Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Implementierung von Java-Threads in Software und rekonfigurierbarer Hardware

Diplomarbeit

Leipzig, April 2008

vorgelegt von
Stefan Endrullis
Studiengang Informatik

Betreuender Hochschullehrer: Prof. Dr. Udo Kebschull
Institut für Informatik, Abteilung Technische Informatik

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die diese Arbeit unterstützt haben. Mein besonderer Dank gilt Prof. Dr. Udo Kebschull für die Möglichkeit, diese Arbeit schreiben zu können und seine intensive Unterstützung. Ebenso danke ich meinem Betreuer Norbert Abel sehr für die durch ihn gelegten Grundlagen, auf denen diese Arbeit aufbaut. Ich danke ihm ebenfalls für seine wertvollen Hinweise und Ratschläge sowie für seine ständige Bereitschaft, mir zu helfen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Markt tragbarer Geräte gewinnt eine immer stärkere Bedeutung. Mobiltelefone, PDAs (Personal Digital Assistant), Smartphones und viele weitere Geräte werden kontinuierlich mit neuen Funktionen ausgestattet und übernehmen zunehmend klassische Aufgaben eines Personal Computers (PC), wie beispielsweise die Textverarbeitung oder die Ausführung multimedialer Anwendungen. Speziell letztere stellen an die Geräte hohe Anforderungen, die sich nicht allein durch den Einsatz leistungsstärkerer Prozessoren lösen lassen. Nicht selten werden deshalb für rechenaufwendige Arbeiten Chips zur Umsetzung der speziellen Anforderungen in Hardware eingesetzt. Diese werden als Application Specific Integrated Circuit (ASIC) bezeichnet. Auf ihrem konkreten Einsatzgebiet unterstützen und entlasten sie den Hauptprozessor. Der Vorteil der ASICs gegenüber einer Software-Implementierung liegt in der deutlich schnelleren Datenverarbeitung, da die Hardware eine Vielzahl von Möglichkeiten zur Parallelisierung der einzelnen Verarbeitungsschritte eröffnet. ASICs haben jedoch den großen Nachteil, dass sie nach ihrer Fertigung nicht mehr angepasst werden können. Ausbesserungen von Fehlern oder Funktionserweiterungen sind daher mit einem kompletten Austausch der Chips verbunden. Demgegenüber zeichnet sich Software vor allem durch ihre Flexibilität und hervorragenden Änderungsmöglichkeiten auch nach der Produktauslieferung aus. Bei der Entwicklung eines tragbaren Gerätes gilt es deshalb, einen geeigneten Kompromiss zwischen Flexibilität der Software und Geschwindigkeit der Hardware zu finden.

Field Programmable Gate Arrays (FPGA) könnten zukünftig diesen Abwägungsprozess zwischen Software- und Hardwareanteilen stark vereinfachen. Sie besitzen die Möglichkeit zur dynamischen Rekonfiguration und vereinen somit die wesentlichen Vorteile von Software und Hardware – Flexibilität und Geschwindigkeit. Die FPGAs der Firma Xilinx sind zudem in der Lage, gezielt Teile der FPGA-Fläche zur Laufzeit

mit einer neuen Schaltung zu versehen. Damit ergeben sich zahlreiche Einsatzmöglichkeiten, die in der Praxis bislang hauptsächlich an den höheren Herstellungskosten der FPGAs gegenüber ASICs scheitern. Um Kosten zu sparen, ist es deshalb wichtig, die Fläche des FPGAs möglichst effizient zu nutzen.

Zu diesem Zweck entstand die Idee des Hardware-Schedulings, welches das aus der Software bekannte Scheduling von Prozessen auf dynamisch rekonfigurierbare Hardware überträgt. In diesem Zusammenhang spricht man auch vom Multitasking, der Fähigkeit, mehrere Aufgaben (Tasks) nebenläufig auszuführen. Da eine parallele Ausführung mehrerer Tasks auf einem Prozessorkern nicht möglich ist, werden jedem Task durch einen Scheduler Zeitscheiben für die Ausführung auf dem Prozessor zugeordnet. Die Ausführung der Tasks erfolgt somit *quasiparallel*. Dieses Prinzip lässt sich ebenso auf die Hardware-Tasks (eigenständige Schaltungsteile) anwenden. Auf diese Weise können mit Hilfe des Hardware-Schedulings auch auf einer relativ kleinen FPGA-Fläche mehrere Hardware-Tasks ausgeführt werden.

Auf der Grundlage dieser Techniken wurde in der vorliegenden Arbeit ein Framework für die Programmiersprache Java geschaffen werden, welches das Thread-Konzept dahingehend erweitert, dass thread-ähnliche Tasks implementiert werden können, die sich sowohl in Software als auch in Hardware ausführen lassen. Die Tasks können miteinander kommunizieren und verfügen über die Fähigkeit, ihren Ausführungsmodus (ob Software oder Hardware) zur Laufzeit zu ändern. Darüber hinaus besitzt ein Task die gleichen Eigenschaften und Methoden wie ein Java-Thread. Dadurch können Threads in bestehenden Anwendungen leicht in äquivalente Tasks umgeformt werden.

Im nächsten Kapitel werden einige Grundlagen zu rekonfigurierbarer Hardware und Java erläutert, die für das Verständnis der Arbeit benötigt werden. Anschließend wird in Kapitel 3 eine Auswahl aktueller Forschungsarbeiten vorgestellt, die sich mit ähnlichen Themen auseinandergesetzt haben. Im darauffolgenden Kapitel findet die Beschreibung des Lösungsansatzes dieser Arbeit statt. Dabei werden die wesentlichen Aspekte der Implementierung beleuchtet. Auf Implementierungsdetails wird in Kapitel 5 eingegangen. Die dort aufgeführten Informationen sind richten sich in erste Linie an Entwickler, die die Arbeit im Detail nachvollziehen oder Weiterentwicklungen vornehmen möchten. Im Anschluss dazu werden die Ergebnisse dieser Arbeit vorgestellt und die Arbeit abschließend im Kapitel 7 noch einmal zusammengefasst.

Kapitel 2

Grundlagen

2.1 Konfigurierbare Hardware

Die Anfänge programmierbarer Hardware gehen bis in die 50er Jahre des letzten Jahrhunderts zurück. 1956 wurde durch Wen Tsing Chowder der Programmable Read-Only Memory (PROM) entwickelt. Das elektronische Bauelement besaß mehrere Ein- und Ausgängen, deren Verhalten auch noch nach dem Fertigungsprozess festgelegt werden konnte. Die Ein- und Ausgänge waren in einer vollverbundenen Schaltmatrix miteinander verknüpft. Mit Hochspannung ließen sich gezielt bestimmte Verbindungen in der Matrix zerstören, um verschiedene ODER-Funktion aus den Eingängen realisieren zu können. Die PROMs konnten daher nur ein einziges Mal programmiert werden.

Mit den Programmable Logic Arrays (PLA), deren Aufbau in Abbildung 2.1 dargestellt ist, wurde dieses Konzept um eine zusätzlich konfigurierbare UND-Matrix erweitert, wodurch sich bereits beliebige logische Funktionen realisieren ließen. Allein die Größe der Matrizen beschränkte die Mächtigkeit der möglichen Funktionen.

Jeder Kreuzungspunkt in der Matrix definiert dabei ein Konfigurationsbit, welches darüber entscheidet, ob die Verbindung leitend oder isolierend geschaltet wird. Die Gesamtheit der Konfigurationsbits ergibt die Konfiguration der programmierbaren Hardware und definiert die Schaltung, die auf der Hardware realisiert ist. Erweitert wurde die PLA-Architektur später durch Speicherglieder wie Flip-Flops, die für eine Rückkopplung in der Schaltung eingesetzt wurden. Auf diese Weise ließen sich bereits Zustandsautomaten realisieren.

Neben Verbesserungen in der Architektur konfigurierbarer Hardware erweiterten auch neue Technologien zur Speicherung der Konfigurationsbits die Möglichkeiten. Auf den 1971 entwickelten EPROMs (Erasable Programmable Read Only Memo-

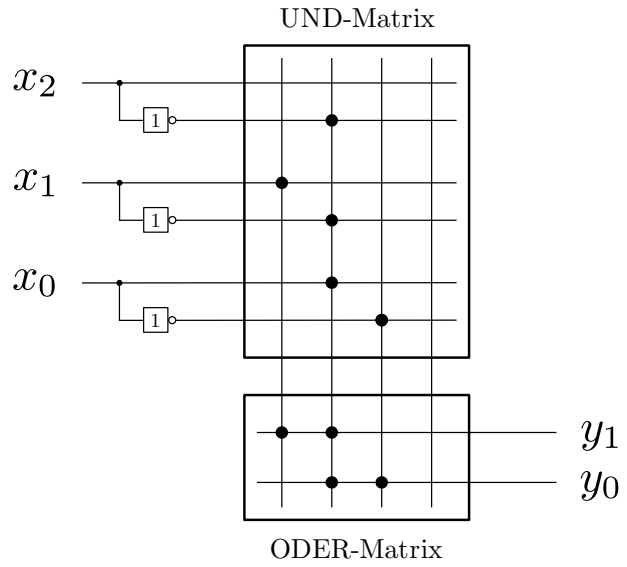


Abbildung 2.1: Programmable Logic Array bestehend aus konfigurierbaren UND- und ODER-Matrizen

ry) konnten Schaltungen nach einer Programmierung mit UV-Licht wieder gelöscht werden und ermöglichten somit das mehrfache Rekonfigurieren. Mit den EEPROMs (Electrically Erasable Programmable Read Only Memory) war es ab 1983 möglich, die Rekonfiguration auch auf elektronischem Wege durchzuführen.

2.2 FPGAs

1985 entwickelte die Firma Xilinx die ersten FPGAs. Statt der bei PLAs üblichen starren Verbindungsmatrix bestehen die FPGAs hauptsächlich aus einer Matrix konfigurierbarer Logikblöcke (Configurable Logic Blocks, CLB). Diese setzen sich, wie Abbildung 2.2 verdeutlicht, aus vier Slices, zwei Tristate Buffer (TBUF) und einer Switch-Matrix zusammen. Die Slices bestehen im Wesentlichen aus zwei Look-Up-Tables (LUT) und zwei Flip-Flops. Die LUTs von Xilinx basieren auf SRAM-Zellen. Zu jeder Kombination der Eingangswerte wird der Wert des Ausgangs in einer SRAM-Zelle gespeichert. Der Vorteil gegenüber der Speicherungsform bei EEPROMs liegt in der Geschwindigkeit, mit der auf den Inhalt der Zellen zugegriffen werden kann. Nachteilig ist jedoch, dass die Daten in den SRAM-Zellen flüchtig sind und somit nach einer Stromunterbrechnung neu konfiguriert werden müssen. Mit den Flip-Flops in den CLBs lassen sich kleinere Datenmengen speichern, die wiederum für Rückkopplungen in den Schaltungen verwendet werden können. Auf die Bedeutung

der TBUFs wird im Kapitel 2.6.1 eingegangen.

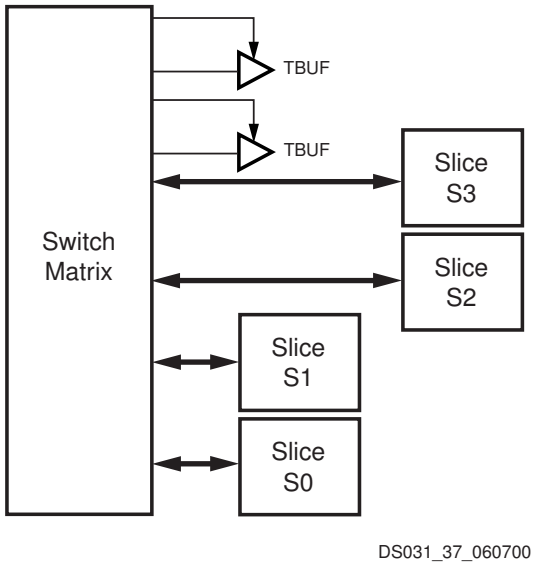


Abbildung 2.2: Aufbau eines CLBs des Virtex-II Pro [Xil07b]

Untereinander verbunden sind die CLBs über die Switch-Matrizen. Sowohl die Belegungen der LUTs als auch die Verdrahtung zwischen den CLBs ist durch den Anwender konfigurierbar und kann durch eine Folge von Bits repräsentiert werden. Diese Folge wird von Xilinx als Bitstream bezeichnet. Über die Programmiereinheit kann ein solcher Bitstream auf den FPGA geschrieben werden, um auf diesem einen Schaltkreis zu realisieren. Auch kann die Programmiereinheit dazu genutzt werden, die vorhandene Konfiguration auf dem FPGA auszulesen und den gewonnenen Bitstream zum PC zu übertragen.

In Verbindung mit dem Xilinx University Program (XUP) wurde eigens das XUP-Board entwickelt, welches zur Durchführung dieser Arbeit eingesetzt wurde. Ausgerüstet ist es mit einem FPGA vom Typ *XC2VP30* der Serie *Virtex-II Pro*. Abbildung 2.3 zeigt den konkreten Aufbau dieses Chips. Anhand dieser Abbildung sollen nun weitere Details des Chips erläutert werden.

Neben der gut zu erkennenden regelmäßigen CLB-Struktur innerhalb des FPGAs befinden sich am Rande des Chips die IOIs (Input/Output Interconnection) und IOBs (Input/Output Block) sowie Elemente zur Taktaufbereitung, sogenannte Digital Clock Manager (DCM). Die IOIs und IOBs stellen die Verbindung zwischen der Schaltung auf dem FPGA und den Pins des Chips (der Außenwelt) her. Die Pins sind wiederum mit unterschiedlichen Komponenten auf dem Board verbunden, darunter beispielsweise mit dem SDRAM, der seriellen Schnittstelle und den LEDs.

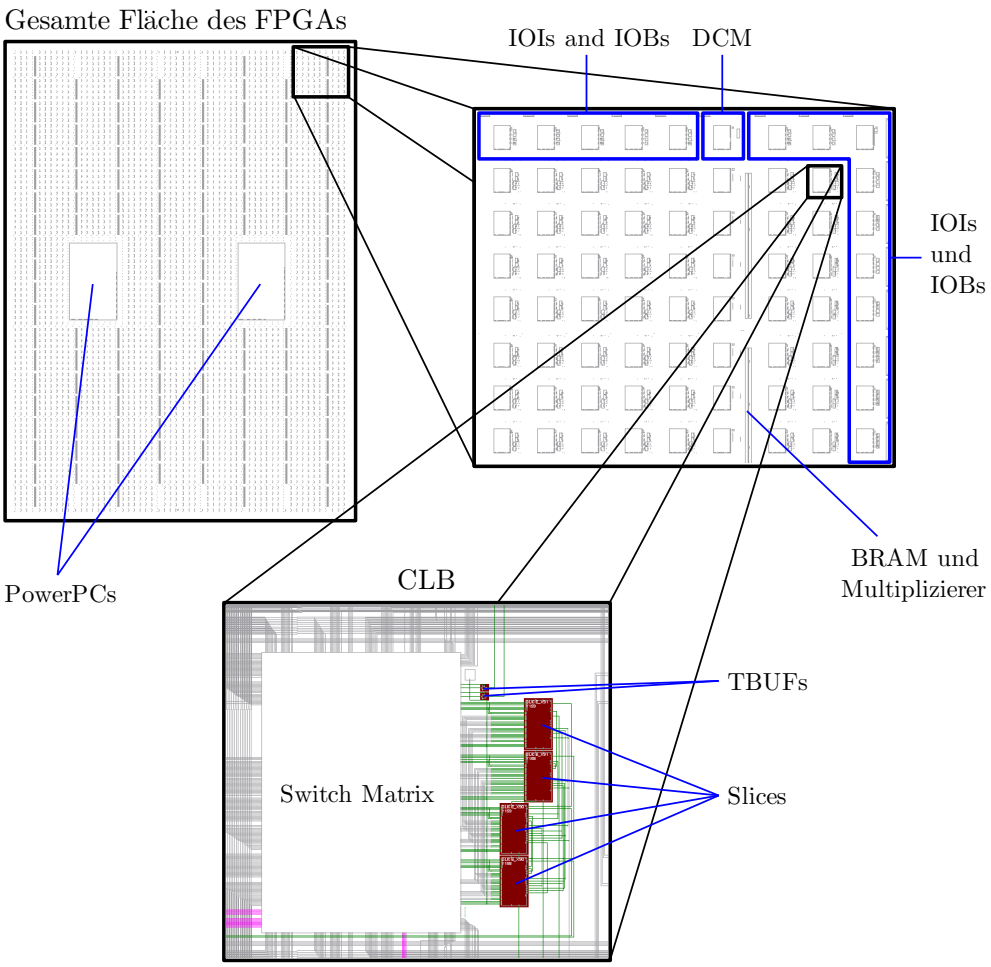


Abbildung 2.3: Zusammensetzung der Xilinx FPGAs aus einer Matrix aus CLBs

Neben den hauptsächlich konfigurierbaren Blöcken auf dem FPGA befinden sich auch noch feste Bestandteile, sogenannte Hard-Cores, auf dem Chip. Zu diesen zählen zum einen die zwei Prozessor-Kerne vom Typ PowerPC und insgesamt acht, auf dem Chip gleichmäßig verteilte Spalten mit Block RAMs (BRAM) und Multiplizierern. Die BRAM-Blöcke eigenen sich zur Speicherung kleinerer Datenmengen, die die Kapazität der vorhandenen Flip-Flops übersteigen würden. Die Verdrahtung der BRAMs erfolgt in konfigurierbaren BRAM-Interconnect-Spalten, die die BRAMs umschließen. Mit den festen und optimierten Multiplizierern lassen sich jeweils zwei 18-Bit-Werte multiplizieren. Um eine äquivalente Logik mittels CLBs zu realisieren, würde eine deutlich größere Fläche benötigt.

2.3 Dynamische partielle Rekonfiguration

Die FPGAs der Virtex-II-Serie zeichnen sich unter anderem dadurch aus, dass sie sowohl die vollständige als auch die partielle Rekonfiguration des Chips unterstützen. Zudem kann die Rekonfiguration dynamisch, d.h. zur Laufzeit des FPGAs, erfolgen. Dies macht es möglich, Teile einer aktiven Schaltung im Betrieb auszutauschen oder zu erweitern. Die Rekonfiguration kann auf verschiedene Arten erfolgen. Unter anderem kann der FPGA mit Hilfe eines angeschlossenen Gerätes über eine externe Schnittstelle programmiert werden. Eine weitere Möglichkeit ist die Nutzung des internen Konfigurationsports (Internal Configuration Access Port (ICAP)), wodurch der Chip die Rekonfiguration selbst durchführen kann.

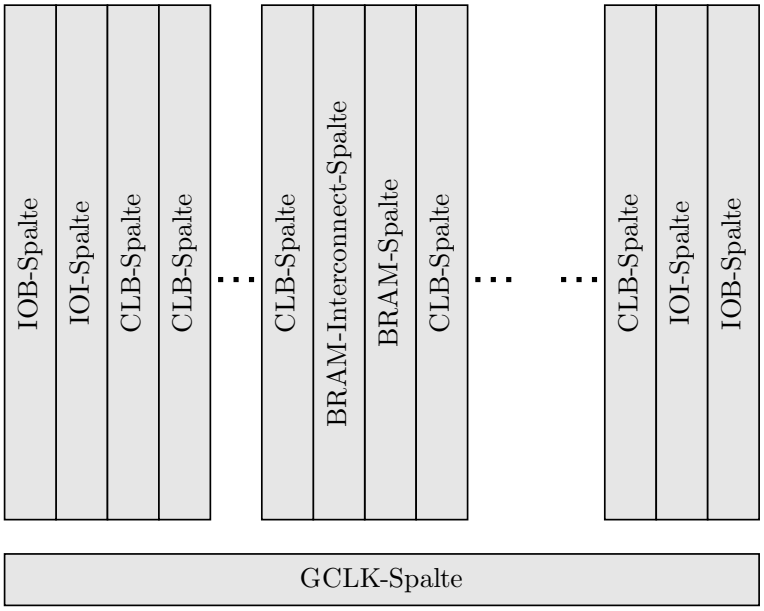


Abbildung 2.4: Chip-Einteilung in Spalten

Für die Rekonfiguration wird der Chip in Spalten unterschiedlicher Typen eingeteilt, wie in Abbildung 2.4 dargestellt. Dabei werden alle IOIs am linken und rechten Rande des Chips jeweils zu einer Spalte zusammengefasst. Ebenso wird mit den IOBs verfahren. Zwischen den beiden Seitenrändern ist der Chip in zahlreiche CLB-Spalten geteilt, wobei sich zwischen diesen auch noch einige BRAM- und BRAM-Interconnect-Spalten befinden. Mit Hilfe der BRAM-Spalten kann auf den Inhalt der BRAMs über die Programmiereinheit zugegriffen werden. Die IOIs und IOBs am oberen und unteren Rande des Chips werden den jeweils auf gleicher Breite liegenden CLB-Spalten zugeordnet. Die GCLK-Spalte stellt eine Ausnahme gegenüber den an-

deren Spalten dar. Sie ist für die grobe Verteilung des Taktes auf dem gesamten Chip zuständig, wird aber für die Rekonfiguration ähnlich wie die CLB-Spalten behandelt.

Diese Spalten werden wiederum in drei Gruppen untergliedert:

Gruppe 1: enthält alle GCLK-, IOB-, IOI- und CLB-Spalten

Gruppe 2: enthält alle BRAM-Spalten

Gruppe 3: enthält alle BRAM-Interconnect-Spalten

In jeder Gruppe werden die Spalten getrennt durchnummeriert. Abbildung 2.5 verdeutlicht die Nummerierungsreihenfolge.

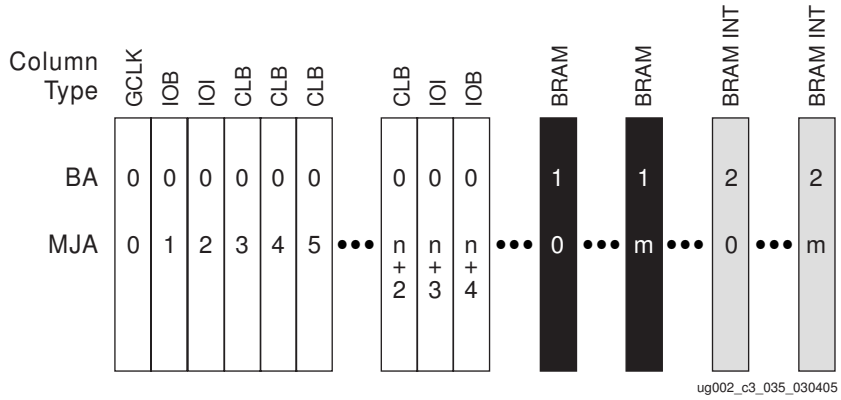


Abbildung 2.5: Nummerierung der Konfigurationsspalten [Xil07a]. n bezeichnet die Anzahl der CLB-Spalten, m die Anzahl der BRAM-Spalten. Über die *Block Address* (BA) und *Major Address* (MJA) werden die Spalten durch die Programmiereinheit identifiziert.

Die kleinste rekonfigurierbare Einheit ist ein Frame. Die Spalten des FPGAs sind in vertikal verlaufende Frames unterteilt, die sich jeweils über die gesamte Höhe des FPGAs erstrecken. Die Anzahl der Frames pro Spalte hängt vom jeweiligen Spaltentyp ab. Ihre Größe ist für alle Spaltentypen gleich. Für den *XC2VP30* beträgt sie 206 Bytes. Abbildung 2.6 zeigt eine Übersicht über die verschiedenen Spaltentypen und deren Frameanzahl auf den unterschiedlichen Chips der Virtex-II-Pro-Serie.

Für die partielle Rekonfiguration können zusammenhängende Bereiche auf dem FPGA in einem kontinuierlichen Datenstrom mit den Inhalten der Frames an die Programmiereinheit übertragen werden. Zuvor wird neben der Anzahl der zu übertragenden Frames auch der Start-Frame übermittelt, der durch eine Spalten-Gruppe, eine Spalten-Nummer und eine Frame-Nummer identifiziert wird. Weitere Informationen zur Kommunikation mit dem ICAP und zum Aufbau der Bitstreams finden sich beispielsweise in der Diplomarbeit von Norbert Abel [Abe05].

Column Type: →	IOB		IOI		CLB		BRAM		BRAM Interconnect		GCLK	
Device: ↓	Columns per Device:	Frames per Column:	Columns per Device:	Frames per Column:	Columns per Device:	Frames per Column:	Columns per Device:	Frames: per Column	Columns per Device:	Frames per Column:	Columns per Device:	Frames per Column:
XC2VP2	2	4	2	22	22	22	4	64	4	22	1	4
XC2VP4	2	4	2	22	22	22	4	64	4	22	1	4
XC2VP7	2	4	2	22	34	22	6	64	6	22	1	4
XC2VP20	2	4	2	22	46	22	8	64	8	22	1	4
XC2VPX20	2	4	2	22	46	22	8	64	8	22	1	4
XC2VP30	2	4	2	22	46	22	8	64	8	22	1	4
XC2VP40	2	4	2	22	58	22	10	64	10	22	1	4
XC2VP50	2	4	2	22	70	22	12	64	12	22	1	4
XC2VP70	2	4	2	22	82	22	14	64	14	22	1	4
XC2VPX70	2	4	2	22	82	22	14	64	14	22	1	4
XC2VP100	2	4	2	22	94	22	16	64	16	22	1	4

Abbildung 2.6: Spaltentypen mit Anzahl der Spalten und Frames auf den Chips der Virtex-II-Pro-Serie[Xil07a]

2.4 Hardware-Plugins und Hardware-Tasks

Gegenüber der klassischen vollständigen Programmierung eines FPGAs eröffnet die dynamische partielle Rekonfiguration völlig neuen Raum für Einsatzgebiete. Indem nur Teile der rekonfigurierbaren Fläche ersetzt werden, verkürzt sich nicht nur die Rekonfigurationszeit deutlich gegenüber einer vollständigen Programmierung des Chips, sondern ermöglicht es auch, Schaltungen an gezielten Bereichen zu erweitern oder zu verändern, wobei der Rest der Schaltung davon unbeeinflusst bleibt.

Ein solches Einsatzgebiet stellen unter anderem die *Hardware-Plugins* dar. Ähnlich wie Software-Plugins sind sie über eine definierte Schnittstelle mit der primären Schaltung bzw. dem Hauptprogramm verbunden und erweitern diese in ihrer Funktionalität. Für die Umsetzung auf einem FPGA werden zunächst Teile der programmierbaren Fläche reserviert, in die später unterschiedlichste Plugins zur Laufzeit nachgeladen werden. Die restliche Fläche, die nicht von der Rekonfiguration betroffen ist, wird als System-Area bezeichnet, da hier im Allgemeinen Komponenten untergebracht werden, die essenziell für den Betrieb der Anwendung sind.

Gegenüber dem Ansatz der Hardware-Plugins, deren Rekonfiguration eher selten stattfindet, richtet sich der Blick dieser Arbeit auf Hardware-Äquivalente zu

Software-Threads, von denen erwartet wird, dass beliebig viele gleichzeitig und relativ unabhängig[1] voneinander ausgeführt werden können. Um eine solche Nebenläufigkeit auf einer beschränkten FPGA-Fläche zu ermöglichen, wird ein Scheduler benötigt, der wie bei einer CPU den Programmteilen Zeitscheiben zur Ausführung zuordnet. In diesem Zusammenhang spricht man nicht mehr von Hardware-Plugins, sondern von Hardware-Tasks.

2.5 Modelle zur Partitionierung

Im Folgenden wird nur auf Hardware-Tasks eingegangen, wobei die erörterten Modelle ebenso auf Hardware-Plugins übertragbar sind.

2.5.1 Anordnung und Aufgaben der System-Area

Zu Beginn der Partitionierung muss der Chip in System- und Task-Areas eingeteilt werden. Während die Task-Area den dynamisch zu konfigurierenden Bereich kennzeichnet, definiert die System-Area den Teil der Schaltung, der stets verfügbar sein soll und von der dynamischen Rekonfiguration ausgeschlossen wird. Die Einteilung hängt im Wesentlichen von der Belegung der Pins am äußeren Rand des Chips und von etwaigen Chip-Besonderheiten ab, wie beispielsweise eingebetteten Prozessor-Kernen.

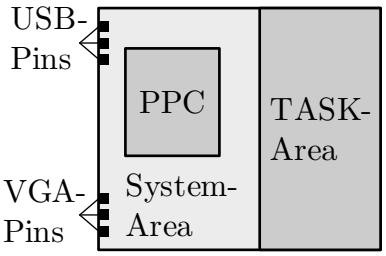


Abbildung 2.7: Aufteilung in System- und Task-Area

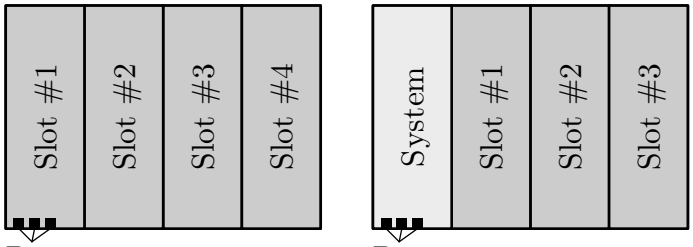
Zur Verdeutlichung soll Abbildung 2.7 dienen. In diesem Beispiel befindet sich ein PowerPC (PPC) auf der linken Hälfte des FPGAs. Ebenfalls links befinden sich die Pins des VGA-Controllers und der USB-Schnittstelle. Die Task-Area auf der linken Seite des Chips unterzubringen, macht aus vielerlei Gründen für dieses Beispiel

[1] *relativ unabhängig* bezieht sich darauf, dass Threads eigenständige Programmteile sind, deren Ausführung unabhängig vom Hauptprogramm und anderen Threads erfolgt, wenn man von der Kommunikation untereinander absieht

wenig Sinn. Zum einen nimmt der PPC bereits einen großen Bereich der Fläche in Anspruch, wodurch die Task-Area in die Breite gezogen werden müsste, um größere Tasks realisieren zu können. Zum anderen liegen dort alle Schnittstellen zur Außenwelt. Die Tasks müssten direkt mit der Außenwelt kommunizieren und evtl. aufwendige, hardwarenahe Protokolle implementieren, die sich negativ in der Task-Größe niederschlagen. Um die Tasks so klein wie möglich halten zu können, bietet sich die Möglichkeit, Schaltungslogik, die von mehreren Tasks genutzt wird, in der System-Area unterzubringen. Das System übernimmt dabei die Rolle einer Kommunikationsschicht. Es implementiert die eigentliche Kommunikation mit den externen Geräten und stellt den Tasks eine vereinfachte Schnittstelle zur Ansteuerung dieser zur Verfügung. Ebenfalls denkbar wäre, die System-Area auch als Hardware-Abstraktionsschicht zu nutzen. Beispielsweise könnten verschiedene Externspeicher durch die System-Area so verwaltet werden, dass den Tasks ein einheitliches, kompaktes Protokoll für den Zugriff zur Verfügung gestellt werden kann. Selbst wenn die Speicher-Hardware ausgetauscht würde, wären die Tasks von der Änderung nicht betroffen. Lediglich die System-Area müsste neu angepasst werden.

2.5.2 Chip-Aufteilung ohne System-Area

Wird der Chip von außen rekonfiguriert, kann auf die System-Area theoretisch verzichtet und die gesamte Fläche des Chips für die dynamische Rekonfiguration verwendet werden. Dazu müssen die Tasks allerdings selbst die Kommunikation mit der Außenwelt übernehmen. Ist die Task-Area, wie in Abbildung 2.8a dargestellt, in mehrere Task-Slots unterteilt, können jeweils nur die Tasks die Verbindung zur Umwelt herstellen, deren Rekonfigurationsbereich die Pins für die Kommunikation einschließt. Bezogen auf das Beispiel sind nur Tasks im Slot #1 dazu in der Lage.



(a) ohne System-Area (b) mit System-Area

Abbildung 2.8: Aufteilung in feste Rekonfigurationsbereiche

Da die restlichen Tasks in den Slots #2 bis #4 ohne die Möglichkeit, Eingabedaten zu erhalten und Berechnungsergebnisse in irgendeiner Form zurück zu liefern, keine sinnvolle Tätigkeit ausführen können, ist der Task in Slot #1 dafür verantwortlich, ihnen eine entsprechende Kommunikationsschnittstelle zur Verfügung zu stellen. Auch dieser Einsatz von Tasks, wichtige Kommunikationsaufgaben zu übernehmen, kann eine sinnvolle Lösung sein, wenn die Logiken zur Ansteuerung der externen Geräte so komplex sind, dass sie nicht gleichzeitig in einem Task-Slot bzw. einem System-Bereich untergebracht werden können. In dem Fall könnten sich mehrere Tasks im Slot #1 abwechseln und zeitscheibenweise unterschiedliche Schnittstellen zur Verfügung stellen. Die Konsequenz dieser Abhängigkeit von bestimmten Pins ist, dass jeder Task nur in gewisse Slots geladen werden kann.

2.5.3 Aufteilung der Task-Area

Im Folgenden sollen verschiedene Partitionierungsmöglichkeiten rechteckiger Task-Areas beleuchtet werden. Welche Varianten für die Aufteilung der Tasks bestehen, hängt von der Form und der Größe der Tasks, aber auch von den Einschränkungen der Rekonfigurationshardware ab.

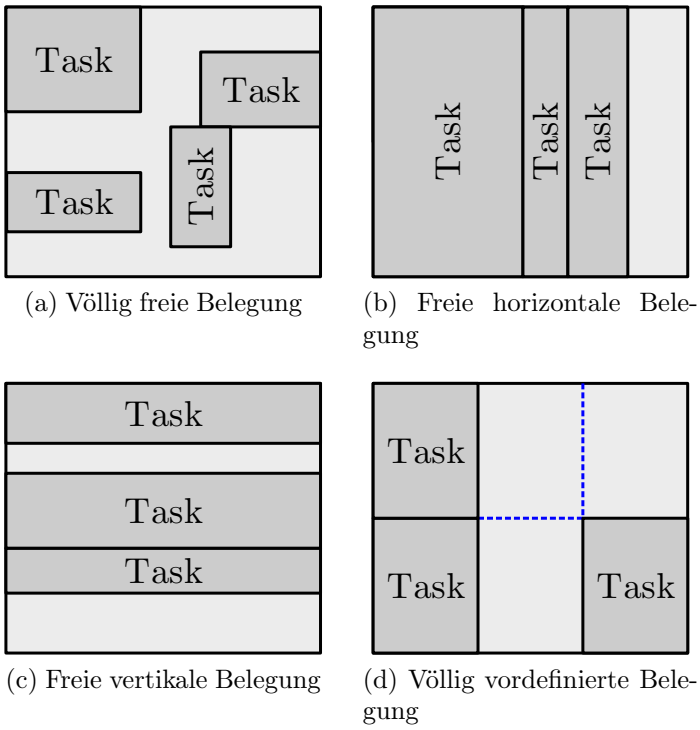
Beliebig geformte Tasks auf einem zweidimensionalen Feld zu platzieren, stellt große Ansprüche an die Platzierungsalgorithmen. Diesem Problem widmet sich unter anderem [PEW+02]. Um die Fläche optimal auszunutzen, ist es entscheidend zu wissen, welche Tasks zu welcher Zeit auf die Task-Area gebracht werden sollen, um daraus eine möglichst gute Anordnung der Tasks berechnen zu können. Je komplizierter die Formen und je größer die Anzahl der Tasks desto schwieriger wird das Problem, in angemessener Zeit eine optimale Lösung zu finden. Aus diesem Grund sind auch heuristische Verfahren von Interesse. Auch wenn diese nicht zwingend eine optimale Lösung liefern, skalieren sie gegenüber exakten Verfahren deutlich besser bezüglich der Anzahl der Tasks und die Berechnungszeit kann erheblich verkürzt werden.

Betrachtet man nur rechteckige Tasks, so lassen sich die unterschiedlichen Modelle gut anhand ihrer Freiheitsgrade bezüglich Verteilung und Ausdehnung der Tasks charakterisieren. Dabei stellt sich für die Tasks die Frage, ob für die Platzierung x- und y-Koordinate und für die Ausdehnung Task-Breite b und Task-Höhe h frei wählbar sind.

Modelle mit fester Task-Breite aber variabler Positionierung in x-Richtung machen hinsichtlich der Platzausnutzung wenig Sinn. Optimale Algorithmen würden die Tasks stets so eng wie möglich anordnen, was einer Einteilung in feste Task-Slots

gleich kommt. Die andere Variante mit einer vorgegebenen Position aber einer flexiblen Task-Breite ermöglicht es zwar, die Rekonfigurationszeit für kleine Tasks zu verkürzen, bringt jedoch bezüglich Platzausnutzung ebenfalls keine Vorteile und wird deshalb nicht weiter betrachtet. Gleiches gilt für die Positionierung in y-Richtung und Höhe h.

Demzufolge verbleiben 4 wesentliche Modelle zur Partitionierung der Task-Area:



(a) Völlig freie Belegung

(b) Freie horizontale Belegung

(c) Freie vertikale Belegung

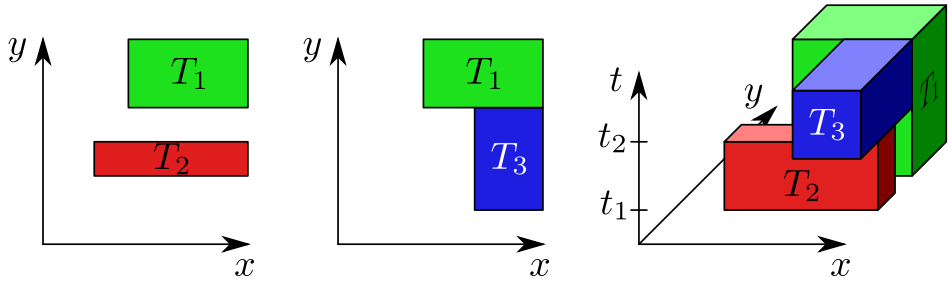
(d) Völlig vordefinierte Belegung

Abbildung 2.9: Partitionierung des Chips

(a) **Völlig freie Belegung** (x, y, b und h sind frei wählbar)

Diese Variante bietet die meisten Freiheiten für die Aufteilung. Größe und Platzierung der Tasks können beliebig gewählt werden. Indem die Tasks auf ihre minimal benötigte Fläche reduziert und so eng wie möglich gepackt werden, lässt sich die Anzahl parallel ausführbarer Tasks erhöhen und damit die Performance des Systems steigern. Problematisch hingegen ist die zunehmende Fragmentierung, je mehr Rekonfigurationen stattgefunden haben. Intelligente Platzierungsalgorithmen und gezielte Defragmentierungen können dem entgegenwirken und Platz für die Einlagerung weiterer Tasks schaffen. Die Suche nach einer optimalen Anordnung entspricht dabei dem *2D Bin Packing Problem.*[BKS00]

Kenntnisse darüber, welcher Task wie lange und in welchem möglichen Zeitraum auf der Task-Area untergebracht werden soll, sind für die Platzierungsalgorithmen von großem Nutzen. Die Algorithmen können zeitlich vorausschauend die Tasks so anordnen, dass sie bei der Platzierung späterer Tasks keine Hindernisse darstellen. Mit Einbeziehung der zeitlichen Komponente kann das Platzierungsproblem als *3D Bin Packing Problem* betrachtet werden (siehe Abbildung 2.10). Strategien dazu wurden in [BKS00] erörtert.



(a) Belegung zum Zeitpunkt t_1

(b) Belegung zum Zeitpunkt t_2

(c) Belegung im 3D-Raum

Abbildung 2.10: Beispiel einer Task-Platzierung unter Einbeziehung der Zeitkomponente. Die Abbildung skizziert die Problematik der Task-Anordnung im dreidimensionalen Raum, der sich aus der 2D-Fläche des FPGAs und der zeitlichen Komponente ergibt. T_1, T_2 und T_3 markieren 3 verschiedene Tasks, die zu den Zeitpunkten t_1 und t_2 auf dem FPGA platziert werden.

Ein weiteres, schwerwiegendes Problem stellt die Verdrahtung der Tasks dar. Während das Taktsignal über *Global Clock Networks*[2] auf dem gesamten Chip verbreitet werden kann, die zudem von der Rekonfiguration der Task-Area nicht beeinflusst werden, bleibt die Kommunikation mit der Außenwelt und zu anderen Tasks kritisch. Die entsprechenden Schnittstellen müssen überall auf dem Chip verfügbar sein. Um diese Verbindungen herzustellen, können beispielsweise die nicht belegten Bereiche der Task-Area für die Verdrahtung genutzt werden, was jedoch bei der Platzierung der Tasks berücksichtigt werden muss. Eine andere Möglichkeit wäre, die gesamte Task-Area mit einem Kommunikationsnetz zu überziehen. Damit durch das Einspielen neuer Tasks keine Lücken in dem Netz entstehen, muss sich das Netzwerk auch durch die Tasks selbst ziehen – was zur Synthesezeit zu beachten ist.

(b) **Freie horizontale Belegung** (nur x und b sind frei wählbar)

[2]Details zu *Global Clock Networks* der Virtex-II Pro-Architektur lassen sich in [Xil07a] finden. Simon Steinegger erstellte in [Ste04] ein solches Clock-Netz mittels JBits.

Bei dieser Variante ist die freie Platzierung und Ausdehnung der Tasks nur in einer Dimension möglich. Damit reduziert sich auch das Platzierungsproblem. Während die y-Position und die Höhe der Tasks vorgegeben sind, kann die Positionierung in x-Richtung sowie die Breite der Tasks frei gewählt werden. Wie auch schon in Variante (a) besteht damit die Möglichkeit, die Tasks bezüglich ihrer Größe zu optimieren. Die Schnittstelle zur Kommunikation werden in der Regel durch einen horizontale verlaufenden Bus realisiert, der sich über die gesamte Breite der Task-Area erstreckt. Eine Prototyp-Implementierung dieser Variante fand in [WNP04] statt.

(c) **Freie vertikale Belegung** (nur y und h sind frei wählbar)

Diese Variante unterscheidet sich von der in (b) nur durch die Vertauschung der Achsen und muss deshalb nicht weiter erläutert werden. Beachtet werden muss jedoch, dass bei der Rekonfiguration von FPGAs der Virtex-I und Virtex-II-Serie stets ganze vertikale Frames beschrieben werden. Um einen Task auf den Chip zu bringen, müssen also die eventuell darüber und darunter liegenden Bereiche/Tasks vorher über die Programmiereinheit ausgelesen und in den neuen Konfigurationsbitstream eingearbeitet werden. Das Beschreiben der eigentlich nicht zu rekonfigurierenden Bereiche kostet zudem Zeit, die bei einer horizontalen Task-Belegung nicht anfällt.

(d) **Völlig vordefinierte Belegung** (keine Variable ist frei wählbar)

Bei dieser Variante besitzen die Tasks eine feste Breite und Höhe. Der Chip kann somit in feste Rekonfigurationsbereiche, genannt Task-Slots, eingeteilt werden. Das Platzierungsproblem reduziert sich nur noch auf die Suche nach einem geeigneten Task-Slot. Durch die festen Grenzen der Slots, lassen sich wohldefinierte Schnittstellen ohne größere Probleme realisieren. Nachteilig ist jedoch, dass die Tasks durch die feste Slot-Größe in ihrem Ressourcenverbrauch beschränkt sind. Die maximale Größe der zu implementierenden Tasks muss daher frühzeitig im Entwurf der Task-Area berücksichtigt werden. Gleichzeitig ist darauf zu achten, dass die Slots keine übertriebenen Dimensionen annehmen, da sonst während der Ausführung der Tasks wertvolle Ressourcen ungenutzt blieben.

Implementiert wurde diese Variante beispielsweise in [WP03].

2.5.4 Interne Rekonfigurationslogik

Bei der internen Rekonfiguration muss darauf geachtet werden, dass die für die Rekonfiguration verantwortliche Logik in keinem Fall zur Laufzeit geändert werden darf. Die Programmierung des FPGAs erfolgt frameweise. Noch während Daten an die Programmiereinheit gesendet werden, wird der Chips bereits Frame für Frame neu konfiguriert. Eine kleine Veränderung des Verdrahtungsweges zum ICAP hätte dabei fatale Folgen. Sobald die Programmiereinheit den ersten Frame mit der Verdrahtungsabweichung konfiguriert, wäre der Pfad zum ICAP unterbrochen, der Konfigurationsbitstream könnte nicht weiter übertragen werden und die Rekonfiguration bliebe damit stehen. In diesem Zustand wäre das ICAP von der Rekonfigurationslogik solange abgeschnitten, bis der Chip von außen neu programmiert wird.

Das Problem kann entweder umgangen werden, indem die zuständige Logik im System-Bereich untergebracht wird, der von der dynamischen Rekonfiguration niemals betroffen ist, oder indem die Logik bei einer Rekonfiguration exakt an die gleiche Stelle auf dem Chip geschrieben wird. In dem Fall garantiert Xilinx, dass laufende Signale auf diesen Leitungen nicht beeinflusst werden. Somit ist auch mit Hilfe des internen Konfigurationsports das Programmieren aller Chip-Bereiche möglich.

2.6 Modelle der Task-Kommunikation

Würden Tasks, abgesehen vom Taktsignal, keinerlei Verbindungen zur Außenwelt oder zu anderen Chip-Komponenten aufweisen, so wären die von ihnen berechneten Ergebnisse wertlos. Es würde keinen Unterschied machen, ob ein Task tatsächlich auf dem Chip läuft oder nicht. Aus diesem Grund müssen Tasks mit einer Kommunikationsschnittstelle ausgestattet werden, die mindestens die Möglichkeit zur Ausgabe von Berechnungsergebnissen zulässt. Prinzipiell müssen Tasks nicht unbedingt Eingabedaten erwarten. Tasks ohne Eingabendaten werden stets das zur Synthesezeit festgelegte Verhalten mit einer festen, deterministischen Ausgabe ausführen. Auch diese Form von Tasks hat ihre Berechtigung. Beispiele hierfür wären deterministische Zufallsgeneratoren und Zähler. Die Ausgabe kann wiederum als Eingabe für andere Tasks eingesetzt werden. In diesem Sinne müssen in jedem Fall auch Leitungen für Eingabedaten in der Kommunikationsschnittstelle vorgesehen werden. Auf der anderen Seite muss rechenaufwendigen Tasks ein Weg zur Speicherung des eigenen Zustandes bereitgestellt werden, um durch den Task-Scheduler unterbrochen und später fortgesetzt werden zu können.

Als nächstes sollen physikalische Verbindungsmöglichkeiten erörtert werden, die

die Grundlage für die Kommunikation zwischen mehreren Rekonfigurationsbereichen bereiten. Darauf aufbauend werden verschiedene Modelle der Task-Kommunikation vorgestellt.

2.6.1 Physikalische Verbindungsmöglichkeiten

Der allgemeine Arbeitsablauf bei der Erstellung eines Designs ist heutzutage vollständig automatisiert. Professionelle Entwicklungsumgebungen übernehmen nach der Synthese auch die Platzierung und die Verdrahtung der Komponenten. Um bei der dynamischen partiellen Rekonfiguration wohldefinierte Schnittstellen zwischen den Rekonfigurationsbereichen zu schaffen, müssen die Signalleitungen der Schnittstelle fest auf dem FPGA platziert werden. Hierbei kommen Busmacros als spezielle Formen von Hardmacros zum Einsatz. *Hardmacros* realisieren logische Funktionen auf der Schaltkreisebene. Die Erstellung von Hardmacros erfordert eine genaue Kenntnis darüber, welche Logikzellen (LUTs, TBUFs, ...) sich wo auf der Zielarchitektur befinden und welche Verbindungsmöglichkeiten zwischen ihnen bestehen. Aus diesem Grund werden sie in einem herstellerspezifischen Format abgelegt und sind nur auf der speziellen Zielarchitektur einsetzbar. Ein Hardmacro definiert, aus welchen einfachen Logikzellen die Funktion realisiert ist und welche Verbindungswege zwischen den Zellen genutzt werden. Weiterhin können die Pins der verwendeten Logikzellen mit Namen versehen werden. Diese bilden die Ein- und Ausgänge des Macros und werden später in einer VHDL-Entity zusammengefasst. Das Macro kann dann als *Black Box*[3]-Komponente in VHDL eingesetzt werden. Da die FPGAs im Allgemeinen aus einer homogenen CLB-Matrix bestehen, lassen sich Hardmacros bis auf wenige Einschränkungen beliebig auf dem Chip platzieren.

Busmacros sind Hardmacros, deren Aufgabe nicht in der Realisierung einer komplexen logischen Funktion besteht, sondern in der Fixierung der Schnittstellensignale, die über die Grenzen von Task-Slots führen. In der Implementierung wird zwischen unidirektionalen und bidirektionalen Busmacros unterschieden.

Bidirektionale Busmacros auf Basis von TBUFs

Um bidirektionale Busmacros zu schaffen, werden TBUFs (siehe Abbildung 2.11) eingesetzt, welche drei Zustände, statt den gewöhnlichen zwei (0 und 1), annehmen können.

[3]Verhaltensbeschreibungen von Hardmacros sind in VHDL nicht unbedingt erforderlich. Ist das Verhalten unbekannt, so bezeichnet man die Komponente als *Black Box*. Zur VHDL-Simulation sind Black Boxes allerdings ungeeignet.

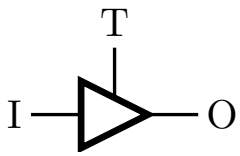


Abbildung 2.11: TBUF

Der dritte Zustand (Z) ist der hochohmige Zustand, der es erlaubt, die Ausgänge mehrerer TBUFs zusammenzuschließen, ohne dass es dabei zu einem Kurzschluss kommt. Dieser Zustand tritt ein, wenn am Eingang T eine 0 anliegt. Vereinfacht kann der TBUF daher als Schalter angesehen werden, der genau dann das Eingangssignal I auf den Ausgang O legt, wenn T auf 1 gezogen wurde.

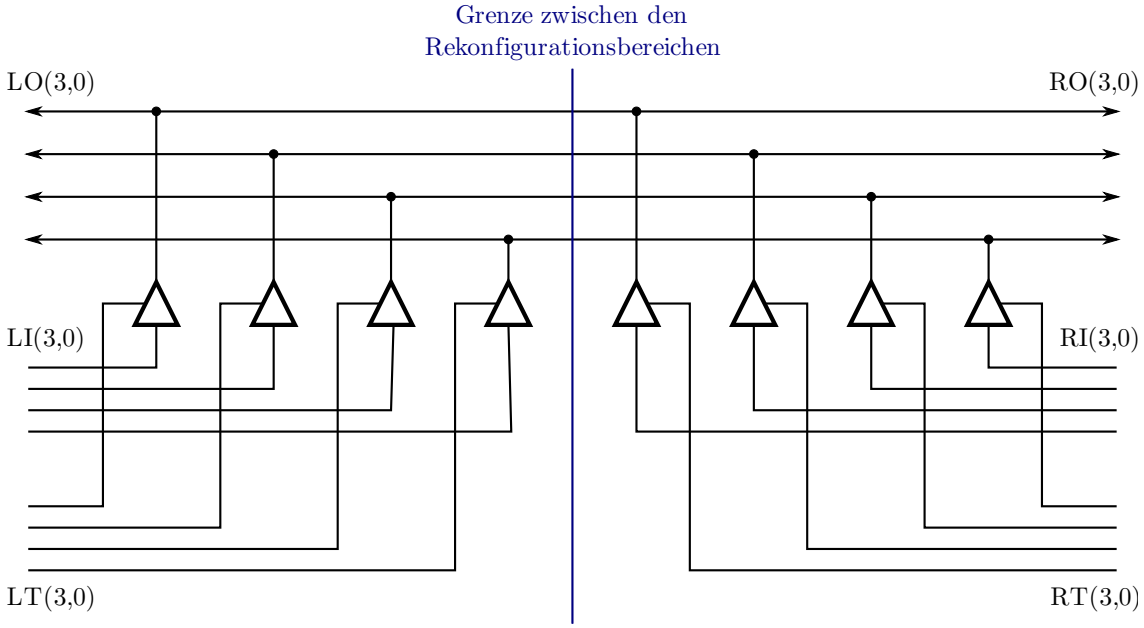


Abbildung 2.12: Busmacro mit 8 TBUFs [Xilb]

Abbildung 2.12 zeigt ein 4 bit-breites, bidirektionales Busmacro, dass sich über 2 Rekonfigurationsbereiche erstreckt. Auf beiden Seiten befinden sich jeweils 4 TBUFs, die mit den Busleitungen verbunden sind. Ausschließlich die Busleitungen passieren die Grenze zwischen den Rekonfigurationsbereichen. Daten können sowohl von links nach rechts als auch von rechts nach links übertragen werden, da beide Seiten in der Lage sind, Signale auf den Bus zu legen und von diesem abzugreifen.

Völlig unkritisch ist der Einsatz von TBUFs allerdings nicht. Sind zwei zusammengeschlossene TBUFs gleichzeitig aktiv und legt einer eine 0 und der andere eine

1 auf den Ausgang, so kommt es zu einem leichten Kurzschluss. Laut [Xil97][4] fließt dabei ein Strom von 6 mA, der aber kurzzeitig vertretbar ist. Hält der Zustand jedoch über tausende Stunden an, so kann dies zur Hardware-Schädigung[5] führen. Aus diesem Grund sollten Kurzschlüsse zwischen TBUFs vermieden oder zumindest minimiert werden, indem z.B. weitere Busleitungen zur Abstimmung der Senderichtung herangezogen werden.

Unidirektionale Busmacros auf Basis von Slices

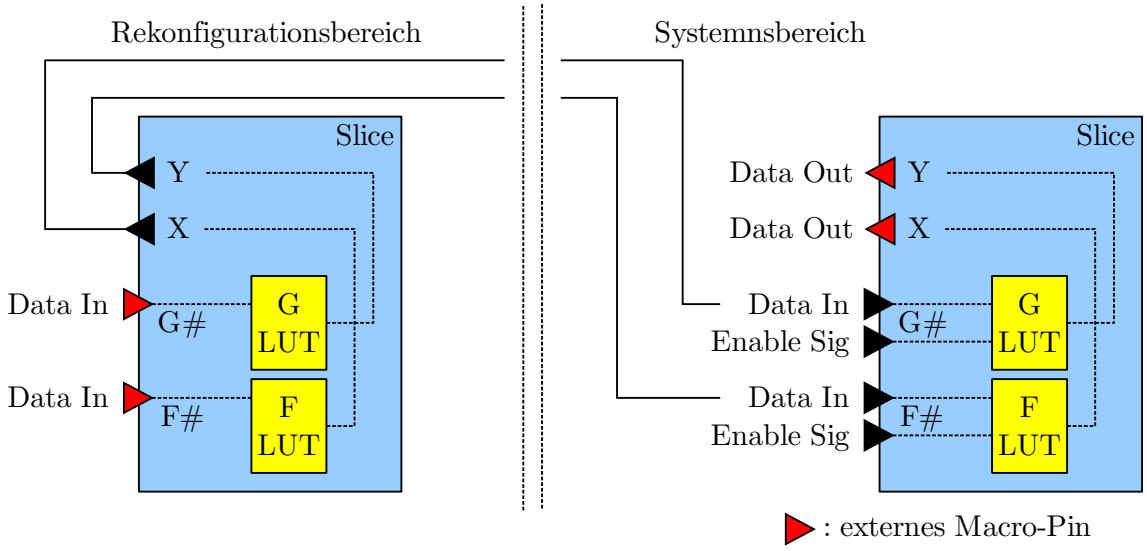


Abbildung 2.13: Busmacro über 2 Slots mit Slices [CZH+07]

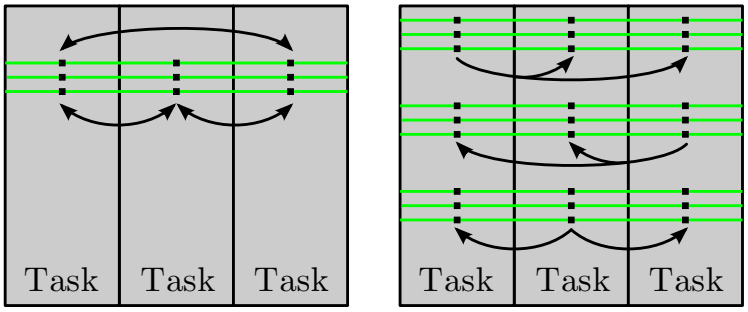
Soll die Kommunikation nur in einer Richtung stattfinden, so kann man auf den Einsatz von TBUFs verzichten und stattdessen Slices verwenden. Die Slices dienen dabei hauptsächlich der Fixierung der Ein- und Ausgabe-Ports des Macros. Abbildung 2.13 zeigt ein solches Busmacro mit einer Busbreite von zwei Bit. Ein Slice kann aufgrund seiner zwei LUTs mit zwei Busleitungen verbunden werden. Ein CLB eines Virtex-II-FPGAs könnte damit theoretisch acht Busleitungen anbinden. Nebenbei können die LUTs auch eine AND-Funktion realisieren, um über ein zusätzliches Enable-Signal steuern zu können, ob die eingehenden Signale zu der auswertenden Logik durchgestellt oder doch unterdrückt werden sollen. Eine Unterdrückung ist vor allem während und kurz nach der Rekonfiguration eines Tasks sinnvoll, denn

[4] [Xil97] bezieht sich auf die bereits älteren XC3000- und XC4000-Serien von Xilinx. Inwiefern die Aussagen auf aktuelle Serien (Virtex, Virtex-II) noch zutreffen, ist unklar.

[5] Der dauerhafte hohe Stromfluss führt zur Ionenabwanderung im Leiter (Elektromigration), wodurch die Zuverlässigkeit des Leiters abnimmt.

durch den Task-Wechsel werden die Flip-Flops in dem Bereich nicht zurückgesetzt. Solange der Task kein Reset-Signal bekommen hat, ist sein Zustand daher nicht vorhersagbar und es wäre möglich, dass der Task fälschlicherweise Signale auf den Bus legt. Wie auch in der Abbildung dargestellt, kann ein mit den LUTs auf der Empfängerseite verbundenes Enable-Signal das Problem ohne zusätzliche Ressourcen beheben.[CZH+07]

Prinzipiell können beide Macrotypen zur Realisierung einer komplexen Kommunikationsinfrastruktur eingesetzt werden. Dabei gilt es zwischen Datendurchsatz und Ressourcenverbrauch abzuwägen. Bidirektionale Busmacros haben den Nachteil, dass sich die Tasks den gemeinsamen Bus teilen müssen und damit die Datenrate eines einzelnen Tasks mit zunehmender Anzahl von Tasks sinkt.



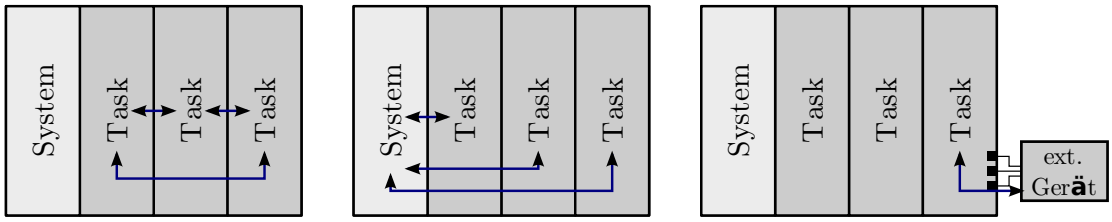
(a) bidirektionaler Bus (b) unidirektionale Busse

Abbildung 2.14: Ressourcenverbrauch bi- und unidirektionaler Busse

Zur Verdeutlichung soll Abbildung 2.14 dienen. 2.14a zeigt, dass bereits ein bidirektionaler Bus für die Verständigung der drei Tasks ausreicht. Um die gleichen Kommunikationsmöglichkeiten mit unidirektionalen Bussen zu erreichen, wird für jeden Task ein eigener Bus benötigt, vgl. Abbildung 2.14b. Auf der anderen Seite müssen Tasks bei der unidirektionalen Variante nicht erst auf das Freiwerden des Busses warten, sondern können jederzeit Daten versenden, sofern die Gegenseite zur Annahme bereit ist.

2.6.2 Kommunikationsmodelle für Tasks

Im Folgenden sollen Modelle der Task-Task-, Task-System- und Task-Umwelt-Kommunikation näher betrachtet werden. Abbildung 2.15 zeigt die Modelle grafisch.



(a) Task-Task-Kommunikation (b) Task-System-Kommunikation (c) Kommunikation mit der Außenwelt

Abbildung 2.15: Kommunikationsmodelle für Tasks

(a) **Task-Task-Kommunikation**

Im Vordergrund der Arbeit steht die Task-Task-Kommunikation, oft auch Inter-Task-Kommunikation bezeichnet. Dabei gilt es zwei grundlegende Formen zu unterscheiden: die direkte und die indirekte Kommunikation. Erfolgt der Datenaustausch über Speicher, in die Daten durch die Tasks abgelegt werden, um von anderen Tasks zu späteren Zeitpunkten ausgelesen zu werden, so spricht man von indirekter Kommunikation. Dagegen ist die Form der direkten Verständigung dadurch gekennzeichnet, dass zwischen den Tasks eine direkte Verbindung besteht, über die die Signale vom Sender ohne Zwischenspeicherung zum Empfänger gelangen.

1. Direkte Inter-Task-Kommunikation

   Zwei Implementierungsbeispiele direkter Task-Task-Verbindungen, die alle möglichen Kommunikationswege unter den Tasks abdecken, wurden mit den uni- und bidirektionalen Busstrukturen bereits in dem Kapitel 2.6.1 bezüglich ihrer Verbindungswege und ihres Ressourcenverbrauchs vorgestellt und sollen nun in Bezug auf die Art und Weise der Kommunikation etwas genauer betrachtet werden.

   Bei unidirektionaler Infrastruktur kann das Protokoll zwischen Sender und Empfänger relativ einfach gehalten werden. Sind die Empfänger-Tasks zu jeder Zeit empfangsbereit, so kann auf eine Quittierung des Erhalts der Daten verzichtet werden und die Kommunikation verläuft nur in einer Richtung, sodass tatsächlich nur ein Bus genutzt wird. Anderenfalls ist eine Synchronisation zwischen den Tasks erforderlich, um untereinander abzustimmen, wann beide Tasks für die Übertragung bereit sind. Für diese Absprache müssen in beiden Richtungen Daten gesendet werden. Abbildung 2.16 zeigt hierzu ein Beispiel, in dem mehr als 2 Tasks an einem Bus hängen. Deshalb müssen

ein paar der Busleitungen für die Übertragung der Nummer des gewünschten Empfängers vorgesehen werden.

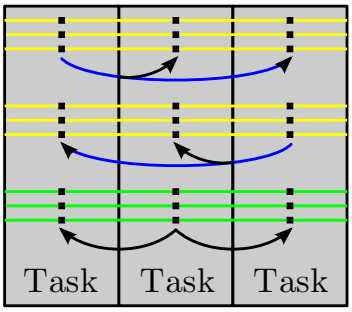
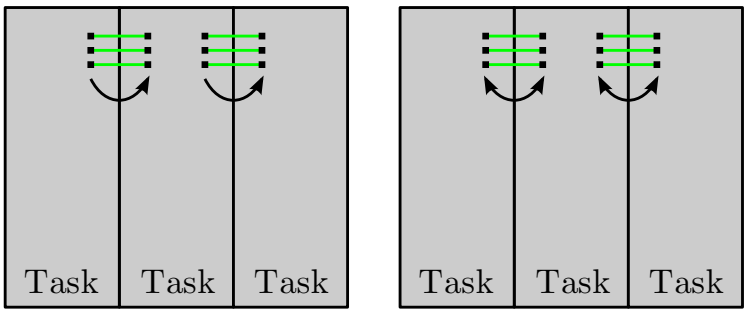


Abbildung 2.16: Synchronisierte Kommunikation auf unidirektionalem Bus

Völlig anders erfolgt demgegenüber der Zugriff auf einen bidirektionalen Bus. Um Hardware-Schäden und ein Kommunikationschaos zu verhindern, müssen die Zugriffe auf den gemeinsamen Bus kontrolliert werden. Diese Aufgabe wird vom Bus-Arbiter übernommen. Über spezielle Bus-Signale können Tasks Sendeanfragen an den Arbiter stellen und bekommen bei Gelegenheit das Senderecht. Der Arbiter ist ebenfalls für eine faire Vergabe der Senderechte zuständig, um auszuschließen, dass ein Task den Bus auf unbestimmte Zeit blockiert. Auch hier müssen weitere Busleitungen für die Empfängernummer reserviert werden. Eine Synchronisation kann wie beim On-Chip Peripheral Bus (OPB) von IBM durch weitere Busleitungen realisiert werden, die vom Empfänger-Task für die Quittierung von Nachrichten oder für Antworten auf Anfragen genutzt werden.



(a) Pipeline (b) nächster Nachbar

Abbildung 2.17: weitere Kommunikationsinfrastrukturen

Eine nicht so mächtige, aber dennoch häufig eingesetzte Kommunikationsinfrastruktur stellen Pipelines dar, siehe Abbildung 2.17a. Jeweils zwei benach-

barte Tasks sind über unidirektionale Busmacros verbunden. Die Übertragung erfolgt über die gesamte Task-Reihe nur in einer Richtung. Das Ziel der Struktur liegt in einer möglichst schnellen Verarbeitung eines Datenstroms. Jeder Task übernimmt dabei einen Teil der Verarbeitung und reicht seine Ergebnisse an den nächsten Task weiter. Große Bedeutung hat dieses Prinzip vor allem in der Signalverarbeitung, beispielsweise um Audio- oder Videodatenströme in Echtzeit zu filtern oder zu manipulieren.

Eine ebenfalls populäre Form der Kommunikation ist durch die Beschränkung ihrer Reichweite auf die nächsten Nachbarn gekennzeichnet. Sie ist in beiden Richtungen möglich, wie in Abbildung 2.17b dargestellt.

Alle Varianten der direkten Kommunikation haben gemeinsam, dass die Inter-Task-Kommunikation sehr schnell vollzogen werden kann, da die Daten in der Regel noch im gleichen Takt beim Empfänger anliegen. Die Pipeline-Struktur baut diese Möglichkeit so weit aus, dass Daten in sehr wenigen Takten gleich mehrere Verarbeitungsschritte (Tasks) durchlaufen. Der große Nachteil all dieser Varianten ist jedoch, dass ein Datenaustausch nur zwischen den gleichzeitig auf dem FPGA laufenden Tasks erfolgenden kann. Lässt sich auf einem Chip nur ein Task-Slot realisieren, so ist dieses Konzept der Kommunikation leider nicht anwendbar.

2. Indirekte Inter-Task-Kommunikation

   Demgegenüber kann bei der indirekten Inter-Task-Kommunikation selbst bei nur einem Task-Slot ein Datenaustausch zwischen Tasks erfolgen. Hierzu wird ein gemeinsamer Speicher eingesetzt, der im System-Bereich untergebracht wird. Aus diesem Grund müssen die Tasks nicht untereinander verbunden werden. Es reicht eine Verbindung zum System, dass die Zugriffe auf den gemeinsamen Speicher koordiniert. Eine Kommunikation kommt zustande, wenn Task A Daten in den Speicher schreibt, die von Task B zu einem beliebigen späteren Zeitpunkt ausgelesen werden. Damit besitzt die indirekte Form der Kommunikation einen entscheidenden Vorteil gegenüber der direkten: Produzent- und Verbraucher-Tasks müssen nicht gleichzeit auf dem FPGA untergebracht sein, um Informationen austauschen zu können. Dabei kann der Datenaustausch auch über die Hardware-Tasks hinausgehen. Die Kontrolle des Systems über den Speicher bietet ideale Bedingen dafür, externen Geräten oder anderen auf dem Chip befindlichen Komponenten wie Prozessoren, Zugriff die Inhalte des Speichers zu gewähren. D.h. selbst Software könnte relativ leicht mit den Hardware-Tasks kommunizieren.

Abgesehen von dem Gedanken des Datenaustauschs kann der Speicher durch die Tasks auch zur Sicherung des eigenen Zustands eingesetzt werden, um nach einem Task-Wechsel mit den vorherigen Einstellungen und Zwischenergebnissen fortzufahren.

Letztendlich ist auch die Entscheidung zwischen direkter und indirekter Kommunikation eine Abwägung zwischen Geschwindigkeit der Datenübertragung und Flexibilität bezüglich des Zeitpunkts der Verarbeitung. Eine Kombination beider Formen ist ebenso denkbar.

(b) **Task-System-Kommunikation**

In der Realisierung der Task-System-Verbindung stehen sämtliche Modelle, die bereits für die Task-Task-Kommunikation untersucht worden, zur Auswahl. Von bi- und unidirektionalen Bussen, über Pipelines und ähnliche Infrastrukturen sind alle Formen der Anbindung der Tasks an den System-Bereich möglich. Zu den Gründen, eine solche Verbindung einzurichten zählen vor allem:

- Implementierung einer indirekten Inter-Task-Kommunikation
- Anbindung der Tasks an die vom System verwalteten Ressourcen
- Übertragung von Steuersignalen, die zur Ausführung der Tasks und für Task-Wechsel eingesetzt werden

Werden externe Geräte oder andere Ressourcen durch das System verwaltet, so kann den Tasks die Implementierung der Gerätetreiber abgenommen werden, indem die entsprechende Logik in dem System-Bereich untergebracht wird. Die Schnittstelle zwischen System und Task bringt zum einen den Vorteil einer hohen Verfügbarkeit der Ressourcen, da der System-Bereich nicht der Rekonfiguration unterliegt. Zum anderen kann die Auslagerung der Gerätetreiber in den Systembereich den Ressourcenverbrauch der Tasks deutlich reduzieren, sofern die Task-System-Schnittstelle eine Vereinfachung gegenüber den Gerätetreibern darstellt. Ein weiterer Grund für eine Verbindung zwischen Task und System ist die Übertragung von Steuerinformationen. Dazu zählen beispielsweise der Takt mit dem ein Task versorgt wird oder ein Reset-Signal, um Tasks in ihren initialen Zustand zu versetzen. Weitere Busleitungen können sinnvoll für Signale reserviert werden, die den Task-Wechsel betreffen. Signal-Beispiele wären:

- Nachricht an das System, dass ein Task seine Berechnung beendet hat und ausgelagert werden kann

- Nachricht an einen Task, dass er seinen Zustand sichern und die Berechnung anhalten soll

(c) **Kommunikation mit der Außenwelt**

Wie bereits in Kapitel 2.5.2 angedeutet, kann ein Task direkt mit externen Geräten und Speichern verbunden werden, falls die entsprechenden Pins im Rekonfigurationsbereich des Tasks liegen. Der Umgang mit externen Ressourcen lässt sich auf diese Weise flexibel gestalten. Zum einen könnte ein und derselbe Rekonfigurationbereich abwechselnd zur Ansteuerung verschiedener externer Geräte genutzt werden. Zum anderen wäre es denkbar, verschiedene Instanzen einer Kommunikationskomponente mit unterschiedlichen, festen Parameterwerten zu synthetisieren – beispielsweise für Übertragungsraten oder Fehlerkorrekturverhalten. Mit Hilfe der dynamischen Rekonfiguration wäre man dann in der Lage, stets die Instanz mit den gewünschten Eigenschaften auf dem FPGA laufen zu lassen. Statt eine universelle Komponente einzusetzen, die zwar keine Rekonfigurationslogik voraussetzt und deren Parameter ebenfalls zur Laufzeit gesetzt werden können, hätten die parametrisierten Instanzen den Vorteil, dass sie durch die Einschränkung in ihrem Funktionsumfang deutlich weniger Platz benötigen.

Anwendung findet dieses Prinzip der Verkleinerung von Hardware-Komponenten durch die Festlegung von Eigenschaften beispielsweise in der *UART-Lite*-Komponente von Xilinx, dem zur Synthesezeit optimierten Äquivalent des *Universal Asynchronous Receiver and Transmitter (UART)*.

2.7 Entwicklungswerkzeuge

An dieser Stelle soll die im Rahmen der Diplomarbeit eingesetzte Software kurz vorgestellt werden.

Xilinx bietet für die von ihnen stammenden FPGAs zwei große Entwicklungsumgebungen an, die den gesamten Design-Prozess unterstützen und automatisieren. Zu diesen gehören das Integrated Synthesis Environment (ISE) und das Embedded Development Kit (EDK).

2.7.1 Xilinx ISE

Das ISE umfasst eine Vielzahl von Kommandozeilen-Programmen, die zusammen den kompletten Arbeitsablauf für die Design-Erstellung abdecken, beginnend beim

VHDL-Syntax-Check, über die Synthese der Schaltung, die Platzierung und Verdrahtung der Schaltungselemente, und schließlich der Programmierung des FPGAs über die externe Schnittstelle. Eine Automatisierung dieses Arbeitsablaufs lässt sich entweder durch eine Batch-Datei realisieren, welche die entsprechenden Programme nacheinander startet, oder durch den *Project Navigator*. Der Project Navigator besitzt eine grafische Benutzeroberfläche und ermöglicht eine einfache Projektverwaltung. Zudem enthält er einen kleinen VHDL/Verilog-Editor und führt bei der Übersetzung der VHDL-Quellen in das Bitstream-Format nur die tatsächlich notwendigen Programme zur Designaktualisierung aus, wodurch Zeit gespart werden kann.

Ein weiteres grafisches Werkzeug der ISE-Entwicklungsumgebung ist der *FPGA-Editor*. Er kann zur Anzeige und und zur Bearbeitung fertiger Designs (NCD-Dateien[6]) oder zur grafischen Erstellung von Hardmacros herangezogen werden. Hardmacros können ebenso mit dem Kommandozeilenwerkzeug `xdl` erzeugt werden, jedoch ist dazu eine genaue Kenntnis der Beschreibungssprache *Xilinx Design Language (XDL)* erforderlich.

2.7.2 Xilinx EDK

Das EDK ist eine Zusammenstellung von Werkzeugen und Intellectual Property Cores (IP-Core), mit deren Hilfe eingebettete Prozessor-Systeme (embedded processor systems) für die Xilinx FPGAs entworfen werden können. Dabei wird sowohl der Hardware- als auch der Software-Entwurf abgedeckt. Unter den IP-Cores versteht man wiederverwendbare integrierte Schaltkreise unterschiedlichster Art, von einfachen arithmetischen Operationen bis hin zu komplexen System-Komponenten wie Filter, VGA-Controller oder BRAM-Speicher. Sie liegen entweder als Netzliste oder in einer Hardwarebeschreibungssprache vor. Zu den Werkzeugen des EDK zählen abermals eine Reihe von Kommandozeilen-Programmen sowie einige wenige Anwendungen mit einer grafischen Benutzeroberfläche. Zu letzteren zählt das *Xilinx Platform Studio (XPS)*, in welchem sich auf einfache Weise eingebettete Prozessor-Systeme aus verschiedenen IP-Cores zusammenstellen und verwalten lassen. Neue Projekte lassen sich am schnellsten mit dem Base System Builder (BSB), einer Dialogführung im Xilinx Platform Studio (XPS), erstellen. Einige der Fragen, die dabei beantwortet werden müssen, sind:

- Wahl des Prozessors: PowerPC oder MicroBlaze

[6]Die Native Circuit Description (NCD)-Dateien enthalten die Abbildung der Logik auf die Zellen (CLBs, IOBs, BRAMs, ...) des FPGAs

- Prozessor-Cache: ja oder nein
- Programm- und Datenspeicher: im BRAM oder SDRAM
- welche Peripherie wird gewünscht: LEDs, PushButtons, VGA, serielle Schnittstelle, ...
- mit welchem Bus soll die Peripherie jeweils verbunden werden: mit dem Processor Local Bus (PLB) oder dem On-Chip Peripheral Bus (OPB)

Der Processor Local Bus (PLB) ist direkt mit der Central Processing Unit (CPU) verbunden und wird meist zur Anbindung von prozessornahem Speicher und schneller Peripherie genutzt. Hauptaufgabe des On-Chip Peripheral Bus (OPB) hingegen ist die Anbindung der etwas langsameren Peripherie. Beide Busse sind über eine Brücke miteinander verbunden. Die Kommunikation unter den Bus-Komponenten erfolgt durch Lese- und Schreibezugriffe auf den Bus unter Angabe einer Adresse, die im Adressbereich der Ziel-Komponente liegen muss. Diese Adressbereiche können im XPS manuell oder automatisch zugewiesen werden. Zu jedem IP-Core enthält das EDK zugleich die C-Bibliotheken zur Ansteuerung der Hardware-Logik. Somit entfällt in den meisten Fällen eine aufwendige Software-Treiber-Entwicklung und die genaue Kenntnis über die Register der IP-Cores.

Da der OPB im Kapitel 5 eine bedeutende Rolle spielt, sollen an dieser Stelle die wesentlichen Eigenschaften des Busses vorgestellt werden. Grob lässt sich der Bus in folgende Bestandteile gliedern:

- Adressbus, 32-Bit breit
- Datenbus, 32-Bit breit
- Steuersignale für die Abstimmung der Kommunikation (Quittierung von Nachrichten, Meldung von Fehlern)
- Steuersignale für die Master-Vergabe

Der OPB ist mit einem Arbiter ausgestattet. Er steuert die Vergabe der Master-Rolle. Möchte eine Komponente auf den Bus schreiben oder lesen, so stellt sie eine Master-Anfrage. Sobald diese durch den Arbiter bestätigt wurde, kann die Komponente aktiv Daten versenden oder anfragen. Möchten zwei oder mehr Komponenten jedoch gleichzeitig auf den Bus zugreifen, so entzieht der Arbiter nach einer bestimmten Anzahl von Takten der aktiven Komponente wieder das Master-Recht, um es der nächsten zu übergeben und ihr ebenfalls die Möglichkeit zu Kommunikation zu gewähren. Dieses Verfahren lässt sich allerdings auch unterbinden. Eine Komponente,

die die Master-Rolle besitzt, kann durch das OPB-Steuersignal `busLock` den Bus auf unbestimmte Zeit sperren lassen. In diesem Zeitraum besitzt sie einen exklusiven Zugriff auf den Bus, d.h. die Master-Rolle kann nicht wechseln.

2.7.3 Weitere Werkzeuge

Der Markt der Entwicklungswerkzeuge für System-on-a-Chip (SoC)-Designs oder allgemein Hardware-Entwicklung wird zunehmend auch durch OpenSource-Projekte geprägt. Darunter befinden sich die Projekte GHDL und GTKWave, die in dieser Arbeit ihren Einsatz fanden. GHDL ist ein VHDL-Simulator. Mit ihm wurde das Verhalten des VHDL-Quellcodes getestet. Während der Abarbeitung der Tests schreibt GHDL die Signaländerungen in eine Datei, die von GTKWave grafisch dargestellt werden kann (siehe Abbildung 5.12 auf Seite 97).

2.8 Java

In diesem Kapitel soll auf die Eigenschaften von Java näher eingegangen werden, die für die Aufgabenstellung von besonderer Bedeutung waren. Für eine umfassende Darstellung der Programmiersprache sei unter den vielen Lehrbüchern exemplarisch auf das auch online verfügbare Buch *Java ist auch eine Insel*[Ull07] verwiesen.

2.8.1 Eigenschaften

Java ist eine objektorientierte Programmiersprache und setzt damit die Vorteile der Objektorientierung um, wie:

- Kapselung von Datenstrukturen und ihren Methoden in Klassen und Objekten
- Spezialisierung von Klassen durch die Vererbung

Gegenüber herkömmlichen Programmiersprachen, deren Quellcode durch den Compiler in Maschinensprache übersetzt wird, werden die Java-Quellen in einen plattformunabhängigen Java-Bytecode überführt, der durch die Java Virtual Machine (JVM) auf allen gängigen Betriebssystemen interpretiert und ausgeführt werden kann. Trotz der Interpretation des Bytecodes ist Java in der Performance etwa vergleichbar mit C und C++, da die JVM von Sun einen JIT(Just-In-Time)-Compiler einsetzt, der zur Laufzeit den Bytecode in Maschinencode übersetzt. Alternativ lässt sich Java

auch auf speziellen Java-Prozessoren ausführen. Der unter der GPLv3-Lizenz stehende Prozessor *JOP*[7] ist einer von ihnen.

2.8.2 Vererbung

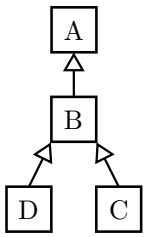
Das Prinzip der Vererbung ermöglicht Entwicklern bei der Erstellung neuer Klassen, die Funktionalität von bestehenden Klassen auf eine sehr einfache Weise zu übernehmen. Es reicht aus, den Klassennamen mit dem `extends`-Schlüsselwort und dem Namen der Oberlasse zu versehen:

```
class Unter extends Ober {
}
```

Auf diese Weise erbt die Unterklassen die Attribute und Methoden der Oberklasse. Man spricht auch davon, dass die Unterklasse von der Oberklasse abgeleitet wurde. Die neue Klasse kann alle sichtbaren Attribute und Methoden der Oberklasse so nutzen, als wären sie direkt in der neuen Klasse deklariert worden. Unter- und Oberlasse stehen in einer Ist-Eine-Art-Von-Beziehung. Durch die Übernahme der Methoden der Oberklasse implementiert die Unterklasse auch die gleiche Schnittstelle. Das heißt jedoch nicht, dass das Verhalten der Unterklasse mit dem der Oberklasse identisch sein muss. Im Gegenteil: Das Ziel der Vererbung ist eine Erweiterung und/oder Spezialisierung einer Klasse. In der Unterklasse können weitere Methoden angelegt und geerbte Methoden überschrieben werden, um das Verhalten wunschgemäß anzupassen. Dennoch bleibt die Ist-Eine-Art-Von-Beziehung stets in der Vererbungshierarchie bestehen. So lassen sich Instanzen einer Unterklasse in Java auch wie die Instanzen der Oberklasse verwenden.

```
class A {}
class B extends A {}
class C extends B {}
class D extends B {}
```

(a) Klassendeklarationen



(b) Klassenhierarchie

Abbildung 2.18: Beispiel einer Klassenhierarchie

[7] Zu finden ist der freie Prozessor JOP ist als IP-Core auf www.opencores.org

Abbildung 2.18 zeigt ein Beispiel einer Klassenhierarchie. Klasse `B` ist von `A` abgeleitet. Die Klassen `C` und `D` sind Unterklassen von `B`. Auch wenn die `C` und `D` keine direkten Abkömmlinge von `A` sind, lassen sich die Instanzen von `C` und `D` im Java-Code dennoch als vom Typ `A` deklarieren und als solche ansprechen:

```
A b = new B();
A c = new C();
B d1 = new D();
A d2 = new D();
```

2.8.3 Abstrakte Klassen

Abstrakte Klassen sind Klassen, die nicht instantiiert werden können. Sie werden gezielt eingesetzt, um ein Grundgerüst einer Klasse zu bauen, dass durch abgeleitete Klassen genutzt werden kann. Das Grundgerüst umfasst in der Regel eine Menge von Methoden, die in allen Unterklassen genutzt werden und auf diese Weise nur einmal implementiert werden müssen. Abstrakte Klassen können auch Methoden ohne Funktionskörper deklarieren. Diese werden als *abstrakte Methoden* bezeichnet und dienen nur der Schnittstellendefinition der Klasse. Abgeleitete Klassen müssen stets alle abstrakten Methoden implementieren, oder selbst als abstrakt gekennzeichnet werden.

2.8.4 Zeiger und Referenzen

In Java werden Verweise auf Objekte nicht durch Zeiger sondern durch sogenannte Objektreferenzen realisiert. Während sich Zeiger in C und C++ auf beliebige Speicherbereiche setzen lassen, sind die Referenzen in Java an Objekte gebunden. Aus diesem Grund ist Java in Bezug auf Sicherheit den Programmiersprachen mit Zeigern überlegen. Auch die in C/C++ gefürchteten Pufferüberläufe stellen für Java-Programme keine Gefahr dar, da der Zugriff auf ein Array in Java stets über einen Index statt über einen Zeiger läuft. Liegt der Index außerhalb der Array-Grenzen, so wird eine Exception geworfen, die eine Ausnahmebehandlung nach sich zieht.

2.8.5 Garbage-Collector

Das Anlegen und Löschen von Objekten ist stets mit einer Speicher-Allokation und -Freigabe verbunden. In Java wird diese Aufgabe automatisch durch den *Garbage-Collector (GC)* übernommen. Er ist ein Teil der JVM und arbeitet als nebenläufiger

Thread im Hintergrund. Sobald auf ein Objekt keine Referenz mehr existiert, wird es vom Garbage-Collector (GC) markiert. In unregelmäßigen Abständen werden die markierten Objekte entfernt und der zugehörige Speicher freigegeben. Vor dem Entfernen ruft der GC die `finalize`-Methode des Objektes auf. In dieser können letzte Aktionen, wie das Freigeben von Ressourcen, durchgeführt werden.

2.8.6 Threads

Jedes moderne Betriebssystem verfügt heutzutage über die Fähigkeit, mehrere Prozesse scheinbar gleichzeitig nebeneinander laufen zu lassen. Um diese Illusion selbst auf Ein-Prozessor-Maschinen umzusetzen, werden den Prozessen einzelne Zeitscheiben für die Ausführung auf der CPU zugeordnet. Für den Prozess-Wechsel verantwortlich ist der *Scheduler* des Betriebssystems. Da die Prozesse nicht wirklich gleichzeitig laufen, bezeichnet man diese Form der Parallelität als *Quasiparallelität* oder *Nebenläufigkeit*.

Die Nebenläufigkeit ist dabei nicht auf Prozesse untereinander beschränkt. Oftmals ist es sinnvoll, auch innerhalb eines Prozesses, bestimmte Programmteile gleichzeitig ausführen zu lassen. Diese gleichzeitig ausführbaren Programmteile werden als Threads (zu Deutsch *Fäden*) bezeichnet. In modernen Betriebssystemen besteht jeder Prozess aus mindestens einem Thread und es werden im eigentlichen Sinn nicht die Prozesse parallel ausgeführt, sondern die Threads. Wenn das Betriebssystem dies nicht unterstützt, wird die Verwaltung der Threads in Java durch die virtuelle Maschine übernommen und die Parallelität auf dieser Ebene simuliert.

In Java ist auch ein Thread ein Objekt, dass Eigenschaften und Methoden besitzt, um gestartet und gestoppt werden zu können. Um einen neuen Thread zu programmieren, muss eine neue Klasse erstellt und von `java.lang.Thread` abgeleitet werden. In der neuen Klasse kann nun die `run`-Methode überschrieben und mit dem Programmcode des Threads gefüllt werden (siehe Abbildung 2.19).

Nachdem ein Thread-Objekt erzeugt wurde, kann es über die `start`-Methode gestartet werden. Zu der Art, wie ein Thread zu stoppen sei, gibt es mittlerweile keine feste Vorgabe mehr. Die frühere `stop`-Methode gilt heute als veraltet, da sie den Thread abrupt abbricht, ohne dabei auf die Freigabe der Ressourcen zu achten, die der Thread zu diesem Zeitpunkt besitzt. Sun rät daher dringend vom Einsatz der `stop`-Methode ab und empfiehlt stattdessen die Methode `interrupt()`, die lediglich eine Variable – den Interrupted-Status – auf `true` setzt, welche durch den Thread selbstständig in regelmäßigen Abständen abgefragt werden sollte, um auf einen Unterbrechungswunsch zu reagieren. Im Zusammenhang mit `interrupt()` stehen den

```
public class MyThread extends java.lang.Thread {
  public void run() {
    System.out.println( "Thread wurde gestartet" );
    while( !isInterrupted() ) {
      System.out.println( "Thread arbeitet" );
      try {
        Thread.sleep( 1000 );
      } catch( InterruptedException e ) {
        interrupt();
      }
    }
    System.out.println( "Thread wird beendet" );
  }
}
```

Abbildung 2.19: Benutzerdefinierter Thread

Anwendern folgende Thread-Methoden zur Manipulation des Interrupted-Status zur Verfügung:[Mic07]

- `interrupt()`: zur Unterbrechung eines Threads; setzt den Interrupted-Status
- `isInterrupted()`: zur Überprüfung, ob eine Unterbrechung angefordert wurde; gibt `true` zurück, wenn der Interrupted-Status gesetzt ist
- `interrupted()`: das Gleiche wie `isInterrupted()`, aber mit anschließender Löschung des Interrupted-Status

Der Thread aus dem Beispiel 2.19 nutzt dieses Prinzip und lässt sich demnach mit dem Kommando `interrupt()` sauber beenden. Sobald ein Thread gestoppt ist, bleibt er in diesem Zustand. Ein Neustart ist leider nicht möglich.

Mit `sleep()` und `wait()` lassen sich Threads schlafen legen, um entweder wie bei `sleep()` nach einer bestimmten Zeit wieder aufzuwachen, oder wie bei `wait()` durch ein `notify()` geweckt zu werden. Einen weiteren Anlass, einen Thread vorübergehend anzuhalten, liefern Codebereiche, die nur von einem Thread gleichzeitig betreten werden dürfen. Dieser Aspekt wird im folgenden Kapitel näher betrachtet.

In Abbildung 2.20 werden die 6 verschiedenen Zustände eines Java-Threads noch einmal zusammengefasst.

2.8.7 Wechselseitiger Ausschluss

Wann immer mehrere Threads eine gemeinsame Ressource nutzen, muss geprüft werden, ob und an welchen Stellen ein gleichzeitiger Zugriff ausgeschlossen werden muss.

Zustand	**Beschreibung**
NEW	Neuer Thread, noch nicht gestartet.
RUNNABLE	Läuft in der JVM.
BLOCKED	Wartet auf einen Monitor-Lock, wenn er etwa einen synchronized Block betreten möchte.
WAITING	Wartet etwa auf ein notify().
TIMED_WAITING	Wartet etwa in einem sleep().
TERMINATED	Ausführung beendet.

Abbildung 2.20: Zustände eines Java-Threads [Ull07]

Sofern alle Threads nur leseseitig auf die Ressource zugreifen, sind die Zugriffe unproblematisch. Sobald jedoch mindestens ein Thread die Ressource manipuliert, wird es kritisch. Während der Manipulation befindet sich die Ressource in der Regel in einem inkonsistenten Zustand, der durch andere Threads weder gelesen noch gleichzeitig geändert werden sollte. Die Programmabschnitte, die vor dem Zugriff anderer Threads auf die gleiche Ressource geschützt werden müssen, nennt man *kritische Abschnitte*. Die einfachste Form, um in Java einen wechselseitigen Ausschluss für diese Programmblöcke zu gewährleisten, ist, sie durch einen `synchronized`-Block zu umschließen. Ein `synchronized`-Block ist immer mit einer Ressource verbunden, auf die sich der wechselseitige Ausschluss bezieht. Einer Ressource lassen sich mehrere kritische Blöcke zuordnen. Sobald ein Thread sich in einem dieser Blöcke befindet, kann kein weiterer Thread die Abschnitte betreten. Er wird vor dem Betreten des Blockes angehalten und erst wieder aufgeweckt, sobald der andere Thread den Block verlassen hat oder oder sich mittels *`resource`*`.wait()` vorübergehend schlafen legt.

Ein Beispiel für die Notwendigkeit der Synchronisation zwischen Threads ist in Abbildung 2.21 dargestellt. Zu sehen sind zwei kritische Bereiche, die durch unterschiedliche Threads ausgeführt werden, von denen beide auf eine gemeinsame Liste zugreifen. Die kritischen Bereiche der Threads sind durch `synchronized`-Blöcke bezüglich der Liste `list` umrahmt, sodass sich zu einem Zeitpunkt stets nur ein Thread im `synchronized`-Bereich in Ausführung befinden kann. Für die Abarbeitung der Threads kann es daher nur zwei verschiedene Szenarien geben. Entweder betritt Thread A den kritischen Bereich zuerst oder Thread B.

Wenn A zuerst den kritischen Bereich betritt, so testet der Thread zunächst, ob die Liste leer ist. Wenn das nicht der Fall ist, wird das erste Listenelement ausgegeben. Anderenfalls legt sich der Thread schlafen und wartet auf ein `list.notify()`

Lesezugriff im Thread A

```
synchronized( list ) {
  if (list.isEmpty()) {
    list.wait();
  }
  System.out.println(list.get(0));
}
```

Schreibzugriff im Thread B

```
synchronized( list ) {
  list.add("a");
  list.notify();
}
```

Abbildung 2.21: Beispiel für einen wechselseitigen Ausschluss [Ull07]

von B. In dem schlafenden Zustand wird der exklusive Zugriff auf die Liste aufgehoben. Thread B kann nun also seinen kritischen Bereich betreten und fügt als erstes ein Element der Liste hinzu. Durch `list.notify()` benachrichtigt er danach den wartenden Thread A und gibt dabei die Sperre auf `list` seinerseits wieder frei. Thread A läuft nun weiter, gibt das erste Listenelement auf der Konsole aus und verlässt danach den kritischen Abschnitt. Nach dem Verlassen des `synchronized`-Blocks ist `list` nicht mehr gesperrt und Thread B setzt seine Abarbeitung nach dem `list.notify()` fort und gelangt damit ebenfalls zum Ende seines kritischen Abschnitts.

Anders sieht es dagegen aus, wenn Thread B zuerst den kritischen Abschnitt betritt. Zunächst fügt er der Liste ein neues Element hinzu. Der nächste Befehl `list.notify()` kehrt sofort zurück, da kein Thread auf die Liste wartet. Solange B sich im kritischen Bereich befindet, bleibt dem Thread A das Betreten des `synchronized`-Blocks weiter verwehrt. Erst nachdem Thread B das Ende des Blockes erreicht, wird die Sperrung auf `list` aufgehoben und Thread A kann seinen kritischen Abschnitt betreten. Dieser testet, ob die Liste leer ist, wobei der Test negativ ausfällt und Thread A dadurch sofort das erste Listenelement ausgibt.

Ohne die `synchronized`-Blöcke in den beiden Threads könnte es passieren, dass Thread B während der Ausführung von `list.add("a")` vom Scheduler unterbrochen wird. Wenn dies in dem Moment geschieht, in dem die Größe der Liste bereits angepasst, das Listenelement jedoch noch nicht hinzugefügt wurde, befindet sich die Liste in einem inkonsistenten Zustand. Sie würde nicht als leer gelten und Thread A würde das erste Element zu lesen versuchen, welches allerdings noch nicht verfügbar ist. In Folge dessen würde eine Exception geworfen werden.

Neben den `synchronized`-Blöcken bietet Java auch objektorientierte Lösungen für den wechselseitigen Ausschluss. So lassen sich beispielsweise Instanzen von `java.util.concurrent.locks.ReentrantLock` durch die Methoden `lock()` und `unlock()` sperren bzw. wieder freigeben. Der Vorteil hierbei ist, dass die Sperre und

Freigabe anders als beim `synchronized`-Block auch in verschiedenen Methoden untergebracht werden können.

Im Framework wurden letztlich beide Formen eingesetzt. Es überwiegen allerdings die `synchronized`-Blöcke aufgrund ihrer sehr kompakten Schreibweise.

Kapitel 3

Stand der Technik

Der Grundgedanke, die Flexibilität von Software mit der Geschwindigkeit von Hardware zu kombinieren, reicht in die 60er Jahre zurück. Damals entstanden die ersten wissenschaftlichen Arbeiten auf dem Gebiet des Reconfigurable Computing, welches sich mit dem Einsatz rekonfigurierbarer Hardware in Software-Systemen beschäftigt. Heutige Hardware-Technologien setzen dieser Idee kaum noch Grenzen. Die stetige Verbesserung und Weiterentwicklung rekonfigurierbarer Hardware sowie die sinkenden Preise für FPGAs sorgten in den letzten Jahren für eine zunehmende Bedeutung dieses Forschungsgebiets.

Im Blickfeld dieser Arbeit liegt speziell das Thema der Integration rekonfigurierbarer Hardware in moderne Programmiersprachen wie Java. Eine Auswahl wesentlicher Forschungsarbeiten zu diesem Thema soll nun vorgestellt werden, um den Stand der Technik zu verdeutlichen.

Die erste Arbeit beschäftigt sich mit einer kompletten Spezifikation eines Software/Hardware-Co-Designs in Java und hebt dabei die Vorteile von Java gegenüber anderen objektorientierten Programmiersprachen hervor. Anschließend werden zwei Arbeiten vorgestellt, die sich mit der Entwicklung einer Software/Hardware-Co-Design-Umgebung auf Basis von Java und unter Einsatz rekonfigurierbarer Hardware auseinandersetzen. Dabei werden die wesentlichen Aspekte einer solchen Umgebung beleuchtet und verschiedene Lösungsvarianten vorgestellt. Abschließend wird eine Arbeit erörtert, die sich mit der Beschleunigung der dynamischen partiellen Rekonfiguration beschäftigt, welche Voraussetzung für ein effizientes Hardware-Scheduling ist. Die dazu entwickelte Prototyp-Implementierung bildet zudem die Grundlage für die hier vorliegende Arbeit.

3.1 Java als Spezifikationssprache für Hardware-Software-Systeme

Die Sichtweisen von Hardware-Beschreibungssprachen (HDL) und Software-Programmiersprachen sind grundsätzlich verschieden. Während die Anweisungen in einem Software-Programm sequentiell ausgeführt werden, sind die Operationen in einer HDL parallel zu interpretieren. Doch gerade im Bereich des Hardware/Software-Co-Designs wäre es von großem Vorteil, wenn das komplette Co-Design in nur einer Sprache beschrieben werden könnte. Eine solche Lösung würde die Simulations- und Analysemöglichkeiten stark begünstigen.

Die Arbeit [HO97] widmet sich dieser Idee unter Verwendung von Java als Beschreibungssprache. Im Gegensatz zu auf C++ basierenden Sprachen wie SystemC[1] und Scenic [LTG97], die die C++-Syntax um typische Eigenschaften von Hardware-Beschreibungssprachen erweitern, sieht diese Arbeit vor, das komplette System in reinem Java zu formulieren. Die Spezifikation des Systems erfolgt somit – wie in Java üblich – auf einer algorithmischen und verhaltensbeschreibenden Ebene. Die fertige Spezifikation wird anschließend in einem automatisierten Prozess analysiert, um die Nebenläufigkeit zu ermitteln. Darauf aufbauend wird die Hardware-Software-Partitionierung und die Synthese mit dem Ziel einer optimalen Performance des gesamten Systems vorgenommen. Die Arbeit befasst sich hauptsächlich mit der Ermittlung der Nebenläufigkeit und erörtert dazu Lösungsmöglichkeiten, die in einer Prototyp-Implementierung umgesetzt wurden.

Die Ermittlung der Nebenläufigkeit erfolgt in der Arbeit in drei Stufen. Zuerst wird die Parallelität auf Thread-Ebene ermittelt, die durch die Ableitung von der Klasse Thread in Java explizit beschrieben ist. Folglich ist eine Analyse der Klassenstruktur ausreichend, um festzustellen, welche Klassen von Thread abgeleitet sind.

Die feinkörnigere Nebenläufigkeit in prozeduralem Quellcode ist hingegen nur implizit gegeben, da bestimmte Software-Operationen in Hardware gleichzeitig ausgeführt werden können, sofern keine Datenabhängigkeiten zwischen diesen bestehen. Eine bedeutende Form der feinkörnigen Nebenläufigkeit stellen Schleifen dar. Oftmals bauen die Schleifen-Iterationen nicht auf den Ergebnissen vorheriger Iterationen auf und können unabhängig voneinander und theoretisch gleichzeitig ausgeführt werden. Je mehr Iterationen in einer Schleife durchlaufen werden, desto größer ist das Potential einer Performancesteigerung in Hardware. Wie viele Iterationen letztendlich auf der Hardware parallel ausgeführt werden, hängt auch von den verfügbaren

[1] www.systemc.org

Hardware-Ressourcen ab. Um die Abhängigkeiten zwischen den Iterationen zu bestimmen, müssen die Schleifen zunächst analysiert werden. Die größte Schwierigkeit dabei betrifft die Behandlung von Referenzen, da diese gegenüber primitiven Datentypen einen indirekten Zugriff auf den physischen Speicher bedeuten. Während Variablen primitiven Typs nach der Deklaration an einen festen Speicherbereich gebunden sind, können Objektreferenzen auch innerhalb der Schleife umgesetzt werden. Die Compile-Zeit-Analyse kann daher nur schwer Aussagen über den Inhalt der Objektattribute treffen und muss im Zweifel eine Entscheidung gegen die Parallelisierung der Iterationen treffen. Dennoch werden an dieser Stelle die Vorteile von Java gegenüber C++ deutlich. Die strikte Beschränkung auf typisierte Objektreferenzen anstelle der Nutzung von Adress-Zeigern wie in C++ reduziert die fälschlich erkannten Datenabhängigkeiten in den Analysen deutlich. Beispielsweise ist es in C++ nicht unüblich, dass Zeiger innerhalb einer Schleife berechnet werden. Diese Technik wird unter anderem beim Schreiben oder Lesen aus einem Array genutzt. In Java hingegen kann auf die Inhalte eines Arrays nur über Indizes zugegriffen werden. Desweiteren sind Zugriffe außerhalb des zulässigen Array-Bereiches in Java nicht möglich, während der Pointer in C oder C++ durchaus die Grenzen überschreiten kann, wodurch die typischen Pufferüberläufe entstehen.

Nachdem die Analyse der Schleifen abgeschlossen ist, wird mit der Ermittlung der Nebenläufigkeit auf Bytecode-Ebene fortgefahren. Dazu wird der Bytecode der einzelnen Methoden analysiert, um die Reihenfolge der Rechenoperationen und Methodenaufrufe sowie deren Abhängigkeiten untereinander zu bestimmen. Auch hier wird die statische Inspektion durch die dynamischen Referenzen erschwert, da Objektreferenzen erst zur Laufzeit mit den entsprechenden Klassen verknüpft werden. D.h. nur während der Ausführung des Java-Programms lassen sich die Objektreferenzen exakt auflösen und der Inhalt der Objektmethoden ermitteln. Eine Lösungsmöglichkeit ist daher, bei der Ausführung einer Java-Anwendung stets alle Methodenaufrufe durch die Laufzeitumgebung durchführen zu lassen, was jedoch die Möglichkeiten für die Parallelisierung in Hardware deutlich reduzieren würde. Aufgrund dieser Nachteile wurde in der Arbeit ein etwas anderer Ansatz mit einer statischen Analyse des Bytecodes verfolgt, wobei zugunsten der Analyse einige Anforderungen an den Java-Code gestellt werden. Unter anderem wird angenommen, dass zu jeder Objektreferenz das entsprechende Objekt nur ein einziges Mal instantiiert wird und sich die Referenz danach nicht mehr ändert. Um das Problem der dynamischen Zuordnung der Objektreferenzen zu Klassen zu lösen, wird weiter vorausgesetzt, dass der Typ eines Objektes immer exakt mit seinem Deklarationstyp übereinstimmt. Erfüllt der

Quellcode diese Einschränkungen, so können aus der statischen Analyse korrekte Ergebnisse gewonnen werden.

Die Analyse beginnt mit der Ermittlung des Kontrollflusses, indem festgestellt wird, welche Methodenaufrufe innerhalb welcher Methoden erfolgen. Diese Abhängigkeiten können in einem Graphen abgebildet werden, dessen Knoten die verschiedenen Objektmethoden repräsentieren. Im Paper wird anschließend ausführlich beschrieben, wie sich der Kontrollfluss und die Datenabhängigkeiten auf Methodenebene weiter bestimmen lassen. Das Resultat der Analysen ist ein Control-Data Flow Graph (CDFG), der alle nötigen Informationen enthält, die für die Umsetzung der Spezifikation in Hardware benötigt werden. Auf letztere wird im Paper jedoch nicht weiter eingegangen.

Das Paper zeigt zum einen, dass Java für die Spezifikation eines Software/Hardware-Co-Designs einige Vorteile gegenüber anderen Programmiersprachen besitzt. Weiter wurde gezeigt, dass unter bestimmten Annahmen über den Quellcode auch eine statische Analyse des Byte-Codes möglich ist, um den Kontroll- und Datenfluss der Anwendung zu bestimmen. Mit den gewonnenen Ergebnissen sind die Voraussetzungen für die anschließend folgende Partitionierung und Co-Synthese geschaffen.

In den folgenden Kapiteln sollen nun Arbeiten vorgestellt werden, die sich mit dem vollständigen Prozess der Spezifikation, Partitionierung und Co-Synthese eines Hardware/Software-Co-Designs beschäftigen. Dabei wird ebenfalls beleuchtet, wie eine Java-Anwendung in einer solchen Umgebung ausgeführt wird.

3.2 Co-Design eingebetteter Systeme auf Basis von Java und rekonfigurierbarer Hardware

In [FBK99] beschreiben Josef Fleischmann und andere eine Software/Hardware-Co-Design-Umgebung auf Basis von Java, die es ermöglicht, Java-Anwendungen zu entwickeln, die zum Teil in Software und zum Teil auf rekonfigurierbarer Hardware ausgeführt werden. Die Autoren gehen dabei auf den Prozess der Co-Synthese und auf die Problematik der Ausführung einer solchen Anwendung ein. Weiter wurden die beschriebenen Aspekte in einer Prototyp-Implementierung umgesetzt.

Die Idee der Arbeit besteht darin, die Java-Spezifikation einer Anwendung auf der Ebene der Methoden in Software- und in Hardware-Programmteile aufzuteilen, welche später entweder auf der CPU oder der rekonfigurierbaren Hardware ausgeführt werden. Um die entsprechenden Hardware/Software-Anwendungen auszuführen, gibt es verschiedene Möglichkeiten. Zwei grundlegende Ansätze werden in der Arbeit vor-

gestellt und ihre Vor- und Nachteile verglichen.

Die Architektur, auf der die erstellten Anwendungen ausgeführt werden, besteht aus einem Mikroprozessor, auf dem eine Java Virtual Machine (JVM) läuft, und mindestens einem FPGA. Der Entwurf der Anwendung beginnt mit einer Spezifikation in Java. Um aus dieser die zur Ausführung nötigen Software- und Hardware-Bestandteile zu generieren, wird der in Abbildung 3.1 dargestellte Arbeitsablauf durchlaufen.

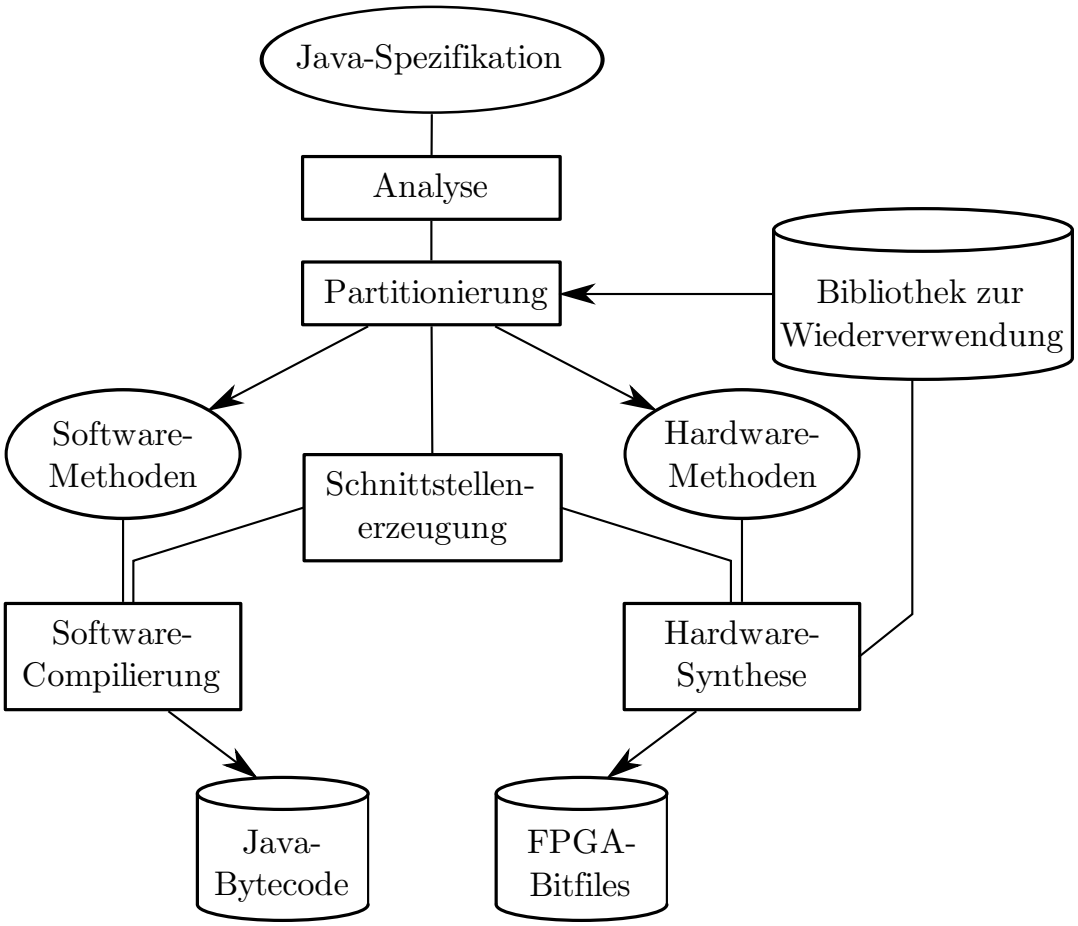


Abbildung 3.1: Arbeitsablauf ab der Spezifikation bis zur Co-Synthese

Zunächst wird dazu die Anwendung in Software ausgeführt und dabei im Hinblick auf die darauffolgende Partitionierung analysiert. Die Ergebnisse der Analyse werden in einer grafischen Benutzeroberfläche dargestellt, welche gleichzeitig den Anwendungsentwickler durch den Partitionierungsvorgang führt. Dabei werden die einzelnen Methoden der Anwendung in Software- und Hardware-Methoden eingeteilt. Die Software-Methoden können durch einen Java-Compiler in Java-Bytecode übersetzt werden. Die Hardware-Methoden müssen hingegen noch einmal in einer Hardware-

Beschreibungssprache formuliert werden, um sie anschließend mit Hilfe von High-Level- und Logik-Synthese-Werkzeugen in FPGA-Bitfiles zu übersetzten. Für eine mögliche spätere Wiederverwendung werden die Hardware-Implementierungen in einer Bibliothek gespeichert, welche aus Effizienzgründen auch in den Partitionierungsprozess einbezogen wird. Im Ergebnis des Co-Syntheseprozesses liegen die Software-Methoden im Java-Bytecode und die Hardware-Methoden als Bitfiles vor.

Bei der Ausführung der Anwendung nimmt der Mikroprozessor eine zentrale Rolle ein, da der Kontrollfluss der Anwendung durch den Software-Teil gesteuert wird, den die JVM auf dem Mikroprozessor ausführt. Wird eine Hardware-Methode aufgerufen, so stellt die Laufzeitumgebung fest, ob sich die Methode bereits in einer rekonfigurierbaren Hardware befindet, lagert sie notfalls ein und initialisiert die Kommunikationskanäle. Dies erfordert jedoch einige Anpassungen an der JVM. Dabei kam die freie JVM *Kaffe*[2] zum Einsatz, da ihr kompletter Quellcode verfügbar ist. Der Classloader wurde so erweitert, dass er neben dem Java-Bytecode auch die Partitionierung der Hardware/Software-Methoden einliest und Hardware-Methoden mit einer Kennzeichnung versieht. Trifft der ebenfalls modifizierte Bytecode-Interpreter bei einer Programmausführung auf eine so gekennzeichnete Methode, wird der FPGA gegebenenfalls mit der Hardware-Methode programmiert. Anschließend werden die Eingabedaten der Methode zum FPGA übertragen, die Ausführung gestartet und danach das Ergebnis zum Mikroprozessor zurückgesendet. Die Zugriffe auf den FPGA werden dabei synchronisiert, sodass stets nur ein Thread den FPGA gleichzeitig nutzen kann.

Die Partitionierung der Methoden erfolgt in speziellen Dateien. Durch Anpassung dieser kann der Ausführungsmodus der Methoden, die sowohl in Hardware als auch in Software implementiert wurden, leicht geändert werden. Liegt zu jeder Methode eine Software-Implementierung vor, so ist eine Ausführung der Anwendung auch ohne den Einsatz rekonfigurierbare Hardware möglich. In diesem Fall muss die Ausführung nicht einmal in der modifizierten JVM erfolgen, da der Java-Bytecode der Anwendung keinerlei Abhängigkeiten zum Hardware/Software-Co-Designs aufweist. Somit lässt sich die Anwendung in jeder beliebigen JVM in Software ausführen.

Aufgrund der Tatsache, dass diese Implementierungsvariante eine vollständig quelloffene Java-Laufzeitumgebung voraussetzt und für verschiedene oder aktualisierte JVMs stets aufs Neue angepasst werden muss, wurde darüber hinaus ein alternatives Konzept erarbeitet, das ohne Erweiterungen der JVM auskommt. Dabei wurde eine Java-Klasse geschrieben, die anderen Klassen eine Schnittstelle für den Zugriff

[2] www.kaffe.org

auf die rekonfigurierbare Hardware bereitstellt. Zur Implementierung dieser Klasse musste auf das Java Native Interface (JNI)[3] zurückgegriffen werden, um die Kommunikation mit dem FPGA zu realisieren. Dennoch sind diese plattformabhängigen Teile leichter zu warten als Anpassungen einer JVM.

Die Vorteile dieser zweiten Implementierungsvariante liegen hauptsächlich in der Unabhängigkeit von einer bestimmten virtuellen Maschine und im Wegfall ständiger JVM-Anpassungen. Ebenso verschafft die Schnittstellen-Klasse zum FPGA dem Entwickler neue Möglichkeit der Einflussnahme. Beispielsweise können so Rekonfigurationen in einer Anwendung gezielt ausgeführt werden, statt vom Scheduler der JVM abhängig zu sein.

Nachteilig ist jedoch, dass die Rekonfiguration des FPGAs in der zweiten Variante nicht mehr durch die JVM verdeckt abläuft, sondern explizit durch den Entwickler im Java-Code gesteuert werden muss. Bestehende Java-Anwendungen lassen sich demzufolge nicht ohne Anpassungen in ein solches Hardware/Software-Co-Design überführen.

3.3 Verbesserung der Java-Performance durch Hardware-Methoden

Den Ansatz, ein Hardware/Software-Co-Design auf Basis von Java durch Anpassungen an der Java Virtual Machine zu realisieren, verfolgten ebenfalls Emanuele Lattanzi und andere im Paper [LGK+04]. Ziel dieser Arbeit war es, die Ausführung von Java-Anwendungen zu beschleunigen, indem rechenintensive Methoden in Hardware ausgeführt werden. Das Paper beschreibt detailliert eine Laufzeitumgebung, die Java-Anwendungen unter Einsatz eines Hauptprozessors und eines FPGAs als Co-Prozessor ausführt und zur Laufzeit Performance-Optimierungen vornimmt.

3.3.1 Systemarchitektur

Wie auch in [FBK99] setzen die Autoren einen Mikroprozessor (Hauptprozessor) ein, dessen Aufgabe die Ausführung von Java-Anwendungen innerhalb einer JVM ist. Die weiteren Bestandteile der Architektur sind ein FPGA, verschiedene Speicher und ein gemeinsamer Bus, über welchen alle Komponenten miteinander verbunden sind. Abbildung 3.2 verdeutlicht den Aufbau grafisch.

[3]Das JNI ermöglicht die Einbindung von nativem C-Code in Java-Programme. Da Java keinen direkten Zugriff auf System-Ressourcen zulässt, ist der Einsatz von JNI zur Ansteuerung bestimmter

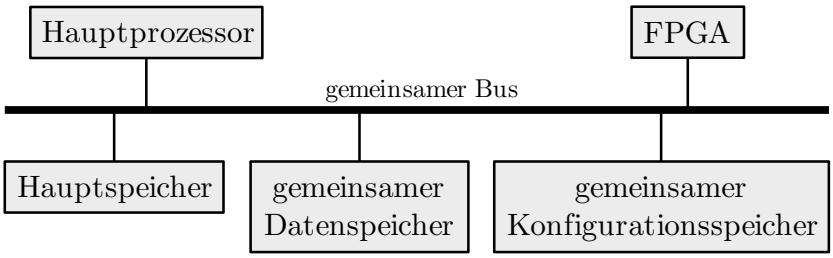


Abbildung 3.2: Architektur des Systems

Unterschieden werden drei Speicher:

- Der *Hauptspeicher* steht nur der CPU zur Verfügung und dient unter anderem der Java-Laufzeitumgebung zur Speicherung von Objekten.
- Der *gemeinsame Speicher* wird hingegen durch die CPU und den FPGA zum Datenaustausch eingesetzt.
- Der *Konfigurationsspeicher* wird zur Ablage der Bitstreams von Hardware-Methoden genutzt.

Die Bitstreams wurden in dieser Arbeit zur Compile-Zeit erstellt und zum Start der Anwendung in den Konfigurationsspeicher geschrieben. Es wird jedoch darauf hingewiesen, dass die Erstellung prinzipiell auch zur Laufzeit denkbare wäre. Der Java-Bytecode, der eine Methode beschreibt, könnte entweder per Software oder durch eine Hardware-Komponenten in einen Bitstream übersetzt werden. Der Konfigurationsspeicher ist daher abstrakt zu sehen.

Die Trennung von Hauptspeicher und gemeinsamen Speicher ermöglicht außerdem Optimierung durch Caches. Die Inhalte des Hauptspeichers können zum Zwecke eines schnelleren Zugriffs in einem prozessornahen Cache zwischengespeichert werden. Die Daten im gemeinsamen Speicher dürfen jedoch nicht auf die gleiche Weise zwischengespeichert werden, da auch der FPGA die Inhalte dieses Speichers ändern kann. In der Prototyp-Implementierung wurde daher auf Cache-Verfahren für den gemeinsamen Speicher verzichtet.

Damit sowohl der Hauptprozessor als auch der FPGA Zugriff auf den gemeinsamen Speicher haben, muss der gemeinsame Bus mehrere Master unterstützen. Desweiteren wurde der FPGA mit einer Busschnittstelle versehen, über welche die CPU die Möglichkeit besitzt, auf Register des FPGAs adressbezogen zuzugreifen. Diese Schnittstelle dient vor allem der Übergabe von Methodenparametern.

Ressourcen unausweichlich.

3.3.2 Dynamische Auswahl der Hardware-Methoden

Wie bereits angesprochen, wurde die JVM an verschiedenen Stellen angepasst, damit die Methoden innerhalb einer Anwendung sowohl in Software als auch in Hardware ausgeführt werden können, ohne dabei in den Quellcode der Anwendung eingreifen zu müssen. Gleichzeitig sollte die Entscheidung, welche Methode in Hardware abgearbeitet wird, dynamisch anhand bestimmter Kriterien, wie beispielsweise der Nutzungshäufigkeit einer Methode, ermittelt werden. Zu diesem Zweck sammelt die JVM während der Anwendungsausführung verschiedene Informationen über die Methoden.

Im Paper wird ein Entscheidungsverfahren beschrieben, das zunächst die *Hitze* jeder Methode bestimmt. Diese ergibt sich dabei aus der Anzahl ihrer Aufrufe innerhalb eines fortlaufenden Zeitfensters. Mit jedem Methodenaufruf wird die Hitze der Methode aktualisiert und mit der maximalen Hitze, die aktuell eine Methode auf sich vereint, verglichen. Ziel ist es, stets die Methode mit der größten Hitze in Hardware auszuführen. Für die tatsächliche Ausführung in Hardware müssen jedoch einige Anforderungen durch die Methode erfüllt sein:

1. Sie muss entweder in einer synthetisierbaren Beschreibung oder als fertiger Bitstream vorliegen.
2. Alle Objekte, auf die die Methode zugreift, müssen sich im gemeinsamen Speicher befinden.
3. Auf der rekonfigurierbaren Hardware muss ausreichend Platz vorhanden sein.

Sobald die JVM eine Methode für die Ausführung in Hardware ausgewählt hat, initiiert sie die Rekonfiguration des FPGAs. Die Übertragung der Bitstreams an die Rekonfigurationseinheit erfolgt dabei per Direct Memory Access (DMA), wodurch eine hohe Rekonfigurationsgeschwindigkeit erzielt wird und der Prozessor während der Rekonfiguration weiterarbeiten kann.

3.3.3 Co-Prozessor-Schnittstelle

Die Kommunikation zwischen dem Hauptprozessor und dem FPGA wurde durch einen gemeinsamen Speicher realisiert. Objekte, die sowohl durch Software- als auch durch Hardware-Methoden gelesen oder geschrieben werden sollen, wurden in der Arbeit als *Shared Objects* bezeichnet. Um diese Objekte, nicht wie die restlichen Java-Objekte im Hauptspeicher, sondern im gemeinsamen Speicher abzulegen, wurde die JVM entsprechend angepasst.

Beim Aufruf einer Hardware-Methode werden die Übergabeparameter und der Rückgabewert ebenfalls als Shared Objects behandelt und im gemeinsamen Speicher abgelegt. Damit die Hardware-Methode auf diese zugreifen kann, werden die Pointer der Objekte dem FPGA über die Register der FPGA-Schnittstell mitgeteilt. Handelt es sich bei den Übergabeparametern um primitive Datentypen, so werden die Inhalte der Parameter direkt in den Registern gesetzt.

3.3.4 Synchronisation

Der wechselseitige Ausschluss beim Zugriff auf den gemeinsamen Speicher wurde umgesetzt, indem dieser Speicher mit einem Zugriffskontrollbit versehen wurde, welches entweder dem Prozessor oder dem FPGA exklusive Zugriffsrechte einräumt. Sobald die JVM die Ausführung einer Hardware-Methode startet, setzt sie dieses Bit, um dem FPGA den Zugriff auf den Speicher zu gewähren. Greift die CPU in der Zeit, in der das Bit gesetzt ist, auf den gemeinsamen Speicher zu, so wird sie vorerst angehalten. Über die gesamte Dauer der Ausführung der Hardware-Methode verbleibt das Zugriffsrecht beim FPGA. Erst nachdem der FPGAs die Berechnung beendet hat, wird das Ergebnis in den gemeinsamen Speicher geschrieben und das Zugriffskontrollbit wieder zurückgesetzt, um somit das Zugriffsrecht für den gemeinsamen Speicher wieder an die CPU zu übergeben.

Diese Art der Verwaltung der Speichernutzungsrechte kann jedoch zu Performanceeinbußen führen. Je länger eine Hardware-Methode ausgeführt wird, desto höher ist das Risiko, dass die CPU aufgrund eines Ressourcenkonflikts angehalten werden muss. Man läuft Gefahr, dass die Zeit, in der CPU und FPGA parallel ausgelastet sind, deutlich sinkt. Um diese Einbußen zu verhindern, schlagen die Autoren vor, stets nur die Objekte, die durch die Hardware-Methode genutzt werden, in dem gemeinsamen Speicher unterzubringen. D.h. erst kurz vor dem Methodenaufruf sollen die Shared Objects angelegt werden. Direkt nach der Ausführung, so der Vorschlag, sollen sie wieder entfernt werden. Einerseits reduziert diese Lösung die Häufigkeit der Ressourcenkonflikte in Bezug auf den gemeinsamen Speicher drastisch, sodass die CPU besser ausgelastet wird. Andererseits steigt damit auch der Overhead der Methodenaufrufe, da die genutzten Objekte ständig zwischen Hauptspeicher und gemeinsamen Speicher kopiert werden müssen. In der vorliegenden Arbeit wurde daher eine alternative Möglichkeit zur Lösung des Synchronisationsproblems umgesetzt, die in Kapitel 4.5.4 beschrieben wird.

3.3.5 Ergebnisse

Um die Implementierung zu testen und zu analysieren, wurde eine Simulationsumgebung für das komplette Hardware/Software-System auf Basis des Befehlssatz-Simulators *Virtutech Simics* entwickelt. Als Betriebssystem wurde dabei ein Linux eingesetzt, dass mit einer modifizierten *Kilo Virtual Machine (KVM)* ausgestattet wurde. Die Kilo Virtual Machine (KVM) ist eine JVM, die von Sun speziell für ressourcenbeschränkte Umgebungen entwickelt wurde.

Für die Performance-Messungen wurden verschiedene Java-Benchmarks zunächst nur in Software, danach jedoch in der Simulationsumgebung mit dem FPGA ausgeführt. Dabei wurden die Ausführungszeiten gemessen und mit der jeweiligen Ausführungzeit in Software verglichen. Abbildung 3.3 zeigt die normalisierten Ausführungszeiten der einzelnen Benchmarks.

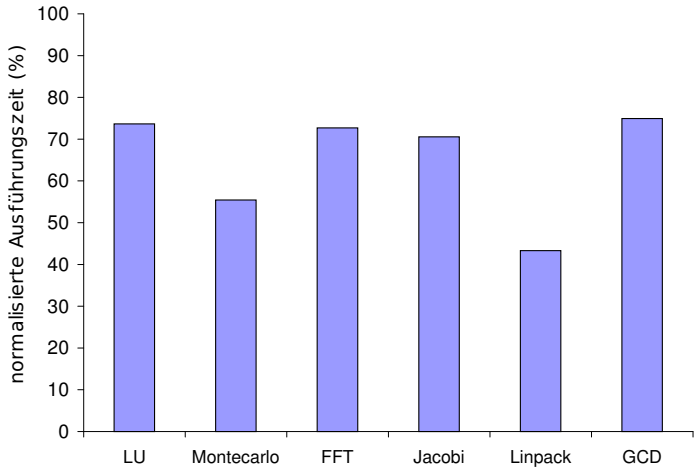


Abbildung 3.3: Ausführungszeiten der Benchmarks in Prozentangabe, um das Verhältnis zur Ausführungszeit in Software zu verdeutlichen [LGK+04]

Gegenüber der Software-Variante verkürzten sich die Ausführungszeiten im Hardware/Software-System demnach um etwa 20 bis 60 Prozent. Auch zeigte sich, dass selbst häufige Rekonfigurationen des FPGAs dennoch zur deutlichen Beschleunigung der Ausführung betragen können. Zu sehen ist dies am Benchmark *Linpack*, bei dem acht verschiedene Methoden (von 15 möglichen) in Hardware ausgeführt wurden. Im Benchmark *LU* hingegen wurde nur eine Methode (von 10 möglichen) für die Ausführung in Hardware eingesetzt.

3.4 Beschleunigung der dynamischen partiellen Rekonfiguration

Ein grundlegender Aspekt der Hardware-Tasks besteht in der gleichzeitigen bzw. quasiparallelen Ausführung mehrerer Tasks. Ziel ist es, in erster Linie so viele Tasks wie möglich parallel auf dem FPGA unterzubringen und auszuführen. Die FPGA-Fläche ist jedoch beschränkt und damit auch die Anzahl der Task-Slots. Übersteigt die Zahl der auszuführenden Tasks die Zahl der verfügbaren Slots, ist man auf eine quasiparallele Ausführung der Tasks angewiesen. Folglich wird ein Scheduler benötigt, der nach geeigneten Scheduling-Verfahren regelmäßig laufende Tasks durch wartende Tasks ersetzt, um jedem verschiedene Zeitscheiben für die Ausführung auf dem FPGA zur Verfügung zustellen. Echtzeit-Anwendungen wie die Audio- oder Video-Verarbeitung erfordern neben einer kontinuierlichen Verarbeitung der Daten zudem eine geringe Latenzzeit. Diese fällt desto geringer aus, je kürzer die Ausführungszeiten der Tasks sind und je früher ein Task-Wechsel stattfindet. Da mit jedem Task-Wechsel eine Rekonfiguration verbunden ist, sind kurze Rekonfigurationszeiten von großer Bedeutung.

In [Abe05] widmete sich Norbert Abel der Beschleunigung der dynamischen partiellen Rekonfiguration und entwickelte dazu eine Prototyp-Implementierung auf einem *Memec Virtex-II Pro Development Board* mit einem FPGA vom Typ *Xilinx Virtex-II Pro 7 FG456*. Durchgeführt wurde die Implementierung in zwei Phasen.

In der ersten Phase wurde der FPGA in eine Task- und eine System-Area eingeteilt und mit Hilfe von Busmacros eine Kommunikationsschnittstelle zwischen den beiden Bereichen geschaffen. Aufgrund des Platzmangels auf dem FPGA konnte nur ein Task-Slot realisiert werden. Für die Inter-Task-Kommunikation wurde die indirekte Variante unter Nutzung eines gemeinsamen Speichers gewählt. Als Speicher kam ein BRAM zum Einsatz, der in der System-Area platziert wurde. Eingeteilt wurde dieser, wie Abbildung 3.4 zeigt, in einen globalen und sieben lokale Speicherbereiche, wobei jedem Task ein unterschiedlicher lokaler Speicherbereich zugeordnet wurde. Jeder Task war in der Lage, auf seinen lokalen Bereich zuzugreifen, um beispielsweise interne Zustände zu sichern, aber auch Daten im globalen Speicher zum Zwecke der Inter-Task-Kommunikation abzulegen oder auszulesen.

Der eingesetzte FPGA war weiterhin mit einen PowerPC ausgestattet, welcher zur Ausführung eines C-Programms genutzt wurde. In der ersten Phase der Implementierung wurde das C-Programm unter anderem dazu eingesetzt, die Konfiguration der Task-Area über den internen Konfigurationsport (ICAP) auszulesen und

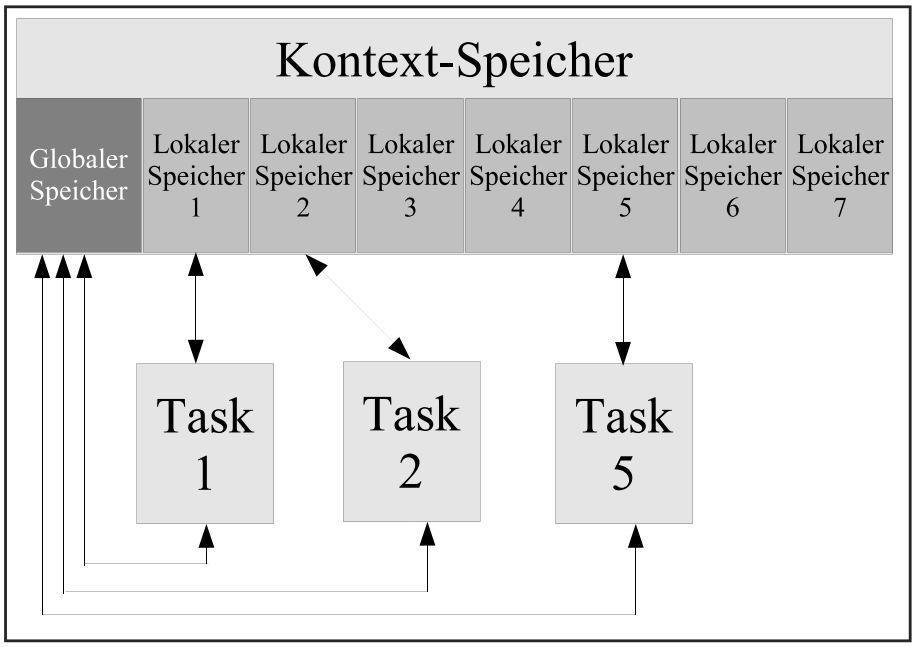


Abbildung 3.4: Aufteilung des BRAMs [Abe05]

zu schreiben. Die entsprechenden Bitstreams wurden im SDRAM auf dem Board abgelegt, bzw. zur langfristigen Speicherung zum PC übertragen und dort auf der Festplatte gespeichert. Abbildung 3.5 zeigt den Datenfluss eines Bitstreams während der Programmierung der Task-Area mit einem Task. Der PPC liest dabei den Bitstreams aus dem SDRAM und sendet ihn an den ICAP-Controller, der durch Xilinx entwickelt wurde und einen BRAM als Zwischenspeicher verwendet, bevor die Daten an das ICAP weitergereicht werden. Tests ergaben, dass auf diesem Wege, unter Nutzung des internen Konfigurationsports, eine deutliche Beschleunigung der Rekonfiguration gegenüber der externen Rekonfiguration über die parallele Schnittstelle zu erzielen war. Doch entsprachen diese Rekonfigurationszeiten noch nicht den gesetzten Zielen der Arbeit.

In der zweiten Phase lag daher die Beschleunigung der partiellen Rekonfiguration im Vordergrund. Ziel war es, den in Abbildung 3.5 dargestellten Weg der Bitstreams, der über den PPC und über einen BRAM-Zwischenspeicher führt, zu verkürzen. Zunächst wurde dazu ein neuer ICAP-Controller entworfen, der den Controller von Xilinx ersetzte. Weiterhin wurde eine neue Hardware-Komponente mit der Bezeichnung TMAN (Abkürzung für Task-Manager) implementiert und innerhalb der System-Area platziert. Diese wurde direkt mit dem SDRAM des Boardes und mit dem neuen ICAP-Controller verbunden und übernahm ab sofort die Aufgabe der Re-

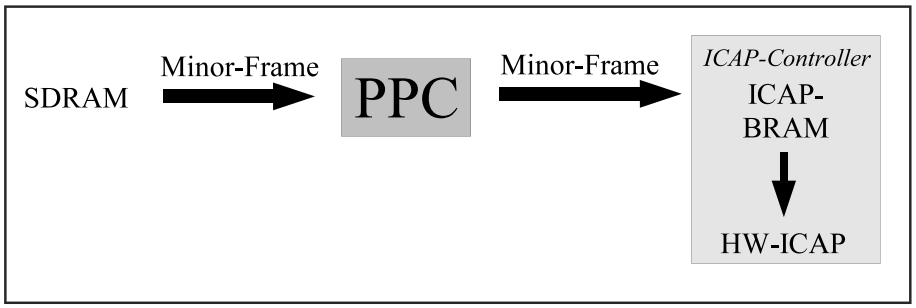


Abbildung 3.5: Datenfluss der Bitstreams in der Software-Lösung [Abe05]

konfiguration der Task-Area. Abbildung 3.6 verdeutlicht den neuen Datenfluss des Bitstreams unter Einsatz des TMANs.

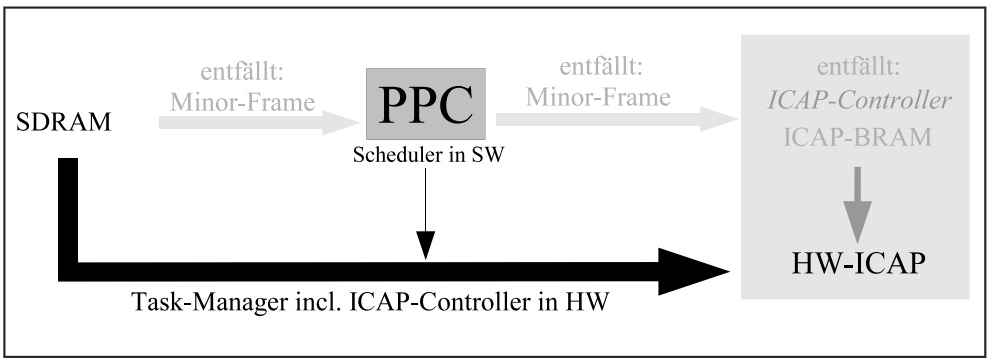


Abbildung 3.6: Datenfluss der Bitstreams in TMAN-Lösung [Abe05]

Im Vergleich zur Rekonfiguration mit Hilfe des PPCs konnte der TMAN die Rekonfiguration nochmals um ein Vielfaches beschleunigen. Messungen zeigten, dass für die komplette Rekonfiguration der 121 kB große Task-Area nur 5,6 Millisekunden benötigt wurden, was einem Datendurchsatz von 21,1 MB/s entspricht. Genauere Betrachtungen ergaben, dass diese Geschwindigkeit hauptsächlich durch den maximalen Datendurchsatz des SDRAMs (21,7 MB/s) beschränkt wurde.

In Phase zwei war der TMAN ebenfalls für die Kommunikation mit dem Task zuständig und übernahm neben der Rekonfiguration auch die Aufgabe des Startens und Stoppens der Tasks. Weiter war der TMAN dafür zuständig, die SDRAM-Zugriff des PPCs durchzustellen.

3.4.1 Portierung auf das XUP-Board

2007 führte Norbert Abel eine Portierung der in [Abe05] entwickelten TMAN-Komponente auf das XUP-Board durch, welche die Grundlage für diese Arbeit darstellt. Während der Portierung überarbeitete er das Design vollständig. Das Blockdiagramm in Abbildung 3.7 veranschaulicht das neue Hardware-Design des FPGAs.

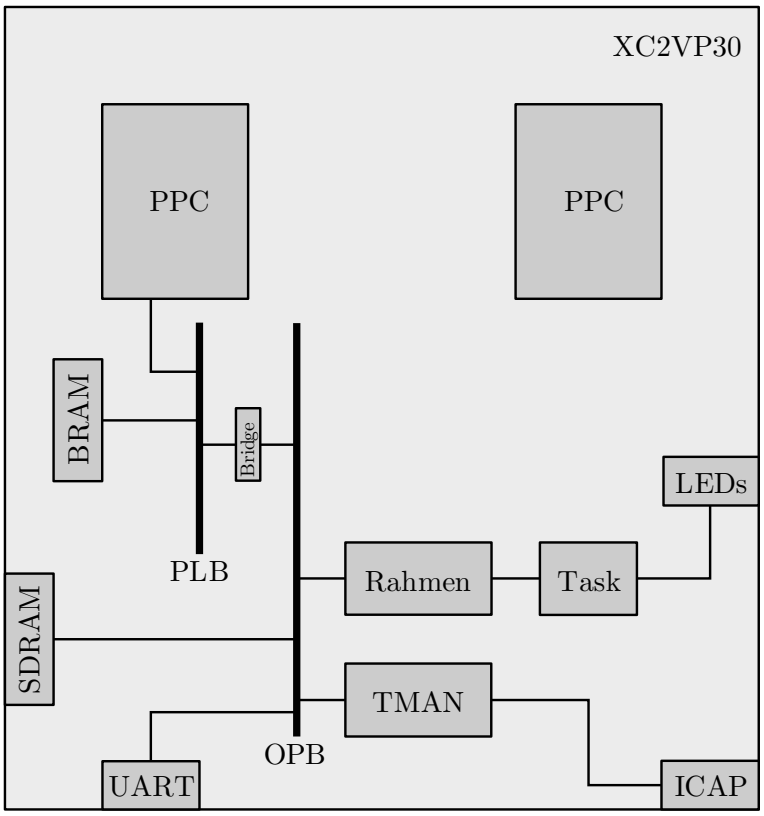


Abbildung 3.7: Hardware-Design nach der Portierung

Zu den grundlegenden Änderungen, die dabei vorgenommen wurden, zählt unter anderem die Trennung zwischen der Rekonfigurationslogik und der Logik der Task-System-Kommunikation. Während die Rekonfigurationslogik weiterhin in der TMAN-Komponente verbleibt, wurde der Teil, der für die System-Task-Kommunikation verantwortlich ist, in einer eigenständigen Hardware-Komponente, dem *Task-Rahmen*, untergebracht. Beide Komponenten sind mit dem OPB verbunden und lassen sich durch den PowerPC steuern. Eine zweite grundlegende Änderung betrifft die Anbindung des SDRAMs. Dieser ist nicht mehr direkt mit dem TMAN verbunden, sondern steht nun als OPB-Komponente dem PowerPC, dem TMAN und dem Task-Rahmen zur Verfügung.

Die Idee, die sich hinter der Trennung zwischen TMAN und Task-Rahmen verbirgt, ist eine modularere Verwaltung mehrerer Task-Slots. Ein Task-Rahmen verwaltet dabei genau einen Task-Slot und ist für das Starten und Stoppen des Tasks in diesem Slot verantwortlich. Um mehrere Tasks auf dem FPGA unterzubringen, müssen mehrere Task-Rahmen instantiiert werden. Zuvor ist auf dem FPGA entsprechender Platz für die Task-Slots zu schaffen.

Die Schnittstelle zwischen Task-Rahmen und Task wurde soweit ausgebaut, dass ein Task auf einen 256 Byte großen SDRAM-Bereich zugreifen kann. Zur Bestimmung der jeweiligen Adresse für den Zugriff verfügt der Task über einen 8-Bit-Adressbus. Die Daten auf diesem Bus bilden die unteren 8 Bit der SDRAM-Adresse. Die oberen 24 Bit der SDRAM-Adresse können für jeden Task-Rahmen durch den PPC festgelegt werden. Die Umsetzung der virtuellen Task-Adressen in SDRAM-Adressen entspricht daher einer Bündelung der 8 Adress- und 24 Registerleitungen.

Neben der Verbindung zum Task-Rahmen verfügt der Task auch über die Möglichkeit der Ansteuerung der vier LEDs auf dem XUP-Board. Des Weiteren ist mit dem OPB eine UART-Komponente verbunden, welche eine Kommunikation über die serielle Schnittstelle erlaubt. Neben dem OPB befindet sich auch der prozessornahe Bus *PLB* in dem Design. Beide Busse sind über eine Brücke miteinander verbunden. Am PLB angeschlossen sind der linke PowerPC und ein BRAM-Speicher, welcher für den PowerPC als Programm- und Datenspeicher dient.

3.4.2 Funktionsweise des neuen TMANs

Auch die Logik für die Rekonfiguration im TMAN wurde während der Portierung überarbeitet und flexibler gestaltet. Die neue Arbeitsweise orientiert sich an den drei Abschnitten, die Bestandteil jeder partiellen Rekonfiguration über das ICAP sind:

1. Die ICAP-Kommunikation beginnt mit der Übertragung einiger Synchronisationsworte. Anschließend werden die notwendigen ICAP-Register gesetzt, um die Rekonfiguration bzw. das Auslesen des Rekonfigurationsbereichs einzuleiten. Dabei werden dem ICAP unter anderem Informationen über den Startframe, die Anzahl der zu schreibenden oder lesenden Frames und die Art des Zugriffs (lesend oder schreibend) mitgeteilt.

2. Im zweiten Abschnitt werden die Frame-Inhalte (Bitstream) verarbeitet. Diese werden nun im Falle eines Schreibvorgangs an das ICAP übertragen oder im Falle eines Lesevorgangs vom ICAP empfangen.

3. Der Übertragungsvorgang wird abgeschlossen, indem bestimmte Desynchronisationsworte an das ICAP gesendet werden.

Die im Abschnitt zwei erwähnten Frame-Inhalte werden nach wie vor aufgrund ihrer Größe im SDRAM abgelegt. Die Daten aus den Abschnitten eins und drei definieren den zu beschreibenden FPGA-Bereich und werden aufgrund ihrer geringen Größe in einem BRAM gespeichert. Im günstigsten Fall müssen diese Daten durch den PowerPC nur einmal bei der Initialisierung des TMANs gesetzt werden und können anschließend die Kopf- (HEAD) und Fußinformationen (TAIL) für alle folgenden Rekonfigurationen bilden. Da auf den BRAM schneller als auf den SDRAM zugegriffen werden kann, entsteht dadurch sogar ein kleiner Geschwindigkeitsgewinn bei der Rekonfiguration. Abbildung 3.8 verdeutlicht noch einmal grafisch die Verbindung zwischen OPB, TMAN, dem BRAM des TMANs und dem ICAP.

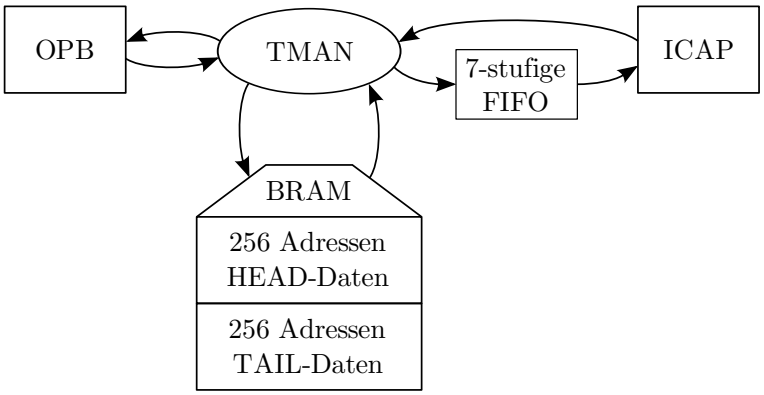


Abbildung 3.8: Aufbau des TMANs. Die Kreise symbolisieren endliche Automaten und die Rechtecke Komponenten außerhalb des Task-Rahmens. Die Pfeile verdeutlichen die Steuersignale und den Datenfluss zwischen den Komponenten.

3.5 Zusammenfassung

Zusammenfassend lässt sich sagen, dass sich bereits einige Forschungsgruppen mit dem Thema des Hardware/Software-Co-Designs unter Einsatz rekonfigurierbarer Hardware auseinandergesetzt haben. Viele der Arbeiten setzen dabei auf die Programmiersprache Java, da sie neben Vorteilen bei der Quellcode-Analyse und dem plattformunabhängigen Bytecode auch Möglichkeiten zur Manipulation der Ausführung einer Java-Anwendungen durch Anpassungen der Virtuellen Maschine bietet. Weiter ist erkennbar, dass mit der Entwicklung einer Hardware/Software-Co-Design-Umgebung

eine Vielzahl von Aufgaben verbunden ist, darunter die Übersetzung der Software- und Hardwarebeschreibungen in eine ausführbare Repräsentation, eine möglichst im Hintergrund ablaufende schnelle dynamische Rekonfiguration sowie eine Lösung für die Kommunikation zwischen Software und Hardware und die damit verbundene Synchronisation.

Die Arbeiten zeigen, dass es durchaus unterschiedliche Ansätze gibt, die genannten Aspekte umzusetzen. Im nächsten Kapitel soll der Lösungsansatz dieser Arbeit vorgestellt werden, der sich von den hier beschriebenen Forschungsarbeiten in folgenden Merkmalen unterscheidet: Zum einen sollen die Hardware-Implementierungen von Software-Teilen nicht auf Methodenebene sondern auf Thread-Ebene stattfinden. Die in Kapitel 2.4 beschriebenen Hardware-Tasks sollen dabei das Hardware-Äquivalent zu den Software-Threads bilden. Desweiteren gilt es, eine allgemeinere Lösung für das Synchronisationsproblem der Inter-Task-Kommunikation zu finden, sodass eine Sperrung des gemeinsamen Speichers nur bei Bedarf erfolgt, um die gleichzeitige Ausführung der Tasks so wenig wie möglich zu behindern.

Kapitel 4

Lösungsansatz

Nachdem im vorhergehenden Kapitel der Stand der Technik vorgestellt wurde, widmet sich dieses Kapitel nun dem Lösungsansatz der vorliegenden Arbeit. Ziel war die Schaffung eines Frameworks zur Realisierung von Hardware/Software-Tasks auf Basis von Java unter Einsatz von Threads. Dabei sollte das Thread-Konzept so erweitert werden, dass die sowohl in Software als auch in Hardware ausführbaren Tasks ähnlich wie Threads in einer Java-Anwendung instantiiert und gestartet werden können. Weiter sollte es möglich sein, für dynamisch erstellte Tasks auch noch zur Laufzeit den Ausführungsmodus jederzeit zwischen Software und Hardware wechseln zu können. Unabhängig von ihrem Ausführungsmodus sollten sich die Task stets wie Threads verhalten. Entwickler, die bereits Erfahrungen mit Threads gesammelt haben, sollten sich ohne größere Umstellung mit den so realisierten Hardware/Software-Tasks zurechtfinden. Das impliziert auch, dass die Eigenschaften und Methoden der Java-Threads übernommen werden. Sind diese Forderungen erfüllt, so lassen sich Threads in bestehenden Anwendungen leicht in äquivalente Tasks umformen.

Um die oben genannten Ziele umzusetzen, wurde eine Prototyp-Implementierung entwickelt. Die wesentlichen Aspekte des Frameworks und maßgeblichen Entscheidungen, die während der Entwicklung getroffen wurden, sollen nun in den folgenden Kapiteln beleuchtet werden. Zuerst widme ich mich der Architektur des Hardwaresystems. Darauf aufbauend wird die Inter-Task-Kommunikation genauer betrachtet. Anschließend werden die Java-Bestandteile des entwickelten Frameworks vorgestellt, die zur Verwaltung der Tasks und zur Kommunikation mit der rekonfigurierbaren Hardware erforderlich sind. Im Mittelpunkt stehen dabei Hardware/Software-Tasks. Beschrieben wird unter anderem, wie sie als Java-Objekte realisiert wurden und wie die Tasks untereinander Daten austauschen können. Weiterhin wird auf die Generierung der Task-Bitstreams und das Scheduling der Tasks eingegangen.

4.1 Architektur des Hardwaresystems

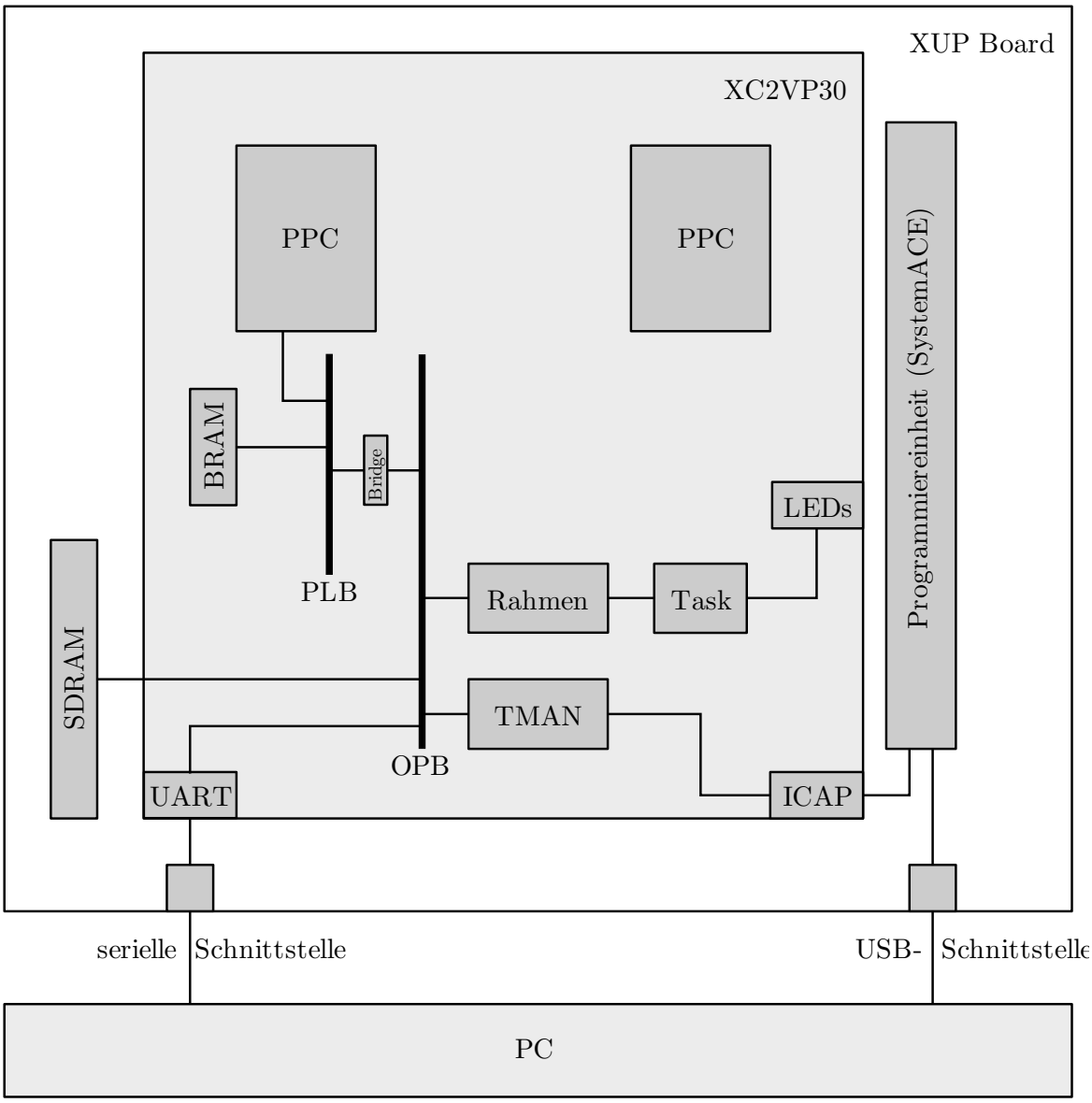


Abbildung 4.1: Komponenten des gesamten Systems

Wie in Abbildung 4.1 dargestellt, setzt sich das System aus einem PC und dem XUP-Board zusammen. Verbunden sind beide über die serielle und die USB-Schnittstelle. Über letztere wird das initiale Design auf den FPGA geladen. Der zu übertragende Bitstream des Designs enthält sowohl die Verdrahtung des FPGAs als auch den Inhalt der BRAMs, die mit dem PLB verbunden sind und für den PPC den Programm- und Datenspeicher bereitstellen. Nach dem Download wird ein Reset auf dem FPGA durchgeführt, sodass alle Hardware-Komponenten initialisiert werden

und ein C-Programm auf dem PowerPC zur Kommunikation mit dem PC gestartet wird. Danach wird die USB-Schnittstelle im Allgemeinen nicht mehr benötigt, da sich PC und Board nur über die serielle Schnittstelle verständigen. Die einzige Ausnahme stellt die Generierung der Bitstreams von Tasks dar. In diesem Fall wird das Board noch einmal von außen per USB konfiguriert.

Das Design, mit dem der FPGA programmiert wird, baut wesentlich auf dem in Kapitel 3.4.1 beschriebenen Entwurf von Norbert Abel auf. Um die Inter-Task-Kommunikation zu realisieren, mussten selbstverständlich der Task-Rahmen sowie der Task selbst angepasst werden. Änderungen an der TMAN-Komponente waren nicht erforderlich. Der rechte PowerPC wird in dieser Arbeit nicht benötigt und ist daher nicht mit dem PLB verbunden. Auf dem linken PPC läuft das erwähnte C-Programm, das auf Kommandos vom PC wartet, die über die serielle Schnittstelle gesendet werden. Die Steuerung der dynamischen Rekonfiguration und die Ausführung der Java-Anwendung werden vom PC übernommen.

4.2 Inter-Task-Kommunikation

Verschiedene Modelle der Inter-Task-Kommunikation wurden bereits in Kapitel 2.6.2 vorgestellt. Dabei wurde eine grundsätzliche Unterscheidung zwischen der direkten und indirekten Form getroffen. Es stellte sich heraus, dass die direkte Kommunikation zwar einen nicht unerheblichen Geschwindigkeitsvorteil gegenüber der indirekten besitzt, dabei aber gleichzeitig voraussetzt, dass zwischen den Tasks eine physikalische Verbindung besteht und die Tasks für die Kommunikation zeitgleich auf dem FPGA laufen müssen. Ein Informationsaustausch zwischen zwei Hardware-Tasks wäre daher mit nur einem Task-Slot nicht realisierbar. In der Wahl der Kommunikationsform fiel deshalb die Entscheidung auf die indirekte Kommunikation, bei welcher die Tasks mit einen gemeinsamen Speicher arbeiten, der für den Datenaustausch und die Zustandssicherung genutzt wird.

4.2.1 Gemeinsamer Speicher

Prinzipiell würden für den gemeinsamen Speicher sowohl der RAM des PCs, der SDRAM auf dem Board als auch der BRAM auf dem FPGA in Frage gekommen. Doch vor allem den Hardware-Tasks sollte ein hoher Datendurchsatz garantiert werden, um den Geschwindigkeitsvorteil der Hardware auch tatsächlich nutzen zu können. Die Speicherung der Daten auf dem PC erwies sich diesbezüglich als die ungünstigste Variante, da jeder Datenzugriff der Hardware-Tasks über die langsame

serielle Schnittstelle hätte laufen müssen. Für eine effiziente Nutzung der Hardware-Tasks blieb also nur die Möglichkeit, einen Speicher auf dem Board zu wählen. Ein weiterer wichtiger Punkt war die Speichergröße, zumal die verbleibenden Kandidaten hier deutliche Unterschiede aufweisen. Bis zu 2 GB SDRAM lassen sich auf dem XUP-Board nutzen. Ein BRAM-Block hingegen fasst nur etwa 2 KB[1]. Mit den 136 BRAMs auf dem Chip XC2VP30 ließen sich aber selbst mit Nutzung der Paritätsbits bestenfalls 306 KB an Daten speichern.[Xil05] Da jedoch ein Teil der BRAMs als Programm- und Datenspeicher der CPU dient und ein weiterer Teil innerhalb der Task-Area liegt, lässt sich realistisch gesehen nur ein BRAM-Speicher von etwa 100 KB realisieren. Um auch größere FIFO-Puffer im Bereich von mehreren Megabyte für die Inter-Task-Kommunikation verwenden zu können, fiel die Entscheidung deshalb zu Gunsten des SDRAMs aus. Durch die Anbindung an den OPB haben sowohl die PLB- als auch die OPB-Komponenten die Möglichkeit, auf den RAM zuzugreifen, sofern sie die Master-Schnittstelle des Busses implementieren. Zu diesen Komponenten gehören unter anderem der PPC und der Task-Rahmen.

RAM-Zugriffe durch den PowerPC können in C problemlos mit Hilfe von Zeigern implementiert werden. Um dem Java-Programm auf dem PC ebenfalls die Möglichkeit zu geben, Lese/Schreib-Operationen auf dem SDRAM des Boardes auszuführen, wurde der PowerPC für das Abhören der seriellen Schnittstelle eingesetzt. Seine Aufgabe war es, die Anfragen, die der PC sendet, in RAM-Zugriffe umsetzen und eine Antwort an den PC zurückzusenden. Abbildung 4.2a verdeutlicht den Datenfluss, der bei einer solchen SDRAM-Anfrage durch den PC entsteht. Grün dargestellt ist der Weg der Anfrage des PCs, rot der resultierende RAM-Zugriff durch den PowerPC.

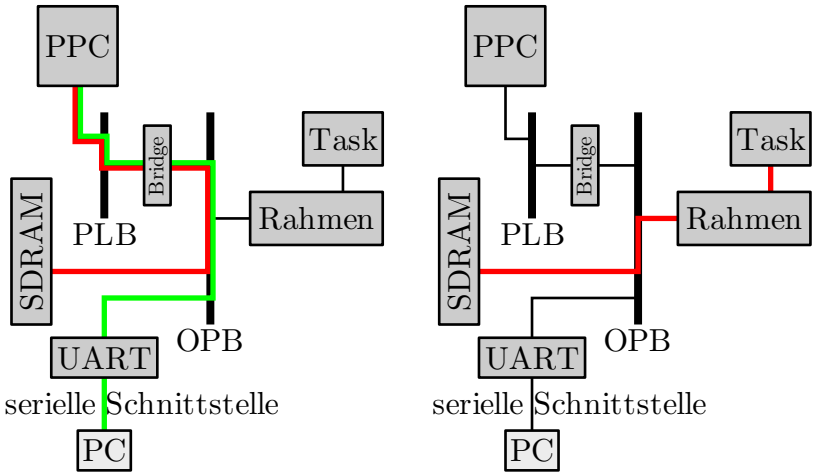
Zugriffe der Hardware-Tasks auf den SDRAM verlaufen dagegen sehr einfach. Auf Anfrage des Tasks stellt der Task-Rahmen die Verbindung zum OPB her und setzt die Lese/Schreib-Aktionen des Tasks in RAM-Zugriffe um. Neben dem OPB werden dazu keine weiteren Komponenten verwendet. Eine grafische Darstellung dieser Kommunikation ist in Abbildung 4.2b zu sehen.

4.2.2 Anforderungen an das Protokoll

Lesen und Schreiben von n aufeinanderfolgenden Bytes

Im Hinblick auf die Performance der Tasks spielt auch die Zeit, die für die Kommunikation aufgewendet wird, eine entscheidende Rolle. Ein Ziel musste deshalb sein, die

[1] Um genau zu sein, besteht ein BRAM aus 18×1024 Bit, da jedes Byte mit genau einem Paritätsbit versehen ist.[Xil07a]



(a) Software-Task (b) Hardware-Task

Abbildung 4.2: Zugriff auf den SDRAM

Kommunikation effektiv zu gestalten. Dazu zählt unter anderem ein möglichst kleiner Overhead für den Datenaustausch. Das käme vor allem der Übertragung über die serielle Schnittstelle zu Gute, denn hier kann eine Zeitersparnis im Bereich mehrerer Millisekunden pro Nachricht erzielt werden, wenn weniger Daten übertragen werden müssen.

Auch ist es sinnvoll, die Kommunikation zwischen Task und Task-Rahmen zu optimieren. Das Protokoll sollte so gestaltet werden, dass sich die Kommunikationslogik im Hardware-Task auf ein Minimum reduziert. Würde man dem Task das reine OPB-Protokoll für die Zugriffe auf den SDRAM weiterreichen, so müsste der Task für das Lesen und Schreiben größerer zusammenhängender Datenbereiche alle vier Bytes eine neue OPB-Anfrage stellen. Dazu wäre ein 32-Bit-Adress-Register im Task zu implementieren, welches nach jedem Zugriff um den Wert 4 inkrementiert werden muss. Genau diese Logik kann in den Task-Rahmen verlagert werden, indem man auch das Protokoll des Hardware-Tasks, wie schon oben beschrieben, auf die Übermittlung der Startadresse und der Länge der Daten beschränkt.

Da sich diese Überlegung sowohl aus Sicht der Software- als auch aus Sicht der Hardware-Tasks rentiert, wurde sie als Anforderung für die Inter-Task-Kommunikation aufgenommen.

Lesen und Schreiben aus bzw. in FIFO-Warteschlangen

Eine speziell für die Kommunikation zwischen Prozessen häufig eingesetzte Technik sind First-in-First-out (FIFO)-Warteschlangen, kurz FIFOs. Ihren Einsatz finden sie in Programmiersprachen und Betriebssystemen, wann immer große Datenströme kontinuierlich verarbeitet und an den nächsten Prozess/Thread weitergereicht werden müssen, da eine komplette Zwischenspeicherung eines Verarbeitungsschrittes möglicherweise zu viel Speicherplatz benötigen würde. In Software werden FIFOs durch einen Speicherbereich im RAM realisiert, der beim Schreiben in die FIFO gefüllt und beim Lesen aus der FIFO wieder Stück für Stück geleert wird. Ist die FIFO voll, wird der schreibende Prozess/Thread angehalten. Sobald Daten aus der FIFO wieder entnommen werden, läuft der schreibende Prozess weiter. Gleiches gilt für den lesenden Prozess, wenn die FIFO leer ist. Die Größe des FIFO-Puffers ist in der Regel fest gewählt und passt sich nicht dynamisch an.

Generell lassen sich solche FIFOs auch mit der oben beschriebenen Technik des Lesens und Schreibens von n aufeinanderfolgenden Bytes realisieren. Abgesehen von der ständig notwendigen Inkrementierung der Startadresse müssen die Schreib- und Leseindizes regelmäßig im gemeinsamen Speicher gesichert und zur Überprüfung des FIFO-Füllstandes wieder ausgelesen werden. Diese aufwendige Implementierung dem Hardware-Task zu überlassen, ließe sämtliche Ressourcen-Einsparungen an anderer Stelle unbedeutend erscheinen. Aus diesem Grund sollen FIFOs gesondert durch das Framework unterstützt werden.

4.2.3 Anforderungen an die Speicherverwaltung

Mit der Nutzung eines gemeinsamen Speichers entsteht auch der Bedarf nach einer Speicherverwaltung. Jedem Task müssen Speicherbereiche zugeordnet werden, die er für die Ablage lokaler Daten verwenden kann und von keinem anderen Task gelesen oder überschrieben werden. Weiterhin müssen Bereiche geschaffen werden, die zum Zwecke des Datenaustauschs durch mehrere Tasks genutzt werden können. Auch soll es möglich sein, mehrere Tasks des gleichen Typs zu instantiieren, die mit unterschiedlichen Daten arbeiten. Hart codierte Speicheradressen in Tasks sind damit inakzeptabel, denn hier würden alle Tasks einer Klasse stets auf den selben RAM-Bereich zugreifen. Aus diesem Grund müssen den Tasks zur Laufzeit Speicherbereiche zugeordnet werden. Dies erfordert wiederum eine Zwischenschicht, die eine Adressauflösung von Task-internen Adressen in SDRAM-Adressen ausführt. Für den Ort dieser Adressauflösung gibt es zwei Möglichkeiten:

(a) Eine Möglichkeit wäre, die Berechnung der Adresse auf Seiten des Tasks durchzuführen. Dazu müsste man den Tasks mitzuteilen, welche Speicherbereiche sie zu nutzen haben und sich darauf verlassen, dass diese Vorgaben eingehalten werden. Beliebige RAM-Zugriffe, auch außerhalb der zugewiesenen Bereiche, wären weiterhin möglich. Zudem würde die Berechnung der SDRAM-Adresse für Hardware-Tasks im Allgemeinen einen 32-Bit-Addierer erfordern und die freien Ressourcen des Task-Slots somit weiter einschränken.

(b) Eine andere Möglichkeit wäre, den Tasks das Recht auf beliebige Speicherzugriffe zu entziehen und ihnen stattdessen eine Datenaustausch-Schnittstelle zur Verfügung zu stellen, über die sie ihre Anfragen nur noch objektbezogen stellen können. Zugriffe würden nicht mehr über RAM-Adressen, sondern beispielsweise über Objektnummern laufen, die erst durch das Framework in SDRAM-Zugriffe umgesetzt werden. Mit der Implementierung der Adressauflösung im Framework könnte auch leicht ein Speicher-Zugriffsschutz realisiert und damit die Sicherheit des gesamten Systems gesteigert werden.

Angesichts des erhöhten Ressourcenverbrauchs von Variante a und des beschriebenen Sicherheitsaspekts ist die Variante b der Variante a vorzuziehen.

4.2.4 Anforderungen an die Synchronisation der Zugriffe

Kapitel 2.8.7 erläuterte bereits die Notwendigkeit eines wechselseitigen Ausschlusses zwischen Threads, die auf dieselbe Ressource zugreifen. Die gemeinsam genutzte Ressource der Tasks ist in diesem Fall der SDRAM. Auch hier müssen die Zugriffe koordiniert werden, um Inkonsistenzen zu vermeiden. Die Schwierigkeit dabei ist, dass die Synchronisation sowohl die Software- als auch die Hardware-Tasks betrifft.

4.3 Umsetzung im Framework

Um die genannten Anforderungen umzusetzen, wurde in dieser Arbeit ein Framework entwickelt, das die Implementierung von Hardware- und Software-Tasks ermöglicht, die in Java-Anwendungen ähnlich wie Threads eingesetzt werden können. Das Framework umfasst das initiale Design auf dem Board, das C-Programm auf dem linken PPC und einige Java-Klassen, mit deren Hilfe Java-Anwendungen mit Tasks realisiert werden können. Zunächst sollen die wesentlichen Java-Klassen kurz vorgestellt werden, um anhand dieser danach den allgemeinen Aufbau einer Task-Anwendung zu erläutern und eine Motivation für die Objektstruktur des Frameworks zu geben.

Anschließend werden die einzelnen Komponenten näher beschrieben. Zum Teil wird dabei auch auf einige Implementierungsdetails eingegangen, um die Hintergründe wesentlicher Design-Entscheidungen darzulegen. Die weiteren, tiefergehenden technischen Details folgen erst im Kapitel 5.

4.3.1 Übersicht über die Java-Klassen

Im Folgenden sollen die wesentlichen Java-Klassen kurz vorgestellt werden. Für einen besseren Überblick sind in Abbildung 4.3 die Beziehungen zwischen den einzelnen Klassen noch einmal grafisch dargestellt.

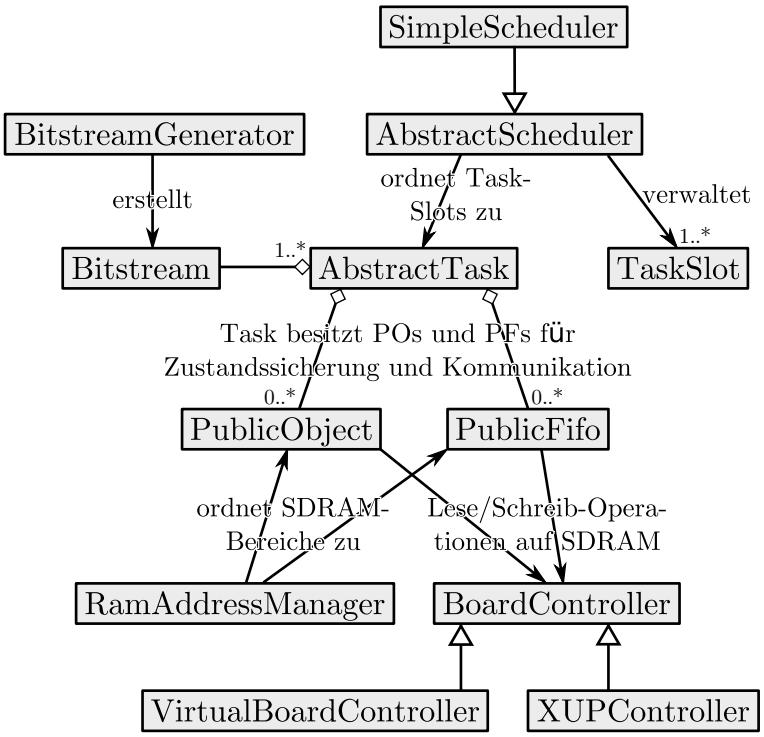


Abbildung 4.3: Die wesentlichen Java-Klassen des Frameworks

Wie bereits im Kapitel 4.2.2 (Anforderungen an das Protokoll der Inter-Task-Kommunikation) beschrieben, soll es möglich sein, zusammenhängende Speicherbereiche des SDRAMs in ihrer Gesamtheit als Block anzusprechen. Diese Blöcke werden im Framework durch Objekte vom Typ `PublicObject` repräsentiert. Der Name der Klasse ergibt sich daraus, dass die Speicherbereiche objektbezogen adressiert werden und dass PublicObjects (PO) grundsätzlich durch mehrere Tasks gemeinsam genutzt werden können. Die zweite Kommunikationsform der Tasks ist das Lesen und Schreiben aus bzw. in FIFOs. Zur Umsetzung dieses Konzeptes wurde die Klasse `PublicFi-`

`fo` geschaffen. Sowohl POs als auch PublicFifos (PF) bekommen nach ihrer Instantiierung dynamisch einen Speicherbereich im SDRAM zugeordnet. Diese Aufgabe übernimmt der `RamAddressManager`. Die Tasks selbst werden durch eine abstrakte Klasse namens `AbstractTask` repräsentiert. Um neue Tasks zu implementieren, müssen sie von `AbstractTask` abgeleitet werden und die `run`-Methode implementieren. Besitzt ein Task eine Hardware-Implementierung, so kann der zugehörige partielle Bitstream durch den `BitstreamGenerator` erzeugt werden. Die fertigen Bitstreams der Tasks sind durch die Klasse `Bitstream` vertreten.

Welcher Task zu welchem Zeitpunkt in Hardware auf dem FPGA oder in Software auf dem PC läuft, entscheidet der Task-Scheduler. Dieser wurde im Framework durch die abstrakte Klasse `AbstractScheduler` umgesetzt. Um den Task-Wechsel durchzuführen, muss der Scheduler auch Kenntnis über die Zustände der Task-Slots haben. Für diese wurde die Klasse `TaskSlot` geschaffen und die Verwaltung der TaskSlots im `AbstractScheduler` realisiert. Ein Scheduling-Algorithmus ist in der Klasse jedoch nicht implementiert. Für die Umsetzung eines konkreten Algorithmus' muss eine Scheduler-Klasse angelegt und von `AbstractScheduler` abgeleitet werden. Eine Beispiel-Implementierung bildet der `SimpleScheduler`, der im Kapitel 4.8 genauer beschrieben wird. Ähnlich flexibel wurde die Kommunikation mit dem Board gestaltet. Hier läuft jegliche Kommunikation über den `AbstractBoardController`. Auch diese Klasse wurde abstrakt definiert, um dem restlichen Java-Framework eine von der eingesetzten FPGA-Hardware abstrahierte API für den Zugriff auf das Board zur Verfügung zu stellen. So können zum Beispiel unterschiedliche Implementierungen des `AbstractBoardControllers` für verschiedene Boards entwickelt werden. Zu den bereits angefertigten Unterklassen des `AbstractBoardControllers` gehören der `VirtualBoardController` und der `XUPController`. Letzterer realisiert die Nachrichtenübertragung zum XUP-Board. Der `VirtualBoardController` hingegen simuliert nur die Verbindung zu einem Board mit FPGA und dient nur dem Testen von Task-Anwendungen ohne den Einsatz rekonfigurierbarer Hardware. Um zu Beginn einer Task-Anwendung sämtliche Framework-Einstellungen zu laden, wurde die Klasse `TaskFramework` angefertigt. Mit Hilfe dieser Klasse lassen sich sämtliche Parameter komfortabel aus einer Konfigurationsdatei einlesen.

4.4 Allgemeiner Aufbau einer Task-Anwendung

Jede Java-Anwendung, welche die Hardware/Software-Tasks nutzt, lässt sich grob in die folgenden Bereiche unterteilen:

1. Initialisierung des TaskFrameworks
   - baut die serielle Verbindung zum Board auf und testet die Kommunikation mittels Ping-Nachricht, die eine Pong-Antwort erwartet
   - erstellt den RamAddressManager
   - erstellt den Task-Scheduler
2. beliebiger Programmcode, geprägt durch
   - Erstellung von POs, PFs, Tasks
   - Starten und Stoppen von Tasks
   - Wechsel der Ausführung-Modi von Tasks
3. Stoppen aller Tasks am Ende der Anwendung
4. Herunterfahren des TaskFrameworks
   - schließt die serielle Verbindung zum Board

4.5 Realisierung der Inter-Task-Kommunikation

Wie bereits angesprochen bilden die PublicObjects und PublicFifos die Grundlage für die Inter-Task-Kommunikation. Beide Zugriffsformen wurden so gestaltet, dass sie den gestellten Anforderungen an das Protokoll nachkommen und einen effizienten Zugriff auf den SDRAM ermöglichen. Software- und Hardware-Tasks sollten sich dabei nicht mit einem direkten Zugriff auf den SDRAM auseinandersetzen, sondern über die durch das Framework bereitgestellten Schnittstellen objektbezogen auf den gemeinsamen Speicher zugreifen.

4.5.1 PublicObject

Die Software-Tasks nehmen ihre Zustandssicherung mittels Datentypen vor, die von `PublicObject` abgeleitet wurden und in den SDRAM abgebildet werden. Die Klasse `PublicObject` stellt den Unterklassen dabei die grundlegenden Methoden für den lese- und schreibseitigen Zugriff auf den jeweiligen SDRAM-Bereich und die Registrierung beim `RamAddressManager` zur Verfügung. Zur Veranschaulichung soll die Beispielimplementierung eines ganzzahligen Datentyps in Abbildung 4.4 dienen.

Der Datentyp speichert einen `short`-Wert (16-Bit) im SDRAM. Im Konstruktor muss die Größe des Datentyps in Bytes festgelegt werden und der Datentyp anschließend beim `RamAddressManager` registriert werden, um einen SDRAM-Speicherbereich

```
public class PublicShort extends PublicObject {
  public PublicShort() {
    length = 2;  // Groesse des Datentyps
    registerAtRamAddressManager();
  }

  public short getValue() throws InterruptedException {
    // Lesen der 2 Bytes aus dem SDRAM
    byte[] data = readDataFromRam();
    // Konvertierung der 2 Bytes in einen short-Wert
    return (short) ((0xff & data[0]) << 8 | (0xff & data[1]));
  }

  public void setValue(short value) throws InterruptedException {
    // Konvertierung des short-Wertes in 2 Bytes
    byte[] data = new byte[]{
        (byte) (value >>> 8 & 0xff),
        (byte) (value & 0xff)
    };
    // Schreiben der 2 Bytes in den SDRAM
    writeDataToRam(data);
  }
}
```

Abbildung 4.4: `PublicShort`: Implementierung eines ganzzahligen Datentyps als PublicObject

zugeordnet zu bekommen. Weiterhin muss der Datentyp dem Nutzer Methoden zum Setzen und Auslesen des Wertes zur Verfügung stellen. Bezogen auf das Beispiel sind dies die Methoden `getValue()` zum Lesen und `setValue()` zum Schreiben des Wertes. Um die Daten im SDRAM abzulegen, müssen sie in eine Byte-Folge gewandelt werden, die per `writeDataToRam()` in den entsprechenden SDRAM-Bereich geschrieben wird. Der Lesezugriff erfolgt umgekehrt. Zuerst wird der Speicherbereich per `readDataFromRam()` ausgelesen und die erhaltene Byte-Folge in den Datentyp gewandelt, der dem Nutzer zurückgeliefert werden soll. Die beiden Methoden `readDataFromRam()` und `writeDataToRam()` werden durch die Klasse `PublicObject` bereits bereitgestellt.

Nach der JavaBean-Spezifikation ist es üblich, die get- und set-Methoden für den Zugriff auf den Inhalt so zu wählen, dass sie namensgleich mit der zu lesenden oder zu schreibenden Eigenschaft sind. Im Beispiel hieß die Eigenschaft `value`. Beachtet man die Konvention, so sind die von `PublicObject` abgeleiteten Datentypen äußerlich mit den übrigen Datentypen im Java Runtime Environment (JRE) vergleichbar,

abgesehen von primitiven Typen wie int, short und float, die nicht durch Klassen realisiert sind. Aber auch zu diesen gibt es entsprechende Klassenrepräsentationen wie Integer, Short und Float, deren Wert über get- und set-Methoden verwaltet wird.

Für die Hardware-Task gestaltet sich der Zugriff auf PublicObjects etwas anders. Möchte ein Hardware-Task auf ein PO zugreifen, so übermittelt er dem Task-Rahmen die Objektnummer und die Art des Zugriffs. Die POs und PFs, auf die der Task zugreifen kann, werden in Form von zwei Listen im Java-Objekt des Tasks festgelegt. Die Nummerierung der POs und PFs ergibt sich aus der Position des Elements in der jeweiligen Liste. Direkt vor dem Start eines Hardware-Tasks werden alle Adress- und Längeninformationen der POs und die nötigen Informationen zu den PFs dem Task-Rahmen übermittelt, der die Verwaltung des Hardware-Tasks übernimmt. Dort werden die Informationen in einem BRAM abgelegt und bei einer Anfrage eines Tasks entsprechend ausgelesen und für den SDRAM-Zugriff verarbeitet. Dabei erfolgt die Berechnung der RAM-Adressen komplett im Task-Rahmen. Darunter fällt auch eine notwendige Inkrementierung der Adresse, falls das PublicObject aufgrund seiner Größe mehrere SDRAM-Zugriffe erfordert. Aus Sicht des Hardware-Tasks gestalten sich die Lese- und Schreiboperationen dementsprechend einfach. Nach dem Übermitteln der PO-Nummer kann der Task mit dem Senden bzw. Empfangen des Datenstroms beginnen. Auf die Details des Protokolls wird in Kapitel 5.4.5 eingegangen.

4.5.2 PublicFifo

Die gebräuchlichste Variante, eine FIFO zu realisieren, ist einen Speicherbereich zu reservieren und zwei Indizes anzulegen, die jeweils die Position für den nächsten Schreib- bzw. Lesezugriff in dem Puffer markieren. Initial sind beide Indizes am Beginn des FIFO-Bereiches positioniert. Werden Daten in die FIFO geschrieben, so werden sie an der Position des Schreibindexes abgelegt, wobei dieser fortlaufend inkrementiert wird. Beim Lesen aus der FIFO erfolgt eine Inkrementierung des Leseindexes mit gleichzeitigem Lesen von dieser RAM-Position. Erreicht ein Index dabei das Ende des FIFO-Bereiches, so wird er zum Anfang zurück gesetzt. Treffen Lese- und Schreibindex aufeinander, blockiert die FIFO, denn sie ist dann entweder voll oder leer. Die Indizes können bzw. dürfen sich also nicht überholen. Anderenfalls würden ungelesene Daten überschrieben oder ungültige Daten gelesen werden.

Implementiert wurde dieses Konzept in einer etwas abgewandelten Form. Im Blick lag dabei eine effizientere Speicherausnutzung, die dadurch erreicht wurde, dass mehrere FIFOs unter gewissen Voraussetzungen einen gemeinsamen Puffer-Bereich nutzen können. Gerade im Bereich der Audio- und Videoverarbeitung ist diese Voraus-

setzung häufig erfüllt, denn hier wird der Datenstrom in der Regel manipuliert ohne dabei das Datenvolumen zu vergrößern oder zu verkleinern. D.h. jede verarbeitende Einheit produziert gleichermaßen Daten wie sie konsumiert. Je länger die Kette der Verarbeitungsschritte ist, desto ineffizienter wäre eine Lösung, die pro FIFO einen eigenen FIFO-Bereich anlegen würde. Ausreichend wäre in diesem Fall, für alle FIFOs einen gemeinsamen FIFO-Bereich zu nutzen, der so organisiert ist, dass jeder Task die von ihm gelesenen Daten in dem Puffer-Bereich nach der Verarbeitung wieder überschreibt. Abbildung 4.5 illustriert diese Vorgehensweise für zwei FIFOs.

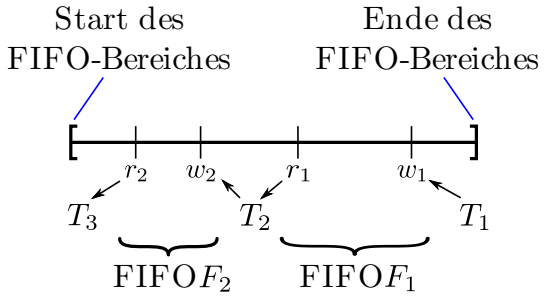


Abbildung 4.5: FIFO-Puffer mit mehreren FIFOs

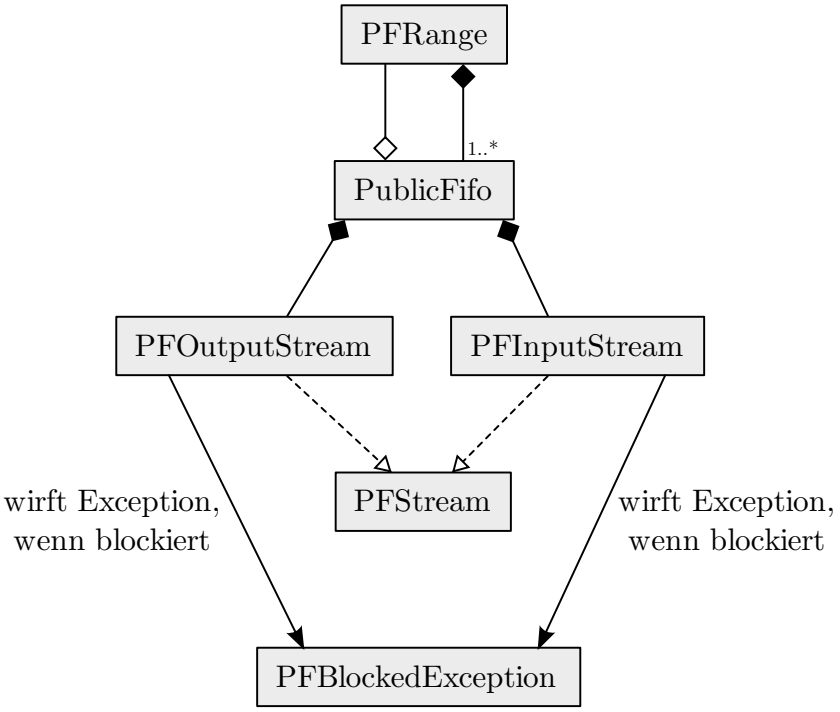
Zu sehen sind vier Indizes, wobei w_1 und w_2 die Schreibindizes von FIFO F_1 bzw. F_2 markieren, r_1 und r_2 die jeweiligen Lesenindizes. Task T_1 produziert ausschließlich Daten und füllt mit diesen die FIFO F_1. Task T_2 liest aus F_1 und schreibt das verarbeitete Ergebnis in FIFO F_2, das später durch den Task T_3 wieder ausgelesen und ausgegeben wird. Dabei nutzen die FIFOs F_1 und F_2 den gleichen FIFO-Bereich. Beachtet werden muss nur, dass die Schreibindizes nun bereits beim Leseindex der nächsten FIFO halten müssen.

Dieses Prinzip wurde auch in der `PublicFifo` umgesetzt und für die Verwaltung mehrerer PFs in einem FIFO-Bereich die Klasse `PublicFifoRange` angelegt, welche die Schreibindizes der FIFOs koordiniert. Die Indizes wurden durch PublicObjects realisiert, um sowohl der Software als auch der Hardware zur Verfügung zu stehen.

Die Lese- und Schreibzugriffe auf eine PublicFifo wurden, wie in Java üblich, über Input- und OutputStreams realisiert. Dazu dienen die Klassen `PublicFifoInputStream` und `PublicFifoOutputStream`. Sie stellen neben den normalen Stream-Funktionen noch Möglichkeiten zur Abfrage des Fifo-Füllstandes und andere Konfigurationsmöglichkeiten zur Verfügung. So lässt sich einstellen, wie die Streams während des Schreibens in eine volle FIFO oder während des Lesens aus einer leeren FIFO reagieren sollen. Zur Wahl stehen dabei zwei Möglichkeiten. In Variante eins wird in dieser Situation eine `PublicFifoBlockedException` geworfen. In Variante zwei wird

die Lese- bzw. Schreiboperation blockiert und kehrt erst wieder zurück, wenn sich der FIFO-Füllstand geändert hat und die Operation ausgeführt werden konnte.

Abbildung 4.6 zeigt die Zusammenhänge zwischen den FIFO-Klassen noch einmal in Form eines Klassendiagramms.



"PublicFifo" wurde in den Klassennamen durch "PF" abgekürzt

Abbildung 4.6: Klassen zur Umsetzung des FIFO-Konzeptes für Tasks

Um auch bei den FIFOs den Overhead für die Kommunikation zu reduzieren, werden die Daten der Lese- und Schreiboperationen lokal zwischengespeichert, um dann wie bei den PublicObjects in größeren Blöcken übertragen zu werden. Der entscheidende Gewinn wird hier jedoch nicht durch die kompakte Übertragung der FIFO-Inhalte erzielt, sondern bei der Aktualisierung der FIFO-Indizes. Diese müssen in der Regel vor und nach jedem FIFO-Zugriff mit dem SDRAM abgeglichen werden. Nur so lässt sich feststellen, ob sich der Füllstand mittlerweile geändert hat. Dabei gilt: Je größer die Übertragungsblöcke sind, desto seltener muss ein Abgleich der Indizes stattfinden. Festgelegt wird die Blockgröße einer PublicFifo bei deren Instantiierung.

Beachtet werden muss bei der Arbeit mit PFs, dass der Abgleich der Indizes nicht automatisch erfolgt. Der Grund dafür ist, dass selbst nach einer Unterbrechung eines Tasks alle FIFOs zusammen einen konsistenten Zustand bilden sollen. Das folgende

Beispiel erläutert die Problematik: Angenommen ein Task liest vier Bytes aus einer FIFO F_1, möchte diese nun verarbeiten und das Ergebnis in eine FIFO F_2 schreiben, wird aber zuvor vom Scheduler unterbrochen. Die Folge eines automatischen Abgleiches der Indizes wäre, dass die vier Bytes aus F_1 unweigerlich verloren gingen, denn der Task würde bei seinem nächsten Start mit dem Lesen hinter diesen unverarbeiteten Daten fortfahren. Es gibt verschiedene Möglichkeiten, dieses Problem zu umgehen. Eine wäre, den Bereich ab der Leseoperation bis zur Vollendung der Schreiboperation ununterbrechbar zu machen. Für die Hardware-Tasks würde das bedeuten, den OPB über diesen gesamten Zeitraum sperren zu müssen, also auch während der möglicherweise aufwendigen Verarbeitung der Daten. Die grundlegende Einigung, die Sperrung stets auf ein Minimum zu reduzieren, um anderen Tasks ebenfalls eine faire Nutzung des OPB zukommen zu lassen, wäre hier eindeutig verletzt. Deshalb wurde eine Lösung entwickelt, die den ununterbrechbaren Teil des FIFO-Tasks auf das Wesentliche, die Aktualisierung der FIFO-Indizes, reduziert. Um die Konsistenz der FIFO-Zustände zu wahren, müssen nur diese am Ende in einem atomaren Code-Stück in den RAM zurückgeschrieben werden. Entsprechende Methoden werden durch die FIFO-Streams zur Verfügung gestellt. Unterbrechungen vor oder nach diesem Bereich haben so keinen Datenverlust mehr zur Folge. Für Hardware-Tasks gestaltet sich das Zurückschreiben der Indizes sogar noch einfacher als bei den Software-Tasks. Nach jedem Zugriff auf eine PublicFifo speichert der Task-Rahmen den geänderten Index in einer internen Liste. Erhält er vom Task das Signal zum Zurückschreiben der Indizes, so werden die zwischengespeicherten Index-Werte in den SDRAM geschrieben.

4.5.3 Speicherfreigabe

PublicObjects und PublicFifos, auf die im Java-Programm keine Referenz mehr existiert, werden nicht mehr gebraucht und automatisch vom Garbage-Collector bei der nächsten Säuberung entfernt. Dabei wird in erster Linie der durch das Objekt im RAM des PCs belegte Speicher freigegeben. Um auch den reservierten SDRAM-Bereich freizugeben, wurde in beiden Klassen die `finalize`-Methode so implementiert, dass sich das Objekt beim `RamAddressManager` wieder abmeldet und somit auch den belegten SDRAM-Speicher freigibt.

4.5.4 Synchronisation der Zugriffe

Im Kapitel 3.3.4 wurde bereits das in der Arbeit [LGK+04] eingesetzte Synchronisationsverfahren beschrieben, welches einen wechselseitigen Ausschluss zwischen Software- und Hardware-Methoden umsetze. Dabei wurden die Zugriffsrechte auf den gemeinsamen Datenspeicher für die gesamte Dauer der Ausführung einer Hardware-Methode dem FPGA übertragen. Greift die CPU in diesem Zeitraum auf den gleichen Speicher zu, so wird sie bis zur Aufhebung der Sperrung angehalten, was zu Einbußen in der Performance führt. Effizienter wäre es, die Synchronisation nicht strikt auf Methoden-Ebene festzulegen, sondern den Tasks die Möglichkeit zu geben, selbstständig Sperrungen bei Bedarf anzufordern und wieder freizugeben. Dieses Prinzip wurde in der vorliegenden Arbeit umgesetzt. Die Tasks haben damit die Möglichkeit, die Sperrungszeiten auf ein Minimum zu reduzieren.

Um die Zugriffe aller Tasks auf den SDRAM zu koordinieren, reicht es leider nicht aus, die kritischen Abschnitte in Java-Tasks durch `synchonized`-Blöcke zu umschließen. Auch die Zugriffe der Hardware-Tasks müssen synchronisiert werden, und zwar auf einer gemeinsamen Ebene mit den Software-Tasks, um die Wechselbeziehungen zwischen beiden Task-Arten zu berücksichtigen. Eine gemeinsame Routing-Ressource beider Task-Arten bildet der OPB. Über ihn laufen alle SDRAM-Zugriffe. Zugleich werden die Zugriffe verschiedener OPB-Komponenten durch den Arbiter verwaltet und können so voneinander getrennt werden. Besitzt eine OPB-Komponente einmal die Masterrolle, kann sie den Bus durch das `busLock`-Signal (siehe Kapitel 2.7.2) auf unbestimmte Zeit sperren und damit den SDRAM in dieser Zeit exklusiv nutzen. Sind mehrere Task-Rahmen, von denen jeder nur einen Hardware-Task verwaltet, mit dem OPB verbunden, ließen sich bereits alle Zugriffe unter den Hardware-Tasks auf diese Weise abstimmen. Leider verfügt der PPC nicht über die Möglichkeit, den Bus zu sperren. Um auszuschließen, dass die Zugriffe von Software-Tasks durch Hardware-Tasks unterbrochen werden, musste deshalb eine spezielle Lösung für den PPC gefunden werden. Statt den OPB zu sperren, wurde im Task-Rahmen die Möglichkeit zur Unterdrückung der Task-Anfragen implementiert. Äußert ein Software-Task den Wunsch nach einem exklusiven Zugriff auf den SDRAM, schaltet der PPC die Unterdrückung in jedem Task-Rahmen ein und nach dem Zugriff wieder aus. Das Synchronisationsproblem zwischen Software- und Hardware-Tasks ist damit in beiden Richtungen gelöst und es verbleiben nur noch die SDRAM-Zugriffskonflikte zwischen Software-Tasks. Diese Angelegenheit lässt sich jedoch über die bereits im Kapitel 2.8.7 beschriebenen Mechanismen für den wechselseitigen Ausschlusses in Java lösen. Dazu wurde die Klasse `AbstractTask` mit den Methoden `lockBoardRam()`

und `unlockBoardRam()` ausgestattet. Erstere sperrt zu Beginn das SDRAM-Objekt, über das alle Zugriffe der POs und PFs erfolgen, und realisiert somit den wechselseitigen Ausschluss zwischen den Software-Tasks. Als zweites wird in der Methode die Unterdrückung der SDRAM-Zugriffe der Task-Rahmen aktiviert, um die Konflikte mit Hardware-Tasks auszuschließen. Demgegenüber gibt `unlockBoardRam()` das SDRAM-Objekt wieder frei und beendet die Zugriffsunterdrückung in den Task-Rahmen.

Die Tabelle in Abbildung 4.7 fasst noch einmal die vier zu lösenden Problemfälle beim SDRAM-Zugriff zusammen.

Zugriffskonflikt zw. Tasks		**Lösung des wechselseitigen Ausschlusses**
SDRAM-Besitz	**SDRAM-Wunsch**	
Software-Task	Software-Task	in Java mit Hilfe der Sperrung von Objekten
Software-Task	Hardware-Task	durch Aktivierung der Zugriffsunterdrückung in allen Task-Rahmen
Hardware-Task	Software-Task	durch Sperrung des OPB
Hardware-Task	Hardware-Task	durch Sperrung des OPB

Abbildung 4.7: Tabelle zur Lösung der SDRAM-Zugriffskonflikte zwischen den einzelnen Task-Formen

4.6 Task als Java-Objekt

4.6.1 Software- und Hardware-Verhalten

Um Task-Klassen möglichst einfach und thread-nah entwickeln zu können, wurde die Klasse `AbstractTask` geschaffen, die den von ihr abgeleiteten Task-Klassen das nötige Grundgerüst für ihre Arbeit bereitstellt. Um das Software-Verhalten eines Tasks festzulegen, muss dazu die `run`-Methode der Klasse implementiert werden. Die Festlegung des Hardware-Verhaltens erfolgt in einer getrennten VHDL-Beschreibung, die in der Task-Klasse über das `bitstream`-Attribute bekannt gemacht wird. Weiter müssen die durch den Task genutzten PublicObjects und PublicFifos im Konstruktor gesetzt werden, denn auch dem Task-Rahmen müssen diese Informationen bekannt sein, wenn der Task in Hardware ausgeführt wird. Damit ein Task direkt nach seiner Instantiierung ausführungsbereit ist, muss im Konstruktor die Anmeldung am `RamAddressManager` erfolgen. Dabei wird der Bitstream des Tasks von Festplatte gela-

den und in den SDRAM des Boardes geschrieben. Falls der Bitstream noch nicht existiert, wird er durch den `BitstreamGenerator` erzeugt. Welcher spezielle Bitstream geladen bzw. generiert wird, hängt auch von der Konfiguration des Task ab. Diese setzt sich aus den Größen der POs und den Blockgrößen der PFs sowie den benutzerdefinierten Task-Parametern zusammen. Die Werte der Parameter lassen sich in dem VHDL-Quellcode des Tasks auswerten, um das Task-Verhalten entsprechend anzupassen. Dabei handelt es sich gezielt um generische Parameter, die zur Synthesezeit festgelegt werden und somit nicht zu einer wachsenden Komplexität des Schaltkreises beitragen. Die Tatsache, dass die Parameter statisch festgelegt werden, stellt jedoch keine Einschränkung der Flexibilität von Tasks dar. So ließen sich beispielsweise auch PublicObjects variabler Größe im Hardware-Task realisieren, sofern man zu jedem PO ein weiteres mit dessen Größenangabe einsetzt. Letztendlich kann die Notwendigkeit einer solchen Lösung nur im Einzelfall entschieden werden und ist stets an die verfügbaren Task-Ressourcen gebunden. Mit den generischen Parametern wurde deshalb insbesondere der Zweck verfolgt, den Hardware-Tasks die Möglichkeit zu geben, sich statisch an die Gegebenheiten anzupassen. Über die benutzerdefinierten Task-Parameter lassen sich weitere Eigenschaften wie beispielsweise eine Ausführungsgeschwindigkeit statisch setzen. Jede neue Konfigurationen hat einen neuen Bitstream zur Folge, der jedoch erst bei Bedarf erzeugt wird. Die gesamte Bitstream-Verwaltung läuft für die Task-Entwickler verdeckt ab.

4.6.2 Die Task-Ausführungsmodi

Teile des Grundgerüstes vom `AbstractTask` wurden bereits in den letzten Kapiteln vorgestellt. Darunter die privaten Methoden für den wechselseitigen Ausschluss und für die Anmeldung am `RamAddressManager`. Der Ausführungsmodus eines Tasks hingegen sollte auch im Hauptprogramm geändert werden können. Diese Funktionalität wurde daher in der öffentlichen Methode `setMode()` bereit gestellt. Unterschieden werden die drei Modi: *Software*, *Hardware* und *Hardware-oder-Software (HoS)*. Ein Task im Software-Modus wird auf dem PC wie ein Thread ausgeführt. Das Scheduling mehrerer Software-Tasks übernimmt die JVM. Im Hardware-Modus erfolgt die Ausführung des Tasks auf dem FPGA. Falls sich mehr Tasks im Hardware-Modus befinden als Task-Slots vorhanden sind, kann nur ein Teil der Tasks gleichzeitig ausgeführt werden. Die Aufgabe des Schedulers übernimmt an dieser Stelle der `AbstractScheduler`. Durch ihn werden ebenfalls die Tasks im Hardware-oder-Software (HoS)-Modus verwaltet, der angibt, dass der Task bevorzugt in Hardware auszuführen ist. Nur wenn alle Task-Slots belegt sind, wird der Task in Software abgearbeitet.

Der Wechsel zwischen verschiedenen Ausführungsmodi ist so realisiert, dass zunächst die aktive Ausführung unterbrochen wird, um den Task danach im neuen Modus wieder zu starten. Dieser Wechsel läuft im Task intern ab, sodass der Task während des Zustandsübergangs äußerlich zu keinem Zeitpunkt als gestoppt gilt. Abbildung 4.8 verdeutlicht dazu die wesentlichen Taskzustände und die Zustandsübergänge. Wird die `interrupt`-Methode eines laufenden Tasks aufgerufen, so findet eine externe Unterbrechung statt und der Task verlässt den Zyklus, der für den Modi-Wechsel verantwortlich ist, und stoppt. Durch einen erneuten Aufruf von `start()` lässt sich der Task wieder starten.

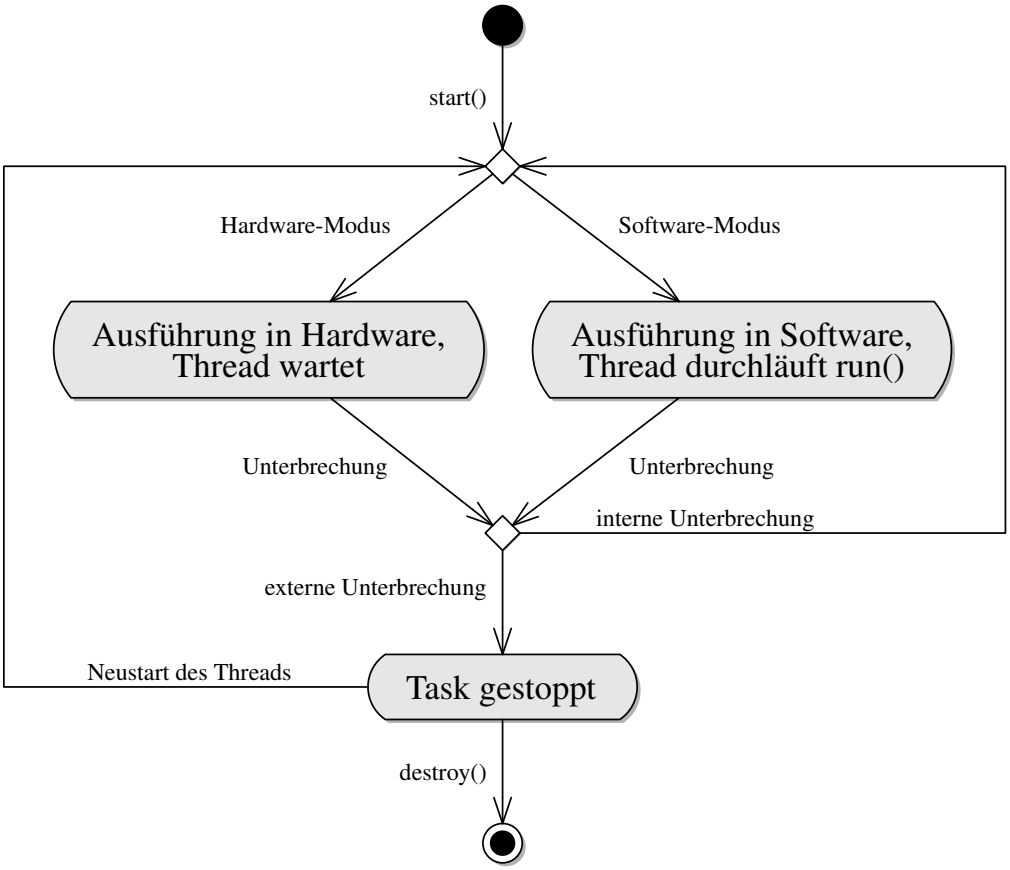


Abbildung 4.8: Die wesentlichen Task-Zustände

4.6.3 Thread vs. RestartableThread

Ursprünglich sollte die Klasse `AbstractTask` von `java.lang.Thread` abgeleitet werden, um Tasks komplett wie Threads behandeln können. So wäre es ein Leichtes gewesen, die Threads in bestehenden Anwendungen durch Tasks zu ersetzen. Leider

ließen sich mit dieser Vererbung nicht alle Anforderungen wie gewünscht umsetzen. Das Hauptproblem bestand darin, eine benutzerfreundliche Lösung für das mehrfache Starten und Stoppen eines Tasks zu finden. Der normale Java-Thread unterstützt diese Funktionalität nicht. Eine Thread-Instanz lässt sich höchstens einmal starten. Die Erweiterung des Threads um die Möglichkeit des erneuten Startens wäre ein klassischer Fall für den Einsatz von Vererbung. Es würde ausreichen, die `start`-Methode zu überschreiben und entsprechend anzupassen. Unglücklicherweise sind diesem Vorgehen Schranken gesetzt, da sämtliche Thread-Methoden, darunter auch `start()`, als `final` definiert sind und sich folglich nicht überschreiben lassen. Die Verwendung eines anderen Methodennamens zum Starten von Tasks könnte zwar das Problem lösen, doch die nun verbotene, ursprüngliche `start`-Methode stünde den Anwendungsentwicklern weiterhin zur Verfügung und würde stets eine potentielle Fehlerquelle bedeuten. Desweiteren würde das Umlernen vom Thread-Konzept zum Task-Konzept und die Umstellung bestehender Projekte erschwert. Letztendlich wiederspricht die Abweichung von den Thread-Methoden bei gleichzeitiger Ableitung von `Thread` dem Prinzip der Vererbung. Zusammenfassend sind die Nachteile dieser Lösung so gravierend, dass nach anderen Wegen gesucht werden musste.

Die übliche Form, einen Thread in Java neu zu starten, besteht in der Instantiierung eines neuen Threads. Soll die Berechnung des neuen Threads auf den Daten des alten aufbauen, so müssen die entsprechenden Attribute von der alten Instanz zur neuen kopiert werden. Ein weiteres Problem bleibt jedoch unberücksichtigt. Nicht selten werden in einer Anwendung an verschiedenen Stellen Referenzen auf Threads gehalten. Nach einem Neustart eines Threads ist es in den meisten Fällen erwünscht, dass die Referenz auf den logisch gesehenen Thread erhalten bleibt, der nun lediglich durch eine neue Instanz repräsentiert wird. Daher müssten alle alten Referenzen entsprechend der neuen Instanz aktualisiert werden, was sich als höchst aufwendig herausstellen kann. Letztenendes ist diese Lösung keinesfalls optimal, da eine zu große Verantwortung durch den Gebrauch mit neuen Thread-Objekten in die Hände der Anwendungsentwickler gelegt wird.

Da also die Versuche, `AbstractTask` von `Thread` abzuleiten, zu keinem zufrieden stellenden Ergebnis führten, wurde auf diese Vererbung verzichtet und der Fokus darauf gerichtet, zumindest die Signatur[2] der Klasse `Thread` im `AbstractTask` zu übernehmen. Realisiert wurde dies zunächst in der Klasse `RestartableThread`, die

[2]Die Signatur einer Klasse definiert die Methodennamen der Klasse zusammen mit ihren Parametern. Zwei Klassen können zwar die gleiche Signatur besitzen, müssen deshalb aber laut Klassenhierarchie nicht zwangsläufig die gleiche Schnittstelle implementieren. Demgegenüber vereinbaren Schnittstellen (sogenannte Interfaces) jedoch gemeinsame Signaturen.

die `Thread`-Klasse komplett nachempfindet und zugleich das mehrfache Starten und Stoppen ermöglicht. Von dieser wurde danach der `AbstractTask` abgeleitet. Im Ergebnis verfügen damit auch die Tasks über die Funktionalität des Neustartens und besitzen zugleich auch die Methoden eines normalen Threads.

4.7 Generierung der partiellen Bitstreams

Sobald ein Task erstellt wird, wird überprüft, ob ein Bitstream zu dem Task mit der gegebenen Konfiguration vorliegt. Falls nicht, erstellt der `BitstreamGenerator` den zugehörigen parametrisierten Bitstream. Die Möglichkeit einer Parametrisierung von Task-Komponenten ist sinnvoll, um für kleinere Änderungen, z.B. bei der Geschwindigkeit eines Zählers, nicht extra eine komplett neue Task-Klasse erstellen zu müssen. Solche einfachen Anpassungen lassen sich sowohl in Java als auch in VHDL leicht realisieren.

In VHDL werden Parameter dieser Art im `generic`-Block der Hardware-Komponente festgelegt. Diese Möglichkeit wurde auch in den VHDL-Dateien der Tasks genutzt. Dort lassen sich Parameter für die Größe eines POs und die Blockgrößen von PFs sowie weitere benutzerdefinierte Parameter auswerten. Die Werte dieser Parameter werden durch den BitstreamGenerator beim Erstellen des Bitstreams gesetzt und hängen von der Konfiguration des Tasks in Java ab. Weiterhin ist es möglich, auch in VHDL beim SDRAM-Zugriff Variablen für die POs und PFs statt deren Nummern zu verwenden. Die Zuordnung der Nummern zu den Variablen übernimmt ebenfalls der `BitstreamGenerator`.

Die Generierung des partiellen Bitstreams erfolgt in mehreren Schritten. Zuerst wird mit dem EDK das gesamte System mit dem parametrisierten Task synthetisiert und über die USB-Schnittstelle auf den FPGA geladen. Danach wird der TMAN auf dem Board angewiesen, den Hardware-Task via ICAP auszulesen und in den RAM zu schreiben. Dieser wird, mit einer einfachen Komprimierung (siehe Kapitel 5.5) versehen, an den PC übertragen und dort komprimiert auf der Festplatte abgelegt. Die Parameter des Hardware-Tasks werden dabei im Dateinamen des Bitfiles untergebracht.

4.8 Task-Scheduler

Der Task-Scheduler hat die Aufgabe, die verfügbaren Task-Slots zu verwalten und entscheidet, wann welcher Hardware-Tasks in welchem Slot laufen darf. Das Grund-

gerüst für diese Funktionalität stellt der `AbstractScheduler` bereit. Von diesem abgeleitet wurde mit der Klasse `SimpleScheduler` ein prioritätengesteuerter Round-Robin-Scheduler implementiert. Dabei genießen die Hardware-Tasks eine höhere Priorität als HoS-Tasks, da letztere auch in Software ausgeführt werden können.

		wartender Task	
		Hardware-Task	HoS-Task
Task-Slot enthält	keinen Task	1	4
	HoS-Task	2	5
	Hardware-Task	3	-

Abbildung 4.9: Reihenfolge der Task-Ersetzung im `SimpleScheduler`. Die Tabelle enthält die Prioritäten für die Auswahl des wartenden Tasks und die Suche nach einem geeigneten Task-Slot für die Ausführung des Tasks.

Die Tabelle in Abbildung 4.9 verdeutlicht die Reihenfolge, nach der der `SimpleScheduler` periodisch die wartenden Tasks auf die Task-Slots verteilt und dabei auch einen Task-Wechsel vornimmt, falls der Slot bereits belegt ist. Zuerst werden die wartenden Hardware-Tasks verarbeitet. Für jeden dieser Tasks wird zunächst nach einem freien Task-Slot gesucht, in dem der Task ausgeführt werden kann. Wenn kein freier Slot gefunden werden kann, wird nach einem Slot gesucht, in dem ein HoS-Task läuft und ein Task-Wechsel durchgeführt. Dabei versetzt der Scheduler den HoS-Tasks in den Software-Ausführungsmodus und programmiert anschließend den freigewordenen Task-Slot mit dem wartenden Hardware-Task. Stehen auch keine Task-Slots mit HoS-Tasks mehr zur Verfügung, so wird nach einem bereits laufenden Hardware-Tasks gesucht und dieser durch den wartenden Hardware-Task ersetzt. Sobald es keine wartenden Hardware-Tasks gibt, werden die wartenden HoS-Tasks für die Ausführung in Hardware berücksichtigt. Dabei wird wieder zuerst nach freien Task-Slots gesucht bevor die bereits laufenden HoS-Tasks durch die wartendenen ersetzt werden. Die Verdrängung von Hardware-Tasks durch HoS-Tasks wird ausgeschlossen.

4.9 Zusammenfassung

Abschließen lässt sich sagen, dass sich die Ziele der Arbeit mit einigen Kompromissen umsetzen ließen. Im entwickelten Framework lassen sich Tasks ähnlich wie Java-Threads programmieren und in Anwendungen einsetzen. Im Gegensatz zu den

Threads können die Tasks sowohl in Software als auch in Hardware ausgeführt werden. Das ursprünglich angestrebte Ziel, dass die Tasks die Eigenschaften der Java-Threads erben, um sie ebenfalls vom Typ `Thread` deklarieren zu können, erwies sich letztlich als sehr benutzerunfreundlich, da es nicht möglich gewesen wäre, Tasks neu zu starten. Als Kompromiss wurde deshalb auf die Ableitung von Thread verzichtet, die Thread-Signatur jedoch vollständig übernommen und das Verhalten der Threads in den Tasks nachempfunden. Ebenfalls implementiert wurde die Möglichkeit, den Ausführungsmodus eines Tasks zur Laufzeit zu ändern.

Um die Inter-Task-Kommunikation speziell auch zwischen Hardware- und Software-Tasks umzusetzen, wurde ein gemeinsamer Speicher eingerichtet. Damit verbunden musste ein wechselseitiger Ausschluss beim Zugriff auf den gemeinsamen Speicher realisiert werden. Gelöst wurde das Problem durch softwareseitige Sperrmechanismen und durch die Möglichkeit zur Sperrung des On-Chip Peripheral Busses. Trotz einiger Optimierungen in der Kommunikation zwischen Board und PC bleibt die serielle Schnittstelle der Flaschenhals des Systems. Deshalb erbringt das Framework in dieser Form eher einen Beweis des Möglichen. Bei Einsatz einer schnelleren Schnittstelle wie beispielsweise USB oder Ethernet wäre auch eine praktische Anwendung denkbar.

Kapitel 5

Implementierung

In diesem Kapitel sollen nun die Details der Implementierung erläutert werden. Zuerst wird das EDK-Projekt zur Entwicklung des Hardware-Designs vorgestellt. Dabei wird die Erstellung von eigenen Busmacros, die Schnittstelle zwischen Task-Rahmen und Hardware-Task und die Funktionsweise des Task-Rahmens beschrieben. Danach werden Details des Java-Frameworks angesprochen, darunter die Realisierung der PC-Board-Kommunikation, die Speicherverwaltung für den SDRAM und eine genauere Betrachtung der Klasse `AbstractTask`. Der letzte Teil des Kapitels widmet sich der Simulation von Tasks, automatisierten Tests und Werkzeugen zur Fehleranalyse.

Zunächst soll jedoch die Verzeichnisstruktur der Implementierung kurz vorgestellt werden.

5.1 Verzeichnisstruktur

Abbildung 5.1 gibt einen Überblick über die wesentlichen Verzeichnisse der Implementierung. Ausführlichere Darstellungen erfolgen in dem Kapitel zum jeweiligen Thema. Im Anhang A wird die strukturelle Gliederung umfassend bis auf Dateiebene beschrieben.

5.2 Das EDK-Projekt

Das Hardware-Design des Frameworks baut auf dem EDK-Projekt von Norbert Abel auf, das für die Portierung des TMAN auf das XUP-Board entworfen wurde, vgl. Kapitel 3.4.1. Um ein solches Projekt von Grund auf neu zu erstellen, kann man den Base System Builder im Xilinx Platform Studio verwenden. Mit diesem kann das

Verzeichnis bzw. Datei	Beschreibung
`conf/`	Konfigurationsdateien
`edkProject/`	EDK-Projekt-Verzeichnis
`javadoc/`	Dokumentation der Java-Klassen im JavaDoc-Format
`lib/`	Verwendete Java-Bibliotheken
`scripts/`	Scripte mit Java-Code, die im BeanShellWindow geladen werden können
`src/`	Java-Quellen
`task/`	Quellen und Bitstreams der Tasks
`bitstream/`	Bitstreams der Tasks
`design/`	FPGA-Designs Tasks
`sim/`	Simulationsdaten der Tasks
`source/`	VHDL-Quelltexte der Tasks
`waveform/`	GTKWave-Diagramme für Darstellung der Signaländerungen während der VHDL-Simulation der Tasks

Abbildung 5.1: Die wichtigsten Verzeichnisse der Implementierung

Grundsystem sehr einfach und schnell aus einer Reihe vorgefertigter IP-Cores zusammengestellt werden. Im Resultat entsteht ein EDK-Projekt, das mit einem Makefile ausgestattet ist und sich komfortabel per Kommandozeile übersetzen lässt. Mittels

```
make download
```

werden das gesamte Hardware-Design und die Software für den PowerPC übersetzt und zusammen in ein Bitfile geschrieben, welches anschließend zur Programmierung des FPGAs genutzt wird. Sobald die Übertragung fertig ist, wird auf dem Board ein Reset ausgeführt und damit auch die Anwendung auf dem PPC gestartet.

Nachträgliche Anpassungen an dem Projekt sind im XPS weiterhin möglich. So können jederzeit neue IP-Cores in das Design integriert oder bereits bestehende wieder entfernt werden. Auch lassen sich verschiedene Parameter und die Adressen der IP-Cores setzen. Weiterhin können mit Hilfe des XPS im Menü über *Hardware* → *Geräte or Import Peripheral* neue IP-Cores erstellt werden. Dazu muss zunächst entschieden werden, ob die Komponente an den PLB oder den OPB angeschlossen werden soll, da sich die beiden Schnittstellen unterscheiden. Als nächstes muss die Wahl getroffen werden, ob der neue IP-Core nur als Slave oder auch als Master agieren soll. Eine Slave-Komponente ist nur in der Lage, auf Anfragen, die über den OPB gestellt werden, zu antworten. Soll die Komponente selbst aktive Anfragen auf den

OPB legen können, so muss sie auch die Master-Schnittstelle implementieren. Nach der Beantwortung der Fragen generiert das XPS die nötigen Dateien, darunter auch die VHDL-Datei der Komponente, die bis auf die Entity-Definition noch leer ist. In dieser muss nun das Verhalten der Bus-Komponente festgelegt werden. Der neu erstellte IP-Core wird automatisch in die Liste der verfügbaren IP-Cores aufgenommen und kann dem Hardware-Design hinzugefügt werden. Auf diese Art und Weise wurden die Komponenten TMAN und Task-Rahmen entwickelt und in das Projekt integriert. Sie implementieren beide die Master- und die Slave-Rolle des OPB und haben somit Zugriff auf den SDRAM und können auf Befehle des PPCs reagieren.

Abbildung 5.2 zeigt das Projekt der Diplomarbeit im XPS. Auf der linken Seite sind die verfügbaren IP-Cores zu sehen. Die rechte Seite zeigt die im Projekt instantiierten Komponenten, zusammen mit den Informationen an welchem Bus sie angeschlossen sind.

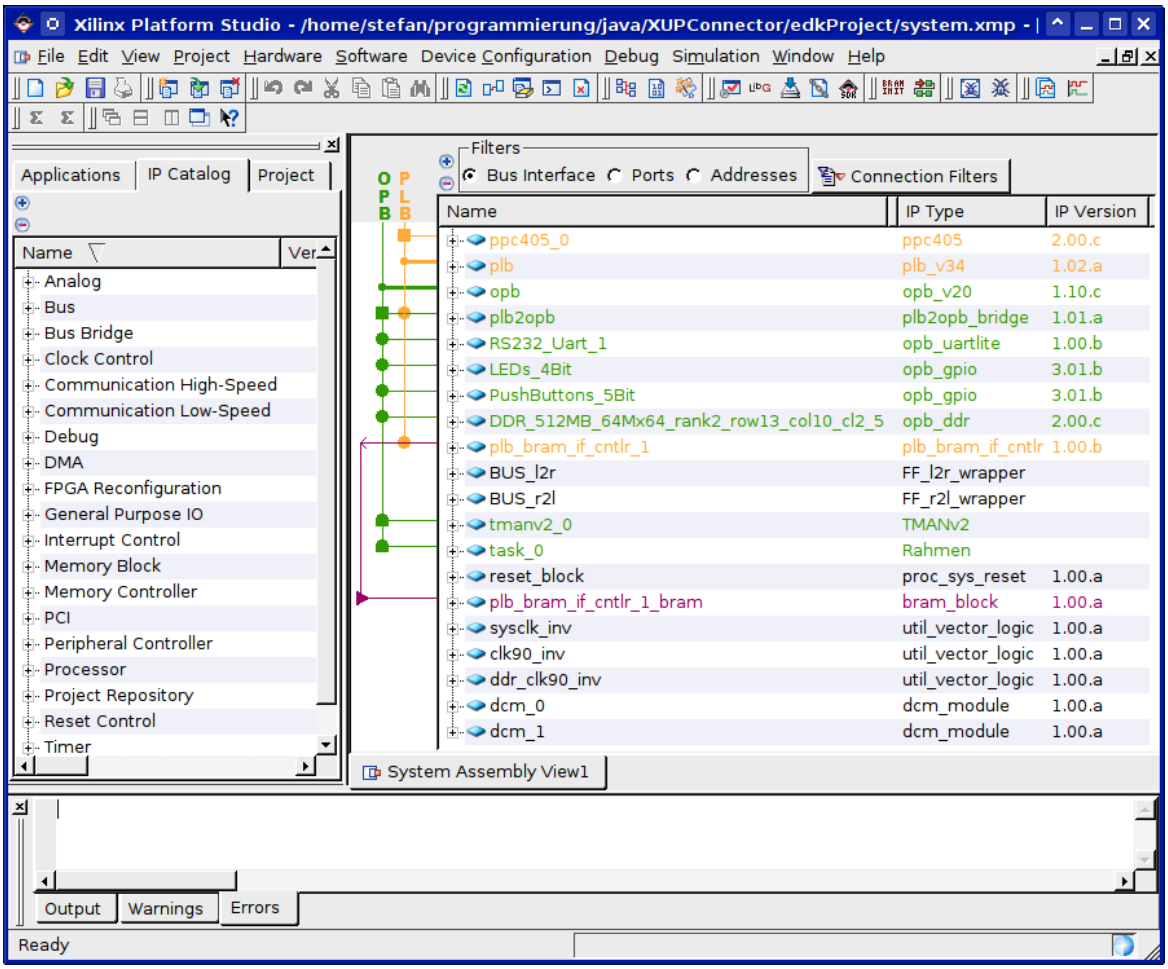


Abbildung 5.2: Screenshot vom EDK-Projekt im XPS

In den weiteren Erläuterungen wird auf einige Dateien des EDK-Projekts verwiesen, die im Laufe dieser Arbeit angepasst oder angelegt wurden. Abbildung 5.3 stellt die wesentlichen Dateien kurz vor und weist auf ihren Standort hin.

Verzeichnis bzw. Datei	**Beschreibung**
`edkProject/`	EDK-Projekt-Verzeichnis
`AppTman/src/`	C-Quellen des PowerPC-Programms
`AppTman_sw.c`	PowerPC-Programm
`icap_comm.c`	Bibliothek zur Ansteuerung des TMANs
`mystdio.c`	Bibliothek zur Kommunikation über die serielle Schnittstelle
`data/system.ucf`	Festlegung der externen Pins und der System- und Task-Area, Platzierung der Busmacros
`pcores/Rahmen/hdl/vhdl/`	VHDL-Dateien des Task-Rahmens und Tasks
`Rahmen.vhd`	VHDL-Code des Task-Rahmens
`TASK.vhd`	VHDL-Code des Tasks
`download.sh`	Script für die Synthese und Übertragung des Bitstreams auf das Board
`settings.sh`	Script zum Laden der Xilinx-Umgebungsvariablen

Abbildung 5.3: Die wichtigsten Dateien und Verzeichnisse im EDK-Projekt

5.3 Partitionierung des FPGAs

Eine Aufteilung des Chips in System- und Task-Area wurde bereits durch Norbert Abel während seiner Portierungsarbeiten vorgenommen. Ohne große Änderungen wurde die Partitionierung in dieser Diplomarbeit übernommen. Lediglich die Task-Area wurde vergrößert. Abbildung 5.4 zeigt die aktuelle Aufteilung in System- und Task-Area sowie die Lage der Ein- und Ausgangspins wesentlicher Komponenten.

Die Task-Area teilt die System-Area in einen linken und rechten Abschnitt. Der rechte Teil enthält keinerlei Berechnungslogik, sondern nur Verdrahtungslogik. Im linken Teil werden alle Hardware-Komponenten bis auf den Task platziert. Dieser wiederum wird in der Task-Area abgelegt.

In der Abbildung ist auch erkennbar, dass die Pins der LEDs, des ICAPs und des UARTs in der rechten System-Area liegen. Die Verbindungen zwischen TMAN

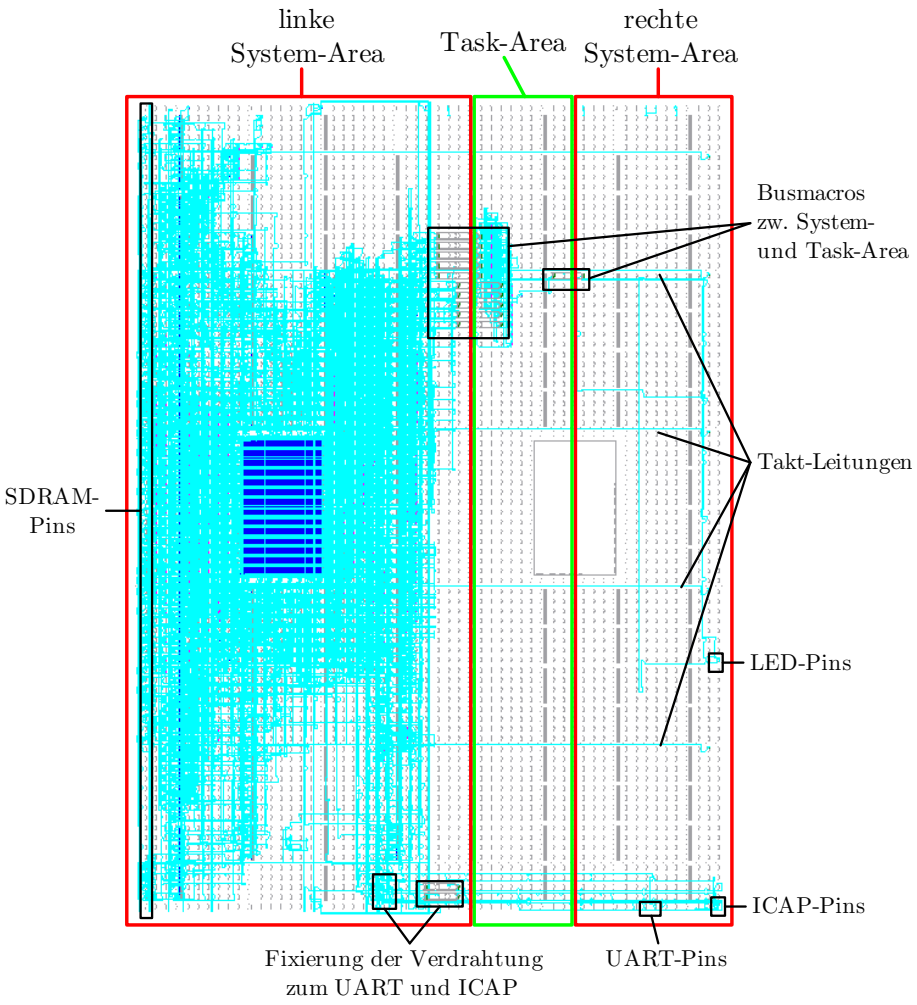


Abbildung 5.4: Chip-Aufteilung in System- und Task-Area

und ICAP sowie zwischen PowerPC und UART führen folglich über die Task-Area und sind von der Rekonfiguration betroffen. Um eine Unterbrechung der Verbindung während einer Rekonfiguration zu verhindern, wurden diese Signalleitungen in der Datei `system.ucf` fixiert. Die Start- und Endpunkte der Leitungen müssen ebenfalls durch fest positionierte Slices oder Busmacros realisiert werden.

Weiterhin enthält die `system.ucf` Informationen über die Platzierung von Komponenten sowie die Belegung der Pins am Rande des Chips und kann dazu genutzt werden, mehrere Komponenten in `Area Groups` (zu Deutsch Flächengruppen) zusammenzufassen, denen Flächen auf dem FPGA zugeordnet werden.[Xila]

Im EDK-Projekt wurde der Chip durch die Zeilen

```
INST "*" AREA_GROUP = system;
INST "task_0/task_0/task0/*" AREA_GROUP = task;
```

in eine System-Area und eine Task-Area eingeteilt. Um die Ausdehnung einer Gruppe auf einen bestimmten Chip-Bereich zu beschränken, kann das `RANGE`-Attribut der Gruppe wie im folgenden Beispiel gesetzt werden:

```
AREA_GROUP "system" RANGE = SLICE_X0Y0:SLICE_X43Y159;
AREA_GROUP "task" RANGE = SLICE_X54Y0:SLICE_X65Y159;
```

Ein solcher Bereich kann aus einem oder mehreren Rechtecken bestehen, die jeweils durch zwei Slice-Positionen definiert werden. Bei der Platzierung werden die Komponenten der Gruppe ausschließlich innerhalb des definierten Bereiches angeordnet. Bei der Verdrahtung hingegen werden die Bereichsangaben leider nicht beachtet und so kann es dazu kommen, dass Signalleitungen zwischen zwei Komponenten der gleichen Gruppe die Grenze der Area-Group kreuzen. Ist dies für Hardware-Tasks der Fall, so ist das Design unbrauchbar, denn nach einer Rekonfiguration des Task-Bereiches wären diese Signalleitungen höchstwahrscheinlich defekt. Laut [Xil08] lässt sich das Problem im EDK 7.1 durch das Setzen des `MODE`-Attributs wie folgt lösen:

```
AREA_GROUP "task" MODE=RECONFIG;
```

Doch leider traten auch mit dieser Angabe weiterhin Signalgrenzüberschreitungen auf. Auch die Parameter älterer EDK-Versionen, wie

```
AREA_GROUP "task" GROUP = CLOSED;
AREA_GROUP "task" PLACE = CLOSED;
AREA_GROUP "task" IMPLEMENT = AUTO;
AREA_GROUP "task" ROUTE_AREA = FIXED;
AREA_GROUP "task" ROUTE_AREA = RECONFIG;
AREA_GROUP "task" DISALLOW_BOUNDARY_CROSSING;
```

führten zu keinem Erfolg. Deshalb kann bei der automatischen Generierung der Bitstreams nicht komplett ausgeschlossen werden, dass die Verdrahtungsressourcen über die vorgesehene Fläche hinausgehen. Das Hardware-Design eines jeden Tasks muss daher manuell mit dem FPGA-Editor auf Verdrahtungsfehler überprüft werden. Die nötigen Design-Dateien werden hierfür nach einer Generierung im Verzeichnis `task/design/`*`task_name/`* abgelegt.

5.4 Realisierung der System-Task-Kommunikation

Für die Überbrückung der Grenze zwischen System- und Task-Area kamen Busmacros zum Einsatz. Xilinx stellt für jeden Chip-Typ einige vorgefertigte Busmacros

bereit. Hardmacros können jedoch auch selbst erstellt werden, um speziellen Anforderungen zu genügen. Das nächste Kapitel geht auf die Erstellung eigener Busmacros genauer ein und erläutert Probleme und Lösungswege, die speziell mit der Verwendung von TBUFs verbunden waren.

5.4.1 Erstellung eigener Busmacros

Die Busmacros von Xilinx müssen generell im Xilinx internen NMC-Format vorliegen, damit sie während der Synthese als Hardmacro verarbeitet werden. Das NMC-Format ist leider binär und folglich ungeeignet für die automatische Erzeugung von Busmacros mit Hilfe eines Scripts oder Programms. Aus diesem Grund bietet Xilinx zusätzlich ein ASCII-basiertes Format für den Austausch von Hardmacros an. In diesem werden die Macros in der sogenannten *Xilinx Design Language (XDL)* beschrieben. Mit dem gleichnamigen Programm (xdl) lassen sich Macros zwischen dem NMC- und Xilinx Design Language (XDL)-Format konvertieren.

Zur Editierung von Hardmacros im NMC-Format kann der grafische FPGA-Editor verwendet werden. Zunächst müssen hierfür die nötigen Komponenten (TBUFs oder Slices) einzeln ausgewählt und per *Edit→Add* hinzufügt werden. Ein Verdrahtungsweg zwischen zwei oder mehr Komponenten kann Stück für Stück mit der Maus bei gedrückter Shift-Taste markiert werden. Die so ausgewählte Verbindung wird anschließend mit *Tools→Route→Manual Route* zum Macro hinzugefügt. Um Ein- und Ausgänge des Macros festzulegen, müssen die entsprechenden Pins der TBUFs bzw. Slices selektiert und mittels *Edit→Add Hard Macro External Pin* mit einem Namen versehen werden.

Laut Xilinx können die so entstandenen Hardmacros ohne weitere Anpassungen in VHDL als Blackbox-Komponenten eingesetzt werden. Die Realität zeigte jedoch, dass die externen Pins der Macros bei der Synthese nicht gefunden wurden. Mit Hilfe des von Jens Thorvinger entwickelten Programms zur Erstellung von Busmacros ([Tho04]) konnte ermittelt werden, dass das durch den FPGA-Editor erzeugte NMC-Format fehlerhaft war.

Zur Lösung des Problems wurde die NMC-Datei zunächst in eine XDL-Datei gewandelt, die nötigen Änderungen an der Textdatei vorgenommen und anschließend die XDL-Datei wieder in eine NMC-Datei konvertiert. Die Formatwandlung erfolgt dann mittels

```
xdl -ncd2xdl bus.nmc
```

bzw. die Rückwandlung durch

```
xdl -xdl2ncd bus.xdl
```

In der XDL-Datei muss für jede TBUF-Instanz die Zeile

```
cfg "␣IINV::#OFF␣TINV::#OFF␣"
```

durch die Zeile

```
cfg "TINV::T␣IINV::I␣_SUPERBEL::TRUE"
```

ersetzt werden. Nach der Überführung der XDL-Datei in das NMC-Format wurden die externen Pins des Macros nun während der Synthese des Hardware-Designs gefunden.

5.4.2 Task-Area und ihre Busmacros

Letztendlich wurden in dieser Arbeit keine bidirektionalen Busmacros auf Basis von TBUFs eingesetzt, sondern die von Xilinx stammenden unidirektionalen Busmacros. Abbildung 5.5 zeigt im Detail die Anordnung der Macros, über die der Task mit dem System kommuniziert. Weiter ist in der Grafik deutlich zu erkennen, dass auch BRAM-Blöcke in der Task-Area liegen. Links von diesen befindet sich eine *BRAM Interconnect*-Spalte. Der restliche Teil der Task-Area besteht aus normalen *CLB*-Spalten. Bedeutung haben diese unterschiedlichen Spaltentypen vor allem für die Rekonfiguration, die in Kapitel 5.5 näher beschrieben wird.

Als nächstes sollen der Task-Rahmen und anschließend die Ein- und Ausgänge des Tasks beschrieben werden.

5.4.3 Task-Rahmen

Der Task-Rahmen bildet das Bindeglied zwischen OPB und Task.

Wie in Abbildung 5.6 dargestellt, setzt sich der Task-Rahmen aus zwei endlichen Automaten (engl. *Finite state machine (FSM)*) und einem BRAM-Block zusammen. In dem BRAM werden die Eigenschaften der POs und PFs gespeichert, die für den Zugriff auf den SDRAM benötigt werden. Darin nicht enthalten sind allerdings die FIFO-Indizes, die stets im SDRAM abgelegt werden, um sie auch anderen Tasks zur Verfügung zu stellen. Statt diesen werden im BRAM nur die Pointer auf die Indizes gespeichert. Nachfolgend soll die Aufteilung des BRAMs beschrieben werden.

Der BRAM besitzt einen 9-Bit-Adressbus und stellt somit 512 verschiedene Adressen zur Verfügung, durch welche jeweils 4 Bytes angesprochen werden. Von den 512 Adressen sind die ersten 256 für PO-Daten und die darauf folgenden 128 für die PF-Daten reserviert, wobei jeweils ein PO zwei Adressen und eine PF vier Adressen be-

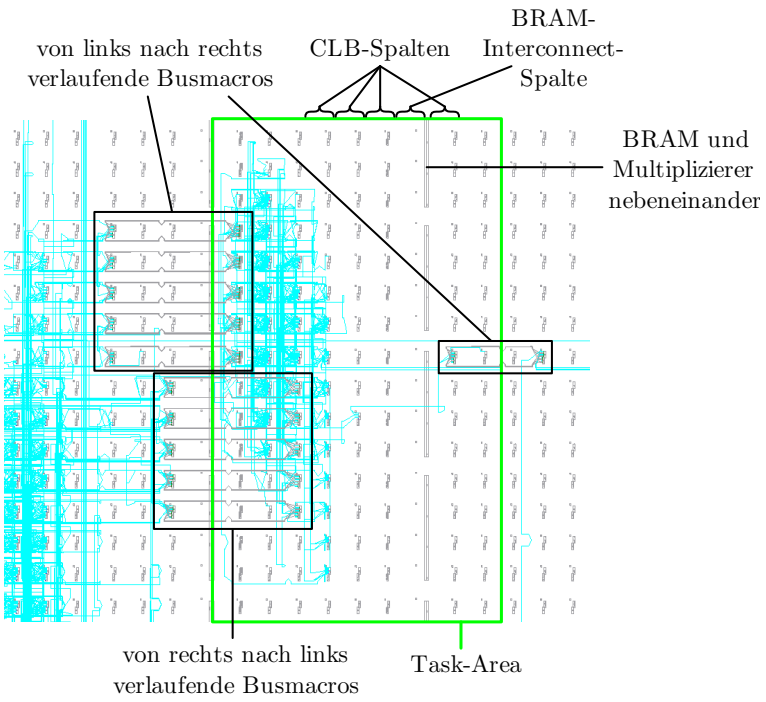


Abbildung 5.5: Busmacros für die Kommunikation zwischen System und Task

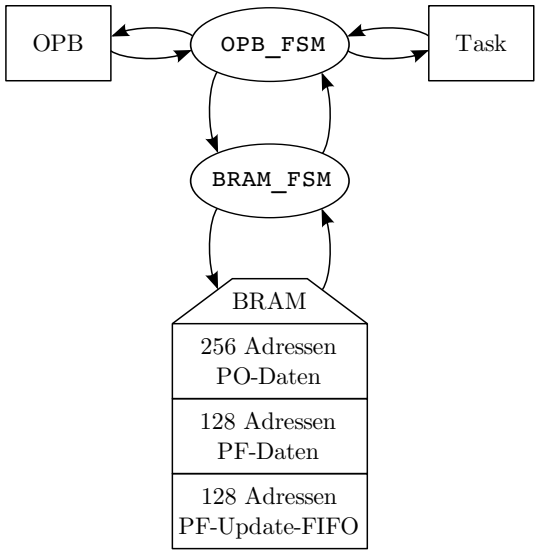


Abbildung 5.6: Aufbau des Task-Rahmens. Die Kreise symbolisieren endliche Automaten und die Rechtecke Komponenten außerhalb des Task-Rahmens. Die Pfeile verdeutlichen Steuersignale zwischen den Komponenten.

legt. Folglich kann ein Task maximal 128 verschiedene POs und 32 verschiedene PFs verwenden. Die verbleibenden 128 Adressen des BRAMs werden zur Realisierung der

PF-Update-FIFO (PUF) genutzt. In dieser Warteschlange landen die aktualisierten FIFO-Indizes nach jedem Zugriff auf eine PublicFifo. Dort werden sie zwischengespeichert, bis der Task über das `UPDATE_PFs`-Signal den Task-Rahmen zum Zurückschreiben der Indizes in den SDRAM veranlasst. Dabei wird die PUF komplett ausgelesen und jeder Eintrag, bestehend aus dem aktualisierten Index-Wert und der Adresse im SDRAM, an die vorgesehene Stelle im SDRAM geschrieben. Da jeder Eintrag also 2 BRAM-Adressen belegt, muss nach spätestens 64 PF-Zugriffen eine Aktualisierung der Indizes durch den Task eingeleitet werden, um einen Überlauf des FIFO-Puffers zu verhindern.

Angesteuert wird der BRAM durch die `BRAM_FSM`. Sie ist außerdem für die Aufbereitung der aktuellen PO/PF-Datenregister zuständig und teilt der `OPB_FSM` mit, wann die Register einsatzbereit sind. Die `OPB_FSM` stellt das Herzstück des Task-Rahmens dar. Sie nimmt Anfragen des OPBs und des Tasks entgegen und verarbeitet diese. Sie ist ebenfalls für die Zugriffe auf den SDRAM verantwortlich und steuert die `BRAM_FSM`.

Über das Schreiben in bestimmte Register des Task-Rahmens lässt sich der Task starten und stoppen, die SDRAM-Zugriffe des Tasks deaktivieren oder PO/PF-Eigenschaften setzen, die im BRAM abgelegt werden. Eine ausführliche Beschreibung der Task-Rahmen-Register ist im Anhang C zu finden.

5.4.4 Schnittstelle des Hardware-Tasks

Die Kenntnis über die Schnittstelle der Hardware-Tasks ist entscheidend für die Implementierung von Tasks. Abbildung 5.7 zeigt schematisch die Ein- und Ausgänge des Hardware-Tasks, wobei die auf der linken Seite angeordneten Ports mit dem Task-Rahmen verbunden sind. Die Ein- und Ausgänge auf der rechten Seite führen hingegen ohne weitere Verarbeitung zu den jeweiligen Pins am rechten Rande des Chips. Darunter befindet sich auch der `LEDs`-Bus, über den der Task die vier LEDs auf dem Board ansteuern kann.

Bis auf das Takt-Signal (`CLK`) führen alle angegebenen Ein- und Ausgänge über eines der 8 Bit breiten unidirektionalen Busmacros zur Überbrückung der Task-Area-Grenzen. Das Takt-Signal wird hingegen über die `Ensure_Clock`-Hardmacros verbreitet, die in regelmäßigen vertikalen Abständen horizontal über die gesamte Breite des FPGAs führen. Über diese Signalleitungen wird auch der Task mit einem Takt versorgt. Das `RST`-Signal teilt dem Task mit, dass er ein Reset auszuführen hat, d.h. alle Register und Signale in den Ausgangszustand versetzen muss. Sobald das Signal wieder auf 0 gezogen wurde, kann der Task mit den ersten Berechnungen oder

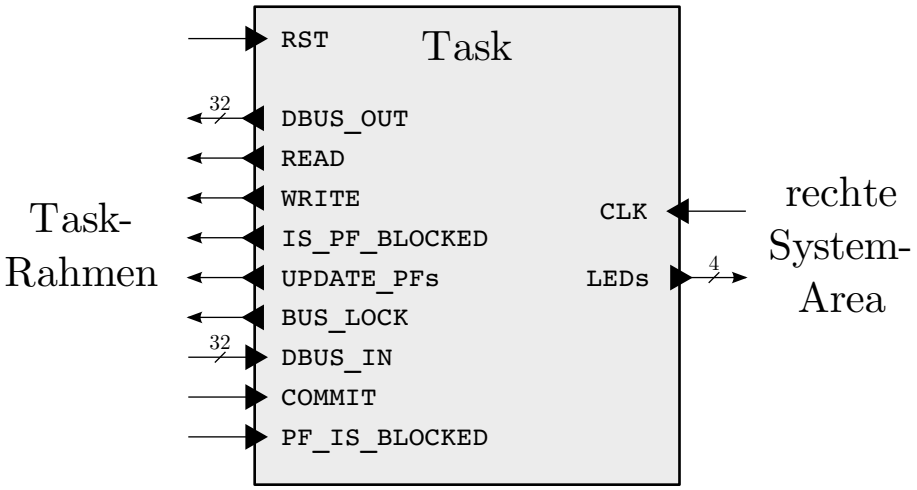


Abbildung 5.7: Schnittstelle des Hardware-Tasks

PO/PF-Zugriffen beginnen. Bis auf das `RST`-Signal dienen die restlichen Signale, die zum Task-Rahmen führen, dem Zugriff auf PublicObjects und PublicFifos. Bei den Zugriffen kann statt der konkreten Nummer des POs bzw. PFs auch die statische Variable angegeben werden, welche durch den BitstreamGenerator automatisch die korrekte Nummer zugewiesen bekommt. Welcher Variablenname mit welchen PO bzw. PF verknüpft ist, wird in der Java-Klasse des Tasks festgelegt. Dort müssen die Methoden `getNamedPOs()` und `getNamedPFs()` implementiert werden. `getNamedPOs()` gibt eine Liste von `NamedPOs` zurück, wobei ein NamedPO sich aus einem PublicObject und dessen Bezeichnung zusammensetzt. Im folgenden Beispiel wird dem PublicObject `controlRegister` der VHDL-Variablenname `CONTROL_REGISTER` zugeordnet, der dadurch in der VHDL-Datei des Tasks für die Referenzierung des PublicObjects verwendet werden kann:

```
protected NamedPO[] getNamedPOs() {
  return new NamedPO[]{
      new NamedPO("CONTROL_REGISTER", controlRegister)
  };
}
```

Die PublicFifos können ähnlich mit Namen versehen werden. In der Methode `getNamedPFs()` muss dazu eine Liste der benannten PFs (`NamedPFs`) zurückgegeben werden.

Neben den Nummern der POs und PFs setzt der BitstreamGenerator noch weitere Werte spezieller Variablen im `generic`-Block der Task-VHDL-Datei. Abbildung 5.8 zeigt die komplette Liste der Wertzuweisungen.

Variable im `generic`-Block	**zugeordneter Wert**
po_name	Nummer des POs
*po_name*_LENGTH	Größe des POs in Bytes
pf_name	Nummer der PF
*pf_name*_STEP_SIZE	Blockgröße der PF
parameter_name	Wert des benutzerdefinierten Parameters

Abbildung 5.8: Liste der in der Task-VHDL-Datei verfügbaren, statischen Variablen. Dabei steht *po_name* für den Namen eines beliebigen PublicObjects, *pf_name* für den einer beliebigen PublicFifo und *parameter_name* für den Namen eines benutzerdefinierten Parameters.

Diese Werte können dazu verwendet werden, ein bestimmtes Task-Verhalten zur Synthesezeit festzulegen.

Im Folgenden wird erörtert, wie ein Task auf ein PublicObject oder eine PublicFifo zugreifen kann und welche Vorgänge dabei im Task-Rahmen stattfinden.

5.4.5 Zugriff auf ein PublicObject

Der Zugriff auf ein PublicObject beginnt mit dem Anlegen der PO-Variable auf dem Datenbus `DBUS_OUT` und dem Setzen des `READ`- oder `WRITE`-Signals, je nachdem ob es sich um eine Lese- oder eine Schreiboperation handelt. Die Signale müssen so lange gesetzt bleiben, bis der Task-Rahmen ein `COMMIT`-Signal zurücksendet. Im Falle eines leseseitigen Zugriffs werden nun die Daten des POs in 4-Byte-Blöcke zerlegt. Jeder Block wird über `DBUS_IN` mit einen `COMMIT`-Signal dem Task übermittelt und muss durch den Task beim nächsten Taktereignis übernommen werden. Das Schreiben eines POs gestaltet sich ähnlich. Nach dem ersten `COMMIT` des Task-Rahmens muss der Task die ersten vier Bytes auf den Bus `DBUS_OUT` legen und so lange angelegt lassen, bis er vom Task-Rahmen ein `COMMIT` erhält. Nach diesem Signal werden die nächsten vier Bytes auf den Bus gelegt und der Vorgang wiederholt sich, bis der Inhalt des POs vollständig geschrieben wurde.

5.4.6 Zugriff auf eine PublicFifo

Die Lese-/Schreib-Zugriffe auf eine PublicFifo erfolgen prinzipiell auf die gleiche Weise wie beim PublicObject. Es werden so viele Bytes gelesen bzw. geschrieben, wie es die Blockgröße der FIFO verlangt. Unterschiede zum PO-Zugriffsprotokoll treten

erst auf, wenn das Lesen aus der FIFO bzw. Schreiben in die FIFO aufgrund des Füllstandes nicht mehr möglich ist. In dem Fall sendet der Task-Rahmen gleich mit dem ersten `COMMIT` auch das Signal `PF_IS_BLOCKED`. Daraufhin bricht der Task den SDRAM-Zugriff vorerst ab und probiert es später erneut. Zum Teil kommt es vor, dass ein Task zunächst sehr große Datenmengen aus einer PF lesen muss, bevor diese verarbeitet werden können. Ist beim Schreiben des Ergebnisses die Ausgabe-PF blockiert, so war der umfangreiche Lesevorgang evtl. umsonst, wenn der Scheduler den Task nun unterbricht. Deshalb steht dem Task auch die Möglichkeit zur Verfügung, zu testen, ob eine PublicFifo blockiert ist, ohne im gegensätzlichen Fall Daten aus der FIFO lesen oder in die FIFO schreiben zu müssen. Dazu muss der Task beim Start der PF-Anfrage zusätzlich noch das Signal `IS_PF_BLOCKED` setzen. Im Ergebnis liefert der Task-Rahmen entweder ein einfaches `COMMIT` für den Fall, dass die FIFO nicht blockiert ist, oder ein `PF_IS_BLOCKED` zusammen mit dem `COMMIT`, um zu signalisieren, dass die PF gerade blockiert ist. Wenn der Task einen kompletten Verarbeitungsschritt abgeschlossen hat, d.h. alle Ergebnisse bereits in eine FIFO geschrieben wurden, muss der Task dem Task-Rahmen das `UPDATE_PFs`-Signal übermitteln, das dafür sorgt, dass die zwischengespeicherten PF-Indizes in den SDRAM geschrieben werden.

5.4.7 Hardware-Task und wechselseitiger Ausschluss

Allgemein kann sich der Hardware-Task darauf verlassen, dass kein anderer Task Zugriff auf den SDRAM bekommt während er auf ein PO oder eine PF zugreift. Auch kann der Task in dieser Zeit nicht unterbrochen werden. Zum Teil ist es jedoch erforderlich, den SDRAM auch über mehrere Zugriffe hinweg zu sperren. Dazu steht dem Task das Signal `BUS_LOCK` zur Verfügung. Solange dieses Signal gesetzt ist, sperrt der Task-Rahmen auch nach einem SDRAM-Zugriff weiterhin den OPB.

Demgegenüber wirkt sich die Sperrung des OPB durch einen anderen Hardware-Task oder die Deaktivierung der SDRAM-Zugriffe im Task-Rahmen nicht auf das Protokoll zwischen Task und Task-Rahmen aus, sondern lediglich auf die Wartezeit, bis ein `COMMIT` vom Task-Rahmen zurückkommt.

5.5 Aufbau der Task-Bitstreams

Bereits im Kapitel 5.4.2 wurde darauf hingewiesen, dass die Task-Area sich aus unterschiedlichen Spaltentypen zusammensetzt. Der TMAN ist jedoch darauf ausgelegt, einen zusammenhängenden FPGA-Bereich genau eines Spaltentyps auszulesen

oder zu beschreiben. Daher wird der TMAN zum Schreiben oder Auslesen der Task-Area zwei Mal angesteuert. Erst werden die *CLB*-Spalten gelesen bzw. geschrieben, danach die *BRAM Interconnect*-Spalten, wobei vor jedem Wechsel des Spaltentyps die HEAD/TAIL-Informationen des TMANs aktualisiert werden. Die entsprechenden Einstellungen werden in der Datei `conf/xupController.properties` über die folgenden Variablen festgelegt:

```
# Groesse eines Frames
reconfig.framesize = 206
# 9 CLB-Spalten (umfasst alle Slices mit x>=52 und x<=69)
reconfig.clb.major.address = 29
reconfig.clb.minor.address = 0
reconfig.clb.dataframes = 198
# 1 BRAM_INT-Spalte (die Spalte zw. den Slices mit x=63 und x=64)
reconfig.bram_init.major.address = 5
reconfig.bram_init.minor.address = 0
reconfig.bram_init.dataframes = 22
```

Dementsprechend ist der Task-Bitstream aufgebaut. Er umfasst keine Informationen über die Spalten und Frames, die für die Rekonfiguration genutzt werden, sondern enthält nur die Frameinhalte, wobei in dem Datenstrom zuerst die Inhalte der *CLB*-Spalten und danach die der *BRAM Interconnect*-Spalten vorkommen. Für die Übertragung der Bitstreams wird die gleiche Komprimierung wie in [Abe05] verwendet. Dabei werden aufeinanderfolgende 0-Werte mit einer einfachen Run-Length-Codierung zusammengefasst. n aufeinanderfolgende Nullen des unkomprimierten Datenstroms werden durch zwei Bytes ersetzt, wobei das erste den Wert Null hat und das zweite die Anzahl der Nullen angibt. Maximal 255 aufeinanderfolgende Nullen werden so komprimiert. In der gleichen Komprimierung werden die Bitstreams auf der Festplatte im Verzeichnis `task/bitstream/` abgelegt. Um den unkomprimierten Inhalt eines Bitstreams anzuzeigen, kann das Shell-Script `./decompressBitstream` eingesetzt werden. Als Parameter wird eine Bitstream-Datei erwartet.

5.6 Kommunikation zwischen PC und Board

Die wesentliche Programmlogik des Frameworks liegt auf Seiten des PCs. Java verwaltet den SDRAM auf dem Board und ist für das Scheduling der Tasks auf dem FPGA verantwortlich. Das auf dem PowerPC laufende Programm wurde dagegen sehr einfach gehalten. Es dient nur der Verarbeitung der Kommandos, die der PC

an das Board sendet. Informationen über die laufenden Hardware-Tasks oder die im SDRAM abgelegten Daten liegen dem PowerPC nicht vor. Folglich besteht die Kommunikation zwischen PC und Board aus elementaren Nachrichten, die durch den PPC wie Kommandos interpretiert werden. Beispiele sind: das Schreiben und Lesen in bzw. aus dem RAM, das Komprimieren und Dekomprimieren von Datenbereichen innerhalb des SDRAMs sowie das Lesen und Schreiben vom bzw. zum ICAP. Die restlichen Nachrichtentypen dienen zum Test der Kommunikation oder zur Bestätigung der eben genannten Nachrichten und Aktionen. Jegliche Aktion wird durch den PC initiiert, d.h. das Board reagiert nur auf Anfragen des PCs.

5.6.1 Kommunikation auf PC-Seite in Java

Im Java-Teil des Frameworks läuft jegliche Kommunikation über die jeweilige Implementierung des `AbstractBoardControllers`. Um mit dem XUP-Board zu kommunizieren kann der `XUPController` eingesetzt werden. Dieser nutzt wiederum weitere Klassen für den Übertragungsweg bis zur seriellen Schnittstelle. Das Diagramm in Abbildung 5.9 verdeutlicht den Datenfluss vom XUPController zur seriellen Schnittstelle.

Eingeteilt ist die Kommunikation in 3 Schichten:

1. In der untersten Schicht wird der durch die RXTX-Bibliothek[1] bereitgestellte `SerialPort` genutzt. Dieser besitzt einen OutputStream und einen InputStream, um auf den seriellen Port zu schreiben und von ihm zu lesen. Leider verhält sich der InputStream nicht ganz Java-konform. Werden mehr Daten angefordert als tatsächlich verfügbar sind, wechselt der Port in einen Zustand, in welchem sich weder der zugehörigen Thread unterbrechen, noch der Port schließen lässt. Letztes sollte in jedem Fall möglich sein, um am Ende eines Programms stets alle Ressourcen sauber freigeben zu können. Für die Behebung des Fehlverhaltens wurde die Klasse `RXTXReader` geschrieben. Dieser wird um den InputStream des `SerialPorts` herum gelegt und stellt einen sauberen Zugriff auf den InputStream bereit. Die Schicht 1 arbeitet nur mit Byte-Folgen, ohne deren Bedeutung zu kennen. Weiterhin sind in dieser Schicht die Lese- und Schreibaktionen voneinander getrennt.

2. In der mittleren Schicht befindet sich der `XUPConntectorCOM`, der einen `XUPReader` und einen `XUPWriter` für den Zugriff auf die darunter liegende Schicht bereitstellt. Der `XUPReader` liest die Rohdaten vom `RXTXReader` und zerlegt

[1] www.rxtx.org

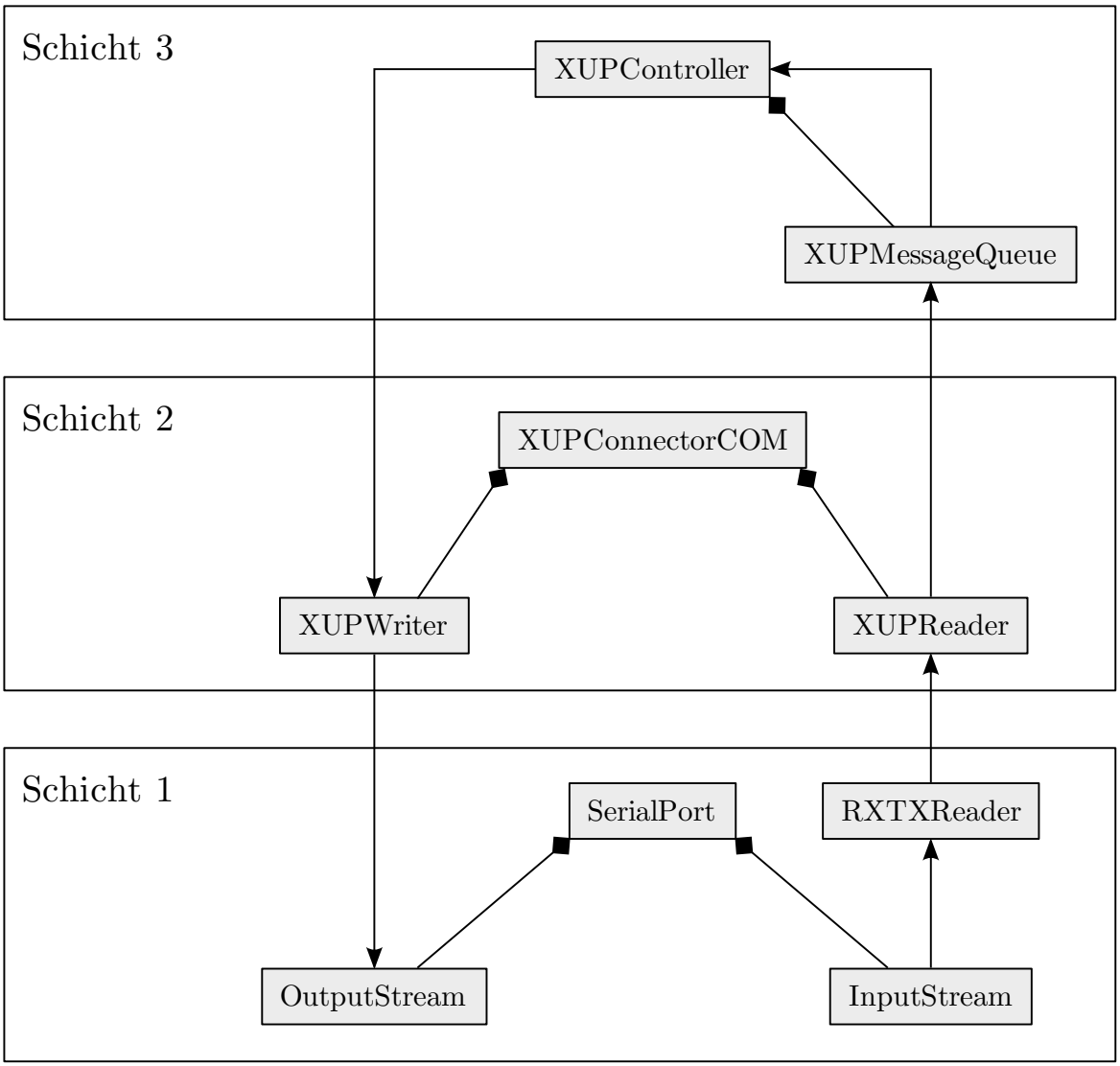


Abbildung 5.9: Kommunikationsschichten und Datenfluss im Java-Teil des Frameworks. Die Linien mit Rauten symbolisieren, wie in UML üblich, die Komposition. Die normalen Pfeile verdeutlichen den Datenfluss der Nachrichten.

Bytefolgen in Nachrichten. Objekte, die über die empfangenden Nachrichten informiert werden möchten, können sich beim `XUPReader` als `XUPReaderListener` registrieren. `XUPReaderListener` ist dabei ein Interface, dass zu jeder eingehenden Nachricht eine entsprechende Methode besitzt. Für die Antwort auf eine SDRAM-Leseoperation wäre dies die Methode:

```
public void clientMsgReadFromRam(byte[] bytes);
```

Der `XUPReader` setzt also den Byte-Datenstrom in Methodenaufrufe mit entsprechenden Parametern um. Der `XUPWriter` stellt hingegen Methoden zum

Versenden von Nachrichten zur Verfügung, die durch ihn in Bytefolgen umgewandelt und an den OutputStream des SerialPort übergeben werden. Die Lese- und Schreibaktionen sind auch in der Schicht 2 voneinander getrennt und erfolgen asynchron.

3. Die oberste Schicht wird vom `XUPController` gesteuert, der ausgehende Nachrichten direkt an den `XUPWriter` übermittelt. Eingehende Nachrichten werden erst in eine Warteschlage gepackt, die durch die Klasse `XUPMessageQueue` implementiert ist. Der `XUPController` ruft diese Warteschlage bei Bedarf ab. In der Schicht 3 sind die Lese- und Schreibaktionen nicht mehr voneinander getrennt. Sendet ein Thread eine Nachricht über den `XUPController`, so wartet dieser so lange, bis eine Antwort vom Board erhalten wurde.

5.6.2 Kommunikation auf Board-Seite in C

Auf dem Board läuft ein C-Programm, dass die Nachrichten von der seriellen Schnittstelle entgegennimmt und dementsprechende Aktionen ausführt. Bis auf die *nop*-Nachricht wird jede Nachricht durch eine Bestätigungsmitteilung quittiert. Die Logik für den Nachrichtenaustausch ist in der Datei edkProject/AppTman/src/mystdio.c implementiert.

5.6.3 Aufbau der Nachrichten

Die Nachrichten, die über die serielle Schnittstelle übertragen werden, bestehen aus einem Nachrichtenkopf und einem Nachrichtenkörper. Der Kopf setzt sich aus einem 4-Byte-Startwort ("XUP!"), einem 1-Byte-Nachrichtentyp und einer Längenangabe von 1 oder 5 Bytes zusammen. Das Startwort markiert den Nachrichtenbeginn und soll eine Desynchronisation verhindern, da beide Seiten den Datenstrom stets auf das Vorkommen dieser 4 Bytes prüfen und nur im positiven Falle mit dem Einlesen und der Verarbeitung der Nachricht beginnen. Die Längenangabe bestimmt die Länge des Nachrichtenkörpers. Sie nimmt genau dann 1 Byte ein, wenn der Nachrichtenkörper kürzer ist als 255 Bytes. Ist der Körper länger als 254 Byte, so wird dem ersten Byte der Wert 255 gegeben und die nachfolgenden 4 Bytes enthalten nun die Länge des Körpers.

5.6.4 Nachrichtentypen und deren Kosten

Die Geschwindigkeit der UART-Komponenten wurde auf 9600 Baud gesetzt. Die Übertragungszeit für ein Byte beträgt somit rund 0,1 ms. Nachfolgend werden die Übertragungskosten nur noch in Bytes angegeben.

Die Kosten für den Nachrichtenkopf betragen

- 6 Bytes, falls der Nachrichtenkörper kleiner als 255 Bytes ist
- 10 Bytes, falls der Nachrichtenkörper größer als 254 Bytes ist

In der Abbildung 5.10 werden die einzelnen Nachrichten und ihre Kosten (ohne Berücksichtigung des Nachrichtenkopfes) vorgestellt. Bis auf die *nop*-Nachricht werden alle Anfragen durch das Board mit Bestätigungsnachrichten quittiert. Anfragen und Antworten wurden deshalb in der Tabelle gruppiert.

Neben den aufgeführten Nachrichten unterstützt der `XUPController` auch Befehle, die sich aus mehreren Nachrichten zusammensetzen. Beispiele sind `startTask` oder `writeTaskToIcap`.

5.7 Simulation und Test von Hardware-Tasks

VHDL-Code ohne die Hilfe eines VHDL-Simulators zu entwickeln, ist höchst aufwendig, da zum Testen des Hardware-Designs ein langwieriger Synthese-Prozess erforderlich ist. Mit der VHDL-Simulation hingegen lässt sich das Verhalten einzelner VHDL-Komponenten sehr schnell und exakt testen. Zudem können die Änderungen sämtlicher Signal- und Registerwerte während der Simulation analysiert werden. Diese Gründe waren letztlich ausschlaggebend für den Aufbau einer kleinen Simulationsumgebung, die sich aus den VHDL-Dateien unter `edkProject/pcores/Rahmen/hdl/sim_vhdl/` und dem `makefile` im Wurzelverzeichnis zusammensetzt und den freien VHDL-Simulator *GHDL* nutzt. Im Blickfeld lag zum einen die Möglichkeit der Simulation einzelner Hardware-Tasks, aber auch die Verifikation des Verhaltens des Task-Rahmens. Um dessen korrekte Arbeitsweise festzustellen, wurden zugleich verschiedene Beispiel-Tasks erstellt, die mit dem Task-Rahmen verbunden wurden, um das Zusammenspiel beider Komponenten testen zu können. Die für die Simulation zuständigen VHDL-Dateien eines Tasks werden im Verzeichnis `task/source/`*`task_name`*`/sim/` abgelegt. Dort befinden sich Dateien mit dem Präfix `Rahmen_` und mit dem Präfix `TASK_`. Erstere dienen zur Simulation des Rahmens mit angeschlossenem Task, letztere nur zur Simulation des Tasks. Abbildung 5.11 stellt diese VHDL-Dateien kurz vor.

Richtung	**Nachrichtentyp**	**Kosten in Bytes**	**Beschreibung**
PC → B	nop	0	keine Operation wird ausgeführt
PC → B PC ← B	ping pong	0 0	zum Testen der Verbindung und ob beide Seite empfangsbereit sind
PC → B PC ← B	echo output	|text| |text|	sendet einen Text an das Board, der danach zurückgesendet wird
PC → B PC ← B	writeToRam writtenToRam	4+|data| 0	schreibt eine Bytefolge an eine bestimmte Stelle im SDRAM
PC → B PC ← B	readFromRam readFromRam	8 |data|	liest eine Bytefolge aus dem SDRAM
PC → B PC ← B	compress compressed	12 4	komprimiert eine Bytefolge innerhalb des SDRAMs
PC → B PC ← B	decompress decompressed	12 4	dekomprimiert eine Bytefolge innerhalb des SDRAMs
PC → B PC ← B	writeRamToIcap ramWrittenToIcap	4 0	sendet die Daten eines SDRAM-Bereiches an das ICAP
PC → B PC ← B	writeIcapToRam icapWrittenToRam	4 0	liest Daten vom ICAP und schreibt sie in einen SDRAM-Bereich

Abbildung 5.10: Nachrichtentypen mit Beschreibung und Kosten der Nachrichten ohne Nachrichtenkopf. In der Richtungsspalte bezeichnet B das Board. |text| und |data| stehen für die Länge der Nutzdaten in Bytes.

Durch den Suffix `_behav` werden die Dateien markiert, die ausschließlich eine VHDL-Konfiguration enthalten. Die Dateien mit dem Suffix `_tb` beinhalten hingegen die sogenannten Testbenches. In diesen können für jeden Simulationszeitpunkt die Signalwerte an den Eingängen mit den erwarteten Werten an den Ausgängen in einer Liste zusammengetragen werden. Per

`make` *`task_name`*`-task-tb`

bzw.

`make` *`task_name`*`-rahmen-tb`

wird der Testdurchlauf gestartet und überprüft, ob die Signale an den Ausgängen den erwarteten Werten entsprechen. Die letzte Zeile der Ausgabe (`Assertion errors:`

Datei	**Beschreibung**
`Rahmen_behav.vhd`	Konfiguration der `Rahmen.vhd`; Setzen generischer Task-Parameter möglich
`Rahmen_tb_behav.vhd`	Konfiguration der `Rahmen_tb.vhd`; Setzen generischer Parameter des Testbenches für den Task-Rahmen möglich
`Rahmen_tb.vhd`	Testbench für den Task-Rahmen
`TASK_tb_behav.vhd`	Konfiguration der `TASK_tb.vhd`; Setzen generischer Parameter des Testbenches für den Task möglich
`TASK_tb.vhd`	Testbench für den Task

Abbildung 5.11: VHDL-Dateien für die Simulation eines Hardware-Tasks

...) gibt eine schnelle Auskunft über die Anzahl der Fehler. Um den genauen Verlauf der Signaländerungen zu betrachten, kann das freie Programm GTKWave (siehe Abbildung 5.12) genutzt werden. Dazu sind die Aufrufe

`make` *`task_name`*`-task-wave`

bzw.

`make` *`task_name`*`-rahmen-wave`

erforderlich.

Zwar lassen sich mit der VHDL-Simulation Fehler im VHDL-Code meistens sehr schnell finden, doch können dadurch nicht alle Fehler erkannt und ausgeschlossen werden. Zum einen hängt die korrekte Ausführung der VHDL-Simulation davon ab, ob alle Prozesse mit ordnungsgemäßen Sensitivitätslisten ausgestattet wurden. Zum anderen können auf der endgültigen Hardware weite Probleme wie beispielsweise Timing-Differenzen auftreten, die während der Simulation nicht erkannt wurden. Deshalb muss im letzten Schritt das komplette Design erzeugt und auf dem FPGA getestet werden. Die Erzeugung mit anschließender Programmierung des FPGAs erfolgt mit dem Befehl

`make` *`task_name`*`-download`

Danach kann man die Funktionsweise zum Beispiel über JUnit-Tests, die im nächsten Kapitel beschrieben werden, überprüfen. Auch eine interaktive Fehleranalyse mit Hilfe des BeanShellWindows (siehe Kapitel 5.10) ist an dieser Stelle nützlich.

Zu guter Letzt muss im FPGA-Editor kontrolliert werden, ob Signalleitungen

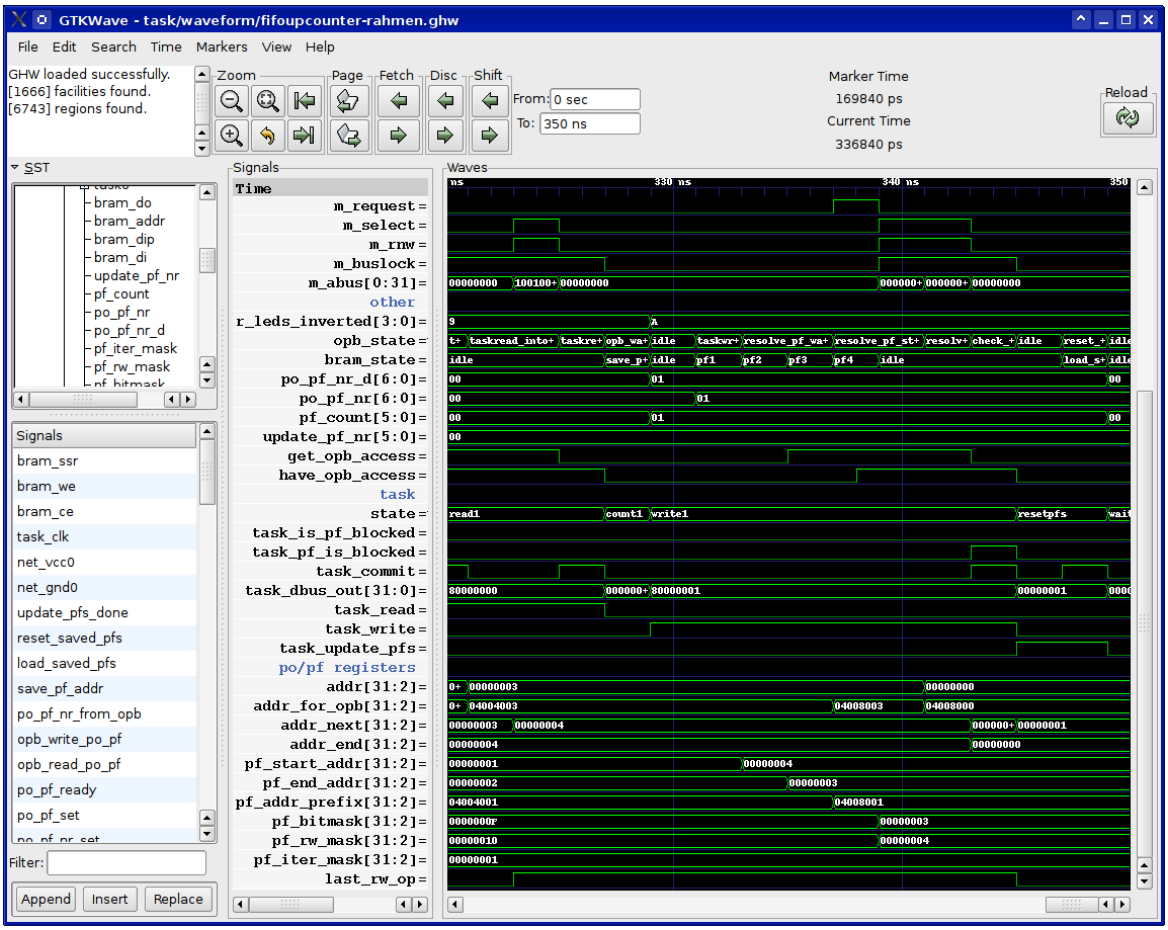


Abbildung 5.12: Screenshot von GTKWave

die Grenze zwischen System- und Task-Area illegal kreuzen. Dazu lässt sich das Hardware-Designs mittels

`make` *`task_name`*`-fpga-editor`

im FPGA-Editor öffnen. Diese Überprüfung ist leider für jede parametrisierte Instanz eines Tasks durchzuführen, die durch den BitstreamGenerator erzeugt wurde.

Änderungen am Task-Rahmen wirken sich in der Regel auf alle Tasks aus. Nach solchen Änderungen kann mit dem Kommando

```
make all-tbs
```

überprüft werden, ob sich der Task-Rahmen weiterhin wie erwartet verhält, wobei dieser Test alle Tasks durchläuft.

5.8 Test von Software-Tasks

Für Tests unter Java bietet sich das *JUnit-Framework* an. Ähnlich wie bei den VHDL-Testbenches lassen sich mit JUnit einzelne Klassen und Methoden testen, indem sie mit unterschiedlichen Parametern instantiiert bzw. aufgerufen werden und anschließend die Resultate mit den entsprechenden Erwartungswerten verglichen werden. Unterstützt wird das JUnit-Framework unter anderem von *Apache Ant* und der freien Entwicklungsumgebung *Eclipse*.

In dieser Arbeit wurden alle JUnit-Tests im Java-Package `test` abgelegt. Darunter befinden sich reine Software-Tests, welche die Funktionsweise einiger Framework-Bestandteile und Software-Tasks testen, aber auch Tests, die das Verhalten der Hardware-Komponenten auf dem FPGA, einschließlich Hardware-Tasks, überprüfen. Zur Ausführung letzterer ist demnach ein eingeschaltetes und mit dem PC verbundenes Board erforderlich. Für die Überprüfung des Verhaltens der Software-Tasks ist dagegen keine Verbindung zum Board nötig. Alle Zugriffe auf den SDRAM des Boardes können auf dem PC simuliert werden. Zu diesem Zwecke wurde der `VirtualBoardController` entwickelt, der anstelle des sonst eingesetzten `XUPControllers` verwendet wird. Von den neun verschiedenen Nachrichten an das Board werden durch den `VirtualBoardController` lediglich die Anfragen an den SDRAM bearbeitet. Diese werden jedoch nicht an das Board weitergereicht, sondern in einen internen Array-Zugriff umgesetzt.

Die kompletten Framework-Einstellungen für diese softwareseitige Simulation können per

```
TaskFramework.setDefault(
  new TaskFramework("conf/virtualController.properties")
);
```

zu Beginn des Tests oder der Anwendung geladen werden.

Alle Tests lassen sich mit Hilfe von Ant hintereinander ausführen. Um die reinen Software-Tests auszuführen, kann auf der auf Kommandozeile

```
ant runAllSwTests
```

aufgerufen werden. Die Tests, die das eingeschaltete Board voraussetzen, werden über

```
ant runAllHwTests
```

gestartet. Alle Tests zusammen können mit

```
ant runAllTests
```

ausgeführt werden. Dabei wird neben den Ausgaben der Tests im Ergebnis auch die Anzahl der fehlgeschlagenen Tests angezeigt.

5.9 Protokollierung und Ereignisse

5.9.1 Nutzung der Logging-API von Sun

Die Protokollierung bestimmter Ereignisse oder Aktionen von Prozessen kann beim Aufspüren von Fehlern in einem Programm sehr hilfreich sein. Mit dem JRE liefert Sun eine Logging-API, die eine prioritätengesteuerte Protokollierung ermöglicht. Der Detailgrad der Logging-Ausgabe kann für jedes Java-Package und für jede Klasse komfortabel in einer Konfigurationsdatei festgelegt werden. Die Logging-Standardeinstellungen für das Framework wurden in der Datei `conf/logging.properties` abgelegt, die zu Beginn der Java-Beispiel-Anwendungen, der JUnit-Tests und auch beim Start des BeanShellWindows geladen werden.

Zu den wesentlichen Einstellungen in dieser Datei gehört der Wert des Schlüssels `handlers`. Über diesen können verschiedene Klassen für die Log-Ausgabe bestimmt werden. Durch Angabe der Klasse `java.util.logging.ConsoleHandler` lassen sich die Log-Ausgaben auf die Standardfehlerausgabe leiten, mit der Angabe von `java.util.logging.FileHandler` hingegen in eine Datei. Genauere Informationen dazu stellt die Firma Sun auf der Webseite [Mic08] bereit.

Um das Logging komplett zu deaktivieren, kann der `ConsoleHandler` wie folgt deaktiviert werden:

```
java.util.logging.ConsoleHandler.level = OFF
```

Statt die Protokollierung mit `OFF` komplett abzuschalten, kann ein konkretes Log-Level angegeben werden, mit dem die Ausgaben auf die Log-Nachrichten dieses und aller höheren Level beschränkt werden. Dabei stehen die in Abbildung 5.13 aufgeführten Log-Level zur Verfügung.

Über die globalen Logging-Einstellungen hinaus lassen sich spezifische Einstellungen für einzelne Klassen und Packages setzen. Die zugehörigen Einträge in der Konfigurationdatei haben die folgende Form:

```
Klassenname.level = Log-Level
Packagename.level = Log-Level
```

Log-Level	Bedeutung
`OFF`	Ausgabe keiner Nachrichten
`SEVERE`	Ausgabe von Fehlern
`WARNING`	Ausgabe von Warnungen
`INFO`	informative Ausgaben
`CONFIG`	Ausgabe von statischen Konfigurationen
`FINE`	Überwachungsausgaben
`FINER`	recht detaillierte Überwachungsausgaben
`FINEST`	hoch detaillierte Überwachungsausgaben
`ALL`	Ausgabe aller Nachrichten

Abbildung 5.13: Die Log-Level der Logging-API von Sun (nach Priorität absteigend sortiert)

5.9.2 Logging-Einstellungen für die wichtigsten Klassen

Die konkreten Protokollierungseinstellungen für die Java-Packages `example.apps` und `framework` befinden sich am Ende der Datei `conf/logging.properties`. Einige häufig verwendete Einstellungen wurden lediglich auskommentiert, um bei Bedarf auf die entsprechenden Konfigurationen schnell zurückgreifen zu können. Um beispielsweise die komplette Board-Kommunikation zu protokollieren, müssen nur die folgenden Zeilen einkommentiert werden:

```
framework.xup.XUPWriter.level = ALL
framework.xup.XUPReader.level = ALL
```

Nachfolgend soll dargestellt werden, welche Ausgaben die einzelnen Log-Level für die implementierten Klassen zur Folge haben:

- example.apps.MyApplication

  `SEVERE`: Fehlerausgabe des Scripts `download.sh` zum Download des initialen Bitstreams auf das Board

  `FINER`: Standardausgabe des Scripts `download.sh`; Details und Fortschritt werden ausgegeben

- framework.BitstreamGenerator.level

  `SEVERE`: Fehlerausgabe des Scripts `bitstreamGenerator.sh` zur Generierung des Bitstreams – entspricht der Ausgabe von

`make -f system.make download`
im EDK-Projektverzeichnis

`FINER`: Standardausgabe des Scripts `bitstreamGenerator.sh`; Details und Fortschritt werden ausgegeben

- framework.RamAddressManager.level

  `SEVERE`: Fehlerausgabe beim erfolglosen Laden oder Generieren eines Bitstreams

  `INFO`: Information über Beginn und Ende der Bitstream-Generierung

  `FINE`: Information über das erfolgreiche Laden eines Bitstreams

- framework.SimpleScheduler.level

  `FINE`: Ereignisse wie das An- und Abmelden eines Tasks beim Scheduler

- framework.xup.XUPController.level

  `SEVERE`: schwerwiegende Fehler (Klassen können nicht instantiiert werden)

  `FINER`: alle Nachrichten, die an das Board gesendet werden

- framework.xup.XUPMessageQueue.level

  `FINER`: alle Nachrichten, die vom Board gesendet wurden (bis auf OutputMessage und ErrorMessage)

- framework.xup.XUPWriter.level

  `FINER`: alle Nachrichten, die an das Board gesendet werden

- framework.xup.XUPReader.level

  `INFO`: Bytes, die außerhalb einer Nachricht empfangen werden, d.h. bevor ein Nachrichtenkopf empfangen wurde

  `FINER`: alle Nachrichten, die empfangen werden bis auf OutputMessage und ErrorMessage

- framework.xup.XUPReader.out.level

  `INFO`: Nachrichten vom Typ OutputMessage, die empfangen werden

- framework.xup.XUPReader.err.level

  `SEVERE`: Nachrichten vom Typ ErrorMessage, die empfangen werden

5.9.3 Benachrichtigung über Framework-Ereignisse

Neben der genannten Protokollierung von Framework-Ereignissen gibt es auch die Möglichkeit, dass sich Java-Objekte über das Eintreten bestimmter Ereignisse informieren lassen. Um über neue PublicObjects, neue PublicFifos und neue Tasks informiert zu werden, können sie sich beim `RamAddressManager` über die Methode `addRamAddressManagerListener()` registrieren. Weiterhin können die Zustände der Tasks sehr leicht überwacht werden. Um bei einem Zustandswechsel benachrichtigt zu werden, können sich die zu informierenden Objekte beim jeweiligen Task über die Methode `addTaskListener()` anmelden. Ausgenutzt wurde diese Funktionalität unter anderem in der Beispielanwendung `MyDynApplication` (siehe Kapitel 6.2), um stets die aktuellen Ausführungsmodi der Tasks anzuzeigen und neu instantiierte PublicObjects automatisch in eine Tabelle einzutragen.

5.10 Interaktive Programmausführung mit dem BeanShellWindow

Während der Entwicklung des Task-Rahmens und wurde ein kleines Werkzeug namens BeanShellWindow entwickelt, um das Verhalten des Task-Rahmen schneller testen zu können. Es stellt eine grafische Oberfläche für die BeanShell[2] zur Verfügung und ermöglicht eine interaktive Ausführung von Java-Kommandos, wobei die Ergebnisse sofort ausgewertet werden können. Unterstützt wird die Arbeit mit dem BeanShellWindow durch Tastenkürzel, benutzerdefinierte Ersetzungsregeln und die Möglichkeit zur Ausführung von BeanShell-Scripten.

Die Oberfläche, dargestellt in Abbildung 5.14, besteht aus einem großen Textfeld und einer Eingabezeile. In der Eingabezeile kann der Java-Code eingegeben werden, der beim Betätigen der Enter-Taste sofort ausgeführt wird. Die Ausgabe erscheint in dem Textfeld über der Eingabezeile. Dort werden in Abhängigkeit von der Bedeutung der Ausgabe unterschiedliche Farben eingesetzt:

[2] www.beanshell.org

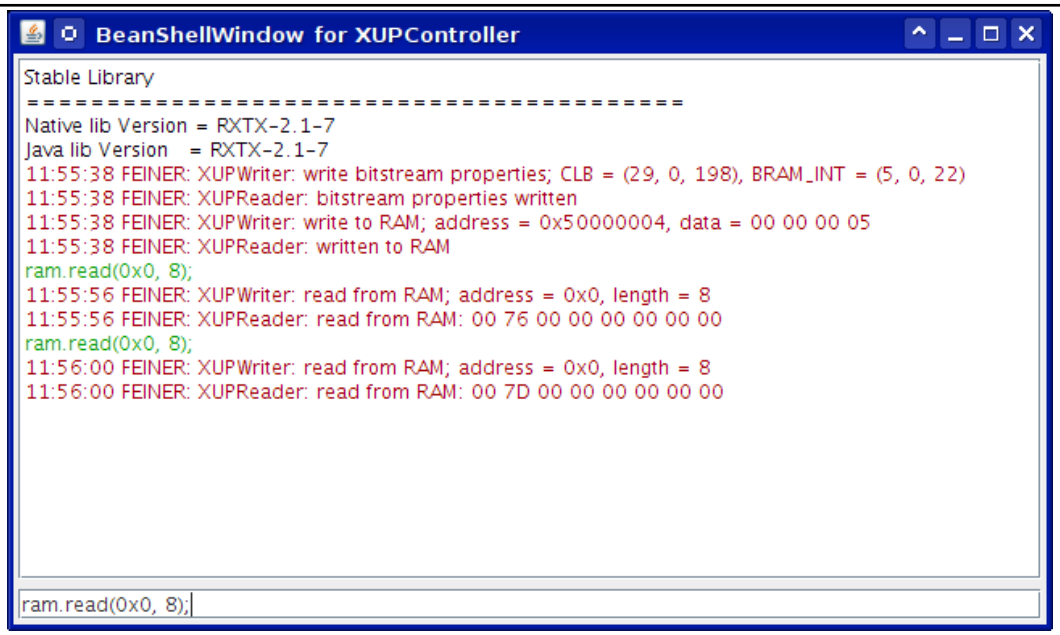


Abbildung 5.14: Screenshot vom BeanShellWindow

Farbe	**Bedeutung**
Grün	Java-Kommandos, die abgesetzt wurden
Schwarz	die Standardausgaben (System.out)
Rot	die Fehler- und Logging-Ausgaben (System.err)
Blau	interne Hinweise, die nicht von den abgesetzten Java-Kommandos stammen

5.10.1 Start des BeanShellWindows

Vor dem Start des BeanShellWindows muss zunächst ein Design auf den FPGA geladen werden, da das BeanShellWindow direkt nach dem Start mit dem Board kommuniziert. Das Standard-Design (fifoupcounter-8-8-24) kann per

```
./download.sh
```

auf das Board gebracht werden. Andere Designs können mit

`make` *`task_name`*`-download`

geladen werden, wobei *`task_name`* der Name eines parametrisierten Tasks (z.B. up-

counter-2-2-1-25) oder eines unparametrisierten Tasks (z.B. upcounter) ist. Mit Ant lässt sich das BeanShellWindow anschließend wie folgt starten:

```
ant runBeanShellWindow
```

Nach dem Start der Anwendung wird das BeanShell-Script `scripts/autorun.bsh` abgearbeitet. In ihm werden einige Java-Import-Statements ausgeführt und der Task-Manager erstellt sowie einige nützliche Variablen gesetzt. Danach können beliebige Java-Befehle zur Ausführung eingeben werden.

5.10.2 Interne kleine Kommandos

Die Eingabezeile für die Absetzung von Java-Code wird gleichzeitig für die Ausführung kleiner interner Befehle genutzt, welche die Arbeit weiter unterstützen sollen. So lassen sich mit der Eingabe des Kommandos `variables` alle lokalen Variablen ausgeben, auf die zurückgegriffen werden kann. Ebenso können die lokal definierten Methoden per Eingabe von `methods` ausgegeben werden. Eine Liste der Ersetzungsregeln, die durch die Tabulator-Taste angestoßen werden, kann mit `replacements` herausgeschrieben werden. Der Befehl `help` gibt die eben genannten drei Gruppen gemeinsam aus. Um größere stets gleichbleibende Quellcode-Stücke nicht nach jedem Start erneut eingeben zu müssen, besteht auch die Möglichkeit, diese in BeanShell-Script-Dateien unterzubringen. Dazu kann eine Datei im Verzeichnis `scripts/` angelegt werden. Durch die Eingabe von `load` *`my_script`* in der Eingabezeile wird dann die Datei `scripts/`*`my_script`*`.bsh` geladen. Um nachzusehen, welche Kommandos zuletzt abgefeuert wurden, kann der Befehl `history` genutzt werden.

Weitere kleine Kommandos lassen sich in der Methode setupSmallCommands() im `BeanShellWindow` hinzufügen.

Innerhalb eines Scripts funktionieren die kleinen Kommandos allerdings nicht. Um dort ein anderes Script aufzurufen, muss `shell.load("`*`my_script`*`");` verwenden werden.

5.10.3 Tastenkürzel

Folgende Tastenkürzel stehen im BeanShellWindow zur Verfügung:

Tastenkürzel	**Beschreibung**
Strg+Leertaste	vervollständigt Variablen-, Methoden- oder kleine Kommandonamen
Tab	ersetzt das Schüsselwort durch den vorgegebenen Text
Pfeil nach oben	geht in der Liste der letzten Kommandos einen Schritt zurück
Pfeil nach unten	geht in der Liste der letzten Kommandos einen Schritt vor
Strg+U	löscht den Text vor dem Cursor
Strg+K	löscht den Text hinter dem Cursor
Strg+Y	löscht den gesamten Text der Eingabezeile
Strg+D	schließt das Fenster

5.10.4 Konfigurationsdatei

Wenn keine andere Konfigurationsdatei über den VM-Parameter `-Dbeanshellwindow.config=` angegeben wurde, werden die Einstellungen aus der Datei `conf/beanshellwindow/default.properties` geladen. Folgende Parameter können in dieser gesetzt werden:

`title=`*`Fenstertitel`*
`replacements=`*`Pfad zur Ersetzungsdatei`*
`x=`*`x-Koordinate des Fensters`*
`y=`*`y-Koordinate des Fensters`*
`width=`*`Breite des Fensters`*
`height=`*`Höhe des Fensters`*
`autorun=`*`mit Semikolon getrennte Liste von BeanShell-Scripten`*

5.10.5 Ersetzungsregeln

In der Datei `conf/beanshellwindow/xupcontroller.properties` werden die Ersetzungsregeln definiert, die in der Eingabezeile später per Druck der Tabulator-Taste expandiert werden können. Das Format der Datei ist einfach. Pro Zeile kann eine Ersetzung definiert werden. Am Anfang der Zeile muss das Schlüsselwort stehen, das ersetzt werden soll, gefolgt von beliebig vielen Leerzeichen und der anschließenden Ersetzungszeichenkette. Das Schlüsselwort darf nur aus Buchstaben, Zahlen, Unter-

strichen und Minuszeichen bestehen. Die Ersetzungszeichenkette hingegen kann sich aus beliebigen Zeichen zusammensetzen. `$C$` markiert in dieser die Position des Cursors nach der Ersetzung. Hier ein Beispiel:

```
serr System.err.println($C$);
```

5.10.6 Anwendungsbeispiele

Im Verzeichnis `scripts/` liegen bereits verschiedene BeanShell-Scripte, die im Laufe der Entwicklung des Frameworks erstellt wurden. Die drei Wesentlichen werden in der Tabelle in Abbildung 5.15 kurz vorgestellt und zeigen exemplarisch den Weg der Entwicklung von den direkten Zugriffen auf die Task-Rahmen-Register bis hin zur finalen Kapselung dieser Zugriffe durch PublicObjects und Task-Objekte. Werden Änderungen am Task-Rahmen oder an der Kommunikation vorgenommen, stellt es sich vor allem bei der Fehlersuche als sinnvoll heraus, auf die unterste Implementierungsschicht zurückzukehren.

Script	**Beschreibung**
`low.bsh`	testet den UpCounterTask in Hardware auf niedrigster Ebene, setzt dabei das geladene Design `upcounter-2-2-1` voraus; Durch direkten Zugriff auf die Task-Rahmenregister wird der Task initialisiert (PO-Informationen gesetzt) und gestartet.
`mid.bsh`	testet den UpCounterTask in Hardware auf mittlerer Ebene, setzt dabei das geladene Design `upcounter-2-2-1` voraus; Es wird kein Task-Objekt erstellt. Gegenüber `low.sh` werden hier jedoch PublicObjects instantiiert, auf die der Task später zugreift. Gestartet wird der Task über die `startTask`-Methode des `XUPControllers`.
`high.bsh`	testet den UpCounterTask in Hardware auf höchster Ebene, setzt ein beliebiges Design auf dem FPGA voraus; Der Task wird als Objekt instantiiert und über die `start()`-Methode gestartet.

Abbildung 5.15: Die wesentlichen BeanShell-Scripte

Im Folgenden soll nun der praktische Nutzen des BeanShellWindows an einem Beispiel demonstriert werden. Für die Ausführung des Scripts `high.bsh` muss zunächst ein Designs auf den FPGA geladen werden. Danach kann das BeanShellWindow gestartet und die folgende Zeile eingegeben werden:

```
load high
```

Dadurch wird ein UpCounterTask erstellt, in Hardware gestartet und der Zählerwert auf 0 gesetzt. Letzterer ist nun unter dem Variablennamen `counter` ansprechbar. Um diesen Variablennamen in der Eingabezeile zu schreiben, genügt es, `cou` einzugeben und danach *Strg+Leertaste* zu drücken. Die Variable wird zu `counter` vervollständigt. Um den Zähler zu stoppen, fügt man noch `.interrupt()` hinzu und drückt anschließend *Enter*. Auf den SDRAM kann über die Variable `ram` zugegriffen werden. Folgende Zeile liest beispielsweise acht Bytes aus dem SDRAM an Adresse 0x0 aus:

```
ram.read(0x0, 8);
```

Diese Eingabe kann wieder durch `r` und *Tab* beschleunigt werden. Mit der Ausführung dieses Java-Kommandos wird zwar der RAM ausgelesen, das Ergebnis aber nicht per `System.out.println(...)` ausgegeben. Empfohlen wird daher, zu Testzwecken die Protokollierung der seriellen Kommunikation in den Logging-Einstellungen zu aktivieren. Auf die Weise lassen sich auch die Inhalte der SDRAM-Zugriffe verfolgen.

Kapitel 6

Beispielanwendungen und Messergebnisse

Zur Demonstration der Implementierung wurden einige Beispiel-Tasks und Beispielanwendungen entwickelt, die in diesem Kapitel vorgestellt werden. Performance-Messungen hinsichtlich der Rekonfigurationszeiten und SDRAM-Zugriffe schließen das Kapitel ab.

6.1 Beispiel-Tasks

Die Beispiel-Tasks und Beispielanwendungen wurden in den Packages `example.tasks` und `example.apps` abgelegt. Zu den Tasks gehören unter anderem die Klassen `UpCounterTask`, `DownCounterTask` und `FifoUpCounterTask`. Die ersten beiden Task-Klassen arbeiten mit drei PublicObjects: eines für die Eingabe, eines für die Ausgabe und zuletzt eines als Steuerregister. Für den UpCounterTask wurden die einzelnen Arbeitsschritte im Diagramm 6.1 veranschaulicht.

Sowohl UpCounter- als auch DownCounterTask lesen zuerst einen ganzzahligen 16-Bit-Zählerwert aus dem Eingabe-PO, warten etwa 500 Millisekunden als Simulation einer aufwendigeren Berechnung, inkrementieren bzw. dekrementieren danach den Zählerwert und schreiben ihn in das Ausgabe-PO, sofern das Kontroll-PO weiterhin auf 0 gesetzt ist. Wenn es in der Zwischenzeit jedoch durch einen anderen Task auf 1 gesetzt wurde, so wird der neu berechnete Zählerwert nicht im SDRAM gesichert, sondern lediglich der Wert des Kontroll-POs auf 0 zurückgesetzt. Dieser Vorgang wiederholt sich so lange, bis der Task unterbrochen wird. Referenzieren Ein- und Ausgabe-PO das gleiche Objekt, dann arbeiten die beiden Tasks wie normale fortlaufende Zähler. Für den Zeitraum ab dem Lesen des Steuerungs-POs bis zum

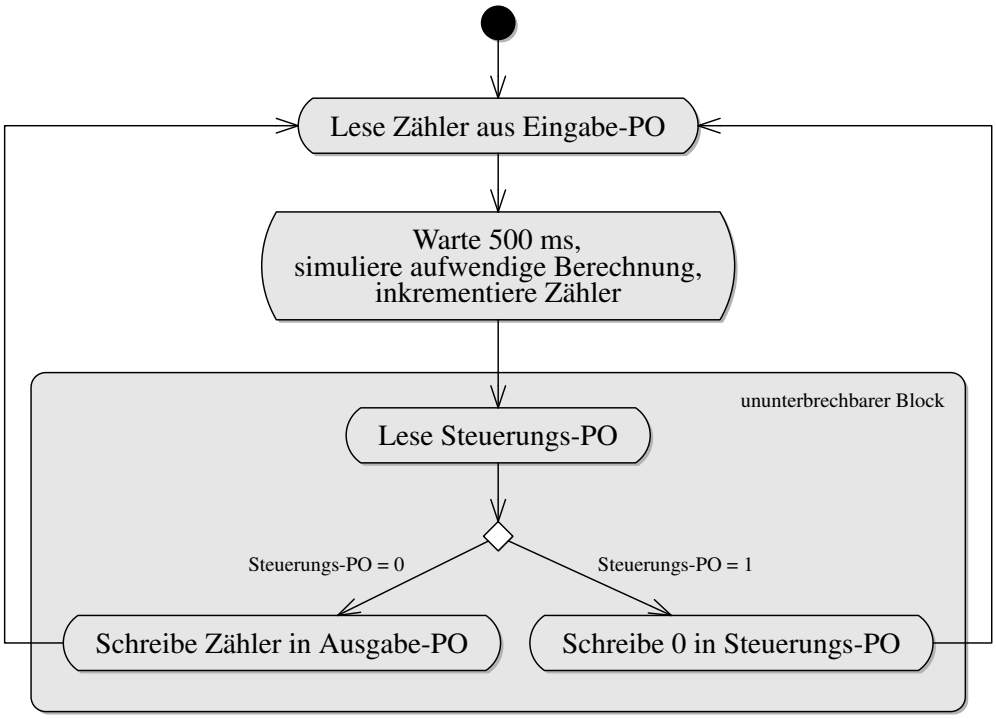


Abbildung 6.1: Arbeitsschritte des UpCounterTasks

Schreiben in das Ausgabe- bzw. Steuerungs-PO holt sich der Task einen exklusiven Zugriff auf den SDRAM. Auf diese Weise lässt sich der Zählerwert in Kombination mit dem Steuerungs-PO auch von außen setzen, und zwar ohne Gefahr, dass der Zählerwert durch den Counter-Task ungelesen überschrieben wird. Es handelt sich um ein typisches Beispiel für eine Synchronisation der SDRAM-Zugriffe. Weiter wurde der Sperrzeitraum weitestgehend minimiert. Für den Hardware-Tasks beträgt die Sperrzeit nur einen Bruchteil einer Millisekunde. Der entsprechende Software-Task hingegen sperrt den SDRAM für mehrere Millisekunden, da die Übertragung über die serielle Schnittstelle doch sehr langsam erfolgt. Dennoch steht der SDRAM während der weitaus größeren, restlichen Ausführungszeit auch anderen Tasks zur Verfügung.

6.2 Beispielanwendungen

Um die Funktionsweise des Frameworks und der Tasks zu demonstrieren wurden drei Java-Beispielanwendungen entwickelt: `MyApplication`, `MyApplication2` und `MyDynApplication`. Bei den ersten beiden Anwendungen handelt es sich um Programme ohne grafische Benutzeroberfläche. Beide Anwendungen instantiieren verschiedene Up- und DownCounterTasks, die zu festgelegten Zeitpunkten gestartet,

gestoppt oder in einen anderen Ausführungsmodus versetzt werden. Die Zählerwerte der Tasks werden in regelmäßigen Abständen auf der Konsole ausgegeben.

Etwas anschaulicher ist dagegen die grafische Anwendung `MyDynApplication`, die in Abbildung 6.2 dargestellt ist. Sie gibt dem Benutzer die Möglichkeit zu entscheiden, wann ein bestimmter Task instantiiert, gestartet, gestoppt oder wieder entfernt werden soll und in welchem Ausführungsmodus er zu laufen hat. Dabei werden die Fähigkeiten des Frameworks genutzt, über Zustandsänderungen von Tasks und neue PublicObjects informiert zu werden.

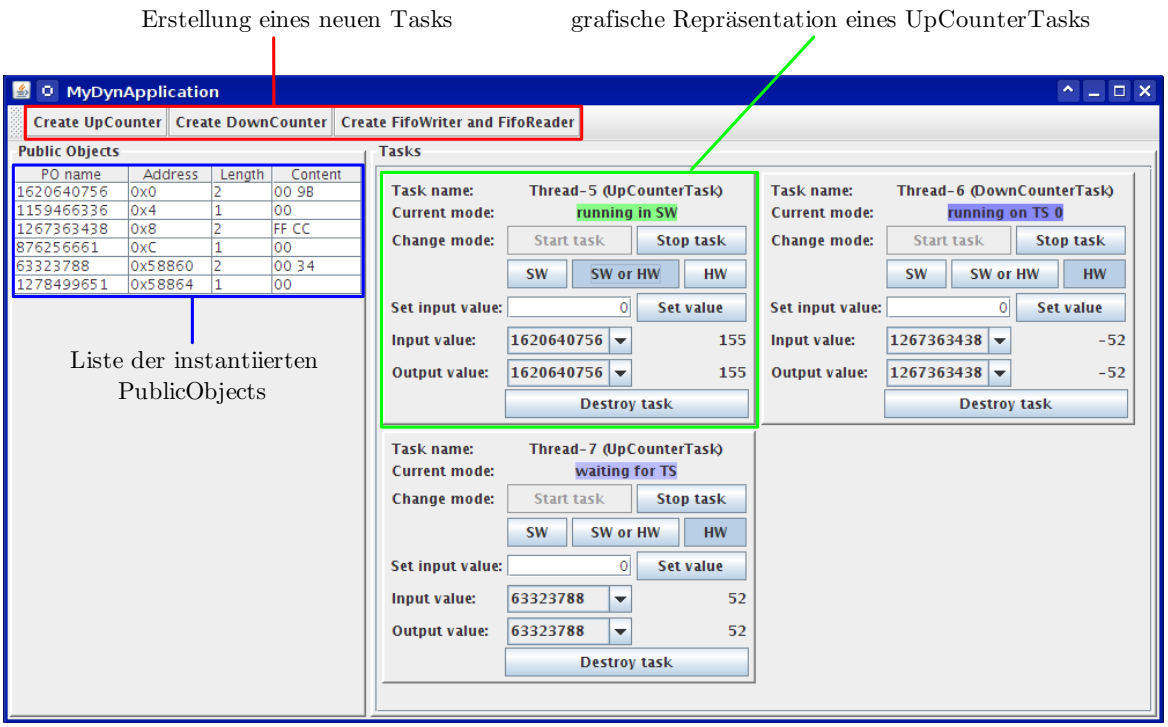


Abbildung 6.2: MyDynApplication mit zwei UpCounterTasks und einem DownCounterTask

Neue Tasks werden per Klick auf einen der oberen drei Buttons erzeugt und eine grafische Repräsentation des Tasks wird in die Liste der Tasks (in der Abbildung rechts zu sehen) dargestellt. Zu jedem Task gibt es Buttons, um den Ausführungsmodus zu ändern oder andere Manipulationen am Task vorzunehmen. Abbildung 6.3 erläutert dazu die einzelnen GUI-Elemente. Links neben der Task-Liste befindet sich die Liste der instantiierten PublicObjects. Alle erstellten PublicObjects werden automatisch in die Liste aufgenommen und einmal pro Sekunde durch die Anwendung ausgelesen, um die *Content*-Spalte der Liste zu aktualisieren. Sobald auf ein PO keine Referenz mehr existiert und durch den Garbage-Collector entfernt wird, verschwindet

es auch aus der Liste.

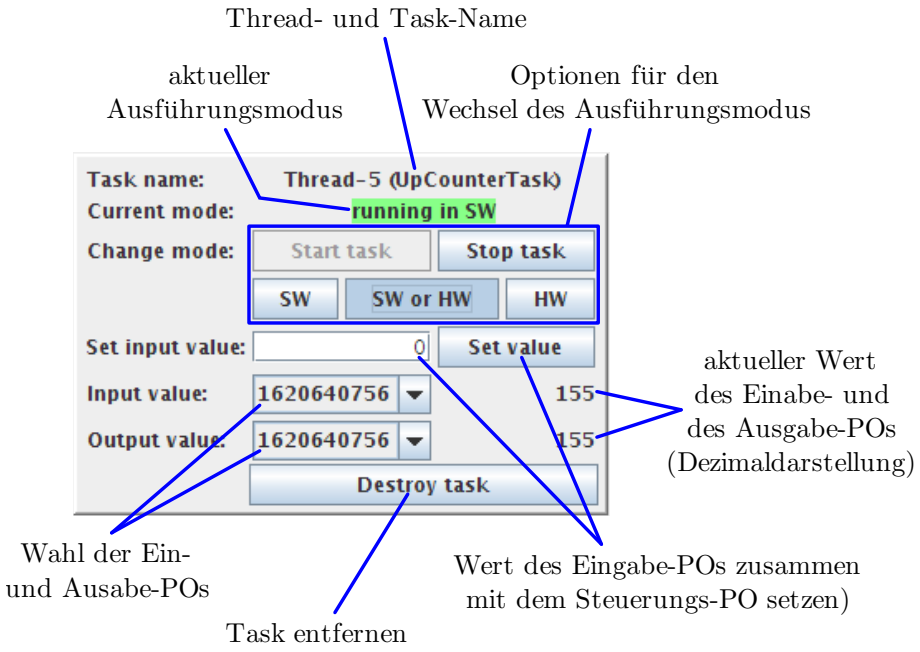


Abbildung 6.3: GUI des UpCounterTasks in der Anwendung MyDynApplication

Zum Testen des PublicFifo-Konzeptes wurde der `FifoUpCounterTask` geschaffen. Er liest einzelne Bytes aus einer PublicFifo, inkrementiert diese und schreibt die neuen Werte in eine zweite PF. Um diesen Task mit Eingabe-Daten zu versorgen, wurde zusätzlich die Klasse `FifoWriterTask` geschrieben. Sie nimmt die Benutzereingaben entgegen und schreibt die eingegebenen Zahlenwerte als Bytes in eine PublicFifo. Ebenso wurde für die Ausgabe der Byte-Werte der `FifoReaderTask` programmiert, der die Daten aus einer PF liest. Die Beiden Tasks werden gemeinsam beim Druck auf den Button *Create FifoWriter and FifoReader* erstellt und sind zu Beginn direkt miteinander verbunden, d.h. die Eingabedaten werden ohne weitere Verarbeitung wieder durch den `FifoReader` ausgegeben. Durch jeden Klick auf den Button *Insert FifoUpCounter before ...* wird ein `FifoUpCounterTask` in die Kette der Verarbeitungsschritte vor dem `FifoReader` eingefügt.

Abbildung 6.4 zeigt die Anwendung mit zwei FifoUpCounterTasks zusammen mit den Ein- und Ausgabe-Tasks, wobei der Datenfluss zwischen den Task folgendermaßen aussieht:

$$\begin{array}{ccccccc} \text{FifoWriter} & & \text{FifoUpCounter} & & \text{FifoUpCounter} & & \text{FifoReader} \\ \text{(Thread-12)} & \rightarrow & \text{(Thread-13)} & \rightarrow & \text{(Thread-14)} & \rightarrow & \text{(Thread-15)} \end{array}$$

Wie in der Abbildung dargestellt, wurde in diesem Beispiel die Zahlenfolge $0, 1, 2, 3$

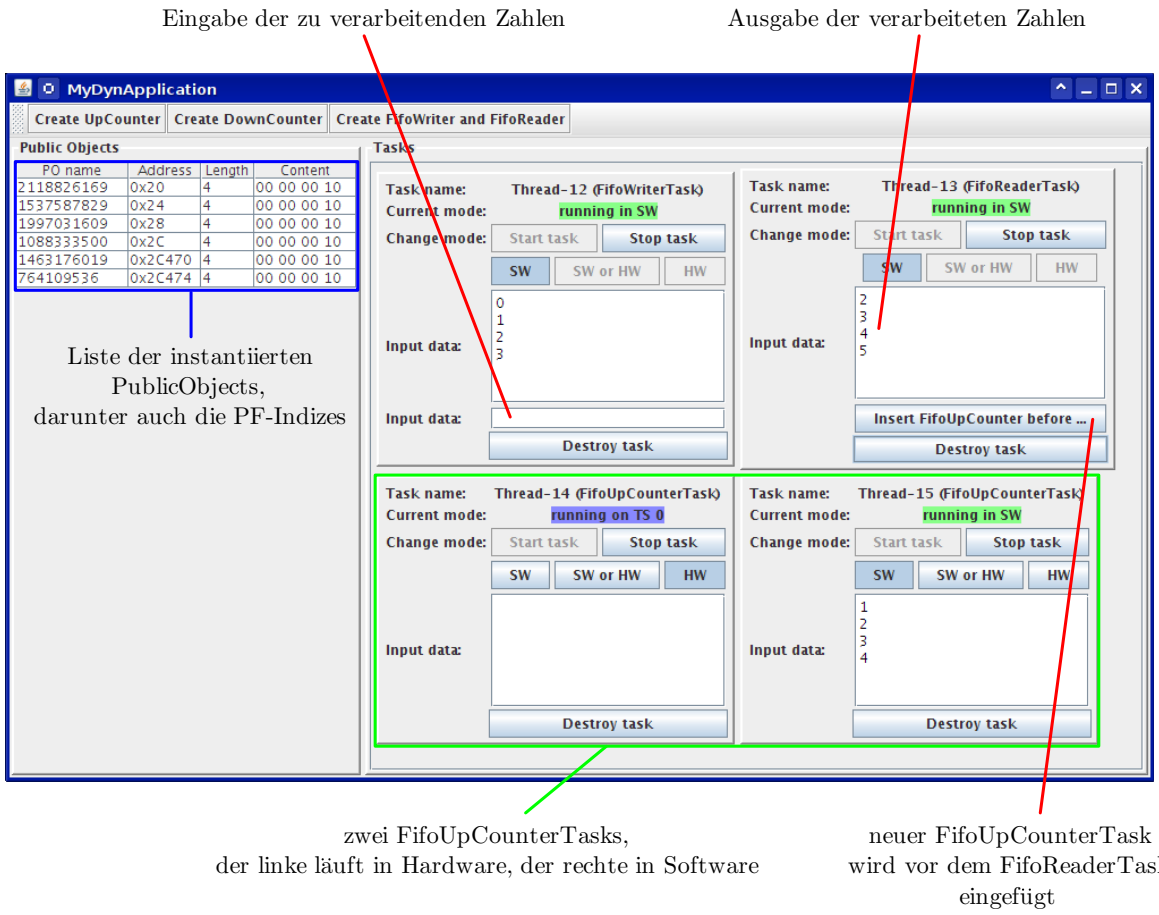


Abbildung 6.4: MyDynApplication mit zwei FifoUpCounterTasks

dem FifoWriterTask übergeben. Jeder Wert durchlief dann insgesamt drei FIFOs und wurde durch die beiden FifoUpCounterTasks jeweils einmal inkrementiert, sodass der FifoReaderTask die Folge $2, 3, 4, 5$ ausgab. Dabei nutzten die 3 FIFOs einen gemeinsamen FIFO-Puffer. Da die PF-Indizes durch PublicObjects realisiert worden sind, lassen sich deren Änderungen ebenfalls in der Liste der PublicObjects verfolgen.

Mit wenigen Tasks kommt die Anwendung sehr gut zurecht. Beachtet werden muss jedoch, dass mit jedem weiteren Task auch weitere PublicObjects erstellt werden und somit die serielle Schnittstelle schon bald komplett ausgelastet ist aufgrund der regelmäßigen Aktualisierung der PO-Liste.

6.3 Performance-Messergebnisse

Zur Ermittlung der Performance des Frameworks wurden einige Messungen auf dem PC und auf dem Board durchgeführt. In Java wurden unter anderem die Zeiten für

die Ausführung von SDRAM-Zugriffen, einer Rekonfiguration des Task-Slots und die Zeit zum Starten eines Tasks gemessen. Die Ergebnisse sind in Abbildung 6.5 dargestellt.

Aktion	**zu übertragene Bytes**	**durchschnittliche Ausführungszeit**
sende Ping, warte auf Pong	12	20 ms
lese 4 Bytes aus dem SDRAM	24	39 ms
schreibe 4 Bytes in den SDRAM	20	39 ms
lese 100 Bytes aus dem SDRAM	120	140 ms
schreibe 100 Bytes in den SDRAM	116	140 ms
Rekonfiguration des Task-Slots	16	44 ms
starte einen `UpCounterTask`	76	143 ms

Abbildung 6.5: Durchschnittliche Ausführungszeiten ausgewählter Aktionen, die durch den PCs ausgeführt werden. Die zweite Spalte gibt die Anzahl der Bytes an, die während der Aktion über die serielle Schnittstelle übertragen werden.

Aus den Ping-Pong-Nachrichten, die weder unter Java noch auf dem Board weitere Verarbeitungsschritte erfordern, ist ersichtlich, dass der geringe Datendurchsatz der seriellen Schnittstelle die Performance wesentlich bestimmt. Die langsame Datenübertragung beeinträchtigt neben der Inter-Task-Kommunikation auch den Task-Wechsel auf dem FPGA. So umfasst der Rekonfigurationsprozess zusammen mit den dafür nötigen Nachrichten eine Dauer von 44 ms. Für den Start eines Tasks werden zu diesen Nachrichten zusätzlich zwischen vier und sechs weitere Nachrichten übertragen, die PO- und PF-Daten und die Aktivierung des Hardware-Tasks betreffen. Wie am Beispiel des `UpCounterTasks` zu sehen ist, werden deshalb für den Start eines Hardware-Tasks im Schnitt 143 ms benötigt. Der Anteil des eigentlichen Rekonfigurationsvorgangs an diesem Wert ist verschwindend gering. Direkte Messungen auf dem XUP-Board ergaben, dass die komplette Rekonfiguration des 117 kB großen Task-Slots in 11,1 Millisekunden durchgeführt wird. Dies entspricht einem Datendurchsatz von 16 MB/s. Folglich wären bei einer Beschleunigung der PC-Board-Kommunikation deutlich schnellere Task-Wechsel möglich.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Das Ziel der Arbeit bestand in der Entwicklung eines Frameworks, mit welchem sich thread-ähnliche Tasks implementieren lassen, die sowohl in Software als auch in rekonfigurierbarer Hardware ausgeführt werden können. Dieses Ziel wurde mit der hierfür entwickelten Prototyp-Implementierung erfolgreich umgesetzt. In dieser können neue Tasks durch die Ableitung von `AbstractTask` geschaffen werden. Sie besitzen die gleichen Methoden wie Threads, verfügen aber zusätzlich über die Fähigkeit, ihren Ausführungsmodus zur Laufzeit zu ändern. Für das Scheduling der Tasks auf dem FPGA wurde in Java ein Scheduler implementiert, der in der Lage ist, die Tasks in Abhängigkeit von den freien FPGA-Ressourcen entweder in Software oder in Hardware auszuführen. Des Weiteren wurde eine effektive, objektbezogene Inter-Task-Kommunikation implementiert, die beliebig vielen Tasks die Möglichkeit zur Synchronisation ihrer Zugriffe bereitstellt.

Die Inter-Task-Kommunikation wurde über einen gemeinsamen Speicher realisiert. Der Zugriff auf diesen erfolgt für die Software- und Hardware-Tasks objektbezogen, d.h. das Framework ist für die Umsetzung der Objektzugriffe in Zugriffe auf den gemeinsamen Speicher verantwortlich. Somit konnte die Kommunikationsschnittstelle der Tasks sehr einfach gestaltet werden. Neben der Kommunikation über gemeinsame Objekte (PublicObjects) stehen den Tasks zum Austausch größerer Datenmengen gemeinsame FIFOs (PublicFifos) zur Verfügung.

Zur Vermeidung von Inkonsistenzen im gemeinsam genutzten Speicher musste eine Lösung zur Synchronisation der Zugriffe geschaffen werden. Dabei kamen sowohl software- als auch hardwareseitige Sperrmechanismen zum Einsatz. Indem die Sperrung des Speichers nur erfolgt, wenn ein Task einen entsprechenden Bedarf meldet,

war es möglich, die Sperrzeiten auf ein Minimum zu reduzieren.

Zu Demonstrationszwecken wurde unter anderem eine interaktive grafische Beispielanwendung geschaffen, über die der Benutzer verschiedene Tasks instantiieren, starten und stoppen kann. Darüber hinaus können die Zustände der Tasks überwacht und die Ausführungsmodi geändert werden.

7.2 Möglichkeiten zur Weiterentwicklung

Um das Framework in der Praxis einzusetzen, wäre zunächst eine Beschleunigung der Kommunikation zwischen PC und Board erforderlich. Statt wie bisher über die serielle Schnittstelle könnte die Kommunikation über USB oder Ethernet stattfinden.

Alternativ könnte das gesamte Framework auf dem Board untergebracht werden. Mit der Einrichtung eines Betriebssystems ließe sich die Java-Anwendung auch auf dem PowerPC ausführen. Gleichzeitig sollte man über eine Trennung zwischen Hauptspeicher, gemeinsamen Datenspeicher und Rekonfigurationsspeicher nachdenken, um die Performance des Systems zu steigern.

Ergänzend könnten weitere Task-Slots eingerichtet werden, um die Ausführung der Task-Anwendungen zu beschleunigen.

Für weitere Performancesteigerungen wäre es denkbar, das Framework um Laufzeitanalysen zu erweitern. Sinnvoll wäre es, zu jedem Task die Datendurchsätze für die Ausführung in Software und in Hardware zu ermitteln. Diese Werte könnten die Grundlage für einen Task-Scheduler bilden, der die Task-Auswahl für die Ausführung in Hardware hinsichtlich einer optimalen Gesamtperformance des Systems trifft.

Aktuell ist für die Spezifikation eines Tasks eine Implementierung in Java und und VHDL erforderlich, damit sich der Task sowohl in Software als auch in Hardware ausführen lässt. Um diesen doppelten Implementierungsaufwand überflüssig zu machen, könnten sich weiterführende Arbeiten mit einer automatisierten Übersetzung des Java-Quellcodes in verhaltensgleiche Hardwarebeschreibungen beschäftigen. Mit der Umsetzung dieses Ansatzes könnten Task-Anwendungen zukünftig komplett in der Programmiersprache Java entwickelt werden.

7.3 Ausblick

Mit der Beseitigung des bereits erwähnten Falschenhalses wären verschiedene praxisrelevante Einsatzmöglichkeiten für das in dieser Arbeit entwickelte Framework vorstellbar. Beispielsweise könnten viele mobile Geräte, die meist mit leistungsschwa-

chen Prozessoren ausgestattet sind, von der Möglichkeit einer beschleunigten Ausführung beliebiger Programmteile stark profitieren. Statt ASICs für die Implementierung rechenintensiver Programmteile zu verwenden, würde es der Einsatz eines FPGAs ermöglichen, beliebige Programmteile für eine Beschleunigung in Hardware vorzusehen. Durch das geschaffene Framework ließe sich in Java komfortabel steuern, ob in Hardware gerade ein MP3-Decoder zum Abspielen von Musik, ein Verschlüsselungsverfahren oder eine Infrarot-Steuerung benötigt würde. Im Falle einer gleichzeitigen Ausführung mehrerer Tasks, könnten sich die zeitkritischen Tasks per Hardware-Scheduling den FPGA teilen und die restlichen Tasks die CPU nutzen. Dank der funktionierenden Inter-Task-Kommunikation wären die Tasks auch in der Lage, untereinander Daten auszutauschen.

Das entwickelte Framework stellt hierzu ein Fundament für weitere Entwicklungen bereit. Durch die Implementierung eines gerätespezifischen `BoardControllers` und die Portierung des initialen Designs ließe sich das Framework auch auf anderen Gerät nutzen.

Glossar

Block RAM

Ein 18 KBit größer RAM-Block innerhalb eines Virtex-II-FPGAs.

Configurable Logic Block

Xilinx-Bezeichnung für einen Logik-Block. Funktionales Element in logischen Schaltkreisen. Setzt sich beim Virtex-II Pro aus vier Slices zusammen, die je zwei Logikzellen enthalten.

Input/Output Block

Ein- und Ausgabe-Blöcke für die Verbindung zwischen der Schaltung auf dem FPGA und den Pins des Chips (der Außenwelt des FPGAs)

Java Runtime Environment

Stellt die Java-Bibliotheken, die Java Virtual Machine und weiter Komponenten bereit, die zur Ausführung von Java-Anwendungen und -Applets benötigt werden.

Java Virtual Machine

Ist der Teil des Java Runtime Environment, der für die Ausführung des Java-Bytecodes verantwortlich ist.

Kilo Virtual Machine

Java Virtual Machine, die von Sun speziell für ressourcen-beschränkte Umgebungen entwickelt wurde.

Abkürzungsverzeichnis

ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
BSB	Base System Builder
CDFG	Control-Data Flow Graph
CLB	Configurable Logic Block
CPU	Central Processing Unit
DCM	Digital Clock Manager
DMA	Direct Memory Access
EDK	Embedded Development Kit
EEPROM	Electrically Erasable Programmable Read Only Memory
FIFO	First-in-First-out
FPGA	Field Programmable Gate Array
FSM	Finite state machine
GC	Garbage-Collector
HDL	Hardware-Beschreibungssprache
HoS	Hardware-oder-Software
ICAP	Internal Configuration Access Port
IOB	Input/Output Block
IOI	Input/Output Interconnection

IP-Core	Intellectual Property Core
ISE	Integrated Synthesis Environment
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KVM	Kilo Virtual Machine
LUT	Look-Up-Table
NCD	Native Circuit Description
OPB	On-Chip Peripheral Bus
PC	Personal Computer
PF	PublicFifo
PLA	Programmable Logic Array
PLB	Processor Local Bus
PO	PublicObject
PPC	PowerPC
PROM	Programmable Read-Only Memory
PUF	PF-Update-FIFO
SoC	System-on-a-Chip
TBUF	Tristate Buffer
UART	Universal Asynchronous Receiver and Transmitter
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XDL	Xilinx Design Language

XPS	Xilinx Platform Studio
XUP	Xilinx University Program

Abbildungsverzeichnis

Literaturverzeichnis

[Abe05] Abel, Norbert: *Schnelle dynamische partielle Rekonfiguration in Hardware mit Inter-Task-Kommunikation.* Diplomarbeit, Universität Leipzig, Fakultät für Mathematik und Informatik, Institut für Informatik, 2005. 8, 47, 48, 49, 50, 90, 121, 122

[BKS00] Bazargan, Kiarash, Ryan Kastner und Majid Sarrafzadeh: *Fast Template Placement for Reconfigurable Computing Systems.* IEEE Des. Test, 17(1):68–83, 2000. 13, 14

[CZH+07] Claus, Christopher, Bin Zhang, Michael Hübner, Christoph Schmutzler, Jürgen Becker und Walter Stechele: *An XDL-based busmacro generator for customizable communication interfaces for dynamically and partially reconfigurable systems.* In: *Workshop on Reconfigurable Computing Education at ISVLSI 2007*, Porto Alegre, Brazil, Mai 2007. 19, 20, 121

[FBK99] Fleischmann, Josef, Klaus Buchenrieder und Rainer Kress: *Codesign of embedded systems based on Java and reconfigurable hardware components.* In: *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, Seite 26, New York, NY, USA, 1999. ACM. 39, 42

[HO97] Helaihel, Rachid und Kunle Olukotun: *Java as a specification language for hardware-software systems.* In: *ICCAD*, Seiten 690–697, 1997. 37

[LGK+04] Lattanzi, Emanuele, Aman Gayasen, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Luca Benini und Alessandro Bogliolo: *Improving Java Performance by Dynamic Method Migration on FPGAs.* In: *IEEE Reconfigurable Architecture Workshop*, April 2004. 42, 46, 69

[LTG97] Liao, Stan, Steve Tjiang und Rajesh Gupta: *An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment*. DAC, 00:70, 1997. 37

[Mic07] Microsystems, Sun: *Java Thread Primitive Deprecation*. Webseite, Dez. 2007. http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html. 32

[Mic08] Microsystems, Sun: *Java Logging Overview*. Webseite, Jan. 2008. http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html. 99

[PEW+02] Plessl, Christian, Rolf Enzler, Herbert Walder, Jan Beutel, Marco Platzner und Lothar Thiele: *Reconfigurable Hardware in Wearable Computing Nodes*. In: *Proc. Int. Symp. on Wearable Computers (ISWC'02)*, Seiten 215–222. IEEE, Okt. 2002. 12

[Ste04] Steinegger, Simon: *Reconfigurable Hardware OS Prototype - Part FPGA*. Diplomarbeit, Swiss Federal Institute of Technology Zurich, Mai 2004. 14

[Tho04] Thorvinger, Jens: *Dynamic Partial Reconfiguration of an FPGA for Computational Hardware Support*. Diplomarbeit, Lund Institute of Technology, Lund University, Juni 2004. http://jens.thorvinger.se/DynPartReconf.pdf. 83

[Ull07] Ullenboom, Christian: *Java ist auch eine Insel*. Galileo Computing, Bonn, 6., aktualisierte und erweiterte Auflage, 2007. http://www.galileocomputing.de/openbook/javainsel6/. 28, 33, 34, 121

[WNP04] Walder, Herbert, Samuel Nobs und Marco Platzner: *XF-Board: Prototype Platform for Reconfigurable Hardware Operating System*. In: *Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Juni 2004. 15

[WP03] Walder, Herbert und Marco Platzner: *Reconfigurable Hardware OS Prototype*. TIK 168, Swiss Federal Institute of Technology (ETH), Zurich, Apr. 2003. 15

[Xila] Xilinx: *Constraints Guide*, 8.2.2 Auflage. Datei: ISE8.2/doc/usenglish/books/docs/cgd/cgd.pdf. 81

[Xilb] XILINX: *Development System Reference Guide, Chapter 5, Partial Reconfiguration*, 8.1i Auflage. Datei: ISE8.1/doc/usenglish/de/dev/partial.pdf. 18, 121

[Xil97] XILINX: *Application Note 024*, 1.0 Auflage, Nov. 1997. http://www.xilinx.com/support/documentation/application_notes/xapp024.pdf. 19

[Xil05] XILINX: *Xilinx University Program Virtex-II Pro Development System - Hardware Reference Manual*, 1.0 Auflage, März 2005. http://www.xilinx.com/univ/xupv2p.html. 57

[Xil07a] XILINX: *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, Nov. 2007. http://www.xilinx.com/support/documentation/user_guides/ug012.pdf. 8, 9, 14, 57, 121

[Xil07b] XILINX: *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, v4.7 Auflage, Nov. 2007. http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf. 5, 121

[Xil08] XILINX: *Modular Design Tips*. Webseite, März 2008. http://toolbox.xilinx.com/docsan/xilinx7/books/data/docs/dev/dev0032_7.html. 82

Anhang A

Quelltexte

Da allein die Java-Quelltexte der Arbeit rund 7000 Codezeilen zählen[1], wurde darauf verzichtet, den Quellcode im Anhang abzubilden. Zum Nachschlagen der Quelltexte wird auf die beigelegte CD verwiesen. Im Folgenden werden alle wichtigen Projektverzeichnisse und -dateien beschreiben.

Verzeichnis bzw. Datei	Beschreibung
`build/classes/`	kompilierten Java-Klassen
`conf/`	Konfigurationsdateien
`beanshellwindow/`	Einstellungen für das BeanShellWindow
`replacements.list`	benutzerdefinierte Textersetzungen im BeanShellWindow
`xupcontroller.properties`	BeanShellWindow-Einstellungen, um mit dem XUPController zu arbeiten
`logging.properties`	Logging-Einstellungen
`virtualController.properties`	Framework-Einstellungen für die Simulation in Software
`xupController.properties`	Framework-Einstellungen für die Task-Ausführung mit XUP-Board
`doc/`	kompakte Dokumentation einiger Aspekte der Arbeit
`edkProject/`	EDK-Projekt-Verzeichnis
`AppTman/src/`	C-Quellen des PowerPC-Programms
`AppTman_sw.c`	PowerPC-Programm
`icap_comm.c`	Bibliothek zur Ansteuerung des TMANs

[1]Dieser Wert wurde mit Hilfe des freien Programms SLOCCount ermittelt.

Verzeichnis bzw. Datei	Beschreibung
mystdio.c	Bibliothek zur Kommunikation über die serielle Schnittstelle
data/system.ucf	Festlegung der externen Pins und der System- und Task-Area, Platzierung der Busmacros
pcores/Rahmen/hdl/	
sim_vhdl/	VHDL-Dateien, die ausschließlich zur Simulation des Task-Rahmens und des Tasks genutzt werden
vhdl/	VHDL-Dateien des Task-Rahmens und Tasks
Rahmen.vhd	VHDL-Code des Task-Rahmens
TASK.vhd	VHDL-Code des Tasks
pcores/TMANv2/hdl/vhdl/	VHDL-Dateien des TMANs
ICAP_cntrl.vhd	VHDL-Code des ICAP-Controllers, der durch den TMAN angesteuert wird
TMAN.vhd	VHDL-Code des TMAN
implementation/	
download.bit	Bitfile des Hardware-Designs zusammen mit dem PowerPC-Programm
system.bit	Bitfile des Hardware-Designs ohne das PowerPC-Programm
system.ncd	Native Circuit Description zum Design, wird zur Betrachtung im FPGA-Editor benötigt
system.pcf	Physical Constraints File zum Design, wird zur Betrachtung im FPGA-Editor benötigt
download.sh	Script für die Synthese und Übertragung des Bitstreams auf das Board
download_color.sh	macht das Gleiche wie download.sh, nur mit farblicher Hervorhebung der ERROR- und WARNING-Zeilen
settings.sh	Script zum Laden der Xilinx-Umgebungsvariablen

Verzeichnis bzw. Datei	Beschreibung
`export/`	temporär angelegtest Verzeichnis, um das komplette Projekt in ein Zip-Archiv zu packen
`javadoc/`	Dokumentation der Java-Klassen im JavaDoc-Format
`lib/`	Verwendete Java-Bibliotheken
`bsh.jar`	BeanShell-Bibliothek
`junit.jar`	JUnit-Bibliothek
`RXTXcomm.jar`	Bibliothek für die serielle Schnittstelle
`UmlGraph.jar`	Bibliothek zur Generierung der Klassendiagramme in der JavaDoc-Dokumentation
`scripts/`	Scripte mit Java-Code, die im BeanShellWindow geladen werden können
`src/`	Java-Quellen
`task/`	Quellen und Bitstreams der Tasks
`bitstream/`	Bitstreams der Tasks
`design/`	FPGA-Designs Tasks
`sim/`	Simulationsdaten der Tasks
`source/`	VHDL-Quelltexte der Tasks
`waveform/`	GtkWave-Diagramme für Darstellung der Signaländerungen während der VHDL-Simulation der Tasks
`bitstreamGenerator.sh`	Script, das durch den `BitstreamsGenerator` aufgerufen wird, um für einen Task das komplette Hardware-Design in ein Bitfile zu übersetzen
`build.xml`	Ant Build Script zur Kompilierung und Ausführung der Java-Anwendungen
`decompressBitstream`	gibt einen komprimierten Task-Bitstream unkomprimiert in hexadezimaler Schreibweiese auf der Konsole aus

Verzeichnis bzw. Datei	Beschreibung
`download.sh`	lädt das aktuelle Bitfile im EDK-Projekt auf den FPGA und aktualisiert es vorher wenn nötig
`includes.make`	Umgebungsvariablen für die VHDL-Simulation
`makefile`	Makefile für die VHDL-Simulation

Anhang B

Installation

Entwickelt und getestet wurde diese Arbeit unter Linux. Prinzipiell ist es dank Java aber auch möglich, das Framework unter Windows einzusetzen. Der Aufwand für die Einrichtung unter Windows ist jedoch um ein Vielfaches größer als unter Linux. Alle unten aufgeführten Programme müssten manuell installiert werden. Unter Linux, speziell unter Debian oder Ubuntu, reicht für die Einrichtung die Ausführung der folgenden Kommandozeile:

```
sudo aptitude install sun-java5-jdk ant ghdl gtkwave make \
zsh bash sed grep coreutils subversion
```

B.1 Benötigte Software

Für die manuelle Installation folgt nun die Liste der benötigten Software:

- Xilinx[1] ISE und EDK 8.2i
- mindestens Java[2] 1.5
- Apache Ant[3], falls man das Projekt mit Ant compilieren, starten, testen, aktualisieren oder exportieren möchte (empfohlener Weg)
- GHDL[4] 0.25 oder besser, falls man das Verhalten der Tasks und des Task-Rahmens simulieren möchte

[1] http://www.xilinx.com/
[2] http://java.sun.com/
[3] http://ant.apache.org/
[4] http://ghdl.free.fr/

- GTKWave[5] 3.0 oder besser; zur Darstellung der Signale während der Simulation
- Bash[6] 3.0 oder besser, da einige Scripte, insbesondere die von Xilinx, die Bash voraussetzen
- Zsh[7] 4.3 oder besser; wird nur in dem Script `edkProject/download_color.sh` verwendet, um die Fehler der Synthese farbig hervorzuheben
- Standard-Linux-Tools (cat, tee, usw.) + sed, grep, lsusb und make
- Subversion-Client[8], für inkrementelle Updates des Projekts

B.2 Durchführung der Installation

- Das Archiv `implementierung.zip` entpacken
- Ausführungsrechte der Shell-Scripte setzen:

```
chmod +x bitstreamGenerator.sh
chmod +x decompressBitstream
chmod +x download.sh
chmod +x edkProject/settings.sh
```

- Die Datei `edkProject/settings.sh` öffnen und in folgenden Zeilen die Installationspfade vom ISE und EDK auf die gegenwärtigen Verzeichnisse setzen:

```
export XILINX_ISE=/opt/ise
export XILINX_EDK=/opt/edk
```

- Die Datei `conf/xupController.properties` öffnen und in der folgenden Zeile den Pfad auf das Gerät der seriellen Schnittstelle setzen:

```
serial.port.linux = /dev/ttyS0
```

- Board einschalten und evtl. ein Initialisations-Script zur Boarderkennung starten.
- Testen, ob der Download des initialen Designs funktioniert:

[5] http://home.nc.rr.com/gtkwave/
[6] http://www.gnu.org/software/bash/
[7] http://www.zsh.org/
[8] http://subversion.tigris.org/

```
./download.sh
```

- Einmal alle Tests ausführen, um sicherzustellen, dass alles funktioniert (dauert mehrere Minuten):

  ```
  ant runAllTests
  ```

  Wenn bei allen Tests `Failures: 0, Errors: 0` ausgegeben wird, kann fortgefahren werden.

- Beispiel-Anwendung `MyDynApplication` starten:

  ```
  ant runMyDynApplication
  ```

- Die Bitstreams zu den Tasks liegen unter `task/bitstream`. Der Inhalt des Verzeichnisses kann jederzeit gelöscht werden. Bei Bedarf werden die Bitstreams neugeneriert. Die Generierung beeinflusst jedoch das Verhalten der Anwendung. Für zuverlässige Ergebnisse muss die Anwendung daher nach dem Durchlauf zur Generierung der Bitstreams neu gestartet werden.

Anhang C

Register des Task-Rahmens

Der Task-Rahmen belegt einen 16-Bit-Adressbereich auf dem OPB. In diesem Bereich können verschiedene Register des Task-Rahmen geschrieben oder gelesen werden. Im Folgenden soll die Bedeutung jedes Registers anhand seiner Adresse vorgestellt werden:

- 0x0000 - wird nicht mehr verwendet
- 0x0004 - allgemeines Kommando-Register
  Die Art des Kommandos wird über das Datenwort auf dem `OPB_DBus` übermittelt:
  - 0x00000000 - Task wird deaktiviert (wird angehalten)
  - 0x00000001 - Task wird aktiviert (läuft weiter) ohne Reset
  - 0x00000003 - Task wird aktiviert (läuft weiter) mit Reset (genau einen Takt lang)
  - 0x00000004 - deaktiviert den SDRAM-Zugriff des Tasks
  - 0x00000005 - aktiviert den SDRAM-Zugriff des Tasks; ist unabhängig davon, ob gerade ein Task läuft
- 0x0008 - Zustandsausgabe des Task-Rahmens über die LEDs; aktuell auskommentiert, da der Task nun einen direkten Zugriff auf die LEDs besitzt
- 0x000C - Register zum Testen der Adressauflösung für POs/PFs. Das Schreiben in dieses Register veranlasst den Rahmen anhand der PO/PF-Nummer auf dem OPB_DBus alle entsprechenden PO/PF-Register zu setzen, so dass das PO bzw. die PF zum Auslesen oder Schreiben bereits ist. Die PO/PF-Nummer befindet sich in den letzten sieben Bit des `OPB_DBus`. Das erste Bit des `OPB_DBus` gibt an, ob es sich dabei um eine PF statt einem PO handelt.

- 0x0100 - liefert das Register `ADDR` zurück
- 0x0104 - liefert das Register `ADDR_END` zurück
- 0x0108 - liefert das Register `IS_PF & X"000000" & PO_PF_NR` zurück
- 0x0110 - liefert das Register `PF_START_ADDR` zurück
- 0x0114 - liefert das Register `PF_END_ADDR` zurück
- 0x0118 - liefert das Register `PF_ADDR_PREFIX` zurück
- 0x011C - liefert das Register `PF_BITMASK` zurück
- 0x0120 - liefert das Register `PF_RW_MASK` zurück
- 0x0124 - liefert das Register `PF_ITER_MASK` zurück
- 0x1*** - Zugriff auf den BRAM des Task-Rahmens, der die PO/PF-Eigenschaften speichert. Von den letzten elf Bit der OPB-Adresse werden die ersten neun Bit für die Adressierung des BRAMs benutzt. Der BRAM lässt sich über den OPB sowohl beschreiben als auch auslesen.

Anhang D

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, den 27. April 2008

Stefan Endrullis