TECHNISCHE UNIVERSITÄT DRESDEN

Professur für Mikrorechner

Fakultät Informatik Institut für Technische Informatik Professur für Mikrorechner

Effiziente externe Beobachtung von CPU-Aktivitäten auf SoCs

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von Dipl.-Ing. Alexander Peter Weiß
geboren am 11. Februar 1967 in Dresden

eingereicht am 15. September 2014
verteidigt am 2. Oktober 2015

Betreuender Hochschullehrer:	Prof. Dr.-Ing. Christian Hochberger Technische Universität Darmstadt
Externer Gutachter:	Prof. Dr. sc. techn. Andreas Herkersdorf Technische Universität München

Danksagung

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. Christian Hochberger für sein besonderes Interesse am Thema der Arbeit, seine Betreuung, Motivation und Unterstützung.

Bedanken möchte ich mich auch ganz ausdrücklich bei Herrn Prof. Dr. sc. techn. Andreas Herkersdorf für die Übernahme des Zweitgutachtens.

Weiterhin möchte ich den weiteren Mitgliedern der Promotionskommission Herrn Prof. Dr. Sebastian Rudolph, dem Fachreferenten Herrn Prof. Dr. Hermann Härtig sowie dem Vorsitzenden Herrn Prof. Dr. Stefan Gumhold für die Sicherstellung des reibungslosen Ablaufes des Promotionsverfahrens danken.

Meinem Kollegen Dipl.-Ing. Alexander Lange danke ich für die langjährige Zusammenarbeit und die immer wieder spannende gemeinsame Suche nach unkonventionellen Lösungsansätzen.

Für die gemeinsame Projektarbeit danke ich Herrn Dipl.-Ing. Rico Backasch.

Für das frühzeitige Erkennen des Potentials der in dieser Arbeit beschriebenen Lösung sowie für seine Unterstützung bei deren industriellen Umsetzung danke ich Herrn Dipl.-Ing. Shyam Gupta.

Mein ganz besonderer Dank gilt Herrn Dipl.-Phys. Detlev Richardt für seine begeisternde Einführung in die Welt der Erfindungen und Patente. Weiterhin danke ich ihm, Franziska Moser sowie Herrn Dipl.-Ing. Karlheinz Kleinhenz für das Korrekturlesen der Arbeit.

Meiner Frau Katrin und meinen Kinder August, Julius und Johannes danke ich von Herzen für ihre Unterstützung, Geduld und Nachsicht während der Entstehung der Arbeit.

Kurzfassung

Die umfassende Beobachtbarkeit von System-on-Chips (SoCs) ist eine wichtige Voraussetzung für das effiziente Testen und Debuggen eingebetteter Systeme.

Ausgehend von einer Analyse verschiedener Anwendungsfälle ergibt sich ein Katalog von Anforderungen an die Beobachtbarkeit von SoCs. Ein wichtiges Kriterium ist hier die Vollständigkeit der Beobachtung und umfasst die Aktivitäten der CPU (ausgeführte Instruktionen, gelesene und geschriebene Daten, Verhalten des Caches, Ausführungszeiten), des Bussystems und von Umgebungsbedingungen. Weitere Kriterien sind die Echtzeitfähigkeit und die Kontinuität der Beobachtung sowie die gleichzeitige Durchführung verschiedener Beobachtungsaufgaben. Dabei soll es zu einer möglichst geringen Beeinflussung des SoCs kommen. Weitere wichtige Aspekt sind die Kosten der Lösung, die Universalität, die Skalierbarkeit sowie die Latenz der Verfügbarkeit der Beobachtungsergebnisse. Für viele Anwendungen, besonders in sicherheitskritischen Bereichen, muss zudem nachgewiesen werden, dass das Beobachtungsverfahren kein Fehlverhalten des SoCs bewirkt bzw. ein solches maskiert. Eine besondere Herausforderung stellen Multiprozessor-SoCs (MPSoCs) dar, da hier die Kommunikation zwischen den einzelnen CPUs im Inneren des SoC stattfindet und entsprechend schwierig für einen externen Bobachter sichtbar zu machen ist.

Der Stand der Technik zur Beobachtung von SoCs wird im Wesentlichen durch zwei Verfahren dargestellt.

Bei der *Software-Instrumentierung* wird zum funktionalen Programmcode zusätzlicher Code hinzugefügt, welcher zur Beobachtung des Programms dient. Diese Methode ist einfach und universell anwendbar, erfüllt aber die genannten Kriterien nur sehr eingeschränkt. Nachteilig ist hier der Ressourcenverbrauch im Falle des Verbleibs der Instrumentierung im fertigen Produkt. Wird die Instrumentierung nur temporär dem Code hinzugefügt, muss sichergestellt werden, dass das Beobachtungsergebnis auch für den finalen Code anwendbar ist – was besonders bei ressourcen-abhängigen Integrationstests nur schwierig erfüllbar ist.

Eine alternative Lösung stellt eine spezielle Hardware-Unterstützung in SoCs (*„embedded Trace“*) dar. Hier werden im SoC Zustandsinformationen (z.B. Taskwechsel, ausgeführte Instruktionen, Datentransfers) gesammelt und mittels Trace-Nachrichten an den Beobachter übermittelt. Dabei stellt die Bandbreite, die zur Ausgabe der Trace-Nachrichten vom SoC verfügbar ist, ein entscheidendes Nadelöhr dar - im SoC sind viel mehr den Beobachter interessierende Informationen verfügbar als nach außen transferiert werden können.

Damit haben beide dem gegenwärtige Stand der Technik entsprechende Beobachtungsverfahren eine Reihe von Einschränkungen, die sich besonders bei der Vollständigkeit der Beobachtung, der Flexibilität, der Kontinuität und der Unterstützung von MPSoCs zeigen.

In dieser Arbeit wird nun ein neuer Ansatz vorgestellt, welcher gegenüber dem Stand der Technik in einigen Bereichen deutliche Verbesserungen bietet. Dabei werden die Trace-Daten nicht vom zu beobachtenden SoC direkt, sondern aus einer parallel mitlaufenden Emulation gewonnen. Die Bandbreite der für die Synchronisation der Emulation erforderlichen Daten ist in vielen Fällen deutlich geringer als bei der Ausgabe von umfassenden Trace-Nachrichten mittels „embedded Trace“-Lösungen. Gleichzeitig ist eine vollständige, äußerst detaillierte Beobachtung der Vorgänge innerhalb des SoC möglich. Das neue Beobachtungsverfahren wurde mittels verschiedener FPGA-basierter Implementierungen evaluiert, hier konnte auch die Anwendbarkeit für MPSoCs gezeigt werden.

Inhaltsverzeichnis

Glossar

Cache	Schneller Puffer-Speicher
Cache-Hit	Cachetreffer – die angefragten Daten sind im Cache vorrätig
Cache-Kohärenz	Sicherstellung der Konsistenz mehrerer Caches in MPSoCs
Cache-Miss	Verfehlen des Caches - die angefragten Daten sind im Cache nicht vorrätig
Determinsmus	Vorherbestimmbarkeit
Double Data Rate	Übertragungsverfahren, bei dem Daten mit der aufsteigenden und der abfallenden Flanke des Taktsignals übertragen werden
Heap	Datenstruktur , zur Laufzeit des Programms zugeteilter Speicher
Mandelbug	Chaotisch und nichtdeterministisch erscheinendes Fehlverhalten
Mikrocontroller	Recheneinheiten mit Speicher (RAM, Flash) sowie umfassenden Peripherieeinheiten (z.B. Zähler, Zeitgeber, serielle und parallele Ein- und Ausgabekanäle, Interrupt-Eingänge, Analog-Digital- und Digital-Analog-Wandler, Kommunikations- und Überwachungseinheiten)
Mikroprozessor	Recheneinheiten mit einer nicht vorhandenen oder untergeordneten Ausstattung an Peripherieeinheiten.
omniscient	allwissend
Scratchpad	Schneller interner Speicher
Stack	Stapelspeicher
System-on-Chip	Komplettes Rechensystemen auf einem Chip, zu dessen Betrieb nur noch wenige externe Bauteile erforderlich sind
Trace	Sequenz von internen Zuständen in einer Recheneinheit
Zyklomatische Komplexität	Softwaremetrik, die angibt auf wie vielen unterschiedlichen Pfaden ein Software-Modul durchlaufen werden kann

Abkürzungsverzeichnis

ADPA	Advanced QorIQ Platform Debug Architecture
AMBA	Advanced Microcontroller Bus Architecture
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
ASIL	Automotive Safety Integrity Level
ATB	Advanced Trace Bus
CPU	Central Processing Unit
CTM	Cross Trigger Matrix
DAP	Debug Access Port
DAL	Design Assurance Level
DDR	Double Data Rate
DSP	Digitaler Signalprozessor
ED	Emulation Device
ETB	Embedded Trace Buffer
ETM	Embedded Trace Macrocell
FMEA	Fehlermöglichkeits- und Einfluss-Analyse
FPGA	Field Programmable Gate Array
Gbps	Gigabit pro Sekunde
hidICE	Hidden ICE
HSSTP	High Speed Serial Trace Port
HW	Hardware
ICE	In-Circuit Emulator
ID	Identifier
I/O	Input/Output
IP	Intellectual Property
ITM	Instrumentation Trace Macrocell
JPL	Jet Propulsion Laboratory – Betreiber von Raumsonden-Projekten für die NASA
JTAG	Joint Test Action Group, Synonym für den IEEE-Standard 1149.1
LUT	Lookup-Tabelle
Mbps	Megabit pro Sekunde
MCDS	Multi-Core Debug Solution
MDO	Message Data Out
MIPS	Million Instructions Per Second
MMU	Memory Management Unit
MPSoC	Multiprocessor System-on-Chip
MSEO	Message Start/End Out
NASA	National Aeronautics and Space Administration - Nationale Luft- und Raumfahrtbehörde der USA

PCIe	Peripheral Component Interconnect Express
PD	Production Device
PTM	Program Flow Trace Macrocell
QDR	Quadruple Data Rate
RAM	Random-Access Memory
RX	Receive, Empfänger von Nachrichten
SoC	System-on-Chip
SPSoC	Singleprocessor System-on-Chip
STM	System Trace Macrocell
SW	Software
SWO	Serial Wire Output
TCL	Tool Confidence Level
TD	Tool Error Detection Level
TI	Tool Impact Level
TLB	Translation Lookaside Buffer
TQL	Tool Qualification Level
TPIU	Trace Port Interface Unit
TX	Transmit, Sender von Nachrichten
WCET	Worst-case execution time
UART	Universal asynchronous receiver/transmitter
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1 Motivation

Hard- und Softwarekomponenten zur Steuerung, Regelung und Überwachung von Systemen, welche auch als eingebettete Systeme bezeichnet werden, sind mittlerweile ein unverzichtbarer Bestandteil vieler Produkte.

Getrieben wird diese Entwicklung u.a. durch den Innovations- und Preisdruck im Automobilbereich sowie im Bereich mobiler Kommunikationslösungen. Sie ist gekennzeichnet durch die stetige Zunahme des Funktionsumfangs, welcher durch fortlaufende Zunahme der Komplexität der Hard- und Softwarekomponenten erzielt wird. Die erzielten Fortschritte führen zu einer Nutzung bzw. Anpassung der zugrunde liegenden Technologien auch für andere Anwendungsgebiete, wie z.B. für industrielle Steuerungen, die Medizintechnik, die Luftfahrt sowie für die Konsum- und Haushaltselektronik.

Eingebettete Systeme ermöglichen eine große Flexibilität in der Produktentwicklung. Die Implementierung vieler technischer Merkmale kann mittels Software erfolgen. Ursprünglich mechanisch implementierte Funktionen in Systemen können durch Elektronik bzw. Elektromechanik ersetzt werden, bisher manuell gesteuerte Systeme können automatisiert werden. Zudem ist es möglich, den in vielen Fällen kaufentscheidenden „ersten Eindruck“ eines Produkts mittels Softwarevarianten zu definieren. Damit stellen eingebettete Systeme ein äußerst wichtiges Differenzierungsmerkmal von Produkten dar und sind eine Grundlage der Wettbewerbsfähigkeit von Unternehmen.

Zentraler Bestandteil eingebetteter Systeme sind programmierbare elektronische Bausteine, welches sich je nach Peripherieausstattung in Mikroprozessoren, Mikrocontroller und *System-on-Chip* (SoCs) unterteilen lassen.

Als Mikroprozessor bezeichnet man Recheneinheiten mit einer nicht vorhandenen oder untergeordneten Ausstattung an Peripherieeinheiten. Befinden sich auf dem Chip darüber hinaus auch verschiedene Speicher (RAM, Flash) sowie umfassende Peripherieeinheiten (z.B. Zähler, Zeitgeber, serielle und parallele Ein- und Ausgabekanäle, Interrupt-Eingänge, Analog-Digital- und Digital-Analog-Wandler, Kommunikations- und Überwachungseinheiten), so spricht man von Mikrocontrollern. Das Ergebnis der konsequenten Weiterentwicklung zu kompletten Rechensystemen auf einem Chip, zu deren Betrieb nur noch wenige externe Bauteile erforderlich sind, wird dann als SoC bezeichnet.

Der Übergang zwischen den dargestellten Kategorien ist oftmals fließend. Wenn im Folgenden von SoCs gesprochen wird, soll dies auch Mikrocontroller und Mikroprozessoren umfassen.

Die Leistungsfähigkeit von SoCs wird nicht allein durch Messgrößen wie die Ausführungszeit (Zeit für die Bearbeitung einer Aufgabe) oder den Durchsatz (Anzahl ausgeführter Aufgaben pro Zeiteinheit), sondern auch durch anwendungsspezifische Kennwerte bewertet. Diese sind z.B.

- Anzahl von Prozessor-Kernen
- Typ und Größe des Speichers
- Umfang und Leistungsfähigkeit der Peripherieeinheiten
- Möglichkeiten der Ereignisbehandlung (Steuerung der Prioritäten, programmierbare Reaktionen)
- Energieverbrauch
- Baugröße
- Anzahl notwendiger peripherer Bauteile
- Determinismus interner Abläufe
- Beobachtbarkeit

Die Software für eingebettete Systeme unterscheidet sich von üblicher, auf PCs laufender Software in einer Reihe von Punkten:

- Echtzeitanforderungen
- Funktionale Sicherheit
- Eingeschränkte Beobachtbarkeit
- Eingeschränkte Wartbarkeit
- Entwicklung auf einem Host-System

Ein wichtiger Aspekt bei der Entwicklung eingebetteter Systeme ist die Verfügbarkeit geeigneter Werkzeuge zum Testen, zur Fehlersuche (Debuggen) und zur Optimierung von Applikationen. Mit Testen, Fehlersuche und Optimieren verbringt ein Entwickler in der Regel einen großen Teil seiner Arbeitszeit.

In Umfragen wurde gezeigt, dass der Anteil dieser Tätigkeiten bei ca. 50% ([1],[2],[3],[4]), teilweise sogar bei 80% [5] des Gesamtentwicklungsaufwandes eines Software-Projektes liegt.

Dies bedeutet, dass jede Verbesserung von Test- und Debug-Prozessen einen wesentlichen Beitrag zur Verkürzung der Entwicklungszeit, zur Verringerung der Entwicklungskosten sowie zur Absicherung des Projekterfolgs leisten kann.

US National Institute of Standards and Technology [5]: *Software developers already spend approximately 80 percent of development costs on identifying and correcting defects.*

Jack Ganssle [1]: *After visiting hundreds of companies and chatting with thousands of developers, I'm left with a qualitative sense that code debug consumes about half the total engineering time for most products.*

Britton et al. [2]: *Developers spend 50% of their time [for] debugging*

Annual worldwide software development costs (2012):	*1250 Mrd. $*
davon:	
Productive (designing code (20%) and writing code (30%))	*625 Mrd. $*
Debugging (fixing bugs (25%) and making code work (25%))	***625 Mrd. $***

Laymann et al. [3]: *The time spent debugging varies depending on the phase in the development cycle. Some subjects described a phase in the development cycle dedicated mostly to debugging where they spent 90-100% of their time debugging. In other phases, subjects reported spending 4% to 50% of their time debugging.*

Eine wichtige Grundlage für das Testen, Debuggen und Optimieren der in eingebetteten Systemen laufenden Software ist die Beobachtbarkeit von Vorgängen innerhalb des Systems, besonders aber innerhalb des verwendeten SoCs.

Diese Beobachtbarkeit stellt eine zentrale Voraussetzung für die Gewährleistung einer hohen Softwarequalität und Zuverlässigkeit des eingebetteten Systems dar. Dem Wunsch nach umfassender Beobachtbarkeit steht eine sehr beschränkte Bandbreite zur Ausgabe interner Zustandsinformationen gegenüber.

Die vorliegende Arbeit beschäftigt sich mit der Frage, wie trotz einer beschränkten Bandbreite zur Ausgabe von Beobachtungsdaten möglichst umfassende interne Zustandsinformationen des zu untersuchenden eingebetteten Systems verfügbar gemacht werden können.

1.2 Ziele der Arbeit

Anhand von allgemeinen Anwendungen wie dem Debuggen und Testen sowie spezielleren Fragestellungen wie der Durchführung von Überdeckungsanalysen, der Ermittlung maximaler Ausführungszeiten, der Analyse von Parallelitätsfehlern, der Beobachtung von Variablen und des Ressourcenverbrauchs, der Analyse des Caches und der Optimierung des Speicherlayouts, der Optimierung des Energieverbrauchs sowie der Beobachtung von Umgebungsbedingungen und der Erkennung von Störungen werden die entsprechenden Anforderungen an die Beobachtbarkeit von SoCs erarbeitet.

Nachfolgend werden die erarbeiteten Anforderungen generalisiert und um weitere Kriterien wie Vollständigkeit, Echtzeitfähigkeit, Gleichzeitigkeit, Intrusivität, Flexibilität, Universalität, Qualifizierbarkeit und Kosten von Beobachtungslösungen ergänzt. Spezielles Augenmerk wird dabei auf die kommenden Herausforderungen gelegt, die die zunehmende Verbreitung von Multiprozessor-SoCs (MPSoCs) mit sich bringen.

Bei einer Analyse des Standes der Technik wird gezeigt, dass aktuell verfügbare Lösungen die ausgearbeiteten Kriterien nur eingeschränkt erfüllen. Dabei wird auf die am weitesten verbreiteten Technologien wie der softwaregestützten Code-Instrumentierung und besonders auf „embedded Trace“-Lösungen eingegangen. Eine wesentliche Limitation, die einer verbesserten Beobachtung im Wege steht, ist hier die Begrenzung der für die Ausgabe umfassender Trace-Nachrichten vom SoC verfügbaren Bandbreite.

Anhand der erkannten Defizite werden Kriterien an eine verbesserte Lösung zur Beobachtung von SoCs definiert. Mittels eines neuartigen Beobachtungsansatzes wird gezeigt, dass ein wesentlicher Teil der zuvor dargestellten Kriterien an die Beobachtbarkeit mit diesem neuen Ansatz in nahezu idealer Weise erfüllt werden. Die Besonderheit der Lösung besteht darin, dass die für die Beobachtung des SoC verfügbare Bandbreite nicht mehr wie bei traditionellen Lösungen zur Ausgabe von Trace-Nachrichten verwendet wird, sondern zur Synchronisation einer parallel laufenden Emulation. Damit ist es möglich, in vielen Fällen die verfügbare Bandbreite deutlich effizienter zu nutzen und bei geringerer oder vergleichbarer Bandbreite eine vollständige Beobachtung wesentlicher Bereiche des SoCs zu erreichen. Dabei wird der Zugriff auf bisher nicht verfügbare Informationen wie die zeitliche Änderung von CPU-Registern oder einen detaillierten Blick auf Busaktivitäten ermöglicht. Um Kosten für die für die Beobachtung des SoC notwendige Schnittstelle zu sparen, wird eine Methode vorgestellt, die aus Sicht der Anwendung auch während einer aktiven Beobachtung aller Pins der Schnittstelle verfügbar macht. Weiterhin wird ein Verfahren erläutert, mit dessen Unterstützung die Verlässlichkeit der gewonnenen Beobachtungsergebnisse nachgewiesen werden kann.

Das Funktionsprinzip des neuartigen Beobachtungsverfahrens wird für verschiedene Implementierungsvarianten erläutert, es werden Schwierigkeiten und Grenzen dargestellt, sowie mehrfache experimentelle FPGA-basierte Implementierungen vorgestellt.

1.3 Gliederung

Die vorliegende Arbeit ist in sechs Abschnitte unterteilt (Abbildung 1-1).

Abschnitt 1 beinhaltet eine allgemeine Einleitung und gibt einen Überblick über die Ziele und die Gliederung dieser Arbeit.

Im Abschnitt 2 werden zu Beginn die beim Testen, der Fehlersuche und beim Optimieren von eingebetteten Systemen verwendeten Begriffe und Zusammenhänge erläutert. Speziell wird auf die verschiedenen Klassifizierungen von Defekten eingegangen. Im Folgenden werden häufig vorkommende Fragestellungen und Anwendungsfälle diskutiert, für deren Lösung die Beobachtbarkeit eines SoCs relevant ist. Diese Anwendungsfälle werden immer wieder zu den initial vorgestellten Klassifizierungen möglicher Defekte in Beziehung gesetzt. Für jeden individuellen Anwendungsfall werden Kriterien definiert, die der Beobachter zur Lösung der mit dem Anwendungsfall verbundenen Fragestellung erfüllen muss. Besonderes Augenmerk wird dabei auf die speziellen Anforderungen von MPSoCs sowie auf die Klasse der Mandelbugs gelegt, da diese Defekte mit traditionellen Lösungen nur unvollständig beobachtbar sind.

In Abschnitt 3 wird der zuvor aufgestellte Kriterienkatalog verallgemeinert, es werden Qualitätsmerkmale für einen „idealen“ Beobachter von SoCs definiert, welche u.a. die Bereiche Vollständigkeit, Gleichzeitigkeit, Echtzeitfähigkeit, Kontinuität, Intrusivität, Serientauglichkeit und Kosten umfassen.

In Abschnitt 4 werden ausgewählte, den Stand der Technik repräsentierende Lösungen zur Beobachtung von SoCs vorgestellt und anhand der Erfüllung der definierten Qualitätsmerkmale bewertet.

Abschnitt 5 stellt eine neuartige Lösung zur Beobachtung von SoCs vor, die sich vom Ansatz her deutlich von traditionellen Lösungen unterscheidet. Neben einer allgemeinen Funktionsbeschreibung werden Implementierungsvarianten, spezielle Implementierungsdetails, der Implementierungsaufwand sowie Grenzen des vorgestellten Ansatzes diskutiert. Anhand von experimentellen Implementierungen wird die Realisierbarkeit der vorgestellten Lösungen demonstriert.

Die Arbeit endet im Abschnitt 6 mit einem Vergleich der vorgestellten Beobachtungsverfahren anhand der zuvor ausgearbeiteten Kriterien. In einer Zusammenfassung wird das neuartige Beobachtungsverfahren kritisch diskutiert und ein Ausblick auf zukünftige Entwicklungen gegeben.

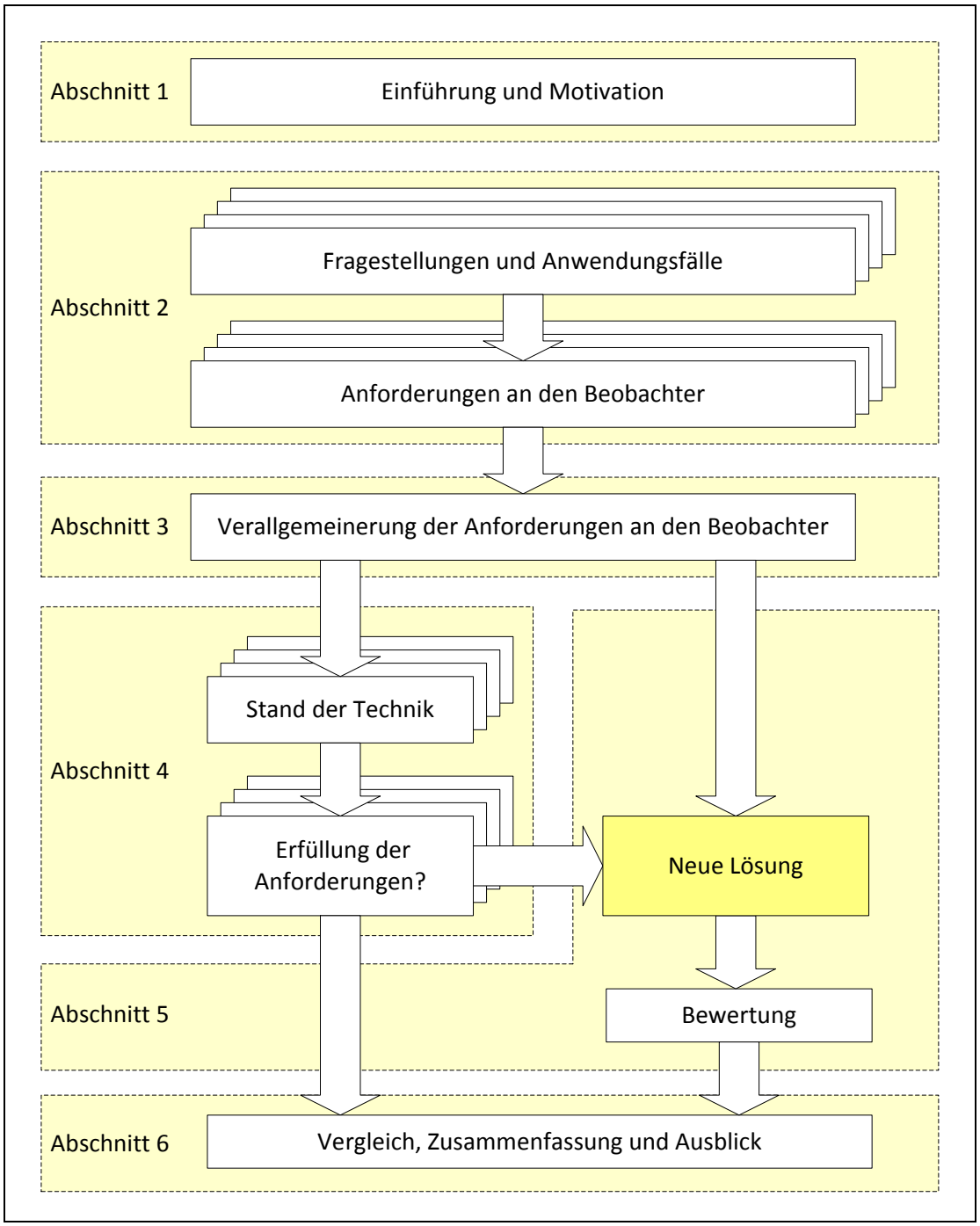


Abbildung 1-1: Überblick über die Gliederung der Arbeit

2. Fragestellungen und Anwendungsfälle

Die Beobachtbarkeit von SoCs in einem eingebetteten System ist eine wichtige Voraussetzung für den Nachweis der Erfüllung vorgegebener Anforderungen sowie für die Suche nach den Ursachen von Fehlverhalten.

Im Folgenden werden typische Fragestellungen und Anwendungsfälle diskutiert.

Im Abschnitt 2.1 werden Ursachen sowie Verfahren zur Aufdeckung und Beseitigung von Fehlverhalten in eingebetteten Systemen erläutert und daraus allgemeine Anforderungen an die Qualität der Beobachtbarkeit von SoCs abgeleitet.

Die nachfolgenden Abschnitte behandeln einzelne Aspekte o.g. Verfahren wie

- Überdeckungsanalysen (Abschnitt 2.2)
- die Bestimmung maximaler Ausführungszeiten (Abschnitt 2.3)
- die Erkennung von Parallelitätsfehlern (Abschnitt 2.4)
- die Beobachtbarkeit von Variablen (Abschnitt 2.5)
- die Bestimmung des Ressourcen-Verbrauchs (Abschnitt 2.6)
- die Analyse des Caches sowie die Optimierung des Speicherlayouts (Abschnitt 2.7)
- die Optimierung des Energieverbrauchs (Abschnitt 2.8)
- die Beobachtbarkeit angeschlossener Hardware sowie von Umgebungsparametern (Abschnitt 2.9)

und leiten die entsprechenden Anforderungen an die Beobachtbarkeit ab.

2.1 Fehlverhalten, Testen und Debuggen

In diesem Abschnitt werden Ursachen von möglichem Fehlverhalten eingebetteter Systeme diskutiert und Verfahren vorgestellt, um diese zu erkennen und zu beseitigen. Daraus werden Anforderungen an die Beobachtbarkeit der verwendeten SoCs abgeleitet.

In diesem Zusammenhang ist der exakte Gebrauch der für dieses Gebiet relevanten Begriffe wie *defect, failure, fault, error* und *bug* wichtig, da in dem genannten Umfeld Bezeichnungen sowie deren deutsche Entsprechungen oftmals unpräzise verwendet werden.

2.1.1 Untersuchung von Fehlverhalten

Der im Zusammenhang mit Fehlverhalten sicherlich am häufigsten verwendete Begriff ist der „Bug“. Dieses Wort wurde bereits im 19. Jahrhundert zur Beschreibung unerwarteter Systemfehler verwendet ([6],[7]) und sollte die Idee illustrieren, dass das Rauschen in einer Telefonleitung vom Knabbern kleiner Wanzen („Bugs“) an der Leitung herrührt.

> Korrespondenz zwischen Thomas Edison und Tivadar Puskás, 1878, aus [8]: *It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise — this thing gives out and [it is] then that “Bugs” — as such little faults and difficulties are called — show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.*

Beachtenswert ist, dass der vielzitierte erste „offizielle“ Computer-„Bug“, ein in einem Relais eingeklemmter Käfer, nichts mit fehlerhaftem Programmcode zu tun hatte, sondern lediglich eine Störung darstellte. In Abbildung 2-1 ist der entsprechende Logbucheintrag von Grace Murray Hopper vom 9. September 1947 (ursprünglich falsch auf den 9. September 1945 datiert) wiedergegeben, welcher eine in einem Relais eingeklemmte Motte als Ursache eines Fehlverhaltens des Harvard Mark II Aiken Relay Calculators beschreibt [9].

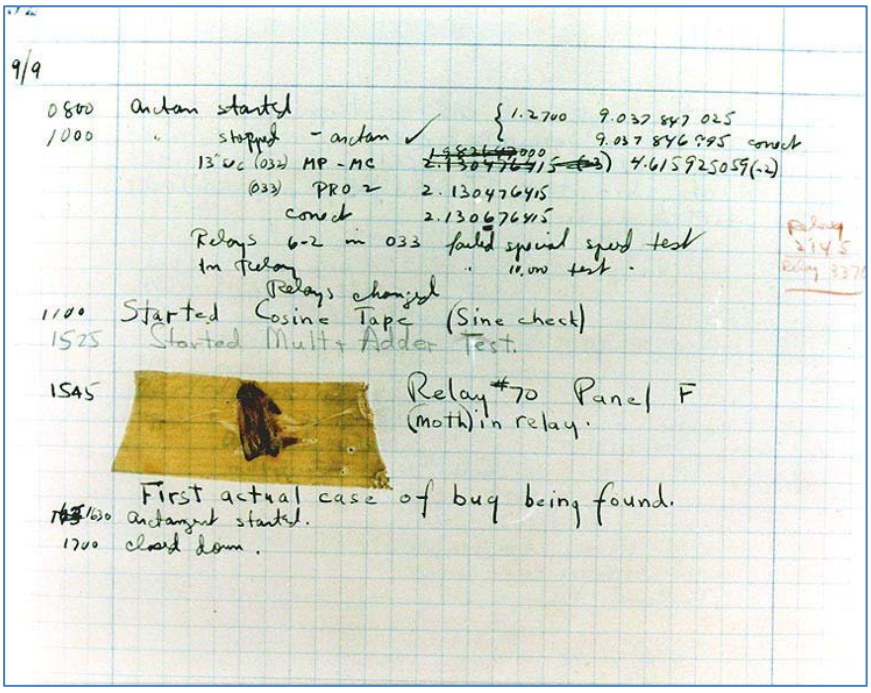


Abbildung 2-1: Der erste Computer-„Bug“ - eine in einem Relais eingeklemmte Motte

Für eine systematische Beschreibung der Mechanismen, die letztendlich zu einem beobachtbaren Fehlverhalten führen, werden die in Tabelle 2-1 (nach [6], Abschnitt 1.6) dargestellten Begriffe verwendet.

Beschreibung	Umgangssprachliche Beschreibung	Klarerer Begriff	Alternativer Begriff
Fehlerhafte Aktion einer Person	Der Programmierer hat einen *Bug* verursacht.	**Irrtum**	Fehlhandlung, Programmierfehler, mistake, error
Nicht korrekter Programmcode	Diese Code-Zeile ist *buggy*	**Defekt**	Innerer Fehler, Fehlerursache, fault
Nicht korrekter Zustand eines Programms	Dieser Null-Pointer ist ein *Bug*.	**Infektion**	
Nicht korrekte Ausführung eines Programms	Dieses Programm ist gecrasht. Dies ist ein *Bug*.	**Fehlverhalten**	Äußerer Fehler, Fehlerwirkung, Ausfall, *failure*

Tabelle 2-1: Beispiele für die Verwendung des Begriffs "Bug"

Ergänzend sind in der Tabelle 2-2 die in der ISO/IEC/IEEE 24765-2010 (*Systems and software engineering - Vocabulary*) [10] aufgeführten Definitionen relevanter Begriffe aufgelistet. Auch hier fällt auf, dass sich die Bedeutung von Begriffen gegenseitig entspricht (z.B. „mistake“ und eine Bedeutung von „error“: *a human action that produces an incorrect result*). Interessant ist auch, dass in der ISO/IEC/IEEE 24765-2010 keine Definition des Begriffs „bug“ enthalten ist, sondern dieser nur als Synonym für eine der verschiedenen Bedeutungen von „fault“ erwähnt ist.

Es gibt eine Reihe von Veröffentlichungen, welche die Untersuchung von Fehlverhalten systematisch darstellen [6][11]. Die folgenden Definitionen lehnen sich an [6], Abschnitt 1.2 an.

Grundlage eines Entwicklungsproduktes ist eine Definition der **Anforderungen**, aus denen eine **Systemspezifikation** erstellt wird, welche das geforderte Systemverhalten beschreibt. Eine Spezifikation muss präzise, einheitlich, vollständig und verbindlich sein. Anhand der Systemspezifikation wird ein **Entwurf** erstellt, auf dessen Basis im Rahmen der **Realisierung** ein Programm erstellt wird. Das korrekte Verhalten des Programms wird durch **Test**s überprüft.

Die **Qualität** eines Entwicklungsproduktes ist das Maß, welches den Grad der Übereinstimmung mit den Anforderungen, der Systemspezifikation und dem Entwurf angibt. Die Nichterfüllung der Anforderungen, der Systemspezifikation oder des Entwurfs wird als **Fehler** bezeichnet.

Be-griff	Definition
mis-take	*a human action that produces an incorrect result* *NOTE The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).*
error	*a human action that produces an incorrect result, such as software containing a fault.* *an incorrect step, process, or data definition.* *an incorrect result.* *the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition*
fault	*a manifestation of an error in software.* *an incorrect step, process, or data definition in a computer program.* *a defect in a hardware device or component.* *NOTE: A fault, if encountered, may cause a failure.*
de-fect	*a problem which, if not corrected, could cause an application to either fail or to produce incorrect results. ISO/IEC 20926:2003, Software engineering - IFPUG 4.1 Unadjusted functional size measurement method - Counting practices manual* *an imperfection or deficiency in a project component where that component* *does not meet its requirements or specifications and needs to be either repaired or replaced. A Guide to the Project Management Body of Knowledge (PMBOK® Guide) - Fourth Edition.* *a generic term that can refer to either a fault (cause) or a failure (effect). IEEE Std 982.1-2005 IEEE Standard Dictionary of Measures of the Software Aspects of Dependability.2.1* *EXAMPLE (1) omissions and imperfections found during early life cycle phases and (2) faults contained in software sufficiently mature for test or operation*
fail-ure	*termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. ISO/IEC 25000:2005, Software Engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE.4.20.* *an event in which a system or system component does not perform a required function within specified limits* *NOTE A failure may be produced when a fault is encountered.*

Tabelle 2-2: Definition relevanter Begriffe aus der ISO/IEC/IEEE 24765-2010

Die Abweichung eines berechneten, beobachteten oder gemessenen Wertes oder einer Bedingung von dem wirklichen, gewünschten oder theoretisch korrektem Wert oder der entsprechenden Bedingung bezeichnet man als **Fehlverhalten** (*failure*), welches verschiedene Ursachen haben kann (Abbildung 2-2).

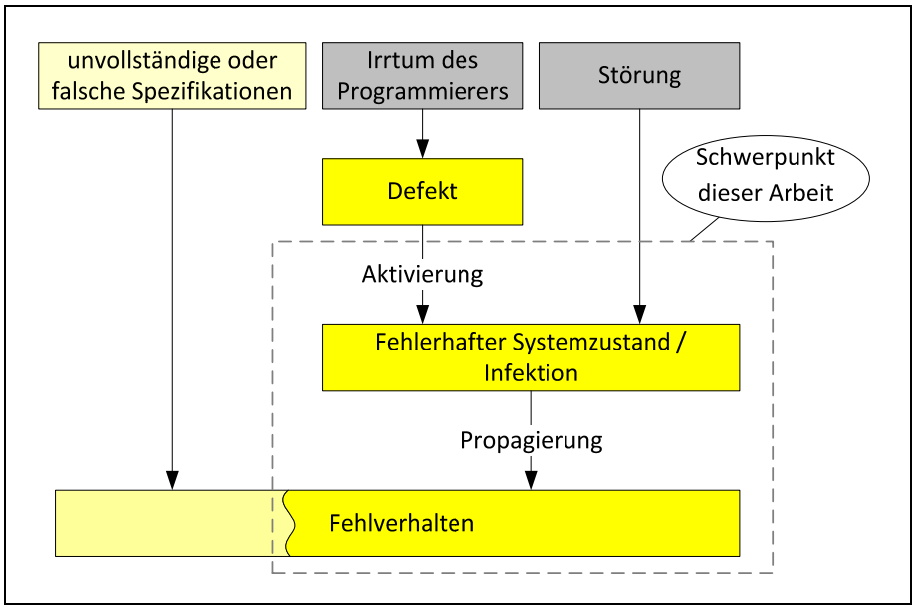


Abbildung 2-2: Ursachen von Fehlverhalten

Beispiele für Fehlverhalten sind:

- Ausgabe falscher Werte
- Program-Crash
- Abweichung vom beabsichtigten Zeitverhalten (z.B. Reaktionszeit auf ein Ereignis)
- Abweichungen vom beabsichtigtem Ressourcenverbrauch (z.B. Energieverbrauch)

Unvollständige oder falsche Spezifikationen (bei korrekter Implementierung der dort definierten Funktionalität) können eine Ursache von Fehlverhalten sein. Beispiele dafür sind:

- Eine initiale Spezifikation sah keine langfristige Verwendung des Entwicklungsproduktes vor. Klassisches Beispiel ist das Jahr-2000-Problem.
- In einem modularen Programm oder Komponenten kann ein ungewolltes Verhalten auftreten, das aus einer mangelhaften Schnittstellenbeschreibung oder unvorhersehbarem Zusammenspiel von Komponenten resultiert. Die einzelnen Programmteile oder Komponenten sind entsprechend der Spezifikation aber korrekt implementiert.

Die folgenden Ausführungen gehen jedoch von einer fehlerfreien Spezifikation aus.

Störungen (*disturbances*) können zu Infektionen und damit einhergehendem Fehlverhalten führen und treten u.a. aufgrund folgender Ursachen auf [12]:

- Umgebungseinflüsse und mechanische Einflüsse (lose Steckverbindungen, Temperatur, Vibration, elektromagnetische Interferenz)
- elektrische Einflüsse (z.B. Übersprechen, zu flache Signalflanken, Reflexionen von Signalen, Störungen der Stromversorgung)
- Strahlungseffekte terrestrischen und extraterrestrischen Ursprungs (durch als Folge radioaktiven Zerfalls entstandene α- und β-Strahlung, hochenergetische Neutronen, elektromagnetische Strahlung). Strahlungseffekte können zu einer Änderung des logischen Zustands von Speicherzellen (*single event upset*) führen. Bei sehr hohen Taktfrequenzen kann die Funktion einer Speicherzelle auch über mehrere Schreibzyklen gestört sein. Da SoCs mit immer kleineren Strukturgrößen, sich verringernder Versorgungsspannung und steigen-

der Taktfrequenz zunehmend empfindlicher für Störungen durch Strahlungseffekte werden, ist perspektivisch – nicht nur in der Luft- und Raumfahrt - mit einer Zunahme derartiger Störungen zu rechnen.

Besonders bei sicherheitskritischen Systemen muss das Design so ausgelegt sein, dass durch Störungen hervorgerufene Infektionen selbständig korrigiert werden können [13].

Eine weitere Ursache von Fehlverhalten kann die fehlende oder fehlerhafte Umsetzung einer Programmspezifikation durch den Programmierer sein. Diese wird als **Defekt** bezeichnet und beruht auf einem **Irrtum** des Programmierers. In [14] sind eine Reihe möglicher Faktoren angegeben, die zu derartigen Irrtümern führen:

- Kommunikationsprobleme bei der Anforderungsspezifikation: Unklarheit oder unterschiedliche Interpretation der Spezifikation
- Psychologische oder kulturelle Gründe: In manchen Kulturkreisen herrscht eine gewisse Scheu, Rückfragen bei Unklarheiten zu stellen. Dies kann besonders zu Tage treten, wenn Entwicklungsprojekte durch Offshoring innerhalb verschiedener geographischer Regionen bearbeitet werden.
- Änderung der Anforderungen während der Projektlaufzeit
- Komplexität der Software: Mit zunehmender Komplexität wird es immer schwieriger, Abhängigkeiten (komplizierte Algorithmen, unterschiedliche Zielplattformen) vollständig zu verstehen und zu testen
- Zeitdruck
- schlecht dokumentierter Code
- Programmierfehler

Die Ausführung eines fehlerhaften Code-Abschnitts, welche zu einer Abweichung des tatsächlichen Systemzustandes vom eigentlich gewollten Systemzustand führt, wird als **Aktivierung** eines Defekts bezeichnet.

Dieser **fehlerhafte Systemzustand** wird auch als **Infektion** bezeichnet.

Ein Defekt, der nicht aktiviert wurde, wird als ruhender (*dormant*) Defekt bezeichnet.

Eine stattgefundene Infektion hat oftmals keinen Einfluss auf die momentane Programmausführung, in vielen Fällen kann der fehlerhafte Systemzustand auch wieder überschrieben, maskiert oder im späteren Programmablauf korrigiert werden.

Erst wenn der weitere Programmablauf zu einem beobachtbaren **Fehlverhalten** führt, spricht man von einer **Propagierung** der Infektion.

In Abbildung 2-3 ist ein Szenario dargestellt (nach [6] (Fig.1.1)), bei dem es nach einer Aktivierung eines Defektes zu einem späteren Zeitpunkt zu einem beobachtbarem Fehlverhalten kommt. Es werden interne Zustände (i_x), extern sichtbare Zustände (e_x) sowie einzelne Programmschritte (z_x) dargestellt. Weiterhin werden die in Tabelle 2-3

erläuterten Klassifizierungen von Infektionstypen verwendet (aus [15], Tabelle 3.1).

Typ	Untertyp	Ursache	
1	1a	Defekt	fehlerhafte Anweisung(en)
	1b		fehlende Anweisung(en)
2	2a	Abhängigkeit von einem infiziertem Wert	direkt (Datenfluss)
	2b		indirekt (Kontrollfluss)

Tabelle 2-3: Klassifizierung von Infektionstypen

Bei einem ersten Verarbeitungsschritt (**z_1 -> z_2**) wird ein vorhandener Defekt aktiviert und führt zu einer Infektion – der interne Zustand i_3 zum Zeitpunkt z_2 entspricht nicht der Spezifikation, führt aber erst einmal zu keinem beobachtbarem Fehlverhalten.

Entsprechend der Klassifikation in Tabelle 2-3 liegt hier eine Infektion vom Typ 1 vor, weil sie unmittelbar durch einen Defekt im Programm (fehlerhafte oder fehlende Anweisung) verursacht wurde.

Im nächsten Verarbeitungsschritt (**z_2 -> z_3**) wird der fehlerhafte Zustand zwar verwendet, es erfolgt aber eine Maskierung, so dass der extern sichtbare Zustände (e_2 zum Zeitpunkt z_3) korrekt ist. Beispielsweise kann dies durch eine VerUNDung geschehen, wenn ein (korrekter) Operand den Werte „FALSE“ hat, so hat ein eventuell falscher zweiter Operand keine Auswirkung auf das Ergebnis.

In einem weiteren Verarbeitungsschritt (**z_3 -> z_4**) bewirkt die Verwendung des fehlerhaften Zustands i_3 einen weiteren nicht korrekten internen Zustand (i_2 zum Zeitpunkt z_4). Hier handelt es sich nun um eine Infektion vom Typ 2, welche durch einen bereits infizierten Wert verursacht wird.

Im nächsten Verarbeitungsschritt (**z_4 -> z_5**) wird der ursprünglich infizierte Zustand i_3 überschrieben.

Im letzten dargestellten Verarbeitungsschritt (**z_5 -> z_6**) bewirkt nun der fehlerhafte Zustand i_2 ein sichtbares Fehlverhalten (e_1 zum Zeitpunkt z_6). Setzt man an dieser Stelle einen Debugger ein, um auch interne Zustände inspizieren zu können, so lässt sich zwar der fehlerhafte Zustand i_2 entdecken, durch das Überschreiben der initialen Infektion (i_2 zum Zeitpunkt z_5) wird sich die Suche nach dem das beobachtete Fehlverhalten verursachenden Defekt als sehr schwierig gestalten.

In dem Szenario in Abbildung 2-3 wirkt ein infizierter Wert direkt auf einen anderen Wert aus (Typ 2a, z.B. arithmetische Operation mit Beteiligung eines zuvor infizierten Wertes), die Infektion kann sich aber auch auf den Kontrollfluss des Programms (Typ 2b, z.B. fehlerhafte Verzeigung aufgrund der Auswertung eines zuvor infizierten Wertes, in Abbildung 2-3 nicht dargestellt) auswirken.

Abbildung 2-4 zeigt das in Abbildung 2-3 dargestellte Szenario in einer komprimierten Form, auf die in dieser Arbeit noch wiederholt zurückgegriffen werden wird.

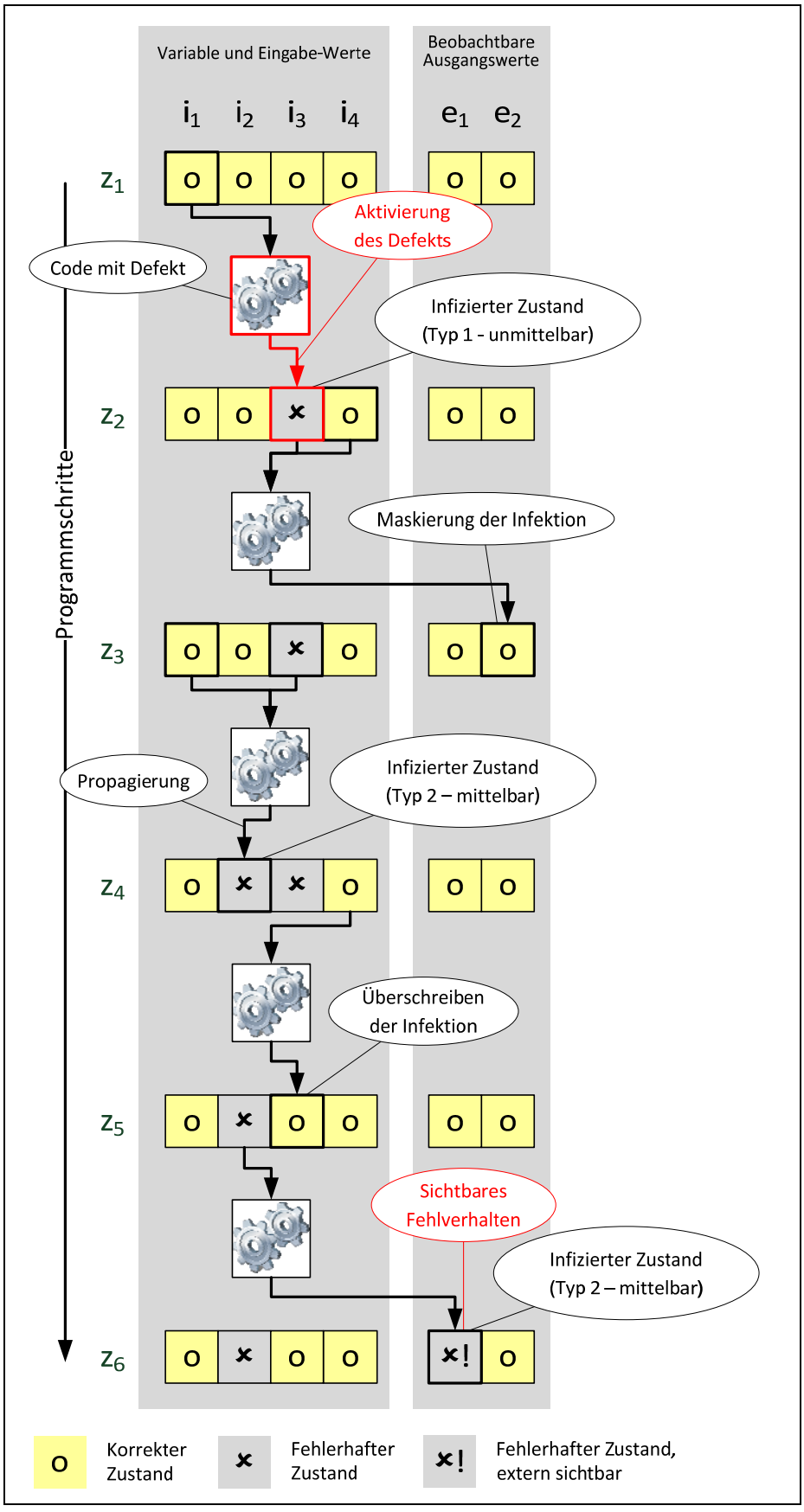


Abbildung 2-3: Verdeckung einer Infektion bei der Ausführung von defektem Programmcode

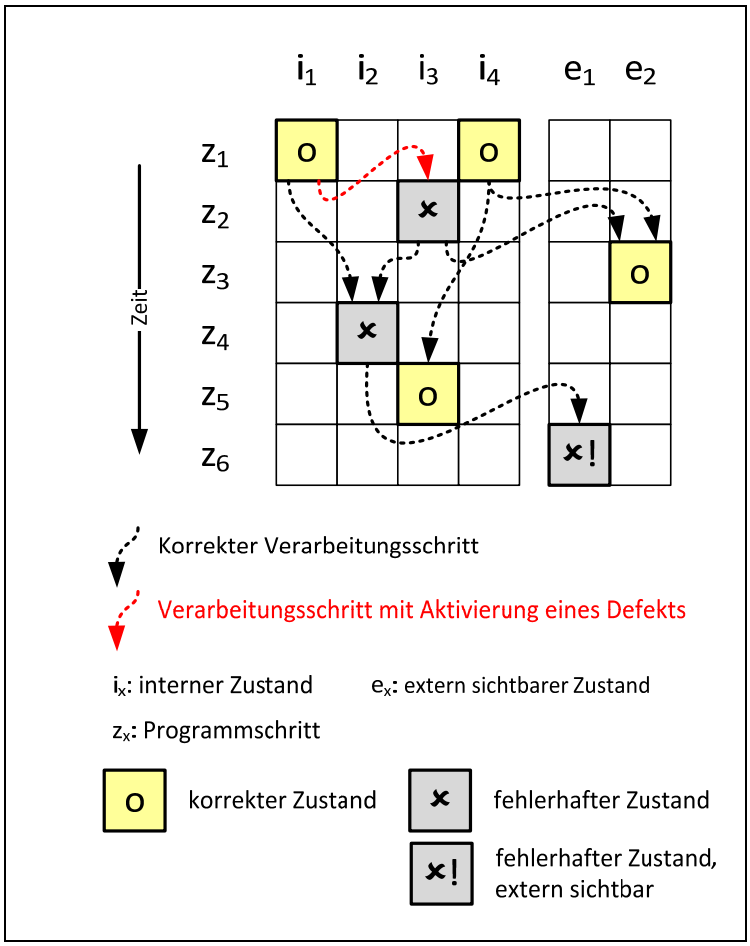


Abbildung 2-4: Komprimiert Darstellung des Szenarios aus Abbildung 2-3

Defekttypen

Je nach Projektgröße, Anwendungsgebiet, Sicherheitsanforderungen und der Erfahrung der Projektmitarbeiter ergeben sich deutliche Streuungen in der Anzahl von Defekten sowie in der Verteilung von Defekttypen.

Ein Beispiel für letzteres ist in [16] angegeben (Tabelle 2-4, Abbildung 2-5), es basiert auf einer Auswertung von 21 NASA-Missionen über einen Zeitraum von 10 Jahren.

Defekttyp	Relative Häufigkeit
Fehlerhafte Anforderungen	32,65%
Design-Fehler	5,60%
Fehlerhafte Kodierung	32,58%
Daten-Probleme	13,72%
Integrations-Fehler	2,27%
Andere	5,80%
Nicht angegeben	7,38%

Tabelle 2-4: Beispiel für die relative Häufigkeit von Defekttypen in einem Projekt

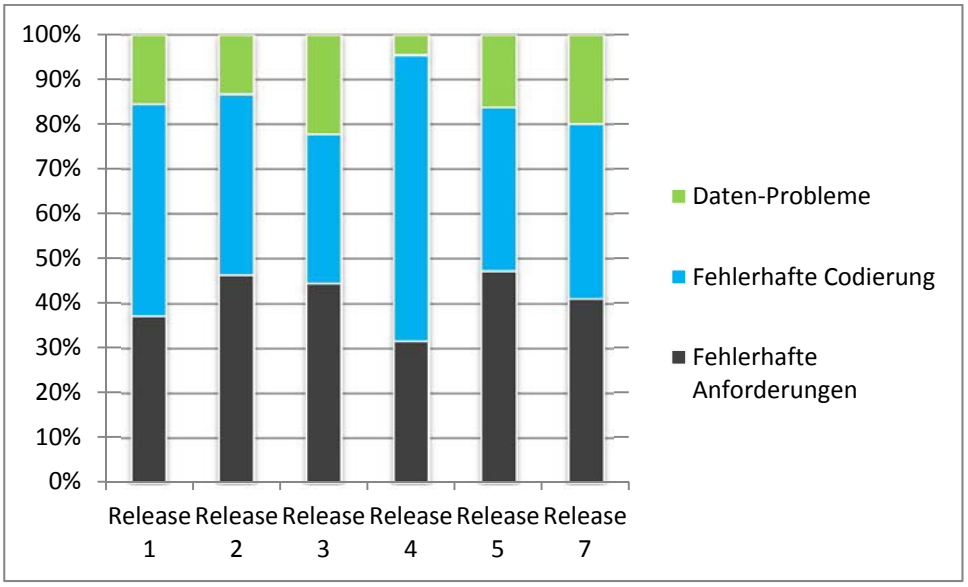


Abbildung 2-5: Verteilung der relativen Defekthäufigkeit über verschiedenen Releases

Es lässt sich erkennen, dass zum Auffinden von mehr als der Hälfte der gefundenen Defekte (Kodierung, Daten-Probleme, Integrations-Fehler) die Beobachtbarkeit von SoCs relevant ist.

Ursachen von Infektionen lassen sich entsprechend der Reproduzierbarkeit ihrer Aktivierung und des Determinismus des beobachteten Fehlverhaltens klassifizieren (Abbildung 2-6, nach [17]). Man unterscheidet Bohrbugs [18] und Mandelbugs [19], wobei letztere auch die Klasse der Heisenbugs und der „*Aging-related bugs*“ umfasst. Zusätzlich muss immer bedacht werden, dass auch physikalische Ursachen (Störungen) ein Grund für infizierte Zustände sein können.

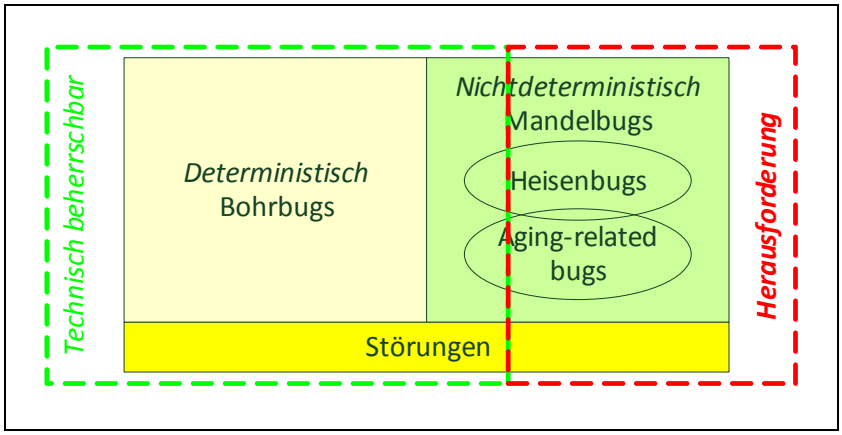


Abbildung 2-6: Klassifikation von aktivierten Defekten

Bohrbugs

Permanente Defekte, die sich deterministisch aktivieren lassen, werden als Bohrbugs (in Anlehnung an das Bohrsche Atommodell) bezeichnet [18]. Bohrbugs sind die am einfachsten zu entdeckenden Defekte, die sich größtenteils durch systematische Softwaretests sowie statische Analysen auffinden lassen. Sie werden auch als „harte“ Defekte bezeichnet.

Ein Beispiel für einen Bohrbug ist ein Defekt, welcher im Jahr 1996 zur Auslösung des Selbstzerstörungsmechanismus aufgrund einer nicht behandelten Ausnahmebedingung beim Jungfernflug der Ariane 5 [20] geführt und das Ariane-Programm um drei Jahre verzögert hat.

Die Software der Trägheitsnavigation wurde unverändert und ohne weitere Tests von dem Vorgängersystem (Ariane 4) übernommen. Aufgrund der vom Vorgängersystem abweichenden Flugbahn traten bei der Ariane 5 andere Werte als bei der Ariane 4 auf. Der Defekt in der Software war letztendlich eine fehlerhafte Konvertierung einer 64-Bit- Gleitkommazahl in einen nur 16 Bit breiten Integerwert ohne eine Implementierung eines Überlaufschutzes.

```
declare
  ...
  horizontal_veloc_sensor: float;
  ...
  horizontal_veloc_bias: integer;
  ...
begin
  declare
    pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    ...
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
    ...
  exception
    ...
    when others => use_irs1();
end;
end irs2;
```

Der Defekt wurde aufgrund der unglücklichen Kombination von fehlerhafter Spezifikation und unvollständigen Tests nicht vorab gefunden.

Mandelbugs

Als Mandelbug wird in Anspielung auf die Mandelbrot-Menge [19] ein chaotisch und nicht deterministisch erscheinendes Fehlverhalten bezeichnet, dessen rationale Ursache nicht immer gefunden werden kann.

Die Komplexität von Mandelbugs kann folgende Ursachen haben [21]:

- Zeitverlauf zwischen Aktivierung eines Defekts und dem beobachteten Fehlverhalten
- Einfluss indirekter Faktoren
    - Interaktionen von Software mit der system-internen Umgebung (Hardware, Betriebssystem, andere Anwendungen)
    - Einfluss des Zweitverhaltens von Eingaben und Operationen (relativ zueinander oder in Bezug auf die Systemzeit)
    - Einfluss der Reihenfolge von Operationen

Mandelbugs lassen sich mit Tests nur schwer erkennen und treten meist erst lange nach dem Release einer Software auf. Sie werden auch als „weiche“ (*soft*) Defekte bezeichnet. Die Häufigkeit von nach einem Software-Release gefundenen Mandelbugs variiert stark und wird in der Literatur mit Werten von 15% bis 80% angegeben [22].

Für den Entwickler und Tester ist es sehr schwierig, als Ursache eines Defektes zwischen einem Mandelbug und einer Störung zu differenzieren.

Aging-related bugs

Auch die „*aging-related bugs*“ [23] gehören zur Klasse der Mandelbugs. Es handelt sich hier um Defekte, die erst nach einer bestimmten Laufzeit des Programms aktiviert werden. Dies können beispielsweise sich fortsetzende Rundungsfehler oder Speicherlecks (*memory leaks*) sein.

Ein sehr oft zitiertes Beispiel eines „*aging-related bugs*“ ist die Fehlfunktion eines Patriot-Raketenabwehrsystem auf der Dhahran Air Base, Saudi-Arabien, welche im Februar 1991 auftrat [24].

Ursache des Defekts war die Multiplikation einer gemessenen Zeit (in einem 24 Bit breiten Integerwert abgelegt) mit dem Faktor „0,1“ in einer Repräsentation als Fließpunktzahl (mit einer 24 Bit breiten Mantisse) ohne Berücksichtigung eines dabei auftretenden Rundungsfehlers. Dieser Rundungsfehler führte nach mehr als 8 Stunden Betriebsdauer zu relevanten Ungenauigkeiten.

Bei dem Unglück wurde die Flugbahn einer anfliegenden Rakete nach einer Betriebszeit von über 100 Stunden falsch berechnet. Als Folge dieser fehlerhaften Berechnung wurden für die vermeintlich ihr Ziel verfehlende Rakete keine Abwehrmaßnahmen ergriffen. Die Rakete schlug in der Militärbasis ein, was 29 Menschen das Leben kostete.

Es könnten aber auch noch weitere Ursachen für das Versagen des Systems eine Rolle gespielt haben - eine Analyse von Videodaten stellt die Tauglichkeit des Systems für den vorgesehenen Einsatzzweck nach damaligem Entwicklungsstand grundsätzlich in Frage [25] .

Heisenbugs

Eine weitere Untergruppe der Mandelbugs sind Defekte, die erst durch die Suche nach Defekten aktiviert oder deaktiviert werden. In Anlehnung an die Heisenbergsche Unschärferelation werden diese Defekte als Heisenbugs [18] bezeichnet und beschreiben das Phänomen, dass eine vollständig rückwirkungsfreie Beobachtung eines Systems nicht möglich ist.

Werner Heisenberg : “*The more closely you look at one thing, the less closely can you see something else.*” [26].

In Bezug auf die Beobachtung von SoCs bedeutet dies, dass Defekte verdeckt werden (insbesondere im Bereich der Synchronisation und Kommunikation), dass Leistungsdaten verfälscht werden und nicht real vorkommende Abläufe vorgetäuscht werden.

Alternativ werden Heisenbugs auch als „*probe effect*“ bezeichnet.

Heisenbugs treten oft in Zusammenhang mit der Instrumentierung von Software auf (siehe Abschnitt 4.1). Wenn die Instrumentierung für die finale Version der Software aus Kostengründen entfernt wurde, ändert sich das Verhalten der Software deutlich. Durch Wegfall eines Teils des Codes sowie eventuell angewandte Compiler-Optimierungen ändert sich die Ausführungszeit eines Programms. Damit besteht eine zusätzliche Möglichkeit des Auftretens von Wettlaufsituationen, da in der instrumentierten Version die entsprechenden Routinen langsamer waren.

Ein Beispiel für die Verteilung von Bohr- und Mandelbugs ist in [27] angegeben. Nach dem Software-Release wurden in 18 Missionen der NASA sowie des JPL insgesamt 520 Software-Defekte gefunden, deren Verteilung nach Determinismus ihrer Aktivierung in Abbildung 2-7 dargestellt ist.

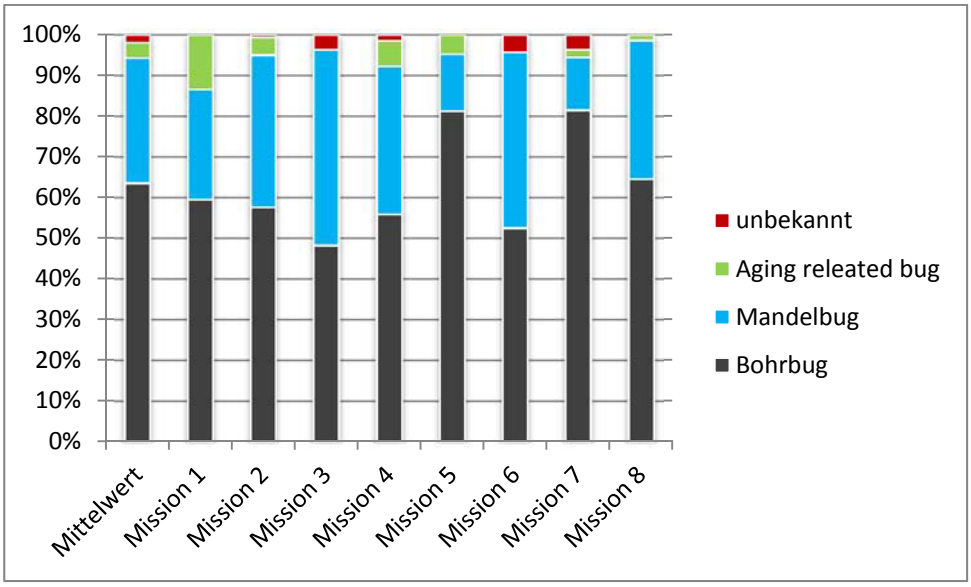


Abbildung 2-7: Verteilung von Bohr-, Mandel- und *aging-related bugs*

Es ist zu erkennen, dass trotz sicher umfangreicher Tests die Mehrzahl der nach dem Release gefundenen Fehler immer noch Bohrbugs gewesen sind. In der Veröffentlichung wird aber auch betont, dass der Übergang zwischen Bohr- und Mandelbugs fließend ist und die Reproduzierbarkeit des beobachteten Fehlverhaltens aufgrund der Vielzahl eingebauter Beobachtungsfunktionen deutlich erleichtert wurde. In einem weniger anspruchsvollen Umfeld wären demnach viele Bohrbugs als Mandelbugs charakterisiert worden.

2.1.2 Testen

Tests sind stichprobenartige Läufe eines Programms oder eines Teils des Programms mit dem Ziel, ein eventuelles Fehlverhalten zu beobachten und damit den zugrunde liegenden Defekt gezielt suchen zu können.

> Edsger W. Dijkstra [28]:
> "Testen kann die Anwesenheit von Fehlern aufzeigen, aber nie einen Nachweis von Fehlerfreiheit liefern."

Das Testen eines Programms erfolgt entsprechend dem V-Modell (Abbildung 2-8, nach [29]). Nach dem Test einzelner Komponenten werden auf den übergeordneten Ebenen Integrations-, System- und Abnahme-Tests durchgeführt. Fehler und Defekte summieren sich über die einzelnen Entwicklungsstufen. Je später sie erkannt und beseitigt werden, desto höher sind die dabei entstehenden Kosten.

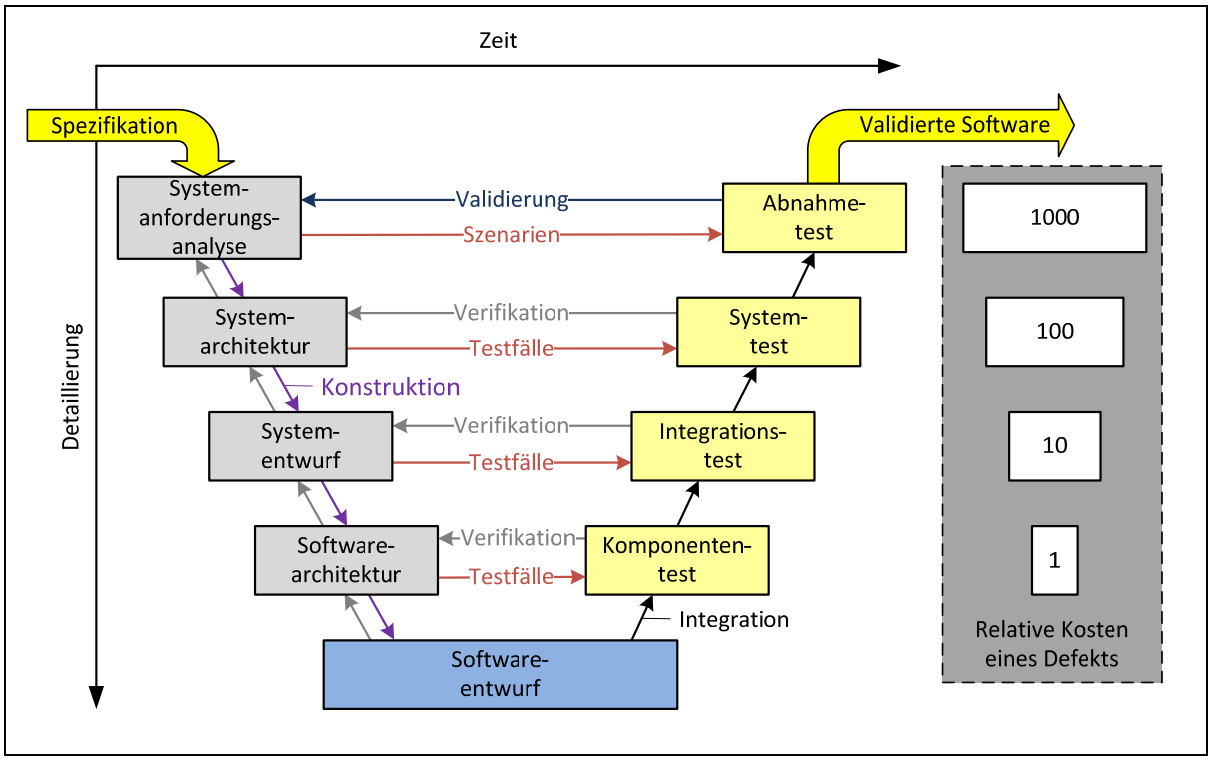


Abbildung 2-8: V-Modell sowie die relativen Kosten eines Software-Defektes

Grundsätzlich ist davon auszugehen, dass es ab einer gewissen Komplexität keine defektfreien Programme gibt. In [4] sind eine Reihe empirisch ermittelter Kennzahlen angegeben, die auf einer Analyse von über 13000 Software-Projekten beruhen. Dabei wird anhand verschiedener Kriterien wie Komplexität der Software-Strukturen, der zyklomatischen Komplexität, der Wiederverwendbarkeit von Programmcode oder von nachträglichen Spezifikationsänderungen zwischen niedriger, mittlerer und hoher Software-Qualität der Projekte unterschieden ([4], Tabelle 7-3).

Die erwartete Zahl von Defekten[1] (vor Reviews und Tests) ist beispielsweise bei einem Projektumfang von 1000 *Function Points*[2] (FP [30]) mit 4 Defekten pro FP bei hoher Software-Qualität angegeben, bei niedriger Software-Qualität sind es 6 Defekte pro FP. Dies bedeutet, dass davon ausgegangen werden muss, dass in einem Projekt dieser Komplexität durchschnittlich 4000 bzw. 6000 Defekte enthalten sind ([4], Tabelle 7-19).

Nach ihrem Release enthält die Software immer noch Defekte, bei einer Komplexität von 1000 FPs muss immer noch von ca. 4% nicht erkannter Defekte bei hoher Software-Qualität (dies entspricht der absoluten Zahl von 4% * 4000 = 160 Defekte) und von ca. 16% bei niedriger Software-Qualität (dies entspricht der absoluten Zahl von 16% * 6000 = 960 Defekte) ausgegangen werden.

Um auch noch nach dem Release einer Software diese noch verbliebenen Defekte aufspüren zu können, ist es sinnvoll, die Beobachtungsmöglichkeiten, die man während der einzelnen Teststufen zur Verfügung hatte, auch nach dem Release der Software verfügbar zu haben.

Im Folgenden werden die einzelnen Teststufen (nach [31]) kurz charakterisiert und daraus Anforderungen an die Beobachtbarkeit abgeleitet.

Komponententest

Beim Komponententest (auch Modul- oder Unit-Test) werden einzelne, isolierte Teile eines Programms an ihren Schnittstellen gegen die jeweilige Spezifikation getestet. Diese Isolation hat den Vorteil, dass im Falle eines Fehlverhaltens der zugrunde liegende Defekt innerhalb der getesteten Komponente zu suchen ist.

Da in den meisten Fällen die isolierte Komponente allein nicht lauffähig ist, müssen die fehlenden über- und untergeordneten Module simuliert werden. Nach einer Initialisierung rufen Testtreiber (z.B. cUnit [32]) die zu testende Komponente auf und kontrollieren die zurückgegebenen Ergebnisse. Von der Komponente selbst genutzte Funktionen, die zum Zeitpunkt des Tests nicht verfügbar sind, werden über *Dummies* bereitgestellt. Diese können als Stub-Code (Rückgabe vorkonfigurierter, zum Kontext passender Daten) oder als Mock-Objekte (Implementierung einer Schnittstelle mit Überprüfung des Verhaltens der zu testenden Komponente) implementiert sein [33](Abbildung 2-9).

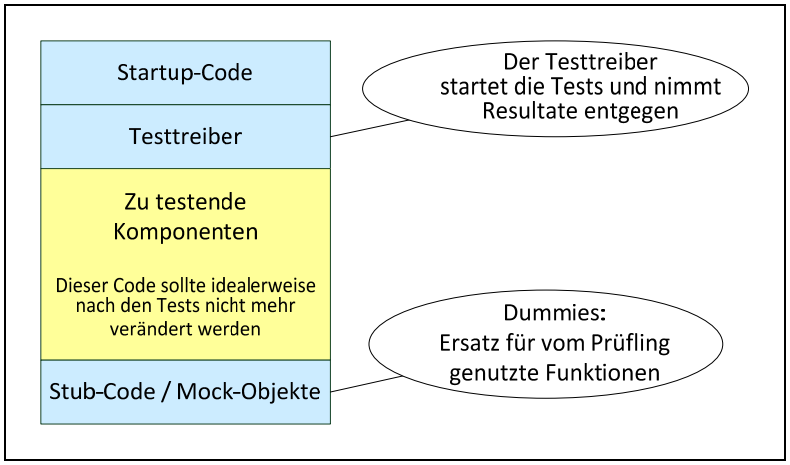


Abbildung 2-9: Speicherlayout für Komponententests

[1] Die Anzahl von Defekten umfasst hier Fehler in der Spezifikation, im Design, bei der Codierung, der Dokumentation und fehlerhafte Korrekturen)

[2] Ein Maß für den Umfang von Source-Code, ein FP entspricht – je nach Programmierstil - durchschnittlich ca. 100 Zeile C-Code oder ca. 50 Zeilen C++-Code

Eine Möglichkeit des Nachweises der Vollständigkeit der durchgeführten Tests (auch als Testabdeckung bezeichnet) kann mittels verschiedener Überdeckungsanalysen erfolgen. Dabei wird analysiert, welcher Teil des Codes während der Tests bereits ausgeführt wurde. Die einzelnen Analysemethoden sowie deren Anforderungen an die Beobachtbarkeit werden in Abschnitt 2.2 ausführlich diskutiert. Neben den Überdeckungsanalysen gibt es noch Vollständigkeitsnachweise von Tests auf funktionaler Ebene. Hier wird beispielsweise der Umfang von getesteten Anforderungen, von Wertebereichen oder von Grenzwerten ermittelt.

Der Test von Funktionen einer Komponente erfolgt oftmals mit einer diskreten Menge vordefinierter Parameter, die aus sogenannten Äquivalenzklassen [14] erstellt werden. Dabei werden alle Eingangswerte als äquivalent angesehen, bei denen das Vorliegen eines Defekts ein vergleichbares Fehlverhalten der Komponente zur Folge hätte. Dieses Vorgehen kann noch durch eine Grenzwertanalyse verfeinert werden, hier testet man die Werte, die am Rande einer Äquivalenzklasse liegen.

Trotzdem kann es vorkommen, dass gerade bei zufälligen Eingabewerten eine Fehlfunktion der Komponente beobachtbar ist. Das entsprechende Verfahren, welches die Komponente mit einer Vielzahl zufälliger Eingabeparameter aufruft, wird als *Fuzzing* [34] bezeichnet. Neben der Prüfung, ob es bei den beliebigen Eingabewerten zu Abstürzen oder unbehandelten Ausnahmen gekommen ist, kann durch Überprüfung der von der zu testenden Komponente zurückgegebenen Ergebnisse die Aussagekraft der Tests deutlich gesteigert werden (Testorakel). Diese Testorakel können auch noch bei höheren Testebenen sowie nach dem Release der Software Verwendung finden, indem die Überprüfung automatisiert bei beliebigen Läufen des Programms erfolgt (*Runtime Verification*, siehe Abschnitt 2.1.3).

Zur Durchführung von Komponententests muss der Beobachter in der Lage sein, die Rückgabewerte (im Stack oder in CPU-Registern abgelegt) bzw. Änderungen an globalen Variablen den einzelnen Tests zuzuordnen, um diese auf Korrektheit prüfen zu können.

Herkömmliche Komponententests werden Schritt für Schritt abgearbeitet, so dass hier keine besonderen Anforderungen an die Beobachtungsdauer gestellt werden müssen. Im Gegensatz dazu ist bei *Fuzzing*-Tests oder bei der Anwendung von *Runtime Verification* eine möglichst lange Beobachtungsdauer der Ein- und Ausgabewerte hilfreich, um die Wahrscheinlichkeit zu erhöhen, ein beobachtbares Fehlverhalten zu provozieren.

Integrationstest

Die Schnittstellen zwischen Software-Komponenten (Software/Software-Integrationstest) sowie zwischen Hardware- und Software-Komponenten (Hardware/Software-Integrationstest) und deren gegenseitiges Zusammenspiel wird mit dem Integrationstest getestet. [14] beinhaltet eine ausführliche Diskussion der Integrationstests. Die folgenden Ausführungen fassen die wesentlichen Aussagen zu dem Thema aus dieser Quelle unter besonderer Berücksichtigung der Anforderungen an die Beobachtbarkeit zusammen.

Ziel des Integrationstests ist es, Schnittstellen- und Protokollfehler aufzudecken. Dies können beispielsweise inkompatible Schnittstellenformate, unterschiedliche Interpretation der übergebenen Daten oder Timing-Probleme sein.

Ebenso wie beim Komponententest werden die zu prüfenden Programmteile mittels Treibern aufgerufen. Benötigte, aber nicht verfügbare Programmteile oder Hardwarekomponenten werden durch *dummies* ersetzt.

Software/Software-Integrationstests können mittels verschiedener Verfahren durchgeführt werden.

Bei Bottom-up-Unit-Tests werden schrittweise die Schnittstellen einer Komponente zu der jeweils übergeordneten Komponente getestet, wobei mit den jeweils "untersten" Komponenten begonnen wird. Diese Methode hat bei zyklischen Abhängigkeiten ihre Grenze.

Beim strukturierten Integrationstest werden die Funktionsaufrufe eines Moduls zu anderen Modulen getestet. Dabei kann der Kontrollflussgraph auf die für diesen Test wesentlichen Merkmale reduziert und mit Methoden des Komponenten-Tests durchlaufen werden.

Eine dritte Methode arbeitet mit dem Messen der Integrationstestabdeckung *(Call Pair Coverage)*, welche den Anteil der ausgeführten Aufrufe von Unterprogrammen angibt. Diese Messung kann beispielsweise innerhalb von Systemtests durchgeführt werden und gibt Hinweise auf die Vollständigkeit eines Systemtests.

Im Rahmen des Integrationstests getestete Schnittstellen können beispielsweise Funktionsaufrufe, per Streaming übertragene Daten, aber auch gemeinsam genutzte globale Variablen sein. Zugriffe auf letztere stellen für Integrationstests ein Problem dar und sollten möglichst durch `get()`/`set()`-Methoden in Funktionsaufrufe gewandelt werden.

Auch bei Integrationstests muss berücksichtigt werden, dass eine vollständige Testabdeckung noch keine Gewähr bietet, dass die an der Schnittstelle übergebenen Daten für alle Anwendungsfälle korrekt sind. Ebenso wie beim Komponententest sollten Äquivalenzklassen gebildet und Grenzwerte getestet werden.

Ein weiteres Problem ist der Test von Systemen, bei denen eine Abhängigkeit von der Reihenfolge der Funktionsaufrufe besteht. Die diskutierten (und aktuell hauptsächlich verwendeten) Integrationsteststrategien gehen von einem sequentiellen Programmablauf aus. Dieser sollte konsequent eingehalten werden, um eine Vergleichbarkeit der Testergebnisse zu gewährleisten. Unterschiedliche Sequenzen können zu von außen nicht sichtbaren Änderungen innerer Zustände von Komponenten und damit zu einer Vergrößerung des Zustandsraums führen. Einen ähnlichen Effekt haben Unterbrechungen (Multitasking, Interrupts) und mögliche Wettlaufsituationen (*data races, deadlocks*), auch diese können die Vergleichbarkeit verschiedener Testläufe einschränken.

Für Software/Software-Integrationstests ist die Beobachtbarkeit der Rückgabewerte von Funktionsaufrufen erforderlich. Da die Tests von einem Testtreiber ausgeführt werden, kann die Kontrolle der Rückgabewerte in diesem durchgeführt werden.

Ebenfalls Bestandteil von Integrationstest sind **Ressourcentests**, die sich zwischen Software/Software- und Hardware/Software-Integrationstests ansiedeln lassen. Es wird zwischen statischen und dynamischen Ressourcentests unterschieden, wobei im Umfeld der Beobachtbarkeit von SoCs nur letztere relevant ist. Sie umfassen die Messung der CPU-Last und des Speicherverbrauchs und werden in Abschnitt 2.6 ausführlich diskutiert. Hier werden auch die entsprechenden Anforderungen an die Beobachtbarkeit erläutert.

Der **Hardware/Software-Integrationstest** prüft das Zusammenspiel zwischen Hardware und Software. Hier ist es wichtig, dass im Rahmen des Tests festgestellt werden kann, ob der gewünschte Hardware-Effekt erreicht wurde. Eine Beschreibung dieser Test sowie der Anforderungen an die Beobachtbarkeit ist in Abschnitt 2.9 zu finden.

Eine Verbesserung der verschiedenen Arten von Integrationstests kann erreicht werden, wenn mittels vorab definierter Regeln und Erwartungswerten bei beliebigen Programmläufen geprüft werden kann, ob die ausgetauschten Daten sowie vordefinierte Hardware-Parameter (CPU-Last, Speicherverbrauch) den Erwartungen entsprechen (*Runtime Verification*, siehe Abschnitt 2.1.3).

Systemtest

Beim Systemtest wird geprüft, ob das System den funktionalen und nichtfunktionalen Anforderungen des Anwenders genügt. Es wird das Zusammenspiel der einzelnen Komponenten des Systems in einer anwendungsnahen Umgebung geprüft.

Systemtest umfassen folgende Aufgaben:

- Funktionstests
- Leistungstests (Test des Verhaltens im Normalbereich an der Grenze zur Überlast)
- Stresstests (Test des Verhaltens bei Überlastung durch Wegnahme von Ressourcen)
- Alpha- und Beta-Tests (Tests unter Kundenbeteiligung)
- Regressionstests (Wiederholung von Tests mit neuen Versionen einer Software)

Beim Systemtest besteht die Gefahr, dass eventuelle Fehlfunktionen die angeschlossene Betriebsumgebung beschädigen können (z.B. falsche Ansteuerung eines Motors).

Abnahmetest

Der Abnahmetest (auch *User Acceptance Test*) wird gemeinsam mit dem Auftraggeber durchgeführt, um die Software unter realen Einsatzbedingungen (Szenarien) zu prüfen und festzustellen, ob sie die vertraglich festgelegten Merkmale erfüllt (Validierung).

Das Maß an Beobachtbarkeit, welches die Komponenten- und Integrationstests erfordern, entspricht bzw. übertrifft in den meisten Fällen die Anforderungen an die Beobachtbarkeit im Rahmen von System- und Abnahmetests.

Tests auf Host oder Zielsystem

In [14] wird diskutiert, an welchem Ort - Host oder Zielsystem - Tests durchgeführt werden sollen. Folgende Argumente, Softwaretests auf dem Host auszuführen, werden angeführt:

- oftmals schlechte Verfügbarkeit prototypischer Zielsystem, d.h. mehrere Entwickler / Tester konkurrieren um Zeit auf dem Zielsystem
- Zeitaufwand zum Einspielen der Software auf das Zielsystem
- geringere Funktionalität des Debuggers auf dem Zielsystem im Vergleich zum Debugger auf dem Host
- Ressourcenbeschränkung auf dem Zielsystem (z.B. instrumentierter Programmcode passt nicht in den Speicher des Zielsystems)
- leichterer Zugang zu Testergebnissen auf dem Host

Trotzdem ergeht die dringende Empfehlung, die Tests auf dem Zielsystem durchzuführen. Als Gründe dafür werden angeführt:

- Auffinden von Compiler- und Bibliotheksfehlern
- Vermeidung maschinenspezifischer Unterschiede in der Compilierung (z.B. Host mit *big endian*, Target mit *little endian*, unterschiedliche Datenbreiten)
- Mixed Mode Darstellung (Assembler / Hochsprache) ist oftmals nur auf dem Zielsystem möglich.
- Bei der Verwendung von Simulatoren muss berücksichtigt werden, dass sowohl Simulator als auch die CPU des Zielsystems fehlerhaft sein können.
- Routinen des Betriebssystems können teilweise verwendet werden, es müssen keine Stubs implementiert werden.
- Einschlägige Normen (z.B. ISO 26262 [35]) empfehlen den Test auf dem Zielsystem.

Weiterhin wird dafür plädiert, zumindest final alle Tests mit der Release-Version des Object-Codes durchzuführen. Eine Vertiefung dieser Fragestellung erfolgt bei der Diskussion der Software-Instrumentierung in Abschnitt 4.1.

Darüber hinaus kann es in vielen Fällen hilfreich sein, das tatsächliche Laufzeitverhalten sowie die Ressourcenauslastung auf dem Zielsystem unter realen Betriebsbedingungen beobachten zu können.

Die o.g. Argumentation schließt nicht aus, dass während der Entwicklung Tests auf dem Host ausgeführt werden und hier auch ein Großteil der aufzuspürenden Defekte gefunden werden kann. Trotzdem sollte nicht darauf verzichtet werden, zum Schluss alle Tests mit der Release-Version der Software auf dem Zielsystem noch einmal zu wiederholen.

2.1.3 Debuggen

Der Prozess der Feststellung, des Lokalisierens und des Beseitigens von Defekten in Programmen wird als Debuggen bezeichnet (ISO/IEC/IEEE 24765-2010 [10]).

> ISO/IEC/IEEE 24765-2010 [10]: 3.752
>
> *debug*
>
> *1. to detect, locate, and correct faults in a computer program.*
> *2. to detect, locate, and eliminate errors in programs. ISO/IEC 2382-1:1993, Information technology — Vocabulary— Part 1: Fundamental terms.01.05.07*
> *NOTE Techniques include use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operation, and traces.*

Bei der Suche nach Defekten wird unterschieden, ob diese bereits zu einem beobachtbaren Fehlverhalten geführt haben (reaktives Debuggen) oder nicht (proaktives Debuggen) [3].

Reaktives Debuggen

Wird bei den Tests oder – unangenehmer – beim Lauf eines Programms beim Anwender ein Fehlverhalten beobachtet, so kann mittels reaktivem Debuggen versucht werden, den zugrunde liegenden Defekt zu erkennen und zu beseitigen.

Oftmals ist der zugrunde liegende Defekt schnell gefunden und korrigiert. Deutlich aufwändiger ist die in vielen Fällen – besonders bei Mandelbugs - erforderliche systematische Suche nach einem Defekt, welcher ein beobachtetes Fehlverhalten verursacht hat.

Hier wird, teilweise auch unbewusst, oftmals die Methodik des „wissenschaftlichen“ Debuggens angewandt, welches die in Abbildung 2-10 dargestellten Schritte beinhaltet.

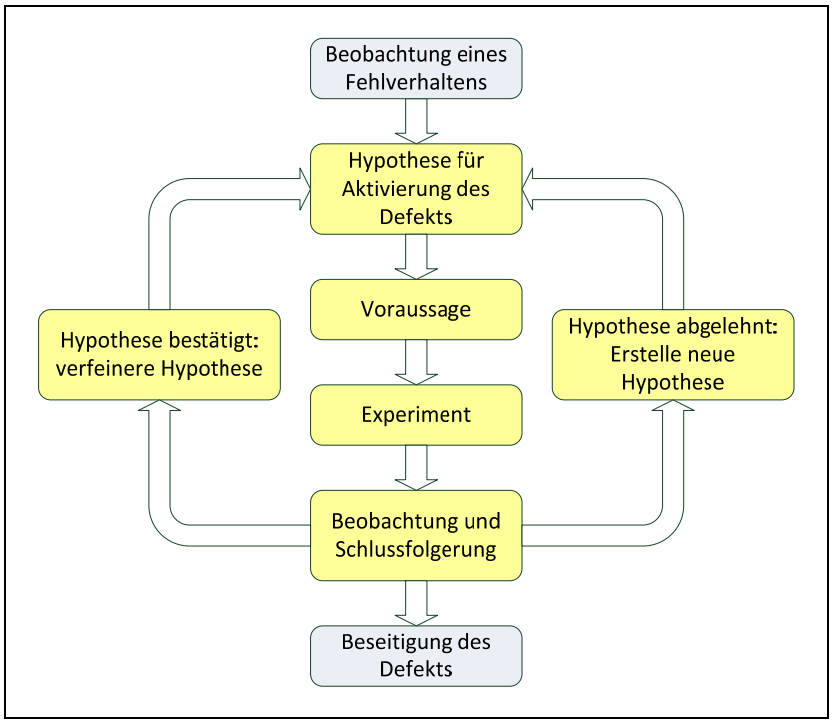


Abbildung 2-10: Wissenschaftliches Debuggen

Ausgehend von einem beobachteten Fehlverhalten werden Hypothesen aufgestellt, welche aktivierten Defekte die Ursache des Fehlverhaltens sein könnten. Für die einzelnen Hypothesen werden die Voraussagen des erwarteten Verhaltens experimentell überprüft. Bestätigt sich die initial erstellte Hypothese nicht, wird die nächste Hypothese überprüft. Entspricht das beobachtete Ergebnis der ursprünglichen Erwartung, so kann der Defekt beseitigt werden oder die Hypothese wird weiter verfeinert und erneut überprüft.

Bei diesem Vorgehen ist zu berücksichtigen, dass ein beobachtetes Fehlverhalten nicht nur die Aktivierung eines singulären Defekts, sondern auch durch die kombinierte Aktivierung mehrerer Defekte verursacht sein kann.

Falls Defekte gefunden und korrigiert wurden, muss sichergestellt werden, dass diese Korrektur auch tatsächlich erfolgte und der Defekt nicht nur maskiert wurde.

Die **Beobachtung** des Experiments ist mit verschiedenen Verfahren möglich. Hauptsächlich werden dazu die Techniken des Trace-Debuggings und des Omniscient-Debuggings eingesetzt.

Beim **Trace-Debugging** (auch **Run/Stop-Debugging**) wird das Programm ausgehend von einem ausgewählten Startpunkt schrittweise ausgeführt und dabei jeweils der Zustand des Programms überprüft (Abbildung 2-11). Weicht dieser von einem korrekten Zustand ab, ist die entsprechende Infektion gefunden. Das Trace-Debugging ist die am häufigsten eingesetzte Debug -Methode. Nachteilig ist, dass die Anzahl der zum Auffinden einer Infektion notwendigen Inspektionsschritte vom Abstand des Startzeitpunkts zur beobachtbaren Infektion abhängig ist. Bei einem ungünstig gewählten Startzeitpunkt kann entweder der Defekt bereits aktiviert sein oder die zu große Anzahl notwendiger Inspektionsschritte verhindert ein effizientes Arbeiten.

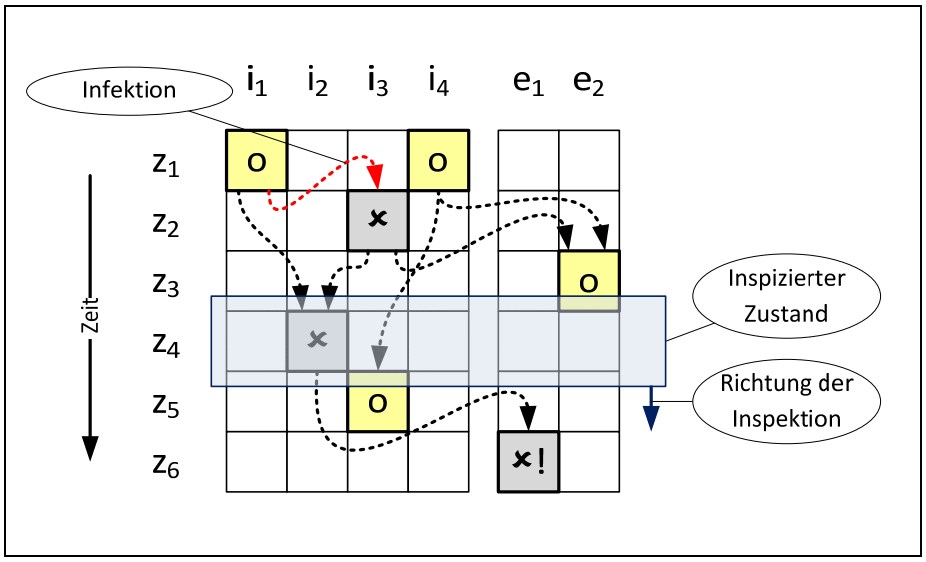


Abbildung 2-11: Trace-Debuggen[3]

Oftmals ist es notwendig, eine Vielzahl von Durchläufen zu starten. An dieser Stelle kann die Verwendung der bereits im Zusammenhang mit den Komponententests diskutierten Mock-Objekte (Abschnitt 0) hilfreich sein. Dabei wird der zu untersuchende Code-Abschnitt gekapselt und die Interaktion mit der restlichen Anwendung durch ein Mock-Objekt emuliert (Abbildung 2-12) [6]. Durch die damit mögliche Reproduzierbarkeit des Programmablaufs kann eine beliebige Zahl von Experimenten mit beliebigen Startzeitpunkten der Trace-Aufzeichnung vorgenommen werden.

[3] Komprimierte Darstellung entsprechend Abbildung 2-4

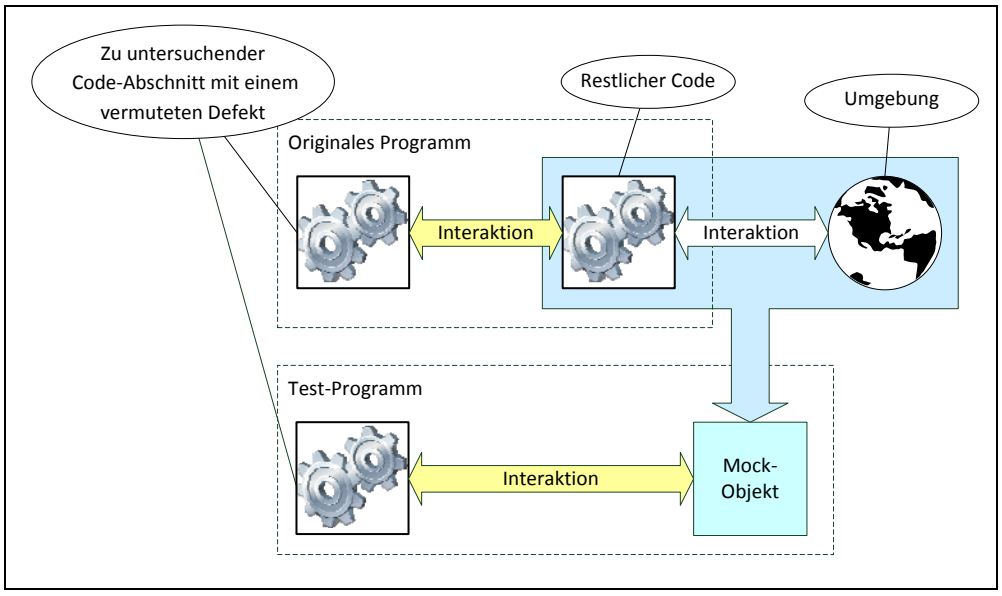


Abbildung 2-12: Verwendung eines Mock-Objekts

Die iterative Vorgehensweise beim wissenschaftlichen Debuggen setzt in vielen Fällen voraus, dass der zu entdeckende Defekt deterministisch aktivierbar ist.

Besonders bei Infektionen vom Typ 2 (Abhängigkeit von einem bereits infiziertem Wert) kann die Suche nach der Ursache eines beobachteten Fehlverhaltens schwierig sein. In diesem Fall müssen für den Debug-Prozess auch ältere, zwischenzeitlich eventuell bereits wieder überschriebene Zustände bekannt sein, von denen der fehlerhafte Zustand abhängig ist.

Hier stellt das **omnisciente Debuggen** eine effizientere Methode dar (omniscient: allwissend, auch *back-intime / reversible debugging*) [36], welches eine temporale Navigation durch den Programmablauf ermöglicht. Dazu werden vom Zeitpunkt des Beginns der Programmausführung an alle Programmzustände mit protokolliert. Ist eine Fehlerwirkung sichtbar, so kann von diesem Zeitpunkt an im Programm Schritt für Schritt rückwärtsgegangen werden (*back stepping*) (Abbildung 2-13). Ziel ist es, die Anweisung zu finden, bei welcher der das beobachtete Fehlverhalten verursachende Defekt aktiviert wurde.

Voraussetzung für das Omniscient-Debugging ist eine vollständige Aufzeichnung des Programmablaufs (Kontrollfluss und Zustandsänderungen). Da ein Teil der Zustände auch in den CPU-Registern gespeichert werden (z.B. Programmzähler, Registervariable) müssen auch diese mit protokolliert werden können.

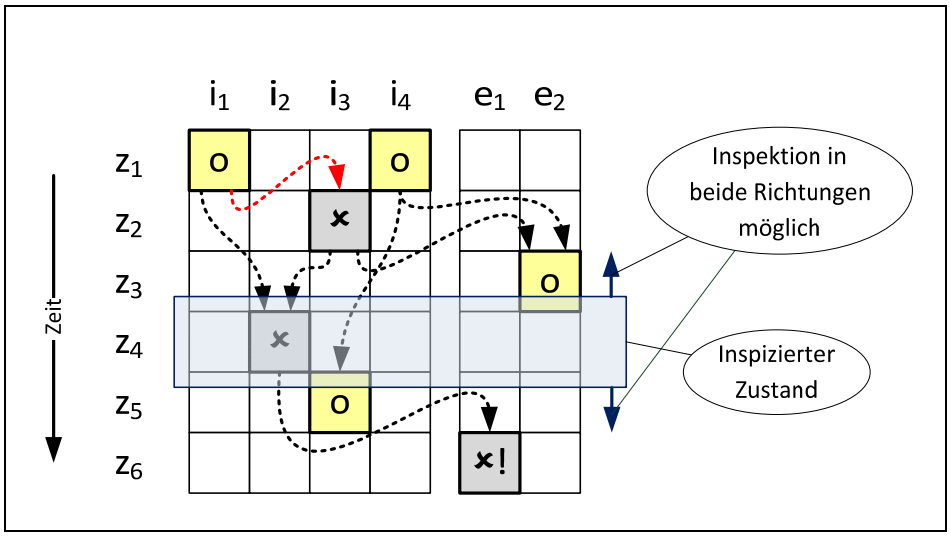


Abbildung 2-13: Omniscientes Debuggen[4]

Das zur Beobachtung von SoCs oftmals verwendete Aufzeichnen von Tracedaten ist im Sinne der o.g. Beschreibung kein Trace-Debuggen, sondern eine eingeschränkte Form des Omniscient-Debugging.

Proaktives Debuggen

Beim **proaktiven Debuggen** handelt es sich um die aktive Suche nach Defekten, die bisher noch nicht aktiviert wurden bzw. deren Aktivierung zu keinem beobachtbaren Fehlverhalten geführt hat.

Diese kann einmal mittels **statischer Methoden** erfolgen, bei denen der Code nur analysiert und bewertet, aber nicht ausgeführt wird. Diese Methoden umfassen Software-Reviews sowie statische Code-Analysen. Moderne Compiler sind die am häufigsten verwendeten statischen Analyse-Tools, sie werden durch spezielle Werkzeuge wie z.B. Coverity Prevent , CodeCheck oder PC-Lint [37] ergänzt.

Da für die statische Analyse eine Beobachtbarkeit der Vorgänge in einem SoC in der Regel nicht erforderlich ist (Ausnahme: hybride WCET-Analyse, siehe Abschnitt 2.3.3), wird dieses - für die Suche nach Defekten durchaus sehr wichtige - Thema hier nicht weiter vertieft.

Auch Software-Tests lassen sich definitionsgemäß (Suche nach Defekten) als proaktive Debug-Methoden charakterisieren.

Einen neuen Ansatz bietet die Methode der Runtime Verification [38]. Hier werden zur Laufzeit eines Programms beobachtete Systemzustände auf die Einhaltung vordefinierter Regeln überprüft (Abbildung 2-14).

[4] Komprimierte Darstellung entsprechend Abbildung 2-4

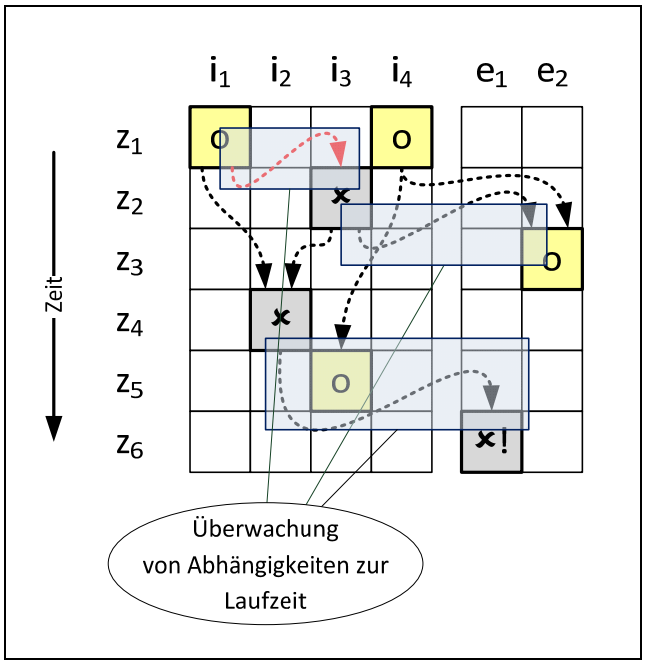


Abbildung 2-14: Runtime Verification[5]

2.1.4 Anforderungen an den Beobachter

Nicht erkannte Defekte können sich in jedem Entwicklungsstand einer Software befinden. Deshalb ist eine umfassende Beobachtbarkeit nicht nur beim Komponententest, sondern auch in den übergeordneten Tests sowie auch nach dem Release der Software erforderlich.

Die Ursachen von Fehlverhalten, welches bereits im Komponententest beobachtet wird, lassen sich üblicherweise leicht und schnell finden.

Idealerweise sollten bei Integrations- und den übergeordneten Tests die Möglichkeiten des omniscienten Debuggens verfügbar sein. Damit können auf der Suche nach der Ursache eines nichtdeterministisch auftretenden Fehlverhaltens auch eventuelle maskierte oder bereits wieder überschriebene infizierte Zustände rekonstruiert und damit der zugrunde liegende Defekt leichter erkannt werden.

Zur Erkennung aktivierter Defekte, die zu keinem falschen Testergebnis führen, ist es außerdem empfehlenswert, zusätzliche proaktive Debug-Methoden wie *Runtime Verification* anzuwenden.

Trotz der Verfügbarkeit vieler Zustandsinformationen wird es Fälle geben, bei denen ein Fehlverhalten in eingebetteten Systemen beobachtbar ist, dessen Ursache nicht eindeutig gefunden werden kann. So kann beispielsweise nachträglich nicht festgestellt werden, ob ein Fehlverhalten durch einen strahlungsinduzierten Defekt hervorgerufen wurde.

Die Beobachtbarkeit eines SoCs sollte aber so umfassend sein, dass die Ursachen von nichtdeterministischem Fehlverhalten weitestgehend nachvollziehbar sind.

Bei komplexen Systemen sind die Testaufbauten (Hardware-in-the-Loop-Systeme, Testracks etc.) sehr teuer und nur in begrenzten Stückzahlen vorhanden. Hier sind die Testzeiten entsprechend limitiert und teuer. An dieser Stelle spielt die Zeitdauer, die für die Durchführung der einzelnen Tests sowie für den Zugriff auf die Beobachtungsergebnisse erforderlich ist, eine wichtige Rolle. Je kürzer sie ist, desto mehr Tests können pro Zeiteinheit durchgeführt werden.

In Tabelle 2-5 sind die Anforderungen an den Beobachter für die Durchführung funktionaler Tests (auf die zugehörigen Überdeckungsanalysen wird im folgenden Abschnitt eingegangen) sowie für das Debuggen zusammengefasst. Bei omniscienten Debuggen muss eine Historie („H“) der während

[5] Komprimierte Darstellung entsprechend Abbildung 2-4

des Programmlaufs erfassten Informationen verfügbar sein. Die zu den Integrationstests gehörenden Ressourcentests werden in Abschnitt 2.6 behandelt.

		Funktionale Tests				Runtime Verification	Debuggen	
		Komponententest	SW/SW -Integrationstest	HW/SW -Integrationstest	Systemtest		einfach	omniscient
Vollständigkeit der Beobachtung	Taskwechsel					✓	✓	✓ (H)
	Funktionen	✓	✓	✓	✓	✓	✓	✓ (H)
	Basic Blocks					✓	✓	✓ (H)
	Instruktionen					✓	✓	✓ (H)
	Sprünge					✓	✓	✓ (H)
	Datenzugriffe (CPU)	✓	✓		✓	✓	✓	✓ (H)
	Datenzugriffe (Peripherie)	✓	✓	✓	✓	✓	✓	✓ (H)
	CPU-Register	✓	✓			✓	✓	✓ (H)
	Cache					✓	✓	✓ (H)
	Abarbeitung					✓	✓	✓ (H)
	Zyklusgenauigkeit					✓	✓	✓ (H)
	Bussystem			✓	✓	✓	✓	✓ (H)
	Umgebungsbedingungen			✓		✓	✓	✓ (H)
	Ereignisse		✓	✓	✓	✓	✓	✓ (H)
Beobachtung von MPSoCs			✓	✓	✓	✓	✓	✓ (H)

Tabelle 2-5: Anforderungen an den Beobachter bei funktionalen Tests und beim Debuggen

2.2 Überdeckungsanalysen

Mit einer Analyse von ausgeführten Instruktionen lässt sich ermitteln, inwiefern tatsächlich alle Teile des Programms sowie alle möglichen Verzweigungen getestet wurden.

Kontrollflussorientierte Testverfahren, die auch als Code-Überdeckungstest oder Code-Überdeckungsanalyse bezeichnet werden, sind ein wichtiges Instrument für den formalen Nachweis einer ausreichenden Testabdeckung und werden in vielen Zertifizierungsprozessen gefordert. Sie stellen ein objektives Vollständigkeitskriterium für Tests dar und können beispielsweise als Testende-Kriterium bei Komponententests verwendet werden. An dieser Stelle muss berücksichtigt werden, dass der Nachweise einer ausreichenden Testabdeckung noch keine Gewähr für die vollständige Erfüllung der gestellten funktionalen Anforderungen darstellt.

Es existiert eine Vielzahl von Werkzeuge zur Code-Überdeckungsanalyse, eine Übersicht ist in [39] zu finden.

Die in der Literatur verwendete Nomenklatur für die unterschiedlichen Tiefen von Code-Überdeckungsanalysen (C0 bis C3) ist teilweise uneinheitlich. Im Folgenden werden die in Tabelle 2-6 aufgeführten Begriffe verwendet.

Name	Englische Bezeichnung	Testabdeckungsgrößen
Funktionsüberdeckungstest	*Function Coverage*	
Anweisungsüberdeckungstest	*Statement Coverage*	C0
Zweigüberdeckungstest	*Branch Coverage*	C1
Pfadüberdeckungstest	*Path Coverage*	C2
Vollständig	*Full Path Coverage*	C2a
Boundary-Interior	*Boundary-Interior Path Coverage*	C2b
Strukturiert	*Structured Path Coverage*	C2c
Bedingungsüberdeckungstest	*Condition Coverage*	C3
Einfachbedingung	*Simple Condition Coverage*	C3a
Mehrfachbedingung	*Multiple Condition Coverage*	C3b
Minimale Mehrfachbedingung	*Minimal Multiple Condition Coverage (MMDC)*	C3c
Modifizierter Bedingungs-/Entscheidungsüberdeckungstest	*Modified Condition / Decision Coverage (MC/DC)*	

Tabelle 2-6: Code-Überdeckungsanalysen

In relevanten Normen werden je nach Kritikalität der Anwendung verschiedene Überdeckungsanalysen gefordert. In Tabelle 2-7 sind die Empfehlungen der ISO 26262 [35], Teil 6, Absatz 9.4.5 für Überdeckungsanalysen abhängig von der Sicherheitsklasse der Software angegeben. Dabei steht „++“ für „besonders empfohlen“ und „+“ für „empfohlen“. Die Sicherheitsklassen sind im Automobilbereich werden als ASIL (*automotive safety integrity level*) A bis D angegeben, wobei letztere die höchste Einstufung darstellt.

		Sicherheitsklasse (ASIL) (D: höchste)	A	B	C	D
Methode	1a	Anweisungsüberdeckungstest	++	++	+	+
	1b	Zweigüberdeckungstest	+	++	++	++
	1c	MC/DC (Modifizierter Bedingungs-/Entscheidungsüberdeckungstest)	+	+	+	++

Tabelle 2-7: Empfehlungen der ISO 26262 für Überdeckungsanalysen abhängig von der Sicherheitsklasse der Software.

In Tabelle 2-8 sind die Vorgaben für den Bereich der (zivilen) Luftfahrt angegeben, die von der Norm DO-178C [40] für Überdeckungsanalysen abhängig von der Sicherheitsanforderungsstufe (*Design Assurance Level, DAL*) der Software festgelegt werden. Die Sicherheitsanforderungsstufen orientieren sich dabei an den Konsequenzen möglicher Softwarefehler:

- Level A: katastrophale Auswirkungen
- Level B: gefährliche/schwerwiegende Auswirkungen
- Level C: erhebliche Auswirkungen
- Level D: geringfügige Auswirkungen
- Level E: keine Auswirkungen

	Sicherheitsanforderungsstufe (DAL) (A: höchste)	C	B	A
Methode	Anweisungsüberdeckungstest	✓		
	Zweigüberdeckungstest	✓	✓	
	MC/DC (Modifizierter Bedingungs-/Entscheidungsüberdeckungstest)	✓	✓	✓

Tabelle 2-8: Forderungen der DO-178C für Überdeckungsanalysen abhängig von der Sicherheitsanforderungsstufe der Software.

Die einfachste Überdeckungsanalyse ist die Ermittlung der Funktionsüberdeckung, d.h. es wird bestimmt, welche Funktionen innerhalb eines Testdurchlaufs aufgerufen wurden. Da die Aussage dieser Analyse sehr begrenzt ist, werden im Folgenden die in Tabelle 2-6 aufgeführten weitergehenden Überdeckungsanalysen diskutiert. Diese werden anhand eines einfachen Code-Segments sowie des zugehörigen Kontrollflussgraphen (Abbildung 2-15) (nach [39], Abschnitt 5.3) erläutert. Dabei stellen die Knoten des Graphen Aktivitäten (Operationen) dar, die Kanten stellen Abhängigkeiten zwischen den Operation dar. Ein *Basic Block* [41] ist eine Folge von Knoten, die genau einen Ein- und einen Austrittspunkt haben. Ein Pfad ist eine Folge von über Kanten verbundenen Knoten.

Die Tests sind auf hochsprachlicher Ebene (C, C++) definiert. Der als Ergebnis der Übersetzung entstandene Assemblercode kann wesentlich mehr Zweige und Pfade beinhalten.

Die Metrik der Testüberdeckung wird wie folgt angegeben:

$$\text{Codeüberdeckung} = \frac{\text{Anzahl der ausgeführten Testbestandteile}}{\text{Anzahl aller möglichen Testbestandteile}}$$

Bei allen Überdeckungstest muss berücksichtigt werden, dass die in der Hochsprache beobachtbare Überdeckung nicht unbedingt der Überdeckung des Object-Codes entspricht [42] [14].

Einfluss auf den Object-Code haben Compiler-Optimierungen oder vom Compiler zusätzlich eingefügter Code.

Der Compiler kann beispielsweise an folgenden Stellen Code einfügen:

- Aufruf von Libraries
- Konvertierungen (z.B. float nach Integer)

Wenn der Compiler angewiesen wird, den Code zu optimieren, ist es in vielen Fällen schwierig, nachträglich eine Beziehung zwischen Quellcode und dem erzeugten Object-Code herzustellen. Der Compiler kann verschiedene Anweisungen im Source-Code zusammenführen, entfernt selbständig überflüssigen *(„dead“)* Code oder ändert die Reihenfolge der Abarbeitung von Instruktionen.

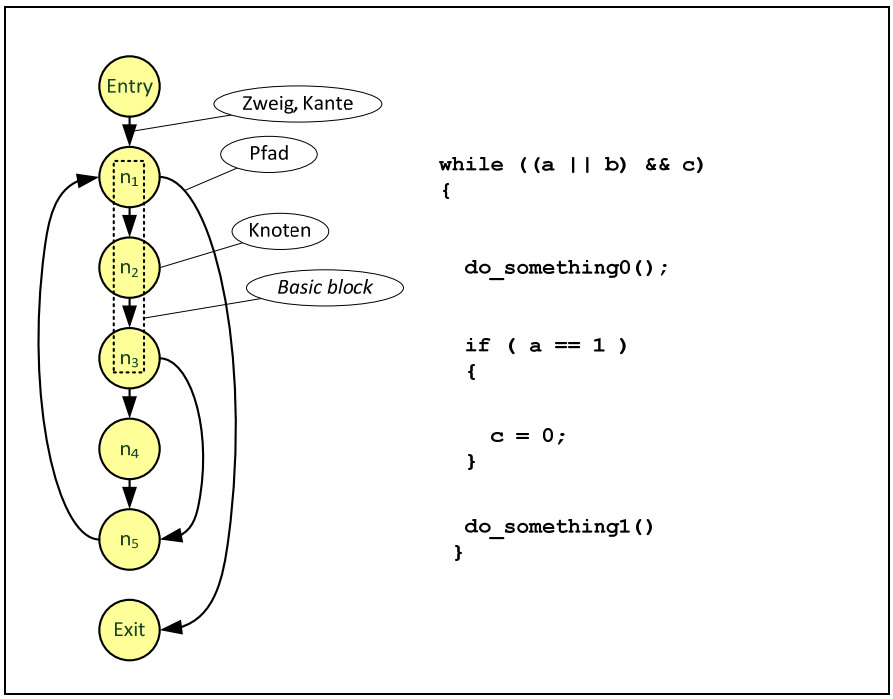


Abbildung 2-15: Kontrollflussgraph

2.2.1 Anweisungsüberdeckungstest

Ziel des Anweisungsüberdeckungstests ist das Protokollieren des mindestens einmaligen Durchlaufs jedes Knotens im zu analysierenden Programmabschnitt.

Der Anweisungsüberdeckungstest stellt sicher, dass im Programm kein „toter“ Code (d.h. niemals ausgeführter Code) existiert. Der Anweisungsüberdeckungstest hat nur eine geringe Fehleridentifizierungsquote, sie wird in der (ggfs. mittlerweile veralteten) Literatur mit 18% angegeben [39].

Selbst wenn in einem Test alle Knoten ausgeführt werden, ist dies noch keine Gewähr dafür, dass auch alle Zweige ausgeführt werden (Abbildung 2-16).

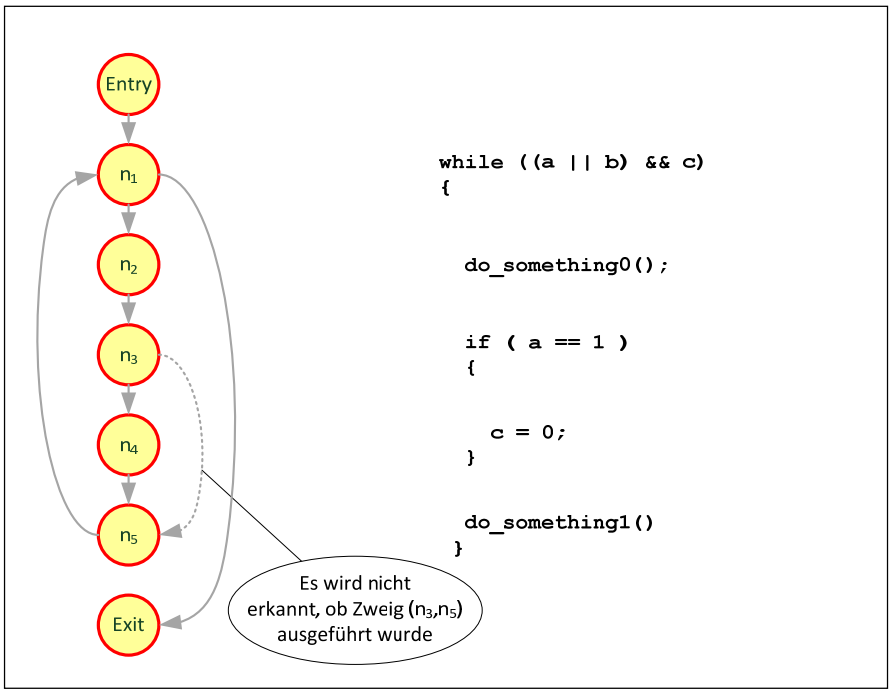
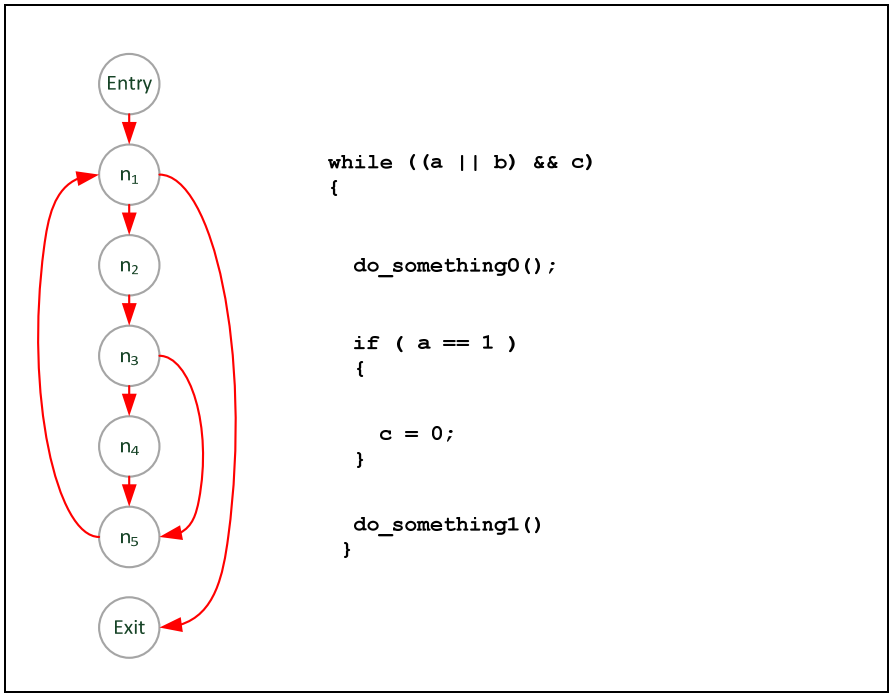


Abbildung 2-16: Test der Anweisungsüberdeckung[6]

2.2.2 Zweigüberdeckungstest / Entscheidungsüberdeckungstest

Bei der Analyse der Zweigüberdeckung wird protokolliert, inwiefern in einem Testlauf alle Zweige eines Programms durchlaufen wurden (Abbildung 2-17). Eng verwandt und oft synonym verwendet wird der Entscheidungsüberdeckungstest (*Decision Coverage*), welcher auf einer Analyse der den Programmablauf beeinflussenden Entscheidungsausgänge beruht. Dabei entspricht eine vollständige Entscheidungsüberdeckung auch einer Zweigüberdeckung von 100% [14].



[6] Erläuterungen in Abbildung 2-15

Abbildung 2-17: Test der Zweigüberdeckung[7]

Der Anweisungsüberdeckungstest ist in dem Zweigüberdeckungstest vollständig enthalten. Die Fehleridentifizierungsquote des Zweigüberdeckungstests wird in der (ggfs. mittlerweile veralteten) Literatur mit 34% angegeben [39].

Werden die Durchläufe der einzelnen Zweige mitgezählt, so ergibt sich eine wichtige Grundlage für weitergehende statistische Analysen und Programmoptimierungen.

Der Zweigüberdeckungstest berücksichtigt weder die Kombination von Zweigen noch komplexe Bedingungen, wodurch entsprechende Fehler nicht erkannt werden.

In Abbildung 2-18 ist ein Beispiel angegeben, welches mit den Werten `a=10` und `b=20` (Zweig (n_1,n_2) und Zweig (n_4,n_6)) sowie mit `a=11` und `b=10` (Zweig (n_1,n_3) und Zweig (n_4,n_5) alle Pfade erfolgreich durchläuft (C_{Zweig} = 100%). Wird aber die Reihenfolge der Pfade geändert (z.B. `a=5`, `b=7`), tritt eine Fehlfunktion (Division durch „0“) auf. Eine weitere Limitation des Zweigüberdeckungstests ist, dass Schleifen nicht ausreichend getestet werden, da ein einzelner Durchlauf des Schleifenkörpers für die Zweigüberdeckung ausreichend ist.

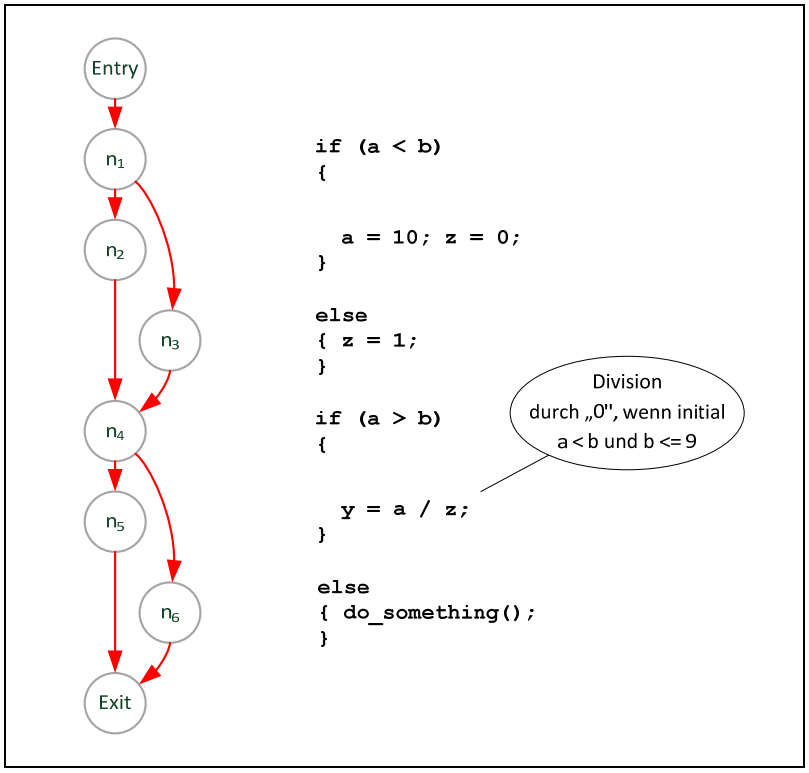


Abbildung 2-18: Zweigüberdeckungstest mit möglichem unerkannten Fehler[8]

2.2.3 Pfadüberdeckungstest

Der Pfadüberdeckungstest ermittelt, inwiefern in einem Testlauf alle möglichen Pfade eines Programms durchlaufen wurden. Wenn initial die Anzahl möglicher Schleifendurchläufe nicht bekannt ist (`do`-, `while`-Schleifen), ergibt sich eine sehr hohe Anzahl möglicher Pfade durch Hinzufügung des im Schleifenkörper vorhandenen Subpfades [39]. In dem Kontrollflussgraphen aus Abbildung 2-19 verlängert sich der Pfad der `while`-Schleife durch Hinzufügen der Subpfade p1oder p2 kontinuierlich bis zum Zeitpunkt des Verlassens der Schleife.

[7] Erläuterungen in Abbildung 2-15
[8] Erläuterungen in Abbildung 2-15

Aus diesem Grund gibt es auch vereinfachte Pfadüberdeckungstests wie den Boundary-Interior-Test oder den strukturierten Pfadüberdeckungstest.

Beim Boundary-Interior-Test werden Testfälle erstellt, die alle Pfade abdecken und sicherstellen, dass für alle Schleifen

- jede Schleife nicht betreten wird
- jede Schleife genau einmal durchlaufen wird (Boundary Test)
- jede Schleife zweimal durchlaufen wird (Interior Test)

Beim strukturierten Pfadüberdeckungstest werden Schleifen n-mal durchlaufen.

Der Pfadüberdeckungstest hat gegenüber dem Zweigüberdeckungstest eine deutlich höhere Fehleridentifizierungsquote, welche je nach Ausprägung Werte von ca. 65% erreichen kann ([43], [44], [45], ggfs. mittlerweile veraltetet).

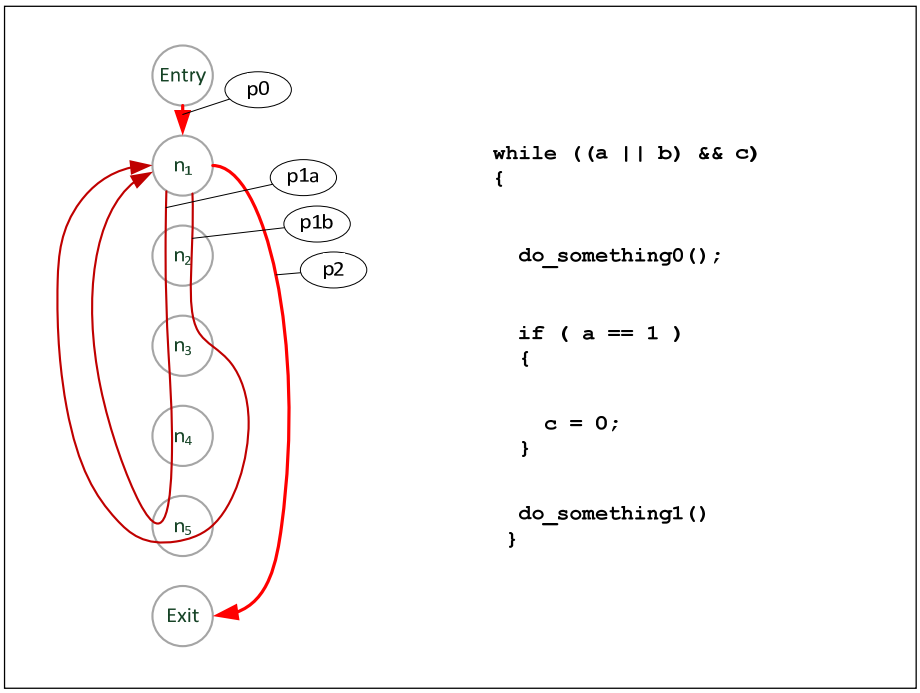


Abbildung 2-19: Pfadüberdeckungstest[9]

[9] Erläuterungen in Abbildung 2-15

2.2.4 Bedingungsüberdeckungstest

Der Anweisungsüberdeckungstest und der Pfadüberdeckungstest sind nicht in der Lage, zusammengesetzte und / oder hierarchische Bedingungen ausreichend zu testen. Dazu wird der Bedingungsüberdeckungstest verwendet. Für diesen gibt es die im Folgenden erläuterten unterschiedlichen Implementierungen.

Einfachbedingungsüberdeckungstest

Beim Einfachbedingungsüberdeckungstest wird jede Bedingung, die keine untergeordneten Bedingungen hat, mit jeweils einmal „Wahr“ und „Falsch“ getestet. Dadurch ist aber nicht sichergestellt, dass die Gesamtbedingung auch einmal „Wahr“ und „Falsch“ wird.

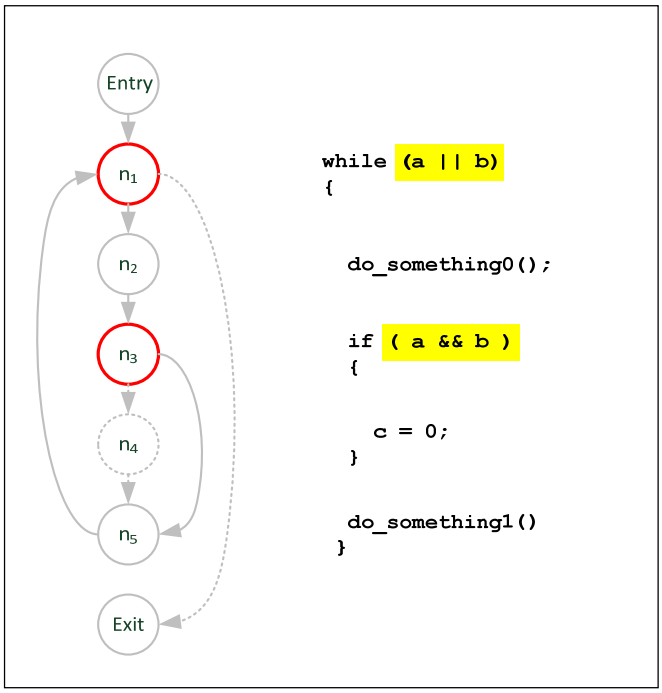


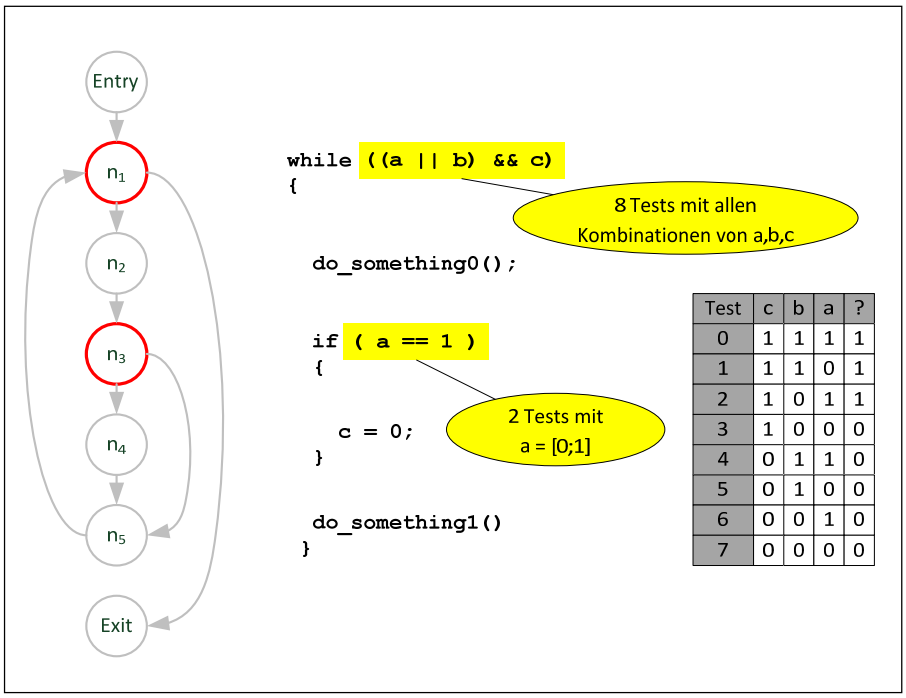
Abbildung 2-20: Einfachbedingungsüberdeckungstest[10]

Das Beispiel in Abbildung 2-20 zeigt, dass bei ungünstig gewählten Testparametern (a = 0, b = 1) einzelne Anweisungen nicht ausgeführt werden, d.h. dass der Einfachbedingungsüberdeckungstest schlechtere Resultate liefert als der Anweisungsüberdeckungstest.

Mehrfachbedingungsüberdeckungstest

Beim Mehrfachbedingungsüberdeckungstest (Abbildung 2-21) müssen alle möglichen Variationen der atomaren Bedingungen getestet werden. Der dafür erforderliche Aufwand ist bei zusammengesetzten und / oder hierarchischen Bedingungen sehr hoch, da bei einer Anzahl n der atomaren Bedingungen die Anzahl der notwendigen Testfälle 2^n beträgt.

[10] Erläuterungen in Abbildung 2-15



Test	c	b	a	?
0	1	1	1	1
1	1	1	0	1
2	1	0	1	1
3	1	0	0	0
4	0	1	1	0
5	0	1	0	0
6	0	0	1	0
7	0	0	0	0

Abbildung 2-21: Mehrfachbedingungsüberdeckungstest[11]

Bei der Implementierung von Funktionen können komplexe Bedingungen auch als bedingte Abfolge atomarer Bedingungen implementiert werden.

```
if (x)
{ if (z)
  { do something();
  }
}
else
{ if (y)
  { if (z)
    { do something();
    }
  }
}
```

Obwohl die Entscheidung nicht als komplexe Bedingung codiert ist, muss das Vorliegen der funktionalen Äquivalenz zu einer komplexen Bedingung erkannt

```
if ((x || y) && z)
{ do_something();
}
```

[11] Erläuterungen in Abbildung 2-15

und ein entsprechender Mehrfachbedingungsüberdeckungstest durchgeführt werden. Gleiches gilt auch für den modifizierten Bedingungs-/Entscheidungs-Überdeckungstest, welcher im Abschnitt 2.2.5 diskutiert wird [46].

Bei der Durchführung von Mehrfachbedingungsüberdeckungstests sollte beachtet werden, dass in einigen Fällen ein eindeutiger Nachweis der Ausführung der Testfälle auf der Basis der Analyse des Programmflusses auf Object-Code-Ebene nicht möglich ist.

Viele Sprachen wie C, C++, C# oder Java arbeiten mit der sogenannten „*lazy evaluation*". Der Compiler bricht die Prüfung weiterer atomarer Bedingungen einer komplexen Entscheidung ab, sobald das Ergebnis der Entscheidung feststeht [47]. Beispielsweise bedeutet das bei einem Ausdruck, der eine logische verUNDung enthält, dass ein nachfolgender Bestandteil des Ausdrucks nicht weiter evaluiert werden muss, wenn schon der erste Operand *„false"* ist, da das Ergebnis in jedem Fall *„false"* sein wird.

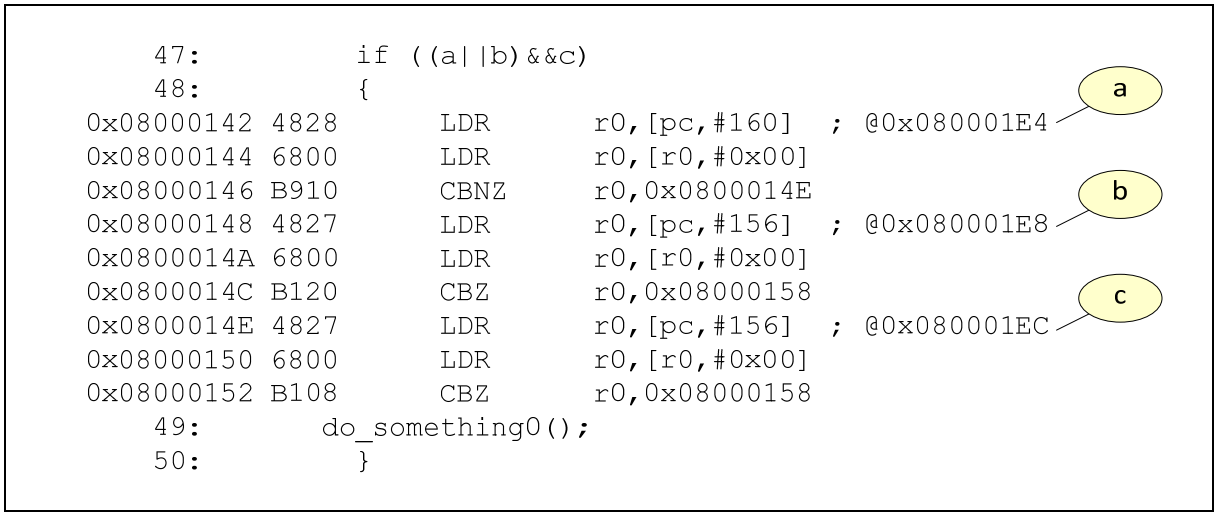


Abbildung 2-22: Beispielcode zur Prüfung der Bedingung ((a||b)&&c)

Das Beispiel in

Abbildung 2-22 (Code generiert mit Keil µVision für ARM Cortex-M3) verdeutlicht dieses Verhalten. Sind hier `a = 0` und `b = 0`, so wird die Prüfung von c niemals ausgeführt (maskiert), sondern es findet ein Sprung von Adresse 0x800014C über die Prüfung von c (0x8000152) statt.

	Atomare Bedingung			Sprungverhalten nicht unterscheidbar von Testfall	Komplexe Bedingung
Testfall	a	b	c		`(a||b)&&c`
0	1	x	1	2	1
1	0	1	1		1
2	1	x	1	0	1
3	0	0	x	7	0
4	1	x	0	6	0
5	0	1	0		0
6	1	x	0	4	0
7	0	0	x	3	0

Tabelle 2-9: Maskierte Testfälle (graue Markierung) durch „*lazy evaluation*"

Damit ergeben sich für dieses Beispiel folgende nicht unterscheidbare Testfälle (Tabelle 2-9):

- 0/2/4/6: der Wert von b wird nicht evaluiert, da er durch a == 1 maskiert wird
- 3/7: der Wert von c nicht evaluiert, da er durch (a||b) == 0 maskiert wird

Nur die Testfälle 1 und 5 lassen sich eindeutig aus der Beobachtung der ausgeführten Instruktionen ermitteln, zur eindeutigen Bestimmung der anderen ist die Kenntnis der jeweiligen Datenwerte erforderlich. An dieser Stelle lässt sich erkennen, dass es auf hochsprachlicher Ebene definierte Testfälle geben kann, die nie auf der Maschinencode-Ebene ausgeführt werden. Hier kann überlegt werden, die Testfälle entsprechend ihrer tatsächlichen Ausführbarkeit im Maschinencode zu reduzieren.

Minimaler Mehrfachbedingungsüberdeckungstest

Um die Zahl der Testfälle beim Mehrfachbedingungsüberdeckungstest zu reduzieren, wird beim Minimalen Mehrfachbedingungsüberdeckungstest (Abbildung 2-23) jede atomare Bedingung sowie die Gesamtbedingung auf „Wahr“ und „Falsch“ getestet.

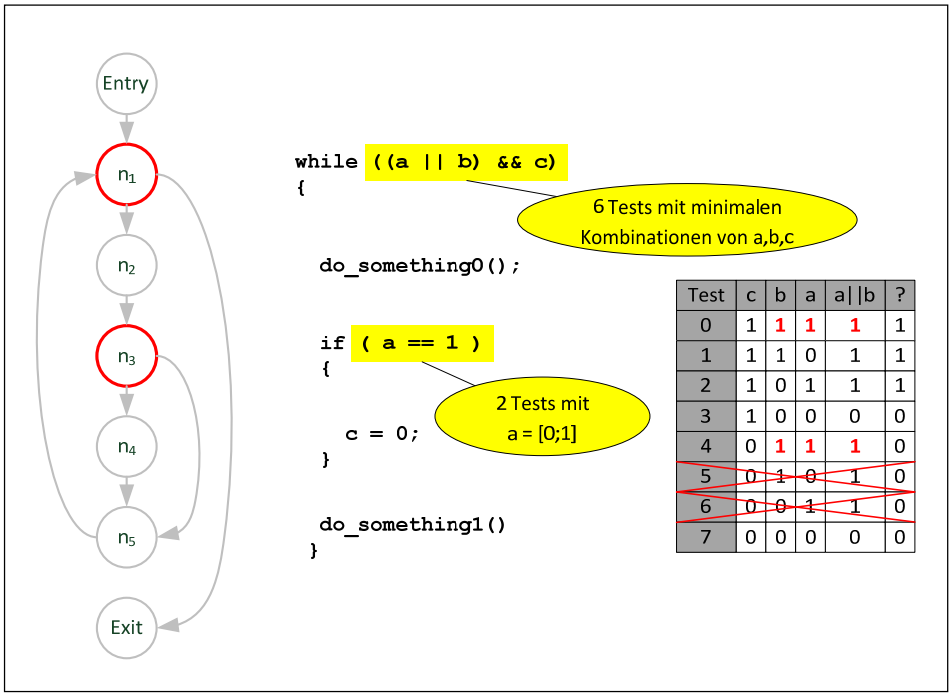


Abbildung 2-23: Minimaler Mehrfachbedingungsüberdeckungstest[12]

2.2.5 Modifizierter Bedingungs-/Entscheidungsüberdeckungstest (MC/DC)

Als Grundlage für die Zertifizierung von Software mit den höchsten Sicherheitsanforderungen (z.B. DO178-B/C [48][49]) ist der modifizierte Bedingungs-/ Entscheidungsüberdeckungstest [42], [50] vorgeschrieben (*Modified condition/decision coverage MC/DC*). Dieser Test stellt einen Kompromiss dar zwischen einem möglichst umfassenden Test der Logik von zusammengesetzten Entscheidungen und einem vertretbaren Testaufwand. Bei dem modifizierten Bedingungs-/ Entscheidungsüberdeckungstest werden Testfälle verlangt, die demonstrieren, dass jede atomare Bedingung, welche die sie umgebende komplexe Bedingung beeinflusst, mit „Wahr“ und „Falsch“ getestet wird [51].

[12] Erläuterungen in Abbildung 2-15

Es kann verschiedene Sätze von Testfällen geben, die diesen Anforderungen genügen. Bei n atomaren Bedingungen sind mindestens n+1 Testfälle erforderlich. Eine Anleitung zur Suche nach geeigneten Testfällen, die den MC/DC-Kriterien genügen, ist in [42] ausführlich erläutert. Hierbei handelt es sich um die strikte Anwendung der in der DO-178B [48] definierten MC/DC-Kriterien, die auch als „*Unique Cause MC/DC*“ [52] bezeichnet werden und für jeden Testfall nur die Änderung einer einzigen Bedingung erlauben.

	Atomare Bedingung				Komplexe Bedingung	Komplementäre Testfälle		
Testfall	a	b	c	a || b	(a || b) && c	a	b	c
T0	1	1	1	1	1			T4
T1	0	1	1	1	1		T3	T5
T2	1	0	1	1	1	T3		T6
T3	0	0	1	0	0	T2	T1	
T4	1	1	0	1	0			T0
T5	0	1	0	1	0			T1
T6	1	0	0	1	0			T2
T7	0	0	0	0	0			

Tabelle 2-10: Beispiel zur Bestimmung möglicher Testfälle für „*Unique Cause MC/DC*“ für den Ausdruck ((a||b)&&c)

Für das Beispiel in Abbildung 2-24 (aus Abschnitt 2.2.4) wurden die in Tabelle 2-10 ausgewählten Testfälle gefunden.

Zum Test von a müssen die Testfälle gefunden werden, bei denen ein Wechsel von a bei konstanten b und c eine Änderung des Resultats bewirkt. Gefunden wurden die komplementären Testfälle T2 und T3 mit a = [0,1] bei b = 0 und c = 1.

Zum Test von b müssen die Testfälle gefunden werden, bei denen ein Wechsel von b bei konstanten a und c eine Änderung des Resultats bewirkt. Gefunden wurden die komplementären Testfälle T1 und T3 mit b = [0,1] bei a = 0 und c = 1.

Zum Test von c müssen die Testfälle gefunden werden, bei denen ein Wechsel von c bei konstanten a und b eine Änderung des Resultats bewirkt. Gefunden wurden mehrere komplementäre Testfälle:

- Testfall T0 und T4 mit c = [0,1] bei a = 1 und b = 1
- Testfall T1 und T5 mit c = [0,1] bei a = 0 und b = 1
- Testfall T2 und T6 mit c = [0,1] bei a = 1 und b = 0

Aus den jeweiligen Testfällen für die Änderungen von a, b und c werden nun diejenigen komplementären Paare ausgewählt, die mit dem geringsten Aufwand (=Anzahl von Tests) alle erforderlichen Tests beinhalten. Im Beispiel ergeben sich folgende Tests (für a,b,c):

- [T2,T3], [T1,T3], [T0,T4] -> hier sind 5 unterschiedliche Testfälle (T0,T1,T2,T3,T4) enthalten
- [T2,T3], [T1,T3], [T1,T5] -> hier sind 4 unterschiedliche Testfälle (T1,T2,T3,T5) enthalten
- [T2,T3], [T1,T3], [T2,T6] -> hier sind 4 unterschiedliche Testfälle (T1,T2,T3,T6) enthalten

Für das besprochene Beispiel könnten mit den Testfällen [T1,T2,T3,T5], [T1,T2,T3,T6] (entsprechen n+1 Testfällen) und [T0,T1,T2,T3,T4] (ungünstiger, da ein Test mehr erforderlich ist) der modifizierte Bedingungs-/ Entscheidungsüberdeckungstest durchgeführt werden.

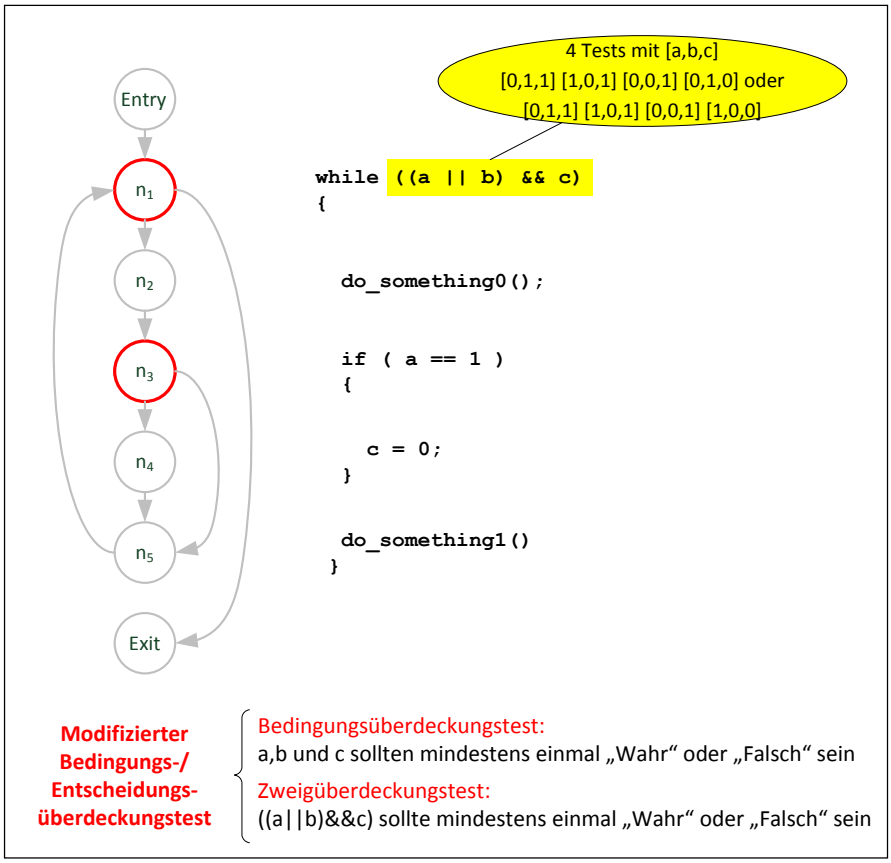


Abbildung 2-24: Modifizierter Bedingungs-/Entscheidungsüberdeckungstest (MC/DC)[13]

Wenn der Beobachter in der Lage ist, die Werte der atomaren Bedingungen zu erkennen, können die Sätze von Testfällen beliebig ausgewählt werden. Die Beobachtung von Variablen- und Registerwerten ist – je nach verfügbarer Methode – oftmals sehr aufwändig und teilweise nicht möglich (siehe Abschnitt 4). Alternativ dazu ist die Beobachtung der Werte atomarer Bedingungen indirekt über die Beobachtung des Sprungverhaltens realisierbar. An dieser Stelle muss wieder die bereits in Abschnitt 2.2.4 diskutierte Maskierung aufgrund der „*lazy evaluation*“ berücksichtigt werden.

[13] Erläuterungen in Abbildung 2-15

	MC/DC Variante			Sprungverhalten nicht unterscheidbar von Testfall
	1	2	3	
Testfall				
0				2
1				
2				0
3				7
4				6
5				
6				4
7				3

Tabelle 2-11: Durch *„lazy evaluation“* bedingte nicht unterscheidbare MC/DC Testfälle bei Beobachtung des Sprungverhaltens

Für o.g. MC/DC-Beispiel bedeutet dies, dass die Durchführung der für den MC/DC-Nachweis erforderlichen Testfälle [1,2,3,5], [1,2,3,6] und [0,1,2,3,4] sich nicht eindeutig aus dem Sprungverhalten bestimmen lassen (Tabelle 2-11). Dies steht der Anforderung für die *„Unique-Cause MC/DC“* Kriterien aus DO178-B [48] entgegen, nach denen für jeden Testfall nur jeweils eine Bedingung geändert werden darf. In [53] werden zwei zusätzliche Varianten von modifizierten Bedingungs-/ Entscheidungsüberdeckungstests (*„Unique-Cause+Masking MC/DC“*, *„Masking MC/DC“)* diskutiert, welche die Änderung mehrerer Bedingungen von Testfall zu Testfall und damit die aufgeführten äquivalenten Testfälle gestatten. Diese Varianten sind nun auch nach der im Jahr 2011 veröffentlichten DO178-C [49] zulässig und gestatten es, durch Beobachtung des Sprungverhaltens die Erfüllung der MC/DC-Kriterien nachzuweisen. Voraussetzung dafür ist eine eindeutige Zuordnung zwischen Hochsprache und Object-Code [54]. Eine ausführliche Diskussion dieser Problematik ist auch in [52] zu finden, im Rahmen des in dieser Veröffentlichung vorgestellten „COUVERTURE“-Projektes wurden auch GCC-Erweiterungen erarbeitet, die diese eindeutige Zuordnung zwischen Hochsprache und Object-Code erleichtern und deren Automatisierbarkeit verbessern.

Trotz der Forderung, den modifizierten Bedingungs-/ Entscheidungsüberdeckungstest beim Test von sicherheitskritischen Systemen anzuwenden, bietet dieser keine Garantie für das Auffinden aller Defekte, die mittels anderer Überdeckungstests identifizierbar wären.

Ein Bespiel dafür ist in [55] zu finden. Hier werden für die komplexe Bedingung (`a&&b||c`) (Tabelle 2-12) die Testfälle 4,5 und 6 sowie einer der Testfälle 1,2 oder 3 gefunden, die den MC/DC Kriterien entsprechen. Der Test 0, der eine fehlerhafte Implementierung (`a&&b` **`xor`** `c`) aufdecken würde, ist in dem modifizierten Bedingungs-/ Entscheidungsüberdeckungstest nicht zu finden.

	Atomare Bedingung			Komplexe Bedingung	Fehlerhafte Implementierung
Testfall	a	b	c	`(a && b || c)`	`(a && b` **`xor`** `c)`
0	1	1	1	1	0 (!)
1	0	1	1	1	1
2	1	0	1	1	1
3	0	0	1	1	1
4	1	1	0	1	1
5	0	1	0	0	0
6	1	0	0	0	0
7	0	0	0	0	0

Tabelle 2-12: Beispiel, bei dem der modifizierte Bedingungs-/Entscheidungsüberdeckungstest versagt

In [56] wird für Anwendungsfälle aus der Automobilindustrie die Wirksamkeit von modifizierten Bedingungs-/ Entscheidungsüberdeckungstests mit dem ernüchternden Ergebnis analysiert, dass hier bis zu 22% der eingebauten Defekte nicht gefunden wurden.

2.2.6 Datenflussorientierte Tests / Daten-Überdeckungsanalyse

Bei datenflussorientierten Testverfahren werden lesende und schreibende Zugriffe auf Variablen sowohl in Anweisungen als auch in Bedingungen zur Erzeugung von Testfällen benutzt [39].

Bei datenflussorientierten Testverfahren werden drei Zugriffe unterschieden:

- Wertzuweisung / Definition einer Variablen (`def`)
- Werteberechnung (*computational-use* , `c-use`)
- Prädikatsauswertung (*predicate-use*, `p-use`)

Ein Datenflussgraph ist ein erweiterter Kontrollflussgraph (Abbildung 2-25). Dazu wird dieser um einen Startknoten n_{in} erweitert, der mit dem alten Startknoten verbunden ist. Wenn in einem Knoten die Definition einer Variablen x stattfindet (z.B. `int x = 0;`), wird dieser Knoten mit `def(x)` markiert. Wird in einem Knoten eine Variable y zur Berechnung einer Zuweisung (z.B. `x = y++;`) verwendet, so wird der Knoten mit einer Werteberechnung `c-use(y)` markiert. Wird in einem Knoten eine von einer Variablen z abhängige Bedingung getestet (z.B. `if (z == 0){do_something()};`), so werden die ausgehenden Kanten mit `p-use(z)` markiert. Vor jedem Endknoten wird ein weiterer Knoten n_{out} eingefügt.

Im Folgenden sind einige Kriterien für datenflussorientierte Tests aufgeführt.

„all defs" Kriterium

Jede Wertzuweisung an eine Variable muss mindestens in einer Bedingung oder Berechnung verwendet werden.

Dieses Kriterium beinhaltet weder die Zweigüberdeckung noch die Anweisungsüberdeckung.

„all p-uses" Kriterium

Jede Wertzuweisung an eine Variable muss mit jeder zugehörigen Prädikatsauswertung dieser Variablen getestet werden, d.h. wenn in einem Knoten eine Wertzuweisung an einer Variable vorgenommen wird, so müssen in den Tests alle Pfade zu den Knoten verfolgt werden, in denen diese (unveränderte) Variable Bestandteil einer Bedingung ist.

Dieses Kriterium beinhaltet die Zweigüberdeckung.

„all c-uses“ Kriterium

Jede Wertzuweisung an eine Variable muss mit allen Werteberechnungen getestet werden, an welcher die (unveränderte) Variable beteiligt ist.

Dieses Kriterium beinhaltet weder die Zweigüberdeckung noch die Anweisungsüberdeckung.

Bei der Analyse der Datenzugriffskriterien muss berücksichtigt werden, dass auch durch Einlesen der Werte in CPU-Register diese damit nicht zwingend verwendet wird (*„lazy evaluation“*, siehe Abschnitt 2.2.4). Von daher ist auch die Kenntnis der ausgeführten Instruktionen für die Daten-Überdeckungsanalyse wichtig.

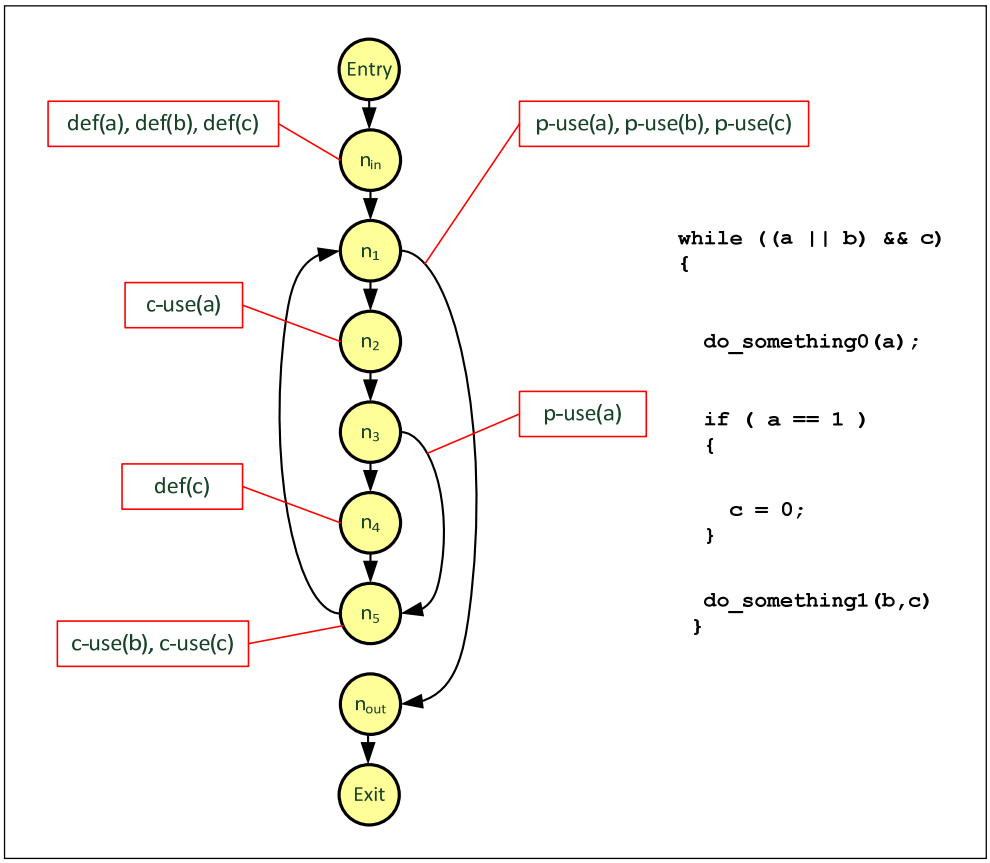


Abbildung 2-25: Beispiel eines mit Datenflussinformationen erweiterten Kontrollflussgraphen[14]

Datenflussorientierte Testverfahren gewinnen besonders beim Test von parallel auf MPSoCs ablaufender Software zunehmend an Bedeutung, da der Datenaustausch zwischen den einzelnen Kernen leichter beobachtbar ist als der zeitlich synchronisierte Programmablauf in den einzelnen Kernen (*Communication-centric debug*, siehe auch Abschnitt 2.1.3).

[14] Erläuterungen in Abbildung 2-15

2.2.7 Anforderungen an den Beobachter

Für die Beobachtung von Überdeckungstests muss der Beobachter folgende Informationen liefern können (Tabelle 2-13):

Funktionen

Wurde mindestens eine Instruktion innerhalb einer Funktion ausgeführt?

Instruktionen/ Basic Blocks

Wurde jede Instruktion ausgeführt bzw. ein *Basic Block* durchlaufen?

Bedingte Sprünge

Wurde ein bedingter Sprung jeweils ausgeführt und nicht ausgeführt?

Zusätzlich ist es für den Pfadüberdeckungstest notwendig zu wissen, welche Pfade vor Erreichen der aktuellen Instruktionsadresse durchlaufen wurden. Die Tiefe der aufzuzeichnenden Historie („H“) muss anhand des Kontrollflussgraphen ermittelt werden und kann ggfs. den vollständigen bisherigen Programmlauf umfassen (z.B. in Schleifen oder Rekursionen ohne Begrenzung).

Daten / CPU-Register

Für das Protokollieren von Entscheidungen ist die Kenntnis des Ergebnisses einzelner atomarer Entscheidungen notwendig. Dazu ist die Beobachtung von Datenzugriffen (Adresse „A“, Richtung „R“, Wert „W“) notwendig. Wenn der Inhalt von CPU-Registern nicht aus der Beobachtung vorangegangener Datenzugriffe (z.B. in Folge unvollständiger Beobachtbarkeit) rekonstruiert werden kann, müssen deren Werte ebenfalls bekannt gemacht werden.

		Überdeckungstest (ÜT)									
									MC/DC		
		Funktions-ÜT	Anweisungs-ÜT	Zweig-ÜT	Pfad-ÜT	Einfachbedingung-ÜT	Mehrfachbedingung-ÜT	Minimaler Mehrfach-bedingungs-ÜT	*"Unique Cause"*	*"Masking"*	Daten-ÜT
Vollständigkeit der Beobachtung	Funktionen	✓									
	Basic Blocks / Instruktionen		✓	✓	✓	✓	✓	✓	✓	✓	✓
	Sprünge			✓	✓ H					✓ H	
	Datenzugriffe (CPU)					✓	✓ ARW	✓ ARW	✓ ARW		✓ AR
	Datenzugriffe (Peripherie)										✓ AR
	CPU-Register						(✓)	(✓)	(✓)		

Tabelle 2-13: Anforderungen an den Beobachter bei Überdeckungstests

2.3 Ermittlung von maximalen Ausführungszeiten

Bei der Ausführung von Programmen in Echtzeitsystemen besteht die Notwendigkeit, dass zeitliche Rahmenbedingungen zuverlässig eingehalten werden [57]. Maximale Ausführungszeiten eines Codeabschnittes können mittels statischer Analyse bestimmt oder durch Messungen ermittelt werden. Die Kombination beider Verfahren wird als „hybride"- Analyse bezeichnet. Zur Analyse wird der zu untersuchende Code in kleine Segmente unterteilt, diese werden dann einzeln analysiert und die Ergebnisse zusammengefügt (*divide-and-conquer strategy*).

2.3.1 Einflussfaktoren

Die Ausführungsgeschwindigkeit einer Programmroutine ist von einer Reihe von Faktoren abhängig. Durch geeignete Programmiertechniken wie der Limitierung der Anzahl von Schleifendurchläufen oder der Beschränkung der Tiefe von rekursiven Funktionen lässt sich die maximale Laufzeit einer Funktion begrenzen. Trotz dieser Maßnahmen kann die Dauer der Ausführung des gleichen Programmteils variieren.

So besteht eine große Lücke zwischen der Taktfrequenz der CPU und der Zugriffszeit auf die verschiedenen Speicher, die je nach Speichertyp auch differieren. Der Zugriff auf den Cache erfolgt am schnellsten, der Zugriff auf internen Programmspeicher langsamer und der Zugriff auf den externen Programmspeicher noch langsamer. Dies bedeutet, dass der Zustand des Caches einen großen Einfluss auf die Ausführungszeit hat. Eventuelle Störungen des Caches, beispielsweise durch das Betriebssystem oder durch den Aufruf von Interrupt-Service-Routinen verursacht, können einen deutlichen Einfluss auf die Ausführungszeit haben. Einen ähnlichen Effekt hat auch der Übersetzungspuffer (*Translation Lookaside Buffer - TLB*) bei SoCs mit Speicherverwaltungseinheiten (*Memory Management Unit* - MMU).

Neben dem Cache und Übersetzungspuffer verfügen moderne SoCs über eine ganze Reihe aufwendiger Mechanismen zur Beschleunigung der Codeabarbeitung, welche damit auch direkten Einfluss auf die Ausführungszeiten haben und die Lücke zwischen der maximalen Ausführungszeit und der durchschnittlichen Ausführungszeit vergrößern, da bei pessimistischer Analyse davon ausgegangen werden muss, dass die effizienzsteigernden Mechanismen in ungünstigen Fällen keine Verbesserung der Ausführungszeit bewirken. Zusätzlich kann die Ausführungszeit auch durch eventuelle Wartezeiten bei Buszugriffen bzw. auf Peripherieeinheiten beeinflusst werden.

Weiterhin wird die Ausführungsgeschwindigkeit einzelner Instruktionen vom Zustand der Pipeline bestimmt. Diese *„scheduling anomalies"* [58][59] resultieren aus der Abhängigkeit der Verarbeitungsdauer einer Instruktionen von der vorher ausgeführten Instruktionen.

Zur Bestimmung der Ausführungszeiten gibt es sowohl statische Analyseverfahren als auch dynamische Analysen. Zu pessimistische Annahmen (Überschätzung) führen zu einer Überdimensionierung der Hardware mit den damit verbundenen höheren Kosten. Zu optimistische Annahmen (Unterschätzung) wiederum können die Einhaltung der zeitlichen Rahmenbedingungen gefährden und sind auf jeden Fall zu vermeiden.

2.3.2 Statische Analyse der Ausführungszeit von Basisblöcken

Werkzeuge zur statischen Analyse der maximalen Ausführungszeit führen drei Teilaufgaben aus:

- Ermittlung des Kontrollflussgraphen
- Berechnung der Ausführungszeiten von Basisblöcken
- Bestimmung des „längsten“ Pfades

Die statische Analyse stellt eine zuverlässige Methode zur Vorhersage von Ausführungszeiten dar und hilft dem Entwickler, die Pfade mit der längsten Ausführungszeit zu identifizieren und ggfs. zu optimieren. Diese Analyse kann ohne Messungen und ohne die Notwendigkeit der Verfügbarkeit einer Zielhardware durchgeführt werden. Dazu wird der Kontrollflussgraph des Programmcodes bestimmt und die einzelnen Zweige analysiert. Es gibt Einflussfaktoren, die eine statische Analyse deutlich erschweren bzw. unmöglich machen. Dies sind z.B. erst zur Laufzeit bekannt werdende Sprunginformationen, welche manuell (und damit fehleranfällig) sowohl initial als auch bei jeder neuen Version der Software in die Analyse eingepflegt werden müssen.

Grundlage für eine statische Messung ist ein Modell, welches den Kern des SoCs präzise nachbildet. Dazu gehören vor allem die Elemente, die die Verarbeitungsdauer beeinflussen wie Pipeline, unterschiedliche Speicherstrukturen, Cache und Bussysteme. In den Modellen müssen eventuelle kundenspezifische Varianten des SoCs sowie unterschiedliche Maskenversionen berücksichtigt werden. Die Erstellung solcher Modelle ist sehr aufwändig und fehleranfällig. Dies begründet den sehr hohen Preis von zertifizierten Werkzeugen für die statische Analyse. Meist werden in dem Modell pessimistische Vereinfachungen vorgenommen, die längere Ausführungszeiten liefern, als sie in der tatsächlichen Hardware auftreten würden. Da von den ermittelten Zeiten die Dimensionierung eines Systems abhängt, kann eine statische Analyse, die auf einem vereinfachten (und damit pessimistischen) Modell basiert, zu deutlichen Mehrkosten des Systems führen (Überauslegung).

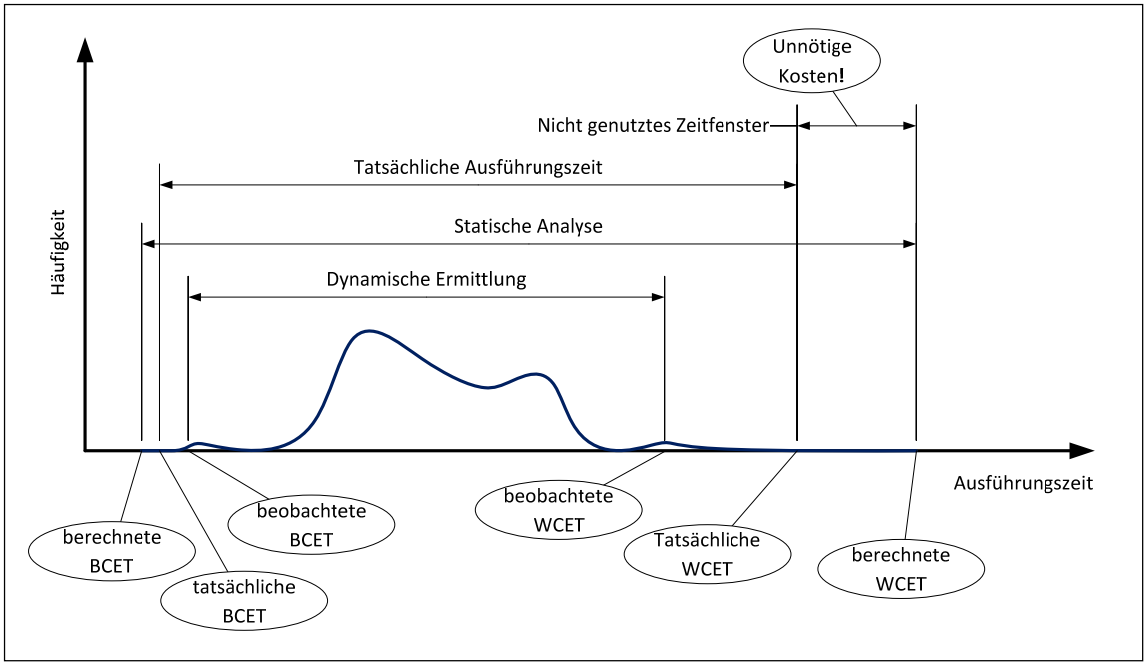


Abbildung 2-26: Statische Analyse und dynamische Ermittlung von Programmlaufzeiten

Die tatsächlichen Ausführungszeiten der einzelnen Pfade sind nicht ohne weiteres bestimmbar, da sie den im Abschnitt 2.3.1 diskutierten Einflussfaktoren unterliegen.

Falls eine statische Analyse möglich ist, kann diese Informationen über die bestmöglichen (BCET - *best case execution time)* und schlechtestmöglichen (WCET - *worst case execution time*) Ausführungszeiten liefern (Abbildung 2-26, nach [57]). Aussagen über die statistische Verteilung von Ausführungszeiten sind mit einer statischen Analyse nur sehr eingeschränkt möglich.

Statische Analysen arbeiten meist mit Modellen, die eine vereinfachte Abbildung der Vorgänge in einem SoC beinhalten. So wurde beispielsweise bei ersten Versionen von WCET-Analyse-Werkzeugen der Cache nicht mit berücksichtigt. Jede Vereinfachung muss pessimistisch ausgelegt sein, d.h. in vielen Fällen liefern die Modelle längere Ausführungszeiten. Diese Lücke verursacht unnötige Kosten, da das System entsprechend den berechneten Ausführungszeiten ausgelegt sein muss.

Eine ausführliche Diskussion von Verfahren zur statischen Analyse zur Ermittlung der WCET ist in [57] zu finden. Kommerzielle Werkzeuge zur statischen Analyse sind z.B. aiT (AbsInt), Bount-T (Tidorum) oder RapiTime (Rapita Systems).

Für die Definition von Anforderungen an die Beobachtbarkeit von SoCs ist die im folgenden Abschnitt diskutierte dynamische Ermittlung von Ausführungszeiten relevant.

2.3.3 Messung / Dynamische Ermittlung der Ausführungszeit

Neben der statischen Analyse kann eine dynamische Ermittlung von Ausführungszeiten durchgeführt werden. Hier werden die Ausführungszeiten von Programmcode mit Hilfe eines Simulators oder durch Messungen an realer Hardware ermittelt.

Die Messung auf der realen Hardware hat den Vorteil, dass kein präzises Modell des Zielsystems erforderlich sein muss und sehr schnell erste Ergebnisse erzielbar sind. Besonders für MPSoCs, bei denen eine Modellierung sehr kompliziert ist, liefert die Messung der tatsächlichen Ausführungszeiten wertvolle Anhaltspunkte zum Verständnis und Auslegung des Systems.

Die Aussagekraft der Messung kann gesteigert werden, indem nicht nur eine Messung vom Anfang bis zum Ende eines Programmabschnitts erfolgt, sondern die Dauer einzelner Code-Abschnitte, vorteilhafterweise die Dauer der Ausführung der einzelnen Kanten des Kontrollflussgraphen ermittelt wird. Bei hinreichend großer Pfadabdeckung und den ermittelten maximalen Ausführungszeiten der jeweiligen Kanten kann nun eine deutlich genauere maximale Ausführungszeit bestimmt werden.

Mit der Messung der Ausführungszeiten ist es nicht möglich, zuverlässig die tatsächliche WCET zu ermitteln, da es aufgrund der schon bei der statischen Analyse störenden Einflussfaktoren (siehe Abschnitt 2.3.1) auch bei einer sehr großen Anzahl von Programmdurchläufen nicht sicher gelingt, die maximale Ausführungszeit zu beobachten. Die dynamische Analyse liefert aber wertvolle Hinweise bezüglich der Häufigkeitsverteilung der Laufzeit von Routinen, welche wiederum für die Dimensionierung der Hardware und die Einstellungen des Betriebssystems sehr hilfreich sind. Da sich die Ausführungszeiten verschiedener Programmteile summieren, kann mit statistischen Verfahren die Wahrscheinlichkeit von Zeitüberschreitungen bestimmt werden. Bei Systemen oder Programmteilen, bei denen die Einhaltung von Laufzeiten nicht zwingend erforderlich ist (z.B. Benutzer-Interface), kann somit die reservierte Ausführungszeit verkürzt werden und somit das System entsprechend kleiner (und damit kostensparender) ausgelegt werden.

Weiterhin können die Ergebnisse der dynamischen Ermittlung der Ausführungszeiten zur Verifizierung der Ergebnisse der statischen Analyse verwendet werden. Damit kann der Nachweis der Zuverlässigkeit (siehe auch Abschnitt 3.10) des verwendeten statischen Analysewerkzeugs unterstützt werden. Bei der Werkzeugqualifizierung gemäß ISO 26262:2011 [35] (Funktionale Sicherheit von Straßenfahrzeugen – Teil 8: Unterstützende Prozesse) kann die Dokumentation der dynamischen Messung beispielsweise eines „*Tool Qualification Kits*“ sein. Gemessene Ausführungszeiten dürfen niemals größer sein als die durch die statische Analyse ermittelten maximalen Zeiten. Werden hier Abweichungen entdeckt, ist von einem fehlerhaften Modell bzw. der fehlerhaften manuellen Ermittlung von dynamisch bekannt werdenden Sprungzielen auszugehen.

Um die in der statischen Analyse angelegten Listen der erst zur Laufzeit des Programms bekannt werdenden Sprungziele zu verifizieren, können diese mit den während der dynamischen Messung auftretenden Sprüngen verglichen werden.

Als vorteilhaft hat sich ein „hybrider" Ansatz erwiesen, bei dem die statische Analyse mit der Messung von Ausführungszeiten auf der Zielhardware kombiniert wird [60].

2.3.4 Anforderungen an den Beobachter

Zur dynamischen Ermittlung von Ausführungszeiten sollte ein idealer Beobachter über folgende Eigenschaften verfügen (Tabelle 2-14):

- Beobachtung einer beliebigen Anzahl von Basisblöcken (Kanten des Kontrollflussgraphen) sowie der Historie („H") des Programmlaufs, um eventuelle Cache-Effekte (z.B. erstmalige und wiederholte Schleifendurchläufe) mit berücksichtigen zu können
- Protokollierung der Ausführungszeiten über eine beliebig lange Zeitdauer
- Protokollierung der Ausführungszeiten mit und ohne Unterbrechungen (z.B. Interrupts)
- Protokollierung von Sprungzielen

Wichtig ist die gleichzeitige Erfüllung dieser Anforderungen. So reduziert sich die Aussage einer Messung deutlich, wenn beliebig viele Basisblöcke nur über sehr kurze Zeit beobachtbar sind bzw. wenn bei einer längeren Beobachtung die Zahl der beobachtbaren Basisblöcke begrenzt ist und somit nicht alle Pfade des Kontrollflussgraphen erfasst werden können.

Zur Ermittlung der Ausführungszeiten ist die Kenntnis der ausgeführten Instruktionen nicht ausreichend, vielmehr ist es erforderlich, die genaue Ausführungszeit messen zu können (z.B. mittels zyklusgenauem Trace).

		WCET-Messung	
		Einfach	Hybrid
Vollständigkeit der Beobachtung	Funktionen	✓	
	Basic Blocks / Instruktionen / Sprünge		✓ (H)
	Zyklusgenauigkeit	✓	✓
	Ereignisse	✓	✓

Tabelle 2-14: Anforderungen an den Beobachter zur Messung der WCET

2.4 Parallelitätsfehler

Parallelitätsfehler (*concurrency bugs*) können entstehen, wenn die Abarbeitung von Software in parallel ausgeführten Segmenten stattfindet. Dabei können Konstellationen auftreten, bei denen der Ausgang eines Prozesses eine kritische Abhängigkeit von weiteren Ereignissen hat [61] und fehlerhaft entworfene Software an dieser Stelle zu einem oftmals nichtdeterministisch erscheinenden Fehlverhalten (Mandelbugs) führt. Dies ist beispielsweise der Fall, wenn Anwendungen mit Betriebssystemen arbeiten und einzelne Programmsegmente nebenläufig ausgeführt werden. Außerdem können Anwendungen auch noch durch Interrupts unterbrochen werden. Die durch das Betriebssystem oder Interrupts kontrollierte Parallelität geht in Systemen mit mehreren parallel arbeitenden Verarbeitungseinheiten (Multiprozessor SoC - MPSoCs) in eine „echte“ Parallelität über, die Abarbeitung des Programmcodes erfolgt nun tatsächlich gleichzeitig.

In Abbildung 2-27 sind zwei Programmläufe für ein ähnliches zu dem in Abbildung 2-3 angegebenen Szenario dargestellt. Der Unterschied besteht darin, dass nun die Zustände des Systems durch parallel und unabhängig voneinander ausgeführte Verarbeitungsschritte beeinflusst werden. Damit ist das Verhalten des Systems nicht mehr deterministisch, der in beiden Programmläufen aktivierte Defekte führt nur in Programmlauf 1 zu einem beobachtbarem Fehlverhalten – in Programmlauf 2 scheint alles korrekt zu funktionieren. Dieser Nichtdeterminismus stellt an die Software-Entwicklung, das Testen und das Debuggen von MPSoCs sehr hohe Anforderungen und macht teilweise völlig neue Lösungsansätze erforderlich.

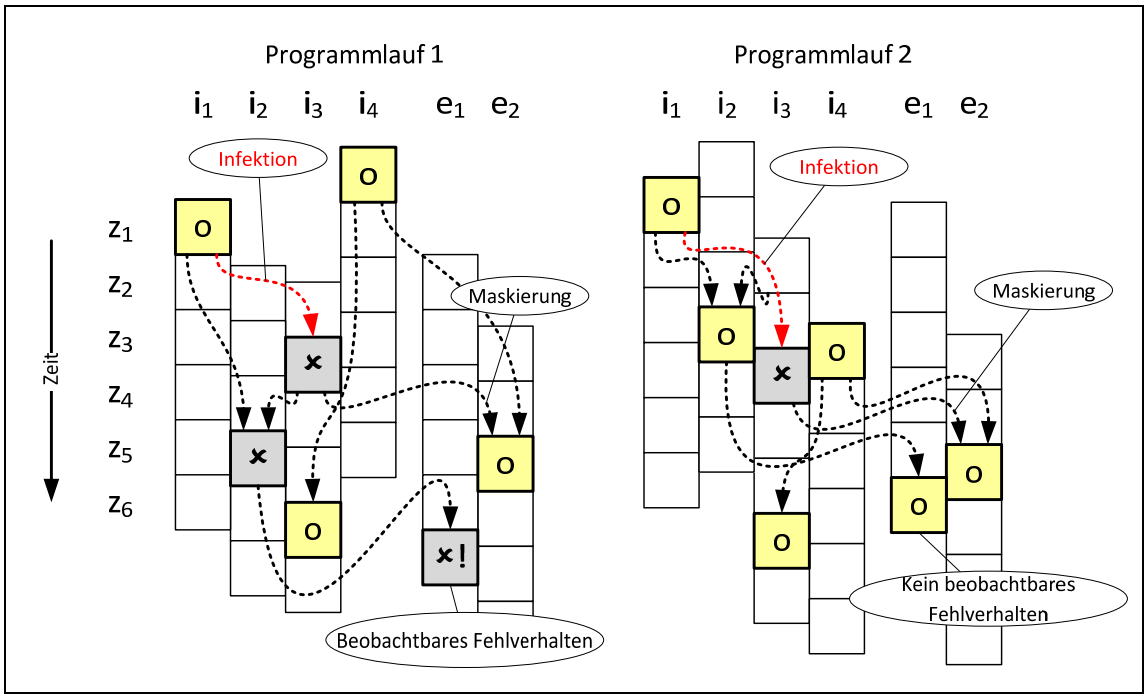


Abbildung 2-27: Unterschiedliche beobachtbare Auswirkungen eines aktivierten Defekts bei nebenläufigen Programmen[15]

Auch bei der Verwendung von vorhandenem Programmcode (*legacy code*), welcher ursprünglich für Singleprozessor-SoCs (SPSoCs) entwickelt wurde, auf MPSoCs kann es zu großen Schwierigkei-

[15] Komprimierte Darstellung entsprechend Abbildung 2-4

ten kommen. Einmal können nach der Konsolidierung Defekte aktiviert werden, die auf dem ursprünglichen SPSoCs nicht in Erscheinung traten. Andererseits muss die Softwarearchitektur überarbeitet werden, da die oftmals bei SPSoCs verwendeten Steuerungsmechanismen zur Synchronisation einzelner Prozesse (z.B. kurzzeitiges Abschalten von Interrupts) bei MPSoCs nicht ohne weiteres übertragbar sind.

Wenn Anwendungen nicht speziell für MPSoCs neu programmiert werden, sondern *legacy code* wieder verwendet wird, müssen mit Unterstützung geeigneter Werkzeuge möglichst alle potentiellen Wettlaufsituationen erkannt und bei der Anpassung der Software berücksichtigt werden.

Parallelitätsfehler haben einen großen Einfluss auf die Stabilität eines Systems. In [62] wurden die Konsequenzen derartiger Defekte untersucht. Dabei führte ca. ein Drittel der Defekte zu einem Absturz des Systems, ein weiteres Drittel brachte das System zum Stillstand. Bei einem Großteil der untersuchten Defekte (96%) waren nur zwei Threads an der Aktivierung des Defekts beteiligt. Die Studie beruht auf die Auswertung von Fehlerprotokollen von Anwendungen (MySQL, Apache, Mozilla, Open Office), die üblicherweise nicht auf SoCs (und speziell auf MPSoCs) laufen und keine Echtzeitanforderungen erfüllen müssen. Daher ist die Übertragung der Ergebnisse der Studie auf Anwendungen, die harten Echtzeitkriterien unterliegen, nicht vollständig möglich.

Übliche Parallelitätsfehler wie Wettlaufsituationen, Verklemmungen und Verhungern werden im Folgenden erläutert.

2.4.1 Wettlaufsituationen

Von einer Wettlaufsituation (*race condition*) spricht man, wenn mehrere Prozesse auf dieselbe Ressource (z.B. eine Variable oder eine Peripherieeinheit) zugreifen und mindestens einer der Zugriffe den internen Zustand der Ressource verändert (z.B. Schreibzugriff auf eine Variable, Lesen eines FIFOs).

In den Jahren 1985 bis 1987 kostete ein Softwaredefekt in dem Strahlentherapiegerät Therac-25 mindestens drei Menschen das Leben und führte zu schweren Verletzungen weiterer Menschen[63]. Der Defekt führte zu einem *data race* zwischen zwei Prozessen, was wiederum eine Überdosis der abgegebenen Strahlung zur Folge hatte.

Ein Beispiel für eine typische Wettlaufsituation beim Zugriff auf eine Variable (*data race*) ist in Abbildung 2-28 angegeben. Thread A und Thread B sollen bei jedem Durchlauf einen gemeinsam benutzten Zähler x inkrementieren. Durch das Auftreten von Wettlaufsituationen kommt es in den Fällen 1 und 4 zu inkorrekten Ergebnissen, der Wert von x ist vom der zeitlichen Reihenfolge der Abarbeitung von Instruktionen zweier nebenläufiger Threads abhängig.

Wenn es sich bei einer Ressource um eine Peripherieeinheit handelt, muss berücksichtigt werden, dass eventuell auch ein Lesezugriff auf eine Peripheriereinheit deren Status ändern kann und folglich wie ein Schreibzugriff zu behandeln ist. Beispiel ist ein gepufferter Nachrichtenspeicher (FIFO) einer seriellen Schnittstelle. Mit jedem Lesezugriff wird ein Teil der empfangenen Nachricht ausgegeben. Wenn ein Thread konkurrierend mit einem anderen Thread den FIFO ausliest, enthalten die im jeweiligen Thread empfangenen Daten Lücken und sind somit fehlerhaft.

Data Races lassen sich in verschiedene Klassen unterteilen.

Eine „***Atomicity violation***“ tritt auf, wenn Anweisungen atomar ausgeführt werden müssen (*critical section*), diese Ausführung aber gestört wird. Ein Beispiel für eine a*tomicity violation* ist in Abbildung 2-28 angegeben, wenn die Threads A oder B unterbrochen werden, führt dies zu falschen Zählergebnissen.

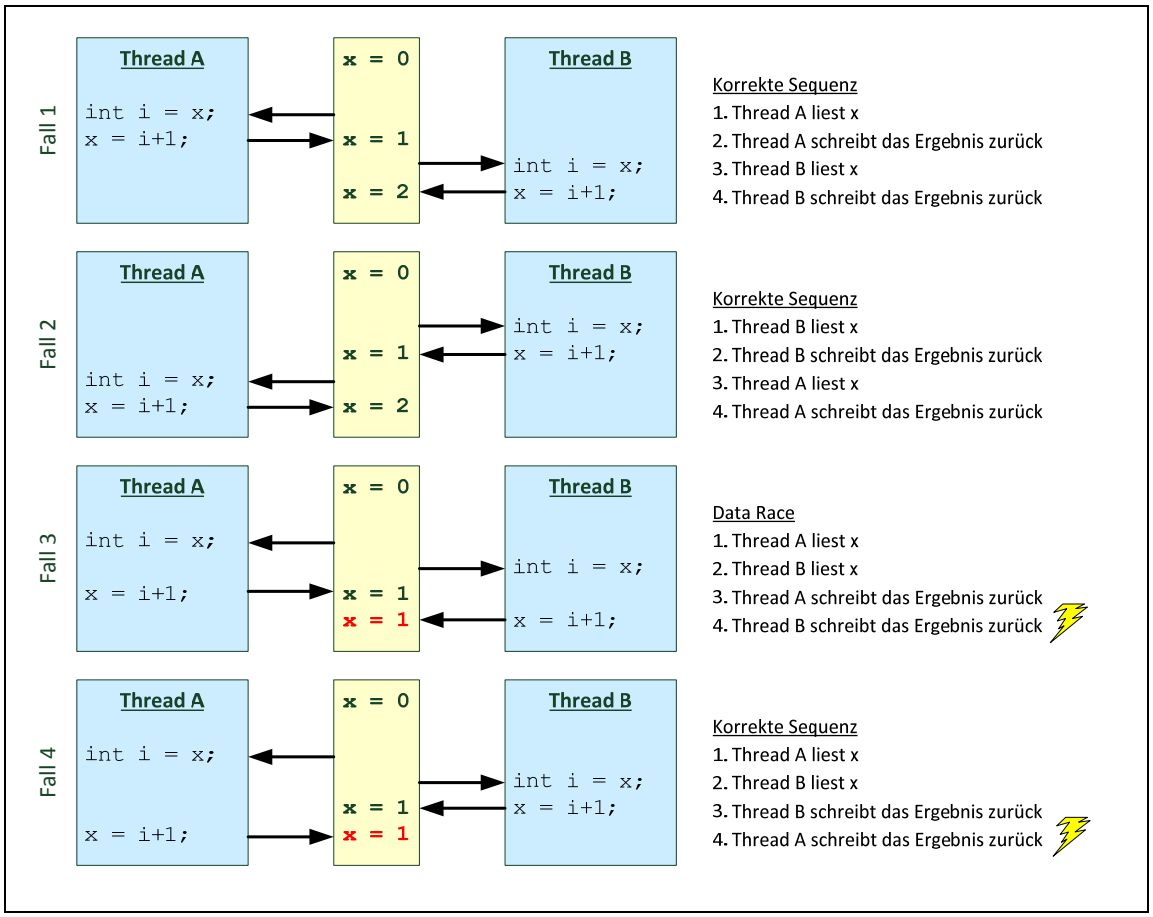


Abbildung 2-28: Beispiele für *Data Races*

Eine korrekte Implementierung der Funktion kann mit Hilfe eines Locks über dem atomar abzuarbeitenden Bereich implementiert werden (Abbildung 2-29).

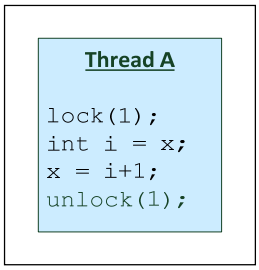


Abbildung 2-29: Vermeidung von *atomicity violations* durch Locks

Eine weitere Variante von *Data Races* sind "***Ordering violations***". Sie treten auf, wenn Programmteile nicht in der Reihenfolge abgearbeitet werden, wie sie der Programmierer beabsichtigt hat. In Abbildung 2-30 ist ein entsprechendes Beispiel angegeben, in dem Thread B auf die noch nicht initialisierte Variable `job` zugreift.

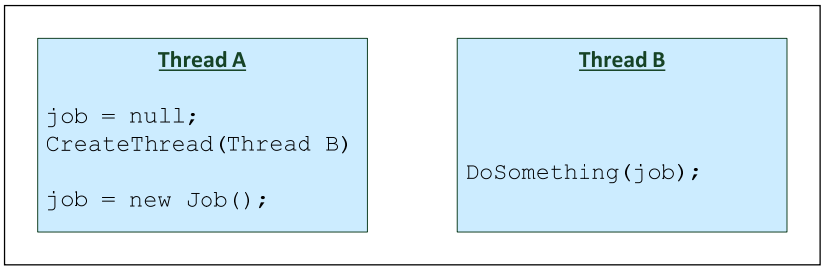


Abbildung 2-30: Beispiel für *ordering violations*

Beim "***Unintended sharing***" teilen sich Threads ungewollt die gleichen Objekte. Im Beispiel in Abbildung 2-31 verwenden beide Threads die gleiche Funktion: Die mit dem Schlüsselwort `static` deklarierte Variable `i` liegt für beide Threads auf dem gleichen Speicherplatz.

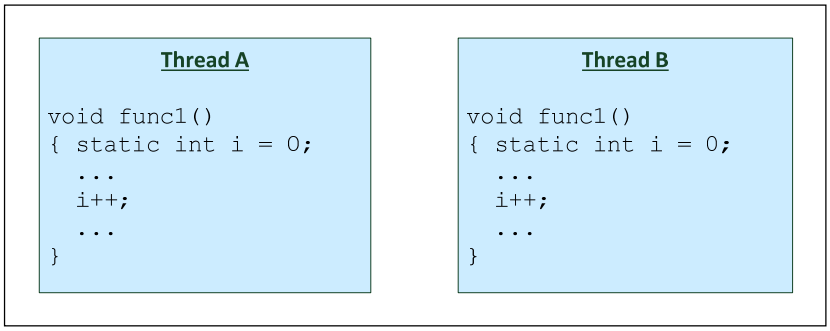


Abbildung 2-31: Beispiel für *unintended sharing*

Es existieren auch Situationen, in denen Wettlaufsituationen vom Programmierer beabsichtigt sind (*benign data race*). Beispielsweise kann ein Thread anderen Threads durch Setzen eines „`done`"-Flags mitteilen, dass eine Aufgabe fertiggestellt wurde (Abbildung 2-32). Andere Threads prüfen dies durch kontinuierliches Abfragen des „`done`"-Flags. Abgesehen davon, dass ein entsprechendes Betriebssystem diese Synchronisationsaufgabe sehr viel eleganter lösen kann, soll dieses Beispiel verdeutlichen, dass nicht jede erkannte Wettlaufsituation auch tatsächlich einen Defekt darstellt.

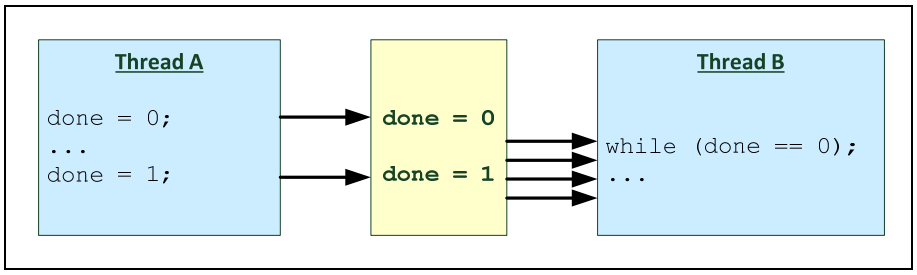


Abbildung 2-32: Beispiel einer beabsichtigten Wettlaufsituation

Weiterhin können auch falsch-positive *Data Races* erkannt werden, wenn optimierende Compiler verwendet werden. Ein Beispiel ist ein optimierter Speicherzugriff, der einen erweiterten Bereich spekulativ liest.

2.4.2 Verklemmungen

Eine Verklemmung (*Deadlock*) entsteht, wenn mehrere Threads aufeinander warten, d.h. wenn jeder Thread auf die Freigabe einer Ressource wartet, die nur von einem anderen Thread aus der Menge freigegeben werden kann (Abbildung 2-33).

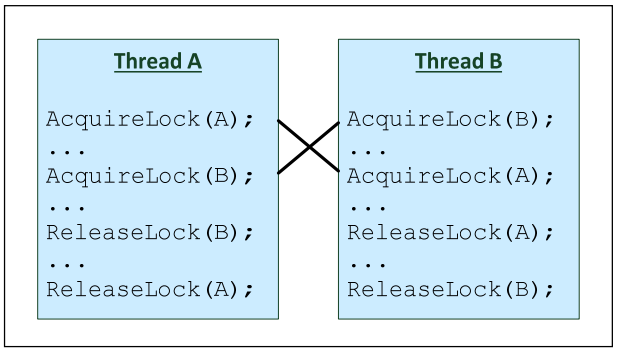


Abbildung 2-33: Beispiel einer Verklemmung

Eine Sonderform von Verklemmungen stellen *Live Locks* dar. Diese treten auf, wenn sich Threads durch die Reaktion auf die Statusänderung des jeweils anderen Threads gegenseitig blockieren. Ein oft verwendetes Beispiel für einen Live Lock ist die Situation, bei der sich zwei Personen in einem engen Gang begegnen und einander jeweils in die gleiche Richtung auszuweichen versuchen, so dass sie niemals aneinander vorbei kommen. *Live Locks* sind ein Problem des Betriebssystems und nur sehr schwer analysierbar.

2.4.3 Verhungern

Im Gegensatz zu einer Verklemmung blockiert beim Verhungern (*Starvation*) ein hochpriorer Thread einen niederprioren Thread. Es besteht keine Abhängigkeit zwischen den Threads, d.h. der hochpriore Thread muss nicht auf die Freigabe einer Ressource durch den niederprioren Thread warten. Vielmehr besteht ein Planungsfehler, der einem (oder mehreren) hochprioren Threads so viel Zeit einräumt, dass dem niederprioren Thread in Folge keine Zeit mehr zugeteilt wird.

2.4.4 Analyse von Wettlaufsituationen

Wettlaufsituationen können sowohl mit statischen Verfahren als auch mit dynamischen Messungen analysiert werden.

Statische Analyse

Die statische Analyse basiert auf der Erstellung und Auswertung von Kontrollfluss- sowie Parallel-Execution-Graphen. Dabei werden Variablen ermittelt, die von mehreren nebenläufigen Prozessen gemeinsam genutzt werden und bei denen von mindestens einem Prozess ein schreibender Zugriff stattfindet. Als nächstes wird geprüft, ob die Zugriffe auf die gefundenen Variablen mit mindestens einem Lock geschützt sind. Dazu sind umfangreiche Annotationen im Code erforderlich, um Beziehungen zwischen Variablen und den zugehörigen Locks herstellen zu können. Kann kein Lock gefunden werden, so ist dies ein Hinweis auf eine mögliche Wettlaufsituation. Mit verschiedenen Methoden wird versucht, die gefundenen potentiellen Wettlaufsituationen zu klassifizieren und mögliche Fehlalarme zu eliminieren.

Vorteil der statischen Analyse ist die Vollständigkeit, d.h. es werden auch Defekte gefunden, die bei dynamischen Tests aufgrund der unvollständigen Testabdeckung nicht erkannt werden würden. Im Gegensatz zur dynamischen Analyse (nach dem gegenwärtigen Stand der Technik) erfordert die statische Analyse keinen Overhead durch Instrumentierung des Codes und eignet sich damit auch zur Analyse zeitkritischer Code-Segmente.

Nachteil der statischen Analyse ist der damit verbundene hohe Aufwand und sehr viele falsch-positive Ergebnisse. Durch Fehler in der Annotation sowie der aufwändigen, teilweise manuellen Begutachtung eventueller Fehlalarme besteht das Risiko, potenzielle Wettlaufsituationen zu übersehen.

Ein Beispiel für ein statisches Analysewerkzeug für Wettlaufsituationen ist „RELAY“[64].

Dynamische Analyse

Die dynamische Analyse erfolgt durch Beobachtung der ausgeführten Threads. Üblicherweise ist dazu eine Instrumentierung des Codes erforderlich, der die Ausführungszeit deutlich verlangsamt und damit besonders bei zeitkritischen Anwendungen eine Limitierung darstellt. Außerdem ist für die Analyse zusätzlicher Speicher erforderlich und es kann Konflikte mit gleichzeitig implementierten Überdeckungstests geben.

Dynamische Analysen von *data races* können mit den im Folgenden vorgestellten „*Eraser*“ – oder „*Happened-before*“-Algorithmen durchgeführt werden.

Eraser-Algorithmus

Der *Eraser*-Algorithmus [65] (Abbildung 2-34) ermittelt auf einfache Art und Weise für jede Variable, ob eine mögliche Wettlaufsituation stattgefunden hat. Dazu wird für jede Variable eine State Maschine implementiert, die aus vier Zuständen besteht. Initial wird der Zustand „*Virgin*“ angesprungen. Sobald ein Schreibzugriff auf die Variable stattfindet, wird die ID `t` des schreibenden Threads zwischengespeichert und der Zustand „*Exclusive-Modified (t)“* angesprungen. Dieser Zustand wird verlassen, wenn ein anderer Thread auf die Variable lesend oder schreibend zugreift.

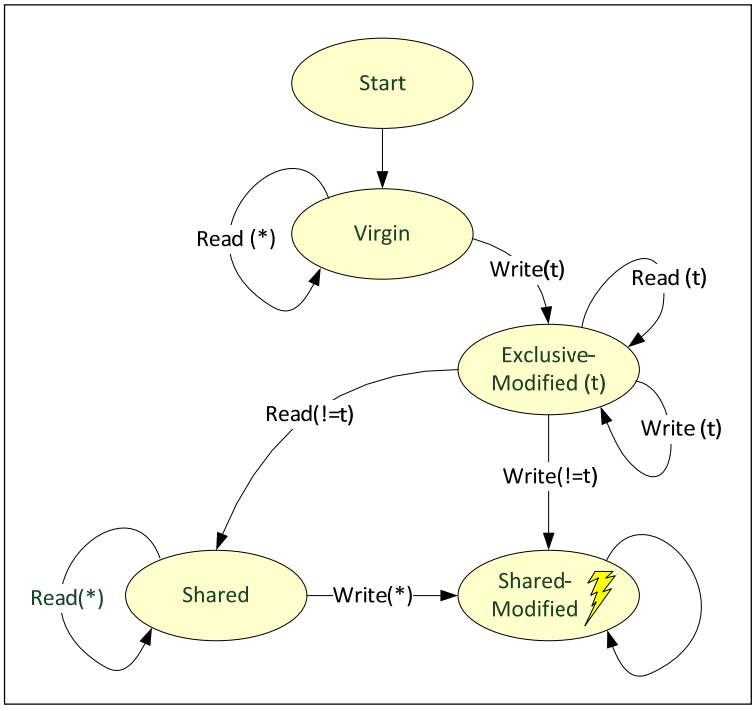


Abbildung 2-34: *Eraser* Algorithmus

Ein Schreibzugriff eines anderen Threads deutet auf eine potentielle Wettlaufsituation hin. Wenn die State Maschine am Ende des Tests den Zustand „*Shared-Modified*“ erreicht hat, so ist dies ein Hinweis auf eine potentielle Wettlaufsituation. Allerdings kann der *Eraser*-Algorithmus sehr hohe falsch-positive Fehlerdetektionsraten aufweisen. Die in Abbildung 2-32 dargestellte beabsichtigte Wettlaufsituation würde mit dem *Eraser*-Algorithmus beispielsweise zu einem falsch-positivem Ergebnis führen. Thread A setzt initial „`done = 0`“, damit wird in den Zustand „*Exclusive-Modified*

(t)“ gesprungen. Das Lesen durch Thread B bewirkt einen Zustandswechsel in „*Shared“*, das Setzen von „`done = 1`“ durch Thread A führt zu einem Wechsel in den Zustand „*Shared-Modified*“ und damit zu einer falsch-positive Fehlerdetektion.

„Happens-before“-Algorithmus

Um die Anzahl der falsch-positiven Ergebnisse des Eraser-Algorithmus zu reduzieren, muss die zeitliche Abfolge der Lese- und Schreibzugriffe einzelner Threads berücksichtigt werden. Dieses Vorgehen wird auch als „*Happens-before*“- Algorithmus [66] bezeichnet.

Es wird davon ausgegangen, dass nach einem Lese-Schreib-Zyklus eines Threads ein weiterer Lese-Schreib-Zyklus eines anderen Threads zulässig ist (Fall 3 und 4 in Abbildung 2-28). Dieser zulässige Zyklus wird aber im *Eraser*-Algorithmus schon als potentielle *race condition* interpretiert.

Der Algorithmus muss folgende Bedingungen erfüllen:

Sollte ein Thread (t) schon einmal einen Lese-oder Schreibzugriff auf eine Variable vorgenommen haben und zwischenzeitlich ein anderer Thread (!=t) einen Schreibzugriff unternommen haben, so muss der Thread (t) vor einem Schreibzugriff einen Lesezugriff ausführen, der nach dem letzten Schreibzugriff eines anderen Threads liegt.

Die Schwierigkeit bei der Implementierung des „*Happens-before*“- Algorithmus in MPSoCs besteht darin, dass die Abläufe der einzelnen CPUs zueinander in Beziehung gesetzt werden müssen.

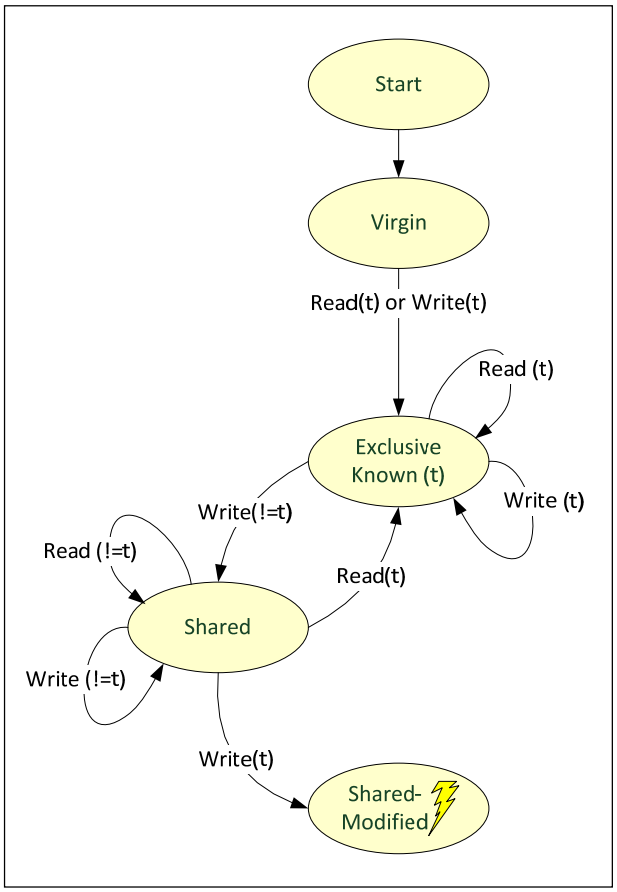


Abbildung 2-35: „*Happens before*“ Algorithmus

Anhand der in Abbildung 2-32 dargestellten beabsichtigten Wettlaufsituation (welche vom *Eraser*-Algorithmus als Wettlaufsituation erkannt wurde) zeigt sich der Vorteil des „*Happens-before*“- Algorithmus. So lang Thread B nur lesend auf `done` zugreift (was im Beispiel der Fall ist), wird keine Wettlaufsituation detektiert.

2.4.5 Anforderungen an den Beobachter

Die dynamische Analyse von Parallelitätsfehlern, speziell von *race conditions* stellt an die Beobachtbarkeit von SoCs, (insbesondere MPSoCs) sehr hohe Anforderungen (Tabelle 2-15).

Einmal müssen die Zugriffe der einzelnen CPUs bzw. Threads auf die gemeinsam genutzten Variablen (Adressen, Zugriffsart, zugreifender Thread) beobachtbar sein, zusätzlich müssen die Abläufe in den einzelnen CPU-Kernen in eine zeitliche Beziehung gesetzt werden (*vector clock*).

Von Vorteil ist auch, wenn der Beobachter in der Lage ist, die einzelnen Locks zu verfolgen und mit den beobachteten Zugriffen auf die gemeinsam genutzten Variablen in Beziehung zu setzen.

Eine große Einschränkung stellt die für die dynamische Analyse aktuell verwendete Instrumentierung dar, die das Code-Volumen und dadurch die Ausführungszeit deutlich aufbläht und somit die Beobachtung zeitkritischer Sequenzen nur eingeschränkt zulässt. Wünschenswert wäre hier die Beobachtbarkeit ohne Instrumentierung.

		Data Race	Verklemmungen	Verhungern
Vollständigkeit der Beobachtung	Taskwechsel	✓	✓	✓
	Instruktionen	✓	✓	
	Datenzugriffe (CPU)	✓	✓	
	Datenzugriffe (Peripherie)	✓		
	Zyklusgenauigkeit	✓	✓	
	Ereignisse	✓	✓	
Beobachtung von MPSoCs		✓	✓	✓

Tabelle 2-15: Anforderungen an den Beobachter bei der Analyse von Parallelitätsfehlern

2.5 Beobachtung von Variablen

2.5.1 Ablageorte für Variablen

Globale Variable sowie flüchtige (`volatile`) oder statische (`static`) lokale Variable werden an einem vorbestimmten Ort im Hauptspeicher abgelegt. Ein häufig vorkommender Fehler (besonders in C und C++-Programmen) ist das Auftreten von Null-Pointern. Hier wird ein nicht initialisierter Pointer (der meistens auf die Adresse „0x00“ zeigt) für Speicherzugriffe verwendet. Es kann bei der Fehlersuche sehr hilfreich sein, wenn diese Zugriffe protokolliert werden.

Die Beobachtung von lokalen Variablen ist schwieriger. Nicht flüchtige lokale Variable können sich – zu unterschiedlichen Zeiten - an unterschiedlichen Stellen im Speicher des SoCs befinden. Ist die mit der lokalen Variablen verbundene Funktion aktiv, so kann sich die Variable in einem Register der CPU (Registervariable) oder im lokalen RAM befinden. Wird die Funktion verlassen, so wird die Variable auf dem Stack gesichert oder sie verbleibt in dem CPU-Register, wobei die aktuell gültige Registerbank umgeschaltet wird.

2.5.2 Anforderungen an den Beobachter

Die Beobachtung globaler Variablen und statischer sowie volatiler lokaler Variablen ist relativ einfach, es müssen nur die jeweiligen Schreib- und Lesezugriffe auf die mit den Debug-Informationen bekannt gemachten Adressbereiche protokolliert werden.

Zur Beobachtung von lokalen Variablen, die während der Gültigkeit der aktuellen Funktion vom Stack auf den Heap kopiert werden, ist die Kenntnis der aktuell ausgeführten Funktion sowie der Schreibzugriffe auf den Heap erforderlich. Zur Beobachtung von Registervariablen ist die Beobachtung der CPU-Register erforderlich, die vom Compiler zur Ablage von Variablen verwendet werden können.

In Tabelle 2-16 sind die Anforderungen an den Beobachter bei der Überwachung von Variablen aufgeführt.

		Beobachtung von Variablen
Vollständigkeit der Beobachtung	Taskwechsel	✓
	Funktionen	✓
	Basic Blocks / Instruktionen / Sprünge	✓
	Datenzugriffe (CPU)	✓
	Datenzugriffe (Peripherie)	✓
	CPU-Register	✓
	Cache	✓
	Ereignisse	✓
Beobachtung von MPSoCs		✓

Tabelle 2-16: Anforderungen an den Beobachter bei Überwachung von Variablen

2.6 Bestimmung des Ressourcenverbrauchs

Für die Durchführung von Ressourcentests sowie für die Zertifizierung sicherheitskritischer Systeme ist die Analyse der Nutzung von Stack und Heap sowie der CPU-Last und der Ausführungszeiten von Programmen notwendig.

2.6.1 Stack und Heap

Ein Überlauf (*overflow*) von Stack und Heap (zur Laufzeit des Programms zugeteilter Speicher) erfolgt, wenn zu viele Daten in einen zu kleinen reservierten Speicherbereich geschrieben werden und damit der Inhalt von Speicherstellen hinter dem Puffer verändert wird. Dies kann als Folge einer falschen Dimensionierung des Speicherlayouts bzw. der Verwendung unsicherer Programmstrukturen (z.B. infolge unbegrenzter Rekursionen) geschehen.

Beinhalten die betroffenen Speicherstellen für die Ausführung des Programms relevante Informationen, so tritt beim Zugriff auf diese Informationen eine ungewollte Änderung des Programmablaufs auf. Andererseits kann auch der Inhalt des Stacks korrumpiert werden, wenn dieser sich im falschen Bereich befindet. Das Programm greift nun auf den ursprünglich z.B. für die Ablage von Daten vorgesehenen Speicherbereich schreibend zu und überschreibt so den Inhalt des Stacks. Auf diese Weise können lokale Variablen oder abgelegte Rücksprungadressen ungewollt überschrieben werden. Überläufe treten besonders häufig bei der Verwendung „unsicherer" Programmiersprachen wie „C" auf, die aber gerade aufgrund ihrer Effizienz für die Erstellung von Programmen in eingebetteten Systemen oft verwendet werden. Beispielsweise können hier Strings als `char`-Arrays realisiert werden, viele C-Funktionen (z.B. `strcpy()`) kopieren Daten ohne Überprüfung der Länge.

Ein Stack- oder Heap-Überlauf kann, aber muss nicht zu einer Infektion führen (siehe Abschnitt 2.1).

Es werden zwei Szenarien von Stack- und Heap- Überlauf unterschieden: Einmal stellt der unbeabsichtigte Überlauf des Stacks eine Gefährdung der Stabilität des Systems dar, zum anderen kann ein gezielt herbeigeführter Stack-Überlauf zur Einschleusung fremden Codes und damit zur Übernahme der Systemkontrolle durch externe Instanzen führen.

Die durch Eingaben gezielte Herbeiführung eines Überlaufs des Stacks kann mittels geeigneter Mechanismen, wie z.B. der konsequenten Prüfungen der eingehenden Daten verhindert werden. Da in vielen Systemen diese Schutzvorkehrungen nicht vollständig implementiert sind, stellt aktuell die bewusste Herbeiführung eines Stack-Überlaufs ein beliebtes Einfallstor für Schadsoftware dar.

Grundlage einer Stack-Analyse [67] ist – wie bei der Analyse der Ausführungszeit – der Kontrollflussgraph. Mit dessen Hilfe wird bestimmt, an welchen Stellen des Programms ein Zugriff auf den Stack erfolgen kann (*call graph*) und wie groß im ungünstigsten Fall der Stack sein muss, um nicht überzulaufen.

Ein zu kleiner Stack kann sehr schwer zu findende Laufzeitfehler verursachen. Wird der Stack zu groß ausgelegt, so werden Ressourcen verschwendet, d.h. das System wird teurer als notwendig.

Diese statische Analyse kann wiederum durch Beobachtung des realen Systems abgesichert werden. Dazu kann der für den Stack reservierte Speicher mit einem Muster beschrieben werden. Während des Programmlaufs wird periodisch geprüft, bis zu welcher Adresse das Muster überschrieben wurde. Eine zuverlässigere Methode ist die Überwachung des Stackpointers, dieser wird zur Laufzeit des Programms verfolgt und auf das Überschreiten von Grenzwerten hin überprüft. Üblicherweise wird für den Stackpointer ein CPU-Register verwendet, dessen Veränderungen normalerweise nicht im Trace verfügbar sind. In diesem Fall lässt sich auch durch die Beobachtung von Schreibzugriffen auf den für den Stack reservierten Speicherbereich sowie den angrenzenden Bereich erkennen, ob und wann ein eventueller Überlauf des Stacks stattfindet.

Ein ähnliches Verfahren kann auch zur Überwachung des Heaps angewandt werden (*memory leak analysis*).

Ein Beispiel für ein Werkzeug zur Stack- und Heap-Analyse ist das Open-Source-Programm Valgrind.

2.6.2 CPU-Last / Ausführungszeiten

Bei der Messung der CPU-Last wird die Zeit gemessen, die eine CPU zur Abarbeitung eines Code-Abschnitts benötigt. Dazu wird ein Timer zu Beginn der Messperiode gestartet und bei deren Verlassen gestoppt. Alternativ kann auch die Differenz zwischen am Anfang und Ende ermittelten Zeitstempeln bestimmt werden. Bei der Beobachtung von Prozessen oder Threads sind die während der Beobachtungszeit gemessenen minimalen und maximalen Ausführungszeiten, der Mittelwert sowie der gleitende Mittelwert von Interesse.

Die gemessene CPU-Last ist kein zuverlässiger Indikator für maximale Ausführungszeiten und noch verfügbare Reserven an Rechenzeit, hier liefern die in Abschnitt 2.3 diskutierten Methoden verlässlichere Werte.

2.6.3 Anforderungen an den Beobachter

Idealerweise lässt sich durch Beobachtung der entsprechenden CPU-Register der zeitliche Verlauf eines oder mehrerer Stackpointer beobachten. Ist die direkte Beobachtung der CPU-Register nicht möglich (was in den allmeisten Fällen so ist), so kann auch durch die Beobachtung der Schreibzugriffe der Stackpointer rekonstruiert werden.

Ein idealer Beobachter sollte über folgende Eigenschaften verfügen:

- Beobachtung der CPU-Register (Stackpointer) oder alternativ Beobachtung von Schreibzugriffen auf vordefinierte Speicherbereiche (Adresse „A“, Richtung „R“, Wert „W“)
- Ermittlung der Initiatoren der Stackzugriffe

In Tabelle 2-17 sind die einzelnen Anforderungen an die Beobachtung von Stack und Heap sowie der CPU-Last und von Ausführungszeiten zusammengefasst.

		Stack und Heap	CPU-Last / Ausführungszeiten
Vollständigkeit der Beobachtung	Taskwechsel	✓	✓
	Funktionen / Basic Blocks	✓	✓
	Instruktionen	✓	
	Sprünge	✓	
	Datenzugriffe (CPU)	✓	
		ARW	
	CPU-Register	✓	
	Zyklusgenauigkeit		✓
	Ereignisse		✓
Beobachtung von MPSoCs		✓	✓

Tabelle 2-17: Anforderungen an die Beobachtung von Stack und Heap sowie von CPU-Last und Ausführungszeiten

2.7 Analyse des Caches / Optimierung des Speicherlayouts

Wichtig für die Steigerung der Performance ist eine optimale Ausnutzung des verfügbaren Cache- und des On-Chip-Speichers. Neben einer Erhöhung der Verarbeitungsgeschwindigkeit hilft auch jeder eingesparte Zugriff auf langsame Speicher (internen Flash-Speicher, externer Speicher) beim Einsparen von Energie.

2.7.1 Cache-Analyse

Es wird zwischen Instruktions- und Datencache unterschieden.

Der Instruktionscache bewirkt im Wesentlichen eine Verringerung der Ausführungszeit des abgearbeiteten Codes. Der Datencache verringert die Zeiten für Schreib- und Lesezugriffe auf den Datenspeicher. Ein wichtiges Ziel bei der Beobachtung von Cache-Zugriffen ist die Bestimmung der Veränderung der Ausführungszeit von Instruktionen durch Cache-Hits/Misses.

Ein wesentliches Problem, besonders bei SoCs mit mehreren Busmastern, wie z.B. bei MPSoCs oder bei SoCs mit busmasterfähiger Peripherie, ist die Gewährleistung der Kohärenz der Daten. Hier muss verhindert werden, dass ein Busmaster mit im Cache gehaltenen Daten arbeitet, während ein anderer Busmaster auf dieselben Daten im Hauptspeicher zugreift [68]. Um die Kohärenz zu gewährleisten, gibt es sowohl softwarebasierte als auch hardwarebasierte Ansätze [69]. Für den Entwickler ist es am einfachsten, wenn auf dem SoC bereits Strukturen integriert sind, die die Cache-Kohärenz gewährleisten.

Sind keine Strukturen zur Gewährleistung der Cache-Kohärenz auf dem SoC implementiert, so kann diese auch durch geeignete Software sichergestellt werden. Dieser Ansatz erfordert eine aufwendige Implementierung und ist entsprechend fehleranfällig. Wenn ein Busmaster schreibend auf Daten zugreift, die auch anderswo im SoC in Caches abgelegt sind, so muss dafür gesorgt werden, dass diese Daten in allen Caches als ungültig gekennzeichnet werden. Da dies nicht innerhalb desselben Taktzyklus geschehen kann, erfordert dieser Vorgang eine aufwändige Implementierung, um Wettlaufsituationen zu vermeiden.

2.7.2 Optimierung des Speicherlayouts

Üblicherweise ist der Speicher von SoCs mehrstufig ausgelegt. Neben dem sich auf dem SoC befindlichem RAM (*Scratchpad* - schnell, einfach beschreibbar, teuer) existiert oft noch ein Flash-Speicher zur Ablage von nicht veränderlichen Informationen (langsamer, kostengünstiger). Über ein externes Businterface können noch weitere Speicher (z.B. Flash, SDRAM) angeschlossen werden. Neben dem Geschwindigkeitsvorteil, den der Zugriff auf einen internen Speicher im SoC bietet, muss bei der Planung der Speicherbelegung auch der erhöhte Energiebedarf eines Speicherzugriffs über das externe Businterface berücksichtigt werden. Eine optimale Nutzung des *Scratchpad*-Speichers kann eine erhebliche Erhöhung der Verarbeitungsgeschwindigkeit sowie eine drastische Reduktion des Stromverbrauchs bewirken (Tabelle 2-18, Daten aus [70], Tabelle 3.1).

Häufig verwendete Daten bzw. Code sollten im internen Speicher abgelegt werden, wobei während der Laufzeit von Programmen die Belegung des internen Speichers dynamisch angepasst werden kann. Die wohlüberlegte Verwendung von internem RAM sowie die Optimierung des Cacheverhaltens haben teilweise einen deutlich größeren Einfluss auf die Verarbeitungsgeschwindigkeit und den Energieverbrauch von SoCs als diverse Optimierungen durch den Compiler.

Aufgrund der begrenzten oder nur sehr aufwändigen Beobachtbarkeit des Cacheverhaltens sowie einzelner Zugriffszeiten (zyklusgenauer Trace) wurde in der Vergangenheit in vielen Fällen dieser Tatsache nicht genügend Aufmerksamkeit beigemessen.

Instruk-tion	Speicher für Instruktio-nen	Speicher für Daten	Energie (nJ)	Ausführungszeit (CPU-Zyklen)
LOAD	extern	extern	113	7
	intern	intern	15,5	3
STORE	extern	extern	98,1	6
	intern	intern	11,5	2

Tabelle 2-18: Energieverbrauch und Zugriffszeiten von Speicherzugriffen auf einem SPSoC (ARM7TDMI)

Weiteres Optimierungspotential bieten spekulative Lesevorgänge von externem Speicher. Speziell beim Zugriff auf SDRAMs, deren Speicherstruktur den blockweisen Zugriff auf Datenbereiche viel effizienter ablaufen lässt, als eine Folge wahlfreier Zugriffe, kann durch spekulatives Lesen eine deutliche Leistungssteigerung erreicht werden.

2.7.3 Anforderungen an den Beobachter

Ein idealer Beobachter soll für jeden Speicherzugriff neben Adresse und Wert auch noch ergänzende Informationen liefern können. Dazu gehört einmal die Zeitdauer des Zugriffs, zum anderen Informationen über den Cache. Hier ist es wichtig zu wissen, ob die Daten / Instruktionen im Cache abgelegt waren, welche Daten eventuell aus dem Cache verdrängt wurden und ob bei SoCs mit mehreren Busmastern Maßnahmen zur Herstellung der Cache-Kohärenz erforderlich waren.

Zur optimalen Beobachtung des Cache-Verhaltens sowie der Speichernutzung sollte ein idealer Beobachter folgende Informationen liefern:

- Zyklusgenauer Instruktionstrace (anhand der Zugriffsdauer kann auf den Ort geschlossen werden, von dem die Instruktion sowie ggfs. der Datenwert geholt wurde)
- Adressen der gelesenen und geschriebenen Daten
- Zeitlich hochauflösende Messung der Stromaufnahme des SoCs

Falls in einem MPSoC die Cache-Kohärenz nicht durch geeignete Hardware-Strukturen gewährleistet ist, ist eine möglichst umfassende Beobachtung der einzelnen Caches erforderlich.

		Cache-Analyse	Optimierung des Speicherlayouts
Vollständigkeit der Beobachtung	Taskwechsel	✓	✓
	Funktionen	✓	✓
	Basic Blocks	✓	✓
	Instruktionen	✓	✓
	Sprünge	✓	✓
	Datenzugriffe (CPU)	✓	✓
	Datenzugriffe (Peripherie)	✓	✓
	CPU-Register	✓	✓
	Cache	✓	✓
	Abarbeitung	✓	✓
	Zyklusgenauigkeit	✓	✓
	Bussystem	✓	✓
	Ereignisse	✓	✓
Beobachtung von MPSoCs		✓	✓

Tabelle 2-19: Anforderungen an die Beobachtung des Caches und zur Optimierung des Speicherlayouts

2.8 Optimierung des Energieverbrauchs

Der Energieverbrauch von eingebetteten Systemen wird zunehmend zu einem wichtigen Differenzierungsmerkmal.

Eine markante Anwendung ist hier der Bereich der batteriebetriebenen Geräte, bei denen ein optimierter Energieverbrauch direkten Einfluss auf Gewicht und Größe des Energiespeichers hat. Aber auch bei Geräten, die ständig an das Stromnetz angeschlossen sind, stellt eine möglichst geringe Stromaufnahme ein wichtiges Differenzierungsmerkmal zu Wettbewerbern dar.

Ein weiteres Anwendungsgebiet, bei dem viel Mühe in die Verringerung des Energieverbrauchs investiert wird, sind mobile Systeme, speziell der Automobilbereich. Hier sind nicht unbedingt die Energieverbrauchskosten, sondern die Kosten der Infrastruktur zur Bereitstellung der elektrischen Energie ein bedeutender Kostenfaktor, der im Bereich von ca. 0,50€ bis 1,00€ pro Watt elektrischer Leistung liegt. Dieser resultiert einmal aus den Teilekosten der Infrastruktur (Größe der Lichtmaschine und der Batterie), zum anderen aber auch aus der damit verbundenen Gewichtszunahme des Automobils mit all seinen Auswirkungen auf Kraftstoffverbrauch und CO_2-Emission. Diese Kosten erscheinen erst einmal gering, werden sie aber mit dem Energieeinsparpotential eines einzelnen SoCs (~40% von 100mW bis 1W), mit der Zahl der SoCs in einem Automobil (bis ca. 100) sowie mit der Zahl der ausgelieferten Automobile (Größenordnung 10^7) multipliziert, ergibt sich eine deutliche Motivation für eine möglichst optimale Nutzung aller Potentiale, die ein SoC sowie die entsprechenden Software-Optimierungen bieten können.

Auch bei stationären Geräten führt eine Optimierung des Energieverbrauchs neben dem Kostenvorteil der eingesparten Energie zu einer Erhöhung der Zuverlässigkeit und Lebensdauer sowie zu einer Verringerung der Kosten für SoC-Gehäuse und eventuell erforderliche Kühlmaßnahmen.

Der Energieverbrauch eines SoCs kann durch eine geeignete Gestaltung des Programms, besonders die konsequente Verwendung von Energiespar-Modi sowie durch die effiziente Verwendung von Caches und des *Scratchpad*-Speichers (siehe Abschnitt 2.7) erheblich gesenkt werden.

Anforderungen an den Beobachter

Zur Entwicklungsunterstützung ist an dieser Stelle neben der Protokollierung der Ausführungszeit von Instruktionen sowie des Zugriffsmusters auf Variable (Cache, *Scratchpad*, externer Speicher) das hochauflösende Monitoren des Energieverbrauchs sehr hilfreich [71]. Oftmals verfügen SoCs über verschiedene Power-Domains (CPU-Core, I/O-Bereiche), welche mit unterschiedlichen Spannungen versorgt werden. Um hier die Abhängigkeit des Stromverbrauchs der einzelnen Power-Domains vom Programmverlauf ermitteln zu können, sollten die Messungen synchronisiert zu den Tracedaten erfasst werden (Tabelle 2-20). Das durch Optimierung des Stromverbrauchs erzielbare Einsparpotential ist beträchtlich und wird mit bis zu 40% angegeben [72].

		Optimierung des Energieverbrauchs
Vollständigkeit der Beobachtung	Taskwechsel	✓
	Funktionen	✓
	Basic Blocks	✓
	Instruktionen	✓
	Datenzugriffe (CPU)	✓
	Datenzugriffe (Peripherie)	✓
	Cache	✓
	Umgebungsbedingungen	✓
	Ereignisse	✓
Beobachtung von MPSoCs		✓

Tabelle 2-20: Anforderungen an die Beobachtung zur Optimierung des Energieverbrauchs

2.9 Beobachtung angeschlossener Hardware sowie von Umgebungsparametern, Erkennung von Störungen

Oftmals ist es notwendig, neben den Vorgängen innerhalb des SoCs auch Informationen über Umgebungsparameter des SoCs zu erhalten.

Besonders beim Hardware/Software-Integrationstest müssen die von außen zugänglichen Eingänge einer Hardware-Komponente sowie deren Ausgaben beobachtet werden können. Dabei kann es sich um analoge Signale (Eingänge von Analog-Digital-Wandlern, Ausgänge von Digital-Analog-Wandlern) oder digitale Signale (digitale Ein-und Ausgänge, Kommunikationsschnittstellen) handeln. Bei automatisierten Tests kann diese Beobachtung sehr anspruchsvoll sein (z.B. Einbindung von Logik-Analysatoren zur Protokollanalyse, Verwendung von Pattern-Generatoren, Beobachtung von grafischen Ausgaben mittels Bildverarbeitung etc.).

Beim Test und der Zertifizierung sicherheitsrelevanter Systeme müssen Fehlerzustände entsprechend der zugrunde liegenden Risikobetrachtung hervorgerufen und die Reaktion des Systems darauf mit getestet und dokumentiert werden.

Die Analyse nichtdeterministischer Störungen und ihre Angrenzung zu korrigierbaren Mandelbugs stellt eine weitere, oftmals sehr komplexe Anforderung an die Beobachtbarkeit von SoCs dar. Jeder Hinweis kann hier die zeitintensive und oftmals erfolglose Suche nach Ursachen von beobachtetem Fehlverhalten erleichtern. Daher ist es hilfreich, wenn während der Beobachtung eines SoCs synchron zu den erfassten Tracedaten noch eine ganze Reihe anderer Parameter beobachtbar sind. Dazu gehört die Protokollierung von Temperatur, Versorgungsspannung, hochfrequenter Abstrahlungen (Messung der elektromagnetischen Verträglichkeit) sowie aller Informationen, die im Rahmen des Hardware/Software-Integrationstest erfasst werden müssen.

Energieversorgung

Neben der bereits diskutierten Anwendung der Stromverbrauchsmessung zur Optimierung des Energieverbrauchs kann auch die Beobachtung der Energieversorgung des SoCs Hinweise auf Störungen liefern. Dazu sollte sowohl die Versorgungsspannung(en) als auch die Stromaufnahme des SoCs beobachtbar sein. Bei einem Fehlverhalten, welches mit beobachteten Spannungseinbrüchen einhergeht, kann als Ursache der Störung beispielsweise ein fehlerhaftes Design der Stromversorgung des SoCs angenommen werden. Liegt die Stromaufnahme über einem Normalwert, können beispielsweise gegeneinander arbeitende I/O-Pins und damit ein Defekt in der Software als Ursache des Fehlverhaltens festgestellt werden.

Temperatur

Zur Untersuchung von temperaturabhängigem Fehlverhalten (gerade in Grenzbereichen) ist die Messung der Temperatur des SoCs sinnvoll.

Analoge Eingänge

Um die korrekte Funktion und Programmierung von Analog-Digital-Wandlern überprüfen zu können, kann es hilfreich sein, die Spannungen an den entsprechenden Eingängen des SoCs beobachten zu können.

Das folgende Beispiel (Abbildung 2-36, aus [73]) zeigt eine Situation, bei der die Kenntnis von Spannungen an I/O-Pins von SoCs hilfreich zur Untersuchung eines Fehlverhaltens ist.

Beim Betrieb eines Analog-Digital-Wandlers in einem SoC wurde die Messung durch Spikes verfälscht. Durch die gleichzeitige Messung an den Eingängen des integrierten AD-Wandlers konnte nachgewiesen werden, dass die Verfälschung der Messwerte nicht durch Störungen der Eingangssignale verursacht worden sind, sondern dass der AD-Wandler selbst falsch programmiert war bzw. ein Hardware-Problem vorlag.

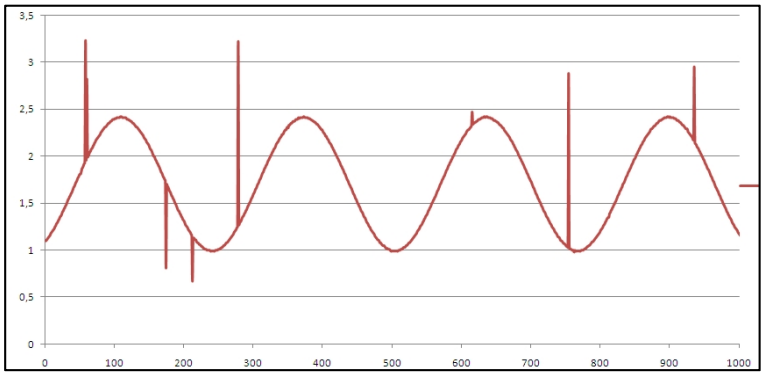


Abbildung 2-36: Mittels ADC erfasste Glitches bei einem NXP LPC1768

Eine Beeinflussung des analogen Eingangssignals durch die Messung muss möglichst minimiert werden („*probe effect*“).

Digitale Eingänge

Die Beobachtung digitaler Eingänge hilft, die durch die Software im SoC erfolgende Auswertung von digitalen Eingangssignalen zu verifizieren. So können beispielsweise Störungen auf digitalen Eingängen ungewollt Interrupts auslösen.

Bei Kommunikationsschnittstellen sollte der Beobachter in der Lage sein, unabhängig vom SoC das Kommunikationsprotokoll zu dekodieren und damit bei eventuellen Störungen der Kommunikation die Information liefern zu können, ob das Fehlverhalten in einer falschen Verarbeitung der Nachricht im SoC begründet ist oder ob die Nachricht bereits beim Empfang gestört war.

Anforderungen an den Beobachter

Ein idealer Beobachter eines SoCs sollte folgende Informationen mit hinreichend großer zeitlicher Auflösung sowie synchron zu den entsprechenden Programm- und Datentraces liefern können (Tabelle 2-21):

- Versorgungsspannung
- Stromaufnahme
- Temperatur
- Analoge Ein- und Ausgangssignale (Funktionalität eines Speicheroszilloskops)
- Digitale Ein- und Ausgangssignale (Funktionalität eines Logikanalysators mit Interpreter für serielle Kommunikationsprotokolle)
- Störabstrahlung (Messung der elektromagnetischen Verträglichkeit)

		Beobachtung angeschlossener Hardware, Umgebungsparametern, Erkennung von Störungen
Vollständigkeit der Beobachtung	Taskwechsel	✓
	Funktionen	✓
	Basic Blocks	✓
	Instruktionen	✓
	Sprünge	✓
	Datenzugriffe (CPU)	✓
	Datenzugriffe (Peripherie)	✓
	Cache	✓
	Abarbeitung	✓
	Zyklusgenauigkeit	✓
	Bussystem	✓
	Umgebungsbedingungen	✓
	Ereignisse	✓
Beobachtung von MPSoCs		✓

Tabelle 2-21: Anforderungen an die Beobachtung angeschlossener Hardware, von Umgebungsparametern sowie der Erkennung von Störungen

3 Kriterien für die Qualität der Beobachtbarkeit

Im folgenden Abschnitt wird ein Katalog von Kriterien definiert, die ein Beobachter erfüllen soll. Diese Kriterien werden an dieser Stelle unabhängig von einer möglichen Implementierbarkeit definiert, d.h. es wird ein Idealzustand dargestellt.

Grundlage für den Kriterienkatalog sind die Anforderungen (*was* soll beobachtbar sein), die im Abschnitt 2 anhand typischer Fragestellungen und Anwendungsfälle ausgearbeitet wurden:

- Vollständigkeit der Beobachtung
- Gleichzeitigkeit der Beobachtung mehrerer Komponenten (z.B. CPUs, Busse)

Ergänzt werden diese mit den folgenden Kriterien:

- Kontinuität
- Echtzeitfähigkeit
- Beeinflussung des SoC
- Beobachtbarkeit von SoCs aus der Serienproduktion
- Beobachtbarkeit in „realer“ Umgebung
- Latenz der Verfügbarkeit der zu beobachtenden Informationen
- Flexibilität des Beobachtungsfokus
- Akzeptanz
- Kosten

Anhand dieser Kriterien können die im Abschnitt 4 diskutierten aktuell verfügbaren Lösungen sowie der in Abschnitt 5 vorgestellte neue Ansatz zur Beobachtung von SoCs bewertet werden.

3.1 Vollständigkeit der Beobachtung

Die Beobachtung aller Schaltelemente in einem SoC ist aufgrund der dafür erforderlichen hohen Bandbreite und der Unmöglichkeit, diese Datenmengen zu speichern und zu verarbeiten nicht möglich. Ausgehend von einem SoC mit ca. 100 Millionen Transistoren und einem Takt von 1 GHz wäre eine Bandbreite von ca. 10^{17} Bit/s erforderlich, um jeden Transistor des SoCs beobachten zu können.

$$10^{8}\,\text{Bit} * 10^{9}\text{s}^{-1} = 10^{17}\text{Bit/s}$$

Die Beobachtung in diesem Detailierungsgrad ist nur beim Debuggen und Testen der Hardwarestrukturen (*silicon debug*) erforderlich. Im Rahmen des Kontextes dieser Arbeit kann ein deutlich höherer Abstraktionsgrad verwendet werden. In Abbildung 3-1 ist dargestellt, wie die o.g. Datenmenge schrittweise reduziert wird, um letztendlich den Zweck der Beobachtung – nämlich die Beantwortung einer bestimmten Fragestellung – zu erreichen.

In einem ersten Schritt kann die Beobachtung auf die programmierbare Aktivität im SoC reduziert werden, wenn die Hardware des SoCs als fehlerfrei angenommen wird.

Für eine CPU ergeben sich beispielsweise folgende relevante Informationen:

- Instruktionsadresse (32 Bit)
- Opcode (32 Bit)
- Datenadresse (32 Bit)
- Gelesene Daten (32 Bit)
- Geschriebene Daten (32 Bit)
- CPU-Register (32 Bit)
- weitere Signale (~100 Bit)

Sollen in einem Multicore-System neben den CPUs auch das Bussystem und busmasterfähige Peripherieeinheiten beobachtet werden, so müssen pro CPU-Takt ca. 10^4 Zustände erfasst werden, was eine Bandbreite von

$$10^4 \text{ Bit} * 10^9 s^{-1} = 10^{13} \text{ Bit/s}$$

erforderlich macht.

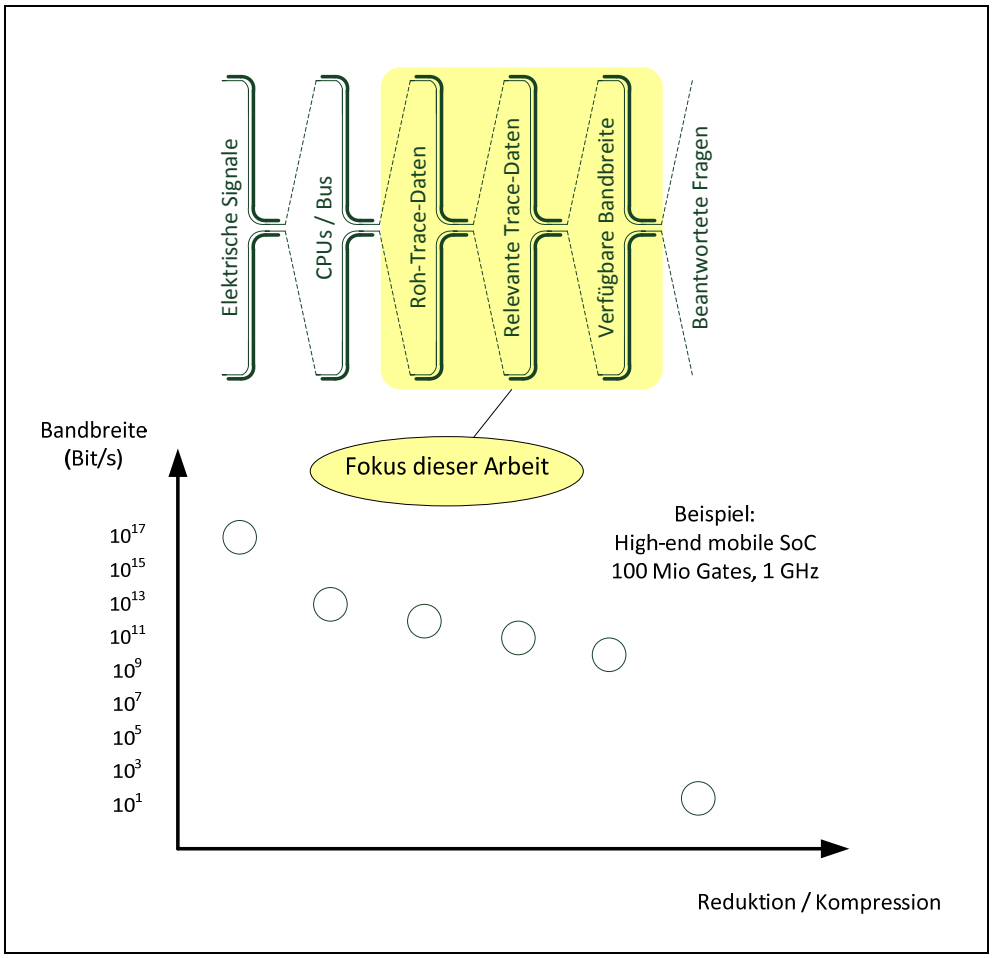


Abbildung 3-1: Zur Beobachtung eines SoCs und entsprechender Datenreduktion / Datenkomprimierung erforderliche Daten / Bandbreite

Die nächste Reduktion findet durch die Beschränkung auf die üblicherweise via Trace-Schnittstelle übertragbaren Informationen statt, dies sind insbesondere:

- Ausgeführte Instruktionen (Instruktionsadresse inkl. Zeitstempel und Anzahl der für die Ausführung benötigten CPU-Takte) (ca. 80 Bit)
- Von CPU(s) gelesene und geschriebene Daten inkl. zugehöriger Adresse und Zeitstempel (ca. 110 Bit)

Bei einem Multicore-System ist zur Ausgabe dieser Informationen eine Bandbreite in der Größenordnung von

$$10^3 \text{ Bit} * 10^9 s^{-1} = 10^{12} \text{ Bit/s}$$

erforderlich.

Werden die Trace-Daten gefiltert und nur die zur Erfüllung der aktuellen Beobachtungsaufgabe relevanten Daten übertragen, so kann die erforderliche Bandbreite weiter eingeschränkt werden. Um die Bandbreite zu reduzieren wird oftmals auch auf die Übertragung von Zeitstempeln verzichtet. Eine weitere Bandbreitenreduktion bringt die Anwendung verschiedener verlustfreier Trace-Komprimierungstechniken, die beim Programm-Trace besonders effektiv sein können. Allerdings muss an dieser Stelle berücksichtigt werden, dass auch bei Reduktion der mittleren Trace-Bandbreite genügend Reserven vorgehalten werden müssen, um auch eventuelle Bandbreitenspitzen übertragen zu können.

Leistungsfähige Trace-Schnittstellen erreichen aktuell Bandbreiten einiger 10 Gbps. So kann beispielsweise der Freescale QorIQ T4240 [74] Trace-Daten mit bis zu ca. 20 Gbps über 4 serielle Hochgeschwindigkeitsschnittstellen ausgeben. Trotz dieser hohen Bandbreiten tritt an dieser Stelle ein Problem auf, welches in den folgenden Abschnitten wiederholt thematisiert werden wird: Im zu beobachtenden SoC sind oftmals wesentlich mehr relevante Informationen verfügbar, als über Trace-Schnittstellen ausgegeben werden können.

Üblicherweise interessieren den Beobachter eines SoCs nicht die detaillierten Trace-Daten, vielmehr benötigt er diese als Ausgangspunkt für die Beantwortung verschiedener Fragestellungen, wie sie beispielhaft im vorhergehenden Abschnitt diskutiert wurden. Wenn man diese Auswertungen ebenfalls als Kompressionsschritt auffasst, kann in diesem speziellen Fall die zur Beobachtung eines SoCs erforderliche Bandbreite auf den Bereich einiger Bit/s reduziert werden – hier erfolgt nur noch die Aussage, ob eine vordefinierte Bedingung erfüllt wurde oder nicht.

Der Fokus dieser Arbeit liegt auf der reinen Beobachtung der Vorgänge im SoC (verlustfreie Kompression der Trace-Daten), die Möglichkeiten zur Auswertung der Trace-Daten zur Beantwortung bestimmter Fragestellungen werden im Abschnitt 6 kurz umrissen.

3.1.1 Prozessor

Die Protokollierung des Verhaltens einer CPU ist bei der Beobachtung von SoCs die wichtigste Aufgabe. Dabei sollten folgende Informationen verfügbar gemacht werden:

- Ausführung von Funktionen, Basic Blocks und Instruktionen
- von CPUs und anderen Busmastern gelesene / geschriebene Daten
- Inhalt der Registerbank der CPU(s)
- Verhalten des Caches
- Spekulative Abarbeitung von Programmcode / Änderung der Ausführungsreihenfolge
- Anzahl benötigter Taktzyklen für die Abarbeitung von Instruktionen und Basic Blocks

Funktionen, Basic Blocks und Instruktionen

Die Beobachtung der Ausführung von Funktionen ist für einige Fragestellungen ausreichend, in sehr vielen Fällen ist aber eine detaillierte Rekonstruktion des Programmablaufs erforderlich. Dazu müssen die von der CPU ausgeführten Instruktionen dem Beobachter bekannt gemacht werden, was üblicherweise durch die Ausgabe des Programm-Zählers (PC) geschieht. Alternativ kann auch nur die Ausführung von *basic blocks* beobachtet werden, aus denen sich dann die Ausführung der einzelnen Instruktionen herleiten lässt.

Anhand des Inhalts des Object-Files können im Entwicklungssystem die der jeweiligen Code-Adresse zugeordneten Instruktionen ermittelt werden.

Zur Beobachtung von Betriebssystemen oder der korrekten Adressberechnung bei virtuellen Adressierungen ist es zusätzlich notwendig, neben dem aktuellen Stand des PC auch den jeweiligen Kontext verfügbar zu machen.

Je nach Beobachtungsaufgabe müssen die Ausführung von Instruktionen sowie die Ausführung von Sprüngen beobachtbar sein. Teilweise ist es auch wichtig, die Sprung-Historie zu kennen.

Von CPU und Busmastern gelesene / geschriebene Daten

Neben dem Programmzähler sind auch die von der CPU und anderen Busmastern (z.B. DMA-fähige Peripherieeinheiten) gelesenen und geschriebenen Daten von Interesse. Hier müssen sowohl die Werte, die zugehörigen Adressen sowie die Richtung des Datentransfers (Lesen oder Schreiben) beobachtbar sein.

Inhalt der Registerbank

Die Beobachtbarkeit des Inhalts der aktuellen Registerbank hilft bei der Überwachung des Stacks sowie von lokalen Variablen.

Cache

Für Optimierungen und Fehlersuche spielt die Beobachtbarkeit des Daten- und Programm-Caches eine große Rolle. Hier ist es interessant, welche Daten / Instruktionen im Cache gehalten werden und welche nachgeladen werden müssen.

Beschleunigung der Abarbeitung

Moderne CPUs verfügen über verschiedene Mechanismen zur Beschleunigung der Abarbeitung des Programmcodes wie die spekulative Abarbeitung von Programm-code (*speculative execution*) oder die Änderung der Ausführungsreihenfolge (*Out-of-order execution*). Diese Mechanismen sollten beobachtbar sein, um die Applikation möglichst so anpassen zu können, dass die mögliche Beschleunigung der Abarbeitung optimal genutzt wird.

Zyklusgenauigkeit

Für die exakte Beobachtung des zeitlichen Verhaltens eines SoCs ist die Beobachtung der ausgeführten Instruktionen nicht ausreichend, vielmehr muss auch die für jede Instruktion benötigte Anzahl an Taktzyklen bekannt gemacht werden. Mit Hilfe dieser Informationen kann beispielsweise eine Messung maximaler Ausführungszeiten (dynamische WCET-Messung) durchgeführt werden.

3.1.2 Ereignisse

Neben dem Prozessor spielen auch die Beobachtbarkeit der Art und des Zeitpunktes des Eintretens von Ereignissen eine wichtige Rolle bei der Analyse von Abläufen in einem SoC. Neben der Beobachtung der tatsächlichen Verarbeitung eines Ereignisses (z.B. Abarbeitung der Interrupt-Serviceroutine) kann auch die Beobachtung der Auslösung eines Ereignisses (z.B. Interrupt-Request einer Peripherieeinheit) wichtig sein. Letzteres stellt beispielsweise eine Möglichkeit dar, das „Verhungern“ einer Interrupt-Anforderung niedriger Priorität entdecken zu können.

3.1.3 Bussysteme

Für die Kommunikation zwischen den einzelnen IP-Blöcken sind auf einem SoC Bussysteme implementiert. Die Kommunikation auf vielen modernen Bussystemen (z.B. ARM AXI) erfolgt dabei transaktionsorientiert. Je nach Bussystem sind diese Transaktionen unterschiedlich komplex und beginnen mit der Bekanntgabe einer Adresse und ggfs. der Länge das Datenblocks durch den initiierenden Busmaster. Danach werden die zu lesenden bzw. zu schreibenden Daten auf den Bus gelegt, mittels Handshakes wird der korrekte Transfer abgesichert.

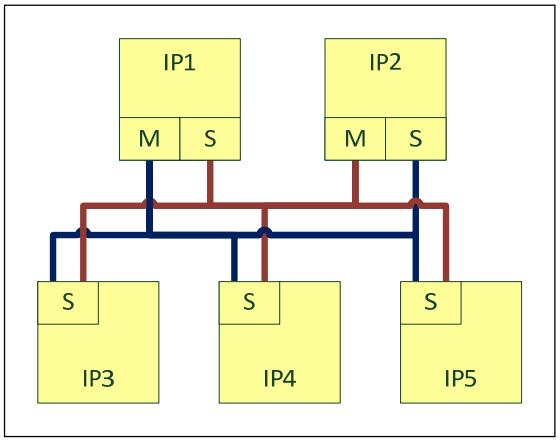


Abbildung 3-2: Beispiel eines Multilayer-Busses

Moderne Multilayer-Bus-Architekturen gestatten den parallelen Transfer von Daten zwischen verschiedenen Busmastern und Slaves (Abbildung 3-2, nach [75]). Dabei sind die Busmaster-Interfaces als „ M“, die Slave-Interfaces als „S“ bezeichnet. In Abbildung 3-3 (nach [75]) werden parallel ablaufende Transfers gezeigt, bei denen die Master-Interfaces IP1.M und IP2.M zeitgleich auf unterschiedliche Slave-Interfaces (Schreiben auf IP4 und IP5, Lesen on IP4 und IP5 usw.) zugreifen.

Die hohe Bandbreite des Busses, die durch die Multilayer-Architektur erzielt wird, macht die Beobachtbarkeit dieser Busse mit oftmals mehreren hundert Signalen besonders schwierig.

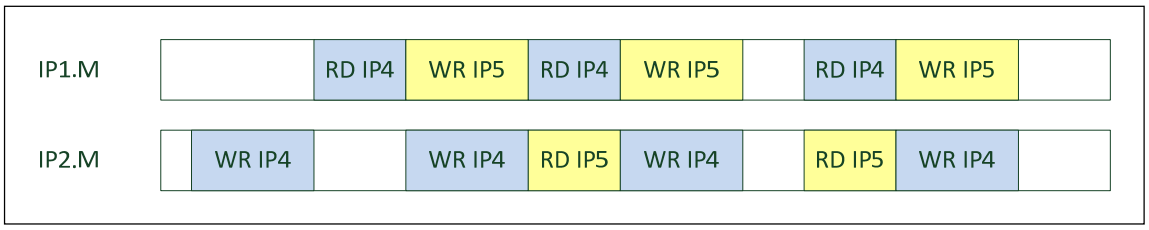


Abbildung 3-3: Beispiel paralleler Transfers auf einem Multilayer-Bus

Bei Bussystemen sind vom Beobachter folgende Aufgaben zu erfüllen:

- Beobachtung der Bus-Master-Schnittstellen
- Beobachtung der Bus-Slave-Schnittstellen (Peripherie), hier besonders die Protokollierung von Zeitüberschreitungen (*timeout*), konkurrierende Zugriffe auf einen Slave durch mehrere Master, unvollständige geteilte Transaktionen (*incomplete SPLIT transaction*)
- Protokollierung der Bus-Leistung (*bus performance monitoring*)

Ein weiterer Schritt zur Steigerung der Leistungsfähigkeit und Flexibilität der Busse besteht in der Verwendung von Kommunikationsprotokollen, hier spricht man dann von Netzwerken auf dem Chip (*Network on Chip - NoC*)[76]. Aufgrund der hohen internen Bandbreiten ist eine detaillierte Beobachtung der einzelnen Transfers sehr aufwändig. In kommerziellen Lösungen wird hier mit hardwareunterstützter Code-Instrumentierung gearbeitet [77], in der Literatur (z.B.[78], [79]) wird die Verwendung spezieller Monitoring-Systeme beschrieben.

3.1.4 Umgebungsbedingungen

Die Erfassung der Umgebungsbedingungen sollte folgendes umfassen:

- Temperatur
- Stromaufnahme , taktzyklusgenaue Auflösung
- Versorgungsspannung
- Analoge und digitale Eingangssignale

Dabei ist es wichtig, dass eine synchronisierte Zuordnung zwischen den beobachteten Umgebungsbedingungen und den Abläufen innerhalb des SoCs hergestellt werden kann.

3.2 Beobachtbarkeit von MPSoCs

Eine umfassende Beobachtbarkeit von MPSoCs ist sehr wichtig, da durch die Komplexität der MPSoCs und die parallele Programmabarbeitung die Wahrscheinlichkeit für das Auftreten von Mandelbugs erhöht ist. Diese sind ohne geeignete Werkzeuge nur sehr schwer oder gar nicht zu entdecken und zu verstehen.

Erschwerend kommt hinzu, dass oft in MPSoCs auch heterogene CPU-Kerne verschiedener Hersteller mit jeweils eigenen Compilern eingesetzt werden. Ein Beispiel hierfür ist der TMS320DM8168 [80], welcher einen ARM Cortex A8 [81] sowie einen C674x DSP [82] von Texas Instruments vereint.

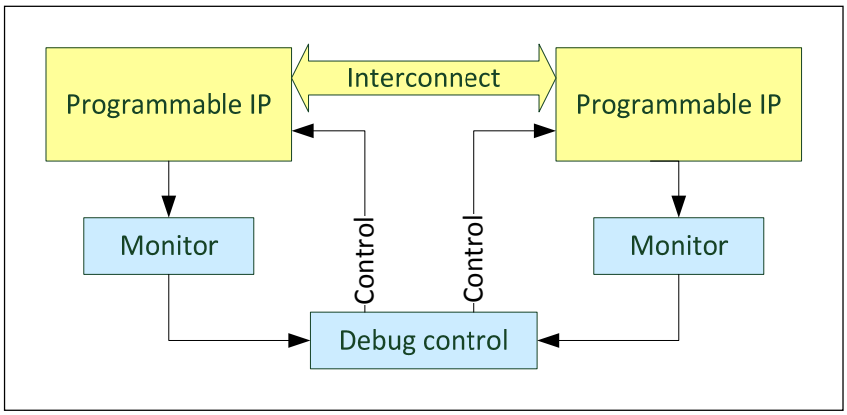


Abbildung 3-4: Beobachtung einzelner programmierbarer Einheiten (*Computation-centric debug*)

Der bei der Beobachtung von SoCs mit einer CPU verfolgte Ansatz der isolierten Beobachtung der einzelnen CPU (*Computation-centric debug,* siehe Abbildung 3-4, nach [83]) lässt sich nicht ohne weiteres auf MPSoCs erweitern. Vielmehr ist eine zeitgleiche Beobachtbarkeit der verschiedenen Busmaster (CPUs, busmasterfähige Peripherieeinheiten) sowie der Busse selbst (Arbitrierungen, Transaktionen) (*Communication-centric debug* - Abbildung 3-5, nach [83]) notwendig, zusätzlich müssen noch Informationen verfügbar gemacht werden, die das zeitliche Verhalten der einzelnen programmierbaren Einheiten in Zusammenhang bringen.

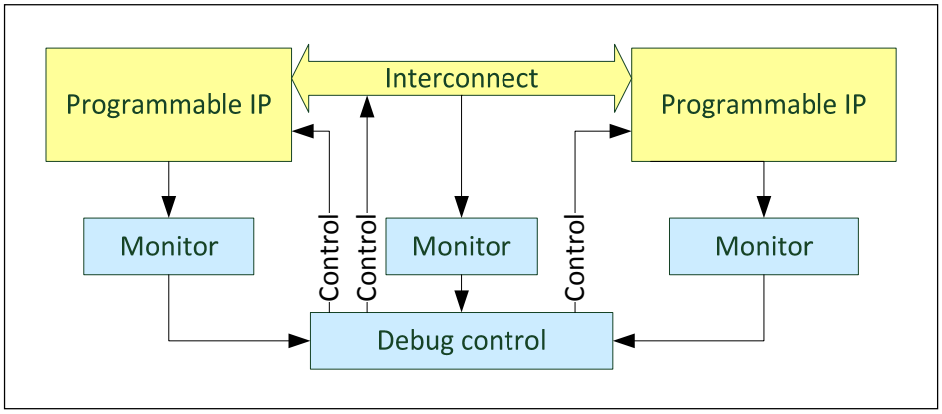


Abbildung 3-5: Beobachtung einzelner programmierbarer Einheiten und der Kommunikation zwischen den einzelnen programmierbaren Einheiten (*Communication-centric debug*)

Eine zusätzliche Erschwernis stellt die erforderliche Berücksichtigung verschiedener Clock- und Power-Domains dar [71].

3.3 Echtzeitfähigkeit und Kontinuität

Um ein SoC unter sicheren Bedingungen und in einer realistischen Umgebung beobachten zu können, müssen die CPUs mit ihrer regulären Geschwindigkeit laufen können.

Auch können durch einen verlangsamten Betrieb bedingte Änderungen von Zugriffszeiten (z.B. Buszugriffe, Speicherzugriffe, Analog/Digital-Wandlungen) Fehler hervorrufen oder verdecken.

Für eine realistische und sichere Interaktion mit externen Systemen, besonders bei zeitkritischen Applikationen (z.B. Motorsteuerungen, Festplatten) stellt eine Beeinflussung der Geschwindigkeit eine nichtakzeptable Einschränkung dar.

Ein wichtiges Qualitätsmerkmal für die Beobachtung von SoCs stellt die mögliche Länge der Beobachtung dar. Je länger die Beobachtungsdauer ist, umso belastbarer sind getroffene statistische Aussagen (z.B. maximale Ausführungszeit einer Funktion), weiterhin können bei nur kurzen Beobachtungsfenstern sporadisch auftretende Fehler außerhalb des Beobachtungszeitraums auftreten und damit unsichtbar bleiben. Den Idealzustand stellt eine unlimitierte Beobachtungsdauer dar.

3.4 Gleichzeitige Durchführung vieler Beobachtungsaufgaben

Bei der Untersuchung komplexer Ursachen von Fehlverhalten ist es hilfreich, eine Vielzahl von Beobachtungs-Fokussen gleichzeitig verfolgen zu können.

Beispielsweise bei der automatischen Erkennung von Wettlaufsituationen sollten möglichst alle Variablen gleichzeitig beobachtbar sein, um die Beobachtungszeit reduzieren und ggfs. komplexe Abhängigkeiten analysieren zu können.

3.5 Beeinflussung des SoCs

Die Beobachtung des SoCs sollte möglichst passiv (nicht intrusiv) erfolgen, ein Einfluss der Beobachtung auf das Systemverhalten soll vermieden werden.

Wenn dies nicht möglich ist, sollen nur leichte, tolerable Störungen des Systemverhaltens (minimal intrusiv) auftreten (z.B. Anhalten der CPU bei Überlauf des Trace-FIFOs).

3.6 Beobachtbarkeit von SoCs aus der Serienproduktion

Die Beobachtung von SoCs sollte auf der gleichen Hardware erfolgen, die auch in der Serienproduktion eingesetzt wird. Für den Fall des Einsatzes spezieller Emulations-Chips kann kein Hersteller eine vollständige Übereinstimmung des Systemverhaltens mit Serien-SoCs garantieren. Oftmals sind Emulations-Chips in einer anderen Technologie hergestellt, besitzen eine andere Peripherieausstattung und beinhalten meist nicht die Fehlerkorrekturen (*silicon patches*), über welche die zu einem späteren Zeitpunkt mit korrigiertem Maskensatz hergestellten Serien-SoCs verfügen (siehe auch Abschnitt 4.2).

Wenn nun der Test und die Zertifizierung eines Systems mit Hardware erfolgt, die nicht das garantiert identische Systemverhalten zeigt, so ist die Vollständigkeit und Aussagekraft der durchgeführten Tests kritisch zu hinterfragen.

3.7 Beobachtbarkeit in „realer“ Umgebung

Die zu untersuchenden Systeme sollten auch „im Feld“ beobachtbar sein. Dies erfordert Hardware, die einmal über eine geringe Baugröße verfügt, andererseits aber auch hinreichend robust ist, um den unter realen Betriebsbedingungen eines Systems auftretenden Belastungen gewachsen zu sein. Weiterhin sollte die Verbindung zum Emulator hinreichend robust sowie das Beobachtungssystem entsprechend thermisch (Automotive) und mechanisch belastbar sein.

3.8 Flexibilität des Beobachtungsfokus

Ein Qualitätsmerkmal für die Beobachtung von SoCs ist die Flexibilität des Beobachters. Hier wird bewertet, wie groß der Aufwand ist, um neue Fragestellungen beantworten zu können.

Oftmals ist es während des Debug- und Optimierungsprozesses erforderlich, den Fokus der Beobachtung zu verändern. Je nach Implementierung des Beobachters ist dies mehr oder weniger komfortabel möglich. Beispielsweise kann dies durch eine einfach zu erlernende Hochsprache erfolgen, in der auch auf symbolische Debug-Informationen zugegriffen werden kann. Für häufig auftretende Fragestellungen sind Wizards sehr hilfreich, die den Entwickler Schritt für Schritt bei der Konfiguration unterstützen. Die Akzeptanz eines Beobachtungswerkzeugs durch Entwicklungs- und Testingenieure ist oftmals nur gegeben, wenn diese sich auf ihre eigentliche Arbeit konzentrieren können und nicht gezwungen sind, sich in die manuelle Konfiguration komplexer Trigger-Logik bzw. Trace-Filter einarbeiten zu müssen [84],[85].

Ein weiterer wichtiger Punkt ist eine mögliche schnelle Veränderbarkeit der Beobachtungsaufgabe während der Laufzeit der Applikation. Wenn während des Debug-/ Optimierungsprozesses der Zugriff auf andere Informationen aus dem SoC erforderlich ist, so sollte eine Anpassung des Beobachtungsfokus auf diese Änderung zeitnah erfolgen können, ohne dass dazu die Applikation neu gestartet werden muss.

3.9 Latenz

Ein weiterer Aspekt bei der Beobachtung von SoCs ist die zeitnahe Verfügbarkeit der für die Beobachtung relevanten Informationen. Um in der Lage zu sein, das zu beobachtende System kontrollieren zu können, sollten diese Informationen zeitgleich für den Beobachter verfügbar sein. Da dies in den meisten Fällen technisch nicht möglich ist, sollte die Latenz möglichst kurz (einige CPU-Takte) und zudem noch wohldefiniert sein, d.h. die Zeitspanne zwischen den zu beobachtenden Ereignissen und der Verfügbarkeit dieser Informationen für den Beobachter sollte konstant bzw. berechenbar sein.

Eine kurze Latenz bietet den Vorteil, zeitnah auf ein vordefiniertes Verhalten des zu beobachtenden SoCs reagieren zu können. Beispielsweise kann bei einem Taskwechsel der Fokus der Beobachtung dynamisch angepasst werden. Eine weitere Anwendung ist das situationsabhängige Triggern von Tests, so kann beispielsweise bei der Beobachtung des Erreichens eines bestimmten Systemzustands ein Interrupt ausgelöst oder eine CAN-Nachricht an das System gesandt werden.

3.10 Qualifizierung des Beobachters

Ein weiteres wichtiges Kriterium für die Qualität eines Beobachtungswerkzeugs von SoCs ist dessen Zuverlässigkeit. Dies ist besonders für Anwendungen mit einem hohen Anspruch an die Verlässlichkeit und Sicherheit relevant.

Für die Beobachtung von SoCs ergeben sich dabei folgende Anforderungen:

- Das Beobachtungsverfahren darf kein Fehlverhalten des zu analysierenden Systems ***bewirken***.
- Das Beobachtungsverfahren darf kein Fehlverhalten des zu analysierenden Systems ***maskieren***.

Je intrusiver das Beobachtungsverfahren und je kritischer die Anwendung ist, desto höher sind die Anforderungen an das Entwicklungswerkzeug, deren Nachweis dann mit entsprechendem Aufwand verbunden ist.

Diese Problematik wird in der generischen Grundnorm für funktionale Sicherheit IEC 61508 (Teil 3: Anforderungen an Software) [86] sowie u.a. folgenden domainspezifischer Standards behandelt:

- DO-178C/ED-12C [40] [87] (Luftfahrt)
- ISO 26262:2011[35] (Straßenfahrzeuge)
- EN 50128 [88] (Bahnfahrzeuge)
- IEC 62304 [89] (Medizinische Software)
- IEC 62061 [90] (Maschinensicherheit)
- IEC 61511 [91] (Prozessindustrie)
- IEC 60880 [92] (Kernkraftwerke)

Diese Standards gelten für alle Arten von Software-Entwicklungswerkzeugen wie Entwicklungsumgebungen, Code-Generatoren, Compiler, Testdaten-Generatoren, statische Analyse-Werkzeuge und Versions-Kontrollsysteme.

Im Folgenden werden von ausgewählten Standards empfohlene oder geforderte Anforderungen an Entwicklungswerkzeuge diskutiert, die zur Beobachtung von SoCs dienen.

IEC 61508

Die IEC 61508 beschäftigt sich allgemein mit Fragen der funktionalen Sicherheit. Die im Jahre 2010 erschienene neueste Version (IEC 61508 Ed. 2.0 [86] Teil 3, Abschnitt 7.4.4) beinhaltet nun auch Anforderungen an die Qualifikation von Entwicklungswerkzeugen. Dabei werden die Werkzeuge in zwei Kategorien unterteilt (IEC 61508 Ed. 2.0 [86] Teil 4, Abschnitt 3.2.10 und 3.2.11):

Online support tools sind alle Werkzeuge, die eine direkte Beeinflussung des sicherheitskritischen Systems zur Laufzeit bewirken können.

Offline support tools werden nur während der Entwicklungsphase eingesetzt und können keine direkte Beeinflussung des sicherheitskritischen Systems zur Laufzeit bewirken. Sie werden in folgende Unterkategorien eingeteilt:

T1: kein direkter oder indirekter Beitrag zum Programmcode (z.B. Texteditor, Unterstützungswerkzeug ohne automatische Code-Generierung)

T2: Werkzeuge zur Verifikation oder dem Test des Designs, eine Fehlfunktion des Werkzeugs kann zu einer Maskierung von Fehlverhalten des Programms führen (z.B. Testfall-Generatoren, Werkzeug zur Coverage-Messung)

T3: Werkzeuge, die direkt oder indirekt ausführbaren Code erzeugen (z.B. optimierende Compiler)

Werkzeuge für die Beobachtung von SoCs lassen sich als *online support tools* oder als *offline support tools* der Unterkategorien T3 und T2 einordnen, wobei letztere den geringsten Aufwand für die Qualifizierung erwarten lässt.

Beim Einsatz von Entwicklungswerkzeugen wird eine auf die einzelnen Anwendungsfälle bezogene Analyse möglicher Gefährdungen gefordert (Tabelle 3-1). Weiterhin werden in Abhängigkeit von der Klassifizierung, unterschiedliche Anforderungen an die Qualifikation des Entwicklungswerkzeugs gestellt.

Referenz	Merkmal	Offline			Online
		T1	T2	T3	
7.4.4.1	Das Entwicklungswerkzeug ist als Bestandteil des sicherheitsrelevanten Systems zu betrachten.				✓
7.4.4.4	Das Entwicklungswerkzeug muss über eine Spezifikation oder eine Dokumentation verfügen.		✓	✓	
7.4.4.5	Das Entwicklungswerkzeug muss hinsichtlich Abhängigkeit, Ausfallmechanismen und ggfs. geeigneten Schutzmaßnahmen beurteilt werden.		✓	✓	
7.4.4.6 / 7.4.4.8	Nachweis der Übereinstimmung des Entwicklungswerkzeugs mit seiner Spezifikation oder Dokumentation. Falls dieser Nachweis nicht möglich ist, müssen wirksame Maßnahmen zur Vermeidung von Ausfällen des sicherheitsrelevanten Systems aufgrund von Fehlern des Entwicklungswerkzeugs implementiert werden.			✓	
7.4.4.16	Es ist ein Konfigurations-Management für das Entwicklungswerkzeug erforderlich.		✓	✓	

Tabelle 3-1: Ausgewählte Anforderungen an die Qualifikation eines Entwicklungswerkzeugs entsprechend IEC 61508

Luftfahrt

Im Rahmen der Revision der DO-178C/ED-12C wurde mit der DO-330/ED-215 [49] [93] ein ergänzender Standard erstellt, der Anforderungen an das Entwicklungswerkzeug definiert. Das Entwicklungswerkzeug wird nach drei Kriterien klassifiziert, die in Tabelle 3-2 dargestellt sind.

Kriterium	Bedeutung
1	Ein Werkzeug, dessen Ausgabe Bestandteil der resultierenden Software ist und hier Fehler einbringen könnte (z.B. Compiler, Linker).
2	Ein Werkzeug, welches den Verifikations-Prozess automatisiert und welches es versäumen könnte, ein beobachtbares Fehlverhalten zu entdecken (z.B. In-Circuit- Emulator, Simulator). Die Ausgabe des Werkzeugs ist dabei aussagekräftig genug, dass auf einen nicht automatisierten Verifikationsprozess verzichtet werden kann ein Entwicklungsschritt der resultierenden Software weggelassen oder vereinfacht werden kann
3	Ein Werkzeug, welches im Rahmen seiner vorgegebenen Verwendung versäumen könnte, einen Fehler zu entdecken.

Tabelle 3-2: Kriterien für die Klassifizierung von Entwicklungswerkzeugen entsprechend DO-330/ED-215

Ausgehend von dieser Klassifizierung und in Abhängigkeit von der Kritikalität der Software (*software level* oder *Design Assurance Level (DAL)* - A: höchste, D: niedrigste) lässt sich der *Tool Qualification Level* (TQL) bestimmen (Tabelle 3-3). Dabei stell TQL-1 die höchsten Anforderungen an den Nachweis der Qualität eines Entwicklungswerkzeugs. So werden beispielsweise für einen nach TQL-1 eingestuften Code-Generator die gleichen Qualitätsanforderungen gestellt wie sie für die „manuelle“ Entwicklung des Anwendungscodes gelten würden.

	Kriterium		
Software Level	1	2	3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

Tabelle 3-3: Ermittlung der *Tool Qualification Level* entsprechend DO-330/ED-215

Für ein Beobachtungswerkzeug ist es an dieser Stelle vorteilhaft, wenn dieses nach Kriterium 3 oder Kriterium 2 klassifiziert werden kann, da dann der mit der Qualifizierung verbundene Aufwand möglichst gering gehalten werden kann.

Automobilbau

Bei der ISO 26262:2011 [35] (Funktionale Sicherheit von Straßenfahrzeugen – Teil 8: Unterstützende Prozesse) unterscheidet die Klassifizierung von Entwicklungswerkzeugen zwischen einem *Tool Impact Level* (TI) und einem *Tool Error Detection Level* (TD).

Der *Tool Impact Level* bewertet die Möglichkeit, dass eine Fehlfunktion eines Werkzeugs zum Versagen einer Sicherheitsanforderung oder einer Risiko-Kontrollmaßnahme führt. Besteht kein derartiger Einfluss, so wird TI0 gewählt, andernfalls TI1.

Der *Tool Error Detection Level* bewertet die Möglichkeit, dass das Werkzeug ein mögliches Fehlverhalten der Anwendung falsch darstellt (falsch-negativ). Zur Klassifikation wird mittels geeigneter Verfahren wie z.B. der Fehlermöglichkeits- und Einfluss-Analyse (*Failure Mode and Effects Analysis*, FMEA) [94] das Gefährdungspotential des Beobachtungsverfahrens ermittelt.

TD1 wird bei einem hohen Vertrauen, dass eine Fehlfunktion des Werkzeugs verhindert oder vor dem Inverkehrbringen eines Produkts entdeckt wird, gewählt. TD2 wird bei mittlerem Vertrauen gewählt. In allen anderen Fällen muss TD3 gewählt werden, dies gilt vor allem, wenn es während des Entwicklungsprozesses keine systematischen Maßnahmen zur Software-Qualitätssicherung erfolgten und damit ein eventuelles Fehlverhalten des Werkzeugs nur zufällig erkannt werden kann.

Aus der Kombination des *Tool Impact Level* und des *Tool Error Detection Levels* wird der notwendige Vertrauensgrad (*Tool Confidence Level* - TCL) des Werkzeugs ermittelt (Abbildung 3-6, nach [95]).

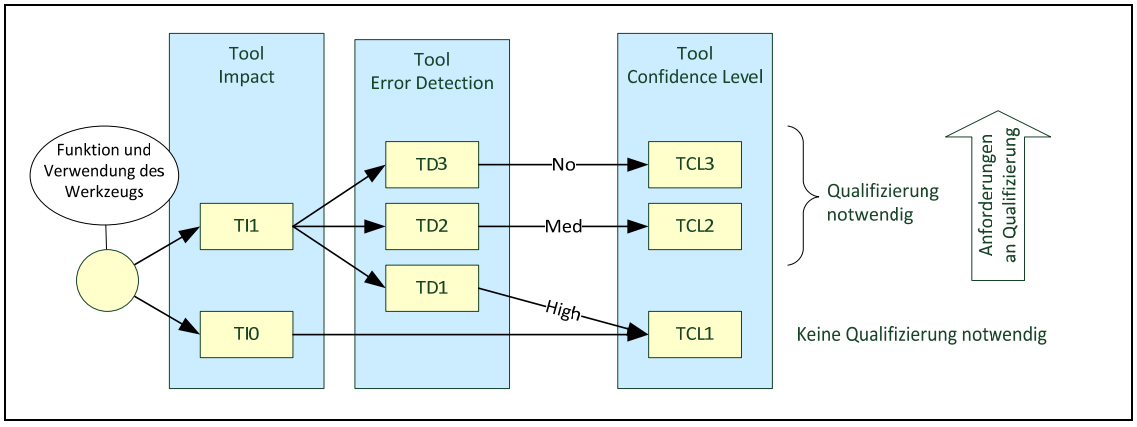


Abbildung 3-6: Bestimmung des Vertrauensgrads nach ISO 26262:2011

Die Qualifizierungsmethode für das Entwicklungswerkzeug ergibt sich aus dem *Tool Confidence Level* und der Software-Sicherheitsklasse (*Automotive Safety Integrity Levels* - ASIL, D: höchste) (Tabelle 3-4, ++: besonders empfohlen, + empfohlen).

	Tool Confidence Level		TCL2				TCL3			
	Sicherheitsklasse (ASIL)		A	B	C	D	A	B	C	D
Methode	1a	Gesteigertes Vertrauen aus der bisherigen Nutzung	++	++	++	+	++	++	+	+
	1b	Evaluierung des Entwicklungsprozesses des Werkzeuges	++	++	++	+	++	++	+	+
	1c	Validierung des Werkzeuges	+	+	+	++	+	+	++	++
	1d	Entwicklung des Werkzeuges gemäß Sicherheitsstandard	+	+	+	++	+	+	++	++

Tabelle 3-4: Empfehlungen für verwendete Methoden abhängig vom Vertrauensgrad und der Sicherheitsklasse der Software

Für die Qualifizierung des Entwicklungswerkzeugs ist der Anwender verantwortlich, er sollte dazu vom Hersteller entsprechend unterstützt werden. Dazu zählen die umfassende Dokumentation des Werkzeugs sowie die Bereitstellung einer Sammlung von Standardargumenten für den Einsatz im vorgesehenen Kontext. Dies kann beispielsweise mit präzise dokumentierten Anwendungsszenarien inklusive den dazu passenden Testfällen geschehen, die dem Anwender in einem sogenannten „*Tool Qualification Kit*“ zur Verfügung gestellt werden.

Um den Aufwand für die Qualifizierung des Entwicklungswerkzeugs gering zu halten oder diese ganz zu vermeiden, kann versucht werden, eine niedrige Klassifikation des Beobachtungswerkzeugs zu erreichen.

Dies bedeutet, dass das Werkzeug möglichst nicht intrusiv ist, d.h. selbst kein Fehlverhalten der Anwendung bewirken kann. Weiterhin müssen die Ausgaben des Beobachtungswerkzeugs so zuverlässig sein, dass eine Maskierung eines eventuellen Fehlverhaltens hinreichend unwahrscheinlich ist.

3.11 Kosten

Die Beobachtbarkeit von SoCs ist mit Kosten verbunden, die im Folgenden diskutiert werden sollen.

3.11.1 Chipfläche

Die Strukturen, die auf dem SoC zur Beobachtung seiner internen Abläufe implementiert sind, werden nur während der Entwicklung und zur Fehleranalyse im Feld benutzt. Beim eigentlichen produktiven Einsatz des SoCs handelt es sich hier um „totes“ Silizium, dementsprechend ist es an dieser Stelle schwierig, Ressourcen bewilligt zu bekommen. Andererseits hängt der Erfolg eines Projektes oft von den verfügbaren Debug-Strukturen ab, deshalb muss eine sorgfältige Abwägung zwischen vermeintlichen Einsparpotentialen und einer möglichst umfassenden Beobachtbarkeit vorgenommen werden.

Bei der Verwendung von permanenten softwarebasierten Beobachtern (Instrumentierung, siehe Abschnitt 4.1) müssen auch die für die Beobachtbarkeit verbrauchten Ressourcen (z.B. zusätzlicher Speicherbedarf und Rechenzeit) mit berücksichtigt werden, oft relativiert sich an dieser Stelle der vermeintliche Kostenvorteil einer softwarebasierten Lösung.

3.11.2 I/O-Pins

Ein weiterer wichtiger Kostenpunkt sind die durch die Beobachtung belegten I/O-Pins. Einmal benötigen auch sie Chipfläche, zum anderen machen sie größere Gehäuse sowie entsprechend größere Leiterplatten erforderlich. Eine Mehrfach-Nutzung der I/O-Pins (Ausgabe von Trace-Daten sowie ihr eigentliche Funktionalität) hat den Nachteil, dass das getestete System nicht mit dem im produktiven Einsatz verwendeten System identisch ist und sich somit eine Testlücke auftut.

Einen Ausweg an dieser Stelle bietet das Verfahren der Port-Ersetzung, bei dem das Entwicklungswerkzeug die Funktionalität der belegten I/O-Ports emuliert und somit aus Sicht der Applikation auch während des Testens und Debuggens alle Pins wie in produktiven Programmläufen zur Verfügung stehen. Dabei ist zu berücksichtigen, dass die Port-Ersetzung durch das Entwicklungswerkzeug zu einer Verzögerung des Programmablaufs führen kann, da speziell beim Lesen entsprechende Anforderung erst an das Entwicklungswerkzeug übermittelt und auf die gelesenen Daten gewartet werden muss.

3.11.3 Implementierungsaufwand

Ein wichtiger Kostenfaktor ist der Implementierungsaufwand für Strukturen zur Beobachtung eines SoCs. Idealerweise liegen diese schon als erprobtes Design vor und können einfach in ein neues SoC-Design eingebunden werden.

3.11.4 Entwicklungswerkzeuge

Leistungsfähige Entwicklungswerkzeuge sind teuer. Die Preise resultieren aus dem hohen Entwicklungs-, Support- und Pflegeaufwand sowie aus den relativ geringen Stückzahlen.

Gute Entwicklungswerkzeuge sind aber oft der Schlüssel für einen nachhaltigen Projekterfolg. Beispielsweise kann ein mit aufwändigen Entwicklungswerkzeugen aufgespürter Mandelbug (siehe Abschnitt 2.1) eine teure Rückrufaktion vermeiden. Legt man zudem die Kosten der notwendigen Entwicklungswerkzeuge auf die Anzahl der ausgelieferten Systeme um, so relativieren sich diese Kosten schnell.

3.11.5 Energiekosten

Die Möglichkeit der Beobachtbarkeit eines SoCs kostet unter Umständen Energie, besonders wenn permanente softwarebasierte Beobachter (Instrumentierung, siehe Abschnitt 2.1) verwendet werden. Die Energiekosten lassen sich in zwei Kategorien einteilen:

Energieverbrauchskosten

Hier werden die durch die Möglichkeit der Beobachtung verursachten Energieverbrauchskosten berücksichtigt. Diese lassen sich, z.B. bei einem Automobil aus den anteiligen Kraftstoffkosten berechnen. Da die Energieverbrauchskosten in den meisten Fällen vom Endkunden übernommen werden und der anteilige Energieverbrauch für die Strukturen zur Beobachtung eines SoCs eher gering ist, sind die Energieverbrauchskosten in den meisten Fällen vernachlässigbar.

Energiebereitstellungskosten

In vielen Systemen muss elektrische Energie erzeugt oder gespeichert werden. Die Kosten für die Bereitstellung elektrischer Energie sind teilweise relativ hoch, so wird z.B. im Automobilbau mit 0,50€ bis 1,00€ pro bereitgestelltem Watt elektrischer Leistung gerechnet.

3.11.6 Kosten des Nachweises der Zuverlässigkeit des Beobachtungswerkzeugs

Der im vorhergehenden Abschnitt behandelte Nachweis der Zuverlässigkeit eines Beobachtungswerkzeugs kann ebenfalls hohe Kosten verursachen. Diese setzen sich zusammen aus dem Ressourcenaufwand zur Führung dieses Nachweises sowie durch länderspezifische Zertifizierungs- und Abnahmekosten (z.B. Kosten der „Benannten Stelle“[16] bei der Abnahme von Medizingeräten) zusammen.

3.12 Universalität und Skalierbarkeit

Ein weiteres Merkmal eines Beobachtungsverfahrens ist dessen **Universalität**. Diese zeigt sich in einer weitgehenden Unabhängigkeit von der Architektur des zu beobachtenden Systems – der Anwender kann leicht zwischen verschiedenen SoCs-Architekturen wechseln.

Die **Skalierbarkeit** eines Beobachtungsverfahrens bildet die Möglichkeit ab, durch das schrittweise Hinzufügen von Ressourcen das Werkzeug steigenden Anforderungen anzupassen. Eine gute Skalierbarkeit gestattet es, initial mit einer kleinen und kostengünstigen Version des Werkzeugs zu beginnen und dieses dann schrittweise auszubauen bzw. die Leistungsfähigkeit mehrerer Werkzeuge miteinander zu kombinieren.

[16] Entsprechend § 3 Nr. 20 des Medizinproduktegesetzes [96] ist eine „Benannte Stelle“ eine für die Durchführung von Prüfungen und Erteilung von Bescheinigungen im Zusammenhang mit Konformitätsbewertungsverfahren nach Maßgabe der Rechtsverordnung nach § 37 Abs. 1 vorgesehene Stelle, die der Kommission der Europäischen Union und den Vertragsstaaten des Abkommens über den Europäischen Wirtschaftsraum von einem Vertragsstaat des Abkommens über den Europäischen Wirtschaftsraum benannt worden ist.

3.13 Zusammenfassung

Zusammenfassend werden in Tabelle 3-5 noch einmal alle Kriterien aufgelistet, anhand derer eine Lösung zur Beobachtung von SoCs zu beurteilen ist.

		Ziel
Vollständigkeit der Beobachtung	Taskwechsel	Vollständig
	Funktionen	Vollständig
	Basic Blocks	Vollständig
	Instruktionen	Vollständig
	Sprünge	Vollständig
	Datenzugriffe (CPU)	Vollständig
	Datenzugriffe (Peripherie)	Vollständig
	CPU-Register	Vollständig
	Cache	Vollständig
	Abarbeitung	Vollständig
	Zyklusgenauigkeit	Vollständig
	Bussystem	Vollständig
	Umgebungsbedingungen	Vollständig
	Ereignisse	Vollständig
Beobachtung von MPSoCs		Vollständig
Echtzeitfähigkeit		Ja
Kontinuität		Ja
Gleichzeitige Durchführung mehrerer Beobachtungsaufgaben		Ja
Beeinflussung des SoCs		Keine
Port-Ersetzung	Ausgang	Ja
	Eingang	Ja
Beobachtbarkeit von SoCs aus der Serienproduktion		Ja
Beobachtbarkeit in „realer“ Umgebung		Ja
Latenz		Keine
Flexibilität des Beobachtungsfokus		Ja
Klassifizierung für die Qualifizierung	IEC 61508	*offline support tool*, T2
	DO-330	Kriterium 2
	ISO 26262	TCL1
Kosten	Chipfläche / Ressourcen	Gering
	Implementierungsaufwand	Gering
	Entwicklungswerkzeuge	Gering
	Energiekosten	Gering
Universalität		Hoch
Skalierbarkeit		Hoch

Tabelle 3-5: Kriterien für die Qualität der Beobachtbarkeit

4 Stand der Technik

In diesem Abschnitt werden die wichtigsten aktuell verwendeten Methoden zur Beobachtung von SoCs erläutert und bewertet. Dabei wird zwischen der software-basierten und der hardware-basierten Beobachtung unterschieden. Es werden nur die Methoden ausführlich vorgestellt und diskutiert, die als Stand der Technik auch kommerziell verfügbar sind und die den in Abschnitt 3 definierten Kriterien an die Qualität der Beobachtbarkeit zumindest teilweise entsprechen.

4.1 Software-basierte Beobachtung

Bei der software-basierten Beobachtung wird zum funktionalen Programmcode zusätzlicher Code hinzugefügt, welcher zur Beobachtung des Programms dient. Diese Methode wird auch als Software-Instrumentierung bezeichnet.

Das Hinzufügen von Code kann manuell oder auch automatisiert erfolgen. Je nach Umfang der zu übermittelnden Informationen kann eine Instrumentierung des Codes zu erheblichen Leistungseinbußen (*application blow up*, *execution slow down*) führen. Grundsätzlich verändert jede Instrumentierung das zeitliche Verhalten des Programmablaufs. Dabei verändert nicht nur der für die Instrumentierung zusätzlich ausgeführte Code den Ablauf des Programms, sondern es muss auch die Beeinflussung des Caches sowie anderer Komponenten im SoC (z.B. Beeinflussung der Busarbitrierung aufgrund unterschiedlicher Zeiten zwischen Buszugriffen, zusätzliche Last auf Peripherieeinheiten wie Timer und serielle Schnittstellen) berücksichtigt werden. Größtenteils führt dies zu einer Verzögerung der Abarbeitung des Programms, durch besondere Effekte (z.B. im Cache) kann eine Instrumentierung aber auch zu einer Beschleunigung führen.

Die Bewertung der Beobachtung eines SoCs mittels Instrumentierung ist vom jeweiligen Beobachtungsziel abhängig. Im Wesentlichen lassen sich die Beobachtungsaufgaben in Instrumentierungen für Komponententests sowie für Integrations- und Systemtests unterteilen.

4.1.1 Instrumentierung für Komponenten-Tests

Bei Komponententests ergeben sich zwei wesentliche Beobachtungsaufgaben. Einmal muss die Funktionalität überprüft werden können, d.h. es muss protokolliert werden, ob beim Lauf eines Tests die Ausgabewerte einer Komponente den erwarteten Werten entsprechen.

Die zweite Beobachtungsaufgabe besteht darin, die im Abschnitt 2 diskutierte Testabdeckung zu protokollieren.

Oftmals ist es gewünscht, die Instrumentierung aus dem im Auslieferungszustand verwendeten Code wieder zu entfernen. Dabei muss aber berücksichtigt werden, dass der *Build*-Prozess [97] oder der Compiler [98] eventuell fehlerhaft arbeiten könnte. Es besteht durchaus die Möglichkeit, dass aus fehlerfreiem Quellcode Object-Code erzeugt wird, der Defekte beinhaltet.

Zur Lösung dieses Konflikts wird beispielsweise in der ISO26262-2011 [35], Teil 6, Absatz 9.4.5 vorgeschlagen, die funktionalen Tests nach Entfernung der für die Überdeckungsanalyse notwendigen Instrumentierung noch einmal vollständig zu wiederholen:

> *If instrumented code is used to determine the degree of coverage, it can be necessary to show that the instrumentation has no effect on the test results. This can be done by repeating the tests with non-instrumented code.*

Erfolgt die Beobachtung der Ergebnisse der funktionalen Tests mittels Instrumentierung, so muss diese dann konsequenterweise auch im Endprodukt verbleiben.

4.1.2 Instrumentierung für Integrations- und Systemtests

Wird die Instrumentierung für Laufzeit- und Performance-Messungen verwendet, so führt das eventuelle Entfernen der Instrumentierung zu einem veränderten Verhalten des Programms, d.h.

die Ergebnisse der mittels Instrumentierung erlangten Testergebnisse sind nicht mehr uneingeschränkt gültig.

In [14], Abschnitt 12.4 wird zur Notwendigkeit des Verbleibs der Instrumentierung im Code eine klare Aussage getroffen:

> *„Bei jeder Zeit-Messung muss beachtet werden, dass die Software im Feld völlig identisch zur vermessenen Software sein muss. Das heißt, eingefügte Instrumentierungspunkte zur Zeitmessung müssen im Code bleiben. Jede geringfügige Änderung, auch das Streichen von Code, macht neue Zeitmessungen notwendig, weil Cache-Effekte i.A. nicht vorhersehbar sind."*

4.1.3 Implementierungsvarianten

Um eine Applikation mit Instrumentierung zu versehen, sind mehrere Wege möglich.

Einmal wird der Quellcode manuell oder automatisiert um zusätzliche Anweisungen ergänzt, die die Protokollierung der Laufzeitinformation durchführen (Quelltextinstrumentierung).

Eine weitere Möglichkeit ist das automatisierte Hinzufügen der Instrumentierung auf Object-Code-Ebene (Bytecode-Instrumentierung, wird u.a. bei Java angewandt).

Eine dritte Möglichkeit zur Instrumentierung ist die instrumentierte Interpretation des Programms („*On-the-fly*"-Instrumentierung). Dies bietet sich bei Systemen mit ausgeprägter Laufzeitumgebung (Interpreter, z.B. Java VM) besonders an. Hier muss der eigentliche Programmcode nicht verändert werden, vielmehr übernimmt der Interpreter die Protokollierung der Laufzeitinformation.

Quelltextinstrumentierung

Die Ausgabe von Zustandsinformationen über eine Standard-Schnittstelle (z.B. `printf()` über einen UART) ist die einfachste Form der Software-Instrumentierung. Neben dem manuellen Einfügen kann die Instrumentierung von Programmcode auch automatisiert erfolgen. Dies kann vor dem Kompilieren (Patchen des Source-Codes) als auch nach dem Kompilieren durch das automatisierte Einfügen bzw. Verändern von Code erfolgen.

Instrumentierungen können an relevanten Stellen des Programms wie Funktionsaufrufen, Interrupts oder dem Zugriff auf bestimmte Speicherbereiche eingefügt werden.

Der Zugriff auf Speicher oder Peripherieeinheiten kann auch indirekt durch Aufrufe von Elementen einer Instrumentierungs-Bibliothek erfolgen, so dass die Zugriffe automatisiert protokolliert werden können. Durch Auswertung des Stacks kann die Stelle im Code identifiziert werden, welche den Zugriff initiiert hat. Dieses Verfahren ist für den Entwickler sehr bequem, erzeugt aber bei jedem protokollierten Zugriff einen erheblichen Overhead. Im Gegensatz zu Desktop-PC spielt dies bei ressourcenbeschränkten SoCs eine große Rolle.

Kommerzielle Werkzeuge zur automatisierten Instrumentierung von C und C++ Code sind z.B. BullseyeCoverage (Bullseye), CodeWarrior (Freescale), CTC++ (Testwell), Insure++ (Parasoft) Cantata++ (IPL), Squish Coco (froglogic) und TCAT/C-C++ (Software Research).

Eine beispielhafte Instrumentierung ist in Tabelle 4-1 (aus [99], Figure 2.1 und Figure 2.8) dargestellt. Dabei wird eine Funktion von dem Werkzeug automatisch so instrumentiert, dass ein Bedingungsüberdeckungstest ausgeführt werden kann. Die vergrößerte Code-Menge sowie der zusätzliche RAM-Bedarf sind leicht ersichtlich. Nicht dargestellt ist die noch notwendige Auswertung der Status-Variablen `inst`.

Originaler Source-Code	Instrumentierter Code (für den Bedingungsüberdeckungstest)

`void foo()`
`{`
`  bool found=false;`
`  for (int i=0; (i<100) && (!found); ++i)`
 	
`  {`
`  if (i==50) break;`
 	
`  if (i==20) found=true;`
 	
`  if (i==30) found=true;`
 	
`  }`
`  printf("foo\n");`
 	
`}`
	`char inst[15];`
	`void foo()`
	`{`
	`  bool found=false;`
	`  for (int i=0;((i<100)?inst[0]=1:inst[1]=1,0)`
	`&&`
	`  ((!found)?inst[2]=1:inst[3]=1,0); ++i)`
	
	`  {`
	`  if ((i==50?inst[4]=1:inst[5]=1,0))`
	`  { inst[6]=1; break;`
	`  }`
	`  if ((i==20?inst[7]=1:inst[8]=1,0))`
	`  { inst[9]=1; found=true;`
	`  }`
	`  if ((i==30?inst[10]=1:inst[11]=1,0))`
	`  { inst[12]=1; found=true;`
	`  }`
	`  inst[13]=1;`
	`  }`
	`  printf("foo\n");`
	`  inst[14]=1;`
	`}`

Tabelle 4-1: Beispiel einer Software-Instrumentierung

Instrumentierung des Object-Codes

Die automatisierte Instrumentierung des Object-Codes bietet den großen Vorteil, dass keine spezielle instrumentierte Variante des Source-Codes erforderlich ist.

Manche Compiler sind in der Lage, während der Compilierung automatisch eine Instrumentierung des erzeugten Object-Codes vorzunehmen. So kann beispielsweise der Open Source Compiler `gcc` mit dem Aufrufparametern `-fprofile-arcs -ftest-coverage` dazu veranlasst werden, zusätzliche Informationen über den Kontrollflussgraphen des Programms auszugeben sowie den erzeugten Object-Code zu instrumentieren. Dies kann dann von dem ebenfalls zur GCC Suite gehörendem Programm `gcov` genutzt werden, um die Testüberdeckung zu analysieren.

Ein weiteres Beispiel für eine automatisierte Instrumentierung des Object-Codes ist in Abbildung 4-1 angegeben (nach [100]).

Hardware-unterstützte Code-Instrumentierung

Um die eigentliche Anwendung von Kommunikationsaufgaben zu entlasten, können mittels geeigneter Hardware die Ergebnisse der Instrumentierung zum Entwicklungsrechner übertragen werden. Beispiele für eine solche Hardware-Unterstützung ist die *System Trace Macrocell* und die *Instrumentation Trace Macrocell* (beides Bestandteile der ARM CoreSight Architektur [101]) sowie die *Data Akquisition Messages* (TCODE = 7) einer Nexus-Implementierung [102].

Die Anwendung (bzw. die automatisch eingefügte Instrumentierung) schreibt dazu Daten in sogenannte Stimulationsregister, die Hardwarekomponente versieht diese mit einem Zeitstempel und generiert daraus Trace-Pakete, die über eine geeignete Trace-Schnittstelle ausgegeben werden.

Gegenüber der Ausgabe von Zustandsdaten über einfache Schnittstellen wie UARTs werden die generierten Datenpakete über einen schnellen Trace-Port ausgegeben. Nachteil dieser schnellen Ausgabe ist die notwendige aufwändige Dekodierung der Datenpakete. Alternativ ist auch eine Ausgabe in den *Embedded Trace Buffer* (ETB) möglich, welcher über den Debug-Port ausgelesen werden kann. Dieser Zugriff ist aber deutlich langsamer als die direkte Ausgabe über den Trace-Port.

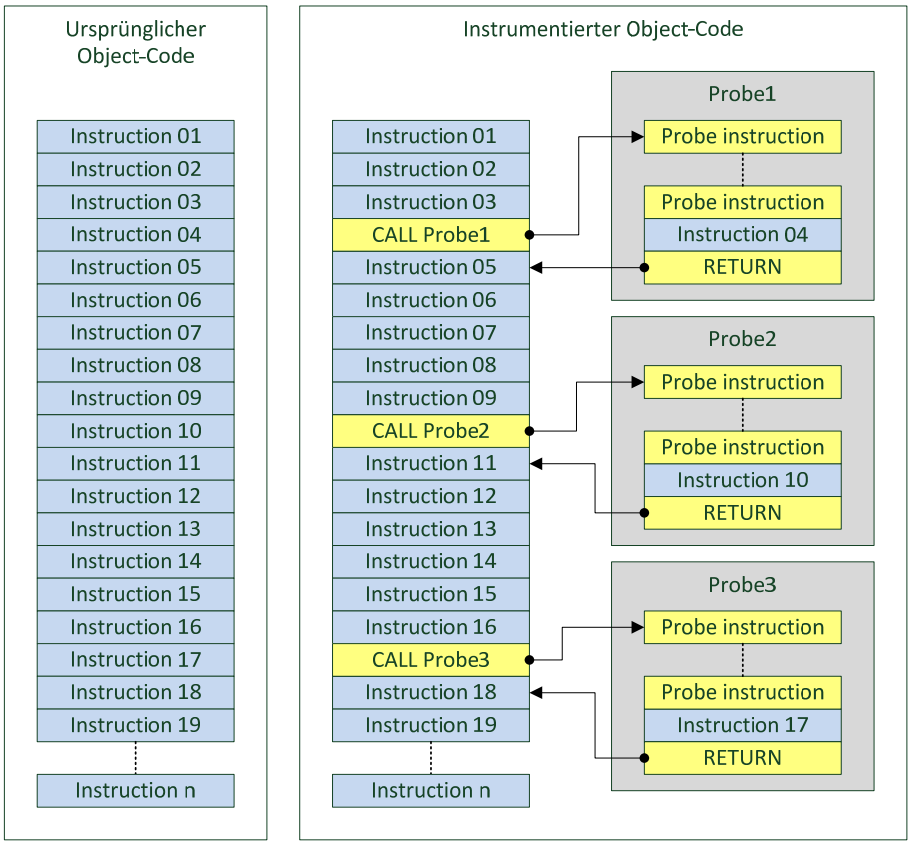


Abbildung 4-1: Object-Code Instrumentierung

4.1.4 Bewertung

Im Folgenden wird die Beobachtung eines SoCs mittels Software-Instrumentierung anhand der in Abschnitt 3 diskutierten Kriterien bewertet. Diese Bewertung ist exemplarisch und muss im Einzelfall an die Möglichkeiten des zu untersuchenden SoCs und an die Anforderungen der Beobachtungsaufgabe angepasst werden.

Einfluss auf die Bewertung hat die Frage, ob für den konkreten Anwendungsfall ein Verbleib der Instrumentierung in der finalen Software erforderlich ist oder ob diese wieder entfernt werden darf. Wie bereits diskutiert, kann letzteres für die Instrumentierung der Überdeckungsanalysen im Rahmen von Komponententests gelten, so dass hier auch ein größerer Overhead für die Instrumentierung akzeptabel ist. Für funktionale Tests und alle Beobachtungen, deren Resultat von Funktionalität, Ausführungszeiten und Ressourcenverbrauch abhängig ist, sind die Beobachtungsergebnisse nur zuverlässig, wenn die Instrumentierung im finalen Code verbleiben kann.

Vollständigkeit der Beobachtung

Bei der Betrachtung der Vollständigkeit der Beobachtung lassen sich für die Software-Instrumentierung drei Varianten unterscheiden:

- pInstr: Permanente Instrumentierung verbleibt im finalen Code
- tInstr: Temporäre Instrumentierung wird aus dem finalen Code entfernt
- ptInstr: Kombination aus permanenter und temporärer Instrumentierung

pInstr: Für alle Beobachtungen, die vom zeitlichen Verhalten oder vom Ressourcenverbrauch abhängig sind, sollte die Instrumentierung im finalen Code verbleiben. Dazu gehören:

Beobachtung von CPU-Registern (z.B. Stackpointer)

Beobachtung des Caches (z.B. mittels Abfrage spezieller Statusregister)

tInstr: Bedingt durch den großen Aufwand für die Beobachtung von *basic blocks*, Instruktionen und Sprüngen muss die hierfür notwendige Instrumentierung in den meisten Fällen wieder entfernt werden. Diese wird beispielsweise für Überdeckungsanalysen benötigt, die während der Durchführung funktionaler Tests vorgenommen werden.

Der Verbleib der Instrumentierung für die Beobachtung von Datenzugriffen der CPU(s) ist meist nur möglich, wenn dieser nur für eine stark reduzierte Auswahl implementiert ist.

ptInstr: Der Aufwand für die Beobachtung von Taskwechseln und Funktionsaufrufen mittels Software-Instrumentierung ist relativ gering, daher kann diese in vielen Fällen auch im finalen Code verbleiben. Bei Verfügbarkeit geeigneter Interfaces können Umgebungsbedingungen (z.B. Temperatur, Versorgungsspannung) mit protokolliert werden. Zur Beobachtung der Stromaufnahme, von Werten analoger Eingangssignale oder der Interpretation verschiedener serieller Schnittstellen müssen externe Geräte eingesetzt werden, deren Aufzeichnungen dann mit der Ausgabe der Instrumentierung korreliert werden müssen.

Viele interne Zustände sind ohne spezielle Hardware-Unterstützung für die CPU (die den instrumentierten Code abarbeitet) nicht sichtbar, so dass beispielsweise für die folgenden Beobachtungsaufgaben in den allermeisten Fällen keine der diskutierten Formen der Software-Instrumentierung geeignet ist:

- zyklusgenaue Beobachtung der Ausführung von *basic blocks* oder Instruktionen
  (Die CPU kann nicht die Anzahl der Takte für die Abarbeitung einer bestimmten Instruktion bestimmen, da diese Messung durch die Ausführung des Instrumentierungs-Codes beeinflusst wird und außerdem ein Vielfaches an zusätzlicher Rechenzeit benötigen würde)
- Beobachtung der Art der Abarbeitung (z.B. spekulativ, *out-of-order*)
  (Die CPU kann üblicherweise und mit vertretbarem Aufwand nicht ihre eigene Pipeline beobachten.)
- Beobachtung von Bus-Systemen (z.B. DMA-Operationen busmasterfähiger Peripherieeinheiten)
  (Die CPU kann ohne zusätzliche Hardware-Unterstützung keine Busoperationen in einem ausreichenden Detaillierungsgrad beobachten.)

Ereignisse

Da bei der Beobachtung von Ereignissen auch das zeitliche Verhalten eine Rolle spielt, sollte auch hier die Instrumentierung im finalen Code verbleiben. Aus Sicht der CPU sind einige Ereignisse beobachtbar, wie z.B. der Aufruf von Interrupt-Service-Routinen. Für die Beobachtung anderer Ereignisse, die für die CPU(s) nur indirekt oder gar nicht sichtbar (z.B. Interrupt-Anforderungen, DMA-Transfers busmasterfähiger Peripherieeinheiten), aber beispielsweise für die Analyse von Wettlaufsituationen hilfreich sind, ist das Verfahren der Software-Instrumentierung im Allgemeinen nicht geeignet.

Beobachtbarkeit von MPSoCs

Jeder Core für sich allein kann mit der Methode der Software-Instrumentierung mit den bereits diskutierten Grenzen beobachtet werden.

Bei der gleichzeitigen und synchronisierten Beobachtung mehrerer Cores spielt das zeitliche Verhalten eine wichtige Rolle, so dass die Instrumentierung auch hier im finalen Code verbleiben sollte. Eine große Schwierigkeit stellt hier die Synchronisierung dar, d.h. die einzelnen Ausgaben der Instrumentierung müssen mittels einer gemeinsamen Zeitbasis zueinander in Beziehung gesetzt

werden. Ohne spezielle Unterstützung ist dies sehr aufwändig, hier kann die hardwareunterstützte Instrumentierung (wenn verfügbar) äußerst hilfreich sein.

Echtzeitfähigkeit und Kontinuität

Für eine fortwährende Beobachtung eines SoCs mit unverminderter Taktfrequenz ist das Verfahren der SW-Instrumentierung sehr gut geeignet, wobei auch hier die Instrumentierung im finalen Code verbleiben sollte.

Gleichzeitige Durchführung vieler Beobachtungsaufgaben

Eine entsprechende Instrumentierung vorausgesetzt, kann diese beliebig viele Beobachtungsaufgaben ausführen. Limitierend sind hier nur die Bandbreite zur Ausgabe der Beobachtungsergebnisse und die Größe des Programmspeichers.

Beeinflussung des SoCs

Die Beeinflussung des Programmablaufs in einem SoC im Gegensatz zu einem Programmablauf ohne Instrumentierung ist abhängig von den jeweiligen Beobachtungsaufgaben. Eine geringe Beeinflussung ergibt sich, wenn beispielsweise im Falle eines unerwarteten Programmverhaltens (z.B. Fehlerbehandlungsroutine) zusätzlich instrumentierter Code ausgeführt wird. Wird die Instrumentierung aber für die Messung von Ausführungszeiten oder Ressourcenverbrauch eingesetzt, so erfolgt eine deutliche Beeinflussung.

Beobachtbarkeit von SoCs aus der Serienproduktion

Da die Software-Instrumentierung als Teil der Anwendung ausgeführt wird, ist keine spezielle Hardwareunterstützung notwendig, so dass SoCs aus der Serienproduktion verwendet werden können. Für die Verwendung einschränkend könnte sich die mit der Instrumentierung einhergehende Vergrößerung des Speicherbedarfs auswirken, die dazu führen kann, dass der modifizierte Programmcode zu groß für den Speicherplatz im Zielsystem ist. Auch die eventuelle Anforderung an eine hardwareunterstützte Instrumentierung kann den Einsatz von speziellen SoC-Varianten (Evaluation-Chips) erforderlich machen.

Beobachtbarkeit in „realer" Umgebung

Für die Beobachtung eines SoCs in seiner realen Betriebsumgebung ist das Verfahren der SW-Instrumentierung sehr gut geeignet, wobei auch hier die Instrumentierung im finalen Code verbleiben sollte.

Flexibilität des Beobachtungsfokus

Falls sich bei einer Quelltextinstrumentierung nachträglich die Forderung nach einem Wechsel des Beobachtungsfokus ergibt, ist eine erneute Compilierung notwendig. Besonders bei der Suche nach der Ursache von nichtdeterministisch auftretendem Fehlverhalten sind hier häufige Änderungen der Instrumentierung notwendig, um verschiedene Hypothesen im Rahmen des Debug-Prozesses überprüfen zu können. Abhängig vom Zeitpunkt des Auftretens des Fehlverhaltens ist eine entsprechende Änderung des Programmcodes nicht ohne weiteres möglich, beispielsweise erfordert die nachträgliche Änderung sicherheitskritischer Software (z.B. im Luftfahrt-Bereich), die sich bereits im produktiven Einsatz befindet, das jeweils erneute Durchlaufen eines aufwändigen Zertifizierungsprozesses.

Latenz

Die Latenz zwischen einem zu beobachtenden Ereignis und der Verfügbarkeit der Beobachtungsergebnisse ist von der Art und dem Ort der Auswertung der Beobachtungsergebnisse und der Übertragungsschnittstelle abhängig. Ist die Auswertung Bestandteil der Instrumentierung, so sind SoC-intern die Beobachtungsergebnisse schnell verfügbar, müssen aber über eine geeignete Schnittstelle (z.B. UART) dem Beobachter bekannt gemacht werden. Oftmals werden die Beobachtungsergebnisse im SoC zwischengespeichert und nach Erreichen eines bestimmten Zwischenergebnisses

an den Beobachter übermittelt. Damit ergibt sich oftmals eine relativ große und variable Latenz, so dass der Beobachter auf Vorgänge im SoC (z.B. ein Taskwechsel) nicht zeitnah reagieren kann.

Qualifizierung des Beobachters

Die Instrumentierung der Software ist ein Prozess, in dem diese um zusätzlichen ausführbaren Code erweitert wird, welcher in vielen Fällen auch im finalen Produkt verbleiben muss. Das automatisierte Hinzufügen von Code bringt bei sicherheitskritischen Anwendungen hohe Anforderungen an die Werkzeug-Qualifizierung mit sich. Dabei muss besonders beachtet werden, dass die finale Version des Object-Codes (nur permanente Instrumentierung) sich von der getesteten Version (permanente und temporäre Instrumentierung, Testumgebung) unterscheidet und man sich somit mit folgenden Fragestellungen auseinandersetzen muss (Abbildung 4-2):

- Ist der Prozess des Einfügens der temporären Instrumentierung und der Testumgebung zuverlässig?
- Sind die Build-Prozesse zuverlässig?
- Entsprechen die Testläufe dem produktiven Programmlauf?

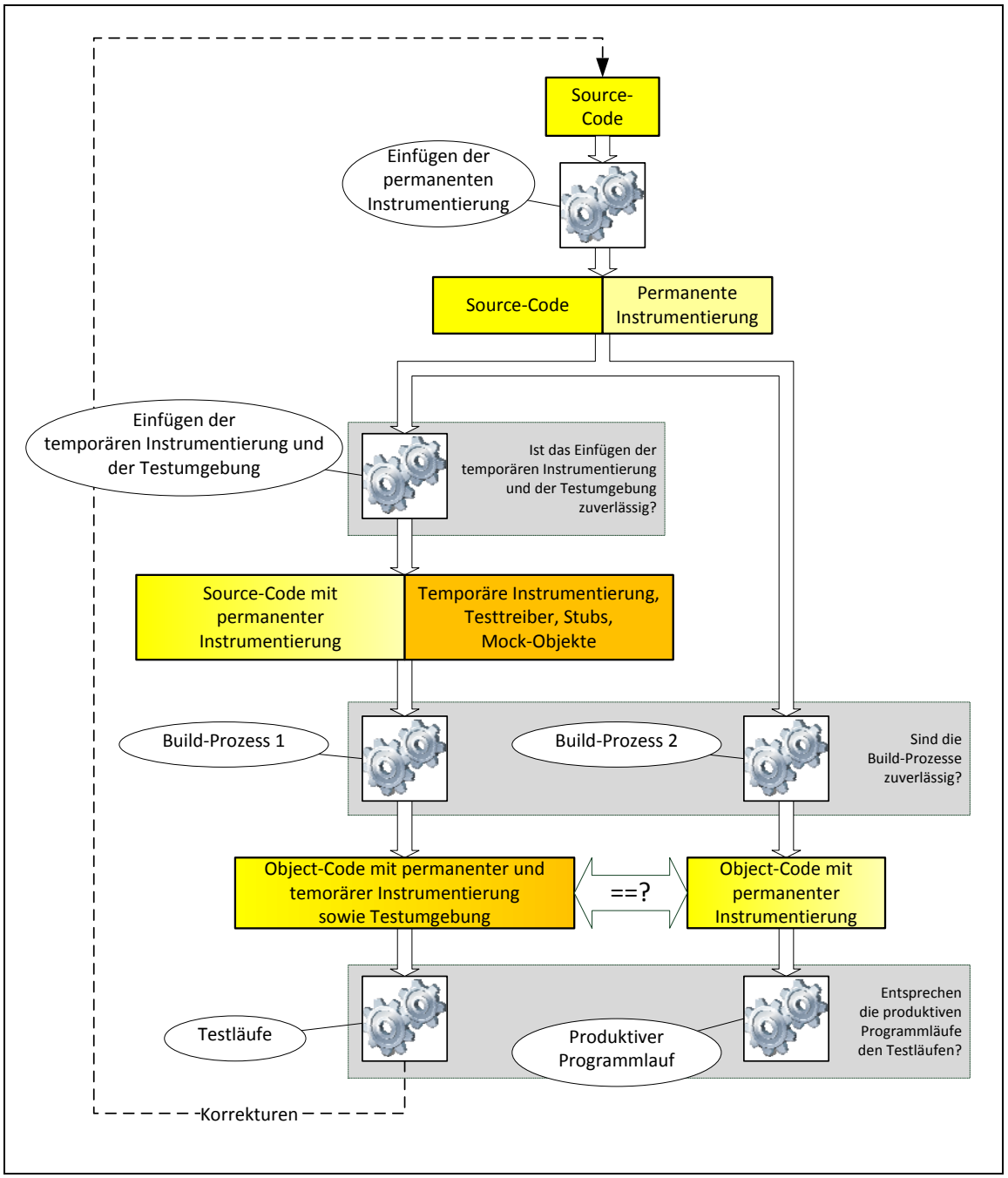


Abbildung 4-2: Unterschiede zwischen getestetem temporär instrumentierten Code und dem finalen Code

Die folgenden Klassifikationen sind als Orientierung zu verstehen und müssen jeweils abhängig vom Kontext des Werkzeugeinsatzes individuell bestimmt werden.

Für die IEC 61508 [86] ergibt sich die Einordnung der Software-Instrumentierung *online support tool*, da diese eine direkte Beeinflussung des sicherheitskritischen Systems zur Laufzeit bewirken kann.

Bei der Klassifizierung entsprechend DO-330/ED-215 [49] [93] ergibt sich das Kriterium 1, da die Ausgabe des Werkzeugs Bestandteil der resultierenden Software ist und hier Fehler einbringen könnte.

Für den Automobil-Bereich erfolgt die Einstufung nach ISO 26262:2011 [35] mit einem *Tool Impact Level* von TI1, da bei der Software-Instrumentierung die Möglichkeit besteht, dass fehlerhafter

Code in das Endprodukt eingebracht wird. Oftmals wird dazu ein mittleres Vertrauen gewählt [103], dass eine Fehlfunktion des Werkzeugs verhindert oder vor dem Inverkehrbringen eines Produkts entdeckt wird. Daraus folgt ein *Tool Error Detection Level* von TD2, was dann zu einem resultierenden *Tool Confidence Level* von TCL2 führt und damit eine Qualifizierung des Werkzeugs notwendig macht.

Kosten

Bei der Bestimmung der Kosten der Software-Instrumentierung sind die für die Instrumentierung im finalen Produkt zu reservierenden Ressourcen zu berücksichtigen. Dies können einmal direkte Kosten für die unterstützende Hardware (z.B. *System Trace Macrocell)*, andererseits die Kosten der für die Instrumentierung verwendeten Ressourcen (Speicher, Rechenzeit, Kommunikationsschnittstelle, Energie, Energiebereitstellung) sein. Falls diese ohne SW-Instrumentierung nicht benötigt würden, könnte ggfs. eine kleinere und langsamere Version des SoCs verwendet werden. Die Kosten fallen besonders bei großen Stückzahlen ins Gewicht und können dabei so erheblich sein, dass das Risiko eingegangen wird, die Instrumentierung zu entfernen und auf die nicht nachweisbare Fehlerfreiheit eines neuen Rekompilierungsprozesses zu vertrauen.

Geht man beispielsweise von 100 SoCs in höherwertigen Automobilen aus, deren durchschnittliche Leistungsaufnahme bei 500mW liegt, so verursacht die Energiebereitstellung für einen Instrumentierungs-Overhead von 10% zusätzliche Kosten in Höhe von 2,50€ bis 5,00€. Multipliziert mit der Anzahl produzierter Automobile ergeben sich schnell sehr hohe Summen, die es attraktiv erscheinen lassen, eine Entscheidung gegen permanente Code-Instrumentierung zu treffen.

Zusätzlich fallen noch Kosten für die Software-Werkzeuge an, mit deren Hilfe die automatisierte Instrumentierung vorgenommen wird. Besonders bei sicherheitskritischen Anwendungen kommen dazu außerdem noch die mit der Werkzeug-Qualifizierung verbundenen Kosten, wobei hier durch die Modifikation des im finalen Produkts verbleibenden Codes besonders hohe Anforderungen an die Qualifizierung gestellt werden (IEC61508: *online support tool*, DO-330: TQL1 bis TQL4, ISO26262: TCL2).

4.2 Hardware-basierte Beobachtung

Als Alternative zur Software-Instrumentierung existieren auch Beobachtungsverfahren, die mittels geeigneter Hardware-Unterstützung eine weitgehend nicht-intrusive Beobachtung eines SoCs erreichen.

Bond-Out-Chips

Die ersten Emulationssysteme basierten auf Bond-Out-Chips. Dies waren spezielle Implementierungen eines SoCs, bei dem für die Beobachtung relevante Informationen an zusätzliche I/O-Pins geführt wurden. Mittels spezieller Sockel wurden dann die eigentlichen I/O-Pins mittels eines „Probe Cables“ mit dem Zielsystem verbunden. Der Bond-Out Chip ersetzte damit den zu beobachtenden SoC (Abbildung 4-3).

Die Beobachtbarkeit des Zielsystems kann – entsprechend der Leistungsfähigkeit der dafür im Bond-Out-Chip implementierten Strukturen – sehr umfassend sein.

Üblicherweise werden Instruktions- und Daten-Trace verfügbar gemacht.

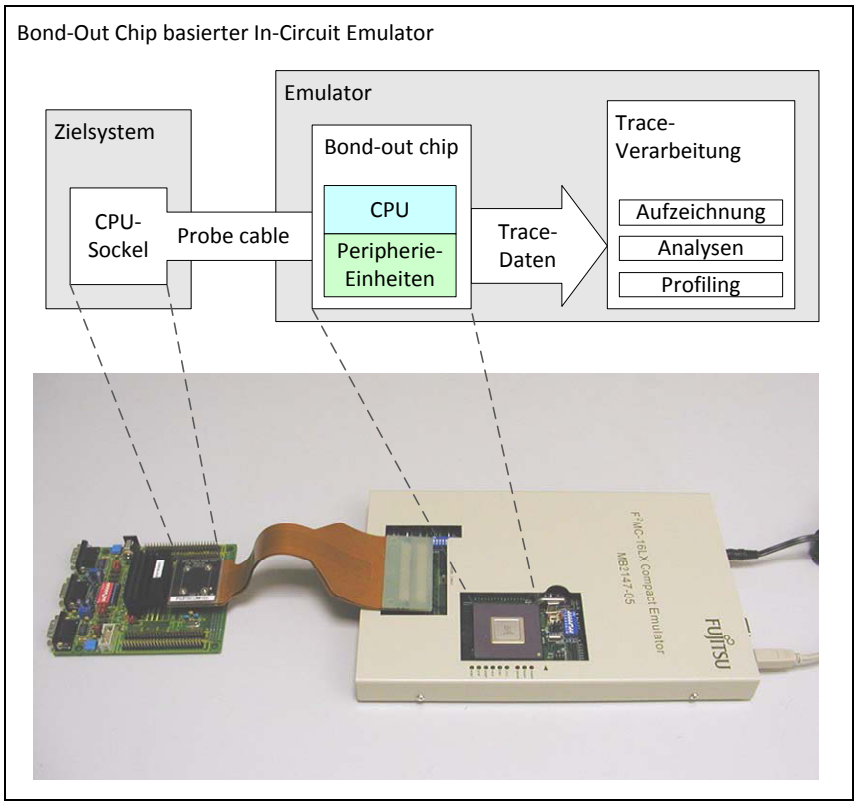


Abbildung 4-3: Beobachtung eines Zielsystems mit einem Bond-Out Chip [17]

Der Bond-Out Chip muss alle Peripherieeinheiten umfassen, die im Serien-SoC ebenfalls verfügbar sind. Aufgrund der großen Variabilität der Peripherie-Ausstattung der Serien-SoCs ist es aus Kostengründen nicht möglich, zu jedem SoC passend einen Bond-Out Chip anzubieten. Vielmehr werden Bond-Out-Chips mit einer maximal möglichen Peripherieausstattung bereitgestellt, welche mindestens die Funktionalität jedes Mitglieds einer SoC-Familie beinhalten. Ergibt sich während der Lebensdauer einer SoC-Familie die Notwendigkeit, auf Kundenwunsch eine erweiterte Peri-

[17] Wiedergabe des Fotos mit freundlicher Genehmigung des Rechte-Inhabers Fujitsu Semiconductor Europe GmbH / Spansion International Inc.

pherieausstattung zu implementieren, wird diese nicht durch den regulären Bond-Out-Chip unterstützt. Falls der Kunde die fehlende Debug-Unterstützung an dieser Stelle nicht akzeptiert, muss vom Hersteller des SoCs ein neuer Bond-Out-Chip bereitgestellt werden. Gleiches gilt auch für den Fall von Fehler-Korrekturen auf dem SoC, der Weiterentwicklung bestehender Peripherieeinheiten oder dem Wechsel der Herstellungstechnologie und damit einhergehender Änderung des zeitlichen Verhaltens.

Eine weitere Einschränkung bei der Anwendbarkeit von Bond-Out-Chips besteht darin, dass diese oftmals in einer anderen Technologie hergestellt werden als die im produktiven Einsatz verwendeten SoCs. Daraus resultiert ein unterschiedliches zeitliches Verhalten, außerdem kann es erforderlich sein, dass Peripherieeinheiten (z.B. Analog-Digital-Wandler, Businterfaces) sowie auf dem SoC implementierte Speicher (z.B. Flash) komplett anders anzusprechen sind. Aufgrund dieser Unterschiede ist es für einen Hersteller von SoCs extrem schwierig zu garantieren, dass der für Tests und eventuelle Zertifizierungen benutzte Bond-Out-Chip ein identisches Verhalten zu den später im produktiven Einsatz verwendeten SoCs hat.

Da die Bond-Out-Chips via Sockel und ggfs. Adapterkabel mit dem Zielsystem verbunden sind, ergeben sich bei höheren Signalfrequenzen auf den I/O-Pins zunehmend physikalische Probleme. Die kapazitive Last der Leitungen und Sockel und die damit einhergehende Verfälschung der Signale ist so groß, dass bei Signal-Frequenzen ab dem Bereich um 100 MHz Bond-Out-Chips nicht mehr eingesetzt werden können.

Logikanalysator

Bei Mikroprozessoren mit extern angeschlossenem Speicher kann der Programmablauf durch Beobachtung des externen Busses mittels Logikanalysator verfolgt werden (Abbildung 4-4).

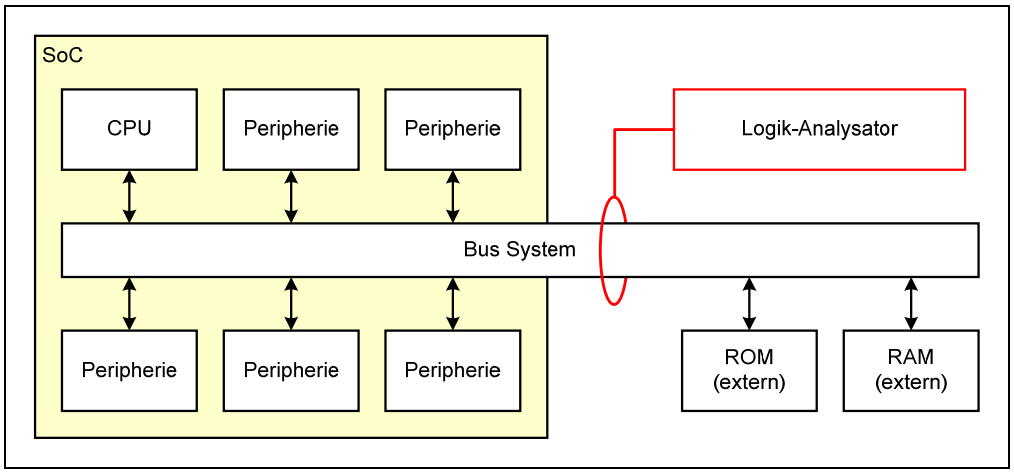


Abbildung 4-4: Beobachtung eines SoCs mittels Logik-Analysator

Dazu werden die Bussignale mit speziellen Adaptern an den I/O-Pins des Mikroprozessors oder des Speicherbausteins abgegriffen. Wenn die Leiterplatte entsprechend vorbereitet wurde, können auch Steckverbinder (z.B. Samtec, Mictor) bzw. Anpressadapter (Abbildung 4-5) zum Anschluss des Logikanalysators verwendet werden. Letztere habe den Vorteil, dass die Signale des Busses nur minimal belastet werden.

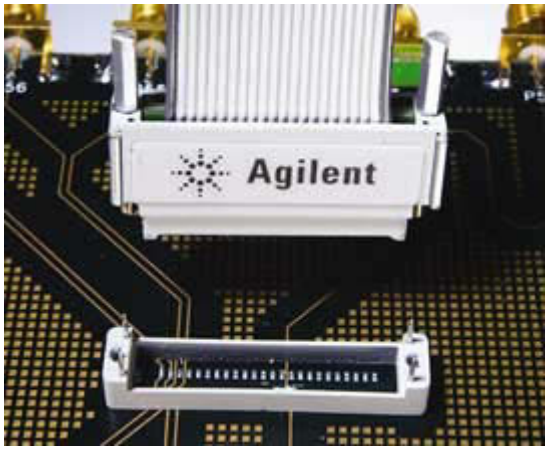


Abbildung 4-5: Anpressadapter eines Logikanalysators[18]

Ein eventuell vorhandener Cache stellt für diese Methode eine Einschränkung dar, da Zugriffe auf den Programmspeicher am externen Bus nur sichtbar sind, wenn der betroffene Code nicht bereits im Cache steht. Sollen alle Zugriffe sichtbar gemacht werden, muss der Cache abgeschaltet werden, was aber das zeitliche Verhalten des Programms massiv beeinflusst. Gleiches gilt für einen eventuellen Daten-Cache. Auch dieser muss abgeschaltet bzw. auf „*write-through*“ gestellt werden, um die Sichtbarkeit von Zugriffen auf den Datenspeicher zu verbessern.

Da moderne SoCs meist über leistungsfähige interne Speicher verfügen und diese über interne Busse mit sehr hoher Bandbreite an die CPU angeschlossen sind, hat die Beobachtung des Programmablaufs durch Betrachtung des externen Busses mit einem Logikanalysator kaum noch Bedeutung.

Doch auch hier gibt es Ausnahmen: Manche Mikroprozessoren lassen sich in einen speziellen Debug-Modus schalten, bei dem die auf dem internen Bus stattfindenden Transfers auf dem externen Businterface sichtbar gemacht werden. Ein Beispiel hierfür ist der i.MX53 (ARM Cortex A8 [81] @ 1,2GHz) von Freescale, bei dem neben diversen Debug-Signalen auch Signale des AXI-Busses (z.B. Arbitrierung, Master-ID, Priorität, Zugriffstyp, Adresse) auf I/O-Pins geroutet werden können ([104], Kapitel 6.7 und 49.1.3.3). Mit dieser Methode können interne Signale sichtbar gemacht werden, die durch die herkömmlichen Trace-Schnittstellen nicht beobachtbar sind.

„embedded Trace“-Lösungen

Um auch für schnellere SoCs eine Trace-Unterstützung anbieten zu können sowie um die dargestellten weiteren Limitationen beim Einsatz von Bond-Out-Chips zu umgehen, werden in aktuellen SoCs fast nur noch die im Folgenden dargestellten „embedded Trace“-Lösungen verwendet. Dazu werden die zur Erfassung von Trace-Daten erforderlichen Strukturen direkt in SoCs integriert (Abbildung 4-6).

[18] Wiedergabe des Fotos mit freundlicher Genehmigung des Rechte-Inhabers Agilent Technologies

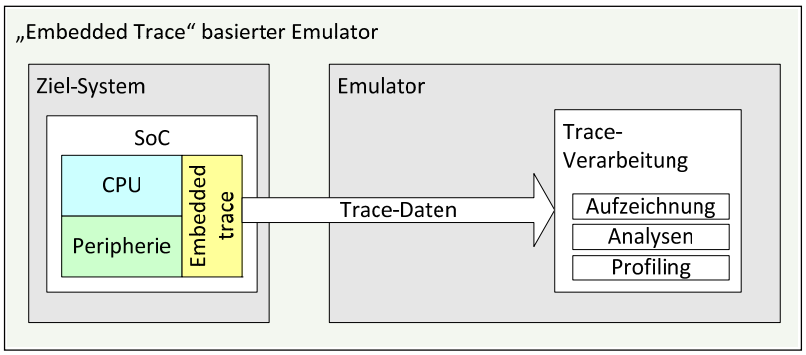


Abbildung 4-6: Prinzip von „embedded Trace“-Lösungen

Die Erfassung von Trace-Daten erfolgt mit zwei unterschiedlichen Strategien (Abbildung 4-7). Einmal können auf dem SoC die Trace-Daten gespeichert und über eine geeignete Schnittstelle später ausgelesen werden (On-Chip Trace-Speicher), zum anderen werden die ggfs. gepufferten Trace-Daten zeitnah ausgegeben und außerhalb des SoCs zur Weiterverarbeitung gespeichert (Off-Chip Trace-Speicher). Aufgrund des begrenzten Speicherplatzes auf dem SoC werden die On-Chip-Lösungen nur für die Beantwortung sehr einfacher Fragestellungen verwendet, der Schwerpunkt liegt bei den Off-Chip-Lösungen.

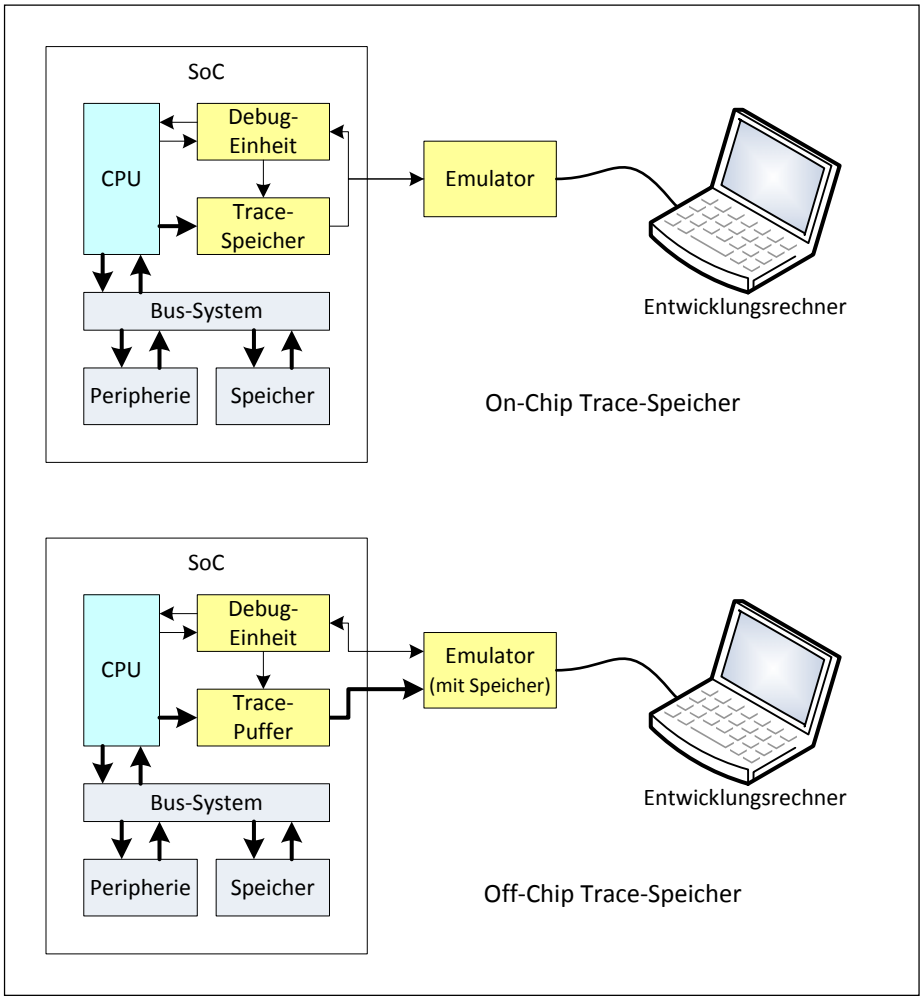


Abbildung 4-7: On-Chip und Off-Chip Trace Speicher

Für die Beobachtung einer CPU ist die Kenntnis von Sprunginformationen (Programm-Trace) sowie erfolgtem Daten-Transfer (Daten-Trace) relevant. Die Häufigkeitsverteilung dieser Operation ist von der CPU-Architektur, dem Compiler, den eingestellten Compileroptimierungen sowie auch vom Programmierstil abhängig. So lässt sich beispielsweise die Auswertung der Bedingung `((a||b)&&c)` mit drei Sprüngen realisieren, andererseits kann aber auch die Bedingung arithmetisch ausgewertet werden, so dass nur ein Sprung erforderlich ist.

Ein Beispiel einer typischen Häufigkeitsverteilung von Instruktionen ist in Tabelle 4-2 (80x86 Architektur, aus einem Mittelwert von fünf SPECint2000 Benchmarks, aus [105]) angegeben.

Instruktion	Häufigkeit	Relevanz
Load	22%	Daten-Trace
Store	12%	
Conditional branch	20%	Programm-Trace
Call	1%	
Return	1%	
Andere	44%	

Tabelle 4-2: Häufigkeitsverteilung von Instruktionen

Um die für die Übertragung der Trace-Daten erforderliche Bandbreite so gering wie möglich zu halten, werden Kompressionsverfahren angewendet. Diese zielen auf die Verminderung des mittleren Bandbreitenbedarfs. Für die kontinuierliche Aufzeichnung eines Traces ist aber die Berücksichtigung von Worst-Case-Szenarien erforderlich, da bei ungünstigen Abfolgen von Sprüngen und Zugriffen auf Daten die benötigte Bandbreite deutlich steigen kann. Aus diesem Grund muss bei der Auslegung der Trace-Schnittstelle ausreichend Puffer und Reserve mit eingeplant werden. In Abbildung 4-8 ist der Bandbreitenbedarf für verschiedene CPUs angegeben (Daten aus [106]), wobei besonders die großen Unterschiede zwischen einem nicht zyklusgenauen Instruktions-Trace (IT) und einem zyklusgenauen Instruktions-Trace (zgIT) erkannt werden können.

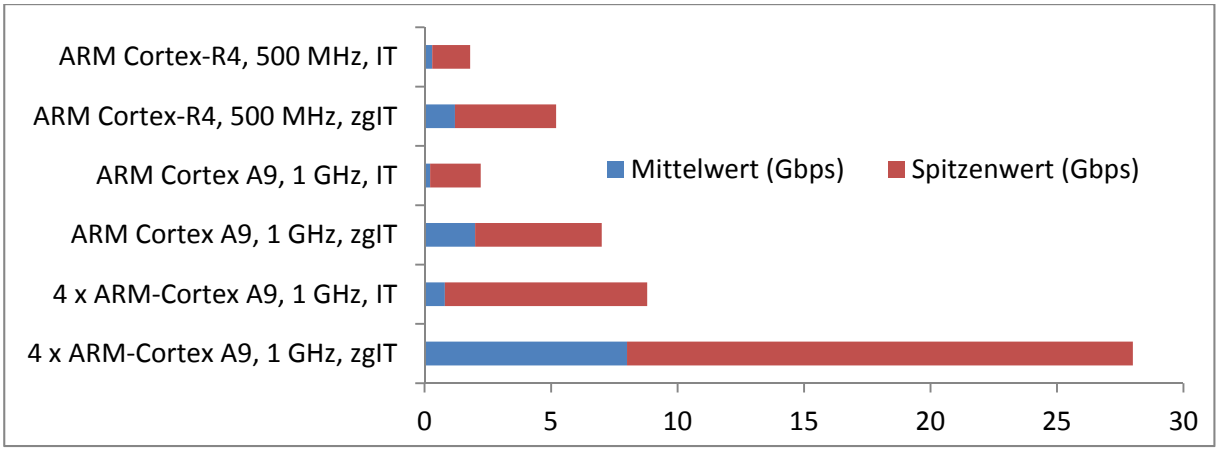


Abbildung 4-8: Unterschiedlicher Bandbreitenbedarf für zyklusgenaue und nicht zyklusgenaue Traces

Falls die Ausgabe der Trace-Daten eine zu hohe Bandbreite erfordert (besonders bei Spitzenwerten), kann die „embedded Trace"-Einheit so konfiguriert werden, dass sie entweder Trace-Nachrichten verwirft oder das System stoppt (*stall*), um die Generierung weiterer Nachrichten zu verhindern und eine lückenlose Beobachtung zu ermöglichen. Informationen über Lücken im Trace-Datenstrom sowie des kurzzeitigen Stoppens einzelner CPUs werden ebenfalls in Form von Trace-

Nachrichten an das Beobachtungswerkzeug übermittelt. Besonders letzterer Ansatz ist sehr kritisch zu sehen, da hier eine massive, nichtdeterministische Beeinflussung des zu beobachteten SoCs stattfindet.

4.2.1 Programm-Trace

Die Ausgabe von Informationen über den Programmablauf in einem SoC ist die wohl wichtigste Trace-Information. Dazu werden im Wesentlichen Sprunginformationen (d.h. stattgefundenen Änderungen des Kontrollflusses) ausgegeben. Das Entwicklungssystem ist dann in der Lage, anhand des Object-Codes und der vom Compiler ausgegebenen Debug-Informationen die ausgeführten Instruktionen zu rekonstruieren.

Es lassen sich direkte und indirekte Sprünge unterscheiden.

Bei direkten Sprüngen ergibt sich der Sprungoffset direkt aus dem Object-Code. Für die ARMv7 Architektur [107] sind dies beispielsweise die folgenden Befehle:

- B (*Branch to target address*)
- BL, BLX <immediate> (*Call a subroutine*)

Die Sprungziele indirekter Sprünge werden erst zur Laufzeit des Programms bekannt.

Die Häufigkeit des Auftretens von direkten und indirekten Sprüngen lässt sich nicht exakt angeben, da diese von vielen Parametern wie Quellcode, CPU-Architektur, Compiler und eingestellten Compileroptimierungen abhängig ist. Die Übermittlung der Sprunginformationen geschieht optimiert, d.h. es wird nicht die gesamte Adresse übertragen, sondern nur der Adressoffset.

Bei der „traditionellen“ Ausgabe von Sprunginformationen werden die durch direkte und indirekte Sprünge bewirkten Adressänderungen übertragen.

Bei direkten Sprüngen ist es nicht notwendig, die Zieladresse des Sprunges zu übermitteln, da diese im Entwicklungssystem bestimmt werden kann. Wird bei direkten Sprüngen nur die Information übertragen, ob dieser Sprung stattgefunden hat oder nicht, kann die zur Beobachtung des Programmablaufs erforderliche Trace-Bandbreite deutlich reduziert werden. Dies erfordert allerdings im Entwicklungssystem einen entsprechend hohen Aufwand zur Rekonstruktion der Sprungadressen. Sie stützt sich auf periodisch übertragene Synchronisations-Informationen, in denen eine aktuelle Instruktions-Adresse enthalten ist.

Ebenfalls sehr bandbreitenintensiv ist die Übertragung eines zyklus-akkuraten Trace-S. Hier werden für jede Instruktion, zumindest aber für die einzelnen Kanten des Kontrollflussgraphen, die benötigte Anzahl von CPU-Takte mit übertragen. Diese Informationen sind beispielsweise für die Worst-Case Execution Time-Analyse relevant.

Aus der Literatur sind noch weitere Methoden zur Reduktion der Bandbreite zur Übertragung von Sprunginformationen bekannt, beispielsweise die Anwendung der Lempel-Ziv-Kompression auf die ermittelten Sprunginformationen [108]. Aufgrund ihrer Komplexität haben diese Methoden aktuell noch keine Anwendung in den industriell genutzten Lösungen wie ARM CoreSight oder Nexus gefunden.

4.2.2 Daten-Trace

Im Vergleich zum Programm Trace erfordert ein Daten-Trace eine deutlich höhere Bandbreite. Zur Beobachtung des Daten-Transfers müssen neben den gelesenen bzw. geschriebenen Daten auch die zugehörigen Adressen sowie zusätzliche Statusinformationen (Breite und Richtung des Zugriffs) mit übertragen werden. Beim Daten-Trace ergeben sich kaum Potenziale zur Komprimierung der zu übertragenden Informationen. Nur die Adresse des Datenzugriffs muss nicht vollständig übertragen werden, hier reicht die Übertragung des Offsets zur letzten bekannten Daten-Adresse.

4.2.3 Erfassung sonstiger Zustandsinformationen

Neben Programm- und Daten-Trace können noch eine Reihe weiterer Zustandsinformationen übertragen werden.

Debug-Status

In einer Debug-Message werden Zustandsänderungen übertragen, die durch einen aktiven Debug-Prozess (Erreichen oder Verlassen einer Ausnahmeroutine - *debug exception handler*, Erreichen eines Breakpoints) bzw. einen Wechsel im Power-Management des SoCs bewirkt werden.

Device-ID

Nach einem Reset sowie auch auf besondere Anforderung hin kann eine Nachricht versendet werden, die eine Kennung (Device-ID) des SoCs enthält.

Ownership Trace

Mittels einer Ownership Trace–Nachricht wird eine geänderte Prozess-ID dem Beobachter bekannt gegeben.

Monitoren des Stromverbrauchs

Erste Ansätze zur Korrelation von ausgeführten Instruktionen und Stromverbrauch sind schon seit Mitte der 1990er Jahre im akademischen Bereich bekannt [109]. Beispiele industrieller Lösungen sind in Tabelle 4-3 aufgeführt (Daten aus [72],[110]).

Hersteller	System	Strommessung	Spannungsmessung	Abtastrate
Hitex	PowerScale	bis zu 3 Kanäle, 200nA bis zu 1A	bis zu 58V	100 kHz
Lauterbach	Analog Probe am Power-Trace-II	bis zu 3 Kanäle	bis zu 4 Kanäle, 0 bis 5V	625kHz

Tabelle 4-3: Beispiele von industriellen Lösung zur Erfassung des Stromverbrauchs

Diese Lösungen gestatten die Bestimmung des Stromverbrauchs bei der Abarbeitung von größeren Codeabschnitten. Um z.B. die Effizienz des Caches zu verbessern, wäre eine feinere Auflösung bis hin zur Messung des Stromverbrauchs einzelner Instruktionen sinnvoll. Neben der Erfordernis deutlich höherer Abtastraten genügt es nicht mehr, die summarische Stromzufuhr zu einem SoC zu protokollieren, da die nahe am SoC befindlichen Blockkondensatoren das Ergebnis der Strommessung verschleifen. Eine technische Lösung wäre hier ein spezielles Entwicklungsboard (oder ein entsprechender Testadapter), bei dem mittels Shunt-Widerständen, die sich zwischen Blockkondensator und den einzelnen Power-Pins des SoCs befinden, die Summe der Stromaufnahme der Power-Pins mit einem entsprechend schnellen Analog/Digital-Wandler gemessen wird.

4.2.4 Quantitative Betrachtungen

Um die Abläufe in einer CPU verfolgen zu können, müssen dem Beobachter sowohl die ausgeführten Instruktionen als auch die Datenzugriffe bekannt gemacht werden.

Abhängig von den zu gewinnenden Informationen müssen für eine Trace-Schnittstelle unterschiedliche Bandbreiten zur Verfügung gestellt werden. Tabelle 4-4 zeigt dies für ARM CoreSight ETMv3 (aus[111], Tabelle 5.1), PFTv1 und ETMv4 (aus [112], Fig. 3).

Für einen einfachen Instruktions-Trace müssen ja nach Implementierung durchschnittlich 0,3 Bits bis 1,2 Bits pro ausgeführter Instruktion übertragen werden. Interessiert zusätzlich die Zahl der für die Instruktion benötigten Takt-Zyklen, so steigt die Bandbreite deutlich an.

ETM Version	Trace-Kategorie	Zyklus-genauigkeit	Durchschnittliche Bandbreite (Bits / Instruktion)		
			Instruktionen	Datentransfer	Summe
ETMv3	Instruktionen und Daten	Nein	4	8	12
	Instruktionen	Nein	1,2	-	1,2
	Instruktionen und Daten	Ja	8	8	16
	Instruktionen	Ja	6	-	6
PFTv1	Instruktionen	Nein	0,3	-	0,3
	Instruktionen	Ja			
ETMv4	Instruktionen	Nein	0,3	-	0,3

Tabelle 4-4: Durchschnittlicher Bandbreitenbedarf einer Trace-Schnittstelle

Eine im Vergleich zum Programm-Trace deutlich höhere Bandbreite erfordert die Beobachtung gelesener und geschriebener Daten. Für die zugehörige Adresse, den Datenwert sowie ergänzende Informationen (Lesen oder Schreiben, Zugriffsbreite) sind durchschnittlich 40 Bits pro Daten-Transfer erforderlich [111]. Der Daten-Transfer unterteilt sich in Lese- und Schreiboperationen, wobei für die Beobachtung eines SoCs meist die Ergebnisse von Lesezugriffen (LOAD) besonders interessant sind. Die durchschnittliche Häufigkeit von Lesezugriffen liegt bei ca. 20% [113], daraus ergibt sich ein durchschnittlicher zusätzlicher Bandbreitenbedarf von 40 Bit/Speicherzugriff * 20% = 8 Bit/Instruktion.

4.2.5 Physikalische Schnittstelle

Die Ausgabe von Trace-Daten von SoCs stellt einen erheblichen Kostenfaktor dar, da die verwendete Schnittstelle über eine hohe Bandbreite verfügen muss.

Verfügt der SoC bereits über schnelle serielle Schnittstellen (z.B. PCIe), so können mit der gleichen Fertigungstechnologie auch entsprechende serielle Schnittstellen zur Ausgabe der Trace-Daten (meist Aurora) implementiert werden. Zukünftig ist es geplant, zunehmend Trace-Daten auch über schnelle Standard-Schnittstellen (z.B. PCIe, GBit-Ethernet) auszugeben. Ist die Implementierung schneller serieller Schnittstellen technologisch nicht möglich, werden die Trace-Daten meist über eine parallele Schnittstelle ausgegeben.

Ein Nachteil aller Trace-Schnittstellen ist, dass die dafür verwendeten I/O-Pins der Applikation nicht oder nur eingeschränkt zur Verfügung stehen. Da der Zugriff auf die Trace-Daten im Wesentlichen nur während der Entwicklung bzw. der Zertifizierung eines Systems erforderlich ist, ande-

rerseits der entstandene Code nicht mehr verändert werden sollte, werden oft im produktiven Einsatz die für die Trace-Ausgabe verwendeten I/O-Pins ungenutzt gelassen, was einen erheblichen Kostenpunkt darstellt.

Parallele Schnittstelle

Eine parallele Trace-Schnittstelle kann Trace-Daten bis zu einer Geschwindigkeit von ca. 300 MHz ausgeben. Da die Ausgabe der Trace-Daten synchron zu beiden Flanken des Taktsignals erfolgt, kann pro I/O-Pin (zzgl. weitere I/O-Pins für Takt und Steuersignale) eine Bandbreite von 600 Mbps erreicht werden. Oftmals ist aber aufgrund der verwendeten Technologie die maximal mögliche Taktfrequenz der I/O-Pins auf einen niedrigeren Wert (z.B. 100MHz) beschränkt.

Aufgrund der hohen Frequenzen muss das Design der PCBs entsprechend aufwändig gestaltet werden, weiterhin müssen möglichst impedanzkontrollierte Steckverbinder verwendet werden. Üblicherweise sind dies die MICTOR-Stecker (*Matched Impedance ConnecTOR* von TE Connectivity) oder die ERF8 bzw. QSH Steckverbinder von Samtec.

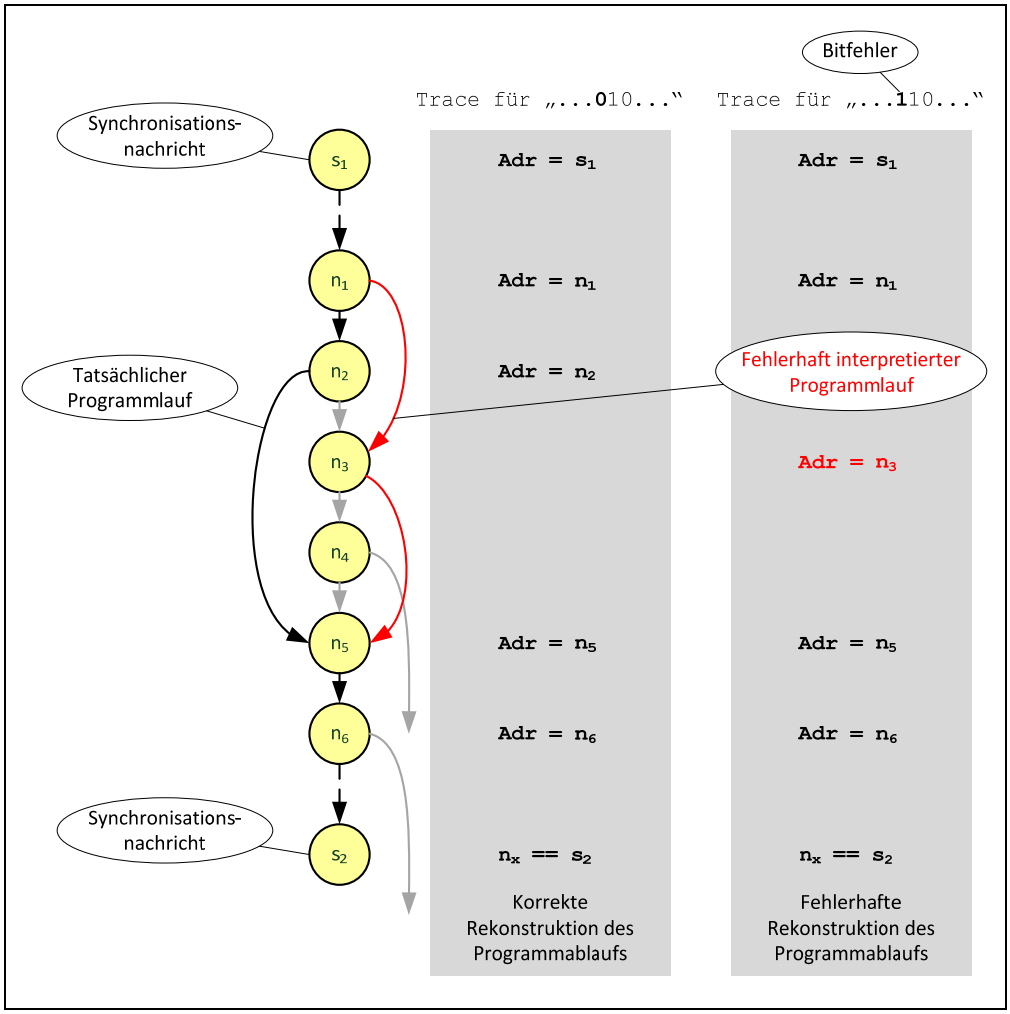


Abbildung 4-9: Beispiel eines nicht erkennbaren Fehlers bei der Rekonstruktion des Programmablaufs

Die Ausgabeprotokolle paralleler Trace-Schnittstellen verfügen üblicherweise nicht über eine Sicherungsschicht, welche mittels Prüfsummen die Fehlerfreiheit der übermittelten Trace-Daten garantieren. Wenn man berücksichtigt, dass die Trace-Daten oftmals mit der maximal möglichen Bandbreite ausgegeben werden, ist davon auszugehen, dass eine gewisse Bitfehlerwahrscheinlichkeit vorhanden ist. Wird ein Bit an einer ungünstigen Stelle verfälscht, kann dies beispielsweise zur

fehlerhaften Annahme der Ausführung bzw. Nichtausführung eines direkten Sprungs führen, für die es keine weiteren Erkennungsmöglichkeiten gibt. Damit könnte dann das Ergebnis einer Coverage-Analyse verfälscht werden.

In Abbildung 4-9 wird ein derartiges Szenario gezeigt. Dabei steht eine „1“ in den Trace-Daten für einen ausgeführten direkten Sprung, bei einer „0“ wird der Sprung nicht ausgeführt. Es werden periodische Synchronisations-Nachrichten übertragen, welche die absolute Instruktionsadresse beinhalten. Diese Adressen können mit den zuvor rekonstruierten Adressen verglichen werden, eine Abweichung weißt hier auf eine fehlerhafte Adress-Rekonstruktion hin. Das Beispiel zeigt aber, dass dieser Mechanismus nicht immer zuverlässig funktioniert.

Diese Fehlermöglichkeit muss bei der Einschätzung der Zuverlässigkeit eines Beobachtungswerkzeugs berücksichtigt werden und bei sicherheitskritischen Anwendungen durch entsprechende Gegenmaßnahmen (z.B. mehrfache Läufe für jeweils die gleiche Überdeckungsanalyse) kompensiert werden. Die zuverlässigste Lösung dieses Problems würde im Hinzufügen einer Sicherungsschicht zum parallelen Trace-Protokoll bestehen, die einerseits über zusätzliche Prüfsummen-Leitungen als auch über das Hinzufügen von speziellen Paketen mit fortlaufenden Prüfsummen implementiert werden kann. Letztere könnten beispielsweise statt der üblicherweise „leeren“ Idle-Pakete gesendet werden.

Serielle Schnittstelle

Alternativ zu einer parallelen Schnittstelle können Trace-Daten auch über eine serielle Hochgeschwindigkeitsschnittstelle ausgegeben werden. Dazu wird hauptsächlich das von Xilinx entwickelte Aurora-Protokoll [114] verwendet.

Implementierungsbeispiele sind ARM’s serielle Hochgeschwindigkeitsschnittstelle HSSTP (*High Speed Serial Trace Port*) mit bis zu 8 differentiellen Leitungen Trace-Daten mit einer Bandbreite von jeweils bis zu 12.5 Gbps [115] oder der von Power.org definierte „*Standard for Physical Connection for High-Speed Serial Trace*“[116] mit ebenfalls bis zu 8 differentiellen Leitungen und einer maximalen Bandbreite von 6.25 Gbps. Beide Standards empfehlen die Verwendung des gleichen Steckverbinders ERF8 (Samtec), wobei die Pinbelegung jeweils unterschiedlich ist.

Serielle Hochgeschwindigkeits-Schnittstellen zur Ausgabe von Trace-Daten verfügen über eine Sicherungsschicht und vermeiden dadurch eine Verfälschung der Beobachtungsergebnisse aufgrund von Bitfehlern während der Übertragung.

Weitere Schnittstellen

Die Vermeidung der Benutzung von I/O-Pins zur Ausgabe der Trace-Daten ist auch das Ziel einer ganzen Reihe weiterer Ansätze zur Implementierung von Trace-Schnittstellen. Beispielsweise ist in [117] eine optoelektronische Schnittstelle beschrieben, welche über 12 Kanäle Trace-Daten mit einer Bandbreite von insgesamt 28,125 Gbps übertragen kann. Bei einem anderen Ansatz [118] werden die Trace-Daten induktiv ausgekoppelt. Hier werden Übertragungsraten von 480 Mbps erzielt.

4.2.6 ARM CoreSight

ARM bietet mit der CoreSight eine umfangreiche Lösung zur Beobachtung von SoCs. An die einzelnen CPUs kann jeweils eine Trace-Einheit angekoppelt werden (*Embedded Trace Macrocell –ETM* für die Cortex M- und R-Serien, *Program Flow Trace Macrocell* – PTM für Cortex die Cortex-A Serie), weitere Einheiten erlauben die Beobachtung des Gesamtsystems (*System Trace Macrocell* - STM) und die hardwareunterstützte Instrumentierung (*Instrumentation Trace Macrocell* - ITM). Optional kann noch eine Daten-Trace-Einheit (*Data Watchpoint and Trace* – DWT*)* implementiert werden. Diese Trace-Quellen sind über ein Bussystem (*Advanced Trace Bus* – ATB) mit einer Einheit zur Ausgabe der Trace-Daten (*Trace Port Interface Unit* – TPIU) bzw. mit einem lokalen Trace-Speicher (*Embedded Trace Buffer* - ETB) verbunden (Abbildung 4-10).

Die ITM und STM können nicht nur über die TPIU, sondern auch über einen *Serial Wire Output* (SWO) ihre Trace-Daten ausgeben.

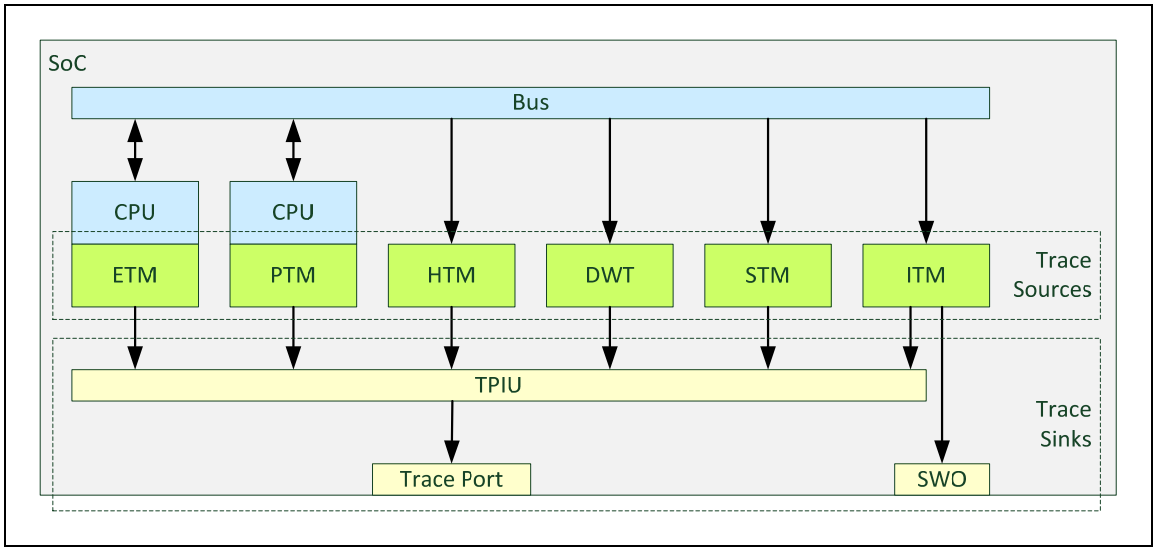


Abbildung 4-10: Blockschaltbild Trace-Quellen in der ARM CoreSight Architektur

Neben den Trace-Quellen und den Einheiten zur Ausgabe der Trace-Daten verfügt die ARM CoreSight Architektur noch über weitere Einheiten wie z.B. den *Debug Access Port* (DAP) und eine *Cross Trigger Matrix* (CTM), auf die hier aber nicht näher eingegangen werden soll.

Die Übertragung von Trace-Datenströmen wird über die TPIU gesteuert, welche einen kontinuierlichen Trace-Datenstrom aus mehreren Quellen mit einem Protokoll versieht. Dazu werden die Trace-Daten um Informationen ihrer Herkunft (*Source ID*) sowie um periodische Synchronisations-Informationen erweitert. Zur Auswertung des Trace-Datenstroms müssen diese Synchronisations-Informationen wiedergefunden werden und der Datenstrom entsprechend seiner Quellen aufgeteilt werden. Innerhalb des quellspezifischen Datenstroms muss nun nach periodischen Synchronisationspaketen (A-Sync) gesucht werden, von denen ausgehend die einzelnen Pakete sequenziell interpretiert werden. Dies erfordert einen entsprechend hohen Rechenaufwand zur Extraktion der einzelnen Nachrichten (Abbildung 4-11).

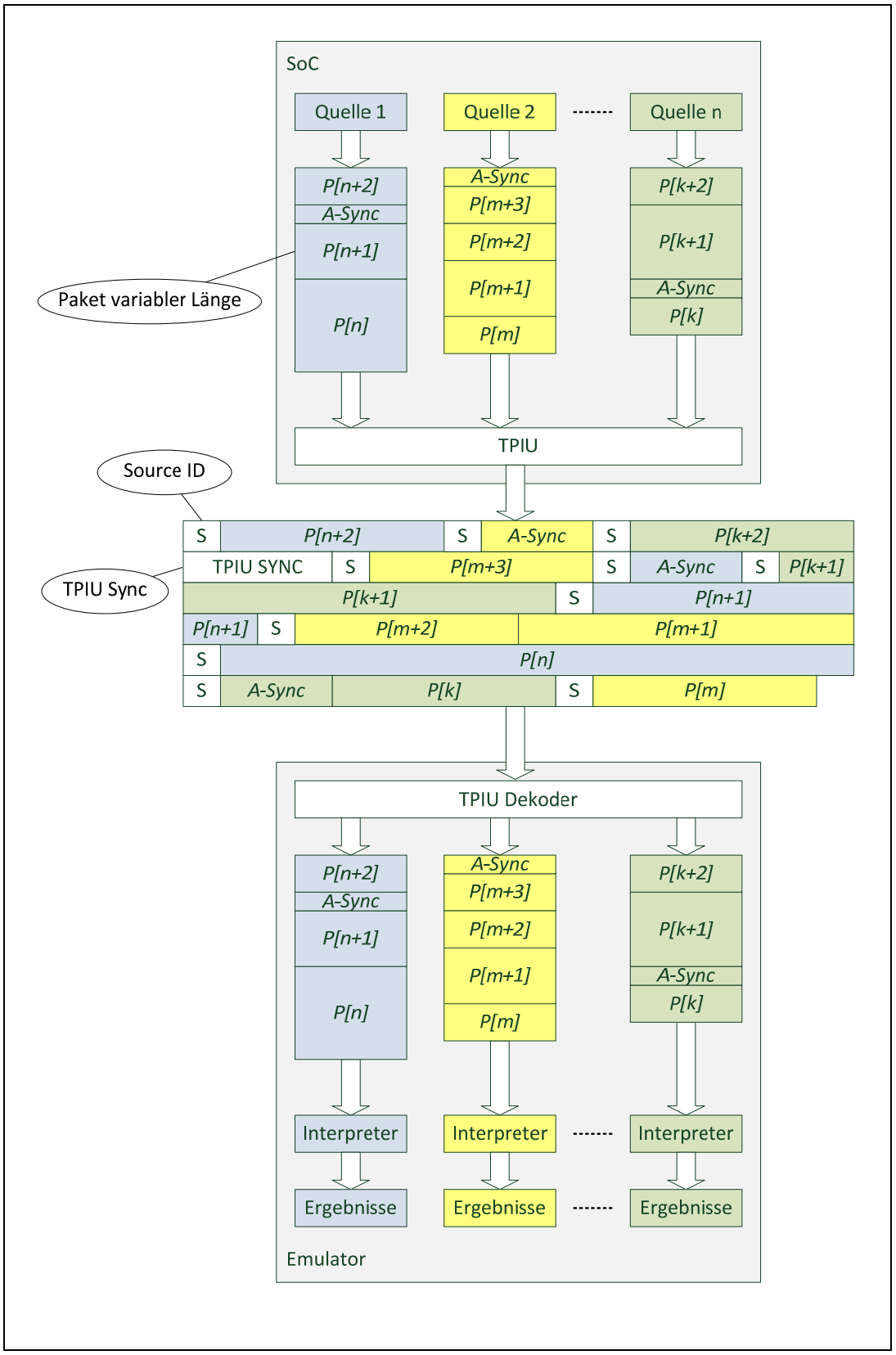


Abbildung 4-11: Übertragung von Trace-Daten aus verschiedenen Quellen

Die TPIU gibt je nach Implementierung die Trace-Daten über eine parallele Schnittstelle (ohne Sicherungsschicht) oder über eine serielle Hochgeschwindigkeitsschnittstelle (*High Speed Serial Trace Port* - HSSTP [115]) aus.

Aktuelle ARM-Prozessoren unterteilen sich in drei Profile (A: Applikation, R: Real-Time, M: Mikrocontroller), die in Tabelle 4-5 aufgelistet sind.

Prozessor-Profil	**A**pplikation		**R**eal-Time	**M**ikrocontroller
Name	Cortex-A8	Cortex A9	Cortex-R	Cortex-M
Macrocell	ETMv3.3	PTMv1	ETMv3.5 (R4, R5) ETMv4 (R7)	ETMv3.5 (M3, M4) DWT
Daten-Trace (Adressen)	Ja	Nein	Ja	Ja (via DWT)
Daten-Trace (Werte)	Nein	Nein	Ja	Ja (via DWT)
Instruktionstrace	Ja	Ja	Ja	Ja
Zyklusgenauer Instruktionstrace	Nein	Ja	Ja	Nein

Tabelle 4-5: Profile der ARM Cortex Prozessoren

4.2.7 Nexus

Als Ergebnis von Standardisierungsbemühungen für Debug- und Trace-Schnittstellen für SoCs wurde der Nexus-Standard verabschiedet, welcher mittlerweile in der dritten Version vorliegt [102].

Die Übertragung der Trace-Informationen erfolgt mittels Nachrichten (*Messages*). Diese bestehen aus einem oder mehreren *Beats*, die sich aus MDO- (*Message Data Out*) und MSEO-Signalen (*Message Start/End Out*) zusammensetzen (Abbildung 4-12).

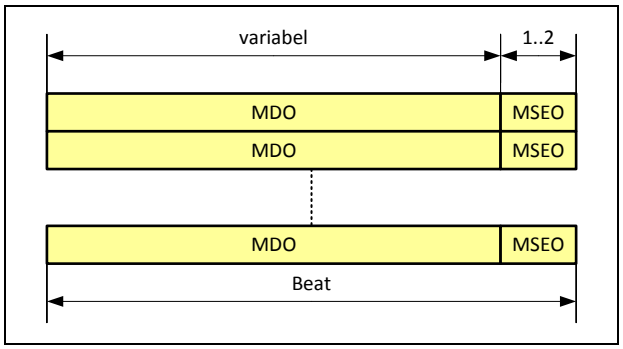


Abbildung 4-12: Formatierung von Nexus-Signalen

Die Nutzdaten werden dabei innerhalb der MDO-Signale übertragen, die MSEO-Flags dienen der Signalisierung der Grenzen einer Nachricht sowie der Abgrenzung von Blöcken variabler Länge innerhalb einer Nachricht. Innerhalb eines Beats können ein oder mehrere Blöcke fester Länge sowie maximal ein Block variabler Länge übertragen werden.

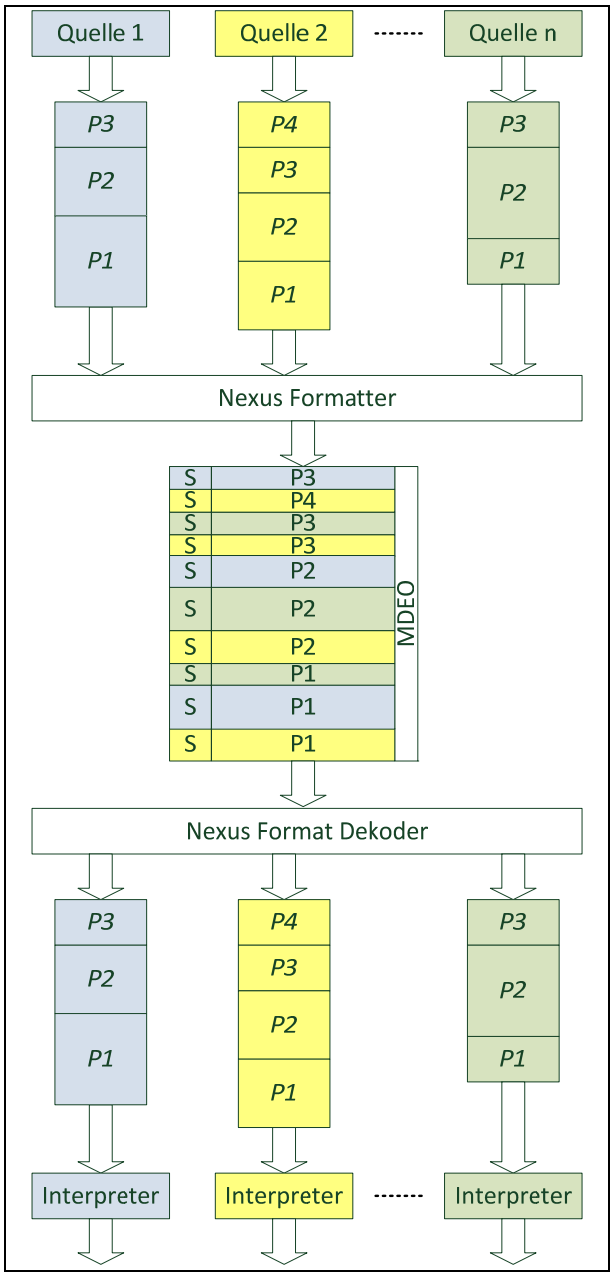


Abbildung 4-13: Übertragung von Trace-Daten aus verschiedenen Quellen und Signalisierung via Seitenkanal (Nexus)

Gegenüber der bei ARM Coresight verwendeten Formatierung (TPIU) ist die Extraktion einzelner Nachrichten aus dem Trace-Datenstrom bei einer Nexus-Implementierung deutlich einfacher, allerdings ist auch der Bandbreitenbedarf des Protokolls höher (Abbildung 4-13).

Eine Nachricht besteht aus einem Header, in dem der Nachrichtentyp (6 Bit TCODE) sowie die Herkunft der Nachricht (SOURCE ID) abgelegt ist. Diesem folgen dann spezifische Nutzdaten.

Neben einer Reihe vordefinierter Nachrichtentypen können auch herstellerspezifische Nachrichten (*Vendor-Defined Messages*) ausgegeben werden.

Die Beats können auch serialisiert (Xilinx Aurora Protokoll) ausgegeben werden.

Eine Nexus-Implementierung unterstützt meist nur eine Untermenge der möglichen Nachrichtentypen. In Tabelle 4-6 sind die verschiedene Implementierungsklassen (*compliance classes*) defi-

niert, welche von einer einfachen Debug-Schnittstelle (Level 1) bis zu komplexen Trace-Schnittstellen (Level 4) reichen. Dabei wird zwischen erforderlichen (R) und optionalen Merkmalen (o) unterschieden.

Eine Besonderheit gegenüber ARM CoreSight ist hier die Möglichkeit der Port-Ersetzung. Dabei werden Nachrichten übermittelt, welche es ermöglichen, die Funktionalität der durch die Nexus-Schnittstelle belegten I/O-Pins zu emulieren. Dies ist sowohl für Eingänge als auch für Ausgänge möglich. Der gepufferte, paketorientierte Transfer der Nexus-Nachrichten führt hier allerdings zu einem nicht determinierten Delay, was im System Design entsprechend berücksichtigt werden muss.

	Klasse			
Merkmal	1	2	3	4
Watchpoint Trace	R	R	R	R
Ownership Trace		R	R	R
Programm-Trace		R	R	R
Daten-Trace (Schreibzugriffe)			R	R
Daten-Trace (Lesezugriffe)			o	o
Datenerfassung (hardwareunterstützte Instrumentierung)			o	o
Timestamps		o	o	o
Port Ersetzung		o	o	o

Tabelle 4-6: Nexus Implementierungsklassen mit ausgewählten Trace-Eigenschaften

Beispiel einer komplexen Nexus-Implementierung für einen MPSoC ist die *Advanced QorIQ Platform Debug Architecture (ADPA)* [119] für den QorIQ P4080 [120] von Freescale (Abbildung 4-14).

Mit Hilfe der ADPA können folgende Funktionseinheiten des P4080 beobachtet werden:

- 8 CPU-Kerne
- Interface zum externen Speicher (DDR)
- Netzwerk-Interface *(Data Path)*
- Serielle Hochgeschwindigkeits-Schnittstellen (PCIe, serial RapidIO)
- Multilayer-Bussystem (CoreNet)

Die Trace-Daten werden über ein Aurora-Interface ausgegeben, welches eine Bandbreite von bis zu 10 Gbps hat.

Trotz dieser eindrucksvollen Leistungsmerkmale sind die internen Vorgänge nur begrenzt beobachtbar:

- Die verfügbare Bandbreite ist nicht ausreichend zur Ausgabe eines kontinuierlichen Programm-Trace-S für alle CPUs (bei maximalem Arbeitstakt).
- Ein Daten-Trace kann nur für einen eingeschränkten Adressbereich aufgezeichnet werden.
- Ein zyklusgenauer Trace ist nicht möglich.

Diese Einschränkung der Beobachtbarkeit wird sich zukünftig noch verschärfen, wenn die QorIQ-Prozessoren der nächsten Generationen mit einer noch höheren Anzahl von CPU-Kernen (aktuell sind diese mit bis zu 24 CPUs angekündigt [121]) verfügbar werden.

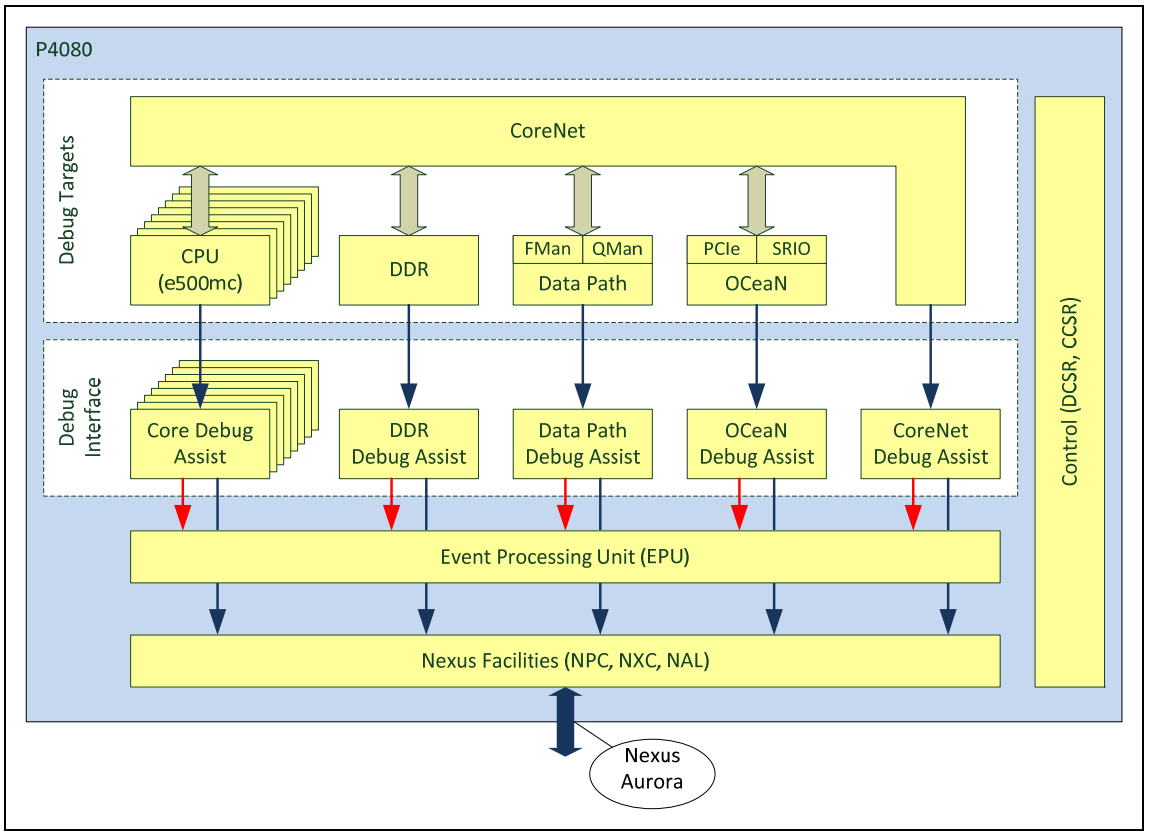


Abbildung 4-14: Überblick über die Advanced QorIQ Platform Debug Architecture für Freescale's P4080

4.2.8 Infineon MCDS

Eine weitere, im industriellen Umfeld relevante Lösung zur Beobachtung von SoCs stellt die *Multi-Core Debug Solution* (MCDS) von Infineon dar. Im Gegensatz zu ARM CoreSight bzw. Nexus ist eine detaillierte Beschreibung dieses Systems nicht frei erhältlich, die folgende Beschreibung basiert daher auf sekundären Veröffentlichungen [122] [85] [123] [124].

Mit MCDS hat Infineon einen neuen Ansatz entwickelt, mit dem die bei der Beobachtung von MPSoC anfallenden großen Datenmengen verarbeitet werden können. Im Gegensatz zu ARM CoreSight bzw. Nexus soll die Verarbeitung der Trace-Daten vorrangig nicht in einem externen System, sondern mittels einer Emulationseinheit (*Emulation Device* - ED) als Erweiterung des zu beobachtenden Serienchips (*Production Device* - PD) vorgenommen werden.

Über eine geeignete serielle Schnittstelle können dann die Resultate der Auswertung der Trace-Daten ausgelesen werden.

Folgende Nachrichten sind verfügbar:

- Instruction Pointer Call Messages (ptrace), beinhaltet
  - Aussprungadresse (*take-off address*)
  - Einsprungadresse (*landing address*)
  - Anzahl der seit dem letzten Sprung ausgeführten Instruktionen
  - Source-ID
- Read / Write Data Trace Messages (CPU, System Peripheral Bus, Shared Resource Interconnect), beinhaltet
  - Datenadresse
  - Wert
  - Source-ID
- Debug Status Message (Status einer CPU oder eines Busses)
- Watchpoint Trace Message (Detektion eines Ereignisses)
- Timestamp Message

Die Trace-Nachrichten können mit einer leistungsfähigen Filter- und Triggerlogik analysiert sowie in einem bis zu 1 Mbyte großen Trace-Puffer gespeichert werden.

Vorteil dieser Lösung ist der breitbandige Anschluss (bis zu 40 Gbps) der Emulationslogik an den zu untersuchenden SoCs, wobei im Gegensatz zu ARM CoreSight / Nexus die Sprunginformationen nicht komprimiert und deshalb entsprechend bandbreitenintensiv sind.

Nachteil ist die Beschränkung der Möglichkeiten zur Trace-Analyse auf die im ED implementierten Ressourcen. Eine Skalierbarkeit und ggfs. Anpassbarkeit an spezielle Beobachtungsaufgaben, wie sie mit einer externen Auswertungslogik möglich wäre, ist mit dieser Lösung nicht gegeben.

4.2.9 Effizienz der Trace-Komprimierung

Im Folgenden werden Effizienz und Funktionsumfang der vorgestellten Lösungen im Hinblick auf die Ausgabe von Programm- und Daten-Trace verglichen. Dabei werden jeweils maximale Ausbaustufen angenommen, bei konkreten Realisierungen können einzelne Funktionsmerkmale (z.B. Timestamps) eventuell nicht mit implementiert sein.

Programm-Trace

Die diskutierten Lösungen unterstützen verschiedene Komprimierungsstufen zur Beobachtung des Programmablaufs (Tabelle 4-7). Die Zieladressen indirekter Sprünge werden stets übertragen, wobei hier nur der geänderte Teil der Zieladresse übermittelt wird. Zusätzlich werden periodisch Instruktionsadressen übertragen, um davon ausgehend den Programmablauf berechnen zu können.

Die ***ARM Coresight PTM*** sowie ***ARM Coresight ETMv4*** gibt Informationen aus, ob ein direkter Sprung ausgeführt wurde. Die Aus- und Einsprungsadresse muss bei der Auswertung des Trace-Datenstroms berechnet werden. Optional kann noch die Anzahl der benötigten Taktzyklen (*cycle-accurate trace*) zwischen zwei direkten Sprungbefehlen übermittelt werden.

Eine ***Nexus***-Implementierung arbeitet ähnlich wie die PTM, auch hier wird die Information übermittelt, ob ein direkter Sprung ausgeführt wurde (*branch history*). Die Ausgabe der Anzahl von Taktzyklen zwischen zwei Sprunginstruktionen ist nicht vorgesehen.

Alternativ können (implementierungsabhängig) auch „*traditional messages*“ (TCODE = 11) ausgegeben werden, hier wird – analog zum indirekten Sprung – die Zieladresse sowie die Anzahl der bisher ausgeführten Instruktionen mit übertragen. Diese Variante verlangt eine höhere Bandbreite, die Berechnung des Programmablaufs ist aber deutlich einfacher. Werden die „*traditional messages*“ mit einem Zeitstempel versehen, so ist hier auch ein zyklusgenauer Trace eines Basisblocks möglich. Aufgrund der größeren Häufigkeit von direkten Sprüngen gegenüber indirekten Sprüngen kann die Verwendung der „*traditional messages*“ zu einer Vervielfachung der erforderlichen Bandbreite für die Übermittlung der Trace-Daten führen und schränkt damit die Anwendbarkeit dieser Variante ein.

Bei ***ARM Coresight ETMv3.5*** wird für jede einzelne Instruktion die Information ausgegeben, ob diese ausgeführt wurde oder nicht. Optional besteht die Möglichkeit zur Ausgabe der zur Ausführung einer Instruktion benötigten Taktzyklen.

Ähnlich den „*traditional messages*“ von Nexus kann auch die ETM so konfiguriert werden, dass neben der Zieladresse eines indirekten Sprungs auch die eines direkten Sprungs dem Entwicklungssystem mitgeteilt wird (z.B. ARM ETMv3.5: ETMCR.BRANCH_OUTPUT = 1 [125]).

Da bei ***Infineon MCDS*** die Verarbeitung der Sprunginformationen auf dem gleichen Chip geschieht, ist hier der Schwerpunkt nicht die Bandbreitenreduktion, sondern eine möglichst einfache Weiterverarbeitung. Aus diesem Grund werden sowohl Ein- als auch Aussprungadresse übertragen.

Zur Ablage der Sprunginformationen im Trace-Puffer sowie zur Ausgabe durch das Aurora-Interface können diese ähnlich zu Nexus / PTM komprimiert werden [126].

Implementierung	Eigenschaften	
	Übertragung direkter Sprünge	Erfassung der benötigten CPU-Takte
Coresight PTM / ETMv4	Ausgeführt ja/nein	Basic Block
Coresight ETMv3.5	Ausgeführt ja/nein	Instruktion
Nexus (Branch history)	Ausgeführt ja/nein	Nein
Coresight ETM (BRANCH_OUTPUT)	Einsprungsadresse	Instruktion
Nexus (traditional)	Einsprungsadresse	Nein
MCDS	Aus- und Einsprungsadresse	Instruktion, Call/Return

Tabelle 4-7: Kompressionsverfahren für Programm-Trace für verschiedene „embedded Trace"-Lösungen

Daten-Trace

Die zur Beobachtung des Daten-Transfers erforderlichen Informationen lassen sich nur schlecht komprimieren, d.h. die zu übertragenden Daten sind bei allen Implementierungen vergleichbar.

Bei ***ARM CoreSight ETM*** und ***PTM*** ist die Interpretation der Trace-Daten deutlich erschwert, da in dem Daten-Trace-Paket auf die Übertragung der Richtung des Daten-Transfers (Lesen oder Schreiben) verzichtet wird. Um diese essentielle Information zu erhalten, muss neben dem Daten-Trace auch der Instruktions-Trace analysiert werden.

Formatierung des Trace-Datenstroms

Einen Einfluss auf die Trace-Bandbreite hat auch die Formatierung des ausgegebenen Trace-Datenstroms.

Die effizienteste Lösung bietet hier die TPIU von ***ARM CoreSight***. Ist nur eine Quelle von Trace-Daten vorhanden, kann die TPIU deaktiviert und die ETM/PTM-Daten direkt ausgegeben werden. Der Effizienzgewinn erfordert aber sehr hohen Rechenaufwand bei der Interpretation des Trace-Datenstroms, da dieser ausgehend von einem Alignment-Synchronisationspaket (A-Sync) sequentiell analysiert werden muss, um die Grenzen der jeweils folgenden Pakete bestimmen zu können.

Der Overhead bei ***Nexus*** ist größer, da neben den Nutzdaten in einem Nebenkanal auch noch die MSEO-Signale ausgegeben werden müssen. Zusätzlich kann es möglich sein, dass bei variablen Blocklängen die Bits in einem Beat nicht vollständig ausgenutzt werden können. Der große Vorteil bei der Verarbeitung des Trace-Datenstroms liegt darin, dass die einzelnen Nachrichten leicht extrahierbar sind.

4.2.10 Bewertung

Im Folgenden wird die Beobachtung eines SoCs mittels „embedded Trace“-Lösungen anhand der in Abschnitt 3 diskutierten Kriterien bewertet. Diese Bewertung ist exemplarisch und muss im Einzelfall an die Möglichkeiten des zu untersuchenden SoCs und die Anforderungen für die Beobachtung angepasst werden.

Vollständigkeit der Beobachtung

Die beobachtbaren Elemente sind von der jeweiligen Hardware-Implementierung des SoCs abhängig. Üblicherweise sind ein Ownership- und ein Instruktions-Trace verfügbar. Für letzteren besteht nur bei einigen Implementierungen die Möglichkeit einer zyklusgenauen Beobachtung. Informationen über die spekulative Ausführung von Programmcode sowie über Änderungen der Ausführungsreihenfolge werden teilweise - sofern diese Beschleunigungsmechanismen implementiert sind - als Bestandteil der Trace-Daten ausgegeben. Cache-Zugriffe können indirekt anhand der für die Ausführung erforderlichen CPU-Takte beobachtet werden.

Der Daten-Trace ist oftmals nicht oder nur eingeschränkt verfügbar, teilweise werden nur Schreibzugriffe übertragen, teilweise wird nur die Zugriffsadresse, nicht aber der Wert übermittelt. Die Ausgabe von Änderungen von CPU-Registern ist in den aktuell verfügbaren Lösungen nicht implementiert.

Die Reaktion der CPU auf Ereignisse (z.B. Interrupt-Anforderungen) kann anhand des Instruktions- oder Ownership-Trace-S verfolgt werden, die auslösenden Ereignisse (z.B. das Setzen eines Interrupt-Request-Flags) sind im Trace nicht sichtbar.

Für die Verfolgung der Vorgänge auf Bussystemen oder Peripherieeinheiten können spezielle Beobachter implementiert sein (z.B. die CoreNet-Trace beim Freescale P4080).

Neben begrenzten Möglichkeiten, Trace-Daten innerhalb des SoCs zu erfassen, stellt die verfügbare Bandbreite der Ausgabe von Trace-Daten eine weitere Limitation der vollständigen Beobachtbarkeit dar. Auch wenn SoC-intern die interessierenden Informationen verfügbar gemacht werden können, scheitert deren Ausgabe oftmals an einer zu geringen Leistungsfähigkeit der Trace-Schnittstelle.

Beobachtbarkeit von MPSoCs

Sollen mehrere CPUs und busmasterfähige Peripherieeinheiten beobachtet werden, so ist eine Synchronisation der quellenspezifisch generierten Trace-Daten oftmals erforderlich. Diese Synchronisation (z.B. mittels eines zyklusgenauen Trace-S oder der engmaschigen Ausgabe von Timestamps) erfordert eine deutliche Zunahme der verfügbaren Übertragungsbandbreite. Aus diesem Grund kann oftmals nur ein Teil der in einem MPSoC implementierten CPUs detailliert beobachtet werden.

Echtzeitfähigkeit und Kontinuität

Verfügt die Trace-Schnittstelle über ausreichend Bandbreite, so kann der SoC beobachtet werden, auch wenn die CPUs mit einem maximal möglichen Takt arbeiten. Ist die verfügbare Bandbreite nicht ausreichend, so kann der CPU-Takt verlangsamt oder die CPU vorübergehend gestoppt werden, um die Menge der anfallenden Trace-Daten der verfügbaren Bandbreite anzupassen. Alternativ können auch Trace-Nachrichten verworfen werden.

Die Ausgabe von Trace-Daten kann grundsätzlich kontinuierlich und über einen beliebig langen Zeitraum erfolgen. Aktuelle Systeme, die Trace-Daten weiterverarbeiten, werten diese durch eine offline Analyse aus, deren Verarbeitungsgeschwindigkeit oftmals deutlich unter der Bandbreite der einströmenden Trace-Daten liegt. Daher ist die tatsächlich mögliche Beobachtungsdauer abhängig von der Größe des Zwischenspeichers, in den die Trace-Daten temporär vor ihrer endgültigen (Offline-) Verarbeitung abgelegt werden. Selbst mit den aktuell verfügbaren Speichertechnologien ist eine (beliebig) lang andauernde Beobachtung bei einer hohen Bandbreite von Trace-Daten bei ver-

tretbarem Aufwand nicht möglich. Dazu kommt dann noch die Wartezeit, bis der zwischengespeicherte Trace-Datenstrom verarbeitet ist (oftmals ein Vielfaches der Beobachtungsdauer), so dass ein flüssiges Arbeiten in vielen Fällen nicht möglich ist.

Gleichzeitige Durchführung vieler Beobachtungsaufgaben

Durch die begrenzten Filter- und Triggermöglichkeiten der sich auf dem SoC befindlichen „embedded Trace"-Einheit können nur wenige Beobachtungsaufgaben gleichzeitig erfüllt werden, da aufgrund der begrenzten Bandbreite zur Ausgabe der Trace-Daten oftmals eine Filterung notwendig ist.

Beeinflussung des SoCs

Die „embedded Trace"-Einheit kann so konfiguriert werden, dass die Ausgabe der Trace-Daten ohne Beeinflussung der CPU, des Bussystems oder von Peripherieeinheiten stattfinden kann.

Falls die Ausgabe der Trace-Daten so konfiguriert wird, dass die Trace-Quelle stoppt, falls die erforderliche Bandbreite nicht zur Verfügung steht, so muss diese Beeinflussung berücksichtigt werden. Eine weitere, häufig aber nur minimale Beeinflussung kann sich ergeben, wenn während der Beobachtung die Konfiguration der „embedded Trace"-Einheit über eine Debug-Schnittstelle verändert wird.

Beobachtbarkeit von SoCs aus der Serienproduktion

Verfügt der in der Serienproduktion verwendete SoC über eine „embedded Trace"-Einheit, so kann dieser auch im Serienprodukt umfassend beobachtet werden. Mit fortschreitender Integration und immer kleineren Strukturen steigt der Anteil der mit einer „embedded Trace"-Einheit ausgestatteten SoCs.

Beobachtbarkeit in „realer" Umgebung

Voraussetzung zur Beobachtung eines SoCs im produktiven Einsatz ist, dass auch hier die erforderliche Trace-Schnittstelle nutzbar ist und das Trace-Werkzeug angeschlossen werden kann. Oftmals wird aus Kostengründen die Steckverbindung zum Anschluss des Trace-Werkzeugs nicht mit bestückt, dies kann aber auch nachträglich vorgenommen werden.

Flexibilität des Beobachtungsfokus

Eine dynamische Anpassung des Beobachtungsfokus beim Wechsel relevanter Zustände des SoCs (z.B. Taskwechsel) setzt voraus, dass diese mit entsprechend geringer Latenz dem Beobachter bekannt gemacht werden können, so dass er über geeignete Debug- Schnittstellen (z.B. JTAG) die Filter und Trigger der „embedded Trace"-Einheit entsprechend umkonfigurieren kann. Dieses Leistungsmerkmal ist in aktuell verfügbaren Trace-Werkzeugen nicht verfügbar.

Latenz

Bedingt durch die Zwischenspeicherung der Trace-Daten und der Offline-Auswertung besteht eine sehr große Latenz zwischen dem zu beobachtenden Vorgängen im SoC und dem Bekanntwerden dieser Vorgänge im Beobachtungswerkzeug. Bei entsprechend großen Mengen an Trace-Daten kann sich die Latenz im Minuten-Bereich bewegen.

Qualifizierung des Beobachters

Entsprechend konfiguriert bietet eine „embedded Trace"-Einheit die Möglichkeit, einen SoC weitgehend einwirkungsfrei beobachten zu können. Besonders vorteilhaft ist, dass die Testläufe auch mit dem finalen Programmcode durchgeführt werden können und die bei der temporären Instrumentierung verbleibenden Unsicherheiten (Abbildung 4-2) aufgrund des erneut notwendigen Build-Prozesses entfallen (Abbildung 4-15).

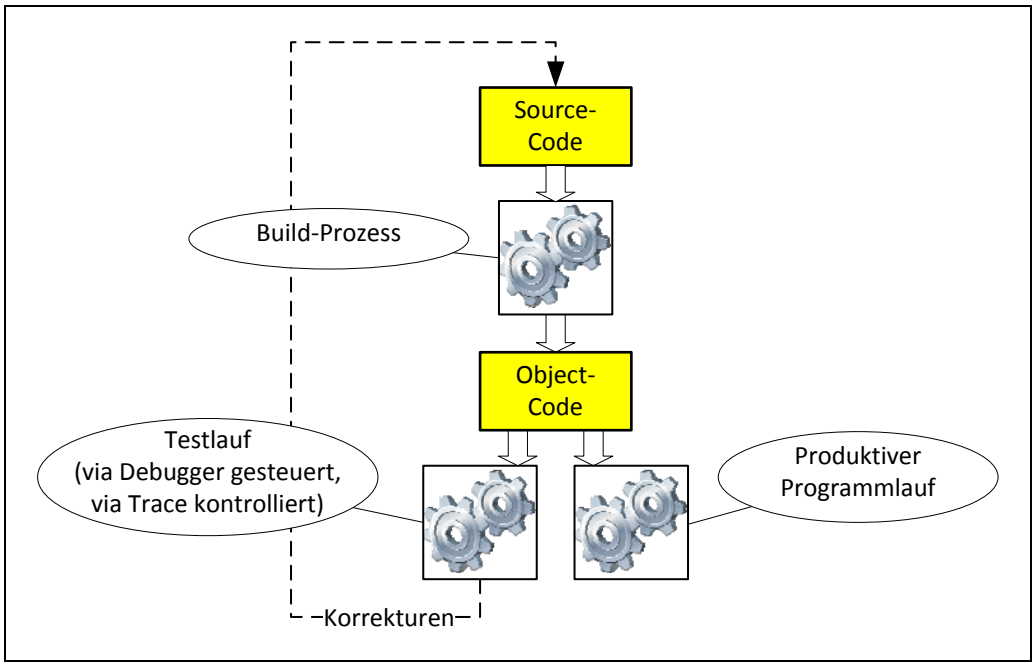


Abbildung 4-15: Mittels „embedded Trace"-Lösungen beobachtete Tests und produktive Programmläufe mit identischem Object-Code

Die folgenden Klassifikationen sind als Orientierung zu verstehen und müssen jeweils abhängig vom Kontext des Werkzeugeinsatzes individuell bestimmt werden.

Für die IEC 61508 [86] ergibt sich die Einordnung der Software-Instrumentierung *offline support tool*, welches zur Verifikation oder dem Test des Designs und damit als Werkzeug der Unterkategorie T2 verwendet wird.

Bei der Klassifizierung entsprechend DO-330/ED-215 [49] [93] ergibt sich das Kriterium 2, d.h. es handelt sich um ein Werkzeug, welches den Verifikations-Prozess automatisiert und welches versäumen könnte, ein beobachtbares Fehlverhalten zu entdecken.

Für den Automobil-Bereich erfolgt die Einstufung nach ISO 26262:2011 [35] mit einem *Tool Impact Level* von TI1 (es besteht die Möglichkeit, dass eine Fehlfunktion des Werkzeugs zum Versagen einer Sicherheitsanforderung oder einer Risiko-Kontrollmaßnahme führt). Der *Tool Error Detection Level* kann auf TD1 bzw. TD2 (hohes bzw. mittleres Vertrauen, dass eine Fehlfunktion des Werkzeugs verhindert oder vor dem Inverkehrbringen eines Produkts entdeckt wird) festgelegt werden, so dass sich ein resultierender *Tool Confidence Level* von TCL1 (keine Qualifizierung erforderlich) bzw. TCL2 (Qualifizierung erforderlich) ergibt.

Folgende Einschränkungen sind zu beachten und können unter Umständen eine Änderung der einzelnen Klassifizierungen bewirken:

Eine Änderung des Beobachtungsfokus zur Laufzeit des Programms über eine entsprechende Debug-Schnittstelle birgt die Gefahr, dass der gesendete Bitstrom fehlerhaft ist und den Programmlauf (z.B. durch Änderung der Taktversorgung, Stoppen einer CPU) beeinflusst.

Die Zuverlässigkeit der Übertragung via paralleler Trace-Schnittstelle ist eingeschränkt, da bei den gängigen Schnittstellen (ARM CoreSight; Nexus) das Übertragungsprotokoll über keine Sicherungsschicht verfügt und die Möglichkeit besteht, dass einzelne Bitfehler unerkannt bleiben, dadurch das Beobachtungsergebnis verfälschen und ein eventuelles Fehlverhalten maskieren können.

Kosten

Die Kosten der Beobachtungsmethode setzen sich zusammen aus den durch die „embedded Trace"-Einheit belegten Ressourcen (Chipfläche, Pins zur Ausgabe der Trace-Daten, Platinenfläche für Trace-Stecker) sowie aus den Kosten des Beobachtungswerkzeugs. Je nach Leistungsfähigkeit bewegen sich die Kosten hierfür im Bereich von ca. 1.000€ bis zu mehreren 10.000€. Bei der Auswahl des Beobachtungswerkzeugs sollte auf folgende Punkte geachtet werden:

- leistungsfähige Eingangsstufe mit individueller Kalibrierung (Pegel, Verzögerung) jeder Signalleitung
- eine möglichst große Speichertiefe
- eine leistungsfähige Unterstützung zur Auswertung der Trace-Daten (z.B. Überdeckungsanalysen, Rekonstruktion von Registerinhalten)
- eine möglichst hohe Geschwindigkeit der Offline-Auswertung

Diese Leistungsmerkmale sind bei einfachen (und damit vergleichsweise günstigen) Trace-Werkzeugen oftmals nicht vollständig gegeben.

4.2.11 Möglichkeiten zur Verbesserung der hardware-basierten Beobachtung

„Embedded Trace"-Einheiten in verschiedenen Ausbaustufen sind in modernen SoCs weit verbreitet. Aus den vorherigen Ausführungen lassen sich Limitationen erkennen, die durch geeignete technische Lösungen ganz oder teilweise überwunden werden können.

Latenz / Flexibilität des Beobachtungsfokus / Gleichzeitige Durchführung vieler Beobachtungsaufgaben

Eine deutliche Verbesserung der Beobachtungsmöglichkeiten eines SoCs mit einer „embedded Trace"-Einheit ergibt sich durch ein Trace-Werkzeug, welches in der Lage ist, die Trace-Daten in Echtzeit zu verarbeiten. Damit könnte die Latenz zwischen dem zu beobachtenden Ereignis im SoC und der Verfügbarkeit dieser Informationen für den Beobachter in den Bereich einiger Mikrosekunden oder einiger Millisekunden reduziert werden. Damit ergibt sich die Möglichkeit, bei einem erkannten Taskwechsel den Beobachtungsfokus mittels einer geeigneten Debug-Schnittstelle taskspezifisch zu ändern und die verfügbare Bandbreite zur Ausgabe der Trace-Daten effizienter zu nutzen. Hierbei ist allerdings zu beachten, dass ein Betrieb der Debug-Schnittstelle während des Programmlaufs - besonders bei sicherheitskritischen Anwendungen - ein mögliches Gefährdungspotenzial darstellt, da bei fehlerhafter Neukonfiguration des SoCs dieser direkt beeinflusst werden kann (z.B. unbeabsichtigtes Stoppen von CPUs, Veränderung des Clocks, Veränderung von Daten). Diese Gefahr ist in die Risikobetrachtung mit einzuschließen und führt gegebenenfalls zu einem höheren Aufwand für die Werkzeugqualifizierung. Für die IEC 61508 [86] ergibt sich an dieser Stelle die Einordnung als *online support tool*, bei der DO-330/ED-215 [49] [93] als Kriterium 1 und bei der ISO 26262:2011 [35] eine Einstufung als *Tool Confidence Level* von TCL2 (*Tool Impact Level* von TI1, *Tool Error Detection Level* von TD2). Um das zu beobachtende sicherheitskritische System vor einer ungewollten Beeinflussung zu schützen, könnte als Bestandteil des sicherheitskritischen Systems ein Filter für die Debug- Signale mit implementiert werden, welcher eine ungewollte Beeinflussung nicht zulässt. Die Implementierung dieses Filters erlaubt die Beibehaltung der ursprünglichen Einstufung der Werkzeugqualifizierung.

Neben der Möglichkeit, mit sehr geringer Latenz auf Ereignisse im SoC reagieren zu können, erlaubt die Verarbeitung der Trace-Daten auch eine unbegrenzte Beobachtungsdauer. Die Trace-Daten müssen nicht unbedingt zwischengespeichert werden, vielmehr werden die empfangenen Daten nach ihrer Analyse verworfen. Dies setzt auf Seiten des Trace-Werkzeugs einen sehr hohen Aufwand voraus, da hier kontinuierlich das Transportprotokoll (TPIU) dekodiert werden muss, gefolgt von der Erkennung der Paketgrenzen und der Rekonstruktion der Sprunginformationen. Hieraus werden Ereignisse generiert, die dann mittels nachgeordneter Einheiten verarbeitet werden (Runtime Verification).

Qualifizierung des Beobachters

Eine Verbesserung der Verlässlichkeit der Beobachtung eines SoCs mittels einer parallelen Trace-Schnittstelle kann erreicht werden, wenn eine Sicherungsschicht implementiert wird, welche mittels Prüfsummen die Fehlerfreiheit der übermittelten Trace-Daten sicherstellt. Dies erfolgt einmal durch das Hinzufügen zusätzlicher Signalleitungen zur Übertragung der Prüfsummen. Eine weitere Lösungsmöglichkeit ist das Einbetten der Prüfsummen in den Trace-Datenstrom, dies kann bevorzugt anstelle der Übertragung von Idle-Nachrichten geschehen und führt damit nicht zu einer Erhöhung der erforderlichen Übertragungsbandbreite.

4.3 Zusammenfassung

Die wesentlichen Techniken zur Beobachtung von SoCs sind die Software-Instrumentierung und verschiedene „embedded Trace“-Implementierungen. Alle anderen Methoden haben Einschränkungen, so dass sie für eine vollständige Beobachtung von SoCs im Sinne der im Abschnitt 3 definierten Kriterien nicht effizient einsetzbar sind.

Die Methode der Software-Instrumentierung kann auf einfache Art angewandt werden, bringt aber aufgrund der notwendigen Intrusivität wesentliche Nachteile mit sich.

„Embedded Trace“-Implementierungen stellen die effizienteste verfügbare Methode dar, um SoCs beobachten zu können. In den meisten Fällen steht zur Ausgabe der Trace-Daten nur eine eingeschränkte Bandbreite zur Verfügung, welche die Beobachtungsmöglichkeiten entsprechend begrenzt. Aktuell verfügbare Systeme zur Erfassung und Auswertung der Trace-Daten arbeiten nach dem Prinzip der Zwischenspeicherung und Offline-Auswertung, welche besonders bei kontinuierlicher Beobachtung bei einer hohen Bandbreite an Trace-Daten weitere Limitationen bewirken. An dieser Stelle kann eine Verarbeitung der Trace-Daten in Echtzeit zu einer Verbesserung der Beobachtbarkeit führen.

5 Eine neuartige Emulationslösung

Wie im vorherigen Abschnitt erläutert wurde, existieren aktuell noch eine ganze Reihe von Lücken zwischen den Anforderungen von Entwicklern an die Beobachtbarkeit von SoCs und den tatsächlich verfügbaren technischen Möglichkeiten. Das Hauptproblem aller „embedded Trace“-Lösungen besteht in der limitierten Bandbreite, mit der die Trace-Daten vom SoC in das Emulationssystem übertragen werden. Bei den „embedded Trace“-Lösungen wird ein großer Aufwand betrieben, um die Trace-Daten optimal zu komprimieren. In der Literatur werden eine Reihe komplexer Lösungen vorgeschlagen, um die Komprimierung noch weiter zu verbessern. Allerdings haben diese Lösungen bisher noch keinen Einzug in kommerzielle Produkte gefunden und bieten oftmals eine Reduzierung des mittleren Bandbreitenbedarfs, lösen das Problem der Bandbreitenspitzen aber nicht. Weitere substantielle Verbesserungen der Komprimierung bei dem Prinzip der „embedded Trace“-Lösungen können aufgrund des zu übertragenden Informationsgehalts der Trace-Daten ausgeschlossen werden.

Um diese Limitationen zu überwinden, scheinen strukturell neue Ansätze notwendig, die im Folgenden vorgestellt werden.

5.1 Der Weg zu hidICE

Ausgangspunkt für die Entwicklung der im Folgenden vorgestellten Lösung war das Bestreben, die Beobachtbarkeit von SoCs zu verbessern, welche über keine geeigneten Möglichkeiten zur Ausgabe von Trace-Daten verfügen. Eine erste Idee bestand darin, auf einem PC die CPU des SoCs zu simulieren - zusätzlich sollte die Möglichkeit bestehen, auf die Peripherieeinheiten des SoC zugreifen zu können. Wie in Abbildung 5-1 dargestellt läuft zu diesem Zweck auf dem SoC ein Kernel, welcher über eine geeignete Schnittstelle (z.B. UART) mit dem PC kommuniziert und Lese- und Schreiboperationen auf Control- und Status-Register der Peripherieeinheiten durchführt.

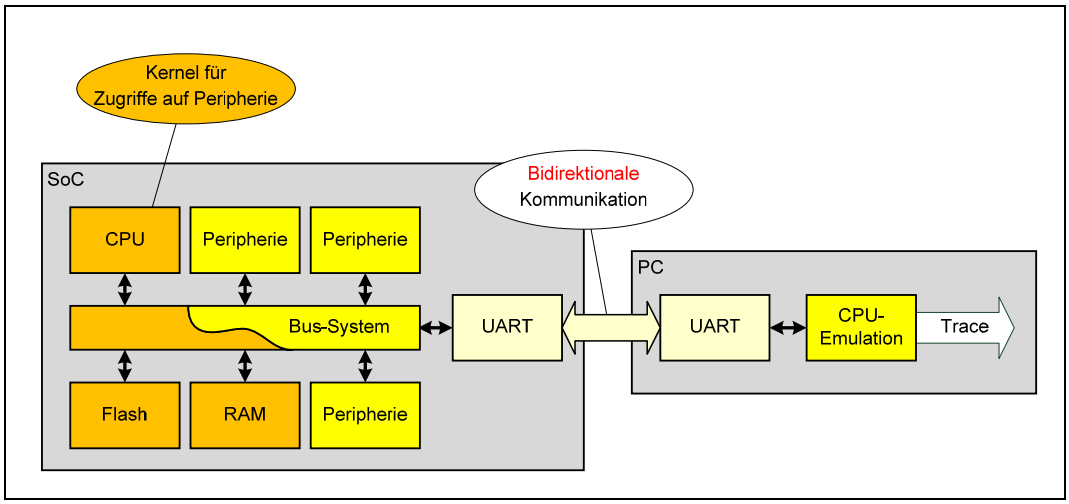


Abbildung 5-1: Emulation auf einem PC mit Zugriff auf die Peripherie-Einheiten des SoC

Problematisch bei dieser Lösung ist die Signalisierung von asynchronen Ereignissen im SoC wie beispielsweise Interrupts. Hierzu müsste der Kernel in der Lage sein, selbstständig dem PC ein derartiges Ereignis zu signalisieren, alternativ könnte der PC periodisch das zwischenzeitliche Auftreten von Ereignissen im SoC abfragen. Bedingt durch die begrenzte Geschwindigkeit der Simulation sowie dem zusätzlichen Zeitbedarf, welcher die Kommunikation mit dem Kernel auf dem SoC erfordert, führt die Anwendung des vorgeschlagenen Verfahrens zu einer massiven Verfälschung des Zeitverhaltens des zu beobachtenden Systems. Der vorgeschlagene Ansatz stößt auch bei dem Vorhandensein von Peripherieeinheiten, welche selbstständig Datentransfers initiieren können,

schnell an seine Grenzen. Trotz dieser Einschränkungen kann die Methode eine deutliche Verbesserung der Beobachtbarkeit bewirken, dies gilt besonders für Systeme, welche über keine Möglichkeit zur Ausgabe von Trace-Daten verfügen. Da es sich um eine reine Softwarelösung handelt, müssen auf dem SoC keine spezifischen Debug-Strukturen implementiert sein. Es wird lediglich eine geeignete Kommunikationsschnittstelle (z.B. ein UART) benötigt, welche auf SoCs üblicherweise immer verfügbar ist.

Eine ähnliche Variante des beschriebenen Ansatzes stellt das „*Virtual Clone*“ Emulationssystem der Firma Dolphin Integration dar (Abbildung 5-2)[127]. Hier läuft die zu beobachtende Anwendung in einem externen Emulationssystem, welches auch mittels eines Software-Simulators auf einem PC realisiert werden kann. Über eine spezielle Schnittstelle (Emulator-Interface) wird das interne Bussystem der Emulation so erweitert, dass für die CPU im Emulator Lese- und Schreibzugriffe auf die Peripherieeinheiten des SoCs möglich sind. Die CPU im SoC ist dabei völlig inaktiv.

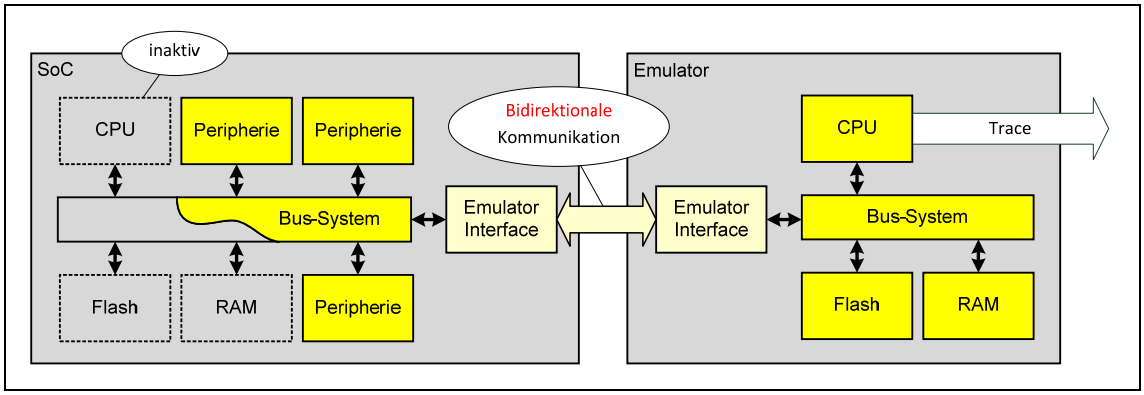


Abbildung 5-2: Prinzip des „*Virtual Clone*“ Emulationssystems von Dolphin Integration

Auch hier treten die zuvor angesprochenen Probleme auf. Selbst wenn die Implementierung der CPU-Emulation ausreichend schnell ist, kommt es zu einer Beeinflussung des zeitlichen Verhaltens aufgrund der Verzögerung des Zugriffs auf die Peripherieeinheiten durch das Emulator-Interface. Weiterhin können Ereignisse im SoC (z.B. Interrupts) dem Emulator nur verzögert bekannt gemacht werden, zudem ist die Unterstützung busmasterfähiger Peripherieeinheiten schwierig oder nicht möglich.

Bei beiden vorgestellten Ansätzen ist eine bidirektionale Kommunikation zwischen dem SoC und dem Emulator notwendig. Dadurch ergibt sich eine Abhängigkeit des Zeitverhaltens der Programmausführung von der Leistungsfähigkeit der jeweiligen Kommunikationsschnittstelle und entspricht nicht dem Zeitverhalten einer Programmausführung in unbeobachtetem Zustand. Weiterhin muss Kommunikationsbandbreite sowohl für Lese- als auch für Schreibzugriffe auf die Peripherieeinheiten vorgehalten werden.

Ausgehend von dem Bestreben, die Abhängigkeit der Programmausführung von der Leistungsfähigkeit der bidirektionalen Kommunikationsschnittstelle aufzuheben, wurde eine alternative Emulationslösung entwickelt. Im Gegensatz zu den bisher diskutierten Lösungen sind nun sowohl die CPU im SoC als auch die emulierte CPU gleichzeitig aktiv. Die Synchronisation erfolgt mittels unidirektionaler Kommunikation und erlaubt damit die Unabhängigkeit des Zeitverhaltens der Programmausführung von der Leistungsfähigkeit der Kommunikationsschnittstelle. Dies stellt eine entscheidende Verbesserung des Beobachtungsprinzips gegenüber den beiden vorher dargestellten Varianten dar.

Der neuen Emulationslösung [128],[129],[130],[131] wurde der Name „hidICE“ (*hidden In-Circuit-Emulator*) gegeben.

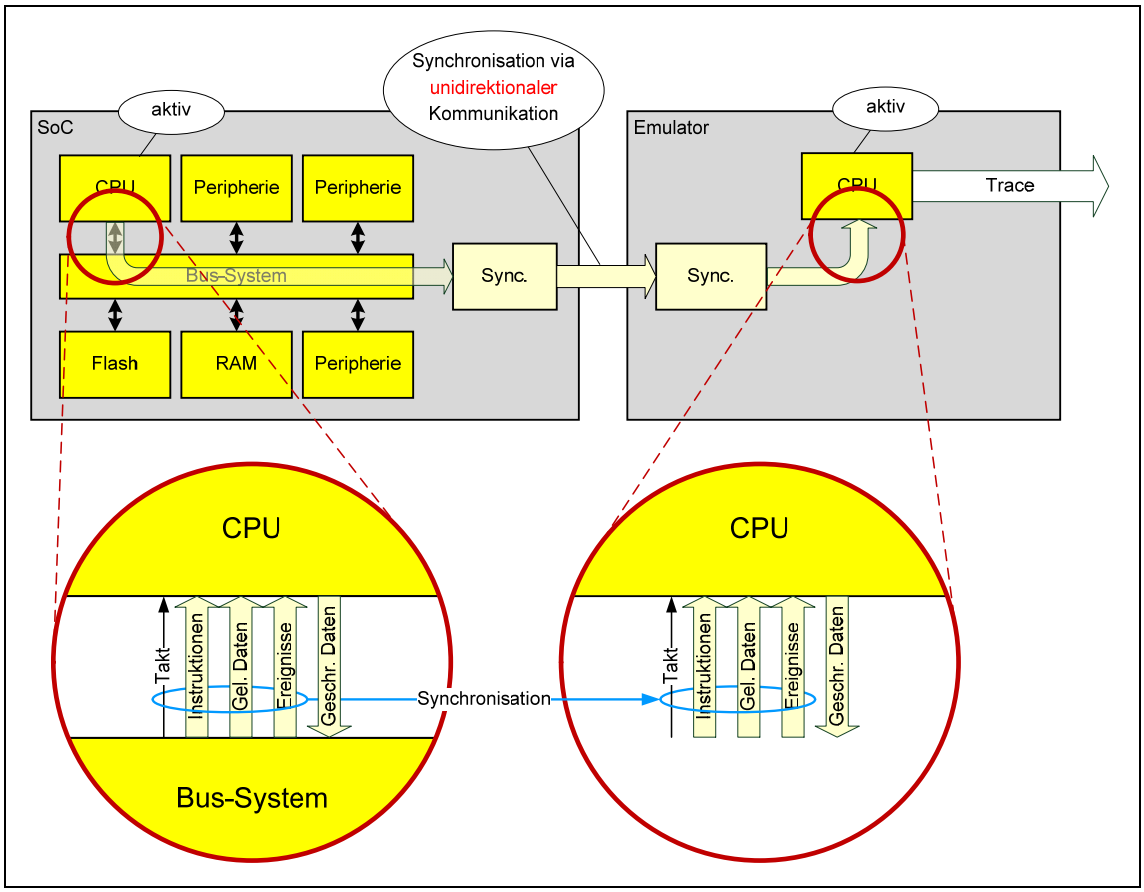


Abbildung 5-3: Neuartiges Emulationsverfahren mit unidirektionaler Kommunikation

Abbildung 5-3 zeigt eine sehr einfache Implementierungsvariante, bei der folgende Informationen über die unidirektionale Kommunikationsschnittstelle zur Emulation übertragen werden:

- CPU-Takt
- Instruktionen
- von der CPU gelesene Daten
- Ereignisse (z.B. Reset, Interrupts, Warte-Zyklen)

In der Emulation werden die Synchronisationsdaten an die Schnittstellen des emulierten Bereiches (in Abbildung 5-3 ist dies vorerst nur die emulierte CPU) angelegt. Diese sind identisch zu den Signalen, die an den Grenzen des für die Emulation vorgesehenen Bereichs des SoC anliegen.

Bedingt durch die Übertragungsdauer der für die Synchronisation erforderlichen Daten arbeitet die Emulation um einige Takte verzögert. Da die Verzögerung wohldefiniert ist und der Datenfluss nur in eine Richtung stattfindet, stellt dies keine funktionelle Einschränkung dar.

Es lässt sich leicht erkennen, dass die Übertragung von Schreibzugriffen für die Synchronisation nicht erforderlich ist. Während die CPU des SoC auf Speicher oder Peripherieeinheiten wie im normalen Betrieb auch schreibt, werden diese Schreibzugriffe von der emulierten CPU auch ausgeführt, können mit protokolliert werden (Daten-Trace) und haben ansonsten keinen weiteren Effekt. Wenn mittels dieser Schreibzugriffe der Zustand von Peripherieeinheiten oder dem Datenspeicher verändert wird, so müssen diese Änderungen in der Emulation nicht berücksichtigt werden. Sie wirken sich nur im Rahmen von Lesezugriffen auf den Programmverlauf aus und werden über den Synchronisationsmechanismus wiederum der emulierten CPU korrekt mitgeteilt.

Voraussetzung für eine exakte Nachbildung der Programmabarbeitung der CPU des SoCs ist die Übereinstimmung der initialen inneren Zustände beider CPUs. Dies kann vorzugsweise durch ein

Reset der CPU des SoCs erfolgen. Dieses Signal wird über die Synchronisations-Schnittstelle an die emulierte CPU übertragen und sorgt dafür, dass diese sich im gleichen Zustand wie die CPU des SoCs befindet. Dies bedeutet allerdings, dass eine Beobachtung einer bereits laufenden Anwendung nicht möglich ist, Voraussetzung für einen synchronisierten Start der Emulation ist die initiale Aktivierung des Reset-Signals.

Alternativ dazu kann auch eine periodische Speicherung der inneren Zustände der CPU des SoCs erfolgen. Diese werden dann als zusätzliche Informationen über die Kommunikationsschnittstelle zur Emulation übertragen und können die emulierte CPU in einen definierten Zustand versetzen, welcher als Startzeitpunkt für eine nachfolgende Beobachtung dient. Dieses Verfahren ist besonders geeignet, wenn die Emulation auf Basis aufgezeichneter Synchronisationsdaten mittels Software-Emulation erfolgt. Hier ist es nicht notwendig, die Synchronisationsdaten seit dem letzten Reset zu speichern, vielmehr ist es möglich, ab Empfang des letzten Satzes von CPU-Zustandsinformationen die darauf folgenden Programmschritte zu berechnen. Diese initiale Idee wurde nicht weiterverfolgt, da ein Eingriff in die Architektur der CPU notwendig ist, um periodisch zeitgleich alle relevanten inneren Zustände erfassen zu können. Weiterhin wird zur Übertragung dieser Informationen zusätzliche Bandbreite der Synchronisations-Schnittstelle benötigt. Bei der bevorzugten Auslegung der Emulation als Hardware-Lösung bedeutet die Forderung nach einem der Beobachtung vorausgehenden Reset in den meisten Anwendungsfällen keine unakzeptable Einschränkung.

Die zur Realisierung der Beobachtbarkeit erforderliche Menge an kontinuierlich vom SoC auszugebenden Informationen ist eine sehr wichtige Eigenschaft einer Beobachtungslösung. Bei „embedded Trace“-Lösungen handelt es sich - wie im vorherigen Abschnitt dargestellt - um die kontinuierlichen Ausgabe von Trace-Nachrichten, bei einer hidICE-basierten Beobachtung sind dies die Synchronisationsinformationen.

Im Folgenden wird eine Strategie zur Minimierung der zu übertragenden Synchronisations-Informationen dargestellt.

Ein erster Schritt besteht darin, auch den Programmspeicher mit in die Emulation einzubinden. Eine identische Initialisierung des Programmspeichers vorausgesetzt, müssen nun nicht mehr die von der CPU des SoC ausgeführten Instruktionen an die Emulation übermittelt werden. Vielmehr kann der Programmcounter der emulierten CPU die als nächstes auszuführenden Instruktionen aus dem Programmspeicher der Emulation holen.

Wird auch der Datenspeicher mit in die Emulation einbezogen, müssen auch die hier gelesenen Werte nicht in den Synchronisations-Informationen enthalten sein. Dies setzt voraus, dass der Inhalt beider Datenspeicher zum Zeitpunkt eines Resets konsistent ist bzw. dass vor einem Lesezugriff auf eine nicht initialisierte Speicherzelle zuerst ein Schreibzugriff erfolgt.

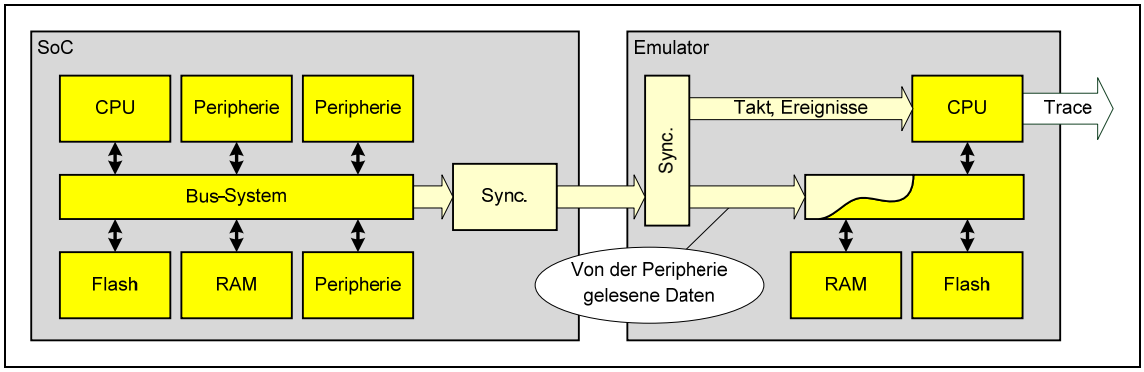


Abbildung 5-4: Erweiterung der Emulation um Programm- und Datenspeicher

Mit diesen beiden in Abbildung 5-4 dargestellten Optimierungen müssen nur noch folgende Informationen übertragen werden:

- CPU-Takt
- von der CPU gelesene Daten aus dem Adressbereich der Peripherieeinheiten
- Ereignisse (z.B. Reset, Interrupts, Warte-Zyklen)

Der hidICE-Ansatz kann auch zur effizienten Beobachtung von MPSoCs angewendet werden. Dazu werden die zusätzlichen CPUs ebenfalls in die Emulation mit eingebunden (Abbildung 5-5).

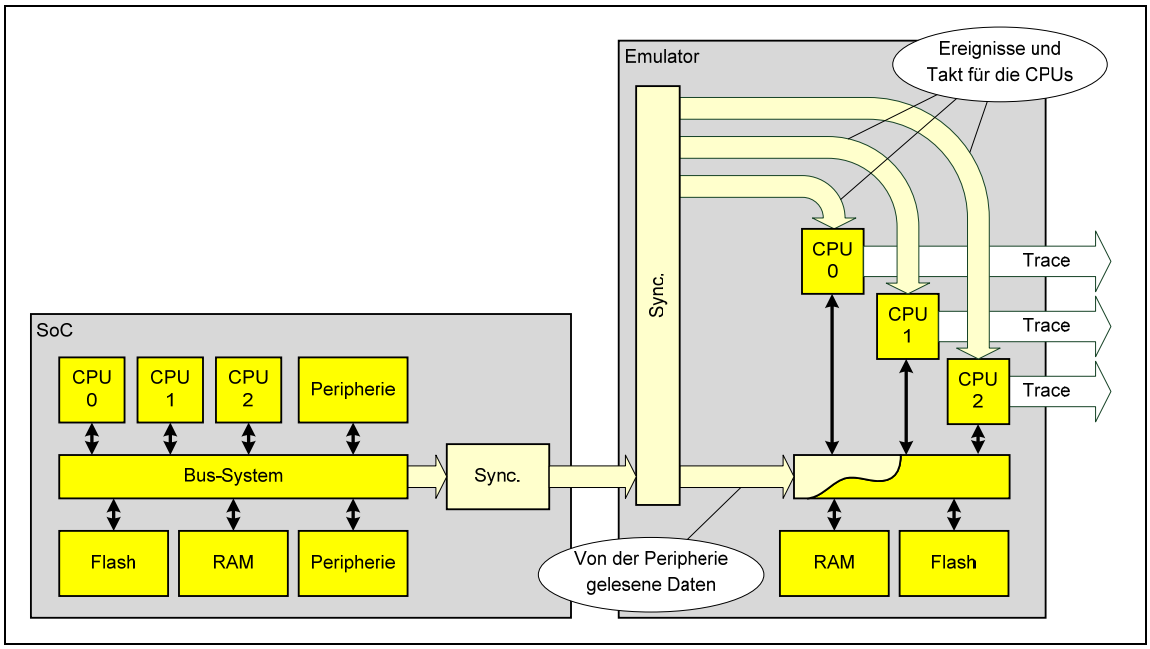


Abbildung 5-5: Beobachtung eines MPSoC

Die zur Synchronisation mehrerer CPUs erforderliche Bandbreite ist meist deutlich niedriger als die zur vollständigen Beobachtung eines MPSoCs via einer „embedded Trace"-Lösung benötigte Bandbreite (Abbildung 5-6). Hinzu kommt noch eine erweiterte Beobachtbarkeit der Vorgänge im SoC, die deutlich über die Möglichkeiten von Trace- Nachrichten aus „embedded Trace"-Lösungen hinausgehen. So ist es beispielsweise möglich, auch Änderungen von CPU-Registern (z.B. Stackpointer, in Registern abgelegte lokale Variable) mit zu verfolgen.

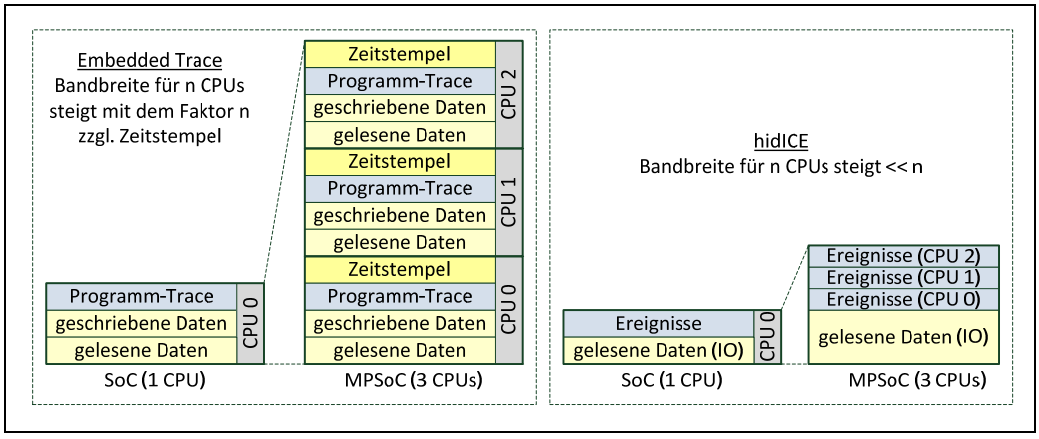


Abbildung 5-6: Bandbreitenbedarf einer hidICE-basierten Emulation vs. einer „embedded Trace"-Lösung

5.2 Synchronisation und Emulation

Wie einführend schon dargestellt kann der Umfang des in der Emulation nachgebildeten Teils des SoCs unterschiedlich groß gewählt werden. Das Prinzip ist immer das Gleiche: Zur Synchronisation müssen alle zur Laufzeit über die Systemgrenze (Abgrenzung des emulierten Bereiches) eingehenden Informationen übermittelt werden.

5.2.1 Einfache SoCs

Die in diesem Abschnitt vorgestellten Implementierungen beziehen sich auf eine einfache SoC-Architektur mit jeweils einem Busmaster sowie optional einem busmasterfähigen DMA-Controller, dessen Transfer aber durch die CPU kontrolliert wird. Je nach Architektur des zu beobachtenden SoCs kann die Implementierung differieren, da z.B. einzelne Signale nicht ohne weiteres zugänglich sind. In diesen Fällen müssen andere Informationen für die Synchronisation übertragen werden, das Grundprinzip bleibt aber erhalten.

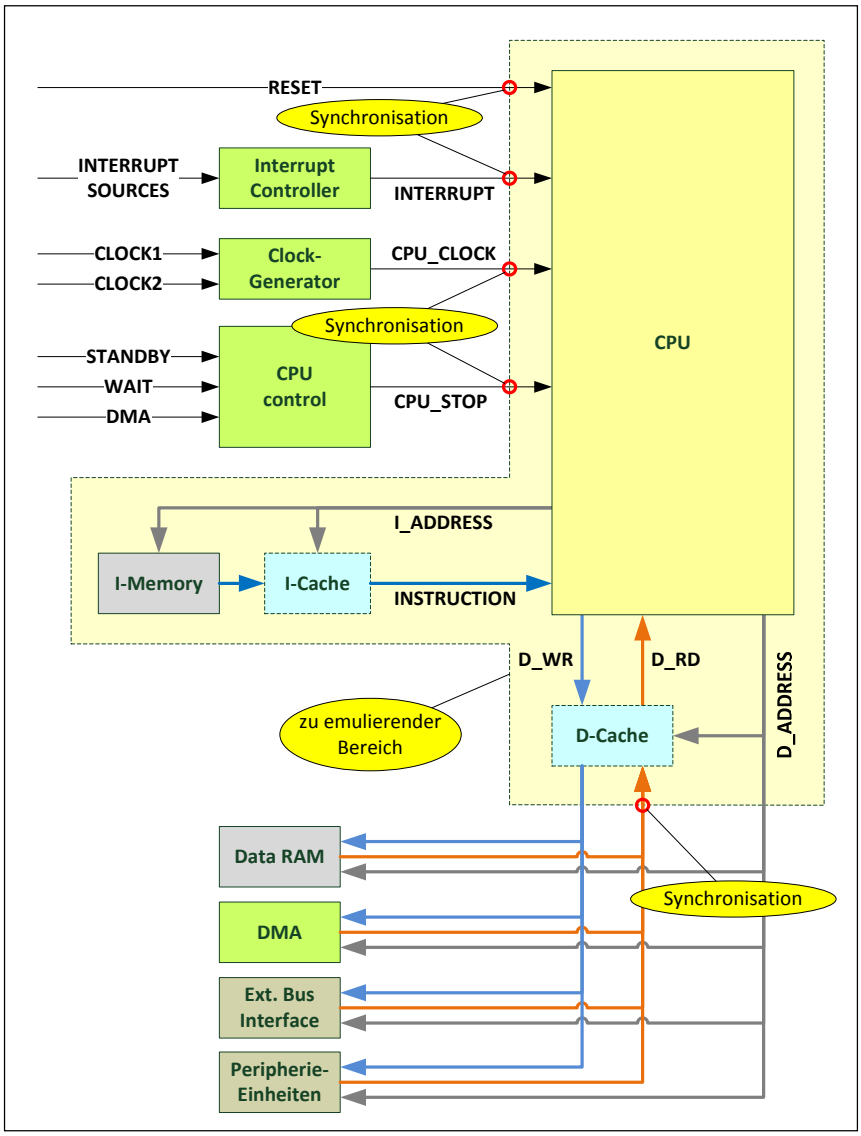


Abbildung 5-7: Emulierter Bereich und zugehörige Synchronisationsinformationen ohne Einbeziehung des Daten-RAMs

In Abbildung 5-7 ist eine SoC-Architektur dargestellt, welche aus einem CPU-Kern mit getrennten Programm- und Datenbussen (Harvard-Architektur) besteht. Die angeschlossenen Speicher können optional über einen Cache verfügen.

Bei der vorgestellten Implementierungsvariante wird neben der CPU noch der zugehörige Programmspeicher sowie ggfs. Programm- und Datencaches in die Emulation einbezogen.

Die in Tabelle 5-1 aufgeführten Informationen müssen in dieser Implementierungsvariante zur Synchronisation übertragen werden, wobei alle Signale synchron zum CPU-Takt (CPU_CLOCK) sein müssen. Liegen hier nicht synchrone Signale vor (z.B. Reset-Signal), so müssen diese im zu beobachtenden SoC mittels geeigneter Schaltungen auf den CPU-Takt synchronisiert werden.

Die in Tabelle 5-1 verwendeten Parameter IL_WIDTH, IN_WIDTH und DATA_WIDTH sind in Tabelle 5-2 erläutert.

Signal	Erklärung	Signalbreite
CPU_STOP	Halte-Signal für die CPU, welches aus externen Warte-anforderungen generiert wird. Ursache eines CPU-Halts können z.B. sein: - WAIT: externe Halteanforderung an die CPU, z.B. beim Erreichen eines Haltepunktes oder bei Verzögerung eines Bustransfers. - DMA: Die Belegung des Busses durch einen DMA-Transfer wird signalisiert. - STANDBY: Die CPU wird in einen Ruhezustand gebracht.	1
RESET	Reset-Signal für die CPU	1
INTER-RUPT	Signalisierung des aktuellen Interrupt-Levels und der zugehörigen Interrupt-Nummer	IL_WIDTH + IN_WIDTH
CPU_CLOCK	CPU-Takt, welcher aus verschiedenen Quellen generiert werden kann. Aufgrund der synchronen Emulation kann der CPU-Takt veränderlich sein (z.B. Verlangsamung des CPU-Takts zur Energieeinsparung, schwankender CPU-Takt zur Optimierung des EMV-Verhaltens).	1
D_RD	vom Datenbus gelesene Daten	DATA_WIDTH

Tabelle 5-1: Synchronisationssignale für eine exemplarische hidICE-Implementierung nach Abbildung 5-7

Aus der Summe der in Tabelle 5-1 angegebenen Signalbreiten lässt sich die zur Synchronisation erforderliche Anzahl von I/O-Pins *N* bestimmen.

Dazu werden die in Tabelle 5-2 angegebenen Parameter verwendet.

Parameter	Erklärung
IL_WIDTH	Breite der Interrupt-Level In SoCs werden üblicherweise bis zu 8 Interrupt-Levels verwendet, so dass das entsprechende Signal 3 Bit breit ist.
IN_WIDTH	Breite der Interrupt-Nummer In SoCs werden üblicherweise bis zu 256[19] verschiedene Interrupts / Exceptions verwendet, so dass das entsprechende Signal 8 Bit breit ist.
DATA_WIDTH	Breite des Datenbusses
CTRL_WIDTH	Breite der Control-Signale (Reset, CPU-Stop)
CLK_PER_READ	Anzahl der CPU-Takte pro Leseoperation auf dem Datenbus

Tabelle 5-2: Parameter zur Bestimmung der für die Synchronisation erforderlichen Bandbreite

Ohne weitere Optimierung wären für einen exemplarischen 32-Bit-SoC mit den Parametern

- IL_WIDTH = 3
- IN_WIDTH = 8
- CTRL_WIDTH = 2 (CPU-STOP, RESET)
- DATA_WIDTH = 32
- CLK_PER_READ = 1

insgesamt 46 I/O-Pins (45 Daten, 1 Clock) erforderlich:

$$DATA\_WIDTH / CLK\_PER\_READ + IL\_WIDTH + IN\_ WIDTH + CTRL\_WIDTH + 1 = 46$$

Diese Synchronisationsinformationen werden über eine geeignete Schnittstelle zur Emulation übertragen und dort an die Schnittstellen einer funktional identischen Kopie des SoC-Kerns verfügbar gemacht (Abbildung 5-8). Unter der Voraussetzung, dass der Programmcode identisch ist, wird sich nun der emulierte Bereich genauso verhalten wie der entsprechende Bereich im SoC. Schreiboperationen der CPU auf Bereiche außerhalb des emulierten Bereiches bleiben wirkungslos, sie sind aber beobachtbar.

Durch die Implementierung geeigneter Schnittstellen können in der Emulation alle relevanten Signale taktgenau bequem beobachtet werden. Dies sind beispielsweise:

- Ausgeführte Instruktionen
- Von der CPU gelesene und geschrieben Daten sowie die zugehörigen Adressen
- CPU-Register
- Busaktivitäten
- Verhalten der Caches

[19] Beispielsweise können bei der Spansion FM4-Familie (ARM Cortex M4) 143 Exceptions auftreten [132], beim PIC24x und dsPIC33x sind es 256.

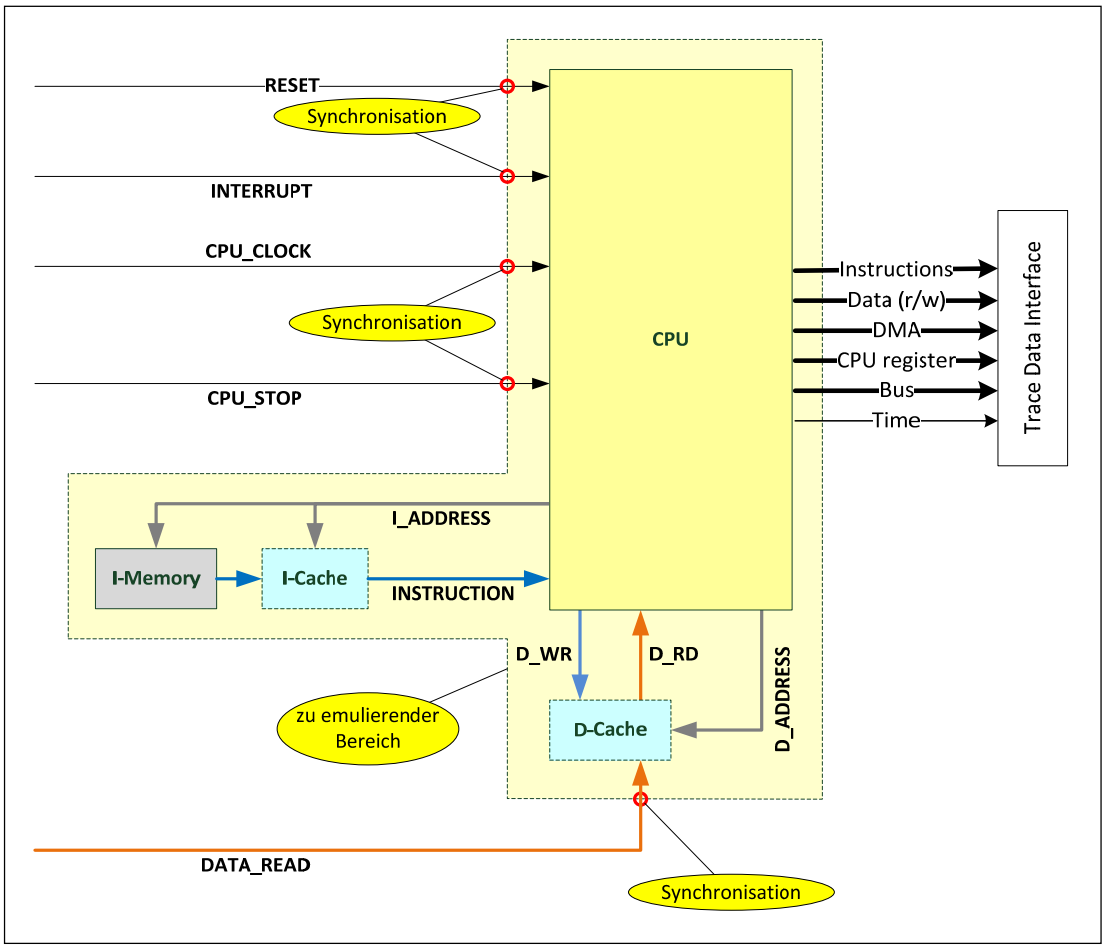


Abbildung 5-8: Emulation entsprechend der Implementierung in Abbildung 5-7

Da die für die beispielhafte Implementierung entsprechend Abbildung 5-7 und Abbildung 5-8 erforderliche Anzahl von I/O-Pins sehr hoch ist, ist es sinnvoll, weitere Schritte zu deren Reduktion zu unternehmen.

Unter Berücksichtigung der Tatsache, dass Interrupts nur relativ selten auftreten, können die zugehörigen Signale zur Übertragung serialisiert werden, wobei im zu beobachtenden SoC dann eine entsprechende Deserialisierung stattfinden muss, um die Synchronität zu gewährleisten. Eine detaillierte Beschreibung dieses Schrittes ist in Abschnitt 5.2.2 zu finden.

Die Serialisierung der für die Synchronisation eines Interrupts erforderlichen Informationen bewirkt, dass beispielsweise die Summe der Parameter IL_WIDTH = 3 und IN_WIDTH = 8 von ursprünglichen Wert von 11 auf 1 reduziert werden kann. In die serialisiert übertragenen Informationen kann zudem auch noch der Reset als Sonderform eines Interrupts einbezogen werden, so dass der Parameter CTRL_WIDTH nur noch die Breite 1 (CPU-STOP) hat.

Die damit einhergehende Verzögerung eines Interrupt-Requests / Resets um einige (im Beispiel: 11) CPU-Takte hat in den meisten Fällen keinerlei nachteilige Auswirkung auf die Funktion des SoCs.

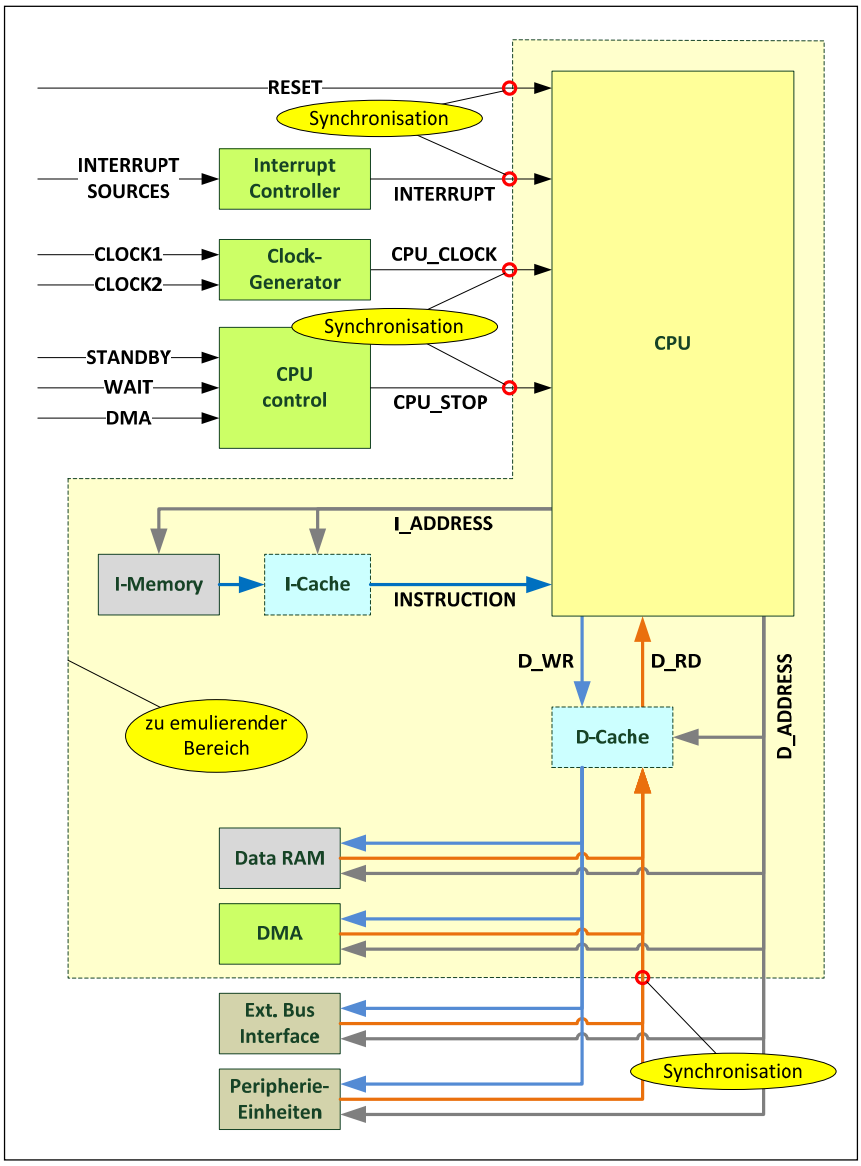


Abbildung 5-9: Erweiterung des emulierten Bereiches

Eine weitere Reduktion der für die Synchronisation erforderlichen Bandbreite ergibt sich, wenn das sich auf dem SoC befindliche Daten-RAM mit in die Emulation einbezogen wird. Dieses ist für schnelle Zugriffe optimiert, während Zugriffe auf die Peripherie sowie das externe Businterface meist mehre Taktzyklen benötigen und auch nicht immer über die volle Busbreite erfolgen.

Damit lässt sich der Parameter DATA_WIDTH vom ursprünglichen Wert (32), der die Breite der Anbindung an den internen Daten-RAM berücksichtigt, auf die Breite des Peripheriebusses reduzieren, die hier exemplarisch mit 16 Bit angenommen wird.

Weiterhin führt die CPU in den meisten Programmen zwischen Lesezugriffen auf die Peripherie mindestens eine andere Instruktion aus (z.B. eine arithmetische Operation), so dass Lesezugriffe auf die Peripherie (architekturabhängig) im Abstand von mindestens drei CPU-Takten (zwei CPU-Takte pro Peripheriezugriff, ein CPU-Takt für weitere Instruktion) erfolgen. Somit kann der Wert für CLK_PER_READ von ursprünglich einer möglichen Leseoperation pro CPU-Takt auf mindestens 3 CPU-Takte pro Lesezugriff erhöht werden. Die dieser Annahme zugrunde liegende Anforderung an die Abfolge von Leseoperationen kann auch durch eine entsprechende Modifikation des Compilers erfüllt werden (siehe Abschnitt 5.2.3).

Die entsprechenden Implementierungen sind in Abbildung 5-9 und Abbildung 5-10 dargestellt, hier ist in dem emulierten Bereich auch das Daten-RAM sowie der DMA-Controller mit einbezogen.

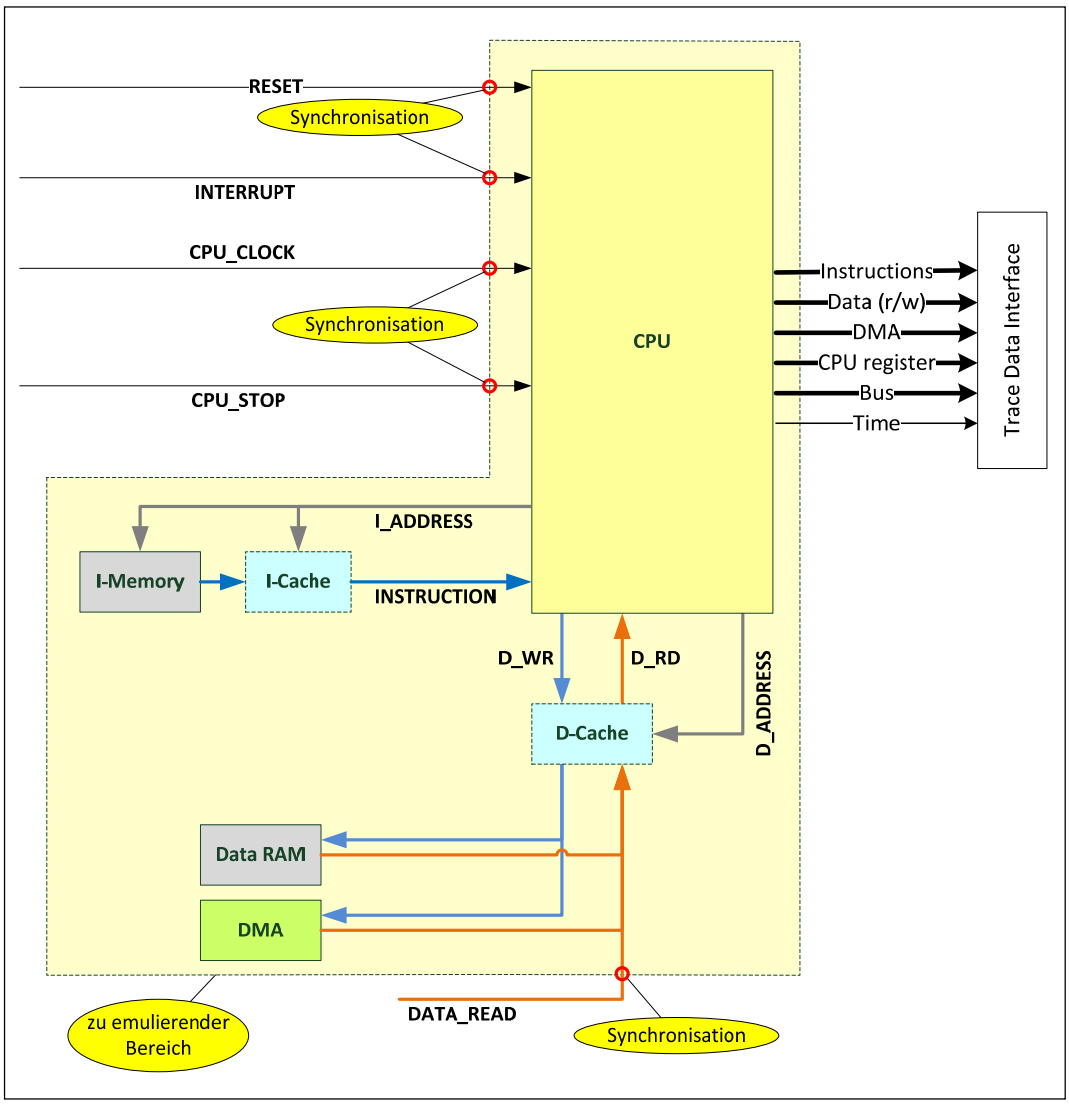


Abbildung 5-10: Emulation entsprechend der Implementierung in Abbildung 5-9

Durch das zeitliche Multiplexen von Signalen kann ebenfalls eine weitere Reduktion der benötigten I/O-Pins erreichet werden, im einfachsten Fall geschieht dies durch Übertragung der Synchronisationssignale mittels *Double Data Rate* (DDR).

Zusammenfassend führen die dargestellten Optimierungen zu folgenden Änderungen der Parameter:

(IL_WIDTH + IN_WIDTH)	= 1
CTRL_WIDTH	= 1
DATA_WIDTH	= 16
CLK_PER_READ	= 3

Mit diesen Annahmen (die bei einer Implementierung jeweils architekturspezifisch angepasst werden müssen) reduziert sich die Synchronisationsschnittstelle auf fünf I/O-Pins (4 Daten, 1 Clock):

$$N = ½ * (DATA\_WIDTH / CLK\_PER\_READ + CTRL\_WIDTH + (IL\_WIDTH + IN\_WIDTH)) + 1 = 4,7$$

In Zusammenarbeit mit einigen SoC-Herstellern wurden Abschätzungen über die Anzahl der für die Synchronisation erforderlichen Pins vorgenommen. Eine Auswahl der Ergebnisse ist in Tabelle 5-3 aufgelistet[20].

Architektur	Beschreibung	Anzahl der benötigten I/O-Pins (DDR @ CPU-Clock)
Microchip PIC24F/H	16 Bit Mikrocontroller	6
Microchip dsPIC33x	16 Bit Digital Signal Controller	4
Renesas V850E2R	32 Bit CPU	8
Fujitsu FR80	32 Bit CPU	8
Toshiba TLCS-900/L1	16 Bit CPU	4
Toshiba TLCS-900/H2	32 Bit CPU	6

Tabelle 5-3: Abschätzung der Anzahl der für die Synchronisation erforderlichen I/O-Pins für verschiedene SoC-Architekturen

[20] Für die Berechnungen wurden nicht öffentlich verfügbare Architekturinformationen verwendet, deshalb muss an dieser Stelle auf eine detaillierte Herleitung verzichtet werden. Die Implementierungsvarianten wurden von SoC-Herstellern begutachtet.

5.2.2 Komplexe SoCs

In Abbildung 5-11 ist ein Beispiel für die hidICE-basierte Beobachtung eines MPSoCs dargestellt. Der emulierte Bereich umfasst dabei alle CPUs, das angeschlossene Bussystem sowie den internen Speicher. Die für die Synchronisation relevanten Informationen werden im SoC erfasst und über die Synchronisations-Schnittstelle an die Emulation übermittelt.

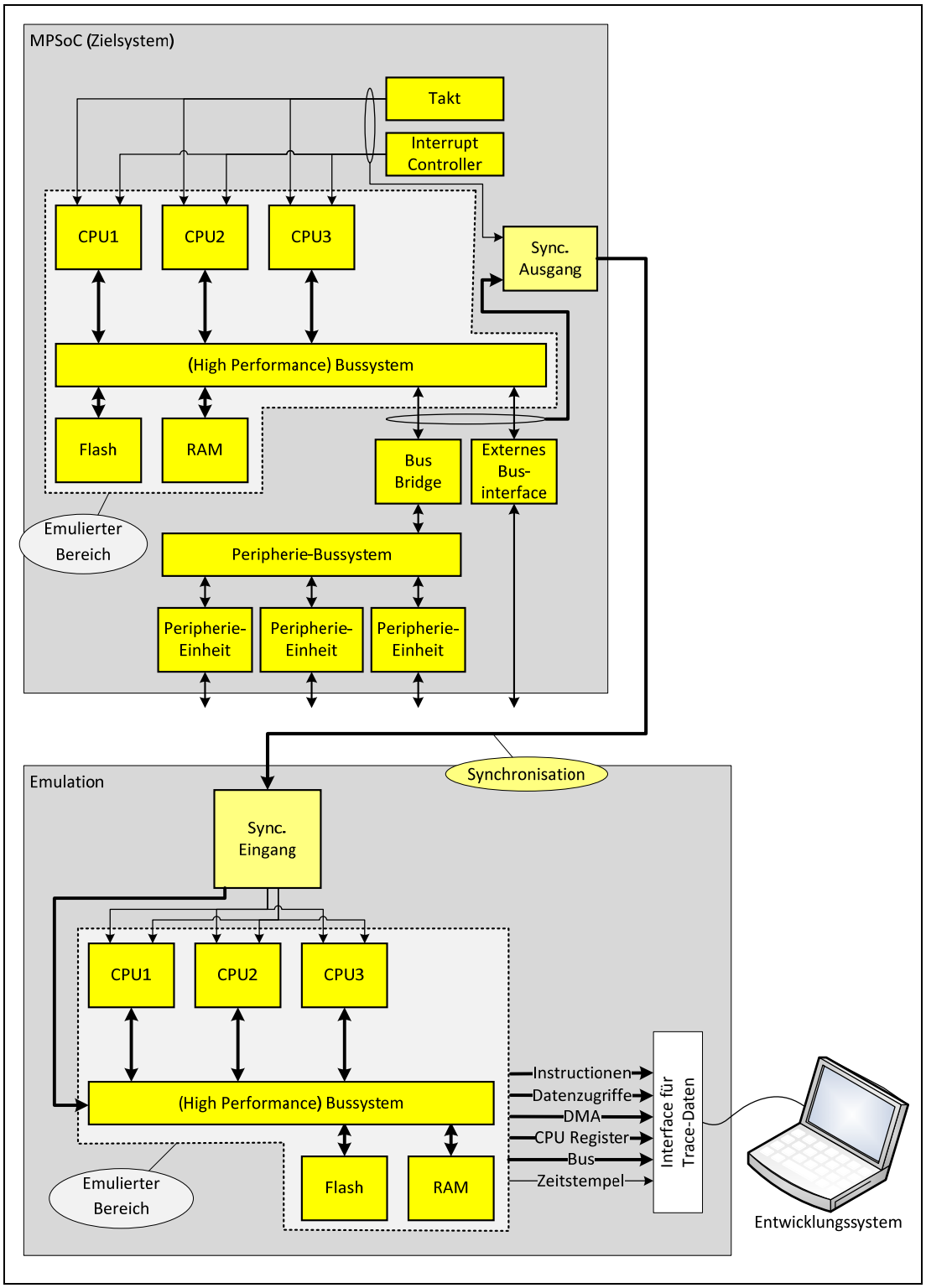


Abbildung 5-11: Beobachtung eines MPSoCs mit einer hidICE-basierten Synchronisation

Die Realisierbarkeit der dargestellten Emulation eines MPSoCs wurde mittels einer FPGA- Implementierung erfolgreich evaluiert. Abbildung 5-12 zeigt ein aus drei SPARC V8 CPUs (LEON3) bestehendes Demonstrationssystem, welches in Abschnitt 5.10.3 detailliert erläutert wird.

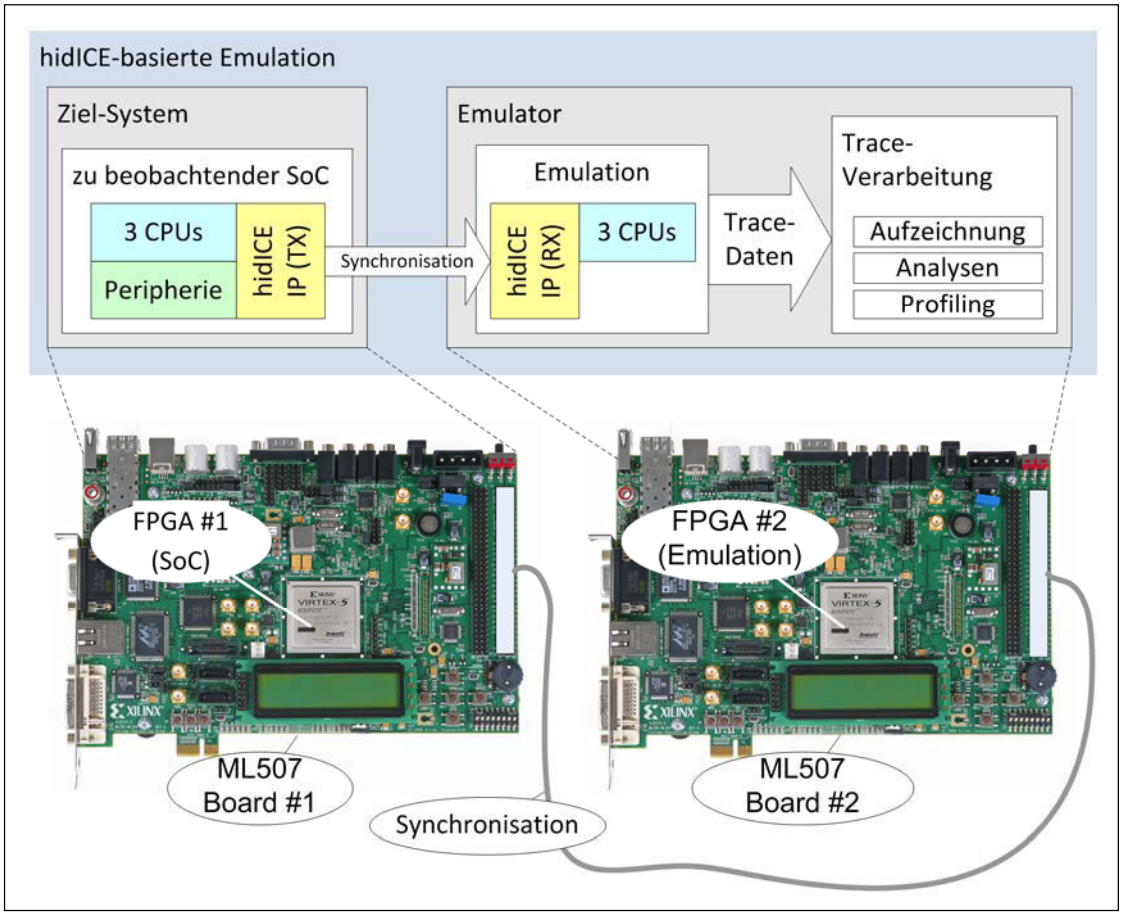


Abbildung 5-12: SPARC V8 Multicore-System mit 3 CPUs und hidICE-basierter Emulation

Die Implementierung der Synchronisation einer Emulation für komplexe SoCs kann sehr aufwändig sein. Schwierigkeiten bereiten hier besonders busmasterfähige Peripherie-Einheiten, komplexe Bussysteme und unterschiedliche Clock-Domains.

Synchronisation busmasterfähiger Peripherie-Einheiten

Moderne SoCs verfügen über Peripherieeinheiten, die über schnelle Businterfaces wie z.B. Flexray, Ethernet oder USB kommunizieren können. Um beim Daten-Transfer die CPU-Last gering zu halten, sind diese Peripherieeinheiten mit eigenen DMA-Controllern ausgestattet, die ohne Beteiligung der CPU in Abhängigkeit von den empfangenen Daten Transfers vornehmen können.

Problematisch ist, dass der serielle Datenstrom asynchron zum CPU-Takt läuft. Innerhalb der Peripherieeinheit werden die Daten interpretiert und dann die entsprechenden Bustransfers initiiert.

Zur Synchronisation muss ein Eingriff in die jeweilige Peripherieeinheit vorgenommen werden. Ziel ist es hier, zum CPU-Takt synchrone Signale zu identifizieren, welche die eingehenden Daten beinhalten, diese auszukoppeln und zu übertragen. Dieses Vorgehen soll an zwei Beispielen erläutert werden.

Beim ersten Beispiel (Abbildung 5-13) handelt es sich um einen busmasterfähigen UART (nach [133]), der zu Debug-Zwecken unabhängig von der CPU im kompletten Adressraum des Busses le-

sen und schreiben kann. Die eingehenden seriellen Signale sind asynchron zum CPU-Takt und müssen zu diesem synchronisiert werden. Dazu werden die Signale, nachdem sie im *Receive shift register* parallelisiert wurden, mit dem CPU-Takt gelatcht. Für eine effiziente Synchronisation erfolgt anschließend eine Serialisierung, die resultierenden Signale werden dann auch an die Emulation weitergegeben. Die Deserialisierung erfolgt auf beiden Seiten, die wieder hergestellten Daten werden dem jeweiligen *AHB master interfaces* übergeben, welche dann die Daten interpretieren und eigenständige Operationen auf dem AHB-Bus ausführen. Die Verzögerung, welche der Prozess der Serialisierung und der Deserialisierung mit sich bringt, beträgt nur einige Takte und ist im Allgemeinen vernachlässigbar.

Mit dieser Lösung ist gewährleistet, dass von der CPU unabhängig durchgeführte Buszugriffe auch in der Emulation taktsynchron reproduziert werden können.

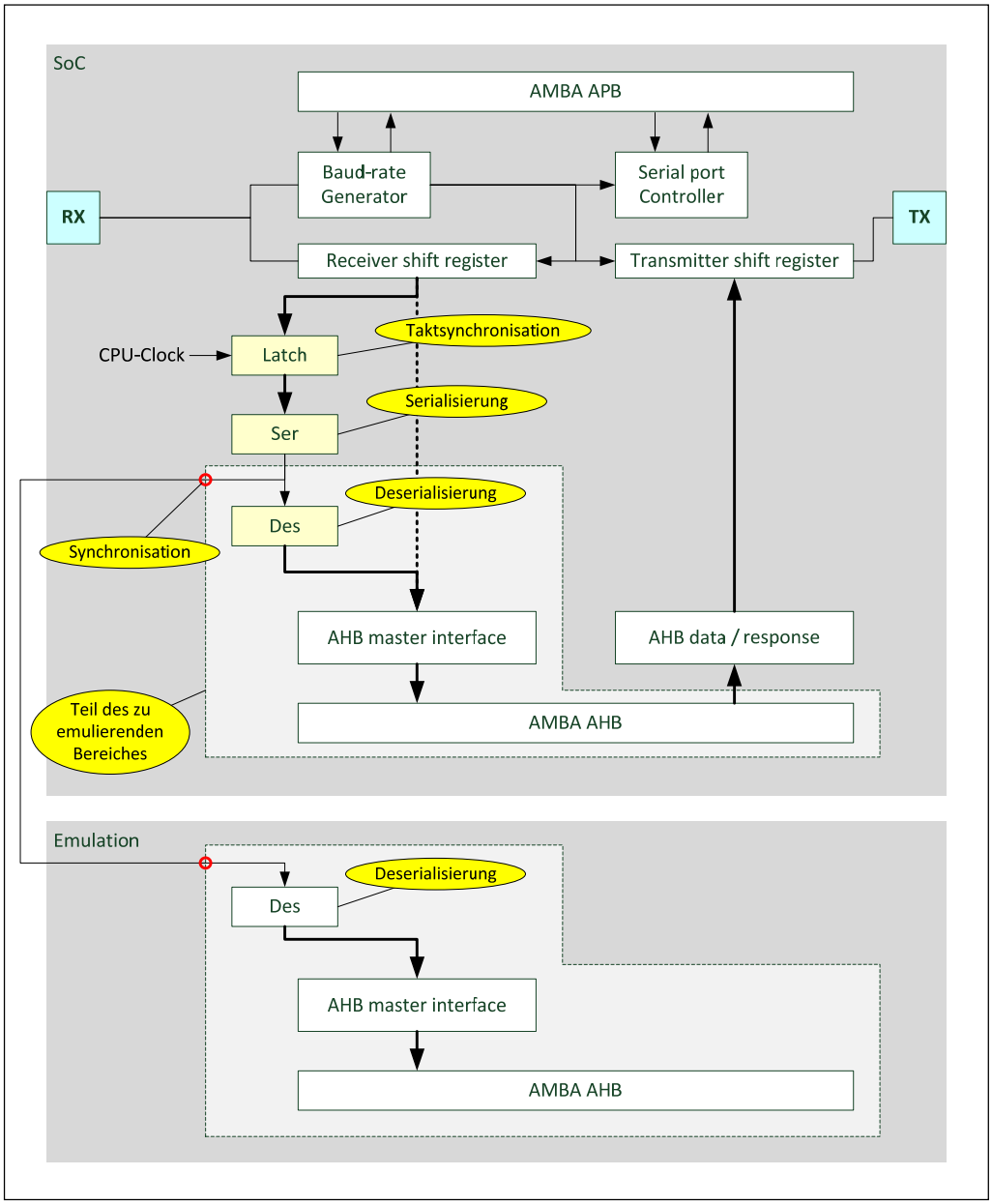


Abbildung 5-13: Synchronisation einer busmasterfähigen Peripherieeinheit

Abbildung 5-14 zeigt ein weiteres Beispiel, welches das Verfahren an einer Ethernet-Schnittstelle (nach [133]) darstellt. Hier müssen Daten mit einer wesentlich höheren Bandbreite als bei dem zuvor diskutierten UART verarbeiten werden.

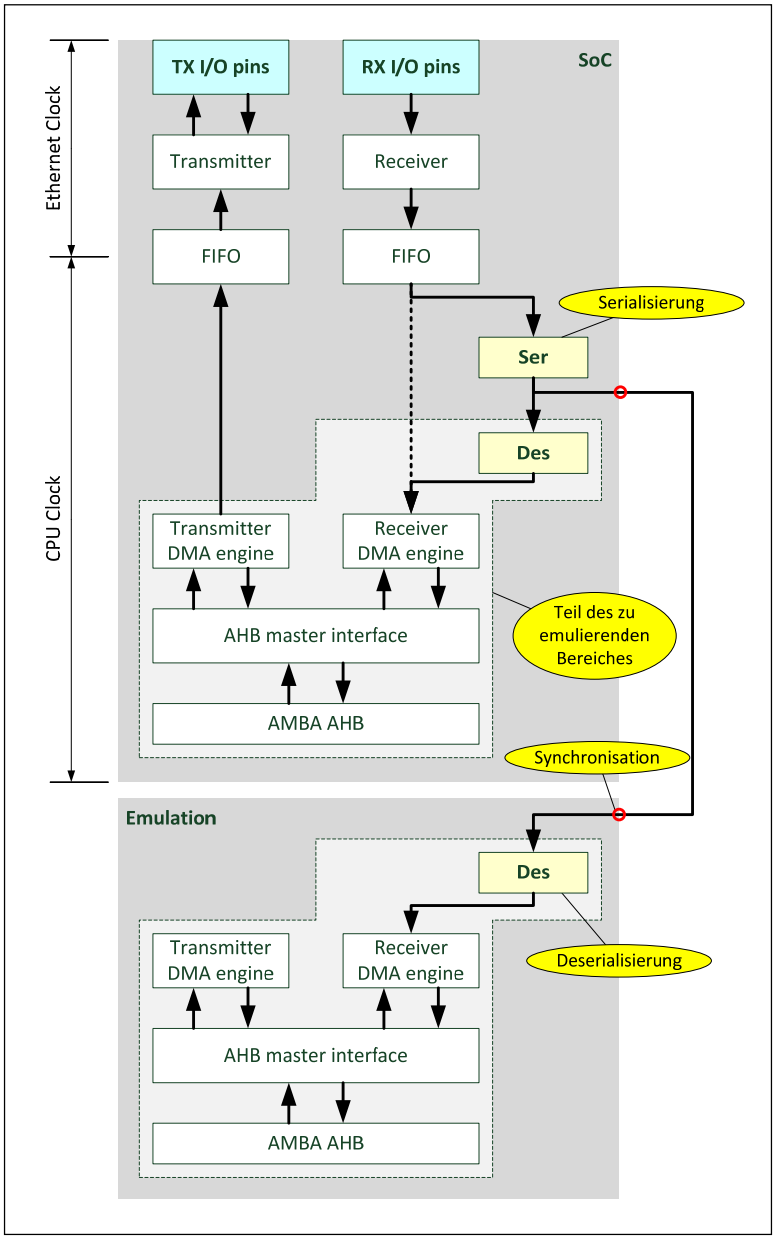


Abbildung 5-14: Synchronisation einer busmasterfähigen Peripherieeinheit

Die eingehenden Signale werden an einer geeigneten Stelle abgegriffen und zur Emulation übermittelt. An dieser Stelle zeigen sich die Grenzen einer hidICE-basierten Emulation. Je mehr busmasterfähige Peripherieeinheiten mit entsprechender Bandbreite Informationen in den SoC einlesen, desto höher wird die für die Synchronisation erforderliche Bandbreite. Aber auch alternative Lösungen zur Beobachtung von SoCs stoßen hier an ihre Grenzen, wenn die konkurrierende Beobachtung von Daten-Transfers ermöglicht werden soll.

Synchronisation mehrerer CPU-Kerne

In MPSoCs können die einzelnen CPU-Kerne mit unterschiedlichen Taktfrequenzen arbeiten. Bisher bestand in den meisten Fällen keine Notwendigkeit, diese verschiedenen CPU-Takte zu synchronisieren, da die Kopplung der SOCs untereinander über das Bussystem erfolgt. Dieses stellt Mechanismen bereit, welche die unterschiedlichen Clock-Domains miteinander koppeln.

Für die Synchronisation zwischen dem SoC und der Emulation ist es hingegen zwingend notwendig, dass die Taktfrequenzen der einzelnen CPUs und des Busses zwar unterschiedlich sein können, die einzelnen Takte aber in festem Bezug zueinander stehen. Dies kann beispielsweise dadurch erreicht werden, dass ein hochfrequenter Takt nicht geteilt sondern „gegated“ wird, um eine gewünschte niedrigere Taktfrequenz zu erreichen oder dass der Clockgenerator phasensynchrone Takte erzeugt (Abbildung 5-15).

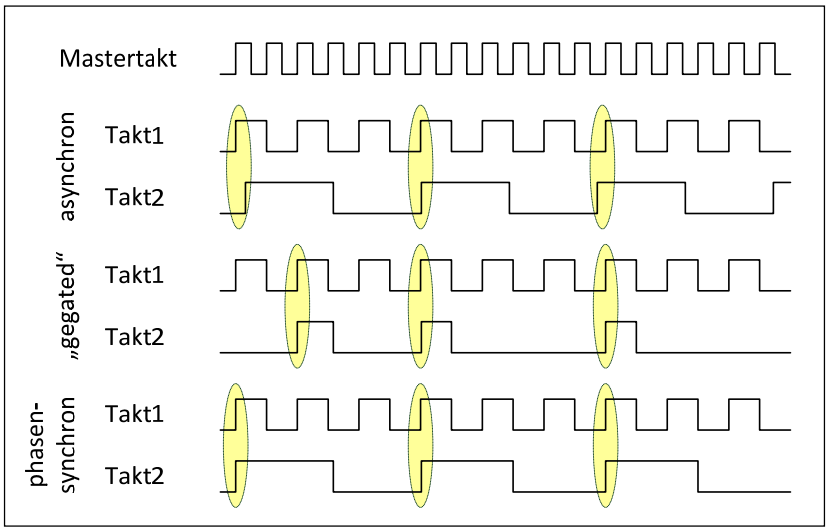


Abbildung 5-15: Verschiedene Möglichkeiten zur Takt-Synchronisation

Die Herstellung der Taktsynchronität bringt auch den Vorteil mit sich, dass Wettlaufbedingungen auf Taktebene im SoC vermieden werden.

Synchronisation von externem Speicher

Bei leistungsfähigen SoCs lässt sich oftmals der interne Speicher durch externen Speicher ergänzen. In vielen Fällen besteht dieser aus preisgünstigem SDRAM unterschiedlicher Technologien (SDR, DDR2, DDR3), welcher während des Bootvorgangs mit Programmcode beschrieben wird, der in einem preisgünstigen (aber langsameren) Flash-Speicher abgelegt ist.

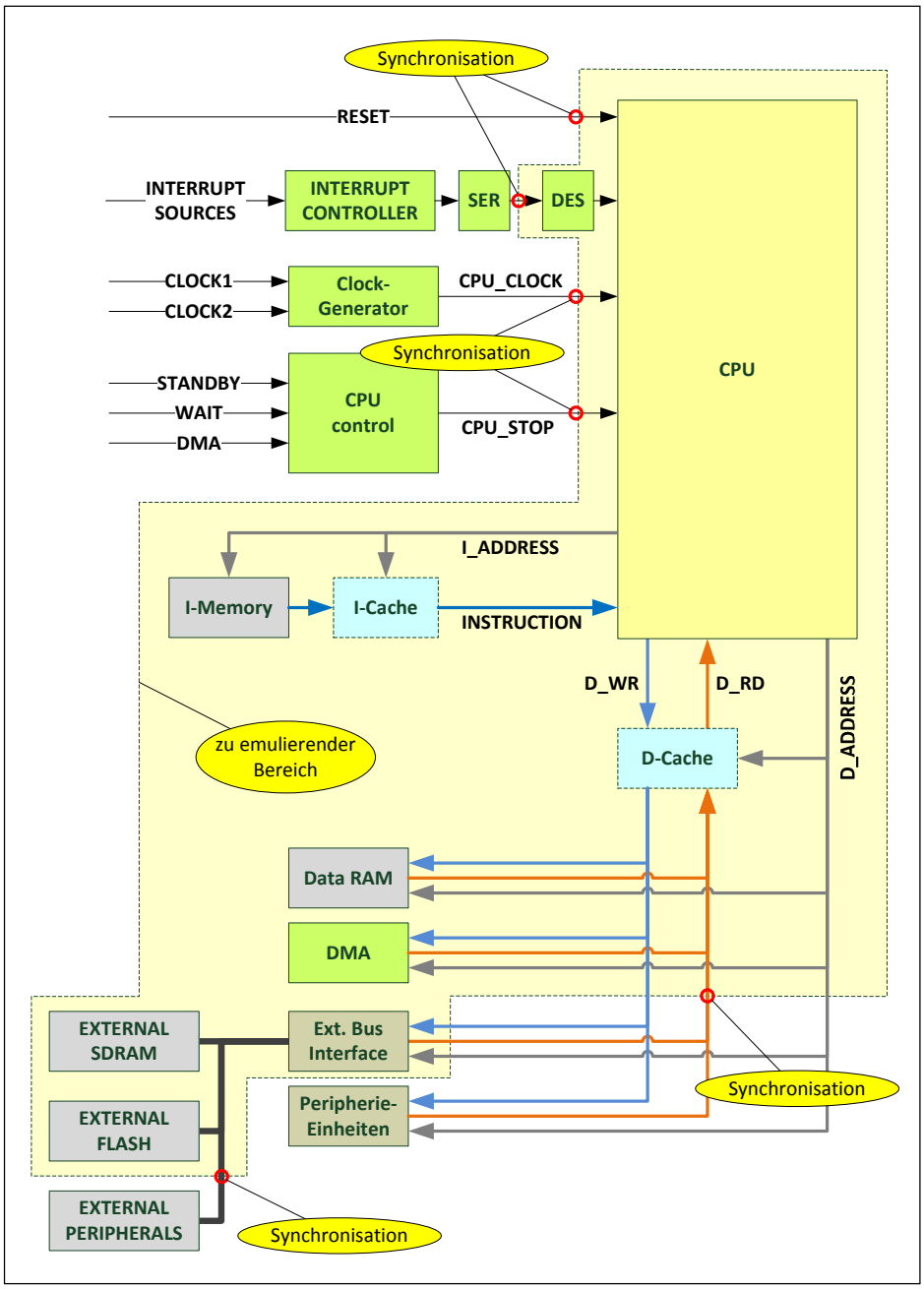


Abbildung 5-16: Einbeziehung von externem Speicher in die Emulation

Um die für die Synchronisation erforderliche Bandbreite zu reduzieren, kann es sinnvoll sein, bei derartigen Systemen den externen Speicher mit in die Emulation einzubeziehen (Abbildung 5-16). Dabei sind die Initialisierungssequenzen der externen Speicher zu berücksichtigen. Der Synchronisationsmechanismus muss gewährleisten, dass sich beide Systeme auch bei verschieden lang andauernden Initialisierungssequenzen (z.B. initiale Kalibrierung bei DDR3-SDRAMs) bei SoC und Emulation nach Abschluss der Initialisierung in identischen Zuständen befinden. Weiterhin müssen die Taktsignale, die zum Betrieb des externen Speichers benötigt werden, phasensynchron zum Takt des jeweiligen Businterfaces laufen.

Die Möglichkeit des Synchronisation von externen DDR-SDRAMs wurde mit einem Hersteller derartiger Speicher diskutiert [134] und grundsätzlich als implementierbar befunden, im Rahmen dieser Arbeit aber nicht experimentell validiert.

Synchronisation von Interrupt-Controllern

Interrupt-Controller werten eine Vielzahl von Interrupt-Anforderungen verschiedener Interrupt-Quellen aus und veranlassen die angeschlossenen CPUs zur Unterbrechung der Arbeit auf einer niedereren Prioritätsebene, wenn ein Ereignis höherer Priorität anliegt.

Idealerweise müssen zur Synchronisation zwischen SoC und Emulation nur die aktuell höchste Priorität sowie die zugehörige Interrupt-Nummer an die CPU übermittelt werden (Abbildung 5-17).

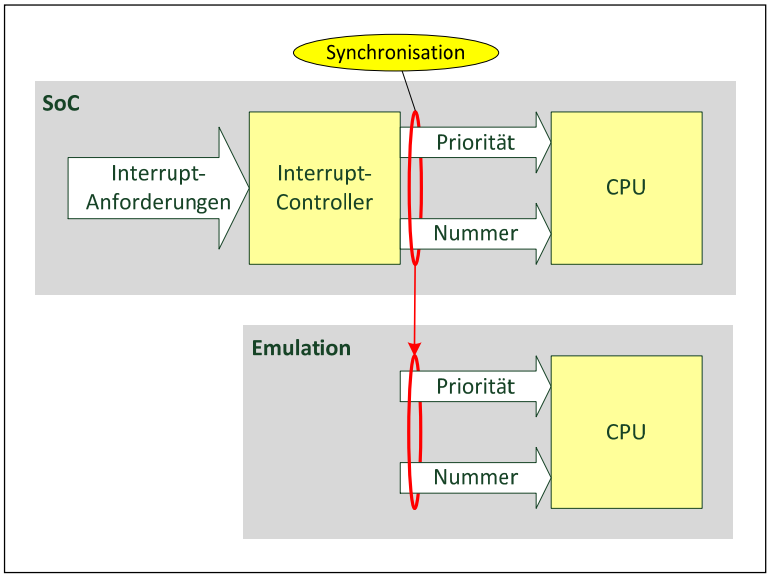


Abbildung 5-17: Interrupt-Synchronisation

In den meisten Anwendungsfällen ist es akzeptabel, dass zwischen dem Auftreten eines Ereignisses und der Reaktion der CPU einige CPU-Takte vergehen. Dann können die zu übermittelnden Synchronisationsinformationen - wie in Abbildung 5-18 dargestellt - serialisiert werden.

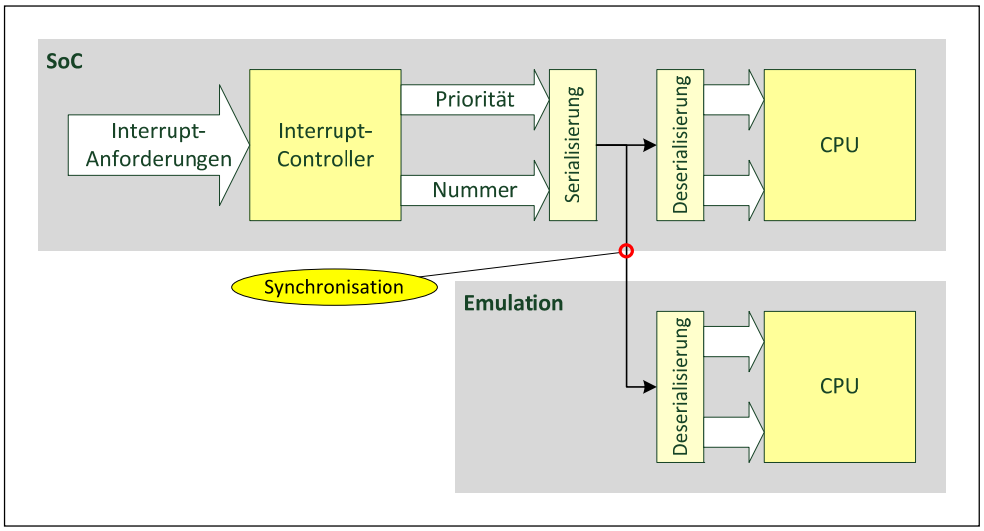


Abbildung 5-18: Interrupt-Synchronisation mit optimierter Bandbreite

Auch hier muss (wie bei der Synchronisation von busmasterfähigen Peripherieeinheiten diskutiert) im zu beobachtenden SoC die Deserialisierung verwendet werden, um die durch den Serialisierungs- und Deserialisierungsprozess verursachte Verzögerung der an der jeweiligen CPU anliegenden Interruptsignale in beiden Systemen konstant zu halten.

Bei manchen CPU-Architekturen ist der Interrupt-Controller sehr eng an die CPU gekoppelt und nicht von dieser trennbar. Beispielsweise werden CPUs von ARM bereits mit dem Interrupt-Controller als IP-Kern geliefert, der Wunsch nach einer (für eine effektive Synchronisation sinnvollen) Aufspaltung in CPU und separatem Interrupt-Controller könnte schwierig durchsetzbar sein. Für diesen Fall müssten alle Eingangssignale des Interrupt-Controllers (was üblicherweise bis zu 256 Signale sind) vom SoC an die Emulation übertragen werden. Um auch hier die erforderliche Bandbreite in handhabbaren Grenzen zu halten, bietet sich wieder eine serialisierte Übertragung an (Abbildung 5-19), was allerdings die Reaktionszeit auf eine Interrupt-Anforderung beeinflusst. Bei einer Übertragung von 256 Eingängen mit einer Taktrate von 100MHz und einer Breite von ½ Bit (eine Flanke bei DDR-Übertragung) ergäbe sich eine maximale Verzögerung der Reaktionszeit von ca. 2 µs, was für die meisten Systeme eine akzeptable Größenordnung darstellt. Weiterhin muss beachtet werden, dass die Reihenfolge des Eintreffens von Interrupt-Anforderungen gleicher Priorität nicht bestimmbar ist. Dies sollte aber in einem ordnungsgemäß aufgesetzten System auch keinen funktionellen Einfluss haben.

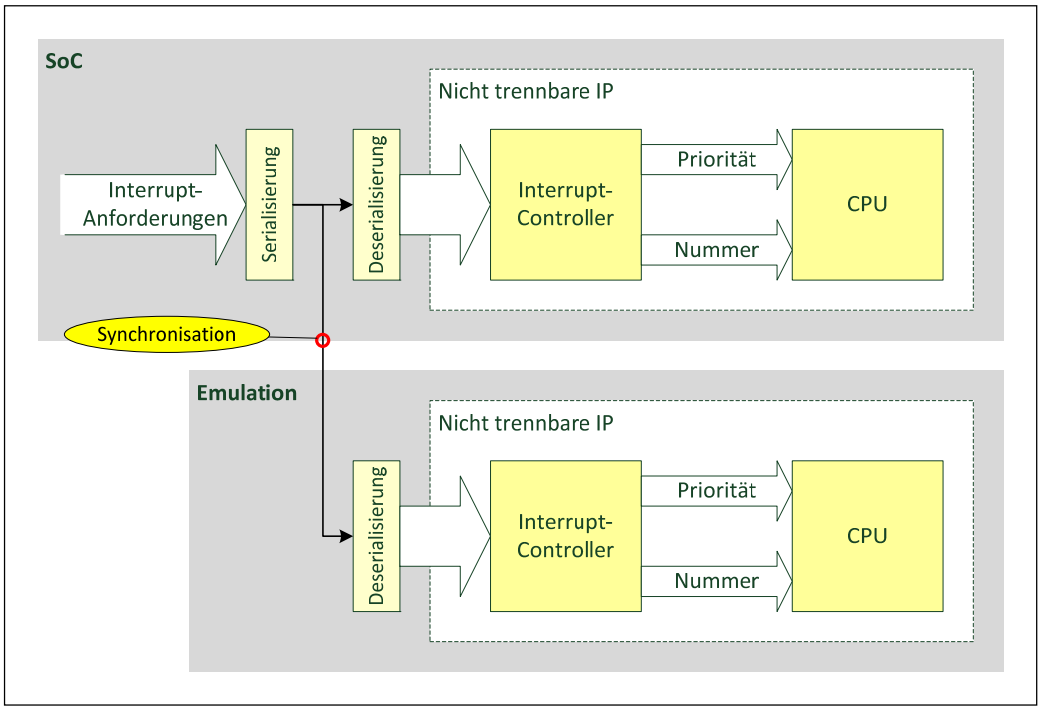


Abbildung 5-19: Interrupt-Synchronisation mit Serialisierung von Interrupt-Anforderungen

Eine weitere Möglichkeit zur Synchronisation von Interrupts ist die Nutzung freier Synchronisations-Bandbreite, die beim Kontextwechsel infolge eines erfolgreichen Interrupt-Aufrufs vorliegen kann. Hier macht man sich die Tatsache zu Nutze, dass in dieser Zeit keine Daten gelesen werden und somit die Bandbreite der Synchronisationsschnittstelle für andere Zwecke (d.h. Übertragung des neuen Interrupt-Vectors) verfügbar ist.

5.2.3 Weitere Möglichkeiten zur Reduktion der für die Synchronisation erforderlichen Bandbreite

Bei der Bestimmung der erforderlichen Bandbreite für die Synchronisation wird ein Worst-Case-Szenario zugrunde gelegt, d.h. die Bandbreite wird so ausgelegt, dass die Synchronisation in jeder ungünstigen Folge von Lesezugriffen des Busmasters / der Busmaster gewährleistet bleibt. Diese ungünstige Abfolge tritt aber nur in den seltensten Fällen ein, d.h. die meiste Zeit bleibt die vorgehaltene Bandbreite ungenutzt. Es gibt nun mehrere technische Möglichkeiten, die erforderliche maximale Bandbreite zu reduzieren und der durchschnittlich erforderlichen Bandbreite anzunähern.

FIFOs

Wenn die Synchronisationsdaten mittels FIFO gepuffert werden, lassen sich kurzzeitig erforderliche Spitzenwerte der benötigten Bandbreite abfedern. Es muss aber sichergestellt werden, dass die durch die Verwendung eines FIFOs erreichbare mittlere Bandbreite nicht überschritten wird.

Optimierung durch den Compiler

Durch Beeinflussung des Compilers kann die maximal benötigte Bandbreite ebenfalls reduziert werden. Hierzu muss der Compiler die Reihenfolge von Instruktionen so anordnen, dass eine bestimmte mittlere Synchronisations-Bandbreite (Lesezugriffe der CPU(s) auf Speicherbereiche, die nicht in der Emulation enthalten sind) nicht überschritten wird. Sollte dies nicht möglich sein, muss der Compiler automatisch *No Operations* (NOPs) einfügen.

Temporäres Anhalten des Systems

Alternativ kann eine CPU auch bei zu hoher erforderlicher Synchronisationsbandbreite temporär angehalten werden. Dies muss dann gleichzeitig sowohl im zu beobachtenden SoC als auch in der Emulation geschehen, um den synchronen Ablauf der Programmverarbeitung in beiden Systemen zu gewährleisten.

Eine optimale Implementierung der Synchronisation beider Systeme ist stark architekturabhängig, in diesem Abschnitt konnten deshalb nur allgemeine Vorgehensweisen dargestellt werden, welche bei konkreten Implementierungen dann individuell angepasst und optimiert werden müssen.

5.3 Integritätskontrolle

Da bei einer hidICE-Implementierung der zu untersuchende SoC und die Emulation parallel laufen, muss sichergestellt werden, dass – bis auf einen definierten zeitlichen Versatz – beide Systeme dasselbe tun. Dazu werden in beiden Systemen Hashwerte relevanter Signale wie z.B. Adress- und Datenbus, Kontrollsignale oder CPU-Register berechnet und miteinander verglichen [135]. Diese Berechnung kann rekursiv erfolgen, d.h. der vorhergehende Hashwert geht in die Berechnung des aktuellen Hashwertes mit ein.

Der SoC sendet gemeinsam mit seinen Synchronisationsdaten die lokal errechneten Hashwerte teilweise oder vollständig zur Emulation. In der Emulation wird die empfangene Prüfsumme mit den lokal berechneten Hashwerten verglichen. Sind beide identisch, so kann davon ausgegangen werden, dass beide Systeme synchron laufen und dass die in der Emulation verfügbar gemachten Trace-Daten die Abläufe im zu beobachtenden SoC exakt widerspiegeln.

Falls die Hashwerte unterschiedlich sind, kann dies durch einen Defekt in der Implementierung der Emulation oder durch eine Störung (siehe Abschnitt 2) verursacht worden sein. Die aus der Emulation gewonnenen Trace-Daten sind ab diesem Zeitpunkt ungültig. Um beide Systeme wieder in einen definierten Zustand zu versetzen, muss am SoC das Reset-Signal aktiviert werden, welches über die Synchronisationsschnittstelle auch ein entsprechendes Reset in der Emulation bewirkt.

In Abbildung 5-20 ist dargestellt, wie das aus Abbildung 5-11 bekannte MPSoC-System mit einer zusätzliche Integritätskontrolle ausgestattet wird.

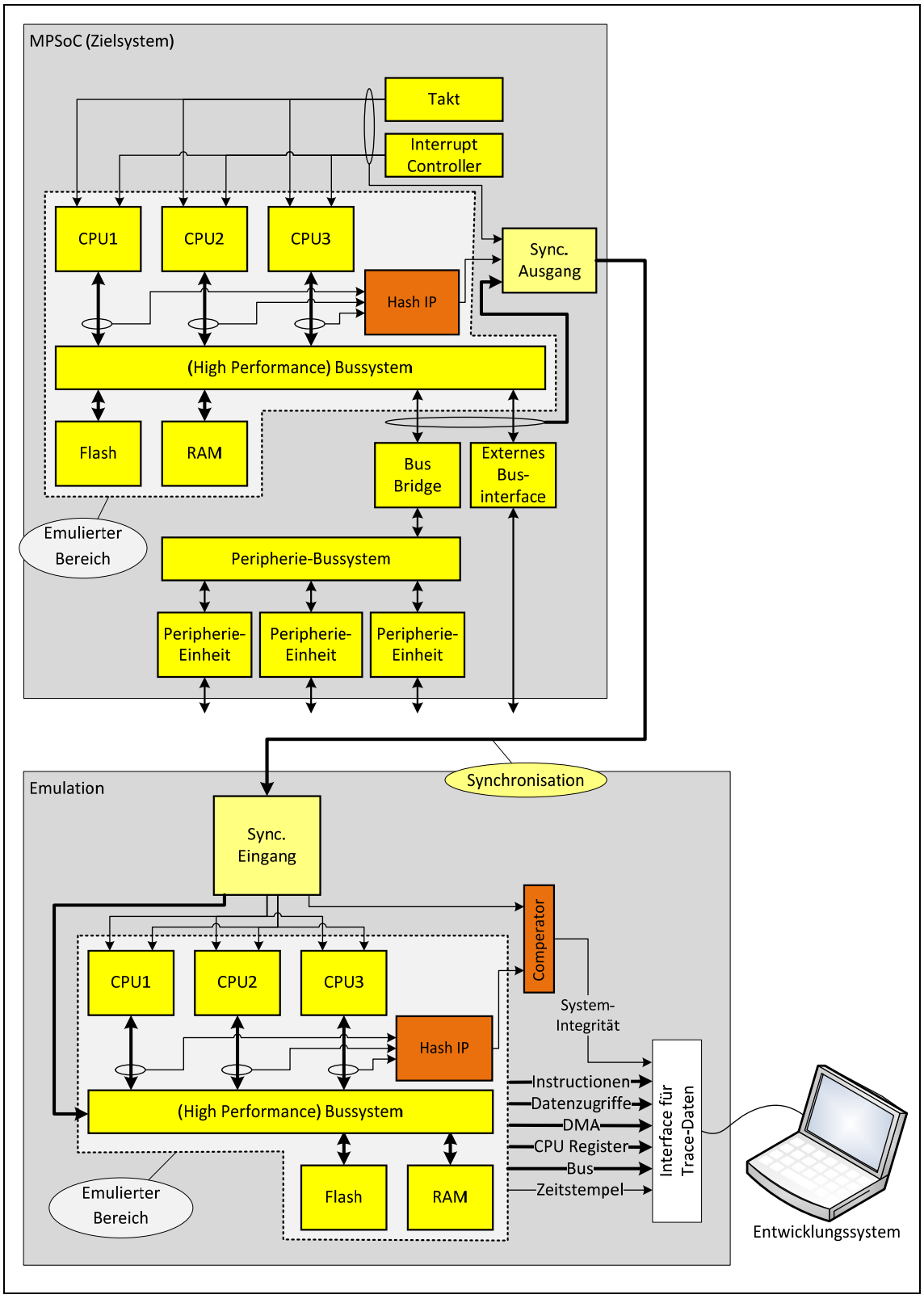


Abbildung 5-20: Systemintegritätskontrolle

Um im Fall einer Abweichung der Hashwerte eine Fehlerdiagnose zu erleichtern, können die Hashwerte aus verschiedenen domainspezifischen Teilsummen (z.B. Adressbus, Datenbus, CPU-Register) zusammengesetzt werden. Tritt ein Unterschied in einem Teil der Hashwerte auf, so liegt damit ein Hinweis auf den Ort der möglichen Ursache der Abweichung vor.

Zur Erzeugung der Hashwerte kann im einfachsten Fall eine exklusive VerODERung (XOR) der zu überwachenden Signale vorgenommen werden. Da hiermit nur Einzelbitfehler erkannt werden können, empfiehlt sich die Verwendung aufwändigerer Hash-Algorithmen (z.B. zyklische Redundanzprüfung – CRC). Ein Überblick über geeignet Hash-Funktionen ist beispielsweise in [136] und [137] zu finden.

5.4 Port-Ersetzung

Für die Synchronisation von einem SoC und der Emulation muss der SoC eine entsprechende Schnittstelle (üblicherweise I/O-Pins) bereitstellen. Grundsätzlich stellt dies eine unerwünschte Beeinträchtigung des Systems dar, da die für die Synchronisation verwendeten I/O-Pins für die eigentlichen Aufgaben des Systems fehlen. Es muss ein SoC mit entsprechend mehr I/O-Pins verwendet werden, was zusätzliche Kosten verursachen würde.

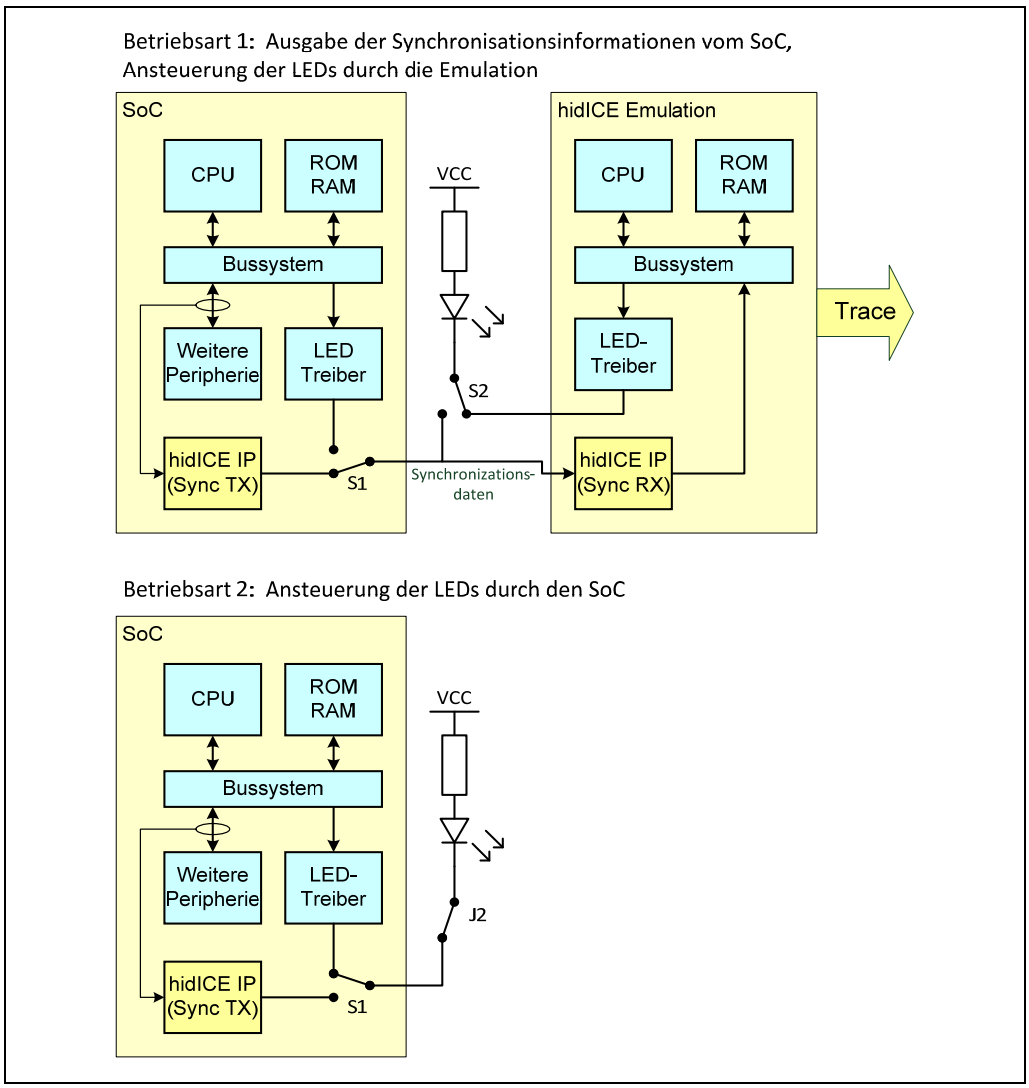


Abbildung 5-21: Prinzip und Betriebsarten der Port-Ersetzung

Eine elegante Lösung zur Vermeidung dieser Kosten stellt das Verfahren der Port-Ersetzung (*port replacement*) dar. Hier werden für die Ausgabe der Synchronisationsdaten I/O-Pins verwendet, die beim produktiven Einsatz des SoCs nur als Ausgänge betrieben werden. Beispiele solcher Ausgänge sind Displaytreiber, Timer-Ausgänge (z.B. PWM für Motorsteuerungen) oder einzelne Pins, die LEDs für diverse Statusanzeigen schalten.

Wenn die an diese Pins angeschlossenen Funktionseinheiten auch in der Emulation mit implementiert werden, kann der Emulator die Funktion der für die Synchronisation verwendeten Pins nachbilden und diese somit ersetzen. Aus Sicht der auf dem SoC ablaufenden Software sind die ersetzten Funktionen immer verfügbar, unabhängig davon ob eine Beobachtung stattfindet oder nicht.

In Abbildung 5-21 und Abbildung 5-22 ist das Verfahren der Port-Ersetzung dargestellt. Soll der SoC beobachtet werden, so werden an den I/O-Pins, die im produktiven Einsatz des SoCs als Treiber für LEDs dienen, via J1 die Synchronisationsinformation ausgegeben (J2 ist offen). In der Emulation ist der LED-Treiber des SoCs ebenfalls integriert. Führt nun die CPU des SoCs einen schreibenden Zugriff auf ihren LED-Treiber aus, so hat das an den I/O-Pins keinen Effekt, da diese für die Ausgabe der Synchronisationsinformation verwendet werden.

In der Emulation führt die CPU nun (mit einem durch die Übertragung der Synchronisationsdaten bedingten Zeitversatz) ebenfalls den Schreibzugriff auf den LED-Treiber aus. Die entsprechenden Signale werden via J3 zu den LEDs geleitet, diese verhalten sich wie vom Programm beabsichtigt. Trotz aktiver Synchronisation sind aus Sicht der Anwendung alle I/O-Pins verfügbar. Die einzigen Einschränkungen sind, dass dieses Prinzip nur mit Ausgangspins funktioniert und dass ein wohldefinierter minimaler Zeitversatz akzeptabel ist.

Im produktiven Einsatz sind die Ausgänge der LED-Treiber wieder auf die regulären I/O-Pins geroutet. Wenn J2 geschlossen wird (und die Anschlüsse J1 und J3 inaktiv sind), verhält sich das System wie im produktiven Einsatz beabsichtigt.

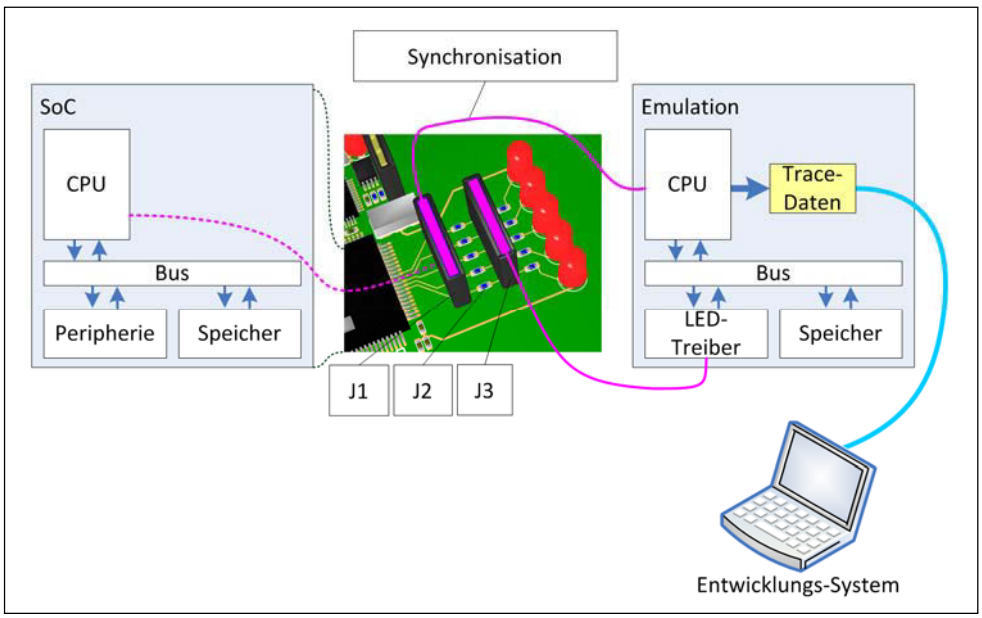


Abbildung 5-22: Implementierungsbeispiels für die Port-Ersetzung

Um die Port-Ersetzung nutzen zu können, muss dem System initial der Betriebsmodus mitgeteilt werden (z.B. durch Auslesen des Status von I/O-Pins während des Bootvorgangs), dies darf keinen Einfluss auf die Synchronität des SoCs und der Emulation haben.

5.5 Physikalische Schnittstellen

Die Übertragung der Synchronisationsinformation zwischen dem SoC und der Emulation sollte mechanisch und elektrisch sehr robust sein, um Störungen zu vermeiden. Da die I/O-Pins eines SoCs in den meisten Fällen nicht geeignet sind, größere kapazitive Lasten wie längere Leitungen zu treiben, empfiehlt sich hier ein externes Interface, welches die Signale auf geeignete Weise wandelt. Dies kann eine differentielle Übertragung sein, welche deutlich störfester als eine einfache Leitung ist. Besonders geeignet ist eine Umsetzung auf ein optisches Übertragungsmedium (Glasfaser), womit selbst größere Entfernungen weitgehend störungsfrei übertragen werden können. Zudem erfolgt eine Potentialtrennung zwischen SOC und Emulation, was in einigen Anwendungen ein sehr willkommener Effekt sein kann.

Da sich Störungen der Übertragung nicht vollständig vermeiden lassen, kann das Interface mit einer zusätzlichen Sicherungsschicht (OSI Schicht 2) [138] ausgestattet werden, welche die fehlerfreie Übertragung prüft und im Falle eines Fehlers die gestörte Information wieder korrigieren kann (Fehlererkennung und Fehlerbehebung).

5.6 Implementierung der Emulation

Die Emulation erfordert eine funktionell identische Implementierung des zu emulierenden Bereiches im SoC. Für langsame SoCs (bis ca. 100 MHz) kann die Emulation besonders vorteilhaft mit Hilfe eines FPGAs realisiert werden. Der große Vorteil dieser Lösung liegt einmal in seiner Flexibilität, d.h. ein FPGA-System kann mehrere unterschiedliche SoC-Architekturen unterstützen, zudem lassen sich eventuelle Erweiterungen bzw. Fehlerkorrekturen im Design des SoCs schnell für die Emulation mit übernehmen. Weiterhin können die Funktionen zur Auswertung der gewonnen Trace-Daten im gleichen FPGA mit implementiert werden, was zu einem sehr kompakten Emulationssystem führt.

Für SoCs mit höherer Taktfrequenz ist die Implementierung der Emulation in einem FPGA zu langsam. Hier muss ein spezieller Chip verwendet werden, der den zu emulierenden Bereich des SOCs fest implementiert hat. Dies ist teuer, bietet aber im Vergleich zu der – durchaus üblichen – Bereitstellung von Evaluationschips oder sogenannten „Umbrellachips“ (Spezialimplementierung eines SoCs mit maximalem Peripherieausbau und umfangreicher Trace-Unterstützung) den entscheidenden Vorteil, dass nur der zu emulierende Bereich implementiert werden muss. Die Peripherieausstattung sowie eventuell später hinzukommende Erweiterungen von Peripherieeinheiten sind automatisch mit unterstützt, solange sich am zu emulierenden Bereich nichts ändert.

Eine weitere interessante und sehr kostengünstige Lösung stellt die Implementierung eines „Emulationsmodus“ in einem SoC dar, bei dem die Strukturen zur Synchronisation und zur Ausgabe der Trace-Daten mit implementiert sind. Dazu muss ein Serien-SoC nur mit der Einheit zur Berechnung der Hash-Daten, der Sende– und Empfangseinheit für Synchronisationsdaten sowie zur breitbandigen Ausgabe der gewonnenen Trace-Daten ausgestattet werden. Das gleiche SoC-Design kann dann sowohl als Zielsystem als auch als Emulation fungieren. Mit dieser Lösung entfällt einmal der Aufwand für die Verfügbarmachung eines speziellen Emulationschips, zum anderen ist sichergestellt, dass die Implementierung der Emulation absolut identisch zum zu beobachtenden Serien-SoC (exakt gleiches Chipdesign, inklusive aller kleinen Maskenänderungen) ist. Weiterhin entfallen die nicht unerheblichen Kosten für einen speziellen Emulationschip.

Das schon weit verbreitete Multiplexen von I/O-Ports kann nun so erweitert werden, dass bestimmte Pins des SoCs zur Ausgabe und zum Empfang der Synchronisationsdaten konfiguriert werden. In der Emulation können dann alle anderen verfügbaren I/O Pins zur Ausgabe von Trace-Daten verwendet werden. Auch die Methode der Port-Ersetzung kann an dieser Stelle vorteilhaft eingesetzt werden. In manchen SoCs sind die Pins von LCD-Interfaces mit dem Trace-Port gemultiplexed (z.B. TI OMAP 4460). Wenn eine entsprechende Weiterentwicklung eines solchen SoCs mit den vorgeschlagenen Strukturen ergänzt werden würde, könnte während der Erfassung der Trace-Daten vom zu untersuchenden SoC die Emulation die fehlenden Signale für die LCD-Ausgabe liefern.

Neben einer viel umfassenderen Beobachtbarkeit des SoCs, welcher durch die nun verfügbare Bandbreite des Trace-Ports ermöglicht wird, besteht nun auch nicht mehr die Einschränkung, dass der LCD-Controller während der Erfassung der Trace-Daten für die Applikation nicht mehr verfügbar ist.

Ein Beispiel einer solchen Implementierung ist in Abbildung 5-23 dargestellt.

5.7 Anbindung an bestehende Entwicklungswerkzeuge

Für die Akzeptanz einer neuen Lösung ist es oft hilfreich, wenn diese gemeinsam mit bestehenden Strukturen genutzt werden kann. Da nun schon hohe Investitionen in traditionelle Emulationstechniken geflossen sind, ist es durchaus sinnvoll, die bestehenden Strukturen zu nutzen, um die via hidICE gewonnenen Trace-Daten weiterverarbeiten zu können. So könnte die hidICE-Emulation mit gängigen Trace-Puffer-Strukturen (z.B. ARM CoreSight ETM/TPIU) ausgestattet werden, um mit bereits verfügbaren Emulatoren (z.B. Lauterbach Trace32, GHS Supertrace Probe) auf die gewonnenen Trace-Daten zugreifen zu können (Abbildung 5-23).

Bei der Implementierung ist eine möglichst breite Trace-Schnittstelle (z.B. 64, 128 oder 256 Bit) anzustreben, welche damit über die aktuell maximal definierte Breite von 32 Bit hinausgeht [101]. Alternativ könnten auch entsprechend leistungsfähige Serializer verwendet werden.

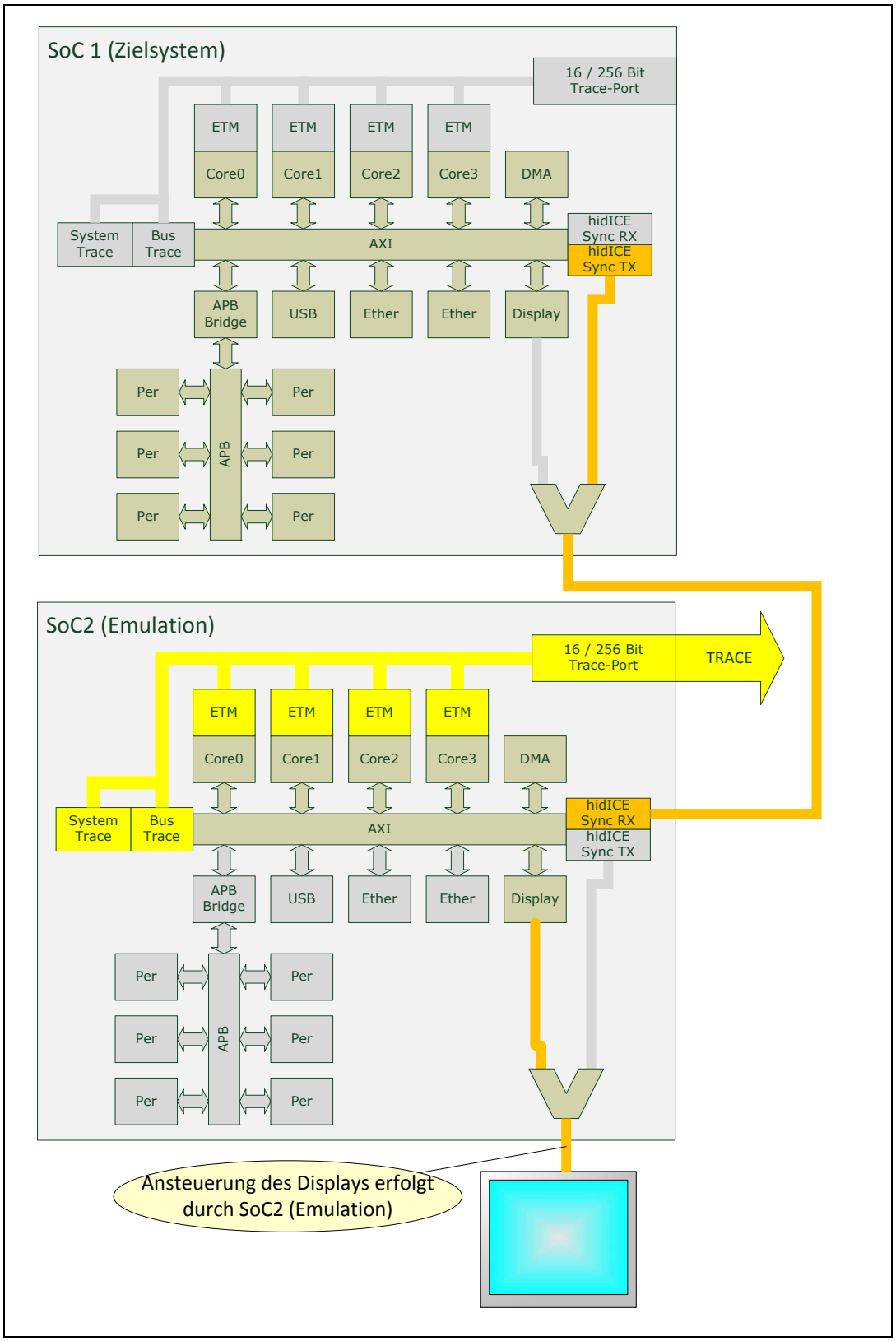


Abbildung 5-23: Beispiel für die Ausgabe von Trace-Daten aus der Emulation mittels ARM CoreSight-Elementen

5.8 Weitere Anwendungen

Neben der Beobachtung von SOCs sind auch noch weitere Anwendungen einer hidICE-basierten Emulation denkbar.

5.8.1 FPGA

Moderne FPGAs verfügen oftmals über eine Komplexität, die es gestattet, eine oder mehrere CPUs (Soft-Cores) mit zu implementieren. Damit kann eine einfache Anpassbarkeit eines FPGA-basierten Systems erreicht werden, ohne dass bei einer Programmänderung ein zeitaufwändiger Syntheseprozess gestartet werden muss. Aufgrund des relativ hohen Preises für FPGA-Ressourcen wird oftmals auf die Verwendung von „embedded Trace"-Strukturen für die Soft-Cores verzichtet, was eine schlechte Beobachtbarkeit mit sich bringt. Eine temporäre Implementierung dieser „embedded Trace"-Strukturen bringt ähnliche Probleme mit sich, wie sie im Rahmen der temporären Software-Instrumentierung diskutiert wurden. Auch hier gilt, dass die Beobachtbarkeit möglichst auch im finalen Design einer FPGA- Anwendung möglich sein sollte.

An dieser Stelle eignet sich eine hidICE-Implementierung ganz besonders [139]. Dazu wird, wie bereits diskutiert, die Emulation in einem zweiten FPGA implementiert. Dieser muss dabei nicht vom gleichen Typ wie der FPGA des zu beobachtenden Systems sein, sondern er kann auch deutlich größer sein und vorteilhafterweise gleich die weitere Verarbeitung der gewonnenen Beobachtungsergebnisse mit übernehmen.

Die in Abschnitt 5.10 beschriebenen experimentellen Implementierungen hidICE-basierter Emulationen wurden auf FPGAs implementiert. Hier ist leicht zu sehen, dass auf dem zu untersuchenden Zielsystem nur ein minimaler Aufwand erforderlich ist, um eine umfassende Beobachtbarkeit zu gewährleisten.

5.8.2 Redundanz

Um bestimmte Störungen (z.B. verursacht durch *Single Events*) erkennen und darauf reagieren zu können, werden besonders bei sicherheitskritischen Anwendungen redundante CPUs eingesetzt, die mit einem bestimmten zeitlichen Versatz identische Operationen ausführen und die Ergebnisse entsprechend vergleichen. Dieser Ansatz wird als *lock-step* bezeichnet. Ein Beispiel hierfür ist die Hercules TMS570LS-Serie von Texas Instruments, die auf dem ARM Cortex-R4F basiert. Beide CPUs sind hier gespiegelt und um 90° gedreht implementiert [140], um die Widerstandsfähigkeit gegen durch Strahlung oder Rauschen verursachte Störungen zu erhöhen.

Ein ähnlicher Effekt kann auch mittels einer hidICE-Implementierung erreicht werden. Durch die implementierte Integritätskontrolle kann zuverlässig erkannt werden, ob die Abläufe im zu untersuchendem Zielsystem und in der Emulation identisch sind. Tritt hier eine Abweichung auf, so wird diese erkannt und das System kann entsprechend darauf reagieren. Gegenüber der redundanten Implementierung der CPU allein bietet eine entsprechende hidICE-Implementierung darüber hinaus den Vorteil, dass entsprechend dem emulierten Bereich auch weitere Systembestandteile (z.B. Bussystem, Speicher) mit in die Überprüfung einbezogen werden können. Die oftmals gestellte Forderung, dass ein sicherheitskritisches System auf zwei Komponenten basieren sollte, die auf unterschiedlichen Chips implementiert sind, ist bei einer Realisierung mittels einer hidICE-synchronisierten Emulation ebenfalls erfüllt.

5.8.3 Silizium-Test

Das Prinzip der Integritätskontrolle kann auch zum Test von SoCs verwendet werden (Abbildung 5-24). Wird ein Unterschied der beiden Prüfsummen beobachtet, so kann mit Hilfe des aufgezeichneten Trace-S untersucht werden, welche Aktivitäten das Fehlverhalten verursacht haben [141]. Diese Methode ist besonders auch zur Untersuchung sehr selten vorkommender Fehler geeignet. Die Testabdeckung erstreckt sich hier aber nur auf den Bereich des SoCs, der auch in der Emulation nachgebildet wird.

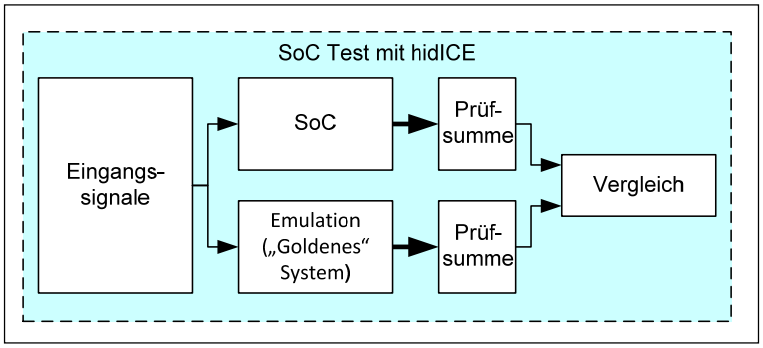


Abbildung 5-24: Fehlersuche bei SoCs

5.9 Limitationen

Es gibt eine ganze Reihe von Punkten, die die Implementierung des vorgestellten Emulationsprinzips erschweren oder gar unmöglich machen.

Synchronität

Zunächst einmal muss sichergestellt werden, dass der zu emulierende Bereich als synchrones Design mit Bezug auf einen festen Bezugstakt realisierbar ist. Es ist möglich, die Taktfrequenz einzelner Bereiche zu verändern, dies darf aber niemals asynchron (z.B. durch PLLs) geschehen, sondern es muss der Bezugstakt entsprechend geteilt bzw. „gegated" werden (Abbildung 5-15).

Busmaster

Busmasterfähige Peripherieeinheiten, deren Verhalten abhängig von den zur Laufzeit des Programms empfangenen Daten (z.B. busmasterfähige CAN-, Ethernet- oder USB-Einheiten) ist, müssen konstruktiv so geändert werden, dass eine Synchronisation der empfangenen Daten möglich ist (Abbildung 5-13 und Abbildung 5-14). Alternativ könnten die jeweiligen Einheiten als Bus-Slaves implementiert werden und der von den CPUs unabhängige Daten-Transfer von eigenständigen DMA-Controllern organisiert werden. In diesem Fall würden die DMA-Controller mit in den zu emulierenden Bereich aufgenommen werden und die von den Bus-Slaves gelesenen Daten wären mit den vorgestellten Methoden leicht zu synchronisieren.

Grundsätzlich wird sich die Frage stellen, ob die notwendigen Designänderungen für bestehende SoC-Familien durchsetzbar sind. Wenn die Forderung nach der Möglichkeit der Synchronisierbarkeit in die Planung neuer IPs und SoC-Familien einfließt, ist sicher die Chance einer Realisierung größer.

I/O-Bandbreite

Die Implementierung der Synchronisierbarkeit für SoC-Familien, deren Applikations-Schwerpunkt im Bereich der Kommunikation liegt und die mehrere schnelle Kommunikationsschnittstellen (Ethernet, USB etc.) beinhalten, ist aufgrund der für die Synchronisation erforderlichen hohen Bandbreite schwierig. Andererseits muss man sich vor Augen halten, dass es für diese SoC-Familien auch keine andere Möglichkeit geben wird, einen vollständigen Daten-Trace zu gewinnen.

Delay

Bedingt durch die Verzögerung, die die Synchronisation mit sich bringt (gesicherte Übertragung der Synchronisationsdaten, physikalische Laufzeit auf der Übertragungsstrecke), läuft die Emulation mit einem wohldefinierten Versatz im Vergleich zu dem zu beobachtenden SoC. Besteht nun die Forderung, anhand der in der Emulation beobachteten Systemzustände zu reagieren (z.B. die Implementierung zusätzlicher Breakpoints), so muss die Verzögerung zwischen Auftreten des Systemzustands im SoC und der Beobachtung in der Emulation akzeptiert werden. Dies hätte beispielsweise im Fall der Breakpoints zur Folge, dass der Haltebefehl an den SoC erst mit einigen Takten Verzögerung wirksam werden kann.

Speziell bei Breakpoints würde sich an dieser Stelle empfehlen, einige einfache Breakpoints im SoC zu implementieren und diese mit beliebig vielen und beliebig komplexen Breakpoints aus der Emulation zu kombinieren.

Nicht initialisierte Speicher

Voraussetzung für die Synchronisierbarkeit eines SoCs und dessen Emulation sind jeweils identische Systemzustände nach Freigabe des Reset-Signals. Erfolgt hier ein Lesezugriff auf einen nicht initialisierten Speicherbereich, ist das Ergebnis dieser Leseoperation unbestimmt, d.h. es würde eine Verletzung der Systemintegrität erkannt werden. Da derartige Zugriffe eigentlich nicht zulässig sind (eine Ausnahme besteht vielleicht in der Generierung von Zufallszahlen, die man aber auf anderem Weg zuverlässiger gewinnen kann), sollte dieses Verhalten keine Einschränkung bei der Anwendung eines hidICE-basierten Emulationssystems darstellen. Vielmehr könnte es an dieser Stelle eine Möglichkeit zur Erkennung derartiger Defekte bieten.

5.10 Experimentelle Implementierungen

Es wurden verschiedene experimentelle Implementierungen einer hidICE-basierten Emulation vorgenommen. Mit diesen Implementierungen konnte das Funktionieren des neuen Emulationsprinzips demonstriert werden.

Folgende Aspekte der Emulation wurden ebenfalls erfolgreich verifiziert:

- Reaktion auf äußere Ereignisse (Interrupts)
- Sich ändernde Taktfrequenzen
- Verletzung der Systemintegrität
- Betrieb mit mehreren Busmastern

5.10.1 Xilinx PicoBlaze

Die erste experimentelle Implementierung einer hidICE-basierten Emulation erfolgte auf der Basis eines PicoBlaze-Softcores (KCPSM3) [142] von Xilinx. Das System wurde auf zwei Xilinx Virtex-4 FX12 Evaluation Kits (Avnet) aufgebaut, die darin verwendeten FPGAs waren jeweils ein Xilinx Virtex-4 XC4FX12 (Abbildung 5-25).

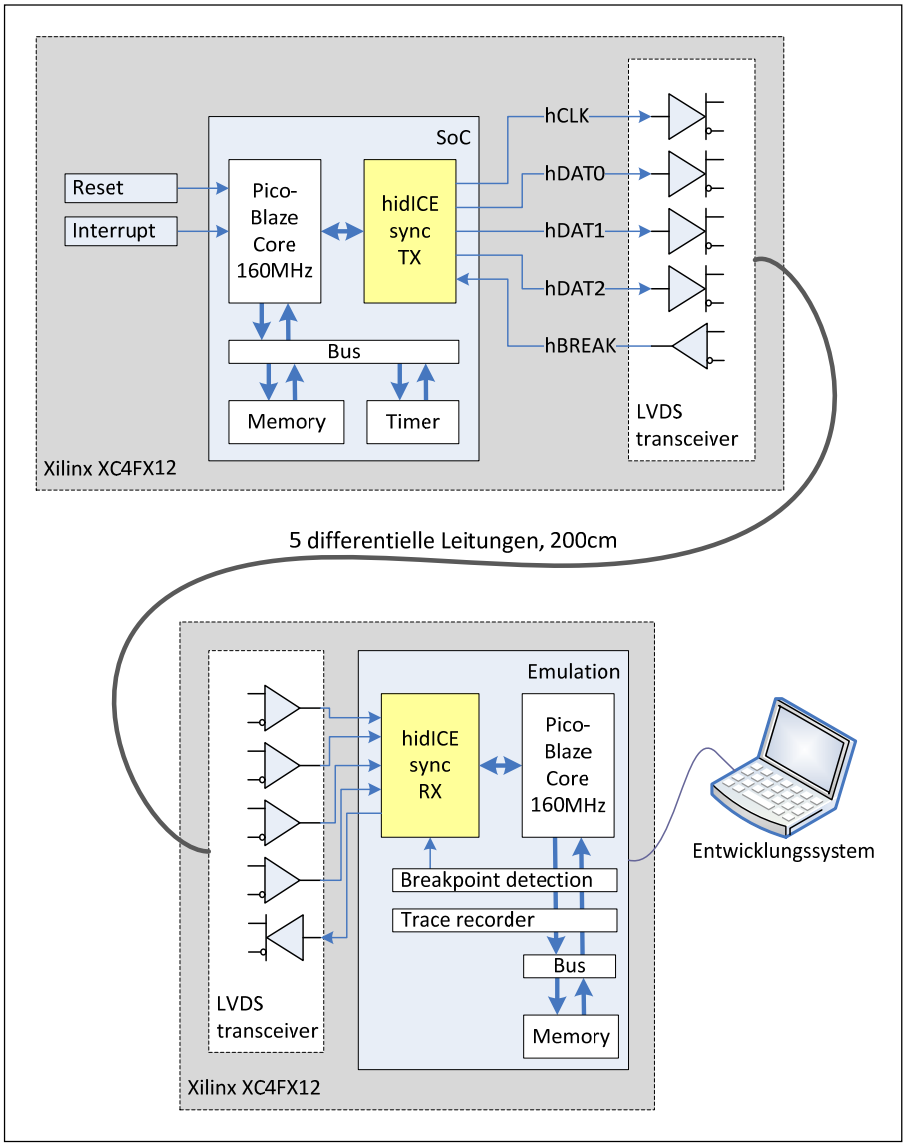


Abbildung 5-25: PicoBlaze Implementierung einer hidICE-Emulation

Der PicoBlaze ist eine einfache, speziell für FPGA-Implementierungen optimierte 8-Bit CPU, welche zur Ausführung einer Instruktion zwei Taktzyklen benötigt. In der vorliegenden Implementierung erreicht die CPU bei einer Taktung mit 160MHz eine Verarbeitungsleistung von 80 MIPS.

Die SoC-Implementierung und die Emulation sind mit 5 differentiell übertragenen Signalleitungen über eine Länge von 2m miteinander verbunden. Eine Leitung wird zur Übertragung des Taktes verwendet, drei weitere Leitungen zur Übertragung der Synchronisationsinformationen. Die Übertragung erfolgt mit jeder steigenden und fallenden Flanke des Taktes (DDR), so dass bei einer Instruktionsdauer von 2 Takten je Instruktion 12 Bits übertragen werden können. Diese setzen sich aus den 8 Bit breiten von der CPU gelesenen Daten (IN_PORT[7:0]) sowie dem INTERRUPT-Signal (periodisch vom Timer generiert) und dem RESET- Signal zusammen. Der Programmspeicher ist sowohl im SoC als auch in der Emulation enthalten.

Initial werden beide FPGAs via JTAG konfiguriert, danach wird sowohl in den SoC als auch in die Emulation der gleiche Programmcode geladen. Nach Freigabe des Reset-Signals fangen beide CPUs an, synchron zu arbeiten. Mittels einer seriellen Schnittstelle kann in der Emulation der Trace-Speicher ausgelesen und an den PC übermittelt werden. Weiterhin kann die Adresse für einen Breakpoint gesetzt sowie das System gestoppt werden.

Wenn die Emulation eine Übereinstimmung der aktuell ausgeführten Programmadresse mit der zuvor eingestellten Breakpoint-Adresse feststellt, wird mittels eines im Emulator generierten Break-Signals die CPU des SoCs und in Folge dann auch die Emulation angehalten. Dies geschieht mit einer Verzögerung, bedingt durch die Latenz der Übertragung der Signale vom SoC zur Emulation und zurück.

Der für die Ausgabe der Synchronisationsinformationen erforderliche Aufwand im SoC ist sehr gering, es werden dafür 17 Flip-Flops und 22 LUTs benötigt.

Anhand des PicoBlaze-Demonstrationssystems können folgende Merkmale demonstriert werden:

- Reset / Interrupts lassen sich synchronisieren
- Trace-Daten können in der Emulation aufgezeichnet werden
- die Online-Überwachung von Variablen ist möglich

5.10.2 Cortex M1

Eine weitere experimentelle Implementierung eines hidICE-basierten Emulationssystems wurde für ein ARM Cortex M1 basiertes System vorgenommen.

Dazu wurden zwei Evaluation-Boards von Actel (jetzt MicroSemi) mit jeweils einem ProASIC3 A3P1000 FPGA verwendet.

Unter Verwendung der Actel CoreConsole IP Deployment Plattform [143] wurde eine SoC-Implementierung (Abbildung 5-26) entwickelt, welche aus folgenden Komponenten besteht:

- Cortex M1 CPU
- AHBlite Bus
- Memory Controller (CoreMemCtrl)
- AHB-to-APB Bridge (CoreAHB2APB)
- APB-Bus
- General-Purpose IO (CoreGPIO)
- UART (CoreUARTapb)
- Timer (CoreTimer)

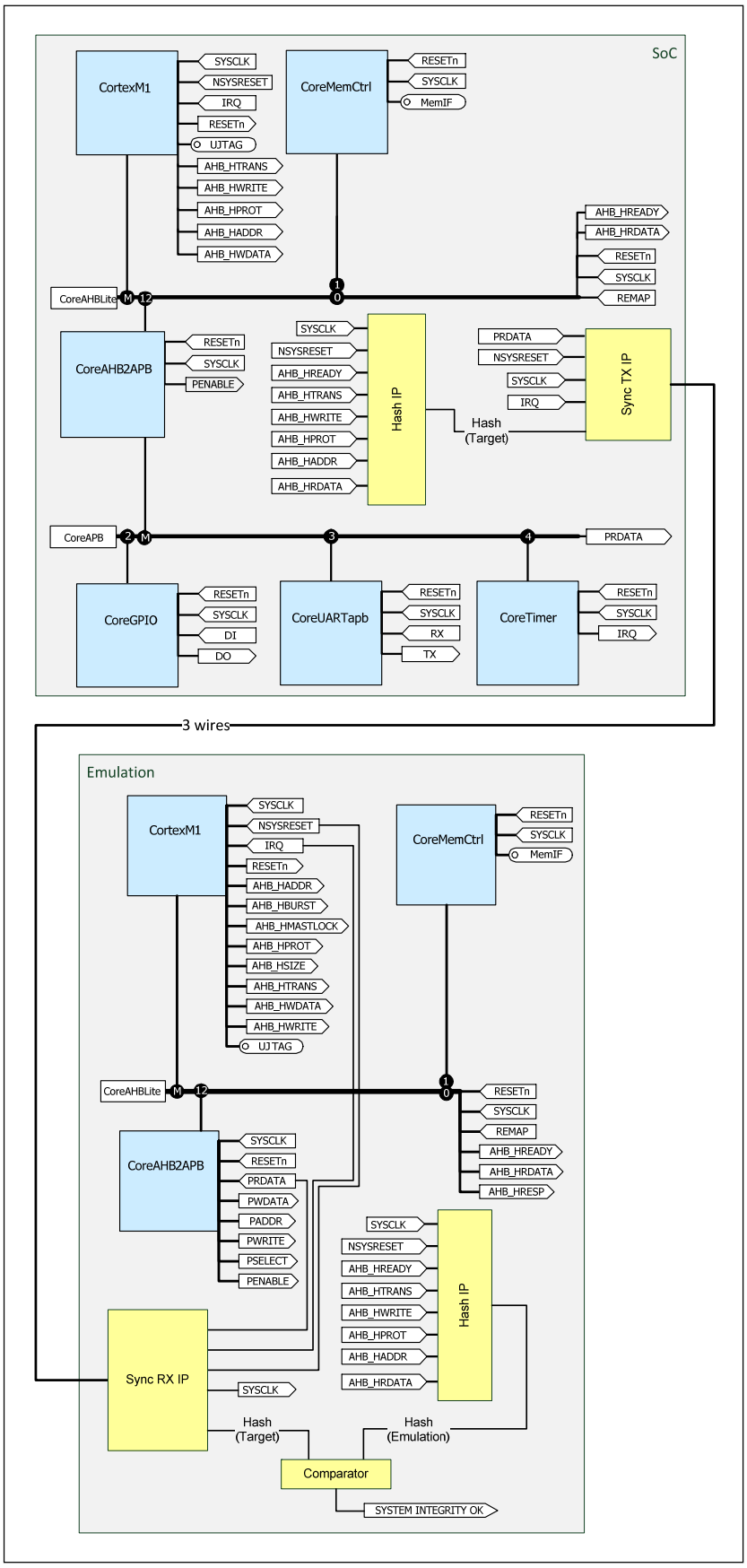


Abbildung 5-26: hidICE Implementierung für ARM Cortex M1

Zusätzlich wurden noch folgende Module entwickelt:

Hash IP

Hier wird eine Prüfsumme über relevante Systeminformationen (in diesem Fall gültige Adressen und Daten auf dem AHBlite-Bus) berechnet. Dazu werden am AHBLite-Bus die gültigen Buszyklen anhand der Bus-Signale AHB_HTRANS und AHB_HREADY detektiert und im Falle eines gültigen Buszyklus jeweils eine neue 4 Bit breite Prüfsumme aus der aktuellen Adresse (AHB_HADDR), dem gelesenen Wert (AHB_HRDATA) sowie der vorherigen Prüfsumme berechnet. Die zur Berechnung des Hashs auf dem FPGA benötigten Ressourcen sind in Tabelle 5-4 dargestellt.

SyncTX IP

Diese Einheit dient zur Übermittlung der Synchronisationsinformationen. Dazu werden folgende Signale erfasst:

- PRDATA (16 Bit, gelesene Daten auf dem APB-Bus)
- NSYSRESET (globales Reset-Signal)
- SYSCLK (globales Clock-Signal
- IRQ (Interrupt-Signal, welches vom CoreTimer erzeugt wird)

Die Synchronisationsdaten werden über 3 Leitungen übertragen, wobei eine Leitung zur Übertragung des Taktes (SYSCLK * 6) dient und die anderen beiden Leitungen zur Übertragung der Nutzdaten.

Die zur Übermittlung der Synchronisationsdaten benötigten Ressourcen im SoC sind ebenfalls in Tabelle 5-4 dargestellt. Die Umrechnung der verwendeten FPGA-Ressourcen in Gates erfolgte durch Projektion der in den FPGA-Core synthetisierten Primitive (z.B. AND3, NOR2 etc.) auf eine CMOS Gate Array Bibliothek [144].

IP	FPGA core cells	Äquivalenter Gate-Count
Hash	85	270
Sync TX	182	912
Summe	267	1182

Tabelle 5-4: Für die hidICE-Synchronisation der Cortex M1-Implementierung benötigte Ressourcen auf dem SoC

In der Emulation befinden sich folgende Einheiten:

- Cortex M1 CPU
- AHBlite Bus
- Memory Controller (CoreMemCtrl)
- AHB-to-APB Bridge (CoreAHB2APB)

Neben der zum SoC identischen Hash IP befinden sich in der Emulation noch zwei weitere Einheiten:

SyncRX IP

Diese Einheit empfängt die Synchronisationsdaten und stellt sie den einzelnen Modulen zur Verfügung.

Comparator

Der Comparator vergleicht die Ergebnisse der Hash-IP der Emulation mit denen der Hash-IP des SoC. Sind diese identisch, kann davon ausgegangen werden, dass die in der Emulation beobachtbaren Zustände den Zuständen im SoC entsprechen.

Initial muss bei beiden Systemen der Programmspeicher mit dem gleichen Inhalt gefüllt sein. Nachdem an den SoC das NSYSRESET-Signal angelegt wurde, erreicht dieses den Emulator mit einer wohldefinierten Verzögerung von 3 Taktzyklen. SoC und Emulation beginnen mit der Abarbeitung des Programms. Erfolgt ein Lesezugriff auf den APB-Bus, so werden die im SoC gelesenen Daten an die Emulation übermittelt. Wenn die CPU der Emulation die entsprechenden Daten liest, stehen auf dem Bus die gleichen Informationen bereit wie 3 Taktzyklen zuvor im SoC. Das gleiche Verhalten tritt bei einem Interrupt auf, auch hier wird der Interrupt-Request mit einem exakten Taktversatz zum SoC an die CPU der Emulation angelegt.

Um den Mechanismus der Systemintegritätskontrolle (Hash IP und Comparator) zu überprüfen, wurde eine Ungleichheit des Programmcodes zwischen SoC und Emulation eingebaut. Die dazu notwendige Verzweigung wurde durch Betätigung eines Tasters erreicht und es konnte demonstriert werden, dass eine Abweichung des Programmablaufs in beiden Systemen erkannt werden kann.

Als Ergebnis lässt sich ein äußerst detaillierter Bus-Trace rekonstruieren (Abbildung 5-27). In dem Ausschnitt ist die Abarbeitung mehrerer Instruktionen inklusive lesendem und schreibendem Datenzugriff dargestellt. Die Signale des beobachteten CoreAHBLite und des CoreAPB (Abbildung 5-26) sind in Tabelle 5-5 erläutert, wobei die mit (*) gekennzeichneten Signale in Abbildung 5-27 nicht mit dargestellt sind. Als Parameter wird „S“ (Anzahl der Slave-Interfaces, hier: 2) sowie „M“ (Anzahl der Master-Interfaces, hier: 1) verwendet.

Die Abläufe in dem Bussystem sind vollständig beobachtbar, obwohl nur die Signale PRDATA, NSYSREST, SYSCLK und IRQ als Synchronisationsinformationen vom SoC übermittelt wurden.

Bus	Signalname	Breite (Bit)	Bedeutung
AHBlite	SYSCLK	1	Bus-Clock, entspricht HCLK (*)
	RESETn	1	Bus-Reset, entspricht HRESETn (*)
	HADDR	32	Adress-Bus
	HTRANS	2	Transfer-Typ 00: IDLE (Transfer wird ignoriert) 01: BUSY (Unterbrechung des laufenden Transfers) 10: NONSEQ (erster Transfer eines Bursts / einzelner Transfer) 11: SEQ (folgende Transfers eines Bursts)
	HWRITE	1	Unterscheidung Schreib / Lese-Transfer 0: Lesen 1: Schreiben
	HSIZE	3	Transferbreite (*) 000: 8 Bit 001: 16 Bit 010: 32 Bit 011: 64 Bit 1xx: weitere Breiten
	HBURST	3	Burst-Typ (*) 000: SINGLE (einzelner Transfer) 001: INCR (inkrementeller Burst unbestimmter Länge) 010: WRAP4 (4-beat wrapping Burst) 011: INCR4 (4-beat inkrementeller Burst) 100: WRAP8 (8-beat wrapping Burst) 101: INCR8 (8-beat inkrementeller Burst) 110: WRAP16 (16-beat wrapping Burst) 111: INCR16 (16-beat inkrementeller Burst)
	HPROT0	1	Schutzsignale 0: OPCODE (Opcode fetch) 1: DATA (Datenzugriff)
	HWDATA	32	Daten vom Master an den Slave
	HSELx	S*1	Slave-Auswahl (*)
	HRDATA	32	Daten vom Slave an den Master
	HREADY	S*1	Verzögerung des Transfers durch den Slave
	HRESP	S*2	Transfer-Antwort (*) 00: OKAY (bei HREADY=1: Transfer okay) 01: ERROR (Transfer fehlerhaft) 10: RETRY (Transfer noch nicht beendet) 11: SPLIT (Anforderung eines gesplitteten Transfers)
	HMAST-LOCK	1	Anzeige, dass aktuell ein geschützter Transfer stattfindet (*)
APB	PCLK	1	Bus-Clock, entspricht HCLK (*)
	PRESETn	1	Bus-Reset, entspricht HRESETn (*)
	PADDR	32	Adress-Bus
	PSELECT	P*1	Auswahl einer Peripherieeinheit
	PENABLE	1	Anzeige eines gültigen Zugriffs
	PWRITE	1	Unterscheidung Schreib- / Lese-Transfer
	PWDATA	32	Daten vom Master an die Peripherieeinheit
	PRDATA	32	Daten von der Peripherieeinheit an den Master

Tabelle 5-5: In der ARM Cortex M1-mplemntierung verwendete Signale des AHBlite / APB Busses

AHBlite Trace							APB Trace					
HADDR	HWDATA	HRDATA	HPROTO	HWRITE	HREADY	HTRANS	PADDR	PWDATA	PRDATA	PSELECT	PWRITE	PENABLE
Temp += *(unsigned int*)(gCoreTimer.baseAddress + 0x04);												
0x00000c8c <OnCoreTimer_irq+16>: ldr r3, [pc, #76] (0xcdc <OnCoreTimer_irq+96>)												
0x00000c8e <OnCoreTimer_irq+18>: ldr r3, [r3, #0]												
00000C8C	C3000000	00000000	OPCODE	RD	False	NONSEQ	0004	20202020	00000000	0010	RD	False
00000C8C	**C3000000**	**009C1FDF**	**OPCODE**	**RD**	**True**	**NONSEQ**	**0004**	**20202020**	**009C1FDF**	**0010**	**RD**	**True**
00000C8C	C3000000	00000000	OPCODE	RD	False	IDLE	0004	20202020	00000000	0000	RD	False
00000C8C	C3000000	681B4B13	OPCODE	RD	True	IDLE	0004	20202020	00000000	0000	RD	False
00000CE0	**C3000000**	**681B4B13**	**DATA**	**RD**	**True**	**NONSEQ**	0004	20202020	00000000	0000	RD	False
00000CE0	C3000000	00000000	DATA	RD	False	IDLE	0004	20202020	00000000	0000	RD	False
00000C8C	C3000000	00001430	OPCODE	RD	True	IDLE	0004	20202020	00000000	0000	RD	False
00000C8C	C3000000	681B4B13	OPCODE	RD	True	IDLE	0004	20202020	00000000	0000	RD	False
00000C8C	C3000000	00000000	OPCODE	RD	True	IDLE	0004	20202020	00000000	0000	RD	False
0x00000c90 <OnCoreTimer_irq+20>: adds r3, #4												
0x00000c92 <OnCoreTimer_irq+22>: ldr r2, [r3, #0]												
00000C90	**C3000000**	**00000000**	**OPCODE**	**RD**	**True**	**NONSEQ**	0004	20202020	00000000	0000	RD	False
00001430	009C1FDF	00000000	DATA	WR	False	NONSEQ	0004	20202020	00000000	0000	RD	False
00001430	**009C1FDF**	**681A3304**	**DATA**	**WR**	**True**	**NONSEQ**	0004	20202020	00000000	0000	RD	False
00001430	009C1FDF	00000000	DATA	WR	False	IDLE	0004	20202020	00000000	0000	RD	False
00001430	009C1FDF	00000000	DATA	WR	False	IDLE	0004	20202020	00000000	0000	RD	False

Abbildung 5-27: Ausschnitt aus einem mit der hidICE Implementierung für ARM Cortex M1 aufgezeichneten Bus-Trace

5.10.3 MPSoC SPARC V8 Demonstrationssystem

Um auch die Beobachtbarkeit eines MPSoCs mittels hidICE zu demonstrieren, wurde ein System auf der Basis der LEON3-CPUs entwickelt [130]. Die verwendete 32 Bit SPARC V8 CPU ist gemeinsam mit einer umfassenden Infrastruktur als VHDL-Sourcecode verfügbar [133].

Die Implementierung des Systems erfolgte auf zwei Xilinx ML507-Boards, welche neben umfangreicher Peripherie mit jeweils einem Virtex-5 FPGA XC5VFX70T-1FFG1136 ausgestattet sind.

Einen Überblick über das implementierte System gibt Abbildung 5-12, die wesentliche Struktur ist in Abbildung 5-11 angegeben.

Die verwendete SPARC V8 CPU hat folgende Eigenschaften:

- 7-stufige Pipeline
- Hardware-Multiplizierer und –Dividierer, MAC-Einheit
- Floating-Point-Einheit
- Konfigurierbarer Instruktions- und Daten-Cache
- *Scratchpad*-RAM für Instruktionen und Daten
- AMBA-2.0 AHB bus interface

Der implementierte SoC (Abbildung 5-28) besteht aus folgenden Einheiten:

- zwei SPARC V8 LEON3 CPUs (`cpu0, cpu1`)
- eine SPARC V8 LEON3 CPU (`cpu2`), welche als DMA-Controller verwendet wird
- On-Chip-Debug-Unterstützung (via busmasterfähigem UART und einer Debug-Support-Einheit) (`ahbuart0, dsu0`)
- vier AHB Busmaster (`cpu0, cpu1, cpu2, ahbuart0`)
- Clock-Controller, welcher während einer laufenden Anwendung den CPU-Takt ändern kann (`dcm0`)
- Multiprocessor Interrupt Controller (`irqmp0`)
- AHB und APB Bussystem, verbunden via einer AHB/APB Bus Bridge (`ahbctrl0, apbctrl0`)
- APB Peripherie-Einheiten: Timer, UART und I/O-Ports (`gpio0, apbuart0, timer0`)
- hidICE Sync IP RX zur Übermittlung der Synchronisationsdaten (`qdr_out_inst`)
- hidICE Hash IPs zur Berechnung von rekursiven Prüfsummen für die System-Integritätskontrolle (`hash_data_inst, hash_addr_inst`)

Zur Synchronisation der Emulation (Abbildung 5-29) ist die Übertragung der in Tabelle 5-6 aufgeführten Signale erforderlich. Für die Implementierung einer Debug-Unterstützung müssen die in Tabelle 5-7 aufgelisteten Signalen zusätzlich übertragen werden. Die Richtung der Signale ist in beiden Tabellen wie folgt angegeben:

- S -> E: SoC an Emulation
- E -> S: Emulation -> SoC

Um die Anzahl der für die Synchronisation notwendigen I/O-pins und Signalleitungen zu reduzieren, werden pro Taktzyklus und Leitung 4 Bit Synchronisationsdaten übertragen (QDR).

In der vorliegenden Implementierung wurden bisher keine Optimierungen der Synchronisationssignale, wie sie in Abschnitt 5.2 beschrieben wurden, vorgenommen. Es wird erwartet, dass sich durch geeignete Optimierungen die für die Synchronisation erforderliche Anzahl von I/O-Pins von 60 Leitungen auf 10 bis 15 senken lässt. Zudem kann – aus Sicht der auf dem SoC laufenden Software – die Anzahl der erforderliche I/O-Pins auf Null reduziert werden, wenn die Technik der Port-Ersetzung Anwendung findet.

Signal	Richtung	Breite	Beschreibung
CLKM	S -> E	1	System-Takt
RESET	S -> E	1	System-Reset
HRDATA[31:0]	S -> E	32	Auf dem AHB Bus gelesene Daten
HREADY	S -> E	1	AHB Bus HREADY-Signal
HRESP[1:0]	S -> E	2	AHB Bus HRESP-Signal
IRQI_IRL	S -> E	12	Interrupt-Level der 3 CPUs (4 Signale pro CPU)
IRQI_RST	S -> E	3	Power-Down Modus der 3 CPUs (1 Signal pro CPU)
HASH_ADR	S -> E	1	Prüfsumme für Adressen
HASH_DATA	S -> E	2	Prüfsumme für Daten
Summe		55	

Tabelle 5-6: SPARC V8 MPSoC: für die hidICE-Synchronisation erforderliche Signale

Signal	Richtung	Breite	Beschreibung
DSU_BREAK_OUT	S -> E	1	Haltesignal für die DSU der Emulation
AHBUART_RXD_OUT	S -> E	1	Eingangssignal für den AHB-UART der Emulation
AHBUART_TXD	S -> E	1	Ausgangssignal der AHB-UARTs
DSU_BREAK_IN	E -> S	1	Externes Haltesignal für die DSU des SoCs (für den Fall einer Integritätsverletzung)
AHBUART_RXD_IN	E -> S	1	Eingangssignal des AHB-UARTs des SoCs
Summe		5	

Tabelle 5-7: SPARC V8 MPSoC: für die Debug-Unterstützung erforderliche Signale

Mit dem MPSoC SPARC V8 Demonstrationssystem lässt sich die folgende Funktionalität nachweisen:

- Erfolgreiche Synchronisation eines Multi-Core Systems mit mehreren Busmastern
- Gewinnung eines detaillierten und gleichzeitigen Trace-S von mehreren CPUs, welcher folgende Informationen beinhaltet:
  - Ausgeführte Instruktionen (zyklusgenau/Zeitstempel)
  - Gelesene und geschriebene Daten inkl. Adresse
  - CPU-Register
  - Zustand des Caches
- Gewinnung eines detaillierten Bus-Trace-S
  - Zeitstempel
  - Gelesene und geschriebene Daten inkl. Adresse
  - Kontroll-Signale des Busses
  - Beobachtung der Bus-Arbitrierung

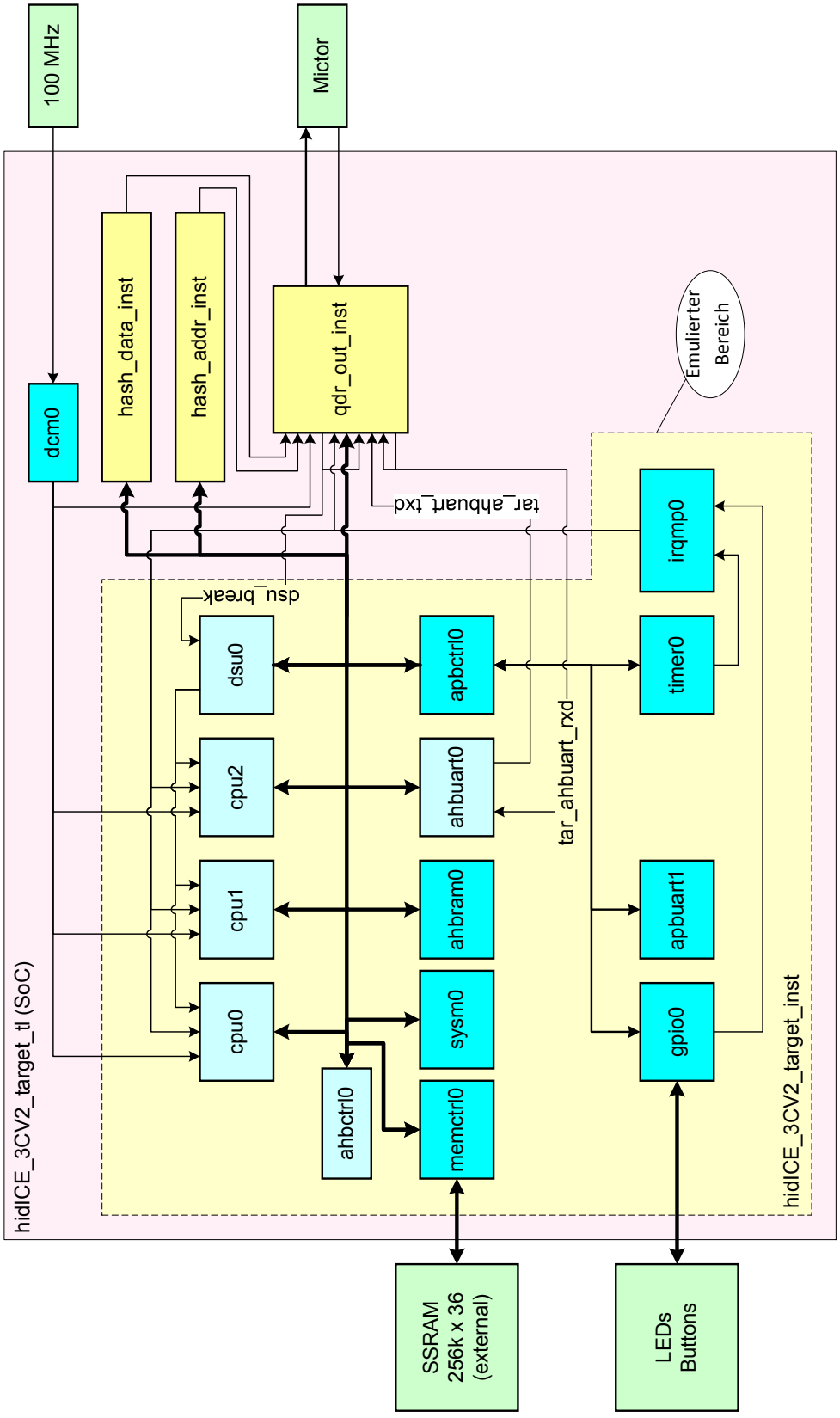


Abbildung 5-28: Implementierung des LEON3 MPSoCs mit hidICE-Synchronisation

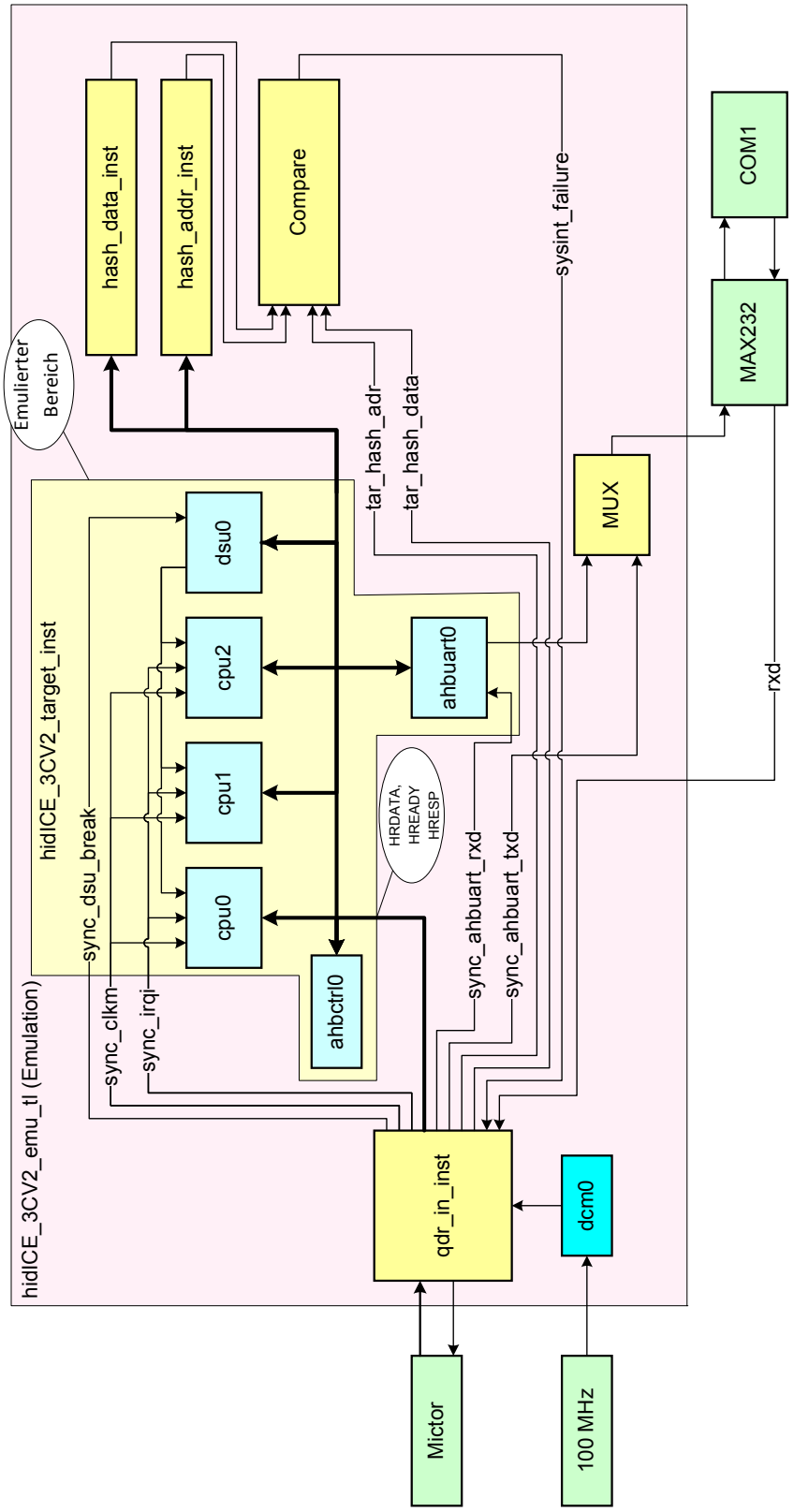


Abbildung 5-29: Implementierung der Emulation des LEON3 MPSoCs

5.11 Bewertung des neuen Ansatzes

Im Folgenden wird der hidICE-Ansatz anhand der in Abschnitt 3 diskutierten Kriterien mit dem aktuellen Stand der Technik verglichen. Diese Bewertung ist exemplarisch und muss im Einzelfall an die Möglichkeiten der tatsächlichen hidICE-Implementierung und die Beobachtungsziele angepasst werden.

5.11.1 Vollständigkeit der Beobachtung

Der große Vorteil des hidICE-Ansatzes besteht darin, dass in der Emulation jeder einzelne Zustand beobachtbar ist. Dies bedeutet, dass neben den von der CPU ausgeführten Instruktionen (und damit auch Funktionen, Basic Blocks und Sprunginformationen), den gelesenen und geschriebenen Daten auch weitere Informationen verfügbar sind, die bei einer „embedded Trace"-Lösung aufgrund der Bandbreitenbeschränkung zur Ausgabe von Trace-Daten nicht ausgegeben werden können. Dazu zählt vor allem die Möglichkeit zur Beobachtung von CPU-Registern, aber auch ein detaillierter Blick in den Cache und in das Bussystem. Die Beobachtbarkeit kann an dieser Stelle die Detailauflösung eines Logikanalysators erreichen (Abbildung 5-27).

Je nachdem, ob die Ereignisverarbeitungseinheit mit in die Emulation aufgenommen wurde, können auch Interrupt-Anforderungen von Peripherieeinheiten mit beobachtet werden.

Die synchronisierte Beobachtung von Umgebungsbedingungen wie Temperatur, Stromaufnahme, Versorgungsspannung, analoger Eingangssignale sowie digitale Eingangssignale inklusive der Interpretation des Protokolls ist mittels synchronisierten externen Messungen möglich. Durch die wohldefinierte Verzögerung des Ablaufs in der Emulation im Vergleich zum Ablauf innerhalb des zu beobachtenden SoCs lässt sich eine taktsynchrone Zuordnung zwischen den beobachteten Umgebungsbedingungen und den Abläufen innerhalb des SoCs herstellen.

5.11.2 Beobachtbarkeit von MPSoCs

hidICE eignet sich sehr gut zur Beobachtung von MPSoCs, da alle CPUs mit in die Emulation aufgenommen werden können. Somit ist eine gleichzeitige und synchronisierte Beobachtung aller CPUs möglich.

5.11.3 Echtzeitfähigkeit und Kontinuität

Wenn die Emulation hinreichend schnell ist, kann der zu untersuchende SOC mit seiner regulären Geschwindigkeit laufen. Die Dauer der Beobachtung ist beliebig, allerdings muss die Beobachtung mit einem definierten Systemzustand (Reset) beginnen. Ein „Einklinken" der Beobachtung in ein bereits laufendes Programm ist nicht möglich.

5.11.4 Gleichzeitige Durchführung vieler Beobachtungsaufgaben

Durch die Zugriffsmöglichkeit auf alle relevanten Systemzustände können, eine entsprechend leistungsfähige Weiterverarbeitungseinheit vorausgesetzt, beliebig viele Beobachtungsaufgaben gleichzeitig durchgeführt werden.

5.11.5 Beeinflussung des SoCs

Durch die rein passive Beobachtung kann keine Beeinflussung des Programmlaufs erfolgen.

5.11.6 Beobachtbarkeit von SoCs aus der Serienproduktion

Wenn die zur Ausgabe der Synchronisations-Informationen erforderlichen Strukturen auf in der Serienproduktion verwendeten SoCs implementiert sind, so sind diese auch beobachtbar.

5.11.7 Beobachtbarkeit in „realer“ Umgebung

Wenn die Synchronisationsschnittstelle zugänglich ist, so kann der SoC auch in seiner realen Betriebsumgebung beobachtet werden. Wird hier die Technik der Port-Ersetzung verwendet, so sind aus der Sicht des Anwendungsprogramms alle Pins des SoCs auch während einer aktiven Beobachtung verfügbar.

5.11.8 Flexibilität des Beobachtungsfokus

Durch die Möglichkeit, grundsätzlich alle internen Zustände des SoC beobachten zu können, ist ein dynamischer Wechsel des Beobachtungsfokus nicht erforderlich. Für den Fall, dass die Bandbreite zur Ausgabe der internen Zustände aus der Emulation zu gering ist, kann diese dynamisch konfiguriert und gefiltert werden, ohne dabei den Programmlauf im zu beobachtenden SoC zu beeinflussen.

5.11.9 Latenz

Die Latenz zwischen einem Vorgang im SoC und dessen Beobachtbarkeit in der Emulation ist wohldefiniert und beträgt wenige Taktzyklen. Die Latenz ist abhängig von der Implementierung der Synchronisations-Schnittstelle und eventuell zwischengeschalteter FIFOs. Damit können beispielsweise zusätzliche Hardware- Breakpoints aus der Emulation heraus bereitgestellt werden, welche dann nach einer wohldefinierten Verzögerung den Programmlauf im SoC unterbrechen können.

5.11.10 Qualifizierung des Beobachters

Durch die kontinuierliche Kontrolle der Systemintegrität mittels Berechnung und fortwährendem Vergleich von Hash-Werten sowohl vom zu beobachtenden System als auch von der Emulation kann sichergestellt werden, dass die Emulation ein exaktes Abbild der Vorgänge im Zielsystem liefert. Da die Ausgabe der Synchronisationsinformationen über eine unidirektionale Schnittstelle keinen Einfluss auf den Programmablauf des Zielsystems hat, ergibt sich eine „ideale“ Beobachtunglösung. Weiterhin kann auch die Auswahl der für die weitere Verarbeitung auszugebenen Trace-Daten situationsspezifisch geändert werden, ohne dass es hier zu einer Beeinflussung des Zielsystems kommen kann.

Damit ergeben sich bestmögliche Klassifizierungen für die in Abschnitt 3 diskutierten Normen. Diese Betrachtung bezieht sich hier nur auf die reine Beobachtung, für die Weiterverarbeitung der Trace-Daten (die nicht Bestandteil dieser Arbeit ist) muss eine eigenständige Betrachtung erfolgen.

Die folgenden Klassifikationen sind als Orientierung zu verstehen und müssen jeweils abhängig vom Kontext des Werkzeugeinsatzes individuell bestimmt werden.

Die Klassifizierung der Beobachtung mittels hidICE nach **IEC 61508** [86] kann je nach Zeitpunkt der Beobachtung sowohl als „*offline support tool*“ als auch als „*online support tool*“ erfolgen. Bei letzterem würden die Unterklassifizierungen T1 (kein direkter oder indirekter Beitrag zum Programmcode) sowie T2 (Werkzeuge zur Verifikation oder dem Test des Designs, eine Fehlfunktion des Werkzeugs kann zu einer Maskierung von Fehlverhalten des Programms führen) gelten.

Durch die passive, nichtintrusive Beobachtung ist eine Fehlfunktion des Beobachtungswerkzeugs, die zum Versagen einer Sicherheitsanforderung oder einer Risiko-Kontrollmaßnahme führt, nicht möglich. Weiterhin ist es durch den kontinuierlich mitgerechneten Hash-Wert ausgeschlossen, dass ein mögliches Fehlverhalten der Anwendung falsch dargestellt wird. Somit ergibt sich gemäß **ISO 26262** [35] aus einem *Tool Impact Level* mit „TI0“ und einem *Tool Error Detection Level* von „TD1“ ein *Tool Confidence Level von* „TCL1“, der für alle Sicherheitsklassen eine spezielle Qualifizierung des Entwicklungswerkzeugs nicht erforderlich macht.

Sollte die Methode der Port-Ersetzung verwendet werden und die emulierte Output-Funktion eine Sicherheitsanforderung oder Risiko-Kontrollmaßnahme erfüllen, so ist auch die Emulation mit in die Risikobetrachtung einzubeziehen.

5.11.11 Kosten

Die zur Implementierung der Synchronisationsschnittstelle und der Integritätskontrolle erforderliche Chipfläche ist abhängig von der Komplexität des SoCs. Für einfache SoCs ist sie vernachlässigbar klein. Verfügt der SoC über mehrere asynchrone Clock-Domains sowie über busmasterfähige Peripherieeinheiten, so entsteht ein zusätzlicher struktureller Aufwand zur notwendigen Synchronisation derselben. Wird die Methode der Port-Ersetzung verwendet, fallen die Kosten für die I/O-Pins weg, welche zur Ausgabe der Synchronisations-Informationen benötigt werden. Auch während einer laufenden Beobachtung sind hier aus Sicht des zu beobachtenden SoC alle I/O-Pins verfügbar.

Einen weiteren Kostenpunkt stellt die Verfügbarmachung der Emulation dar. Dies kann kostengünstig für langsame SoCs mittels einer FPGA-Implementierung erfolgen, für schnellere SoCs ist hierfür aber ein Standardzellen-Design oder ein eigenes Chipdesign notwendig. Die damit verbundenen Kosten lassen sich sparen, wenn in dem SoC-Design bereits eine Betriebsart vorgesehen ist, die es ihm gestattet, als Emulation zu arbeiten und die internen Zustände über die verfügbaren I/O-Pins auszugeben.

Weiterhin muss der Implementierungsaufwand für die Ausgabe der Synchronisationsinformationen sowie der Emulation berücksichtigt werden. Dazu kommen noch die Kosten des Werkzeugs, welches die von der Emulation ausgegebenen Systemzustände weiterverarbeitet.

5.12 Zusammenfassung

Mit dem hidICE-Ansatz wurde ein neuartiges Beobachtungsverfahren für SoCs entwickelt, welches auf einer synchronisiert parallel laufenden Emulation ausgewählter Bereiche des SoCs beruht.

Die Emulation muss mindestens einen CPU-Kern umfassen, in den meisten Fällen ist es aber sinnvoll, darüber hinaus noch das Bussystem, weitere Busmaster (DMA, weitere CPUs), sowie Speicher in die Emulation mit einzubeziehen. Zur Synchronisation müssen alle Informationen, die dem in die Emulation einbezogenen Subsystem erst zur Laufzeit bekannt werden, übertragen werden.

Im Einzelnen sind dies:

- Taktsignale
- Ergebnisse von Lese-Operationen
- Ereignisse (Reset, Interrupts, DMA-Anforderungen, Wartezyklen)

Nach dem aktiven Reset-Signal befindet sich sowohl der zu beobachtende SoC als auch dessen Emulation im gleichen Zustand. Da der Ablauf des Programms nunmehr nur vom Programmcode und den übertragenen Ereignissen abhängt, arbeiten beide Systeme fortan synchron. Die aus der Emulation nach außen übertragbaren Informationen sind identisch zu den Informationen, die man - bei Verfügbarkeit geeigneter Schnittstellen – aus dem zu beobachtenden SoC direkt gewinnen könnte.

Die zur Synchronisation der parallel laufenden Emulation erforderliche Bandbreite ist in vielen Fällen geringer als die Bandbreite, die zur Ausgabe äquivalente Trace-Informationen aus einer „embedded Trace“-Lösung notwendig wäre (Abbildung 5-30). Zudem können mittels Port-Ersetzung auch während einer aktiven Beobachtung die zur Ausgabe der Synchronisations-Informationen benötigten Pins für die Applikation nutzbar bleiben.

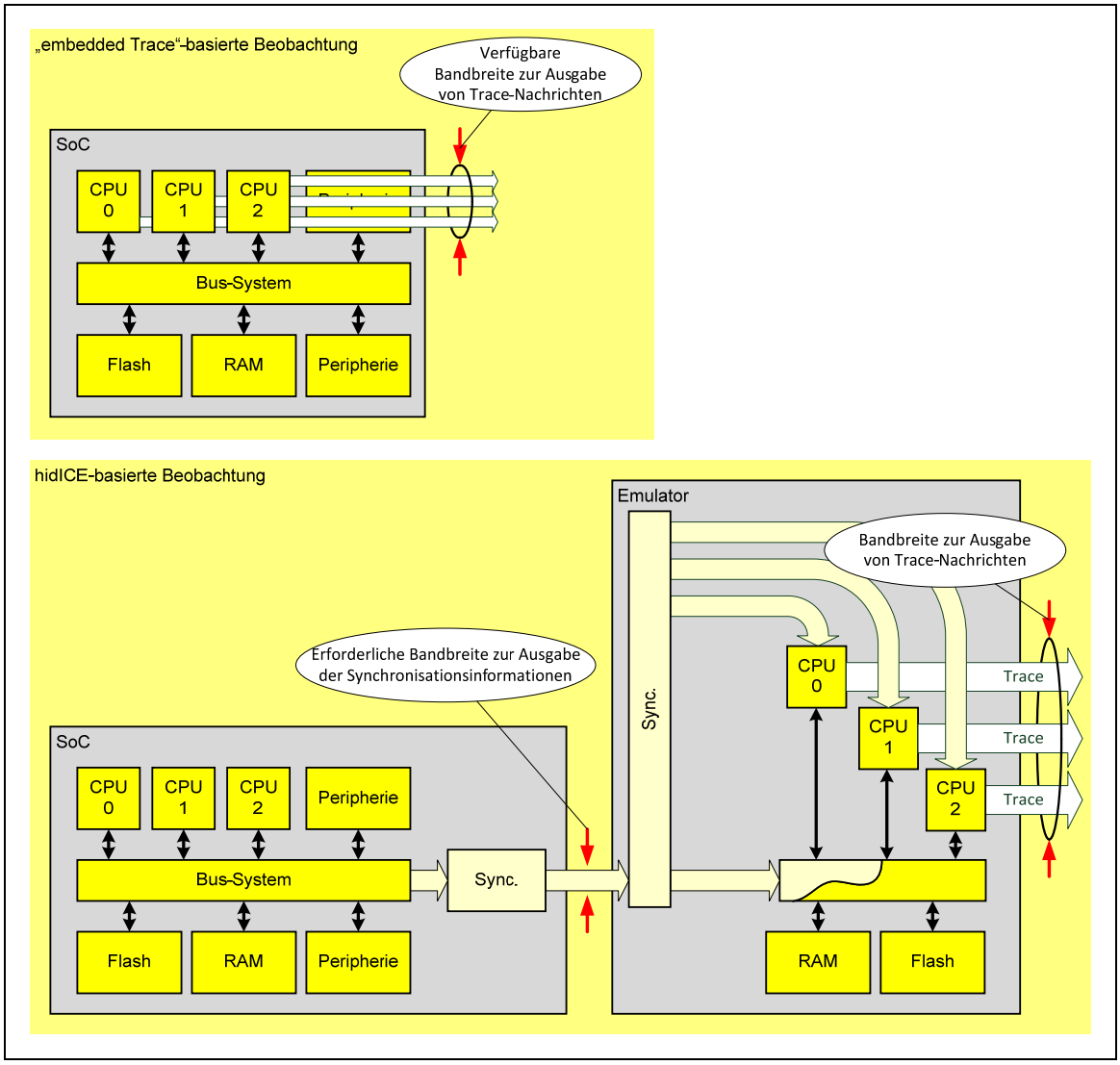


Abbildung 5-30: Vergleich der Ausnutzung der für die Bobachtung von SoCs verfügbaren Bandbreite

Die Beobachtbarkeit geht dabei über die Möglichkeiten des gegenwärtigen Standes der Technik deutlich hinaus, es werden die meisten der in Abschnitt 3 definierten Kriterien zur Qualität der Beobachtbarkeit in idealer Weise erfüllt.

Für einfache SoCs ist der Aufwand für die Ausgabe der Synchronisationsinformationen sehr gering, bei komplexeren SoCs besteht die Schwierigkeit in der Synchronisation unterschiedlicher Clock-Domains sowie in der Einbeziehung busmasterfähiger Peripherieeinheiten.

6. Zusammenfassung und Ausblick

In diesem Abschnitt wird der vorgestellte hidICE-Ansatz mit dem Stand der Technik zur Beobachtung von SoCs verglichen und die jeweilige Erfüllung der in den Abschnitten 2 und 3 erarbeiteten Anforderungen bewertet.

Nach einer anschließenden allgemeinen Zusammenfassung der Ergebnisse dieser Arbeit wird ein Ausblick auf weitere Themengebiete gegeben, die zukünftig bearbeitet werden müssen, um die Beobachtbarkeit von SoCs und damit die Qualität und Leistungsfähigkeit eingebetteter Systeme zu verbessern.

6.1 Vergleich von hidICE mit dem Stand der Technik

In Tabelle 6-1 werden die diskutierten Methoden zur Beobachtung von SoCs miteinander verglichen. Dabei handelt es sich um die Software-Instrumentierung (mit der Unterscheidung, ob der instrumentierte Code in der Release-Version der Software verbleibt), um „embedded Trace“ (unter Betrachtung der Offline- und Online-Verarbeitung der erfassten Trace-Daten) sowie dem in Abschnitt 5 propagierten hidICE-Ansatz.

Die folgenden Bewertungen sind subjektiv geprägt und müssen für den individuellen Anwendungsfall jeweils kritisch geprüft werden. Die Technik der Online-Verarbeitung von Trace-Daten ist aktuell nur in Ansätzen verfügbar, mit der Aufnahme dieses Ansatzes in die folgende vergleichende Betrachtung soll gezeigt werden, dass auch traditionelle Ansätze noch deutliches Verbesserungspotenzial bieten.

Bei einer hidICE-Emulation kann die **Vollständigkeit der Beobachtung** interner Vorgänge innerhalb des emulierten Bereichs des zu beobachtenden SoCs erreicht werden. Die alternativen Beobachtungsverfahren haben hier bereichsweise deutliche Einschränkungen. Diese treten besonders bei der Instrumentierung zu Tage, da die bei der Beobachtung anfallenden Datenmengen von den CPUs selbst verarbeitet werden müssen. Weiterhin ist die Sicht der CPUs nur auf sich selbst beschränkt, Vorgänge auf Bussen oder Peripherieeinheiten können allenfalls indirekt beobachtet werden.

Auch bei der **Beobachtung von MPSoCs** bieten sich bei der Verfügbarkeit einer hidICE-Emulation klare Vorteile, da die Bandbreite bei der Ausgabe der Trace-Daten aus mehreren CPUs nur in geringem Maß von der Anzahl der CPUs abhängig ist. Im Gegensatz dazu steigt bei „embedded Trace“-Lösungen die erforderliche Bandbreite proportional zur Zahl der CPUs an, zusätzlich müssen auch noch Timestamps übertragen werden.

Bei der Beurteilung der **Echtzeitfähigkeit und Kontinuität** (fortwährende Beobachtung eines SoCs mit unverminderter Taktfrequenz) ist die permanente Instrumentierung der Software sehr gut geeignet. Da während einer solchen Beobachtung zeitliche Einflussfaktoren eine Rolle spielen, ist die Anwendung der temporären Instrumentierung an dieser Stelle nicht geeignet, da das beobachtete System nicht mit der Release-Version identisch ist. „Embedded Trace“-Lösungen erlauben unter Berücksichtigung der verfügbaren Bandbreite zur Ausgabe der Trace-Daten eine Beobachtung mit unverminderter Taktfrequenz. Bei der Zwischenspeicherung und anschließenden Offline-Verarbeitung der Trace-Daten ist eine Kontinuität nicht gegeben, da die Beobachtungsdauer von der Größe des Zwischenspeichers abhängt. Eine hidICE-Implementierung erlaubt an dieser Stelle Beobachtung mit unverminderter Taktfrequenz über einen beliebig langen Zeitraum, sofern die Emulation hinreichend schnell implementiert werden kann.

Die **gleichzeitige Durchführung mehrerer Beobachtungsaufgaben** kann gut mittels Software-Instrumentierung erreicht werden. Bei „embedded Trace“-Lösungen besteht das Problem, dass aufgrund der oftmals zu geringen Bandbreite zur Ausgabe der Trace-Daten diese stark gefiltert werden müssen. Wenn diese Filter nicht kontextspezifisch angepasst werden können (was nur bei ei-

ner hinreichend schnellen Online-Verarbeitung der Trace-Daten möglich ist), ergibt sich eine entsprechende Limitation bei der gleichzeitigen Durchführung verschiedener Beobachtungsaufgaben. Da bei einer hidICE-Implementierung alle in der Emulation beobachtbaren Informationen verfügbar sind, können sämtliche interessierenden Informationen für die weitere Verarbeitung zugänglich gemacht werden.

Die permanente Instrumentierung führt zu einer deutlichen **Beeinflussung des SoCs**, da der zusätzliche Programmcode entsprechende Ressourcen und Rechenzeit belegt. Die Entfernung der temporären Instrumentierung umgeht dieses Problem. Ebenso können „embedded Trace"-Lösungen so konfiguriert werden, dass kein oder nur ein geringer Einfluss auf den SoC besteht. Ähnlich verhält sich eine hidICE-Implementierung, auch hier führt die Ausgabe der Synchronisations-Informationen nicht zu einer Beeinflussung des SoCs.

Die Zugänglichmachung von I/O-Ports, die für die Beobachtung des SoCs belegt werden, ist bei einer Software-Instrumentierung kaum möglich. Bei einigen „embedded Trace"-Lösungen ist eine bidirektionale Port-Ersetzung zwar in der Spezifikation vorgesehen (z.B. bei Nexus, TCODE 20/21), wird aber in vielen Implementierungen nicht unterstützt. Nachteilig wirkt sich hier vor allem die unbestimmte Latenz zwischen dem Programmlauf und dem Ereignis auf dem I/O-Port aus. Durch die deterministische Synchronisation der Emulation im Rahmen einer hidICE-Implementierung ist hier die Port-Ersetzung von Ausgangspins mit wohldefiniertem Delay möglich. Die Nachbildung der Funktionalität von Eingangs-Pins kann von einer hidICE-Implementierung allerdings nicht unterstützt werden.

Sofern die erforderlichen Ressourcen verfügbar sind (z.B. zusätzlicher Speicher, Rechenzeit) eignet sich eine permanente oder temporäre Instrumentierung sehr gut für die **Beobachtung von SoCs aus der Serienproduktion**. Der für „embedded Trace"-Lösungen erforderliche Implementierungsaufwand (Chipfläche, I/O-Pins) führt bei vielen SoC-Familien dazu, dass nur einige Derivate mit umfassender „embedded Trace"-Unterstützung gebaut werden, andere verfügen oftmals nur über eingeschränkte Beobachtungsmöglichkeiten. Da die hidICE-basierte Synchronisation nur einen geringen Mehraufwand erfordert, kann diese ohne weitere Nachteile innerhalb einer SoC-Familie einheitlich angeboten werden.

Für die **Beobachtbarkeit in einer „realen" Umgebung** bei Integrationstests und höheren Teststufen ist es wichtig, dass die Schnittstelle zur Ausgabe der Zustandsinformationen auf eine robuste Art und Weise implementiert werden kann. Bei der permanenten Software-Instrumentierung können dazu geeignete robuste standardisierte Kommunikationsschnittstellen (z.B. UART, CAN, Ethernet) verwendet werden. Da die temporäre Software-Instrumentierung aufgrund der Beeinflussung des Zeitverhaltens und des Ressourcenverbrauchs bei Integrationstests nur bedingt einsetzbar ist, eignet sie sich entsprechend nicht zur Beobachtung eines SoCs in seiner realen Umgebung. Die zur Ausgabe der Trace-Daten bzw. der hidICE-Synchronisationsinformationen benötigten Schnittstellen müssen über eine relativ hohe Bandbreite verfügen. Um hier auch außerhalb eines Entwicklungslabors eine stabile Kommunikation zu gewährleisten, können die entsprechenden Daten mittels einer geeigneten Schnittstellenanpassung (z.B. via Lichtwellenleiter) störsicher zum Entwicklungswerkzeug übertragen werden.

Die **Latenz** zwischen den zu beobachtenden Vorgängen im SoC und deren Bekanntwerden im Entwicklungswerkzeug sollte möglichst kurz und deterministisch sein. Bei einer Software-Instrumentierung kann dies nur mit sehr hohem Aufwand erreicht werden. Werden die von einer „embedded Trace"-Lösung ausgegebenen Trace-Daten online verarbeitet, so kann mit einer geringen, aber nicht deterministischen Latenz auf Vorgänge im SoC reagiert werden. Eine hidICE-Implementierung bietet aufgrund ihres systembedingten Determinismus eine äußerst kurze wohldefinierte Latenz.

Ein sehr wichtiges Kriterium für die Qualität der Beobachtbarkeit ist die **Flexibilität des Beobachtungsfokus**. Dabei soll es möglich sein, kurzfristig die Auswahl der zu beobachtenden Ereignisse

zu beeinflussen. Bei der Software-Instrumentierung ist bei jeder Änderung ein erneutes Kompilieren erforderlich, was bei einigen Anwendungsgebieten, besonders in sicherheitskritischen Bereichen eine unangenehme Limitation darstellt. Die Einstellungen für die Auswahl zu beobachtender Informationen können bei „embedded Trace"-Lösungen vor dem Programmstart oder auch während des Laufs eines Programms verändert werden. An dieser Stelle ist ein großer Vorteil der Online-Verarbeitung von Trace-Daten erkennbar, da die „embedded Trace"-Einheit situationsspezifisch (z.B. bei erkannten Taskwechseln) beeinflusst werden kann. Bei einer hidICE-Implementierung kann der Beobachtungsfokus ebenfalls auf einfache Art und Weise angepasst werden, da innerhalb der Emulation alle beobachtbaren Informationen verfügbar sind.

Die **Klassifizierung für die Qualifizierung** der einzelnen Beobachtungsverfahren für sicherheitskritische Anwendungen sollte möglichst niedrig sein, um den Qualifizierungsaufwand gering halten zu können. Da bei permanenter Software-Instrumentierung zusätzlicher Programmcode in das Produkt eingebracht wird, welcher zu einer eventuellen Verletzung von Sicherheitsmerkmalen führt, sind hier die Anforderungen an die Qualifizierung besonders hoch. Einfacher ist hier die temporäre Software-Instrumentierung, welche vor allem für Coverage-Nachweise verwendet wird. „Embedded Trace"-Lösungen können, eine entsprechende Konfiguration vorausgesetzt, das System nicht-intrusiv beobachten. Gleiches gilt für eine Synchronisation mittels hidICE.

Beim Vergleich der **Kosten** der einzelnen Beobachtungsverfahren müssen verschiedene Aspekte berücksichtigt werden. Beim **Ressourcen- bzw. Chipflächenverbrauch** bietet die temporäre Software-Instrumentierung die vorteilhafteste Lösung. Verbleibt die Software-Instrumentierung im Endprodukt, sind dafür viele Ressourcen (Speicher, Rechenzeit, Energie) erforderlich. Die dafür erforderliche äquivalente Chipfläche ist oftmals größer als bei „embedded Trace"-Lösungen. Je nach Komplexität des zu beobachtenden SoCs kommt die Zusammenstellung und Ausgabe der hidICE-Synchronisationsinformationen mit einer vergleichsweise geringen Fläche Chipfläche aus. Bei komplexeren SoCs mit mehreren busmasterfähigen Peripherieeinheiten sowie unterschiedlichen Clock-Domains kann die erforderliche Chipfläche aber deutlich größer als bei einfacheren SoCs sein.

Der **Implementierungsaufwand** für Software-Instrumentierungen ist aufgrund der verfügbaren leistungsfähigen Werkzeuge sehr gering. „Embedded Trace"-Lösungen existieren bereits als vorgefertigte und konfigurierbare IPs, die beim Design eines SoCs auf einfache Art mit aufgenommen werden können. Eine entsprechende hidICE- Implementierung erfordert hier den größten Aufwand, da individuell ein optimaler Weg zur Ausgabe der Synchronisations-Informationen und zur Bereitstellung der Emulation gefunden werden muss. Dazu kommt eventuell ein zusätzlicher Aufwand zur Synchronisation unterschiedlicher Clock-Domains sowie von busmasterfähigen Peripherieeinheiten. Besonders unangenehm ist an dieser Stelle, dass auch fertige und bereits erprobte IPs eventuell geändert werden müssen, um die Synchronisation zu gewährleisten.

Die **Kosten der Entwicklungswerkzeuge** für die Software-Instrumentierung sind vergleichsweise niedrig. „Embedded Trace"-Lösungen mit einem brauchbaren Leistungsumfang sind deutlich teurer, wobei hier noch einmal ein Preissprung zwischen Offline-Auswertung und der aufwändigeren Online-Auswertung der Trace-Daten besteht. Bei einer hidICE-Emulation entstehen hohe Kosten vor allem durch die Bereitstellung der Emulation sowie durch die Weiterverarbeitung der breitbandig verfügbaren Beobachtungsergebnisse.

Bei der permanenten Software-Instrumentierung müssen die durch den zusätzlich auszuführenden Code verursachten **Energiekosten** berücksichtigt werden. Bei „embedded Trace"-Lösungen sowie der hidICE-Synchronisation können die Energiekosten oftmals vernachlässigt werden, da die für die Beobachtung erforderlichen Funktionseinheiten bei Inaktivität ausgeschaltet werden können.

Software-Instrumentierungen sind weitgehend unabhängig von der darunter liegenden Hardware, so dass sie eine sehr große **Universalität** besitzen. „Embedded Trace"-Lösungen setzen die Verfügbarkeit einer entsprechenden Hardwareunterstützung voraus und können mit unterschiedlicher Leistungsfähigkeit für beliebig komplexe SoCs implementiert werden. hidICE-Implementierungen

unterliegen einigen Restriktionen, bei zu hohen I/O- Bandbreiten, einer Vielzahl von busmasterfähigen Peripherieeinheiten sowie vielen unterschiedlichen Clock-Domains kann der Implementierungsaufwand unerreichbar hoch werden. Da es sich hier um eine „Alles oder Nichts“-Lösung handelt, ist die bei „embedded Trace“-Lösungen mögliche **Skalierbarkeit** der Beobachtungsqualität nicht realisierbar.

		Software-Instrumentierung		Embedded Trace		hidICE
		permanent	temporär	Offline Aus-wertung	Online Aus-wertung	
Vollständigkeit der Beobachtung	Taskwechsel	+	+	++	++	++
	Funktionen	+	+	++	++	++
	Basic Blocks	--	+	++	++	++
	Instruktionen	--	+	++	++	++
	Sprünge	--	+	++	++	++
	Datenzugriffe (CPU)	-	+	+	+	++
	Datenzugriffe (Peripherie)	--	--	o	o	++
	CPU-Register	--	o	o	o	++
	Cache	--	o	o	o	++
	Abarbeitung	--	--	++	++	++
	Zyklusgenauigkeit	--	--	+	+	++
	Bussystem	--	--	+	+	++
	Umgebungsbedingungen	o	o	+	+	++
	Ereignisse	o	--	+	+	++
Beobachtung von MPSoCs		o	--	o	o	++
Echtzeitfähigkeit		++	na	+	+	++
Kontinuität		++	na	-	+	++
Gleichzeitige Durchführung mehrerer Beobachtungsaufgaben		+	+	-	+	++
Beeinflussung des SoCs		-	++	++	++	++
Port Ersetzung	Ausgang	--	na	--	+	++
	Eingang	--	na	--	+	--
Beobachtbarkeit von SoCs aus der Serienproduktion		++	++	+	+	++
Beobachtbarkeit in „realer“ Umgebung		++	--	+	+	+
Latenz		-	-	--	+	++

		Software-Instrumentierung		Embedded Trace		hidICE
		permanent	temporär	Offline Aus-wertung	Online Aus-wertung	
Flexibilität des Beobachtungsfokus		-	-	o	+	++
Klassifizierung für die Qualifizierung	IEC 61508	*online support tool*	*offline support tool* / T2	*offline support tool* / T2	*offline support tool* / T2	*offline support tool* / T2
	DO-330	Krit. 1	Krit. 2	Krit. 2	Krit. 2	Krit. 2
	ISO 26262	TCL2	TCL1	TCL1 / TCL2	TCL1/ TCL2	TCL1
Kosten	Chipfläche / Ressourcen	--	++	o	o	+
	Implementierungsaufwand	++	++	+	+	-
	Entwicklungswerkzeuge	+	+	o	o	o
	Energiekosten	--	++	+	+	+
Universalität		++	++	+	+	-
Skalierbarkeit		++	++	++	++	--

Tabelle 6-1: Vergleich der Merkmale der einzelnen Beobachtungslösungen[21]

[21] Legende:
sehr gut geeignet: „++“ gut geeignet: „+“ mit Einschränkungen geeignet: „o“
wenig geeignet: „-“ nicht geeignet: „--“ nicht anwendbar „na“

6.2 Erfüllung verschiedener Fragestellungen und Anwendungsfälle

In Tabelle 6-2 wird die Eignung der diskutierten Beobachtungsverfahren zur Bearbeitung der in Abschnitt 2 dargestellten Fragestellungen und Anwendungsfälle bewertet. Auch diese Bewertungen sind subjektiv geprägt und müssen für den individuellen Anwendungsfall jeweils kritisch geprüft werden.

Bei Softwaretests muss zwischen funktionalen Tests und dem Nachweis der Testüberdeckungen unterschieden werden.

Für die Beobachtung der Ergebnisse **funktionaler Tests** auf der Ebene von Komponententests ist die Software-Instrumentierung sehr gut geeignet. Ab der Stufe der Integrationstests, welche auch Abhängigkeiten vom Zeitverhalten und dem Ressourcenverbrauch beinhalten, sind temporäre Software-Instrumentierungen wenig oder nicht geeignet, da diese das zu beobachtende System beeinflussen und damit die Testergebnisse verfälschen. Eine permanente Software-Instrumentierung bewirkt keine derartige Verfälschung, allerdings ist der Ressourcenaufwand entsprechend hoch. Die Beobachtung einiger Aspekte wie z.B. Cache-Effizienz, Ausführungszeiten oder der Betrieb von Peripherieeinheiten lassen sich oftmals nur indirekt oder gar nicht gewährleisten.

Voraussetzung für die Beobachtung vieler Ergebnisse funktionaler Tests mittels „embedded Trace“-Lösungen ist die Verfügbarkeit eines Daten-Traces. Dies gilt besonders für Komponenten- und SW/SW Integrationstests. Lang andauernde Beobachtungen, wie sie oft bei Ressourcen- und Systemtests erforderlich sind, lassen sich besonders gut mittels Online-Auswertung der Trace-Daten realisieren.

Durch die umfassende Beobachtbarkeit mittels einer hidICE-Implementierung ist dieses Verfahren ohne Einschränkungen für die Beobachtung der Ergebnisse funktionaler Tests geeignet.

Zum Nachweis der **Testüberdeckung** für Komponententests kann gut die temporäre Software-Instrumentierung verwendet werden, sofern nach dem Entfernen der Instrumentierung die funktionalen Tests noch einmal wiederholt werden. Die permanente Instrumentierung ist an dieser Stelle bis auf sehr wenige Ausnahmen nicht geeignet, da sie in Relation zum produktiven Code ein vielfaches Volumen zu dessen Beobachtung benötigt. Gleiches gilt für den Ressourcenbedarf des Nachweises der Daten-Überdeckung, auch hier ist die Software-Instrumentierung nur sehr eingeschränkt verwendbar.

„Embedded Trace“-Lösungen eignen sich sehr gut für den Nachweis von Testüberdeckungen des Programmablaufs. Lediglich bei einer MC/DC-Analyse muss teilweise ein Daten-Trace verfügbar sein (*unique-cause MC/DC)*, sofern nicht auch die Beobachtung anhand aufgezeichneter Sprunginformationen (*masked MC/DC*) akzeptabel ist. Der Nachweis der Datenüberdeckung erfordert die Verfügbarkeit eines Daten-Traces sowie eine hinreichend lange Beobachtungszeit, die vorteilhafterweise mittels Online-Auswertung der Trace-Daten erreicht werden kann.

Für hidICE-Implementierungen gilt auch hier wieder, dass diese aufgrund des erzielbaren Beobachtungsumfangs uneingeschränkt für alle Überdeckungstests geeignet sind.

Die Effizienz des **Debuggens** ist in hohem Maße von einer umfassenden Beobachtbarkeit abhängig. Einfache Debug-Aufgaben können dabei teilweise mittels individueller Software-Instrumentierung (`printf()`) gelöst werden. Allerdings muss an dieser Stelle die Problematik der Heisenbugs berücksichtigt werden. Komplexere Aufgaben wie das omnisciente Debuggen, Runtime Verification oder die Verfolgung lokaler Variablen sind mittels Software-Instrumentierung eher nicht möglich.

Die Eignung von „embedded Trace“-Lösungen zum Debuggen ist in vielen Fällen von der möglichen Beobachtungsdauer abhängig. Hier ist die besondere Eignung der Online-Auswertung von Trace-Daten zu erkennen. Die Beobachtung lokaler Variablen ist von einem ausreichend leistungsfähigen Daten-Trace in Kombination mit einer hinreichend langen Beobachtungsdauer abhängig und deshalb für die meisten Implementierungen nicht verfügbar.

Eine hidICE-Implementierung bietet aufgrund der damit möglichen uneingeschränkten Beobachtbarkeit optimale Voraussetzungen für ein effizientes Debuggen.

Die Messung maximaler Ausführungszeiten anhand der Beobachtung von Kanten des Kontrollflussgraphen (**hybride WCET-Messung**) sowie der taskspezifischen **CPU-Last** sollte über einen möglichst langen Zeitraum erfolgen. Besonders vorteilhaft kann dies mittels der Online-Auswertung von Trace-Daten sowie mittels einer hidICE-Implementierung erfolgen. Die Offline-Auswertung von Trace-Daten ist für diese Aufgabe weniger geeignet, da die Beobachtungsdauer durch die Größe des Zwischenspeichers sowie der Verarbeitungsgeschwindigkeit der Trace-Daten deutlich eingeschränkt wird.

Für die Erkennung von **Parallelitätsfehlern** ist eine temporäre Instrumentierung nicht geeignet, da sie das zeitliche Verhalten sowie den Ressourcenverbrauch der Anwendung beeinflusst. Auch eine permanente Instrumentierung ist an dieser Stelle oftmals schwierig einsetzbar, da die ausführende CPU nur einen Blick auf sich selbst hat und nicht gleichzeitig die Arbeit der anderen CPUs beobachten kann. Weil Parallelitätsfehler oftmals ein nicht deterministisches Erscheinungsbild haben, ist die Wahrscheinlichkeit des Erkennens von der Beobachtungsdauer abhängig. Aus diesem Grund ist für diese Beobachtung die Online-Auswertung von Trace-Daten oder alternativ eine hidICE-Implementierung besonders gut geeignet.

Die Optimierung des **Speicherlayouts**, welches die Kontrolle von Stack und Heap sowie die Beobachtung der Effizienz des Caches umfasst, ist mit einer permanenten Instrumentierung nur mit Einschränkungen möglich. Eine temporäre Instrumentierung ist an dieser Stelle nicht geeignet. Das Beobachtungsergebnis ist vom Ressourcenverbrauch (Speicher) sowie vom Zeitverhalten abhängig und würde durch die Entfernung der Instrumentierung verfälscht werden. „Embedded Trace"-Lösungen eignen sich grundsätzlich gut für diese Beobachtungsaufgaben, besonders vorteilhaft ist auch hier die mittels Online-Auswertung der Trace-Daten erzielbare unbegrenzte Beobachtungsdauer. Gleiches gilt für eine hidICE- Implementierung, auch diese kann optimal für Beobachtungsaufgaben in diesem Umfeld eingesetzt werden.

Bei der **Optimierung des Energieverbrauchs** sowie der Beobachtung von **Umgebungsparametern** müssen oftmals interne Systemzustände mit extern erfassten Messwerten korreliert werden. Dieser Prozess ist abhängig von Ausführungszeiten und Ressourcenverbrauch, so dass eine temporäre Software-Instrumentierung keine zuverlässigen Ergebnisse liefert. Eine permanente Instrumentierung ist mit Einschränkungen verwendbar. Besser geeignet ist die Beobachtung mittels „embedded Trace"-Lösungen, hier besonders die Beobachtung über einen beliebigen Zeitraum durch eine Online-Auswertung der Trace-Daten. Ebenso sehr gut für diesen Zweck geeignet ist eine hidICE-Implementierung.

		Software-Instrumentierung		Embedded Trace		hidICE
		permanent	temporär	Offline Auswertung	Online Auswertung	
Funktionale Tests	Komponententest	++	++	o	o	++
	SW/SW Integrationstest	+	-	o	o	++
	Ressourcentest	o	--	-	+	++
	HW/SW Integrationstest	o	--	o	+	++
	Systemtest	o	--	-	+	++
Überdeckungsnachweise	Funktions-überdeckung	+	+	++	++	++
	Anweisungs-überdeckung	-	+	++	++	++
	Zweigüberdeckung	-	+	++	++	++
	MC/DC	-	+	+	+	++
	Daten-Überdeckung	-	--	-	+	++
Debuggen	Einfaches Debuggen	-	-	+	+	++
	Omniscientes Debuggen	--	--	o	+	++
	Runtime Verification	--	--	o	+	++
	Lokale Variable	--	--	-	o	++
CPU-Last, WCET-Messung		-	--	+	++	++
Parallelitätsfehler		-	--	-	+	++
Speicherlayout		-	--	+	++	++
Optimierung des Energieverbrauchs		-	--	+	++	++
Umgebungsparameter		-	--	+	++	++

Tabelle 6-2: Vergleich der Eignung einzelner Beobachtungslösungen für bestimmte Fragestellungen und Anwendungsfälle[22]

[22] Legende:
sehr gut geeignet: „++“ gut geeignet: „+“ mit Einschränkungen geeignet: „o“
wenig geeignet: „-“ nicht geeignet: „--“

6.3 Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der effizienten Beobachtbarkeit von SoCs.

Grundlage für die Bewertung etablierter Verfahren sowie eines neuartigen Ansatzes war dabei die Erarbeitung eines Anforderungskatalogs, der sich aus der Analyse allgemeiner Anwendungen wie dem Debuggen und Testen sowie spezielleren Fragestellungen ergab. Letztere sind die Durchführung von Überdeckungsanalysen, die Ermittlung von maximalen Ausführungszeiten, die Analyse von Parallelitätsfehlern, die Beobachtung von Variablen und des Ressourcenverbrauchs, die Analyse des Caches und die Optimierung des Speicherlayouts, die Optimierung des Energieverbrauchs sowie die Beobachtung von Umgebungsbedingungen und die Erkennung von Störungen.

Nachfolgend wurden diese Anforderungen generalisiert und um weitere Kriterien wie Vollständigkeit, Echtzeitfähigkeit, Gleichzeitigkeit, Intrusivität, Flexibilität, Universalität, Qualifizierbarkeit und Kosten von Beobachtungslösungen ergänzt. Spezielles Augenmerk wurde dabei auf zukünftige Herausforderungen gelegt, die die zunehmende Verbreitung von MPSoCs mit sich bringen werden.

Es wurde eine Analyse von dem aktuellen Stand der Technik entsprechenden Beobachtungslösungen durchgeführt, wobei hier besonders auf die am weitesten verbreiteten Technologien eingegangen wurde. Dies sind die softwaregestützte Code-Instrumentierung sowie „embedded Trace“-Lösungen. Dabei wurde gezeigt, dass die zuvor erarbeiteten Kriterien von diesen Lösungen nur eingeschränkt erfüllt werden. Eine wesentliche Limitation stellt dabei die verfügbare Bandbreite für die Ausgabe von Trace-Nachrichten vom SoC dar. Dies macht hier eine Reihe von Kompromissen notwendig, die sich besonders auf die Vollständigkeit, die Flexibilität und die Kontinuität der Beobachtbarkeit von SoCs nachteilig auswirken. Dies betrifft besonders die zunehmend eingesetzten MPSoCs.

Mittels eines neuartigen Beobachtungsansatzes kann ein wesentlicher Teil der zuvor dargestellten Kriterien an die Beobachtbarkeit in nahezu idealer Weise erfüllt und Limitationen der bestehenden Lösungen überwunden werden. Dies gelingt durch den neuen Charakter der vom SoC zu Beobachtungszwecken ausgegeben Informationen. Statt der üblichen Trace-Nachrichten der „embedded Trace“-Lösungen werden Informationen ausgegeben, die es einer parallel laufenden Emulation ermöglichen, die Vorgänge in zentralen Bereichen des zu beobachtenden SoCs exakt nachzubilden. Diese externe Emulation kann nun deutlich besser beobachtet werden, die technischen und ökonomischen Zwänge zur Begrenzung der für die Ausgabe der Trace-Daten erforderlichen Bandbreite sind hier nicht mehr gegeben. Die vorgestellte Lösung gestattet einen äußerst detaillierten Blick in das Innere des SoC, so ist beispielsweise die vollständige und kontinuierliche Beobachtung von Busaktivitäten sowie von CPU-Registern möglich. Weiterhin wurde eine Methode zur Ersetzung von Ausgangspins entwickelt, mit deren Hilfe auch während einer aktiven Beobachtung alle Ausgangspins des SoC für die Anwendung verfügbar sind. Zusätzlich wird ein Verfahren erläutert, mit dessen Unterstützung die Verlässlichkeit der gewonnenen Beobachtungsergebnisse nachgewiesen werden kann. Das Funktionsprinzip des neuartigen Beobachtungsverfahrens wird für verschiedene Implementierungsvarianten diskutiert sowie Schwierigkeiten und Grenzen der Methode dargestellt.

Der Aufwand für die Implementierung einer hidICE-basierten Emulation kann für einfache SoCs relativ gering sein, wie dies auch mittels der in dieser Arbeit beschriebenen FPGA-basierten Implementierungen demonstriert werden konnte. Für SoCs mit hoher I/O-Bandbreite, vielen busmasterfähigen Peripherieeinheiten sowie mehreren unabhängigen Clock-Domains gestaltet sich die Bereitstellung der für die Synchronisation erforderlichen Informationen deutlich aufwändiger. Ein weiterer Nachteil ist die fehlende Skalierbarkeit des vorgestellten Beobachtungsverfahrens. Während „embedded Trace“-Lösungen auch mit geringer Trace Bandbreite (bei entsprechend vermindertem Zugriff auf interne Zustandsinformationen) implementierbar sind, muss die Synchronisation für den zu emulierenden Bereich eine hidICE-Implementierung dem „Alles oder Nichts“-Prinzip folgen.

Ist eine hidICE-basierte Emulation verfügbar, so eignet diese sich in idealer Weise zur Durchführung von funktionalen Tests und Überdeckungsanalysen, für das Debuggen, für diverse Optimierungen und besonders auch für die Beobachtung von MPSoCs.

6.4 Ausblick

In den vorangegangenen Ausführungen wurde gezeigt, wie wichtig eine umfassende Beobachtbarkeit für das Testen, das Debuggen, die Zertifizierung, die Optimierung und damit für die Effizienz und Zuverlässigkeit von eingebetteten Systemen ist. Gemessen an der sich daraus ergebenden wirtschaftlichen Bedeutung ist die aktuelle Situation teilweise verbesserungswürdig.

Neben den begrenzten Möglichkeiten der Software-Instrumentierung stellen „embedded Trace"-Lösungen einen Großteil der verfügbaren Beobachtungslösungen dar. Dies sind vor allem die als Quasi-Standard geltende ARM Coresight-Architektur sowie einige alternative Lösungen (z.B. Freescales Nexus-Implementierungen, Infineons MCDS). Aufgrund der Komplexität und teilweisen schlechten bzw. nicht vorhandenen Dokumentation ist es nur einem sehr begrenzten Kreis von Firmen und Forschungseinrichtungen möglich, hierfür passende Entwicklungswerkzeuge herzustellen. Dies behindert die kreative Suche nach neuen Lösungen, Anwender müssen sich oftmals mit Produkten zufrieden geben, die in ihren Leistungsmerkmalen teilweise fernab von den eigentlichen Kundenbedürfnissen liegen. Dies führt dazu, dass selbst verfügbare Lösungen oftmals nicht konsequent von potentiellen Anwendern genutzt werden.

Diese Situation kann verbessert werden, wenn auch Strukturen zur Unterstützung neuartiger Beobachtungsmethoden auf SoCs mit implementiert werden. Dies kann beispielsweise der in dieser Arbeit vorgestellte hidICE-Ansatz sein. Weiterhin sollten SoC-Hersteller die implementierten Lösungen zur Beobachtung ihrer SoCs gut dokumentiert einem großen Kreis von interessierten Entwicklern zugänglich machen, um durch mehr Wettbewerb die Verfügbarkeit verbesserter Lösungen zu erreichen.

Wenn mittels geeigneter technischer Lösungen eine umfassende Beobachtbarkeit von Vorgängen innerhalb des SoC erreicht wurde, müssen die gewonnenen Daten entsprechend verarbeitet werden. Aktuell verfügbare Werkzeuge arbeiten hier meist mit einer Zwischenspeicherung der Trace-Daten und einer anschließenden Offline-Verarbeitung am PC. Problematisch an diesem Ansatz ist, dass oftmals Trace-Daten schneller empfangen werden als der PC diese verarbeiten kann. Dies führt folglich zu einem Überlauf des Zwischenspeichers und damit zu Beobachtungslücken. Gerade bei der Suche nach Ursachen von nicht deterministisch erscheinendem Fehlverhalten ist aber eine beliebig lange Beobachtungsdauer wichtig. Ein weiterer großer Nachteil der Offline-Verarbeitung besteht darin, dass die Steuerung der Ausgabe von Trace-Daten nicht kontextspezifisch erfolgen kann, da der aktuelle Stand der Programmabarbeitung dem Entwicklungssystem nur mit einer langen und nicht vorher bestimmbaren Latenz bekannt gemacht werden kann. Bei zukünftigen SoC-Generationen, besonders MPSoCs, wird sich diese Problematik noch verschärfen, eine Offline-Verarbeitung der Trace-Daten erscheint hier zunehmend als Sackgasse.

Einen Ausweg aus dieser Situation bietet die Verarbeitung der eingehenden Trace-Nachrichten in Echtzeit. Dieser Ansatz ist sowohl für eine auf einer hidICE-Emulation basierende Beobachtung eines SoC, als auch für Trace-Nachrichten, die von „embedded Trace"-Lösung stammen, anwendbar. Bei Letzterem stellt die Online-Auswertung des Trace-Protokolls eine besondere Schwierigkeit dar. Bei ARM CoreSight-Implementierungen müssen dazu die Grenzen der einzelnen Trace-Nachrichten bestimmt werden, ohne dass diese wie zum Beispiel bei Nexus-Implementierungen über einen entsprechenden Nebenkanal angezeigt werden. Eine weitere Schwierigkeit besteht in der Rekonstruktion der komprimiert übertragenen Sprunginformationen. In ersten experimentellen Implementierungen wurden Lösungen für diese Herausforderungen erarbeitet [145], welche das Protokoll des eingehenden Trace-Datenstroms vollständig interpretieren sowie die Instruktionsadressen in Echtzeit rekonstruieren. Vorbestimmte Adressen und Nachrichteninhalte führen zur Ausgabe von

mit Zeitstempeln versehenen Ereignis-Nachrichten, die an nachfolgenden Verarbeitungsstufen weitergegeben werden. Die allgemeine Verfügbarkeit einer solchen Lösung vorwegnehmend wurde dieser Ansatz in die vergleichenden Betrachtungen in Tabelle 6-1 und Tabelle 6-2 bereits mit aufgenommen (Embedded Trace mit Online-Auswertung).

Die weitergehende Ereignisverarbeitung erfordert aufgabenspezifische Verarbeitungseinheiten. Einmal können dies generische FPGA-Strukturen sein, die bereits synthetisiert sind und nur noch entsprechend der Aufgabenstellung parametrisiert werden. Diese sind für die gleichzeitige Bearbeitung einer Vielzahl von Beobachtungsaufgaben geeignet. Mit dieser Hardwareunterstützung ist es möglich, die Methodik der *Runtime Verification* zur Online-Analyse von Trace-Daten anzuwenden [146]. Dieser Ansatz trägt auch der Tatsache Rechnung, dass Systeme so komplex werden können, dass ein systematischer und weitgehend vollständiger Integrationstest nicht mehr durchführbar ist und deshalb das System im produktiven Betrieb umfassend beobachtbar sein muss, um die Aktivierung von Defekten (die nicht zwingend zu einem sichtbaren Fehlverhalten führen müssen) erkennen und den zugrundeliegenden Defekt beseitigen zu können [147].

Ein anderes Anwendungsgebiet ist die hybride WCET-Messung. Durch kontinuierliche Messung der Zeiten, die für die Abarbeitung einzelner Kanten von Kontrollflussgraphen benötigt werden, lässt sich - kombiniert mit dem Wissen über die Wege, die der Programmablauf nehmen kann – eine sehr fundierte Vorhersage über die maximale Ausführungszeiten eines Programms erstellen. Um eine Vielzahl von Basic Blocks beobachten zu können, ist an dieser Stelle die Verwendung von spezialisierten Hardwarestrukturen erforderlich.

Wenn die Beobachtung von Zugriffen auf von mehreren Prozessen gemeinsam genutzten Variablen möglich ist, können potentielle Kandidaten für *race conditions* ermittelt werden, ohne dabei das zeitliche Verhalten der Anwendung zu beeinflussen. Dazu müssen die Adresse der Variablen, die Art des Zugriffs (Lesen oder Schreiben) sowie der zugreifende Prozess identifiziert werden. Mit spezialisierten FPGA-Strukturen lässt sich dann der in Abschnitt 2 dargestellten „*Happens before*"-Algorithmus für eine Vielzahl von Variablen implementieren.

Eine weitere wichtige Anwendung der Echtzeitanalyse von Trace-Daten und der Rekonstruktion ausgeführter Instruktionen sind verschiedene Überdeckungsanalysen sowie die Ermittlung von Profiling-Informationen.

Zusammenfassend lässt sich sagen, dass das Thema der Beobachtung von SoCs sowie der effizienten Auswertung der erfassten Informationen ein wichtiger Faktor zur Verbesserung der Effizienz und Zuverlässigkeit von eingebetteten Systemen ist. Dabei sind bei weitem noch nicht alle vorstellbaren technischen Mittel ausgeschöpft - es ist zu erwarten, dass die Bedeutung dieses Arbeitsgebietes in den kommenden Jahren stark steigen wird.

Anhang

Literaturverzeichnis

[1] J. Ganssle, “Proactive Debugging,” 26-Feb-2001. [Online]. Available: http://www.embedded.com/electronics-blogs/break-points/4023293/Proactive-Debugging. [Accessed: 03-Jan-2014].

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Reversible Debugging Software.” Cambridge Judge Business School, 08-Jan-2013.

[3] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia, “Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers,” in *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, 2013, pp. 383–392.

[4] C. Jones and O. Bonsignour, *The Economics of Software Quality*. Addison-Wesley, 2011.

[5] G. Tassey, *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, 2002.

[6] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

[7] G. Hopper, “The First Bug,” *Ann. Hist. Comput.*, no. 3, pp. 285–286, Jul. 1981.

[8] T. P. Hughes, *American Genesis: A Century of Invention and Technological Enthusiasm, 1870-1970*. University of Chicago Press, 2004.

[9] “Photo #: NH 96566-KN - The First ‘Computer Bug.’” Naval Surface Warfare Center Computer Museum at Dahlgren, 1988.

[10] “Systems and software engineering – Vocabulary,” *ISOIECIEEE 247652010E*, pp. 1–418, 2010.

[11] D. J. Agans, *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. American Management Association, 2006.

[12] B. Fechner, *Transiente Fehler in Mikroprozessoren: Mechanismen zur Erkennung, Behebung und Tolerierung*. Vieweg+Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden, 2009.

[13] M. Barr, “BOOKOUT V. TOYOTA - 2005 Camry L4 Software Analysis,” 2013. [Online]. Available: http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf. [Accessed: 02-Jul-2014].

[14] S. Grünfelder, *Software-Test für Embedded Systems: Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter*. Dpunkt.Verlag GmbH, 2013.

[15] C. Hermanns, *Entwicklung und Implementierung eines hybriden Debuggers für Java.* University of Münster, 2010.

[16] M. Hamill and K. Goseva-Popstojanova, “Common Trends in Software Fault and Failure Data,” *Softw. Eng. IEEE Trans. On*, vol. 35, no. 4, pp. 484–496, 2009.

[17] M. Grottke and K. S. Trivedi, “Classification of Software Faults,” in *Proc. Sixteenth International IEEE Symposium on Software Reliability Engineering*, 2005.

[18] J. Gray, “Why Do Computers Stop and What Can Be Done About It?,” in *Symposium on Reliability in Distributed Software and Database Systems*, 1986, pp. 3–12.

[19] E. S. Raymond, *The new hacker’s dictionary (3rd ed.)*. Cambridge, MA, USA: MIT Press, 1996.

[20] J.-L. Lions, L. Lübeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O’Halloran, “ARIANE 5 Flight 501 Failure,” Ariane 501 Inquiry Board, Paris, Report, Jul. 1996.

[21] J. Alonso, M. Grottke, A. P. Nikora, and K. S. Trivedi, “The nature of the times to flight software failure during space missions,” presented at the 23rd IEEE International Symposium on Software Reliability Engineering, Dallas, TX USA, 2012.

[22] M. Grottke and K. S. Trivedi, “Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate,” *Computer*, vol. 40, no. 2, pp. 107–109, Feb. 2007.

[23] K. Vaidyanathan and K. S. Trivedi, “Extended Classification of Software Faults Based on Aging,” *Proc. Twelfth Int. Symp. Softw. Reliab. Eng. ISSRE IEEE*, 2001.

[24] “Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia,” Information Management and Technology Division, United States General Accounting Office, Washington, D.C., Report GAO/IMTEC-92-26, Feb. 1992.

[25] T. A. P. George N. Lewis, “Video evidence on the effectiveness of Patriot during the 1991 Gulf War,” *Sci. Glob. Secur.*, vol. 4, no. 1, pp. 1–63, 1993.

[26] S. Bourne, “A Conversation with Bruce Lindsay,” *Queue*, vol. 2, no. 8, pp. 22–33, Nov. 2004.

[27] M. Grottke, A. P. Nikora, and K. S. Trivedi, “An empirical investigation of fault types in space mission system software,” in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 2010, pp. 447–456.

[28] E. W. Dijkstra, “Structured programming,” O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. London, UK, UK: Academic Press Ltd., 1972, pp. 1–82.

[29] B. Boehm and V. R. Basili, “Software Defect Reduction Top 10 List,” *Computer*, vol. 34, no. 1, pp. 135–137, Jan. 2001.

[30] IEEE, “Software and systems engineering - Software measurement - IFPUG functional size measurement method,” *ISOIEC 209262009*, 2009.

[31] P. Liggesmeyer, *Software-Qualität : Testen, Analysieren und Verifizieren von Software*. Heidelberg: Spektrum, Akad. Verl., 2009.

[32] A. Kumar and J. St.Clair, “CUnit - A unit testing framework for C,” 2014. [Online]. Available: http://cunit.sourceforge.net/doc/index.html. [Accessed: 02-Jul-2014].

[33] M. Fowler, “Mocks aren’t stubs,” Jan-2007. [Online]. Available: http://martinfowler.com/articles/mocksArentStubs.html. [Accessed: 02-Jul-2014].

[34] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[35] “ISO 26262:2011. Road vehicles – Functional safety.” International Organization for Standardization Std., 2011.

[36] G. Pothier and E. Tanter, “Back to the Future: Omniscient Debugging,” *Softw. IEEE*, vol. 26, no. 6, pp. 78–85, 2009.

[37] T. Grötker, U. Holtmann, H. Keding, and M. Wloka, *The Developer’s Guide to Debugging*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[38] M. Leucker and C. Schallhart, “A Brief Account of Runtime Verification,” *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, Jun. 2009.

[39] H. Balzert, *Lehrbuch der Software-Technik, Bd. 2: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung, inkl. 1 CD-ROM*. Spektrum Akademischer Verlag, 1997.

[40] “DO-178C, Software Considerations in Airborne Systems and Equipment Certification.” RTCA, 13-Dec-2011.

[41] F. E. Allen, “Control Flow Analysis,” in *Proceedings of a Symposium on Compiler Optimization*, New York, NY, USA, 1970, pp. 1–19.

[42] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, “A Practical Tutorial on Modified Condition/Decision Coverage,” NASA Langley Technical Report Server, 2001.

[43] W. E. Howden, “An Evaluation of the Effectiveness of Symbolic Testing.,” *Softw Pr. Exper*, vol. 8, no. 4, pp. 381–397, 1978.

[44] W. E. Howden, “Theoretical and Empirical Studies of Program Testing.,” in *ICSE*, 1978, pp. 305–311.

[45] A. Westley, *Software testing - Infotech state of the art report*. Infotech, Maidenhead, Berkshire, 1979.

[46] “Position Paper CAST-10: What is a ‘Decision’ in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?” FAA Certification Authorities Software Team (CAST), Jun-2002.

[47] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, 2011.

[48] “DO-178B, Software Considerations in Airborne Systems and Equipment Certification.” RTCA, 1982.

[49] “DO-330, Software Tool Qualification Consideration.” RTCA, 13-Dec-2011.

[50] J. J. Chilenski and S. P. Miller, *Applicability of Modified Condition/decision Coverage to Software Testing*. 1994.

[51] P. Liggesmeyer, “Bedingungsüberdeckungstesttechniken: Vergleich, Bewertung und Anwendung in der Praxis,” *Softwaretechnik Trends*, vol. 21, no. 3, pp. 15–17, 2001.

[52] C. Comar, J. Guitton, O. Hainque, and T. Quinot, “Formalization and Comparison of MCDC and Object Branch Coverage Criteria,” in *ERTS 2012*, Toulouse - France, 2012.

[53] J. J. Chilenski, “An investigation of three forms of the modified condition decision coverage (mcdc) criterion,” Office of Aviation Research, 2001.

[54] “Position Paper CAST-17: Structural Coverage of Object Code.” FAA Certification Authorities Software Team (CAST), Jun-2003.

[55] P. V. Bhansali, “The MCDC paradox,” *SIGSOFT Softw Eng Notes*, vol. 32, no. 3, pp. 1–4, May 2007.

[56] S. Kandl and R. Kirner, “Error detection rate of MC/DC for a case study from the automotive domain,” in *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, Berlin, Heidelberg, 2010, pp. 131–142.

[57] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, “The worst-case execution-time problem\—overview of methods and survey of tools,” *ACM Trans Embed Comput Syst*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[58] R. L. Graham, “Bounds for certain multiprocessing anomalies,” *Bell Syst. Tech. J.*, vol. 45, pp. 1563–1581, 1966.

[59] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, “A Definition and Classification of Timing Anomalies,” in *6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[60] A. Betts, N. Merriam, and G. Bernat, “Hybrid measurement-based WCET analysis at the source level using object-level traces,” in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, Dagstuhl, Germany, 2010, vol. 15, pp. 54–63.

[61] U. Gleim and T. Schüle, *Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C*. Dpunkt.Verlag GmbH, 2011.

[62] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2008, pp. 329–339.

[63] N. G. Leveson and C. S. Turner, “An Investigation of the Therac-25 Accidents,” *Computer*, vol. 26, no. 7, pp. 18–41, Jul. 1993.

[64] J. W. Voung, R. Jhala, and S. Lerner, “RELAY: static race detection on millions of lines of code,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, NY, USA, 2007, pp. 205–214.

[65] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: a dynamic data race detector for multithreaded programs,” *ACM Trans Comput Syst*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[66] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[67] D. Brylow, N. Damgaard, and J. Palsberg, “Static checking of interrupt-driven software,” in *In Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE*, 2001, pp. 47–56.

[68] M. Loghi and M. Poncino, “Exploring Energy/Performance Tradeoffs in Shared Memory MPSoCs: Snoop-Based Cache Coherence vs. Software Solutions,” *2008 Des. Autom. Test Eur.*, vol. 1, pp. 508–513, 2005.

[69] A. Stevens, “Introduction to AMBA 4 ACE.” ARM Ltd., 06-Jun-2011.

[70] M. Verma and P. Marwedel, *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[71] M. Loghi, M. Poncino, and L. Benini, “Cycle-accurate power analysis for multiprocessor systems-on-a-chip,” in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, New York, NY, USA, 2004, pp. 410–406.

[72] J. Klein, “Sleep As You Can - Stromsparmodi in der Anwendung,” presented at the DESIGN&ELEKTRONIK Entwicklerforum, 2010.

[73] D. Peters, “AnalogIn ‘glitches’ --- a ‘cure’ and a tool,” 08-Aug-2011. [Online]. Available: http://mbed.org/forum/mbed/topic/1866/. [Accessed: 02-Jul-2014].

[74] *T4240 QorIQ Integrated Multicore Communications Processor Family Reference Manual*. Freescale Semiconductor, Inc., 2013.

[75] C.-Y. Chang, Y.-J. Chang, K.-J. Lee, J.-C. Yeh, S.-Y. Lin, and J.-L. Ma, “Design of on-chip bus with OCP interface,” in *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, 2010, pp. 211 –214.

[76] J.-J. Lecler and G. Baillieu, “Application driven network-on-chip architecture exploration & refinement for a complex SoC,” *Des. Autom Emb Sys*, vol. 15, no. 2, pp. 133–158, 2011.

[77] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, “A clustered manycore processor architecture for embedded and accelerated applications.,” in *HPEC*, 2013, pp. 1–6.

[78] C. Ciordas, T. Basten, A. Rădulescu, K. Goossens, and J. V. Meerbergen, “An event-based monitoring service for networks on chip,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, pp. 702–723, 2005.

[79] P. Gorski and D. Timmermann, “Centralized traffic monitoring for online-resizable clusters in Networks-on-Chip.,” in *ReCoSoC*, 2013, pp. 1–8.

[80] *TMS320DM816x DaVinci Digital Video Processors - Technical Reference Manual*. Texas Instruments Incorporated, 2013.

[81] *Cortex™-A8 Technical Reference Manual*. ARM Limited, 2010.

[82] *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*. Texas Instruments Incorporated, 2010.

[83] K. Goossens, B. Vermeulen, R. van Steeden, and M. Bennebroek, “Transaction-Based Communication-Centric Debug,” in *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, 2007, pp. 95 –106.

[84] J. Braunes and R. G. Spallek, “Generating the trace qualification configuration for MCDS from a high level language,” in *DATE*, 2009, pp. 1560–1563.

[85] J. Braunes and R. G. Spallek, “A High-Level Language and Compiler to Configure the Multi-core Debug Solution (MCDS),” in *Advances in System Testing and Validation Lifecycle, 2009. VALID ’09. First International Conference on*, 2009, pp. 62 –67.

[86] “IEC 61508. Functional safety of electrical/ electronic/programmable electronic safety-related systems.” International Electrotechnical Commission, 2010.

[87] “ED-12C - Software Considerations in Airborne Systems and Equipment Certification.” EUROCAE, 2011.

[88] “EN 50128:2011 Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems.” European Commitee for Electrotechnical Standardization, 2011.

[89] “DIN EN 62304. Medical device software – Software life cycle processes.” 2006.

[90] “IEC 62061 Safety of machinery – Functional safety of safety-related electrical, electronic and programmable electronic control systems.” International Electrotechnical Commission, 2005.

[91] “IEC 61511:2003 Functional safety - Safety instrumented systems for the process industry sector.” International Electrotechnical Commission, 2003.

[92] “IEC 60880:2006 Nuclear power plants – Instrumentation and control systems important to safety – Software aspects for computer-based systems performing category A functions.” International Electrotechnical Commission, 2006.

[93] “ED-215 - Software Tool Qualification Considerations.” EUROCAE, 2012.

[94] “IEC 60812:2006 Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA).” International Electrotechnical Commission, 2006.

[95] M. Conrad, P. Munier, and F. Rauch, “Qualifying Software Tools According to ISO 26262.,” in *MBEES*, 2010, pp. 117–128.

[96] *Gesetz über Medizinprodukte (Medizinproduktegesetz - MPG)*. 2013.

[97] X. Xia, X. Zhou, D. Lo, and X. Zhao, “An Empirical Study of Bugs in Software Build Systems,” in *Quality Software (QSIC), 2013 13th International Conference on*, 2013, pp. 200–203.

[98] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and Understanding Bugs in C Compilers,” *SIGPLAN Not*, vol. 46, no. 6, pp. 283–294, Jun. 2011.

[99] “Squish Coco - Code Coverage.” froglogic GmbH, 2012.

[100] O. Cole, “Aprobe: A Framework for Non-intrusive Software Instrumentation.” OC Systems, Inc., 2009.

[101] *CoreSight™ Components - Technical Reference Manual - DDI 0314H*. ARM Limited, 2009.

[102] IEEE-ISTO, “The Nexus 5001 Forum - Standard for a Global Embedded Processor Debug Interface,” *IEEE-ISTO 5001™-2012*, Jun. 2012.

[103] “Cantata++ Standard briefing - ISO26262 Road Vehicles - Functional Safety.” IPL Information Processing Limited, 2011.

[104] *i.MX53 Multimedia Applications Processor Reference Manual iMX53RM Rev. 2.1*. Freescale Semiconductor, Inc., 2012.

[105] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2012.

[106] “Lauterbach News 2011.” Lauterbach, 2011.

[107] *ARMv7-M Architecture Reference Manual*. ARM Limited, 2010.

[108] C.-F. Kao, S.-M. Huang, and I.-J. Huang, "A hardware approach to real-time program trace compression for embedded processors," *Circuits Syst. Regul. Pap. IEEE Trans. On*, vol. 54, no. 3, pp. 530–543, 2007.

[109] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," in *ICCAD*, 1994, pp. 384–390.

[110] "Logic Analyzer Probe for PowerTrace II," 2014. [Online]. Available: http://www.lauterbach.com/publications/trace32_logic_analyzer_probe_for_powertrace_ii.pdf. [Accessed: 02-Jul-2014].

[111] *CoreSight™ Technology System Design Guide DGI0012D*. ARM Limited, 2010.

[112] M. Williams, "ARM V8 debug and trace architectures," in *Proceedings of the 2012 Conference on System, Software, SoC and Silicon Debug*, Vienna, Austria, 2012.

[113] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[114] *Aurora Protocol Specification*. Xilinx Inc., 2007.

[115] *High Speed Serial Trace Port Architecture Specification*. ARM Limited, 2012.

[116] *Standard for Physical Connection for High-Speed Serial Trace*. Power.org, 2008.

[117] P. E. Sartain, A. B. T. Hopkins, and K. D. McDonald-Maier, "Optoelectronic Measurement Interface for System-on-Chip Debug," in *Instrumentation and Measurement Technology Conference Proceedings, 2007. IMTC 2007. IEEE*, 2007, pp. 1 –4.

[118] S. Kawai, T. Ikari, Y. Takikawa, H. Ishikuro, and T. Kuroda, "A wireless real-time on-chip bus trace system using quasi-synchronous parallel inductive coupling transceivers," in *Solid-State Circuits Conference, 2008. A-SSCC '08. IEEE Asian*, 2008, pp. 113 –116.

[119] *P4080 Advanced QorIQ Debug and Performance Monitoring Reference Manual, Rev. F*. Freescale Semiconductor, Inc., 2012.

[120] *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual*. Freescale Semiconductor, Inc., 2012.

[121] N. Guenow, R. Schnur, and M. Short, "Power Architecture Roadmap FTF-NET-F009," presented at the Freescale Technology Forum, San Antonio, Texas, USA, 2012.

[122] A. Mayer, H. Siebert, and K. D. McDonald-Maier, "Boosting Debugging Support for Complex Systems on Chip," *Computer*, vol. 40, no. 4, pp. 76–81, 2007.

[123] "AURIX Trace Training," 2014. [Online]. Available: http://www.lauterbach.com/pdfnew/training_aurix_trace.pdf. [Accessed: 02-Jul-2014].

[124] "New TriCore™ Brochure including AURIX™," Infineon Technologies AG, 2012.

[125] *Embedded Trace Macrocell Architecture Specification*. ARM Limited, 2011.

[126] A. Mayer, "Private Email-Korrespondenz: Fragen zu MCDS," 06-Sep-2013.

[127] G. Barret, J.-F. Pollet, and F. Lamotte, "Embedded microprocessor emulation method," US7451074, 11-Nov-2008.

[128] C. Hochberger and A. Weiss, "Acquiring an exhaustive, continuous and real-time trace from SoCs," in *ICCD*, 2008, pp. 356–362.

[129] A. Lange and A. Weiss, "Method and apparatus for emulating a programmable unit," EP1720100, 18-Jul-2007.

[130] A. Weiss, R. Backasch, and C. Hochberger, "An Observer Based System for Capturing Full and Real-Time Traces from Multi-Core SoCs," in *Proceedings of the 2010 Conference on System, Software, SoC and Silicon Debug*, Southampton, UK, 2010.

[131] C. Hochberger and A. Weiss, “Trace-Daten aus dem Parallel-Universum,” *Elektronik*, no. 24, pp. 82–86, 2005.

[132] *FM4 Family Peripheral Manual*. Fujitsu Semiconductor Ltd, 2013.

[133] *GRLIB IP Core User’s Manual Version 1.1.0*. Aeroflex Gaisler, 2012.

[134] S. King, “Private Email-Korrespondenz: Synchronization of DDR3 SDRAMs (Micron Technical Support),” 11-Sep-2012.

[135] A. Lange and A. Weiss, “Procedure and Device for Emulating a Programmable Unit Providing System Integrity Control,” US7930165B2, 19-Apr-2011.

[136] S. Lin and D. J. C. Jr., *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[137] J. C. Moreira and P. G. Farrell, *Essentials of Error-Control Coding*. Wiley, 2006.

[138] ISO, “Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 1: Connectionless-mode transmission,” ISO, International Standard ISO 7498-1, 1987.

[139] C. Hochberger and A. Weiss, “A New Methodology for Debugging and Validation of Soft Cores,” in *FPL*, 2008, pp. 551–554.

[140] C. Jiang, *Make the world safer with Hercules™ safety MCU platform*. Texas Instruments Incorporated, 2013.

[141] A. Weiss and C. Hochberger, “A New Methodology for the Test of SoCs and for Analyzing Elusive Failures,” in *MTV*, 2008, pp. 18–23.

[142] *PicoBlaze 8-bit Embedded Microcontroller User Guide*. Xilinx Inc., 2011.

[143] “CoreConsole Information.” [Online]. Available: http://www.actel.com/products/software/coreconsole/info.aspx. [Accessed: 30-Nov-2012].

[144] *CMOS-8L Family CMOS Gate Array Block Library*, vol. Document No. A12213XJ5V1UM00 (5th edition). NEC Corporation, 1998.

[145] A. Weiss, “European Patent Application: Trace-Data Processing and Profiling Device,” EP13192942, 14-Nov-2013.

[146] R. Backasch, C. Hochberger, A. Weiss, M. Leucker, and R. Lasslop, “Runtime Verification for Multicore SoC with High-quality Trace Data,” *ACM Trans Autom Electron Syst*, vol. 18, no. 2, pp. 18:1–18:26, Apr. 2013.

[147] B. Hanke and F. Schulz, “Master Thesis: Assessment of multi-core integration infrastructure for military avionics,” University of German Armed Forces Munich, Manching, Munich, 2014.