Die Ausführung beliebiger Software unter verschiedenen Betriebssystemen ist somit in Echtzeit möglich. Dies wie-

---

<sup>1</sup>engl. für Spur bzw. Aufzeichnung.

<sup>2</sup>Der verwendete Simulator ist aus dem *IBM PowerPC Visual Simulator* hervorgegangen.

derum eröffnet neue Möglichkeiten zur Beurteilung und Analyse der Hardware unter Berücksichtigung unterschiedlicher Softwarekonfigurationen. Die eigentliche Trace-Erzeugung erfolgt als letzter Schritt mit Hilfe des Prozessorsimulators, der aus den Bus-Transaktionen einen Instruktionsstrom generiert. Trotz aller genannten Vorteile ist dieser Ansatz jedoch mit einem beträchtlichen Aufwand verbunden, da entsprechende Spezialhard- und -software zum Einsatz kommt.

Die zweite Möglichkeit zur Erzeugung von Traces ist die mittels Software, bei der jedoch fast immer der Rückgriff auf das Betriebssystem oder spezielle Simulationssoftware notwendig ist. Nachteilig kann sich hier die beschränkte Trace-Tiefe<sup>3</sup> auswirken, da das Betriebssystem im Normalfall die Grenzen sehr eng steckt. Instruktionen, die vom Betriebssystemkern oder bestimmten Bibliotheken ausgeführt werden, können nicht erfasst werden. Ein weiterer kritischer Punkt bei Softwarelösungen ist die mögliche Veränderung des Profils der Zielanwendung<sup>4</sup>. Es liegt in der Natur der Sache, dass zum Beispiel die Ausführungsgeschwindigkeit eines Targets beim Software-Tracing sehr stark absinkt und sich dadurch die Charakteristik des Programms unter Umständen verändert. Betrachtet man zum Beispiel eine Software zur Dekodierung eines MPEG-Videostreams, so hat diese sehr enge Vorgaben im Bezug auf das Timing bei der Ausführung. Der Video-Stream soll ohne Unterbrechungen oder Ruckeln und mit synchronem Ton wiedergegeben werden. Um etwaige Leistungseinbrüche kompensieren zu können, verfügt der MPEG-Dekoder über eine Funktionalität, die es ihm ermöglicht, bestimmte Frames auszulassen. Sinkt nun durch das Tracing die Ausführungsgeschwindigkeit des Dekoders deutlich ab, so steigt der Anteil des Frame-Dropping stark an. Der Trace enthält also nicht das typische Profil des Targets, das bei der Dekodierung eines Video-Streams entsteht, sondern ein drastisch verfälschtes. Ähnliche Auswirkungen sind auch beim Software-Tracing von interaktiven Anwendungen mit grafischer Benutzeroberfläche zu erwarten. Es zeigt sich also, dass die Tracing-Methode genau an den späteren Verwendungszweck der Daten angepasst werden muss, damit sinnvolle Ergebnisse erzielt werden. Es muss ein Gleichgewicht zwischen Aufwand und Nutzwert gefunden werden.

Ausgangspunkt für die vorliegende Studienarbeit ist ein Java-basiertes Simulationssystem, dessen Schwerpunkt im Bereich der Verfahren aus der Rechnerarchitektur liegt. Grundlage für diese Simulationen bilden textuelle Instruction-Traces. Im Rahmen dieser Studienarbeit wurde nun eine softwarebasierte Methode zur Erzeugung von Instruction-Traces entwickelt, mit der sich diese flexibel an das jeweilige Simulationsziel anpassen lassen. In den folgenden Abschnitten wird der Aufbau und die grundlegende Funktionsweise der entwickelten Tracing-Software *snoop* beschrieben. In Kapitel 4 wird darauf aufbauend die Integration der PowerPC-Architektur in das bestehende Simulationssystem vorgestellt.

### 3.1 Anforderungen und Grundlagen

Zu Beginn der Entwicklung einer Tracing-Software ist es sinnvoll, die einzelnen Anforderungen sehr genau zu betrachten und zu ordnen. Prinzipiell lassen sich die An-

---

<sup>3</sup>Der Begriff Trace-Tiefe beschreibt in diesem Zusammenhang die Menge und den Informationsgehalt der erreichbaren Daten.

<sup>4</sup>im Folgenden *Target* genannt

forderungen in zwei Gruppen unterteilen:

1. Anforderungen, die an die Tracing-Software selbst gestellt werden.
2. Anforderungen, die an das zugrunde liegende Betriebssystem gestellt werden.

Für die Tracing-Software gilt, dass sie eine möglichst große Trace-Tiefe und damit einen größtmöglichen Informationsgewinn ermöglichen soll. Ein zweiter, sehr wichtiger Aspekt im Bezug auf die spätere Verwendung der Traces, ist deren flexible Anpassung. Adress-, Opcode- und Mnemonicinformationen müssen sich dafür beliebig mit den verschiedenen Registerinformationen kombinieren lassen. Die dritte große Anforderung an den Tracer stellt die Erzeugung der entsprechenden Ausgabedatei dar. Sie soll ein flexibles Format bei einer möglichst geringen Dateigröße besitzen.

Um dieses Lastenheft erfüllen zu können, muss das verwendete Betriebssystem ebenfalls bestimmte Anforderungen erfüllen. Zum Einen ist die Kontrolle eines Prozesses von außen durch den Tracer zwingend notwendig, zum Anderen muss die Ausführung der Zielanwendung in Einzelschritten möglich sein. Um die entsprechenden Informationen aus den Registern gewinnen zu können, muss der Zugriff auf die Prozessdaten des Targets vom Betriebssystem unterstützt werden. Es zeigt sich also, dass die Wahl eines geeigneten Betriebssystems den eigentlichen Ausgangspunkt für die Entwicklung bildet. Da die PowerPC-Architektur als Zielplattform feststeht, bieten sich grundsätzlich die folgenden Betriebssysteme als potentielle Basis an:

- Apple MacOS 9 & MacOS X
- IBM AIX
- BSD
- Linux
- Diverse Embedded Betriebssysteme (z.B. vxWorks)

Apples MacOS 9 erweist sich aufgrund seiner monolithischen Struktur und seines Alters als weniger geeignet, hinzu kommt, dass es kein UNIX-basiertes Betriebssystem ist. MacOS X dagegen bietet mit seinem BSD-Mach-Kernel eine vollwertige UNIX-Umgebung und fällt damit in den Kreis der möglichen Systeme. Auch AIX von IBM sowie die beiden freien Betriebssysteme BSD und Linux sind grundsätzlich geeignet. Die Klasse der Embedded-Betriebssysteme fällt aufgrund der eingeschränkten Einsetzbarkeit auf Desktopsystemen und dem damit verbundenen größeren Aufwand aus der Wahl heraus.

Da sich die oben beschriebenen Voraussetzungen wie Einzelschrittausführung und Zugriff auf die Daten eines Prozesses am besten durch den Systemaufruf *ptrace* realisieren lassen, wird dieser zum entscheidenden Faktor. Der *ptrace*-Aufruf ist zwar in allen zur Wahl stehenden Betriebssystemen implementiert, unterscheidet sich jedoch zum Teil stark in der Anzahl der möglichen Requests. *ptrace* führt eine Prozesskontrolle durch und kann mit dem entsprechenden Request sowohl vom Target als auch von der Tracing-Software genutzt werden.

Der Standardaufruf hat die folgende Form:

```
int ptrace(request, pid, adr, data)
int request, pid, adr, data, *adr;
```

Tabelle 3.1 zeigt die neun Standardrequests<sup>5</sup> des Systemaufrufs nach Gulbins & Obermayr [13], wie sie auch im POSIX-Standard enthalten sind.

Request	Name	Beschreibung
0	PT_TRACE_ME	wird vom Sohn aufgerufen. Der Prozess soll vom Vaterprozess kontrolliert werden.
1	PT_READ_I	gibt den Wert in <i>adr</i> des Sohnprozesses zurück, hier: <i>instruction space</i> (Codebereich).
2	PT_READ_D	wie 1 allerdings im <i>data space</i> (Datenbereich).
3	PT_READ_U	liefert den Wert in <i>adr</i> aus dem Prozesssystemdatenbereich.
4	PT_WRITE_I	<i>data</i> soll in <i>adr</i> im Codebereich geschrieben werden.
5	PT_WRITE_D	<i>data</i> soll in <i>adr</i> im Datenbereich geschrieben werden.
6	PT_WRITE_U	<i>data</i> soll in <i>adr</i> in u.u_exec geschrieben werden.
7	PT_CONTINUE	setzt die Ausführung des Sohnprozesses an der Stelle <i>adr</i> fort.
8	PT_KILL	beendet den kontrollierten Prozess.
9	PT_STEP	wie 7, es wird jedoch das <i>trace bit</i> im Prozessorstatuswort des kontrollierten Prozesses gesetzt, so dass nach der Ausführung jeder Instruktion ein <i>trace interrupt</i> erzeugt wird.

Tabelle 3.1: *ptrace*-Requests im Überblick

Wie bereits erwähnt, unterstützen prinzipiell alle Betriebssysteme in der engeren Wahl den Systemaufruf *ptrace* mit den obigen Requests. Leider hat sich nach der Implementierung eines ersten Rahmenprogramms sowohl unter MacOS X als auch unter IBM AIX herausgestellt, dass es Probleme beim Zugriff auf die Prozessdaten gibt. Die genaue Ursache für diese Zugriffsprobleme konnte nicht näher ermittelt werden. Da jedoch Software wie der GNU Debugger *GDB* auf den beiden Plattformen verfügbar ist und dieser ebenfalls den Systemaufruf *ptrace* nutzt, scheint eine Implementierung des Tracers prinzipiell nicht unmöglich. Insgesamt bedeutet dies jedoch für die Realisierung einen speziellen Mehraufwand in erheblichem Maß, sowohl unter MacOS X als auch unter AIX. Eine weitere Einschränkung beider Betriebssysteme ist ihre Festlegung auf die Hardware des jeweiligen Herstellers, so ist es zum Beispiel nicht möglich AIX auf einem Apple Power Macintosh oder MacOS X auf einer IBM RS/6000 laufen zu lassen. Dieser Punkt ist insofern nicht zu vernachlässigen, da für die Tracing-Software eine möglichst breite Einsatzplattform erwünscht ist. Abhilfe schafft hier Linux für PowerPC, das auf beiden Rechnerfamilien läuft und das eine sehr gute *ptrace*-Unterstützung bietet. Diese Faktoren sowie die sofortige Verfügbarkeit des Systems waren ausschlaggebend für die Entwicklung der Tracing-Software *snoop* unter LinuxPPC.

<sup>5</sup>Je nach Plattform variiert die Bezeichnung zwischen PT... und PTRACE...

### 3.2 Die Struktur des Tracers

Bei der Entwicklung des Tracers lohnt es sich, zu Beginn einen genauen Blick auf die einzelnen Bestandteile der Software zu werfen. Es ist offensichtlich, dass bei einer so hardwarenahen Aufgabe wie der Erzeugung von Traces, bestimmte Teile der Software stark plattformabhängig sind. Aus diesem Grund bietet es sich an, die Struktur des Tracers nach den einzelnen Komponenten auszurichten. Die Trennung verschiedener funktionaler Gruppen bietet sich vor allem im Hinblick auf eine spätere Portierung der Software auf eine andere Hardwareplattform an. Die Charakteristik der einzelnen Aufgabenbereiche ergibt im Falle von *snoop* eine klammerartige Struktur, bei der ein portabler Teil die architektur- und betriebssystemspezifischen Komponenten umgibt. Abbildung 3.1 verdeutlicht diese Struktur der Tracing-Software.

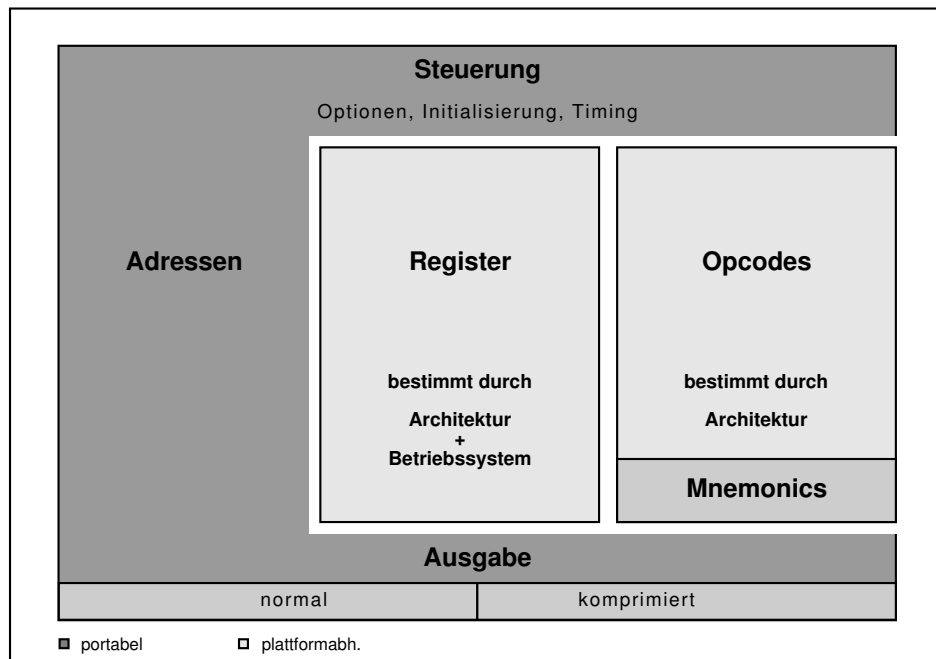


Abbildung 3.1: Struktur der Tracing-Software

**Steuerung.** Der Steuerungsteil von *snoop* ist im Wesentlichen für zwei Aufgaben zuständig. Zum Einen übernimmt er den gesamten Vorlauf des Trace-Vorgangs, was hauptsächlich die Bearbeitung der Optionen umfasst. Zum Anderen initialisiert er den eigentlichen Trace-Vorgang. Die gewünschten Programmoptionen werden *snoop* in der Kommandozeile übergeben und mittels der *getopts*-Bibliothek aufbereitet. Besonders wichtig ist bei diesem ersten Schritt die sorgfältige Trennung zwischen den Trace-Optionen, die *snoop* betreffen und den Optionen der Zielanwendung. Die genaue Aufrufsemantik sowie die möglichen Einstellungen für den Trace-Vorgang des Tracers werden in Abschnitt 3.3 vorgestellt. Jeder der möglichen Optionen ist ein spezieller Wert zugeordnet, wodurch eine leichte Kodierung der gesetzten Einstellungen in einem einzigen Optionswert ermöglicht wird. Der Steuerungsteil addiert hierfür nur die jeweiligen Wer-

te. Der Optionswert enthält dann in einem kompakten Format alle gewünschten Informationen und kann an die eigentliche Trace-Funktion übergeben werden. Durch die oben beschriebene Trennung und Aufbereitung der Optionen, ergeben sich keine Einschränkungen für die Zielanwendung und deren Optionen. Der zweite Aufgabenbereich des Steuerungsteils beinhaltet die Initialisierung des Tracings. Hierfür wird als erstes ein neuer Kindprozess durch den Aufruf von `fork()` erzeugt. Der Kindprozess erscheint zu diesem Zeitpunkt in der Prozessverwaltung des Betriebssystems als weiterer *snoop*-Prozess. Das Kodesegment des erzeugten Prozesses enthält im Wesentlichen nur zwei Aufrufe und sieht wie folgt aus:

```
if((pid=fork()) == 0) {
    //_____CHILD_PROCESS_____
    // wait a little moment
    sleep(1);

    // set PTRACE_TRACEME flag for the father process
    ptrace(PTRACE_TRACEME, 0, 0, 0);

    // run trace target...
    if(target_arg_count == 0){
        execvp(targetname, NULL);
    }
    if(target_arg_count != 0){
        execvp(targetname, argv);
    }
}
```

Nach einer kurzen Wartezeit wird als erster Schritt mittels des *ptrace*-Systemaufrufs das sogenannte *trace bit* gesetzt, um eine spätere Kontrolle des Targets durch *snoop* zu ermöglichen. Hierfür wird der Request `PTRACE_TRACEME` verwendet. Dies ist das erste und einzige Mal, dass *ptrace* vom Kindprozess verwendet wird. Der zweite Schritt besteht nun aus dem Start der eigentlichen Zielanwendung. Dieser Initialisierungsschritt erfolgt mittels eines *execvp()*-Aufrufs. Ab diesem Zeitpunkt tritt der Kindprozess in der Prozessverwaltung des Betriebssystems mit dem Namen des gewählten Targets auf. Der Vaterprozess *snoop* hat durch *ptrace* nun den Zugriff auf das Target und kann es so in Einzelschritten ausführen.

Die letzte Aktion des Steuerungsteils, besteht aus dem Aufruf der eigentlichen Trace-Funktion *sn\_do\_trace()*. Ihr übergibt er unter anderem die PID<sup>6</sup> des Targets und den kodierten Optionswert. Der folgende Programmcode zeigt den stark vereinfachten Kern der Trace-Software.

```
while(waitpid(child_pid, &status, 0)) {
    if((pt_result=ptrace(PTRACE_SINGLESTEP, child_pid,
        1, 0))!=-1) {
        ...
        perror("PTRACE_SINGLESTEP");
    }
}
```

---

<sup>6</sup>PID = Prozess-ID

```

        // Make child terminate & go home...
        kill(child_pid, SIGKILL);
        exit(EXIT_FAILURE);
    }else{
        // Skip some instructions?
        if(skip_count>0) {
            skip_count--;
        }else{
            insn_count++;
            errno=0;
            user_retries=0;
            while(((current_pc = ptrace(PTRACE_PEEKUSER,
child_pid, PT_NIP*4 , 0))!=-1)&&errno) {
                ...
                // Wait a bit
                sched_yield();
                ...
            }
            ...
        }
        ...
    }
}

```

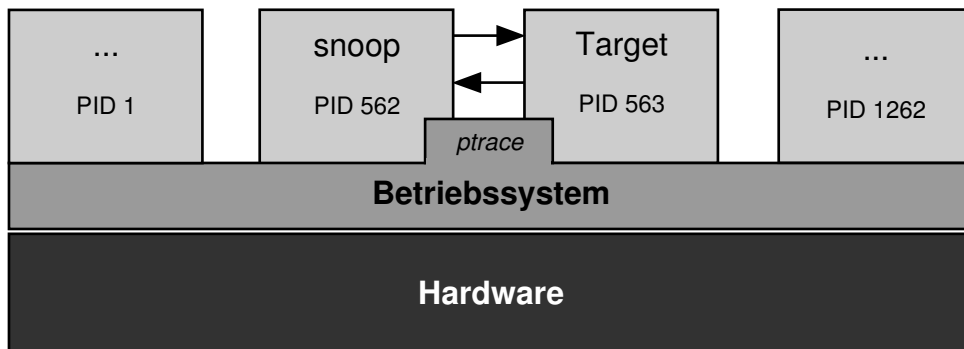


Abbildung 3.2: *ptrace* als Bindeglied zwischen Tracer und Target

**Adressen.** Der Adressenteil der Tracing-Software ist weitgehend unabhängig von der Plattform, da er durch *ptrace* bestimmt ist. Es kann jedoch durch das verwendete Betriebssystem zu kleineren Änderungen bei den jeweiligen Offsets kommen. So wäre bei der Portierung der Software auf die SPARC-Architektur unter Umständen eine Anpassung notwendig, falls dort SUN Solaris anstelle von Linux zum Einsatz käme. Zu beachten ist in diesem Zusammenhang auch die zugrunde liegende Architekturvariante. Die im Rahmen der Studienarbeit implementierte Software arbeitet auf der 32-Bit-Variante der PowerPC-Architektur. Bei einer Portierung auf 64-Bit-Systeme wäre eine Anpassung des Adressteils notwendig. Der Trace-Vorgang selbst läuft innerhalb einer *while*-Schleife ab, deren Abbruchkriterium der aktuelle Status des Kindprozesses ist.

```
while(((current_pc = ptrace(PTRACE_PEEKUSER, child_pid,
```

```
PT_NIP*4 , 0))==-1)&&errno) { ...
```

An dieser Stelle wird bei jedem Durchlauf die aktuelle Adresse erfasst und in einer Variable (`current_pc`) des Typs `unsigned int` abgelegt.

**Register.** Wie in Kapitel 2 bereits vorgestellt, verfügt die PowerPC-Architektur über jeweils 32 General-Purpose- und Floating-Point-Register. Die GPRs haben dabei eine Breite von 32 Bits, die FPRs sind im Gegensatz dazu 64 Bit breit. Neben diesen Hauptregistern gibt es noch eine Reihe weiterer sogenannter Special-Purpose-Register, die jedoch nicht alle frei zugänglich sind. *snoop* erfasst von allen diesen SPRs die in 2.4 beschriebenen Statusregister. Auf alle übrigen Special-Purpose-Register kann nicht mittels *ptrace* zugegriffen werden, da sie nicht der UISA angehören. Der Zugriff auf die Daten der General-Purpose-Register erfolgt mit dem folgenden Aufruf:

```
//The GPRs...
for(i=PT_R0; i<=PT_R31; i++) {
    if(((rex[i] = ptrace(PTRACE_PEEKUSER, child_pid,
        i*4, 0)) == -1) && errno) {
        perror("PTRACE_PEEKUSER could not read gp register content!");
    }
}
```

Die notwendigen Offsets werden durch das Betriebssystem vorgegeben, im obigen Falle lauten sie `PT_R0` bis `PT_R31` für die GPR. Der zurück gelieferte Wert wird in einem Feld vom Typ `unsigned int` abgelegt, der jeweilige Feldindex entspricht dabei dem entsprechenden Register. Analog erfolgt der Zugriff auf die Floating-Point- und die Statusregister mit entsprechend angepassten Offsets. Auch diese Werte werden wiederum in Feldern des Typs `unsigned int` abgelegt und später den Ausgabefunktionen übergeben. Da die PowerPC-Architektur über Floating-Point-Register mit einer Breite von 64 Bit verfügt, werden von *ptrace* immer zwei sogenannte *Slots* für die Abbildung eines Floating-Point-Registers belegt. Der eigentliche Registerwert wird von *snoop* daher aus zwei 32-Bit `unsigned int`-Werten zusammengesetzt. Das entsprechende Feld für die Speicherung verfügt somit über doppelt so viele Einträge wie das der General-Purpose-Register.

Im Gegensatz zu den Statusregistern, die bei jedem Trace-Durchlauf vollständig protokolliert werden, gibt es bei den General-Purpose- und bei den Floating-Point-Registern eine Besonderheit. Über die Option `-c` lässt sich für diese Hauptregister eine zusätzliche Vergleichsstufe aktivieren, die den kompletten Registersatz auf etwaige Änderungen überprüft. Um einen korrekten Vergleich der Registersätze zu ermöglichen, werden sowohl die Inhalte der GPR als auch die der FPR der vorangegangenen Instruktion zwischengespeichert. Der Vorteil dieser zusätzlichen Vergleichsstufe liegt in der deutlichen Reduktion der Tracegröße, da nur noch die Änderungen der jeweiligen Register enthalten sind.

Als kritisch kann sich beim Zugriff auf die einzelnen Prozessdaten des Targets vor allem das Timing erweisen. Erfolgt der lesende Zugriff mittels *ptrace* zu früh, so werden keine oder falsche Daten gelesen. Um dies zu vermeiden wird

durch die Anweisung `sched_yield()` ein kurzer Moment gewartet, bis die entsprechenden Daten durch das Betriebssystem verfügbar gemacht worden sind. Je nachdem, ob nun ein komprimierter oder ein normaler Trace erzeugt werden soll, werden die gesammelten Daten an die entsprechenden Ausgabefunktionen übergeben.

**Opcodes & Mnemonics.** Besonders wichtig bei der Erzeugung von Instruction-Traces sind die Opcodes der jeweils protokollierten Instruktionen. Um die Lesbarkeit für den Menschen zu verbessern und die spätere Weiterverarbeitung innerhalb des Simulationssystems zu beschleunigen, werden auch die Mnemonics der Instruktionen im Trace abgelegt. Man erhält die Opcodes durch einen `ptrace`-Aufruf der folgenden Form:

```
insn_image = ptrace(PTRACE_PEEKTEXT, child_pid, insn_pc, 0);
```

Das Ergebnis wird in einer Variable (`insn_image`) vom Typ `unsigned int` gespeichert. Für die schnelle Dekodierung der Opcodes zu Mnemonics stellt `snoop` eine eigene Dekoderfunktion bereit. Die Funktion `decode_ppc_linux` (`insn_image`) enthält eine kaskadierte *if*-Treppe, die den übergebenen Opcode zuerst der Befehlsgruppe entsprechend erfasst. Je nachdem, um welche Art von Instruktion es sich handelt, können auf diese erste Dekodierstufe bis zu zwei weitere Stufen folgen. In der Dekodierfunktion ist also der vollständige Befehlssatz des PowerPC abgebildet. Der komplett dekodierte Mnemonic wird als `char`-Pointer zurückgegeben. Neben der Protokollierung der Register handelt es sich auch bei diesem Arbeitsschritt um einen extrem plattformabhängigen Bereich der Tracing-Software, der speziell auf die Zielplattform angepasst ist.

**Ausgabe.** Den Abschluss eines jeden Trace-Durchgangs bildet die Ausgabe der gesammelten Daten in die Trace-Datei. Grundsätzlich stellt `snoop` zwei verschiedene Dateioptionen zur Verfügung, die sich besonders beim Speicherbedarf deutlich unterscheiden. Das Format der jeweiligen Traces ist identisch und sieht wie folgt aus:

```
|OPTIONS: fff0ff0 | TRACE: sntest1_trace.stf.gz | VERSION: SNOOP 2.4.1 (C) 2003 C. Braun|
R00 = 000253b0 R01 = 7ffff1f0 R02 = 00000000 R03 = ... R30 = 30027550 R31 = 30002cd8
F00 = 30028a6400000004 F01 = ... F30 = 0000000000000000 F31 = 0000000000000000
XER = 00000000 MSR = 4000d032 CTR = 00000000 LNK = 300032ac CCR = 80000000 MQ = ... FPSCR = ...
PCN = 300033c8 OPC = 7c0b5040 MNC = cmp1
```

Jeder Trace beginnt mit einer Zeile, die den sogenannten Header enthält. Hier legt `snoop` verschiedene Statusinformationen wie zum Beispiel die gewählten Trace-Optionen und die Softwareversion ab. Der Header kann bei zukünftigen Versionen der Tracing-Software beliebig erweitert werden. In den darauf folgenden beiden Zeilen findet sich immer ein vollständiger Abzug der beiden Hauptregistersätze, dies ist auch dann der Fall, wenn vom Benutzer die Option `-c` gewählt wurde. Die vollständigen Registerinformationen am Anfang des Traces sichern später dem Simulationssystem eine korrekte Ausgangssituation zu. Bei jeder Instruktion vollständig abgebildet werden die Statusregister, die

vom eigentlichen Befehl gefolgt werden. Bei der Instruktioninformation ist jeweils die aktuelle Adresse, der Opcode und der entsprechende Mnemonic enthalten. Dieses Trace-Format ist unabhängig von der gewählten Ausgabeform, was einen einheitlichen Zugriff auf komprimierte und nichtkomprimierte Traces gewährleistet. Das Standardformat für *snoop*-Traces ist **.stf**. In diesen Dateien werden die gewonnenen Informationen über die Instruktionen und Register unkomprimiert, in rein textueller Form, abgelegt. Dies hat den Vorteil, dass der Trace überall ohne weitere Zusatzsoftware eingesehen werden kann. Leider wird dieser Vorteil im Normalfall mit einer größeren Trace-Datei erkauft. Da die Traces jedoch für die Weiterverarbeitung innerhalb des Simulationssystems erzeugt werden, ist die Lesbarkeit für den Menschen ein eher untergeordnetes Kriterium, das zugunsten der Platzersparnis bei der komprimierten Variante in den Hintergrund tritt. Durchschnittlich fallen für eine getracete Instruktion rund 650 Bytes Daten an. Wenn man diesen Wert nimmt und auf einen Trace mit einer Million Instruktionen bezieht, so erhält man mit rund 620MB<sup>7</sup> eine beeindruckende Dateigröße. Abbildung 3.3 verdeutlicht diesen Sachverhalt am Beispiel eines Trace-Vorgangs der DES-Verschlüsselung.

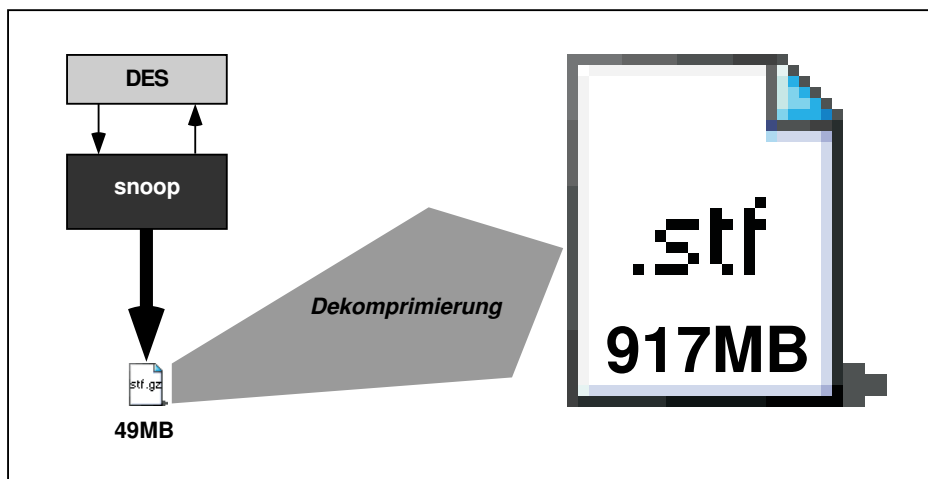


Abbildung 3.3: Vergleich der Trace-Größe am Beispiel DES-Verschlüsselung

Um eine möglichst effektive Kompression der Trace-Daten zu gewährleisten bedient sich *snoop* der *zlib*, auf der auch *GNU Zip* basiert. In der *zlib* werden verschiedenste Ausgabemechanismen bereitgestellt, die das Erzeugen, Lesen und Schreiben zip-komprimierter Dateien erlauben. Die Ausgaberroutinen ähneln dabei stark denen der Standardausgabe unter *C*. Im Vergleich: Die normale Ausgabe...

```
void sn_print_nm_header(unsigned pc, unsigned ins,
    char* opc){
    fprintf(ntf, "PCN = %.8x ", pc);
    fprintf(ntf, "OPC = %.8x ", ins);
```

<sup>7</sup>1.000.000 \* 650 Bytes = 650.000.000 Bytes = 634765,625kB = 619,88MB.

```

        fprintf(ntf, "MNC = %s\n", opc);
    }

```

... und die komprimierte Ausgabe

```

void sn_print_gz_header(unsigned pc, unsigned ins,
    char* opc){
    gzprintf(gtf, "PCN = %.8x ", pc);
    gzprintf(gtf, "OPC = %.8x ", ins);
    gzprintf(gtf, "MNC = %s\n", opc);
}

```

Da es sich bei den Traces um rein textuelle Daten handelt, lässt sich auf diese Art und Weise ein Kompressionseffekt von über 90% erreichen, bei gleichzeitiger Kompatibilität mit einem der verbreitetsten Kompressionsstandards. Neben dem starken Kompressionseffekt ist auch die einfache Weiterverarbeitung der komprimierten Traces ein großer Vorteil. Zum Einen steht auf jeder modernen Plattform ein Ableger der *zlib* zur Verfügung, zum Anderen stellt *Java* reichhaltige Mechanismen zum Öffnen und Lesen zip-komprimierter Dateien bereit. Natürlich beeinflusst jeder zusätzliche Arbeitsschritt die Gesamtperformance der Tracing-Software und hat damit unter Umständen Auswirkungen auf die Zielanwendung. Im Falle der komprimierten Trace-Erzeugung sind diese Auswirkungen jedoch sehr gering und daher zu vernachlässigen. Bei den bisherigen Trace-Vorgängen konnten keine Unterschiede zwischen komprimierten und nichtkomprimierten Traces festgestellt werden.

### 3.3 *snoop* im Einsatz

Nachdem in Abschnitt 3.2 der Aufbau der Tracing-Software vorgestellt wurde, wird im Folgenden auf die einzelnen Möglichkeiten von *snoop* und deren Aufrufsemantik eingegangen.

Da es sich bei *snoop* um ein Kommandozeilenprogramm handelt, entfällt die Notwen-

digkeit einer grafischen Oberfläche. Der Tracer präsentiert sich dem Benutzer in der Kommandozeile, wie in Abbildung 3.4 dargestellt.

```

*****
|__SNOOP_V.2.4.1_for_PowerPC_____Tracing_Made_Easy!_____
|
|  USAGE: snoop -s <#skiplines> [-f|-r|-c|-m <#max>|-x|-z] -t <program> -- [progopts]
|
|  -s: number of skipped instructions.      (REQUIRED).
|  -t: target program which is traced.     (REQUIRED).
|  -r: record general-purpose registers.   (RECOMMENDED).
|  -f: record floating-point registers.
|  -m: maximum number of traced instructions.
|  -c: trace only changed registers.     (RECOMMENDED FOR SMALLER TRACES).
|  -x: extended tracing, logs MSR, XER, CR, etc..
|  -z: zip trace file.                    (RECOMMENDED).
|
|-----
| EXAMPLE: snoop -s 1000 -x -c -z -t zip -- -r test.zip /mydir
|-----
| ratools-Users do snoop -s XXXX -x -c [-z] -t target -- [progopts]
|-----
| ATTENTION: SNOOP trace files grow VERY FAST and
|             they can become VERY LARGE! USE OPTION -z
|-----
|__ (C) 2002/2003 C.Braun _____University_Of_Tuebingen_____
*****

```

Abbildung 3.4: *snoop* in der Kommandozeile

Ein Standardaufruf für die Erzeugung von Traces für das Simulationssystem sieht wie folgt aus:

```
snoop -s 10000 -x -c -z -t zip -- -r src.zip /mysources
```

Es werden die ersten 10000 Instruktionen ausgelassen und dann alle Status-Register, sowie die jeweils veränderten General-Purpose- und Floating-Point-Register erfasst. Der gesamte Trace wird im komprimierten Format *.stf.gz* erzeugt, wodurch der benötigte Speicherplatz deutlich reduziert wird. Das Target ist *zip*, dessen eigene Optionen nach dem Doppelstrich angegeben werden. Die folgenden Optionen stellt die Tracing-Software dem Benutzer zur Verfügung:

- s – gibt die Anzahl der Instruktionen an, die ausgelassen werden soll.
- t – spezifiziert die Zielanwendung.
- r – alle General-Purpose-Register protokollieren.
- f – alle Floating-Point-Register protokollieren.
- m – gibt die maximale Anzahl zu protokollierender Instruktionen an.
- c – es werden nur Register aufgezeichnet, die sich verändert haben.
- x – alle Status-Register protokollieren.
- z – den Trace komprimiert erzeugen.

Die Optionen **-s** und **-t** müssen immer angegeben werden, die übrigen Optionen sind frei wählbar und können beliebig miteinander verknüpft werden. Für die erzeugung eines typischen Instruction-Traces wird im Normalfall die Option **-c** verwendet,

da nur die jeweils an der Instruktion beteiligten Register interessant sind. Ein zweiter Aspekt bei diesem Vorgehen ist wieder die Einsparung von wertvollem Speicherplatz. Wenn man sich vor Augen führt, dass für einen gewöhnlichen Registereintrag der Form

```
R00 = 00000000
```

im Durchschnitt 15 Bytes an Daten anfallen<sup>8</sup>, und dass dies bei der vollen Erfassung der beiden Hauptregistersätze vierundsechzig Mal geschieht, so erhält man pro Instruktion rund 1216 Bytes. Betrachtet man sich in diesem Zusammenhang jedoch den Befehlssatz des PowerPC, so stellt sich heraus, dass zahlreiche Befehle überhaupt nicht direkt schreibend auf Register zugreifen und bei den anderen maximal 4 Register involviert sind. Angenommen, es wären pro Instruktion jeweils zwei General-Purpose-Register beteiligt, so ergebe sich bei einem Trace mit einer Million Befehlen eine Datenmenge von rund 30MB. Dem gegenüber stehen circa 1.160MB bei der vollständigen Protokollierung der Hauptregistersätze.

### 3.4 Plausibilität der Ergebnisse

Ein wichtiger Aspekt bei der Erzeugung von Traces ist natürlich deren Korrektheit. Im Bezug auf die Richtigkeit des erzeugten Traces stellt sich grundsätzlich die Frage, ob die richtigen Daten von der Tracing-Software erfasst werden. Diese Frage lässt sich anhand der Implementation des Tracers und der Konformität zu den Vorgaben des Betriebssystems beantworten. Ob die erfassten Daten jedoch wirklich sinnvoll sind, lässt sich auf diesem Weg nicht bestätigen. Der sicherste Weg die absolute Korrektheit eines Traces zu bestätigen, wäre die Simulation mittels eines PowerPC-Prozessorsimulators. Lässt sich aus dem Trace der wirkliche Ablauf der Zielanwendung rekonstruieren und erhält man am Ende die identischen Ergebnisse wie bei der normalen Ausführung, so kann man davon ausgehen, dass der Trace korrekt ist. Der Aufwand der mit diesem Verifikationsansatz verbunden ist, ist jedoch außerordentlich hoch, da hier auf entsprechende Spezialsoftware zurückgegriffen werden muss, die unter Umständen nicht frei verfügbar ist. Eine Verifikation ist daher nur in sehr engen Grenzen durchführbar, da die Komplexität der Aufgabe in der Größe der Traces liegt. Wenn man davon ausgeht, dass ein Target mit einer Million Instruktionen rund 620MB Daten entspricht, so wird schnell deutlich, dass eine komplette Verifizierung eines solchen Traces nahezu unmöglich ist.

Ein anderer gangbarer Weg ist die genaue Untersuchung zufällig ausgewählter Segmente des Traces. Auf diese Art und Weise lässt sich zum Beispiel überprüfen, welche Instruktionen aufeinander folgen und welche Register wie verwendet werden. Auch die Auswirkungen der einzelnen Instruktionen sind so sichtbar. Abbildung 3.5 zeigt zwei aufeinander folgende Instruktionen mit den dazugehörigen Registersätzen. Hier lässt sich die Überprüfung zufällig ausgewählter Tracesegmente demonstrieren:

---

<sup>8</sup>Bei den Floating-Point-Registern sind es 23 Bytes pro Registereintrag

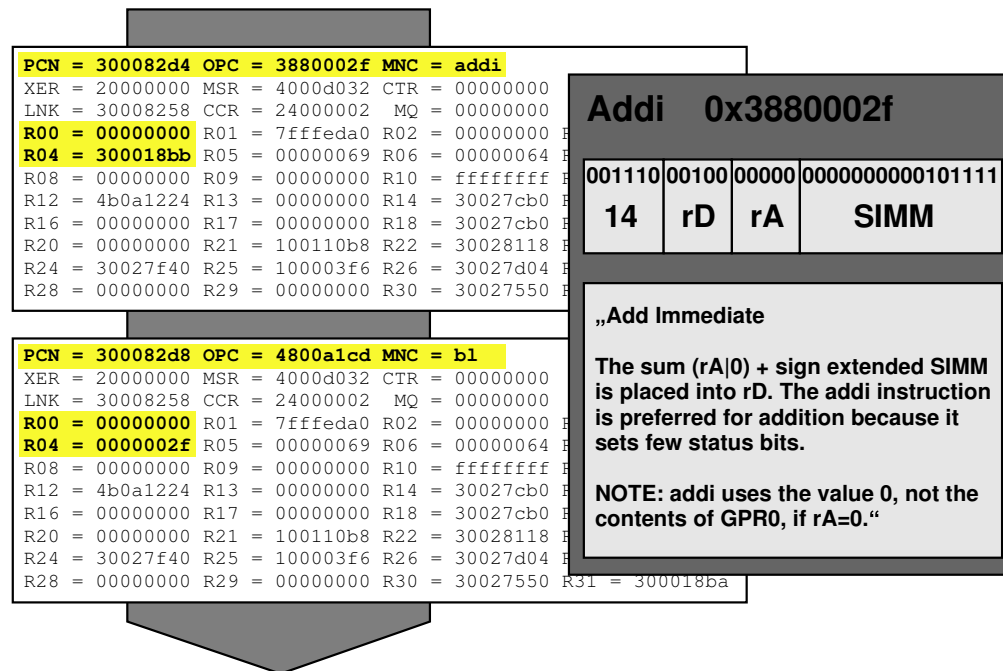


Abbildung 3.5: Plausibilitätsuntersuchung eines Traces

Bei der ausgesuchten Instruktion handelt es sich um `addi`<sup>9</sup>. Der dekodierte Opcode ist in Abbildung 3.5 dargestellt. Die ersten sechs Bits kodieren die Instruktionsgruppe, in diesem Fall 14. Die zwei folgenden 5-Bit-Gruppen codieren die involvierten Register. Ausgangsregister ist R00 und Zielregister ist R04. `Addi` addiert zum Ausgangsregister rA (R00) den Wert des Immediate-Anteils, hier  $2F_{16} = 47_{10}$ , hinzu und legt ihn in Register R04 ab. Bei der darauf folgenden Branch-Instruktion kann man sehen, dass sich der Wert des Registers R04 korrekt verändert hat. Dieser Schritt ist also korrekt abgearbeitet worden. Auf diese Art und Weise lassen sich auch sehr große Traces durch zufällige Stichproben auf ihre Plausibilität überprüfen, was für den geplanten Anwendungszweck der Traces ausreichend ist. Die gewünschte Sicherheit kann durch die Anzahl der Stichproben beliebig erhöht werden. Der oben beschriebene Weg der Plausibilitätsprüfung hat sich im bisherigen Verlauf der Trace-Erzeugung als sehr praktikabel erwiesen. Bei keiner der zahlreichen Stichproben, die bisher ausgewertet wurden, sind Ungereimtheiten aufgetreten, was die hohe Zuverlässigkeit der Tracing-Software unterstreicht.

<sup>9</sup>addi = Add Immediate

## Kapitel 4

# Integration der PowerPC-Architektur

In diesem abschließenden Kapitel wird nun der zweite große Teil der Studienarbeit vorgestellt, die Integration der PowerPC-Architektur in das Simulationssystem. Wie in der Einleitung bereits beschrieben, handelt es sich bei dem System um ein offenes Java-Framework, das dem Benutzer eine Simulationsumgebung in Form verschiedener Klassen zur Verfügung stellt. Diese Klassen erledigen alle notwendigen Aufgaben, die rund um das implementierte Simulationsziel anfallen. Es gibt Komponenten, die die Verarbeitung der Trace-Dateien übernehmen und es gibt Komponenten die der Kommunikation dienen. Man kann diese verschiedenen Teile nach ihrem Aufgabengebiet und ihrer Abhängigkeit von der Plattform ordnen. So gibt es zum Beispiel eine Familie von Klassen, die die Grundfunktionalität einer Instruktion innerhalb des Simulationssystems beinhalten. Diese Klassen sind plattformunabhängig und können daher universell eingesetzt werden. Von ihnen werden die spezifischen Komponenten für die Unterstützung einer bestimmten Architektur abgeleitet. Da das Simulationssystem bis zum Beginn dieser Studienarbeit ausschließlich die S/390-Architektur von IBM unterstützt hat und sich diese doch in vielen Belangen deutlich von der PowerPC-Architektur unterscheidet, war eine komplette Neuimplementierung zahlreicher Klassen notwendig. Es musste unter anderem eine neue Klasse für den Umgang und die Aufbereitung der erzeugten *snoop*-Traces entwickelt und darüber hinaus der komplette PowerPC-Befehlssatz abgebildet werden. Im folgenden Abschnitt 4.1 wird nun zuerst die Klasse *TracePPC* vorgestellt, die die Verarbeitung der *snoop*-Traces übernimmt. Sie steht in enger Verbindung mit der *InstructionPPC*-Klasse, einer sogenannten *factory*, die für die Erzeugung der entsprechenden PowerPC-Instruktionsobjekte verantwortlich ist. Der Datenaustausch zwischen diesen beiden Kernkomponenten erfolgt über die Klasse *InstructionInfo*, die ebenfalls beschrieben wird.

### 4.1 Verarbeitung von *snoop*-Traces

Um die Simulation neuer Verfahren innerhalb des bestehenden Systems auf der Basis der PowerPC-Architektur zu ermöglichen, muss die Versorgung des implementierten Simulationsgegenstandes mit kompatiblen PowerPC-Instruktionsobjekten gewährleistet sein. Die Quelle für diese Instruktionsobjekte stellen die mittels *snoop* erzeugten

Traces dar. Prinzipiell muss dabei sowohl auf die Standardversion (.stf) als auch auf die komprimierte Form der Traces (.stf.gz) zugegriffen werden können, obwohl in der Praxis die Verwendung der komprimierten Variante vorherrschend sein wird. Die Klasse *TracePPC* enthält alle notwendigen Methoden, um auf die Informationen innerhalb der beiden genannten Traceformate zugreifen zu können. Abbildung 4.1 zeigt das zugehörige Klassendiagramm.

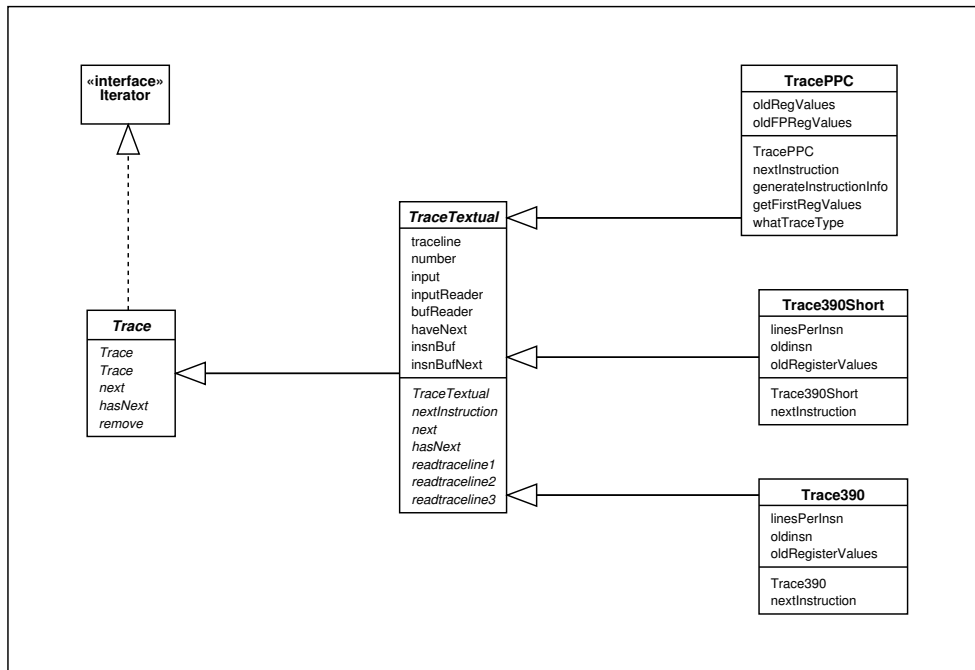


Abbildung 4.1: Klassendiagramm zu TracePPC

Initial stellt das Simulationssystem die Grundklasse *Trace* und die davon abgeleitete Klasse *TraceTextual* zur Verfügung. *Trace* enthält nur grundlegende Definitionen, jedoch keine nutzbaren Methoden. Das Erkennen und Öffnen von Trace-Dateien ist in der Klasse *TraceTextual* implementiert. Mit Hilfe der von Java bereit gestellten *InputStream*-Klassen können sowohl komprimierte als auch unkomprimierte Traces geöffnet werden. Diese Funktionalität ist innerhalb des Konstruktors der Klasse *TraceTextual* realisiert und im folgenden Codesegment dargestellt.

```

public TraceTextual(String filename) {
    input=null;
    number=0;
    if(filename.equals("-")) {
        input = System.in;
    } else {
        File tracefile = new File(filename);
        if (!tracefile.exists() || !tracefile.canRead() ||
!tracefile.isFile()) {
            System.out.println("Can't read trace file " + tracefile);
            return;
        }
        try {
            input = new FileInputStream(tracefile);
        }
    }
}
  
```

```

        catch ( FileNotFoundException e ) {
            System.out.println("Huh? Trace file disappeared!");
            return;
        }
    }
    if(filename.endsWith(".gz")||filename.endsWith(".Z")) {
        try {
            input = new GZIPInputStream(input, 4*1024);
        }
        catch (IOException e) {
            System.out.println("Can't create GZIP stream!");
            input=null;
            return;
        }
    }
    bufReader = new BufferedReader(new InputStreamReader(input));
}

```

Dieser Kode ist plattformunabhängig und wird daher sowohl von den bereits vorhandenen Klassen *Trace390* und *Trace390Short* als auch von der neu implementierten Klasse *TracePPC* verwendet. Alle drei Klassen setzen für den weiteren Umgang mit den Trace-Dateien auf dem *BufferedReader* auf. Neben dieser grundlegenden Funktionalität enthält die Klasse *TraceTextual* darüber hinaus noch weitere Hilfsmethoden für das Lesen innerhalb des Traces. Diese werden allerdings nur von den Klassen des S/390-Teils verwendet und daher hier nicht weiter beschrieben. Für den Benutzer stellt die Klasse *TracePPC* den eigentlichen Anlaufpunkt dar. Sie wird von ihm in seiner Simulation instanziiert und ermöglicht ihm so den Zugriff auf den Trace. Der erste Aufruf innerhalb des Konstruktors von *TracePPC* ist

```
super(filename);
```

wodurch die oben beschriebenen Öffnungssequenzen angestoßen werden. Nach diesem Aufruf steht der *BufferedReader* für die Methoden der Klasse zur Verfügung. Im nächsten Schritt werden zwei Felder (*oldRegValues[]* bzw. *oldFPRegisterValues[]*) für die Zwischenspeicherung der gelesenen Registerwerte angelegt. Dies ist notwendig, da für die spätere Erzeugung des entsprechenden Instruktionsobjektes immer die aktuellen und die vorangegangenen Registerinformationen benötigt werden. Das Feld für die Floating-Point-Register hat 32 Einträge, das Feld der General-Purpose-Register bietet zusätzlich noch den 8 Statusregistern Platz. Für die Speicherung verwendet *TracePPC* die Typen *RegisterValue* und *FPRegisterValue*. Bei beiden Typen handelt es sich um abgeleitete Subklassen des übergeordneten *Value*-Typs. Die *Value*-Klassen vereinfachen die Handhabung der verschiedenen Registerinformationen und beinhalten neben einer Variable für den Wert auch zusätzliche set- und get-Methoden. Im Falle der General-Purpose-Register wird der gewöhnliche *RegisterValue* verwendet, der für die Speicherung eine Variable vom Typ *int* benutzt. Im Gegensatz dazu verwenden die *FPRegisterValues* den Typ *long*. Für die bessere Unterscheidung der Floating-Point-Register bei Single-Precision- und Double-Precision-Instruktionen wurden zusätzlich die Typen *SPFPRegisterValue*<sup>1</sup> und *DPFPRegisterValue*<sup>2</sup> eingeführt. Diese Value-Klassen sind plattformabhängig und können nur im Zusammenhang mit der PowerPC-Architektur verwendet werden.

<sup>1</sup>Single-Precision-FPRegisterValue

<sup>2</sup>Double-Precision-FPRegisterValue

Die Belegung der Puffer mit den initialen Registerinformationen erfolgt vor dem Lesen der ersten Instruktion und wird von der Methode *getFirstRegValues()* übernommen. Ab dem jetzigen Zeitpunkt ist alles bereit für den Beginn der Simulation. Um ein neues Instruktionsobjekt zu erhalten, muss vom Benutzer nur die Methode *nextInstruction()* aufgerufen werden. Um eine möglichst gute Kompatibilität mit den bisher implementierten Simulationen zu gewährleisten, unterscheidet sich dieser Aufruf nicht von denen des S/390-Teils, der Benutzer muss folglich nur *Trace390* durch *TracePPC* ersetzen.

Um den Weg zwischen der Klasse *TracePPC* und der factory *InstructionPPC* zu überbrücken, wird ein Containerobjekt benötigt, in dem sich alle relevanten Daten einheitlich ablegen lassen. Diese Transportaufgabe übernimmt die Klasse *InstructionInfo*. Abbildung 4.2 zeigt den Aufbau dieser Klasse.

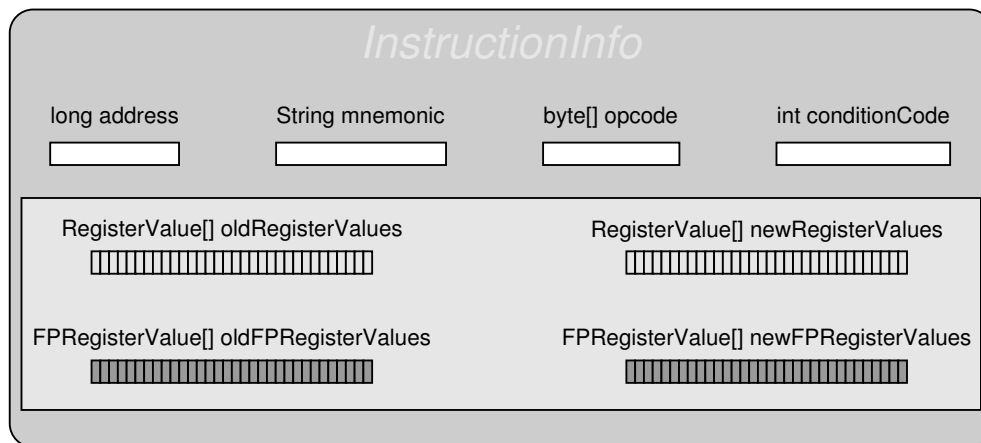


Abbildung 4.2: Informationstransport mit *InstructionInfo*

Die aus dem Trace erhaltenen Daten werden in den entsprechenden Variablen der *InstructionInfo* gespeichert. Bei der aktuellen Implementierung der PowerPC-Architektur wird die Variable für den ConditionCode nicht genutzt. Die Klasse *InstructionInfo* gehört zum Kern des Simulationssystems und wird daher auch in derselben Form für die S/390-Architektur verwendet. Die einzige Modifikation, die vorgenommen werden musste, ist die Einführung der beiden Felder *oldFPRegisterValues[]* und *newFPRegisterValues[]* für die Floating-Point-Register. Diese beeinträchtigen jedoch den S/390-Teil im Augenblick nicht, da hier bisher keine Traces mit FP-Registerinformationen verarbeitet werden. Unter Umständen müssen zu einem späteren Zeitpunkt für den S/390-Teil eigene FP-Values implementiert werden.

Erzeugt und mit den entsprechenden Informationen versehen werden die *InstructionInfo*-Objekte in der *generateInstructionInfo()*-Methode, die von *nextInstruction()* aufgerufen wird. Hier erfolgen die eigentlichen lesenden Zugriffe auf den Trace. Gelesen werden die Trace-Daten mit Hilfe eines *StringTokenizer*s, der von Java zur Verfügung gestellt wird. Damit ist es möglich, eine über den *BufferedReader* gelesene Zeile des Traces in ihre einzelnen Bestandteile zu zerlegen und diese in einem String-Feld abzulegen. Über die einzelnen Tokens kann abgefragt werden, was für eine Zeile gelesen wurde. Beginnt der Eintrag mit "PCN", so handelt es sich um eine Adresszeile, beginnt er mit "R" oder "F", so handelt es sich um eine Registerzeile. Theoretisch ist der Lesevorgang nicht allzu kompliziert, da eigentlich immer ähnliche Daten gelesen werden. Aus diesem Grund ist auch die Verwendung einer eigenen Grammatik an dieser Stelle unnötig. Tatsächlich stellt jedoch das dynamische Auftreten der einzelnen Zeilen eine gewisse Herausforderung an die Leseroutine dar. Handelt

es sich zum Beispiel um einen Sprungbefehl, so werden nur die Zeilen mit der Adresse, dem Opcode und dem Mnemonic, sowie den Statusregistern gelesen. Wurden jedoch im Rahmen der Instruktion auch Register verändert, so muss unter Umständen noch eine weitere Registerzeile gelesen werden. Um die flexible Erfassung der notwendigen Daten zu ermöglichen, verfügt die Leseroutine über eine mehrfach geschichtete Abfragestruktur, die die jeweils gelesenen Tokens prüft und entsprechend reagiert. Unterstützt wird diese Prüfung der Tokens durch die vorgegebene Reihenfolge der einzelnen Informationsteile innerhalb des Traces. Wurde im letzten Durchgang zum Beispiel eine Zeile mit General-Purpose-Registern gelesen, so darf die nächste Zeile nur Floating-Point-Register oder die Adresszeile der nächsten Instruktion enthalten. Das folgende Codesegment zeigt die Einleseroutine für die General-Purpose-Register.

```

if(readLine.startsWith("R")){

    // that's OK, we read a line with general-purpose registers
    // ...st to handle readLine...
    st = new StringTokenizer(readLine);

    // ...how much tokens do we have?
    countTokens = st.countTokens();

    // we store our tokens here...
    tokenizedLine = new String[countTokens];

    // let's fill our storage...
    for(int i=0; i<countTokens;i++){
        tokenizedLine[i] = st.nextToken();
    }

    int regidx = 0;
    int regval = 0;
    for(int i=0; i<countTokens/3; i++){
        regidx = Integer.parseInt(tokenizedLine[i*3].substring(1,3));
        regval = Long.parseLong(tokenizedLine[(i*3)+2],16);
        registerValues[regidx].setValue((int)regval);
        registerValues[regidx].setKnownValue(true);
    }
}

```

*readLine* ist eine *String*-Variable, die die letzte vollständige Zeile enthält, die vom *BufferedReader* gelesen wurde. Sie wird nun mit Hilfe des *StringTokenizer* *st* in ihre Bestandteile zerlegt und in einem Feld des Typs *String* abgelegt. Da die Registerinformation in der Trace-Datei immer in der Form

```
R24 = 00ffe001
```

gespeichert ist, muss aus dem Registerbezeichner (hier R24) der entsprechende Feldindex (*regidx*) extrahiert werden. Mit diesem Index kann dann der Wert (*regval*) an der richtigen Stelle des *Value*-Feldes *registerValues[]* gespeichert werden. Dieser Schritt ist im unteren Drittel des obigen Codesegments zu sehen. Für jede Instruktion wird immer solange weiter gelesen, bis der Beginn des nächsten Befehls erkannt wird. Da der *BufferedReader* immer an der Stelle des letzten Lesezugriffs verweilt, bis er erneut angesprochen wird, muss mit sogenannten Markern gearbeitet werden. Mit Hilfe der Marker lassen sich vor jedem Zugriff auf eine Zeile des Traces Rücksprungpunkte definieren, die durch den Aufruf der *reset()*-Methode des

`BufferedReaders` angesprungen werden können. Notwendig wird dieses Vorgehen durch die oben beschriebene Art der Auswertung. Da die Adresszeile der nächsten Instruktion das Abbruchkriterium jedes Lesevorgangs darstellt, wird diese auch immer gelesen. Würde nun ohne Rücksprung vor diese Zeile der nächste Zugriff erfolgen, käme es zu schwerwiegenden Verschiebungen der Daten und damit zu einem Abbruch. Nachdem alle wichtigen Daten der aktuellen Instruktion an diesem Punkt ausgelesen sind, können sie im neu erzeugten *InstructionInfo*-Objekt abgelegt werden. Der fertige Informationscontainer wird daraufhin zurückgegeben und an die Klasse *InstructionPPC* übergeben.

## 4.2 Instruktionsobjekte und ihre Erzeugung

Ein wichtiger Kernbestandteil des Systems sind die Instruktionen. Sie sind für die Simulation notwendig und bilden deren Informationsgrundlage. Um eine einheitliche Verarbeitung der Informationen aus dem Trace gewährleisten zu können, müssen die Instruktionen in einer sinnvollen Form aufbereitet werden. Es bietet sich daher an, den gesamten Befehlssatz der PowerPC-Architektur auf eine Objekthierarchie abzubilden. Das Simulationssystem gibt eine Grundform für diese Instruktionsobjekte vor, die sowohl von der S/390- als auch von der PowerPC-Architektur eingehalten werden muss.

Neben den grundlegenden Informationen wie Adresse, Opcode und Mnemonic, soll auch der Datenfluss für die Simulation sichtbar werden. Hierfür ist es notwendig, dem System die verwendeten Speicher- und Registerwerte kenntlich zu machen. Aus diesem Grund enthält ein Instruktionsobjekt neben den oben genannten Daten immer drei Felder des Typs *Value* namens *input[]*, *output[]* und *kill[]*. In das *input*-Feld werden alle beteiligten Register einer Instruktion eingetragen. Bei einer Addition wären dies zum Beispiel die beiden Register, die die Summanden enthalten. In das *output*-Feld werden die oder das Zielregister der Instruktion geschrieben. Da bei solchen Befehlen Register überschrieben werden, ist es für die Simulation interessant, sowohl den alten als auch den neuen Wert eines Registers zu kennen. Dies wird mit dem dritten Feld, *kill[]*, realisiert. Hier werden die alten Inhalte der Zielregister abgelegt. Diese Aufteilung der relevanten Daten reduziert zum Einen die Menge der Daten pro Instruktion auf die wirklich notwendigen und ermöglicht es der Simulation zum Anderen den Datenfluss nachzuvollziehen.

Da es viele Instruktionen gibt, die sehr ähnlich aufgebaut sind oder sogar zu einer gemeinsamen Befehlsgruppe gehören, bietet es sich an, die Methoden für das Setzen der obigen drei Felder in einer übergeordneten Klasse zusammenzufassen. Abbildung 4.3 zeigt das Klassendiagramm für die Instruktionsobjekte. Ausgangspunkt ist die abstrakte Klasse *Instruction*, von der wiederum die abstrakte Klasse *ManagedInstruction* abgeleitet wird. Beide Klassen gehören zum Kern des Simulationssystems und werden von allen unterstützten Architekturen verwendet. Im Falle der PowerPC-Architektur setzt auf der Klasse *ManagedInstruction* die *factory InstructionPPC* auf. Der Begriff der *factory* beschreibt in diesem Zusammenhang eine Abstraktionsebene, die zwischen den eigentlichen Instruktionsobjekten und der Klasse *TracePPC* eingeführt wird. Prinzipiell werden von der übergeordneten Trace-Klasse immer nur Objekte des Typs *InstructionPPC* erzeugt, der zugehörige "Bauplan" wird mit der *InstructionInfo* übermit-

telt. Welche Instruktion genau erzeugt wird ist auf dieser hohen Ebene nicht ersichtlich und nicht von Belang. Die eigentliche Typisierung des jeweiligen Objektes erfolgt in der Klasse *InstructionPPC*, die anhand der Informationen aus der *InstructionInfo* die entsprechende Subklasse instanziiert und zurück gibt. Dieses Vorgehen hat unter anderem den Vorteil, dass große Änderungen oder Erweiterungen innerhalb des Befehlsatzes nie Auswirkungen nach oben haben. Kommen zum Beispiel mit zukünftigen Generationen der PowerPC-Architektur zusätzliche neue Befehle hinzu, so müssen diese nur als Klasse implementiert und der *factory* bekannt gemacht werden, ansonsten ändert sich nichts.

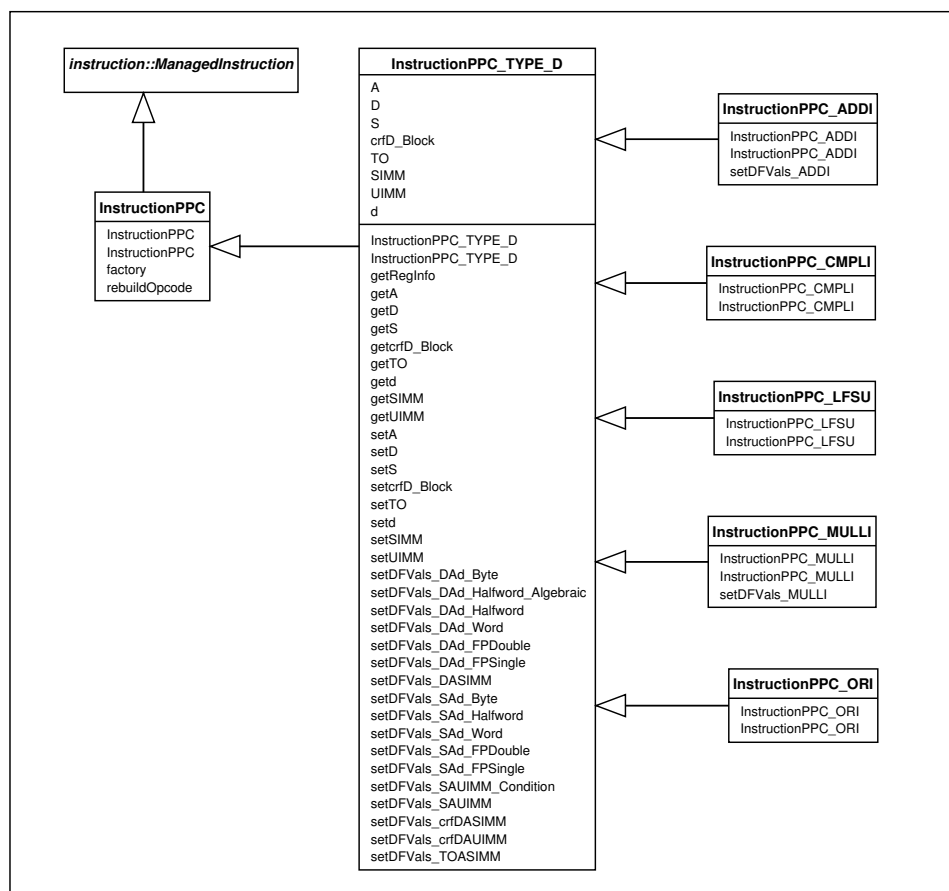


Abbildung 4.3: Klassendiagramm zu InstructionPPC

Abbildung 4.3 zeigt die vorgestellten Klassen und die *factory* *InstructionPPC*. *InstructionPPC\_TYPE\_D* ist eine der bereits erwähnten übergeordneten Klassen, die die gemeinsamen Eigenschaften entsprechender Instruktionen zusammenfaßt. Die zusammengefassten Befehle können dabei aus den unterschiedlichsten Bereichen stammen, sie müssen nur bestimmte Ähnlichkeiten oder Übereinstimmungen in ihrer Form aufweisen.

Kern der Klasse *InstructionPPC* ist eine mehrfach verschachtelte *if*-Treppe, die die Opcodes aus der *InstructionInfo* zur Bestimmung des Befehls nutzt. Da die Tracing-Software *snoop* immer den Opcode und den dekodierten Mnemonic im Trace zur

Verfügung stellt, ist diese zweite Dekodierung prinzipiell nicht notwendig. Um dem System jedoch auch die Verarbeitung fremder PowerPC-Traces zu ermöglichen, wurde diese zusätzliche Dekodierungsstufe implementiert. Der Opcode einer Instruktion, wie ihn die Klasse *TracePPC* ausliest, wird innerhalb der *InstructionInfo* in einem Feld mit acht Stellen gespeichert. Um aus den acht Segmenten nun den kompletten Opcode wiederherstellen zu können, verfügt die *factory* über eine Methode namens *rebuildOpcode()*. Durch shiften und Addition der einzelnen Segmente wird der neue Opcode erzeugt und in einer Variable des Typs `long` abgelegt. Mit Hilfe des Opcodes und der Dekodierungsstufe können die einzelnen Instruktionen zuverlässig identifiziert und erzeugt werden.

Die folgende Abbildung 4.4 zeigt den beschriebenen Ablauf der Entstehung eines Instruktionsobjektes nochmal im Überblick.

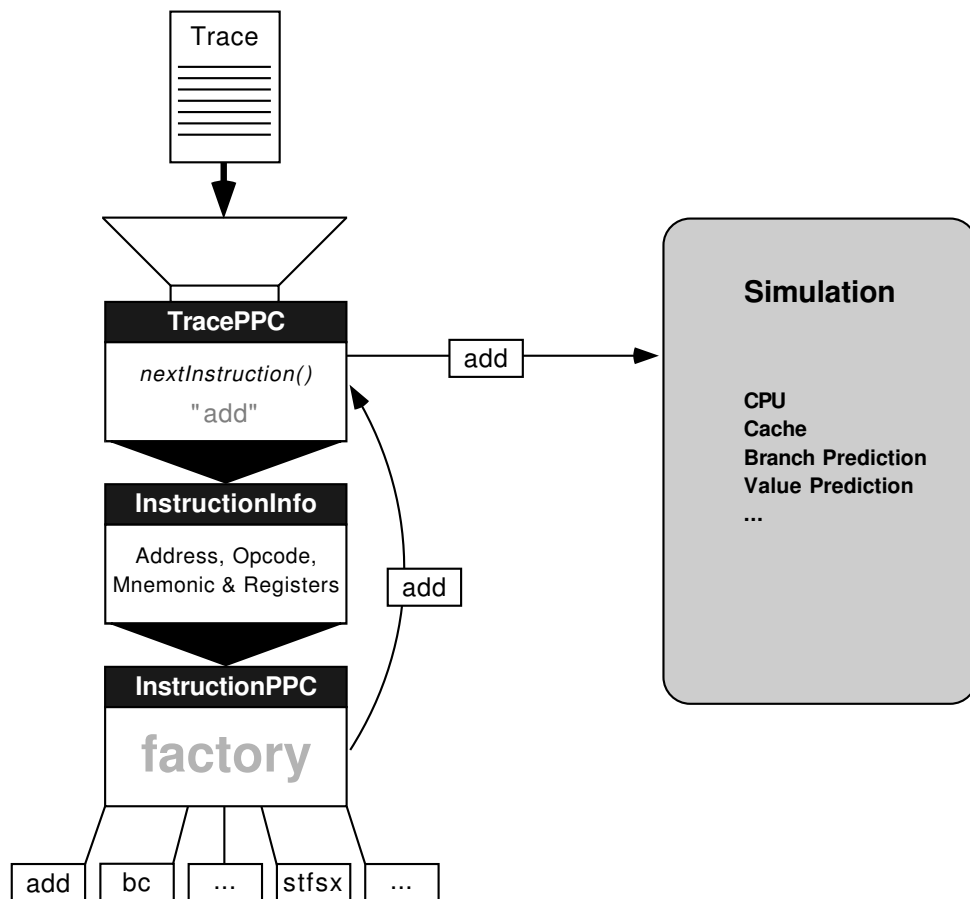


Abbildung 4.4: Von der Trace-Datei zur Simulation

Die aus dem Trace ausgelesenen Informationen über die jeweilige Instruktion und die dazugehörigen Registersätze wird in der *InstructionInfo* abgelegt und der *factory* übergeben. Diese wiederum erzeugt anhand des übermittelten "Bauplans" das passende Instruktionsobjekt und gibt es an die Klasse *TracePPC* zurück. Im letzten Schritt stellt diese das fertige Instruktionsobjekt der Simulation des Benutzers zur Verfügung.

### 4.3 Performance

Wie bei allen modernen Softwaresystemen spielt auch bei den beschriebenen Simulationen die Performance eine wichtige Rolle. Um eine möglichst geringe Simulationszeit erreichen zu können, ist es notwendig, die einzelnen Komponenten weitgehend zu optimieren. Im Allgemeinen liegt der größte Zeitaufwand bei der Behandlung der Traces. Das Öffnen des Traces und vor allem das Lesen und Erzeugen der Instruktionen trägt hier einen großen Teil zur Gesamtlaufzeit bei. Bei der Implementierung dieser Komponenten wurde daher besonders auf den sorgfältigen Umgang mit den Ressourcen Wert gelegt. Die Simulation selbst trägt ebenfalls einen Teil zur Ausführungszeit bei. Dieser Anteil kann jedoch nicht allgemein angegeben werden, da er in besonderem Maße von der Komplexität des implementierten Verfahrens abhängt. Um einen Eindruck von der Performance des Grundsystems zu bekommen, wurde ein kleines Testprogramm entwickelt, das einen Trace vollständig durchläuft und alle gelesenen Instruktionen erzeugen lässt. Die folgende Performancemessung erfolgte auf einem PowerPC System<sup>3</sup> unter MacOS X. Der Testrechner verfügte dabei über 128MB Hauptspeicher und einen 233MHz PowerPC 604e. Die Tabelle 4.1 zeigt die Ergebnisse dieser Untersuchung

Trace	sntest1.stf.gz	sntest3.stf.gz	compress.stf.gz
<b>Größe (komp.)</b>	3MB	3,4MB	10,1MB
<b>Größe (norm.)</b>	65,9MB	76,7MB	349,6MB
<b>Instruktionen</b>	443.823	516.947	2.313.904
<b>Start</b>	16:35:00	16:47:05	17:22:21
<b>Ende</b>	16:43:20	16:56:42	18:01:55
<b>Dauer</b>	500s	577s	2374s
<b>Instr./Sekunde</b>	888	896	975

Tabelle 4.1: Ergebnisse der Performancemessung

Bei den verwendeten Traces handelt es sich zum Einen um spezielle Testprogramme (sntest1 und sntest3), die verschiedene Berechnungen durchführen und einen großen Ausgabeanteil besitzen. Der dritte Trace (compress) wurde bei der Komprimierung einer Textdatei mittels *compress* unter LinuxPPC erzeugt. Es zeigt sich, dass auf dem oben genannten System eine durchschnittliche Rate von 920 Instruktionen pro Sekunde erzielt werden kann. Nimmt man diesen Wert als grundlegenden Anhaltspunkt für die Performance und bezieht das Alter des Testrechners mit ein, so lässt sich für aktuelle Systeme der 3GHz-Klasse ein Performancewert von ungefähr 11.800 bis 12.000 Instruktionen pro Sekunde vorhersagen. Da die Gesamtleistung dieser Systeme weit höher liegt und zahlreiche Optimierungen in den letzten Jahren in die Prozessorarchitekturen eingeflossen sind, können die real erzielten Performancewerte auf diesen Systemen sogar noch höher liegen. Als Fazit dieser kleinen Messung läßt sich feststellen, dass ein Trace mit rund 10 Millionen Instruktionen in einer knappen viertel Stunde verarbeitet werden kann. Der zusätzliche Aufwand, der durch den eigentlichen Simulationsgegenstand entsteht, dürfte diesen Wert nicht gravierend erhöhen.

<sup>3</sup>Apple Power Macintosh 9600/233 aus dem Jahre 1997

## Kapitel 5

# Zusammenfassung

“*Methoden zur Unterstützung tracebasierter Simulation für die PowerPC-Architektur*” lautet der Titel der hier vorliegenden Studienarbeit. Die Aufgabenstellung der Arbeit umfasste zwei große Komplexe, die getrennt voneinander auf unterschiedlichen Systemen zu realisieren waren. Als Grundlage für spätere Simulationen sollte eine Methode zur softwarebasierten Erzeugung von Traces auf der PowerPC-Architektur entwickelt werden. Das Ergebnis dieser Entwicklung ist die Tracing-Software *snoop*, die auf der Basis des Systemaufrufs *ptrace* unter Linux für PowerPC entwickelt wurde. Der Tracer ermöglicht die Erzeugung von PowerPC-Instruction-Traces, die neben den grundlegenden Informationen über Adressen und Instruktionen auch alle Register der *User Instruction Set Architecture* enthalten können. Eine flexible Anpassung der Traces an die spätere Simulation ist dabei ebenso möglich, wie die direkte Komprimierung der gesammelten Daten. Bei der Entwicklung des Tracers wurde, im Hinblick auf eine spätere Portierung der Software auf eine andere Zielplattform, besonderer Wert auf eine Trennung der einzelnen Funktionen gelegt. Somit müssen nur einzelne Komponenten angepasst werden. Die bisher erzielten Ergebnisse des Tracers wurden stichprobenartig auf ihre Plausibilität überprüft und in Probesimulationen verwendet, wobei keine Fehler aufgetreten sind.

Der zweite Teil der Studienarbeit umfasste die Anpassung des Simulationssystems an die PowerPC-Architektur, beziehungsweise deren Integration in das System. Im Rahmen dieser Anpassung wurden bestehende Klassen, wie zum Beispiel die *Value*-Klassen, überarbeitet und zusätzliche neue Komponenten implementiert. Die Verarbeitung der *snoop*-Traces wird durch die Klasse *TracePPC* ermöglicht, die die Traces Öffnen und Lesen kann. Um der späteren Simulation die gewünschten Informationen zugänglich machen zu können, wurde der gesamte PowerPC-Befehlssatz auf eine Java-Objekthierarchie abgebildet. Jede Instruktion wurde hierfür in einer eigenen Klasse implementiert. Die Erzeugung der einzelnen Instruktionsobjekte wird dabei von einer sogenannten *factory* übernommen. Erste Performancemessungen des angepassten Grundsystems ergaben gute Werte, die auch die Simulation komplexer Verfahren in kurzer Zeit garantieren.

Auf den bisherigen Ergebnissen aufbauend, kann nun die Erzeugung verschiedenster Traces erfolgen. Als Grundlage hierfür sollen vor allem die Programme der *SPEC-Benchmark 2000 Suite* dienen. Diese Traces können dann in einer Sammlung dem Simulationssystem zur Verfügung gestellt werden.

# Anhang A

## PowerPC-Architekturebenen

Die folgende Abbildung A.1 stellt die drei unterschiedlichen Ebenen der PowerPC-Architektur und die dazugehörigen Register dar.

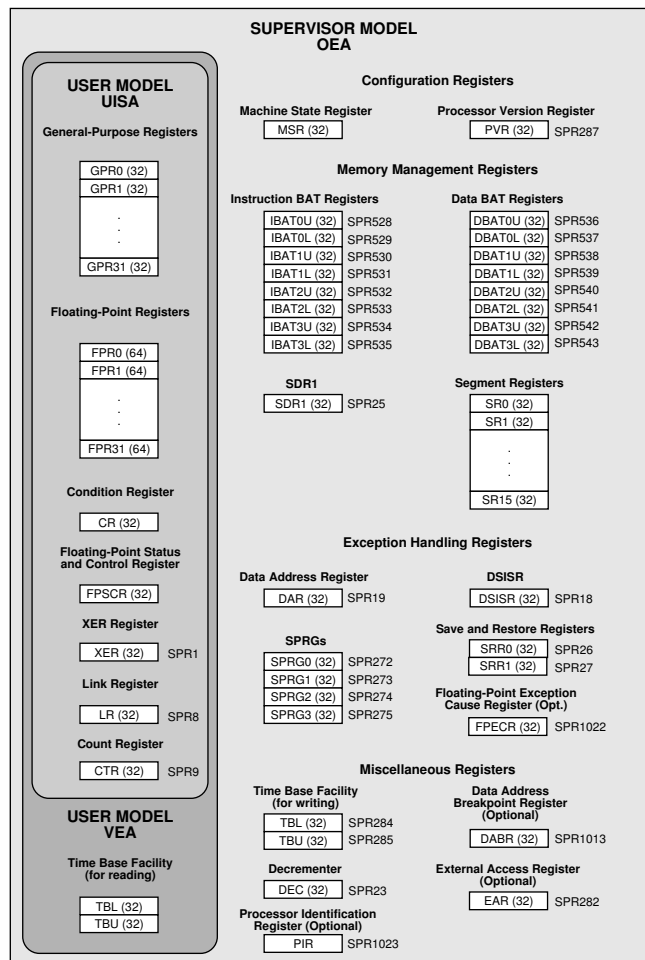


Abbildung A.1: Ebenen der PowerPC-Architektur

# Literaturverzeichnis

- [1] INTERNATIONAL BUSINESS MACHINES CORP.:  
*PowerPC Microprocessor Family – The Programming Environments for 32-Bit Microprocessors*  
International Business Machines Corp., (2000) 02/21/2000 Ident.-Nr.G522-0290-01
- [2] SANDON, P.A. ET AL.:  
*NStrace: A bus-driven instruction trace tool for PowerPC microprocessors*  
International Business Machines Corp., (1997) 04/30/1997
- [3] BRIGHAM YOUNG UNIVERSITY:  
*Performance Evaluation Lab*  
<http://traces.byu.edu/InstructionTraces/>
- [4] UNIVERSITY OF ALABAMA IN HUNTSVILLE - ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT:  
*LaCASA - Laboratory for Advanced Computer Architectures and Systems*  
<http://www.ece.uah.edu/lacasa/>
- [5] BASE.COM:  
*Sky - Kernel Simulator and Trace Generator*  
<http://www.base.com/gordoni/sky.html> <http://www.base.com/gordoni/sky/description/tech/traces.html>
- [6] UNIVERSITY OF WISCONSIN-MADISON:  
*WARTS - Wisconsin Architectural Research Tool Set*  
<http://www.cs.wisc.edu/larus/warts.html> <http://www.cs.wisc.edu/larus/qpt.html>
- [7] HENNESSY, JOHN L. / PATTERSON, DAVID A.:  
*Computer Architecture – A Quantitative Approach Second Edition*  
Morgan Kaufmann Publishers, (1996) ISBN 1-55860-329-8
- [8] VON STAUDT, HANS MARTIN:  
*Das professionelle PowerPC-Buch*  
Franzis-Verlag GmbH, 85586 Poing, (1994) ISBN 3-7723-6962-6
- [9] ECKEL, BRUCE:  
*Thinking in Java Second Edition*  
Prentice-Hall, (2000) ISBN 0-13-659723-8
- [10] LITWAK, KENNETH:  
*PURE Java 2 – A Code-Intensive Premium Reference*  
SAMS Publishing, (2000) ISBN 0-672-31654-4
- [11] ARNOLD, KEN / GOSLING:  
*The Java Programming Language Second Edition – The Java Series from the Source*  
Addison-Wesley Publishing, (1998) ISBN 0-201-31006-6

- [12] KERNIGHAN, BRIAN W. / RITCHIE, DENNIS M.:  
*Programmieren in C, 2. Ausgabe*  
Prentice-Hall / Verlage Carl Hanser, (1990) ISBN 3-446-15497-3
- [13] GULBINS, JÜRGEN / OBERMAYR, KARL:  
*UNIX System V.4, 4., überarbeitete Auflage*  
Springer Verlag, (1995) ISBN 3-540-58864-7