TECHNISCHE UNIVERSITÄT WIEN

DIPLOMARBEIT

Parallele VHDL Simulation mit einem Standard Hardwaresimulator

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Diplom-Ingenieurs unter der Leitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Karl Riedling
und Dipl.-Ing. Dr.techn. Gerhard R. Cadek

E 366
Institut für Industrielle Elektronik und Materialwissenschaften

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik

von

Martin Matschnig
Matrikelnummer 9325875
Auhofstraße 4/3, 3423 Wördern

Wien, im März 2001 Unterschrift

Kurzfassung

Diese Diplomarbeit beschreibt eine Möglichkeit zur *parallelen VHDL Simulation* auf *Systemebene* in der *HW/SW Coverification*[1] unter Verwendung des sequentiellen Hardwaresimulators *ModelSIM* [2]. Dabei wird als Beispiel ein typisches industrielles *Multiboard Design* zerlegt, und die so entstandenen Partitionen werden parallel auf einer eigenen Workstation simuliert. Die entwickelte Lösung verwendet zur Kommunikation zwischen den Partitionen *TCP/IP Socketverbindungen.* Die Schnittstelle zwischen *TCP/IP* und dem Hardwaresimulator stellt das *FLI-Interface*[3] dar. In einer konkreten Implementierung konnte für ein spezielles Beispiel eine fast direkte Proportionalität zwischen dem Geschwindigkeitsgewinn und der Anzahl der Partitionen durch Messwerte belegt werden. Die Arbeit wurde in Kooperation mit der *Siemens AG* durchgeführt, und nimmt daher auch Bezug auf einige abgeschlossene Siemens Projekte im Bereich des *EWSD*[4].

Als Einführung werden die grundlegenden Konzepte der *Diskreten Event Simulation (DES)* und der *Parallelen Diskreten Event Simulation (PDES)* erklärt. Dabei wird im Speziellen auf die Unterschiede zwischen *konservativen Methoden (Chandy-Misra Algorithmus)* und *optimistischen Ansätzen (Time Warp)* eingegangen. Diese Möglichkeiten werden in Hinblick auf die *parallele VHDL Simulation* gegenübergestellt.

Keywords

Parallele Simulation, Verteilte Simulation, konservativ, optimistisch, Chandy-Misra, Time Warp, VHDL, FLI, EWSD

---

[1] Hardware/Software Coverification

[2] ModelSIM SE/EE PLUS 5.4b von Model Technology

[3] Foreign Language Interface

[4] Europäsches Wähl System Digital

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die moderne Hardwareentwicklung ist geprägt durch das ständige Verlangen nach mehr Geschwindigkeit in der Hardwaresimulation. Dies ist bedingt durch die rasante Entwicklung in allen Bereichen der Mikroelektronik, die durch das erste Moor'sche Gesetz[22] beschrieben wird. Demnach verdoppelt sich die Komplexität der realisierten integrierten Schaltungen alle 18 Monate. Selbst der beachtliche Anstieg der Rechenleistung moderner Workstations wird leider gänzlich durch die wachsende Komplexität aktueller Hardwareentwürfe kompensiert. Besonders bei der Simulation auf Boardebene, bei welcher ganze Baugruppen, bestehend aus einer Vielzahl einzelner Komponenten simuliert werden, spielt die Simulationsgeschwindigkeit eine entscheidende Rolle. So ist es heute durchaus üblich, dass einzelne Hardwaresimulationen, sogar auf den modernsten Workstations, eine Laufzeit von mehreren Wochen benötigen. Dies schlägt sich auch darin nieder, dass oft bis zu 80 Prozent der Entwicklungszeit und auch ein beträchtlicher Teil der Entwicklungskosten auf die Simulation entfallen.

Ein Geschwindigkeitsgewinn durch den Einsatz neuer Werkzeuge ist in nächster Zeit allerdings noch nicht absehbar.

Ein Ansatz zur Geschwindigkeitssteigerung in der Hardwaresimulation ist der Einsatz der *parallelen Simulation*. Parallele oder verteilte Simulation steht seit über zwanzig Jahren im Zentrum des Interesses einer großen Zahl von Forschungsaktivitäten [20]. In vielen Bereichen der Simulation werden parallele Methoden heute ganz selbstverständlich eingesetzt.

Auch das Thema der *parallelen Simulation* von Hardwareentwürfen, wird seit geraumer Zeit untersucht. Eine Vielzahl unterschiedlicher Lösungsansätze existiert bereits[16]. Da nebenläufige Prozesse [1] ein Kernelement der Hardwarebeschreibungs-

---

[1] vgl. concurrent statements

sprache VHDL[2] sind, eignet sich diese besonders zur Parallelisierung.

Seit Aufkommen der VHDL Simulation gab es zwei Quantensprünge in der Simulationsperformance. Ersterer war die Einführung von Simulatoren, die die HW-Beschreibung in compilierter Form simulierten. Diese lösten die bis dahin gebräuchlichen Interpreter-basierten Simulatoren ab. So brachte etwa der Umstieg auf compilierende Simulatoren einen mehr als 15-mal höheren Simulationsdurchsatz. Ein zweiter Quantensprung erfolgte mit der Einführung der zyklenbasierten Simulatoren. Diese reduzieren zwar die Simulationsgenauigkeit auf die Zeitpunkte der Taktflanken, erreichen dadurch aber eine wesentlich bessere Performance.

Leider gibt es bis heute keinen kommerziellen Hardwaresimulator der parallele Methoden ausnützt und für aufwendige Entwürfe mit mehreren Millionen Gattern, wie sie für die Simulation auf Baugruppenebene typisch sind, geeignet ist. Der Grund dafür ist sicher nicht im mangelnden Bedarf zu suchen, sondern in der Komplexität der Aufgabe.

Diese Diplomarbeit wurde in Kooperation mit der Siemens AG Österreich durchgeführt, mit dem Ziel die zeitaufwendige *Hardwaresimulation* zu beschleunigen. Diese Arbeit verfolgt nicht das Ziel, einen eigenständigen parallelen Simulator für beliebige VHDL Entwürfe zu entwickeln, sondern es wird der Spezialfall der *Multiboard-Simulation auf Systemebene* betrachtet. Dadurch ergeben sich bestimmte Vereinfachungen, die die Parallelisierung erleichtern.

Die Grundidee ist, einen vorhandenen, ursprünglich nicht parallel arbeitenden, Hardwaresimulator für parallele Simulation zu verwenden. Dazu soll ein vorgegebener VHDL-Entwurf in geeigneter Weise aufgespalten und verteilt über mehrere Workstations simuliert werden. Die eigentliche Aufgabe ist es nun, mit Hilfe der C-Programmierschnittstelle des Simulators ein C-Modell zu entwickeln, welches die nötige Kommunikation zwischen den entstandenen Partitionen übernimmt. Als Grundlage für den Entwurf dieses Modells sollen die aus der Literatur bekannten unterschiedlichen Möglichkeiten für parallele Simulation hinsichtlich ihrer Eignung betrachtet werden.

Anhand von zwei typischen Beispielen aus der industriellen Praxis soll das Konzept überprüft werden. Die daraus gewonnenen Erkenntnisse sollen dann für den Einsatz in zukünftigen Projekten aufbereitet werden.

---

[2] **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

Kapitel 2

Motivation

2.1 Aufgabenstellung

Im Rahmen dieser Diplomarbeit wird ein Ansatz zur *parallelen VHDL-Simulation* unter Verwendung eines sequentiellen Standard Hardwaresimulators auf seine Durchführbarkeit getestet werden. Dabei soll nicht versucht werden, einen neuen Simulator zu entwickeln, sondern es werden existierende und erprobte Hardwaresimulatoren für verteilte Simulation erweitert. Im Idealfall ist die Lösung herstellerunabhängig. Letztendlich soll es möglich sein, einen Entwurf aus mehreren Baugruppen derart aufzuteilen, dass jede einzelne Baugruppe von einem eigenen Simulator auf einer eigenen Workstation simuliert werden kann. Wichtigstes Ziel ist dabei die Verringerung der Laufzeit der Simulation.

2.2 Testgebiet

Die Lösung soll speziell auf die Simulation auf Systemebene zugeschnitten sein. Eine typische Konfiguration ist in Abbildung 2.1 dargestellt.

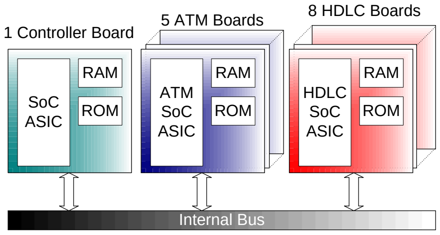


Abbildung 2.1: Eine typische System Architektur

Dabei werden mehrere eigenständige Baugruppen, sogenannte Boards, innerhalb einer Testbench verbunden und das Gesamtsystem wird dann mit einem Simulator simuliert. Verglichen mit der Anzahl interner Signale ist die Zahl der Verbindungssignale, bzw. der Leitungen zwischen den Boards relativ gering. Weiters darf angenommen werden, dass Signalübertragung über die Verbindungen nur mit einem gemeinsamen Takt erfolgt. Asynchrone Signalübertragung zwischen den Baugruppen wird ausgeschlossen. Diese Einschränkung auf synchrone bzw. zyklusorientierte Signalübertragung stellt einen wichtigen Unterschied zu bereits vorhandenen parallelen Simulatoren dar. Die beschriebene Konstellation legt es nahe, die Verbindungen wie in Abbildung 2.2 durch Kommunikationskanäle *(logical channels LCs)* zu ersetzen und die einzelnen Boards getrennt zu simulieren.

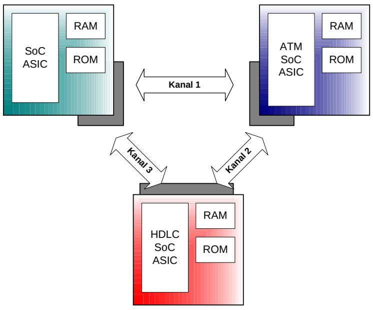


Abbildung 2.2: Verbindung über Kommunikationskanäle

Natürlich müssen die so entstandenen Subsysteme laufend Information über ihren aktuellen Simulationsfortschritt austauschen. Der durch den Informationsfluss entstehende Overhead beeinflusst in entscheidendem Maße den zu erwartenden Performancegewinn.

2.3 Lösungsweg

Aus der Literatur sind zwei grundsätzlich verschiedene Ansätze für parallele Simulation bekannt.

1. konservative Methoden *(Chandy-Misra Algorithmus)*[3]

2. optimistische Methoden *(Time-Warp Algorithmus)*[12][11]

Diese Möglichkeiten werden in Kapitel 3 genau beschrieben. Konservative Methoden stellen durch Synchronisation und streng sequentielle Abarbeitung der Events sicher, dass sich die einzelnen Komponenten zu jedem Zeitpunkt in einem gültigen Zustand befinden. Der optimistische Ansatz arbeitet grundsätzlich anders. Dabei kann es vorkommen, dass sich die Simulation kurzzeitig in einem ungültigen Zustand befindet. Wird dieser entdeckt, wird zu einem gespeicherten Zustand zurückgesprungen, und die Simulation von dort wieder aufgenommen. Dazu ist es notwendig, dass die einzelnen Komponenten laufend Prozessabbilder speichern, um diese bei Bedarf wieder laden zu können.

Grundsätzlich sind beide Möglichkeiten auch zum Einsatz für die parallele VHDL Simulation denkbar, und es existieren für beide Fälle konkrete Implementierungen, allerdings großteils innerhalb universitärer Forschungsprojekte[19][5]. Doch für die gestellte Aufgabe scheiden optimistische Verfahren aus mehreren Gründen aus:

1. Nicht jeder Hardwaresimulator erlaubt es, den Simulationsfortschritt abzuspeichern und dorthin zurückzukehren (fehlendes Rollback Feature).

2. Da die Partitionen aus ganzen Boards bestehen, würde ein Rollback sehr viel Speicherplatz benötigen und auch sehr lange dauern.

3. Innerhalb der Schaltung können auch C-Modelle und Hardware-Modelle eingebunden sein, deren innerer Zustand nicht vom Simulator gespeichert werden kann. Dadurch wird ein Rollback sogar unmöglich.

Für die Implementierung wurde daher ein konservatives Verfahren gewählt. Die oben genannten Einschränkungen, nämlich wenige Signale zwischen den Partitionen und taktsynchrone Signale auf den Leitungen, sind der Ausgangspunkt für die folgenden Betrachtungen.

Ein allgemeiner paralleler Simulator kann nicht von der Prämisse ausgehen, dass Events nur zu bestimmten Zeitpunkten auftreten. Im beschriebenen Fall ist dies allerdings garantiert, da Signalflanken nur gleichzeitig mit Taktflanken auftreten können. Im Gegensatz zum allgemeinen Fall müssen daher nicht laufend Events zwischen den Simulatoren verschickt werden, sondern nur zu jeder Taktflanke. Die Übertragung kann damit zyklusorientiert erfolgen. Nur zu jeder Taktflanke ist somit ein Signalabgleich erforderlich. Danach kann jeder Simulator den ganzen folgenden Taktzyklus eigenständig abarbeiten. Abhängig vom Entwurf reicht eventuell sogar ein Signalabgleich zu jeder positiven Taktflake. Dies würde den Verkehr über die LCs sofort halbieren. Da nur wenige Signale über die Verbindung laufen, ist zu erwarten, dass der Verkehr zwischen den Partitionen so gering gehalten werden kann, dass die Simulation dadurch nicht gebremst wird. Das beschriebene Prinzip setzt also voraus, dass alle Simulatoren taktsynchron laufen. Dies muss durch geeignete Synchronisationsmechanismen garantiert werden. Weiters bedingt dieser Synchronlauf, dass der Fortschritt der Gesamtsimulation durch das langsamste Teilsystem bestimmt wird.

Es ist also zu erwarten, dass das Verhältnis der Komplexität der einzelnen Boards den Performancezuwachs recht gut wiederspiegelt. Für eine ausgeglichene Partitionierung würde sich somit im Idealfall ein linearer Zusammenhang zwischen Geschwindigkeitszuwachs und der Anzahl der Partitionen erwarten lassen. Ein zentrales Thema der parallelen Simulation ist die Wahl der Partitionierung *load balancing.* Im vorliegenden Fall ist die Partitionierung durch die Architektur schon vorgegeben. Load balancing gewinnt aber eine neue Bedeutung. Da die Geschwindigkeit immer durch das langsamste Glied bestimmt wird, liegt es nahe, falls unterschiedlich starke Workstations zur Verfügung stehen, die zeitkritischste Partition auf der schnellsten Maschine zu simuliern. Zusätzlich muss noch mit Geschwindigkeitseinbußen bedingt durch den Netzwerkverkehr gerechnet werden. Weiters ist zu vermuten, dass der Netzwerkverkehr mit zunehmender Komplexität der Einzelkomponenten abnimmt und dessen Einfluß auf die Gesamtperformance damit geringer wird.

Diese hier aufgestellten Thesen gilt es in der Arbeit durch konkrete Ergebnisse zu untermauern.

Um eine konkrete Lösung zu implementieren, müssen sowohl ein bestimmter Kommunikationsmechanismus, als auch ein geeigneter Hardwaresimulator ausgewählt werden. Für die nötigen Kommunikationkanäle bieten sich *TCP/IP Socketverbindungen* an. Um Sockets an den Simulator anzubinden, muss dieser über eine Programmierschnittstelle verfügen. Durch Verwendung eines standardisiertes Interfaces würde Herstellerunabhängigkeit garantiert werden. Für Verilog existiert zwar ein standardisiertes Interface (Verilog-PLI), doch ein entsprechendes Interface für VHDL (VHPI) befindet sich zur Zeit noch in Entwicklung. Daher soll ein herstellerspezifisches Interface verwendet werden und bei der Entwicklung darauf Rücksicht genommen werden, dass später leicht auf ein anderes Interface umgestiegen werden kann.

Ein Simulator der diese Anforderungen erfüllt ist *Modelsim 5.4* von Mentor Graphics. Dieser stellt ein *Foreign Language Interface (FLI-Interface)* bereit, mit dessen Hilfe *C Code* in die Simulation integriert werden kann. Zu diesem Zweck stellt das Interface Funktionen zur Signalbehandlung bereit. Der erstellte *C Code* kann mit einem normalen *C Compiler*, wie dem *GNU C++ Compiler (gcc)* compiliert werden. Der Compiler erzeugt eine Objektdatei, die der Simulator dann laden kann. Über das *FOREIGN* Attribut von *VHDL* kann der Bezug zwischen einer *VHDL ARCHITECTURE* und dem betreffenden *C* Modell hergestellt werden.

Mit Hilfe des *FLI-Interfaces* sollen nun *C* Modelle entwickelt werden, die über *TCP/IP Sockets* miteinander kommunizieren können. Die Synchronisation kann auf diese Art über *blockierende Sockets* implementiert werden. Diese angestrebte Lösung zeigt Bild 2.3.

Die Konfiguration des Gesamtsystems soll anfangs auf einfache Weise direkt im Quellcode erfolgen. Dies bedingt allerdings, dass bei jeder neuen Konfiguration neu compiliert werden muss.

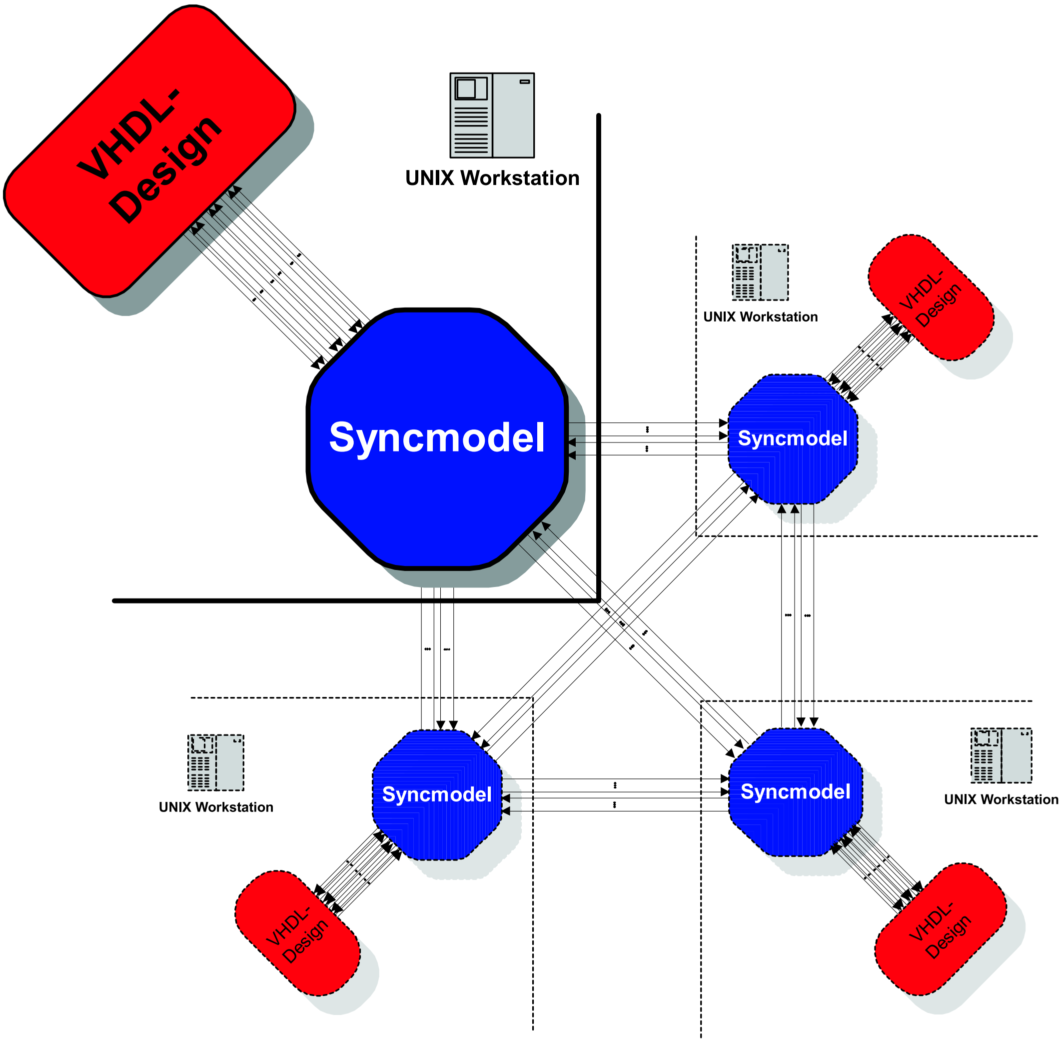


Abbildung 2.3: Die angestrebte Lösung

Kapitel 3

Rechnergestützte Simulation

Aus der modernen Wissenschaft und Technik ist Simulation heute nicht mehr wegzudenken, da sie oft das einzige praktikable Mittel zur Untersuchung komplexer Systeme darstellt. Die Gründe dafür sind vielfältig und abhängig vom System. Ein Grund kann zum Beispiel der zu lange Beobachtungszeitraum sein. So kann ein atomarer Zerfallsprozess, der in der Realität tausende Jahre dauern würde, mit einem Rechner in wenigen Augenblicken simuliert werden. Dass auch die Auswirkungen einer Atombombenexplosion besser nur per Simulation untersucht werden, ist wünschenswert. Allerdings ist immer zu bedenken, dass jede Simulation nur auf einem, oft sehr stark vereinfachten, Modell der Realität basieren kann und so die Ergebnisse in der Regel nur als Näherungen betrachtet werden können.

3.1 Grundprinzipen der rechnergestützten Simulation [19]

Bei der rechnergestützten Simulation wird ein Modell eines realen physikalischen Systems in ein Computerprogramm abgebildet, welches dann auf einem Rechner ausgeführt wird. Je aufwendiger die zugrundeliegenden Modelle sind, desto rechnerintensiver werden auch die Simulationen. Für die vielfältigen Anforderungen denen Simulationen gerecht werden müssen, wurden verschiedene Methoden der rechnergestützten Simulation entwickelt. Diese kann man je nach Art der zugrundeliegenden Modelle in zwei Klassen einordnen (s. Abbildung 3.1): *kontinuierliche* und *diskrete Methoden*.

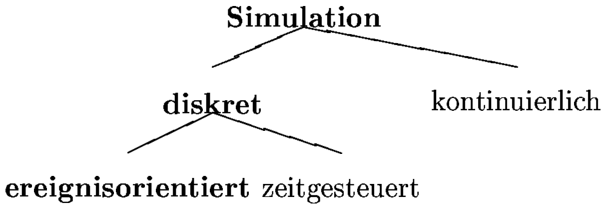


Abbildung 3.1: Methoden der rechnergestützten Simulation

Diese beiden Arten unterscheiden sich im Wesentlichen durch die Modellierung der Simulationszeit.

Bei der *kontinuierlichen* Simulation wird ein System durch Differentialgleichungen mit der Zeit als freie Variable abgebildet. Diese Gleichungen werden dann mit numerischen Methoden gelöst. Da für die Variable Zeit somit jeder beliebige Wert eingesetzt werden kann, kann der Systemzustand für jeden beliebigen Zeitpunkt berechnet werden. Im Interesse der folgenden Betrachtungen steht allerdings die zweite Klasse, nämlich die *diskrete* Simulation.

Im Falle der *diskreten* Simulation schreitet die Zeit nur um diskrete Zeitschritte fort. Der Systemzustand kann also nur für bestimmte Zeitpunkte beschrieben werden. Würde man alle möglichen Zeitpunkte graphisch darstellen, würde sich im Falle der kontinuierlichen Simulation eine Gerade ergeben, während man bei der diskreten Simulation immer Punkte erkennen würde. Bei der diskreten Simulation unterscheidet man zwischen *zeitgesteuerter* und *ereignisgesteuerter* Simulation.

3.1.1 Zeitgesteuerte Simulation

Bei der *zeitgesteuerten* Simulation schreitet die Simulationszeit (virtual time, VT) bei jedem Simulationsschritt um ein festes Zeitintervall Δt fort. Alle Zustandsänderungen, die im realen System innerhalb dieses Zeitintervalls erfolgen, werden in der zeitgesteuerten Simulation erst am Ende des Intervalls wirksam. Die geeignete Größe des Zeitintervalls ist abhängig vom simulierten System. Für die Hardware-Simulation von integrierten Schaltungen werden typische Zeitintervalle im Bereich von Nanosekunden gewählt, während für Wettersimulationen Werte von einigen Minuten typisch sind. Die Wahl von Δt bestimmt die Zeitauflösung und somit die Genauigkeit der Simulation. Je kleiner Δt gewählt wird, desto genauer werden die Simulationsergebnisse und umso rechenintensiver wird die Simulation. Die Wahl eines großen Zeitintervalls Δt beschleunigt zwar die Simulation, doch können dabei eventuell wichtige Details nicht mehr erfasst werden.

3.1.2 Diskrete ereignisgesteuerte Simulation

Bei der *diskreten ereignisgesteuerten Simulation (discrete event simulation, DES)* erfolgt die Abbildung des realen Systems durch die Modellierung von *Objekten*. Diese Objekte können verschiedene *Zustände (States)* einnehmen. Ein Zustand wird dabei durch eine Menge von *Zustandsvariablen (state variables)* $S = (S_1, S_2, ...S_n)$ beschrieben. Aktionen, die in der Realität Zustandsänderungen verursachen, werden durch *Ereignisse (Events)* dargestellt. Jedes *Ereignis* besitzt einen *Zeitstempel (timestamp)*, welcher angibt wann das betreffende Ereignis eintreten soll.

Weiters ist jedem Ereignis eine Ereignisroutine zugeordnet. Alle Ereignisse werden in einer *Ereignisliste (event list, EVL)* gespeichert. Die Speicherung erfolgt in aufsteigender Reihenfolge der Zeitstempel. Tritt ein Ereignis ein, wird die betreffende *Ereignisroutine* ausgeführt. Eine Ereignisroutine kann

- Zustandsvariablen eines Objekts verändern,
- neue Ereignisse in die EVL einfügen, oder Ereignisse aus der EVL entfernen.

Egal wie viel Rechenzeit zur Ausführung einer Ereignisroutine real verbraucht wird, geschieht sie konzeptuell verzögerungslos. Die interne Simulationszeit wird vor der Ausführung der Ereignisroutine vorgestellt, steht während der Ausführung still und springt nach vollzogener Zustandsänderung auf den Zeitpunkt des nächsten Ereignisses.

Ein Zustandsübergang eines Objekts kann seinerseits wieder neue Ereignisse auslösen, die dann neue Zustandsänderungen bewirken. Die Ablaufkontrolle der Simulation erfolgt nun nach dem in Abbildung 3.2 dargestellten Schema.

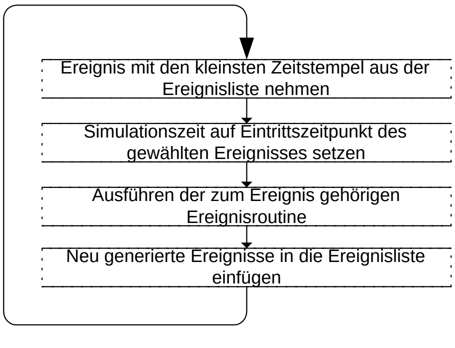


Abbildung 3.2: Prinzip der ereignisgesteuerten Simulation

Dabei werden die, in der Ereignisliste nach ihren Eintrittszeitpunkten geordneten, Ereignisse sequentiell abgearbeitet (*next event approach*). Bei dem dargestellten

Algorithmus wird die Simulationszeit also nicht wie bei der zeitgesteuerten Simulation um konstante Zeitintervalle erhöht, sondern um die Zeitdifferenz zwischen dem aktuellen und dem nächsten Ereignis in der Ereignisliste. Daraus folgt auch, dass Intervalle in denen keine Zustandsänderungen eintreten einfach übersprungen werden, und überhaupt keine Rechenzeit benötigen.

3.2 Logik Simulation

Unter Logik Simulation versteht man die Simulation von digitalen, logischen Schaltungen. Dabei kann es sich sowohl um einfache Entwürfe mit einigen wenigen Gattern, als auch um komplette ASICs mit Millionen von Gattern, oder sogar um ganze Boards die ihrerseits mit einer Vielzahl von ASICSs bestück sind, handeln. Vordergründiges Ziel der Logik Simulation in der ASIC-Entwicklung ist die Vermeidung von Design-Fehlern, bzw. deren Früherkennung. Gerade in der Chip-Entwicklung ist es wichtig *"first time right"* Systeme zu erzeugen, da die Fertigung von fehlerhaften Chips und ein damit verbundenes Redesign extrem hohe Kosten verursacht. In der ASIC-Entwicklung verwendet man heute fast ausschließlich Hardware-Beschreibungssprachen, wie VHDL oder auch Verilog. Eine derartige Hardware-Beschreibung kann direkt als Ausgangspunkt für die Simulation des Entwurfes verwendet werden.

Für gewöhnlich wird heute ein neuer ASIC nie von Grund auf neu entworfen, sondern es werden bereits vorhandene Blöcke, vom einfachen ODER-Gatter bis zum kompletten Prozessor, eingebunden. Diese Modelle stammen meist aus vorangegangenen Entwürfen, oder aus vom Chip-Hersteller bereitgestellten Bibliotheken. Selbst wenn man davon ausgeht, dass die verwendeten Elemente keine Fehler enthalten, muss deren Zusammenspiel auf einer höheren Abstraktionsebene getestet werden.

Logiksimulation kann auf mehreren Abstraktionsebenen betrachtet werden, angefangen von der Architekturebene bis hinunter zur Gatterebene. Auf hoher Abstraktionsebene sind Schaltungen leichter zu programmieren als auf Gatterebene. Sie liefern aber weniger genaue Resultate, da nicht mehr jedes Detail simuliert werden kann. Niedrige Abstraktionsebenen bedeuten eine hohe Auflösung und bieten damit große Genauigkeit. Der Preis dafür ist in der Regel allerdings eine längere Simulationsdauer. In Bild 3.3 sind mögliche Abstraktionsebenen mit typischen Simulationsobjekten für den Entwurf von logischen Schaltungen dargestellt.

Abbildung 3.3: Abstraktionsebenen für den Entwurf logischer Schaltungen

3.2.1 Ereignisgesteuerte Logiksimulation

Definition 3.2.1 *Ein* Ereignis *entspricht in der Logiksimulation einem projektierten Wertewechsel eines Signals, d.h. bei der Ausführung eines Ereignisses wird dem Signal ein neuer Wert zugewiesen.*

In der Ereignisroutine wird der Wertewechsel durchgeführt[1]. Wenn sich ein Eingangssignal eines Gatters ändert, so muss dieses Gatter anschließend ausgewertet werden. Dies bedeutet im Speziellen, die Schaltfunktion des Gatters mit den aktuellen Eingangssignalen zu berechnen. Das Ergebnis der Schaltfunktion muss den Ausgangssignalen des Gatters zugewiesen werden. Dazu werden für diese Signale neue Ereignisse generiert.

In der Logiksimulation findet man eigentlich nur zwei Aktionen: Nämlich

- Ereignisse auswerten und
- Schaltfunktionen von Gattern berechnen.

---

[1] In der Logiksimulation existieren keine unterschiedlichen Ereignistypen und daher gibt es auch nur genau eine Ereignisroutine

Diese beiden Aktionen werden nacheinander ausgeführt und bilden einen *Simulationsschritt*. In jedem Simulationsschritt werden zunächst alle Ereignisse mit dem kleinsten Eintrittszeitpunkt (=Zeitstempel) ausgewertet. Im zweiten Schritt werden dann alle Schaltfunktionen der Gatter, deren Eingangssignale sich geändert haben, ausgeführt. Jede Änderung eines Ausgangssignals bedingt wieder ein neues Ereignis. Auch hier werden alle Ereignisse genau nach dem bereits in Abbildung 3.2 dargestelltem Algorithmus abgearbeitet.

3.3 Parallele ereignisorientierte Simulation

3.3.1 Möglichkeiten zur Parallelisierung

Im Gegensatz zur Simulation eines realen Systems unter Benutzung eines sequentiellen Computerprogramms, laufen in der Wirklichkeit in einem System oft mehrere Vorgänge gleichzeitig ab. Diese Tatsache lässt sich auch in der Simulation ausnutzen, indem man das Gesamtsystem auf mehrere Rechner aufteilt, die dann parallel arbeiten können. Die Parallelisierung der Simulationsaufgabe kann auf unterschiedlichen Ebenen in Angriff genommen werden. Ferscha[6] unterscheidet dabei folgende Möglichkeiten:

Anwendungs-Ebene:
Dies ist die augenscheinlichste Methode, eine Vielzahl gleichartiger Simulationen mit unterschiedlichen Parametern durchzuführen. Dabei wird ein und dasselbe Simulationsmodell gleichzeitig mehreren Prozessoren zugeführt. Da die Simulationen vollkommen unabhängig sind, ist die Geschwindigkeitssteigerung sehr hoch. Der entscheidende Nachteil dieser Methode liegt allerdings darin, dass für jeden Prozessor sehr viel Arbeitsspeicher benötigt wird, um die gesamte Simulation bewältigen zu können.

Unterprogramm-Ebene
Manchmal kann es notwendig sein, dass mehrere gleichartige Simulationen unbedingt hintereinander ausgeführt werden müssen. Dies ist immer dann der Fall, wenn das Ergebnis eines Simulationslaufes die Eingangsparameter für die nächste Simulation liefert. In diesem Falle kann es sinnvoll sein, einzelne, voneinander unabhängige, Unterprogramme parallel auszuführen. So können zum Beispiel Funktionen zur Erzeugung von Zufallszahlen, statistischen Auswertung, Verarbeitung von Ereignissen und zur Zustandsaktualisierung gleichzeitig arbeiten.

Komponenten-Ebene

Keine der beiden bisher genannten Möglichkeiten nutzt die eventuell im realen System vorhandene Parallelität aus. Zu diesem Zweck muss das Gesamtsystem in mehrere Untermodelle aufgeteilt werden, die dann auf verschiedenen Prozessoren simuliert werden können. Diese Untermodelle sollen möglichst die einzelnen Komponenten des realen Gesamtsystems abbilden. Betrachtet man als Beispiel die Bearbeitung von Dokumenten in einem größeren Betrieb, so könnten darin die einzelnen Sachbearbeiter durch eigene Modelle abgebildet werden. Die Verarbeitung eines Dokuments kann damit auf einem eigenen Prozessor simuliert werden. Die Weitergabe an einen weiteren Bearbeiter kann dann durch Versenden einer Nachricht an einen anderen Prozessor modelliert werden.

Ereignis-Ebene (zentrale Ereignisliste)

Speziell für die diskrete Event Simulation existieren zwei weitere Möglichkeiten zur Parallelisierung. Eine davon ist die Implementierung einer zentralen Ereignisliste, die von einem eigenen Steuerprozess verwaltet wird. Alle Prozesse stellen die von ihnen erzeugten Events in diese zentrale Ereignisliste. Der Steuerprozess verteilt die Ereignisse dann in geeigneter Weise an die Simulationsprozesse zur Abarbeitung. Um einerseits Kausalität zu garantieren und andererseits aber möglichst effektiv arbeiten zu können, muss der Steuerprozess die Prozessabhängigkeiten kennen. Zu diesem Zweck muss zu Beginn das Gesamtsystem analysiert werden, und die Abhängigkeiten müssen in geeigneter Weise erfasst werden. Nur auf diese Weise kann der Steuerprozess sicherstellen, dass Events, die voneinander abhängig sind, nicht gleichzeitig den betreffenden Prozessen zugewiesen werden.

Ereignis-Ebene (dezentrale Ereignisliste)

Bei dieser Methode der Parallelisierung fehlt die globale EVL und die Prozesse verfügen somit nur über ihre eigenen lokalen Ereignislisten. Bei dieser Lösung muss zusätzlich ein Mechanismus zur lokalen Synchronisation von abhängigen Prozessen implementiert werden. Dies ist allerdings mit einem erhöhten Kommunikationsaufkommen zwischen den Prozessen verbunden.

3.3.2 Prinzip der verteilten Simulation

Alle Strategien für verteilte Simulation verfolgen das Ziel, die Simulationsaufgabe auf mehrere logische Prozesse (LP_i) aufzuteilen, welche gleichzeitig ausgeführt werden können. Diese Prozesse tauschen untereinander laufend Informationen aus. Abbildung 3.4 zeigt solch ein aus mehreren logischen Prozessen bestehendes System.

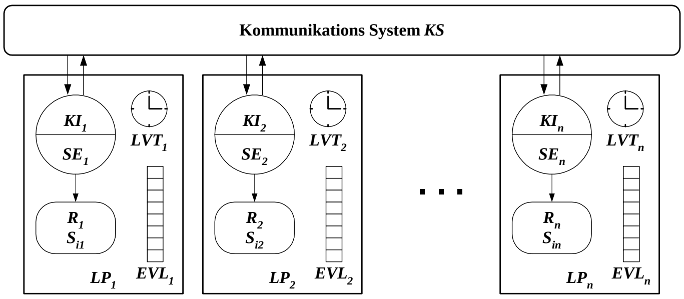


Abbildung 3.4: Zusammenspiel mehrerer LPs

Dabei sind mehrere *logische Prozesse* LP_i über ein *Kommunkationssystem* (KS) miteinander verbunden. Jeder LP bearbeitet nur einen Teil des Gesamtsystems, bezeichnet als *Region* (R_i). In jeder Region ist nur eine Untermenge der Zustandsvariablen des Gesamtsystems enthalten $(S_{i_1,i_2,..i_nq} \subset S)$. Somit hat jeder LP_i nur auf eine Teilmenge der Zustandsvariablen des Gesamtsystems direkten Zugriff. Die eigentliche Abarbeitung der Events erfolgt durch einen *Simulation Engine* (SE). Der SE kann über ein *Kommunikations Interface* (KI_i) auf das Kommunikationssystem zugreifen. Jeder SE_i muss zwei Arten von Events verarbeiten:

- **interne Ereignisse** beeinflussen nur Zustandsvariablen innerhalb der lokalen Region $(S_i \subset S_{i_n})$.
- **externe Ereignisse** beeinflussen Zustandsvariablen in fremden Regionen$(S_i \not\subset S_{i_n})$. Dies kann nur erfolgen indem über das Kommunikationssystem eine Nachricht an den LP geschickt wird, der die betreffende Region verwaltet.

Die *lokale Simulationszeit* LVT_i *(local virtual time)* wird von jedem LP_i mit Hilfe eines Zeitzählers verwaltet. Jedes Event, egal ob intern oder extern, erhält den aktuellen Wert dieses Zählers als Zeitstempel.

3.3.3 Das Grundproblem der verteilten Simulation

Ein verteiltes Simulationsmodell besteht aus einer Vielzahl einzelner Simulationsprozesse, von denen aber keiner die globale Sicht auf das Gesamtsystem besitzt. In einem derartigen System existiert somit keine globale Zeit, auf die die einzelnen Prozesse Bezug nehmen könnten. Da innerhalb der einzelnen Prozesse die Simulationszeit unterschiedlich schnell fortschreitet, unterscheiden sich deren lokale Simulationszeiten (LVT_i).

Definition 3.3.1 *Zwei Simulationsprozesse* P_1 *und* P_2 *sind voneinander abhängig, wenn es mindestens ein Ereignis* E *gibt, das in* P_1 *generiert wurde und* P_2 *beeinflusst.*

In der Regel sind die einzelnen Prozesse allerdings nicht voneinander unabhängig, was bedingt, dass deren Simulationsforschritt explizit aufeinander abgestimmt werden muss. Nur auf diese Weise kann die kausale Abhängigkeit der Ereignisse berücksichtigt werden. Dies ist eine unbedingte Voraussetzung um deterministische und korrekte Simulationsergebnisse zu erhalten. Dieses Grundproblem der verteilten Simulation kann durch folgendes einfaches Beispiel verdeutlicht werden:

Angenommen ein Prozess P_1 erzeugt ein Ereignis E, welches einen anderen Prozess P_2 beeinflusst. Prozess P_1 muss nun eine Nachricht an P_2 versenden, um ihm dieses Ereignis zu übergeben. Diese Nachricht enthält dabei den Eintrittszeitpunkt des Ereignisses $T(E)$. Dieser entspricht der lokalen Simulationszeit $LVT(P_1)$ des Prozesses P_1 beim Eintritt von E. Da die lokalen Simulationszeiten der einzelnen Prozesse voneinander unabhängig sind (dh. $LVT(P_1) \neq LVT(P_2)$), könnte der Fall eintreten, dass P_2 beim Erhalt der Nachricht mit der Simulation schon zu weit fortgeschritten ist, sodass gilt $T(E) < LVT(P_2)$. Das Ereignis E hätte von P_2 also schon in der Vergangenheit ausgewertet werden müssen, was zu einer Verletzung der Kausalität führen kann.

Entscheidend für garantierte Kausalität ist somit die Reihenfolge der Abarbeitung der Ereignisse. Eine Verletzung der Reihenfolge führt allerdings nicht notwendig zu einem Kausalitätsfehler. Dies kann nur der Fall sein wenn durch die betreffenden Ereignisse auch auf gemeinsame Zustandsvariablen zugegriffen wird.

Definition 3.3.2 (nach Fujimoto[7]) *In einem System von logischen Prozessen, welche durch den Austausch von mit Zeitstempeln versehenen Ereignissen kommunizieren, ist die Kausalität genau dann garantiert, wenn die Prozesse die Ereignisse in aufsteigender Reihenfolge der Zeitstempel, verarbeiten.*

Das Hauptproblem der verteilten Simulation besteht nun darin, Kausalität zu garantieren. Dazu werden geeignete Koordinationsstrategien bzw. Protokolle benötigt. Alle existierenden Protokolle lassen sich in zwei Klassen einteilen: *synchrone* und *asynchrone Methoden.*

3.3.4 Methoden der verteilten Simulation

3.3.4.1 synchrone parallele Simulation

In einer synchronen Simulationsumgebung arbeiten die einzelnen logischen Prozesse eng miteinander gekoppelt unter Verwendung einer gemeinsamen Simulationszeit.

Dabei liefert also ein zentraler Zeitgeber die Zeitbasis für die logischen Prozesse. Dieses Prinzip ist vergleichbar mit der sequentiellen zeitgesteuerten Simulation, die bereits in Abschnitt 3.1.1 beschrieben wurde, mit dem Unterschied, dass die einzelnen Objekte, nun durch logische Prozesse repräsentiert werden, welche parallel auf mehreren Prozessoren ausgeführt werden. Abbildung 3.5 soll das Prinzip der synchronen parallelen Simulation verdeutlichen. Es ist zu erkennen, dass die einzelnen *LPs* jeweils einen Simulationszyklus parallel abarbeiten. Am Ende des Zyklus werden die generierten Events an die betreffenden *LPs* weitergeleitet. Jeder *LP* der auf diese Weise ein Event für den aktuellen Zyklus erhalten hat, muss diesen neu berechnen, während alle anderen warten müssen. Erst wenn alle *LPs* fertig evaluiert wurden, kann der zentrale Zeitgeber die Simulationszeit fortschreiten lassen, wonach alle *LPs* mit der Berechnung des nächsten Zyklus beginnen können. Auf diese Weise wird Synchronlauf der *LPs* garantiert. Ein Nachteil dieser Methode ist, dass die Geschwindigkeit des Gesamtsystems immer durch das langsamste Teilsystem bestimmt wird. Besonders negativ wirkt sich dies in "mixed-level" Simulationen aus, wo ein Model eventuell nur aus wenigen Gattern besteht, während ein anderes einen ganzen Prozessorkern enthalten könnte. Eine synchrone Simulationsumgebung ist relativ einfach zu implementieren, da prinzipiell keine Kausalitätsfehler auftreten können.

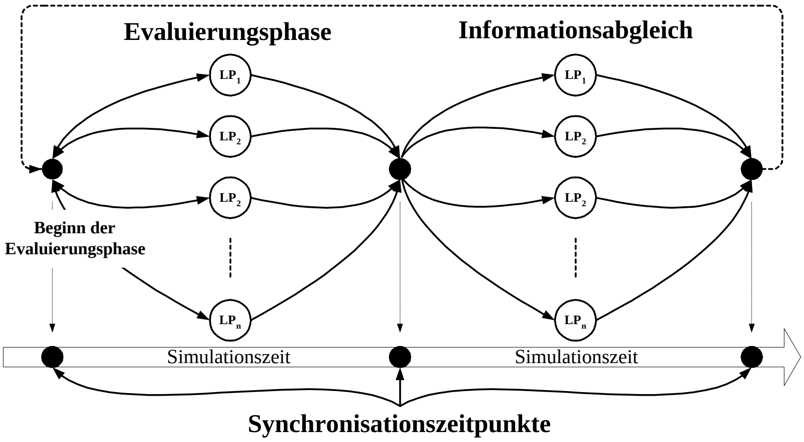


Abbildung 3.5: Synchrone parallele Simulation

3.3.4.2 asynchrone parallele Simulation

Einen komplett anderen Ansatz stellen asynchrone Methoden dar. Entscheidender Unterschied ist das Fehlen des zentralen Zeitgebers, und somit auch des globalen Synchronisationsmechanismus. Jeder *LP* verfügt nun, wie auch schon in Bild 3.4

gezeigt wurde, über ihre lokale Simulationszeit (LVT). Das Entscheidende bei asynchroner paralleler Simulation ist nun, dass die LVTs völlig unabhängig voneinander sind. (s. Bild 3.6) Jeder LP verarbeitet pro Simulationszyklus die in seiner lokalen EVL für diesen Simulationszeitpunkt anstehenden Ereignisse und trägt am Ende die von ihm generierten internen Ereignisse (vgl. Abschnitt 3.3.2) wieder in diese ein. Alle externen Ereignisse werden über LCs in die $EVLs$ der entsprechenden LPs eingefügt. Im Gegensatz zum synchronen Ansatz ist hier die Kausalität keineswegs garantiert, da es immer möglich ist, dass über einen LC ein Ereignis geliefert wird, dessen Zeitstempel kleiner als die aktuelle LVT ist. Da dieses Ereignis schon in der Vergangenheit verabeitet hätte werden müssen, befindet sich der betreffende LP in einem ungültigen Zustand.

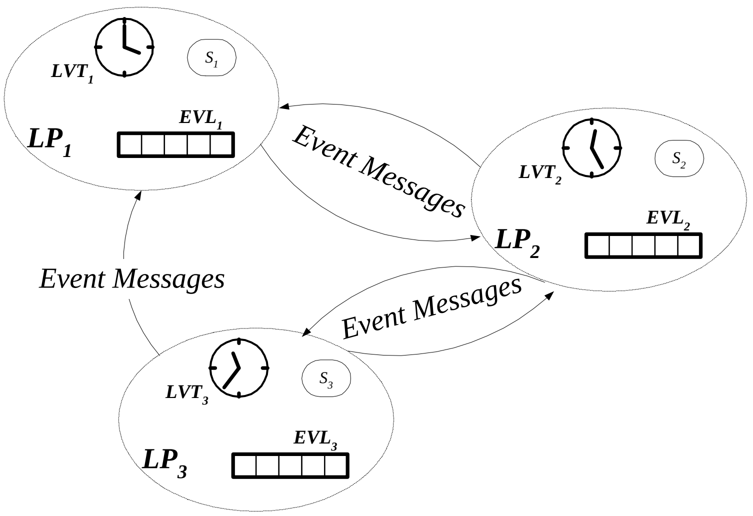


Abbildung 3.6: Asynchrone parallele Simulation

Um derartige Kausalitätsverletzungen zu verhindern, müssen geeignete Mechanismen entwickelt werden.

In der Literatur unterscheidet man zwischen konservativen und optimistischen Verfahren.

3.3.4.2.1 konservative Methoden (Chandy Misra Bryan Protokolle) Die konservativen Lösungsansätze gehen auf die Arbeiten von Chandy und Misra [3] [4], bzw. Bryan[2] zurück. In der Literatur spricht man daher häufig vom *CMB-Algorithmus* (**C**handy **M**isra **B**ryan). Kernelement dieses Algorithmus ist die Garantie, dass ein LP ein lokales Ereignis nur dann verarbeiten darf, wenn sichergestellt ist, dass kein Ereignis mit kleinerem Zeitstempel eintreffen wird. Dies wird auf folgende Weise garantiert:

Es wird vorausgesetzt, dass die lokalen Simulationszeiten LVT der einzelnen LPs unabhängig voneinander sind. Zu Beginn bauen die LPs die nötigen Kommunikationskanäle LCs auf, über welche dann die mit Zeitstempeln versehenen Ereignisse verschickt werden können. Jeder LC verfügt über eine Eingangswarteschlange InQ sowie eine Ausgangswarteschlange $OutQ$ für die eingehenden bzw. zu versendenden Ereignisse. Die LP dürfen Ereignisse nur in aufsteigender Reihenfolge in die $OutQ$ stellen. Für jeden LC wird auch die sogenannte Kanalzeit T_{ch} festgehalten. Dieser Zähler enthält immer den Wert des Zeitstempels des ersten Ereignisses in der InQ, bzw. falls diese leer ist, den Zeitstempel des letzten Ereignisses in dieser Warteschlange. Um Kausalität zu garantieren, müssen die auf allen LCs eingehenden Ereignisse unbedingt in aufsteigender Reihenfolge verarbeitet werden. Jeder LP wählt nun jenes Ereignis zur Verarbeitung aus, das sich an erster Stelle der InQ des LC mit der kleinsten Kanalzeit ${T_{ch}}^{min}$ befindet. Danach wird die LVT auf den Wert von ${T_{ch}}^{min}$ erhöht.

Der entscheidende Punkt ist nun folgender: Falls die InQ des LC mit der kleinsten Kanalzeit leer ist, muss der betreffende LP in einen blockierenden Wartezustand versetzt werden, anstatt einfach den LC mit der nächsthöheren T_{ch} auszuwählen. Dies ist notwendig, da es keine Garantie dafür gibt, dass der LP am anderen Ende des LC in der Zukunft nicht doch noch ein Event mit kleinerem Zeitstempel liefert.

Dieser erforderliche Wartezustand bringt leider die Gefahr eines Deadlocks mit sich. Eine Deadlock-Situation tritt ein, wenn sich mehrere LPs gegenseitig blockieren. Eine derartige Konstellation zeigt Bild 3.7.

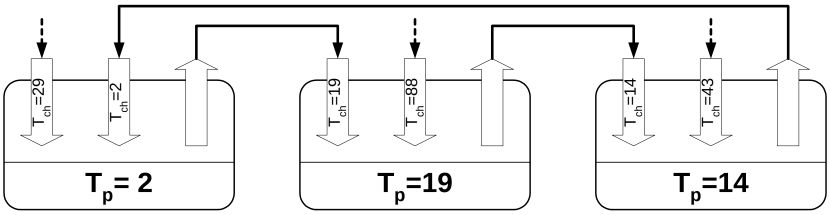


Abbildung 3.7: Beispiel für eine Deadlock-Situation

Um eine derartige Situation verhindern zu können gibt es mehrere Ansätze, von denen hier nur der einfachste Fall exemplarisch behandelt wird, nämlich die Einführung sogenannter *Null-Nachrichten*. Eine Darstellung weiterer Vermeidungsstrategien für Deadlocks kann bei Ferscha[6] nachgelesen werden.

Deadlock-Vermeidung durch Einführung von Null-Nachrichten:
Deadlocks können vermieden werden indem das CMB-Protokoll durch sogenannte Null-Nachrichten erweitert wird. Diese bestehen aus einem Ereignis, welchem keine Ereignisroutine zugeordnet ist, dem Null-Ereignis, und einem Zeitstempel. Null-Nachrichten werden immer dann versendet, wenn ein *LP* nach Verarbeitung eines Ereignisses keine externen Ereignisse generiert hat. Um den anderen *LPs* zu signalisieren, dass er sicher kein Ereignis mit geringerem Zeitstempel in die Warteschlangen stellen wird, verschickt der betreffende *LP* Null-Nachrichten, wodurch die Kanalzeiten der blockierenden *LCs* erhöht werden. Dadurch werden auch die wartenden *LPs* aus ihrem Wartezustand entlassen. Zu beachten ist allerdings auch, dass durch die Einführung von Null-Nachrichten ein beachtlicher Kommunikations-Overhead entstehen kann.

3.3.4.2.2 optimistische Methoden (Time Warp Protokolle)

Bei einem optimistischen Verfahren werden immer die Ereignisse mit den kleinsten Zeitstempeln in der Ereignisliste ausgeführt ohne die Kanalzeiten zu den Nachbarprozessen zu berücksichtigen. Die Ereignisse müssen also nicht im Sinne der konservativen Protokolle sicher sein. Erhält ein Prozess über einen *LC* ein Ereignis *E* mit einem Zeitstempel der kleiner als seine lokale Simulationszeit *LVT* ist (oft bezeichnet als *Stranggler*), muss die nun fälschlicherweise ausgeführte Simulation rückgängig gemacht werden. Zu diesem Zweck wird ein sogenannter *(*Rollback) ausgeführt. Dabei muss die Simulation wieder in einen Zustand der zeitlich vor dem Zeitstempel des Strangglers liegt zurückversetzt werden. Dazu müssen alle falsch abgearbeiteten Ereignisse zurück in die lokale *EVL* gestellt werden, und auch die Zustandsvariablen der aktuellen Region müssen zurückgesetzt werden. Zusätzlich müssen auch für alle versendeten externen Ereignisse sogenannte *Antinachrichten* an die entsprechenden Nachbarprozesse versendet werden. Bei Empfang einer Antinachricht wird das entsprechende Event aus der *EVL* entfernt. Wenn das Ereignis allerdings schon verarbeitet wurde, muss auch dieser *LP* einen Rollback durchführen.

Ein Rollback ist in der Regel sehr zeitaufwendig. Weiters wird sehr viel Speicherplatz benötigt, da die einzelnen *LPs* laufend ihre Zustände abspeichern müssen um diese im Falle eines Rollbacks wieder laden zu können. So müssen in regelmäßigen Abständen alle Zustandsvariablen, alle Ereignisse und alle versendeten Ereignisnachrichten gespeichert werden. Um Speicherplatz auch wieder freigeben zu können, muss laufend die sogenannte *global virtual time (GVT)* berechnet werden. Diese enspricht dem kleinsten Wert der *LVT* aller *LPs* im System. Da ein *LP* nie ein Ereignis mit einem Zeitstempel, der kleiner als seine *LVT* ist, erzeugen kann, kann davon ausgegangen werden, dass keine Strangglers mit einem kleineren Zeitstempel eintreffen werden. Es können somit alle Zustandsabbilder, die zu einem Zeitpunkt vor der aktuellen *GVT* gehören, verworfen werden.

Es gibt verschiedene optimistische Protokolle, die sich meist in der Art des Rollbacks unterscheiden. Das bekannteste optimistische Protokoll ist das *Time Warp Protokoll*, welches in [12] beschrieben wird. Eine Zusammenfassung weiterer auf dem Time Warp Protokoll basierender Protokolle, wie *Lazy Cancellation*, *Lazy Reevaluation* und *Optimistic Time Windows* finden sich bei Fujimoto[8].

3.3.5 Der Unterschied zwischen paralleler und verteilter Simulation

In der Literatur werden die beiden Begriffe *parallele Simulation* und *verteilte Simulation* weitgehend synonym verwendet. Nach Ferscha [6] erfolgt allerdings eine Unterscheidung zwischen paralleler und verteilter Simulation anhand der zugrundeliegenden Rechnerarchitektur. In einer SIMD[2]-Architektur führen mehrere Prozessoren gleiche Befehle mit unterschiedlichen Daten aus. Die auszuführenden Befehle werden dabei von einer zentralen Einheit an den Prozessor weitergegeben. Über diese zentrale Einheit erfolgt somit die Synchronisation der einzelnen Prozessoren. Wird dieses Prinzip der *globalen Synchronisation* ausgehend von einer SIMD-Architektur auf die Simulation angewendet, spricht man von *paralleler Simulation.*

Eine Alternative zur SIMD-Architektur stellt die MIMD[3]-Architektur dar. Dabei arbeiten die Prozessoren parallel aber asynchron. Zur Kommunikation werden zwischen den Prozessen Nachrichten versendet. Es erfolgt also keine explizite globale Synchronisation durch eine übergeordnete Einheit. Eine zeitliche Abfolge von einzelnen Prozessen kann nur durch *lokale Synchronisation* mit Hilfe der ausgetauschten Nachrichten erfolgen. Man spricht nun von *verteilter Simulation* wenn die Synchronisation von den einzelnen Prozessen selbst garantiert wird, und keine globale Steuereinheit vorhanden ist.

Nachdem nun auf diese mögliche Definition hingewiesen wurde, wird im Folgenden nicht mehr zwischen paralleler und verteilter Simulation unterschieden.

---

[2] **S**ngle **I**nstruction Stream**M**ultiple **D**ata Stream

[3] **M**ultiple **I**nstruction Stream **M**ultiple **D**ata Stream

Kapitel 4

Beschreibung der Zielhardware

In diesem Kapitel soll ein sehr grober Überblick über das *Elektronische Wählsystem Digital (EWSD)* von Siemens gegeben werden, um einen Eindruck der Komplexität der später zur Parallelisierung herangezogenen Designs vermitteln zu können.

4.1 Die Systemarchitektur des EWSD

Das *Elektronische Wählsystem Digital (EWSD)* von Siemens ist ein flexibles Vermittlungssystem für öffentliche Kommunikationsnetze[17]. Dank seines modularen Aufbaus und der Transparenz von Hardware und Software erlaubt es die Anpassung an verschiedene Netzumgebungen, sowie die Skalierung auf unterschiedliche Benutzergrößenordnungen. So lassen sich im EWSD alle Arten und Größen von Vermittlungsstellen realisieren. Neben der Vermittlung normaler Telefongespräche erlaubt das EWSD den Ausbau des Fernsprechnetzes zum diensteintegrierenden digitalen Nachrichtennetz *ISDN*[1]. Seine große Flexibilität erreicht das EWSD durch seinen modularen Aufbau und die konsequente Verwendung standardisierter Schnittstellen. Der Aufbau des *EWSD* ist in Bild 4.1 dargestellt.

Die wesentlichen Bestandteile einer Vermittlungsstelle bilden die digitale Teilnehmerleitungseinheit *(digital line unit DLU)* und die Anschlusseinheit *(line/trunk group LTG)).* Die *DLU* konzentriert den Teilnehmerverkehr, bestehend aus analogen Teilnehmerleitungen, *ISDN*-Basisanschlüssen und Primärleitungen. Jede *DLU* ist mit einem oder zwei *LTGs* verbunden, die von dieser mit Daten versorgt werden. Die Verbindung zu einem LTG erfolgt über ein oder zwei *primary digital cariers (PDCs)* mit einer Bandbreite von 2048 kbit/s. Die Schnittstelle zum Koppelnetz *switching network SN* bilden die Anschlussgruppen *LTG.* Über diese werden dem *SN*

---

[1] **I**ntegrated **S**ervices **D**igital **N**etwork (siehe [9])

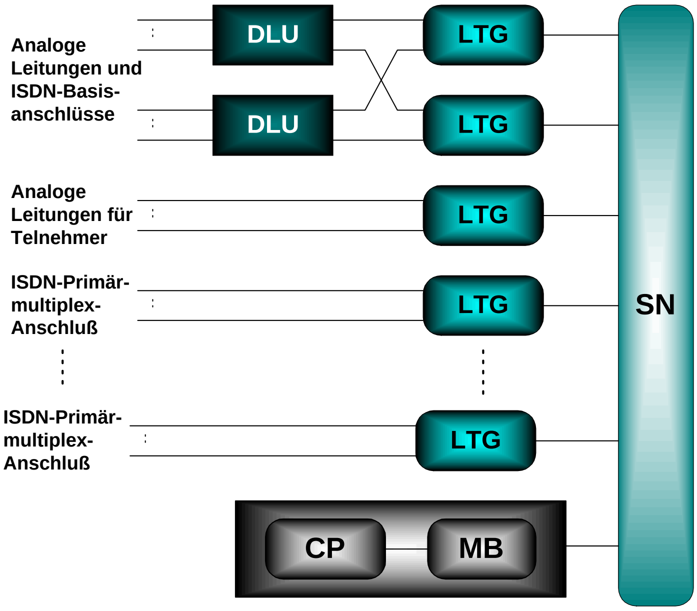


Abbildung 4.1: Aufbau des EWSD

Daten von verschiedenartigen Quellen (DLUs, direkte Teilnehmeranschlüsse, digitale Verbindungsleitungen oder ISDN-Primäranschlüsse) einheitlich zugeführt. Die *LTGs* konzentrieren diese Daten und senden sie mit 8 Mbit/s an das *SN*. Den Backbone des *EWSD* stellt somit das *SN* dar. Dieses routet die von den *LTGs* kommenden Datenströme zu anderen *LTGs*. Das *SN* ist gedoppelt ausgeführt (Ebene 0 und 1), wodurch Verbindungen auch bei Störungen aufrecht erhalten werden können. Die Steuerung des *SN* wird vom *coordination processor (CP)* übernommen. Dieser ist die zentrale Steuereinheit im *EWSD*. Vor Einführung des *Message Buffer D (MBD)* mussten fast alle Signalisierungsnachrichten über den *CP* laufen, wodurch dieser stark belastet wurde. Der MBD stellt eine Weiterentwicklung des *Message Buffer B* dar, der dazu diente, von den *LTGs* über das *SN* kommende Nachrichten zum *CP* zurück zu puffern. Der *MBD* ist dagegen in der Lage, viele Nachrichten am *CP* vorbeizuleiten. Signalisierungsnachrichten zwischen *LTGs* werden von ihm direkt untereinander vermittelt, ohne durch den *CP* zu gehen. Durch diese Maßnahme kann der *CP* entlastet werden.

4.2 Der Message Buffer Typ D

Der Aufbau des MBD ist in Abbildung 4.2 dargestellt. Der MBD besteht aus eigenständigen Baugruppen, die über einen Bus, genannt IBUS, Daten austauschen.

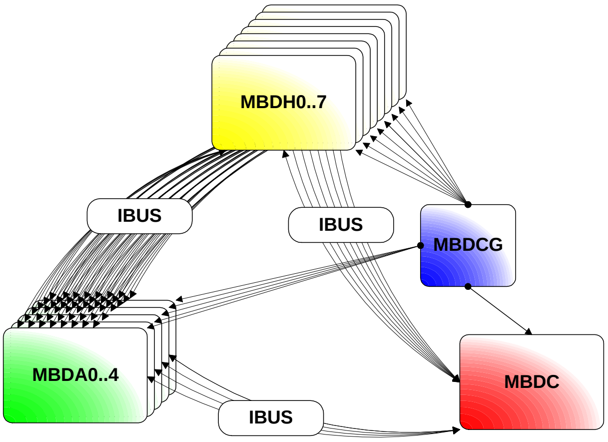


Abbildung 4.2: Aufbau des MBD

Alle benötigten Baugruppen sind zu einer MBD-Side (MBD0 / MBD1) zusammengefasst. Aus Redundanzgründen sind zwei solcher MBD-Sides vorhanden, die unabhängig voneinander arbeiten können. Im Normalfall arbeiten beide Sides in Lastteilung. Im MBD gibt es vier verschiedenartige Baugruppen (MBDA, MBDH, MBDC, MBDCG).

In einer MBD-Side können bis zu acht MBDH-Baugruppen vorhanden sein. Diese sind über HDLC[2]-Leitungen mit dem Koppelnetzwerk (SN) verbunden. Die MBDH Baugruppe überträgt somit Nachrichten zu den LTGs. Jede MBDH-Baugruppe kann bis zu 252 LTGs verwalten. Für acht MBDHs ergeben sich somit 2016 LTGs.

Die MBDA-Baugruppe ist mit dem SSNC V13 über zwei redundante ATM[3]-Strecken verbunden. Pro Side können maximal fünf MBDA-Baugruppen eingesetzt werden. Jede dieser Strecken wird mit 200 Mbit/s betrieben.

Die nur einfach vorhandene MBDC-Baugruppe dient der Verbindung mit dem CP. Sie übernimmt auch die Kontrolle bei einem totalen Systemreset.

Die MBDCG-Baugruppe dient hauptsächlich der Erzeugung der Taktsignale.

In Summe ergeben sich für eine voll bestückte Side somit 15 Baugruppen. Ein voll ausgebautes MBD-System ist in Abbildung 4.3 dargestellt.

---

[2] **H**igh-level **D**ata **L**ink **C**ontrol

[3] **A**synchronous **T**ransfer **M**ode

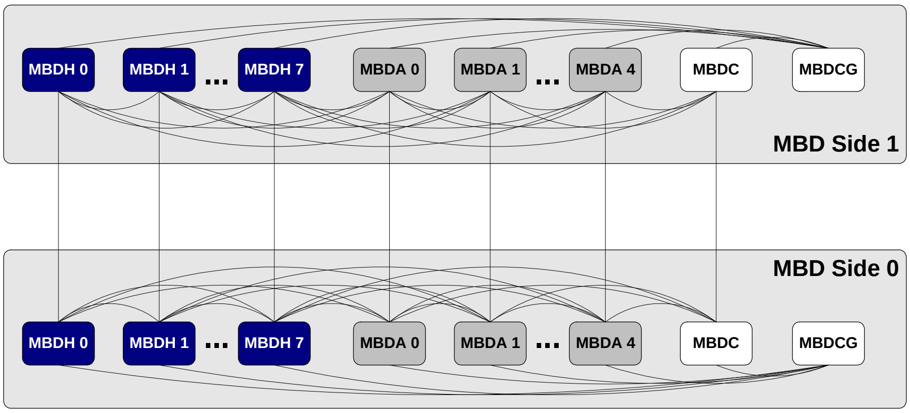


Abbildung 4.3: Verbindung der Baugruppen über IBUS

Herzstück jeder Baugruppe sind spezielle ASICs, die die Kopplung der baugruppenspezifischen Teile, wie HDLC-Links oder ATM-Schnittstelle, an den IBUS übernehmen. Neben der Kommunikationslogik enthalten diese ASICs jeweils einen eigenen MIPS-Prozessor von LSI[15] als Embedded-Core. Auf diese Weise steuern sie die Datenübertragung zwischen den eigenen spezifischen Schnittstellen und den anderen ASICS über den IBUS. Somit entspricht der Aufbau des MBD dem eines lose gekoppelten Multiprozessorsystems [10].

4.3 Der IBUS

Im MBD sind die einzelnen Baugruppen über den IBUS untereinander verbunden [13]. Jeder ASIC einer Baugruppe enthält ein IBUS-Interface mit je einer Leitung für Receive- und Transmit-Richtung. Für die Verbindung der ASICs untereinander befindet sich auf jeder Baugruppe ein IBUS-Switch. Alle IBUS-Switches sind bidirektional miteinander verbunden (s. Abb. 4.4). Der gesamte IBUS wird mit dem synchronen Systemtakt von 50 MHz betrieben. Die spezifizierte Gesamtdatenrate beträgt 240 Mbit/s. Der gesamte IBUS besteht somit aus seriellen Punkt-zu-Punkt-Verbindungen.

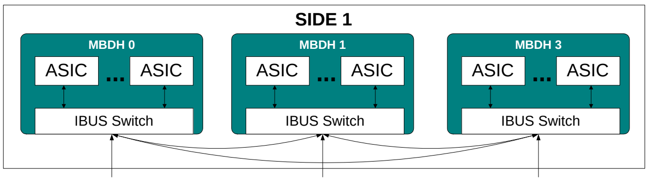


Abbildung 4.4: Verbindung ASICS mittels IBUS-Switches

Eine genaue Beschreibung des IBUS und des verwendeten Protokolls kann [13] entnommen werden.

4.4 Der Cooordination Processor CP113E

Der *coordination processor CP* ist die zentrale Steuer- und Verwaltungseinheit im *EWSD*. Als Multiprozessor kann er an den jeweiligen Leistungsbedarf angepasst werden und übernimmt folgende Aufgaben:

- Speichern und Verwalten der Programme sowie der Netzknoten- und Teilnehmerdaten
- Kommunikation mit dem Bedienungs- und Wartungszentrum
- Verarbeitung der Informationen für Verkehrslenkung, Wegesuche, Verzonung, Gebührenspeicherung, Verkehrsdatenverwaltung, Netzverwaltung
- Überwachung der Subsysteme
- Fehlererkennung
- Fehlerlokalisierung
- Verwaltung von Vermittlungsstellen- und Teilnehmerdaten

Der CP113E ist eine Multiprozessor-Recheneinheit. Die Anzahl der sogenannten *Program Execution Units (PEX)* ist von 2 bis 24 skalierbar. Die Kommunikation zwischen den einzelnen PEXen erfolgt über ein Shared Memory. Die beiden zuvor beschriebenen Funktionseinheiten werden auf zwei verschiedene Baugruppentypen abgebildet, die Baugruppe **PEXE** und die Baugruppe **CMYE**.

Die Baugruppe PEXE realisiert ihrerseits wiederum 4 verschiedene Betriebsarten und Interfaces (je nach Steckplatz im Baugruppenrahmen):

- Base Processor (BAP)

- Call Processor (CAP)
- ATM Bridge Processor (AMP) und
- Input Output Controller (IOC).

Kern der Baugruppe ist ein System on Chip ASIC, welcher Funktionsblöcke für alle angeführten Betriebsarten sowie ein *serielles High Speed Interface* zum Common Memory beinhaltet.

Die Baugruppe CMYE realisiert ein bis zu 4 Gigabyte großes Shared Memory. Außerdem wird durch einen weiteren SoC[4] ASIC die Arbitrierung der Speicherzugriffe der 24 Prozessoren durchgeführt.

---

[4] **S**ystem **o**n **C**hip

Kapitel 5

Implementierung des Synchronisationsmodells

Im Folgenden soll nun die konkrete Implementierung für den Simulator *Modelsim 5.4* von Mentor Graphics beschrieben werden. ModelSIM stellt ein *Foreign Language Interface (FLI-Interface)* bereit, mit dessen Hilfe *C Code* in die Simulation integriert werden kann. Zu diesem Zweck stellt das Interface Funktionen zur Signalbehandlung bereit. Mit Hilfe des *FLI-Interfaces* wurden nun *C* Modelle entwickelt, welche über *TCP/IP Sockets* miteinander kommunizieren. Der Synchonisationsmechanismus entspricht einem ***synchronenen Protokoll mit zentraler Steuereinheit***. Die Synchronisation wurde über ***blockierende Sockets*** implementiert. Das betreffende Modell wurde in ANSI C programmiert und anschließend mit dem GNU C-Compiler (gcc) compiliert. Der entstandene C-Quellcode umfasst ca. 1500 Zeilen. Das fertige Modell wurde dann in verschiedene Testbenches integriert, wozu eingehende Analysen und Modifikatioen des vorhandenen VHDL-Quellcodes nötig waren.

Die Konfiguration des Gesamtsystems wurde direkt im Quellcode vorgenommen. Dies bedingt allerdings, dass bei jeder neuen Konfiguration neu compiliert werden muss.

5.1 Das FLI-Interface von ModelSIM

Das *Foreign Language Interface von ModelSIM* gestattet es, eine VHDL `ARCHITECTURE` oder den `BODY` einer VHDL `Function`, durch in C geschriebenen Code zu ersetzen[21].

Um das FLI mit C Modellen zu benutzen, muss zuerst eine `ARCHITECTURE` mit dem `FOREIGN` Attribut erstellt und compiliert werden. Über dieses Attribut werden der Name der Initalisierungsfunktion sowie der Name der zu ladenden Objekt-Datei spezifiziert. Wenn ModelSIM die `ARCHITECTURE` elaboriert, wird die Initialisierungsfunktion ausgeführt. Dabei werden die Liste der Ports sowie die angegebenen Generics als Parameter übergeben. In VHDL93 kann das `FOREIGN` Attribut in folgender Weise verwendet werden:

```
ATTRIBUTE foreign:String;
ATTRIBUTE foreign of smodel_v1_arch: architecture is
                 cmodel_init $SYNCMODEL/cmodel.so";
```

Die Initialisierungsfunktion (im Beispiel die Funktion cmodel_init in Library cmodel.so) stellt den Einstiegspunkt in das C Model dar. Typische Aufgaben sind folgende:

- Allocieren von Speicher für dynamische Variablen
- Anlegen der Handles für Port-Signale
- Anlegen der Handles für Port-Drivers
- Registrierung von Callback-Funktionen
- Erzeugung von Prozessen
- Festlegen auf welche Signale die jeweiligen Prozesse sensitiv sind

Die zweite Methode zur Verwendung des FLI ist die Ersetzung von VHDL-Funktionen durch C-Code. Dabei wird eine C-Funktion programmiert, die identische Parameter und gleichen Returncode besitzt. Bei der Parameterübergabe muß bedingt durch Unterschiede zwischen C-Datentypen und VHDL-Datentypen besondere Sorgfalt angewendet werden. Details dazu finden sich in [21]. Die eigentliche Einbindung in VHDL erfolgt wieder über das `FOREIGN` Attribut.

5.2 Die Socket-Schnittstelle

Da die Kommunikation zwischen den logischen Prozessen über IP-Socket Verbindungen erfolgen soll, werden hier in aller gebotener Kürze einige relevante Grundlagen des TCP-IP Protokolls dargestellt.

Durch die Verwendung der Socket-Schnittstelle können zwei Anwenderprogramme, wovon das eine lokal und das andere auf einem entferntem System läuft, auf geregelte Weise miteinander kommunizieren. Dabei wird eine bidirektionale Verbindung aufgebaut. Die Socket-Schnittstelle ist zur Zeit das gebräuchlichste API[1] für TCP/IP[2]. Normalerweise wird über die Socket-API eine Client-Server-Beziehung zwischen zwei auf verschiedenen Rechnersystemen laufenden Anwendungskomponenten realisiert. Die beiden betreffenden Komponenten nehmen Verbindung auf, indem jede bestimmte API-Funktionen aufruft. Die Verbindung beruht auf bestimmten Datenstrukturen, genannt *Sockets.* Diese regeln den Zugang zu den Kommunikationsdiensten des Betriebssystems. Für den Datenaustausch rufen die Komponenten die Funktionen zum Senden und Empfangen in ähnlicher Weise auf, wie das bei der herkömmlichen Dateiein- und ausgabe der Fall ist. Die Socket-API kann auch dazu verwendet werden, um Zugang zu anderen Protokollen außer TCP/IP zu ermöglichen. So unterstützen Sockets zum Beispiel auch die Kommunikation zwischen einzelnen Prozessen innerhalb eines einzigen Hosts auf einem UNIX-System.

Die Socket-API definiert drei Socket-Typen, die auf unterschiedlichen Protokollebenen aufsetzen (s. Abb. 5.1):

- **Stream-Sockets** können mit Hilfe des Protokolls der Transportschicht in TCP[3] für die Kommunikation verwendet werden. Stream-Sockets unterstützen eine verbindungsorientierte (connection oriented service) Datenübertragung. Dies bedeutet, dass Datenpakete über diese Verbindung sicher und zuverlässig übertragen werden.

- **Datagramm-Sockets** stellen eine nicht verbindungsorientierte Verbindung auf UDP[4]-Ebene dar (connectionless service). Dies bedeutet, dass das Protokoll nicht garantiert, dass versendete Pakete mit Sicherheit ihr Ziel auch erreichen müssen. Daraus folgt unmittelbar, dass sich das Anwenderprogramm selbst um die Fehlersicherung kümmern muss.

---

[1] **A**pplication **P**rogramming **I**nterface

[2] **T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocoll [14]

[3] Das **Transmission Control Protocol (TCP)** ist ein verbindungsorientiertes Transportprotokoll. Die wesentlichen Dienstleistungen die das TCP-Protokoll für die Anwendungsprozesse bereitstellt, sind die hohe Zuverlässigkeit, da jede Nachricht bestätigt wird, die Verbindungsorientierung, Reihenfolgegarantie, Verlustsicherung, Zeitüberwachung einer Verbindung, das Multiplexing , die Flusskontrolle , der transparente Datentransport sowie der gesicherte Verbindungsauf und -abbau.

[4] Das **User Datagram Protocoll (UDP)** ist ein Transportprotokoll auf Schicht 4 des OSI-Referenzmodells und unterstützt den verbindungslosen Datenaustausch zwischen Rechnern. UDP wurde definiert, um auch Anwendungsprozessen die direkte Möglichkeit zu geben, Datagramme zu versenden und damit die Anforderungen transaktionsorientierten Verkehrs zu erfüllen. UDP baut direkt auf dem darunterliegenden IP-Protokoll auf.

- **Raw-Sockets** ermöglichen den Zugang zu tieferliegenden IP[5]-Prozessen. Raw-Sockets werden im Allgemeinen lediglich für Spezialaufgaben, die direkt auf die Netzwerkschicht zugreifen, verwendet. Dazu gehört zum Beispiel die Analyse von Netzwerkfehlern.

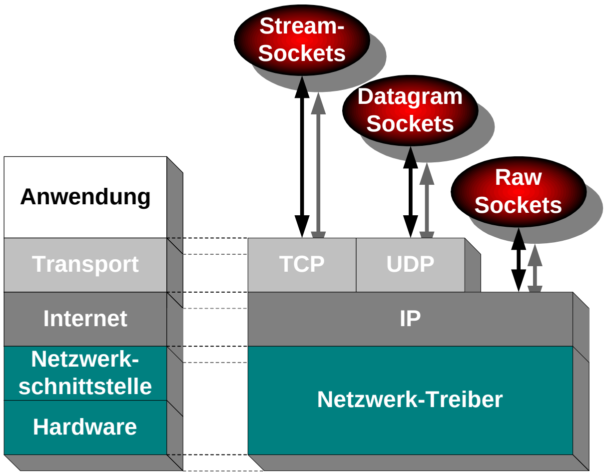


Abbildung 5.1: Drei verschiedene Socket-Typen

Die Identifizierung der Endpunkte einer Socketverbindung erfolgt über Socket-Adressen. Diese bestehen aus der IP-Adresse des Hosts und einer bestimmten Port-Nummer. Die Vergabe der Port-Nummern dient der Identifikation der verschiedenen Datenströme, die TCP gleichzeitig abarbeitet. Über diese Port-Nummern erfolgt der gesamte Datenaustausch zwischen TCP und den Anwendungsprozessen. Die Vergabe der Port-Nummern an Anwendungsprozesse geschieht in der Regel dynamisch und wahlfrei. Für bestimmte, häufig benutzte Anwendungsprozesse sind feste Port-Nummern vergeben. Diese werden als *Assigned Numbers* bezeichnet. Die Zuordnung dieser Ports wird im RFC Nr. 1700[1] festgelegt.

5.3 Spezifikation des Synchronisationsmodells

Im folgenden Abschnitt wird die Arbeitsweise des eigentlichen Synchronisationsmodells erörtert und die konkrete Implementierung vorgestellt.

---

[5] Die Aufgabe des **Internet-Protokolls (IP)** besteht darin, Datenpakete von einem Sender über mehrere Netze hinweg zu einem Empfänger zu transportieren. Die Übertragung ist paketorientiert, verbindungslos und nicht garantiert. IP garantiert weder die Einhaltung einer bestimmten Reihenfolge noch eine Ablieferung beim Empfänger.

Wie schon eingangs erklärt, kann für die zu betrachtenden Entwürfes davon ausgegangen werden, dass die einzelnen Baugruppen mit synchronem Takt arbeiten bzw. mit einem zentralen Takt versorgt werden. Speziell im MBD übernimmt diese Aufgabe sogar eine eigene Baugruppe, nämlich die MBDCG. Dieses Prinzip legt es nun nahe, mit Hilfe dieses Taktgenerators die einzelnen Modelle zu synchronisieren. Dazu werden, wie in Abbildung 5.2 die einzelnen Synchronisationsmodelle mit einem zentralen Taktgenerator mittels Sockets verbunden.

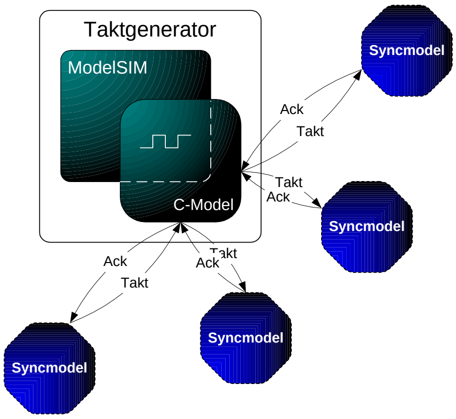


Abbildung 5.2: Der zentrale Taktgenerator

Jede Baugruppe erhält nun ihr Taktsignal direkt vom Synchronisationsmodell. Über die Socketverbindungen versendet der Taktgenerator Messages, die den genauen Zeitpunkt der nächsten Taktflanke, sowie den aktuellen Wert des Taktsignals enthalten. Die Synchronisation erfolgt durch die Verwendung von blockierenden Sockets. Dies bedeutet, dass ein Lesezugriff auf einen Socket den aufrufenden Prozess solange blockiert, bis Daten am Socket verfügbar sind. Der Taktgenerator selbst verwendet ebenfalls blockierenden Lesezugriff. Nachdem eine neue Taktflanke an alle Synchronisationsmodelle versendet wurde, wartet der Generator auf Bestätigungen (Ack) von allen Modellen, bevor neue Messages versendet werden.

5.3.1 Die Initialisierung des Synchronisationsmodells

Wie bereits in Abschnitt 5.1 gezeigt, erfolgt die Intitialisierung des Synchronisationsmodells in einer eigenen Funktion, die vom Simulator ausgeführt wird, nachdem das Modell geladen wurde. Erst nach Rückkehr aus dieser Initialisierungsfunktion, kann der Simulator die restlichen Komponenten laden.

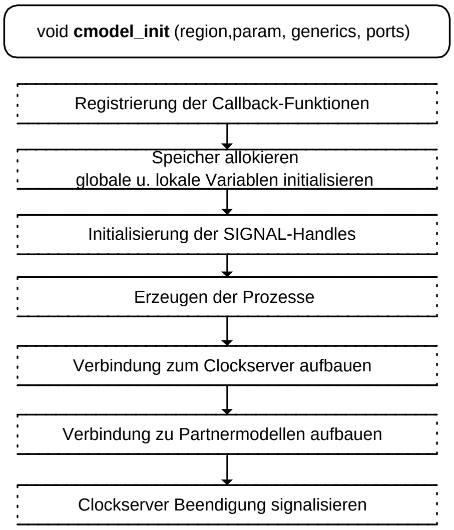


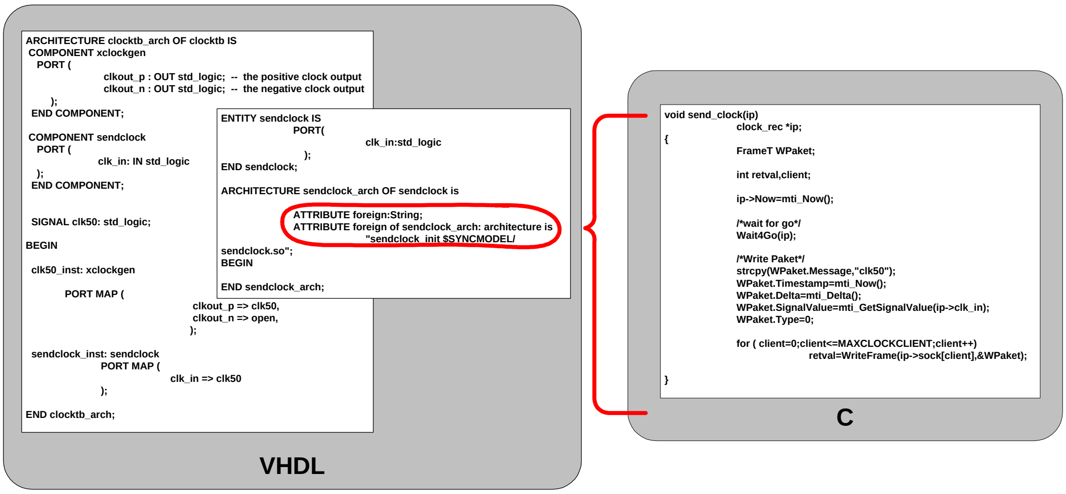
Abbildung 5.3: Die Initialisierungsfunktion

5.3.2 Der Taktgenerator

Wie schon in Bild 5.2 angedeutet, wurde auch der Taktgenerator mittels ModelSIM und FLI implementiert. Es wurde ein mit VHDL beschriebenes, konfigurierbares Modell eines Taktgenerators mit seinem Taktausgangssignal an ein C-Modell angeschlossen, welches die Anbindung an die Synchronisationsmodelle übernimmt. Die Beschreibung des VHDL-Modells kann in [18] nachgelesen werden. Das VHDL Modell wird über eine Steuerdatei konfiguriert. Innerhalb des C-Modells existiert ein auf das Taktsignal sensitiver Prozess, der bei jeder Taktflanke entsprechende Messages an alle Synchronisationsmodelle versendet.

Nach dem Versenden wartet derselbe Prozess auf Bestätigung der Messages. Auf diese Weise wird der gesamte Simulator angehalten, und somit Synchronlauf garantiert. Im Prinzip wäre auch möglich, den gesamten Taktgenerator direkt in C zu schreiben, doch die gewählte Methode bietet folgenden Vorteil. Wie zum Beispiel beim MBD wird der Takt manchmal von einer eigenen Baugruppe geliefert, die in VHDL beschrieben ist. Es ist nun einfacher an dieses VHDL-Modell ein weiteres C-Modell anzubinden als die ganze Baugruppe in C neu zu beschreiben. Die Implementierung des Taktgenerators wird durch Bild 5.4 verdeutlicht.

Auf VHDL-Seite wird der Takt von einem Generator geliefert, der über Kommandos in einer ASCII-Steuerdatei konfiguriert werden kann. Er erzeugt einen Takt mit folgenden Parametern[18]:



```
ARCHITECTURE clocktb_arch OF clocktb IS
 COMPONENT xclockgen
   PORT (
                clkout_p : OUT std_logic;  --  the positive clock output
                clkout_n : OUT std_logic;  --  the negative clock output
     );
  END COMPONENT;

 COMPONENT sendclock
   PORT (
               clk_in: IN std_logic
   );
  END COMPONENT;


  SIGNAL clk50: std_logic;

BEGIN

  clk50_inst: xclockgen

          PORT MAP (
                                   clkout_p => clk50,
                                   clkout_n => open,
                                  );

  sendclock_inst: sendclock
                PORT MAP (
                               clk_in => clk50
                );

END clocktb_arch;
```

```
ENTITY sendclock IS
            PORT(
                        clk_in:std_logic

             );
END sendclock;

ARCHITECTURE sendclock_arch OF sendclock is

            ATTRIBUTE foreign:String;
            ATTRIBUTE foreign of sendclock_arch: architecture is
                        "sendclock_init $SYNCMODEL/
sendclock.so";
BEGIN

END sendclock_arch;
```

```
void send_clock(ip)
            clock_rec *ip;
{
            FrameT WPaket;

            int retval,client;

            ip->Now=mti_Now();

            /*wait for go*/
            Wait4Go(ip);

            /*Write Paket*/
            strcpy(WPaket.Message,"clk50");
            WPaket.Timestamp=mti_Now();
            WPaket.Delta=mti_Delta();
            WPaket.SignalValue=mti_GetSignalValue(ip->clk_in);
            WPaket.Type=0;

            for ( client=0;client<=MAXCLOCKCLIENT;client++)
                        retval=WriteFrame(ip->sock[client],&WPaket);

}
```

Abbildung 5.4: Implementierung des Taktgenerators

- Init-Time
- Offset-Time
- Active-Time
- Period-Time

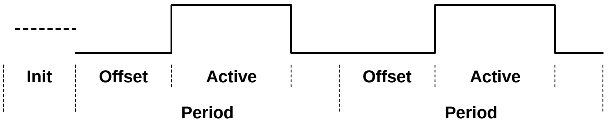


Abbildung 5.5: Die Parameter für den Taktgenerator

Eine Kommandozeile in der Steuerdatei ist folgendermaßen aufgebaut

`<timestamp> <command> [<option1> [<option2>]]`

`timestamp`	......	`Simulationszeitpunkt zu dem das Kommando ausgeführt wird`
`command`	......	`auszuführendes Kommando`
`option`	......	`Optionen für das Kommando`

Dabei sind folgende Kommandos möglich:

`timebase` *ps* | *ns* | *us* | *ms*

Setzt die Zeitbasis auf den angegebenen Wert `set` *init* | *offset* | *actice* | *period* `<intervall>`

Setzt die entsprechenden Parameter auf den angegebenen Wert.

`clockon`

Der Takt wird bei der angegebenen Zeitmarke eingeschaltet.

`clockoff`

Der Takt wird bei der angegebenen Zeitmarke ausgeschaltet.

Die Steuerdatei wird nicht in ASCII-Form in den Simulator geladen, sondern aus Performancegründen zuvor in ein quasi-binäres Format umgesetzt. Dadurch werden die zeitaufwendigen `textio` Aufrufe zur Laufzeit minimiert. Die Umsetzung erfolgt mit dem Kommando:

```
clkcom <Name der Steuerdatei>
```

Da alle Synchronisationsmodelle ihren Takt vom zentralen Taktgenerator erhalten, kann durch Modifikation des `set` Statements in der Steuerdatei die Taktperiode der Geamtsimulation auf einfache Weise verändert werden.

Die Synchronisationsmodelle reagieren auf die empfangenen Messages wie in Algorithmus 5.3.1 dargestellt. Durch den Aufruf der *FLI*-Funktion `ScheduleProcess()` ruft sich der Prozess `Set_Clock` zum Zeitpunkt der nächsten Taktflanke selbst wieder auf. Die zugehörige Funktion wird dabei aber vollständig beendet, wodurch kein rekursiver Funktionsaufruf erfolgt.

```
Process Set_Clock()
      receive next ClockPaket
      create ClockEdges (positive and negative)
      ScheduleProcess Setclock at time ClockPaket.NextClock
      Send GO to Clockserver
end Process
```

Algorithmus 5.3.1: Arbeitsweise der Funktion Set_Clock

5.3.3 Verbindungsaufbau

Während der Initialisierung des Synchronisationsmodells wird zuerst die Verbindung zum Taktgenerator und danach die logischen Verbindungen (=*LCs*) zu den Partnermodellen aufgebaut. Der Taktgenerator erwartet die Anbindung der Synchronisationsmodelle in aufsteigender Reihenfolge. Schon bei der Konfiguration muss der Anwender darauf achten, dass beim Verbindungsaufbau keine Deadlock-Situation eintreten kann. Dies wird durch geeignete Wahl der Connection-IDs erreicht. Eine beispielhafte Konfiguration ist in Abbildung 5.6 dargestellt.

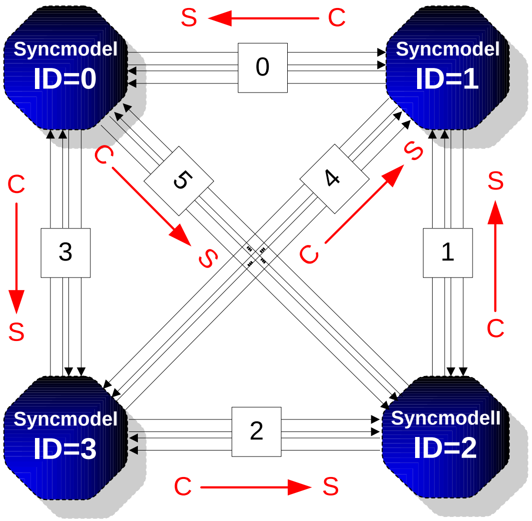


Abbildung 5.6: Beispielhafte Konfiguration von vier Synchronisationsmodellen

Bei den logischen Verbindungen ist zu beachten, dass jeder Kanal ein *Server- (S)* und ein *Client-Ende (C)* besitzt. Der Verbindungsaufbau erfolgt in der Weise, dass eine Seite einen Server-Socket öffnet und danach auf die Anbindung des zugehörigen Clients wartet. Die Kanäle werden dabei von jedem Modell in aufsteigender Reihenfolge der zugehörigen Connection-IDs aufgebaut, wodurch ein Deadlock verhindert wird. Die gesamte Topologie wird, wie in Abbildung 5.7 dargestellt, in einer Tabelle gespeichert. Zusätzlich zu den Endpunkten der logischen Kanäle muss diese Tabelle auch Information über Anzahl und Richtung der Signale, die über diesen Kanal versendet werden sollen, enthalten. Zusätzlich enthält die Tabelle auch die serverseitigen Port-Nummern der *LCs*. Konkret wird die beschriebene Tabelle in der Header-Datei `connections.h` abgespeichert.

Der zeitliche Ablauf des Verbindungsaufbaus für das angegebene Beispiel wird in Abbildung 5.8 gezeigt.

Connection-ID	Server-Model	Client-Model
0	0	1
1	0	2
2	2	3
3	0	3
4	1	3
5	1	3

Abbildung 5.7: Konfigurationstabelle für das angegebene Beispiel

Modell 0 wird gestartet, öffnet den Server-Socket von Verbindung 0 und wartet auf die Anbindung von Modell 1 auf diesem Kanal. Nachdem Modell 0 den Kanal 0 verbunden hat, kann Modell 0 den Kanal 3 öffnen. Danach öffnet Modell 1 Kanal 1 und wartet auf die Anbindung von Modell 2, usw. Alle anderen Kanäle werden ebenfalls nach der gleichen Sequenz verbunden.

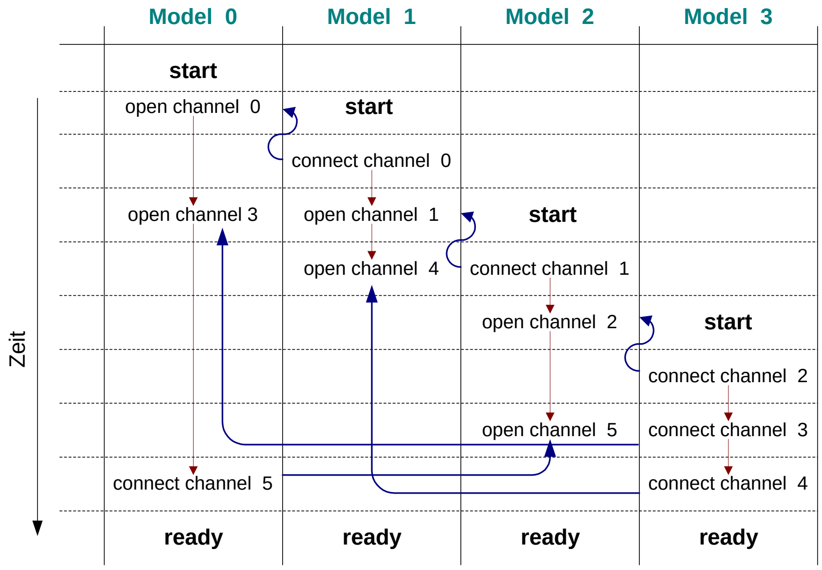


Abbildung 5.8: zeitlicher Ablauf beim Verbindungsaufbau

Die eigentliche Implemetierung in C erfolgt mit Hilfe von Funktionen aus der Standard-Library `sockets`. Um diese Funktionen verwenden zu können, muss die betreffende Header-Datei, die in der Regel auf allen UNIX Systemen vorhanden ist, mittels "`#include <sys/socket.h>`" im Quellcode eingebunden werden. Der Aufbau der Socket-Verbindung erfolgt nun auf Server- bzw. Client-Seite auf unterschiedliche

Weise. In beiden Fällen muss im ersten Schritt durch Aufruf der Funktion `socket()` ein File-Descriptor für den neuen Socket erzeugt werden. Serverseitig wird im Anschluss die Funktion `bind()` aufgerufen. Diese ordnet dem Socket-Descriptor eine lokale Port-Nummer zu. Danach erfolgt der Aufruf der Funktion `listen()`, welche erst verlassen wird, wenn sich eine andere Anwendung auf den betreffenden Socket verbunden hat. Auf der Client-Seite folgt auf den Aufruf von `socket()` sofort die Funktion `connect()`, wodurch der Server aus der `listen()`-Funktion zurückkehren kann. Ein Aufruf von `connect()` liefert einen Fehlercode zurück, falls noch kein Server am betreffenden Socket lauscht. Sonst ist der Socket-Descriptor nach der Rückkehr aus dieser Funktion verfügbar. Die Funktion `accept()` liefert schließlich einen neuen Socket-Descriptor an den Server zurück, auf dem nun sowohl Schreibe- als auch Leseoperationen erfolgen können. Der beschriebene Ablauf ist in der folgenden Abbildung 5.9 dargestellt.

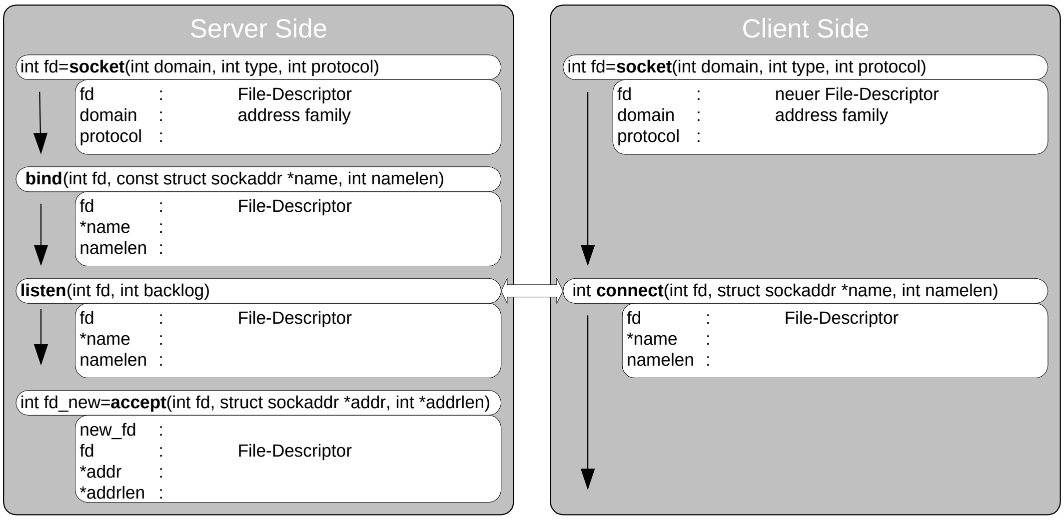


Abbildung 5.9: Die benötigten Funktionen aus der Library `sockets`

5.3.4 Der Signalabgleich

Wie eingangs erwähnt müssen die einzelnen Synchronisationsmodelle zum Zeitpunkt jeder Taktflanke alle relevanten Signale abgleichen. An dieser Stelle ergibt sich ein entscheidendes Problem aus der inneren Arbeitsweise des Event-gesteuerten Simulators *ModelSIM*. Dies erfordert eine eingehendere Betrachtung.

Das Prinzip der diskreten Event Simulation wurde in Abschnitt 3.1.2 beschrieben. Aus dem dort Gesagten ergibt sich, dass während der Abarbeitung der Eventqueue nicht vorhergesagt werden kann, wieviele Events innerhalb des aktuellen Simulationsschrittes noch folgen werden. Da jedes Event einen darauf sensitiven Prozess anstoßen kann, ist es von vorne herein unbestimmt wie oft ein Prozess ausgeführt wird, bzw. wie viele Delta-Zyklen innerhalb eines Simulationsschritts erfolgen. Entscheidend für die korrekte Funktion der Synchronisationsmodelle ist, dass der Signalabgleich erst erfolgt, wenn sichergestellt werden kann, dass sich die betreffenden Signale in einem folgenden Delta-Zyklus nicht wieder verändern. Der Abgleich darf also erst erfolgen, wenn der Vektor der Eingangssignale am Synchronisationmodell stabil anliegt.

Ab *VHDL93* wurde das Schlüsselwort `postponed` eingeführt, welches die Lösung des beschriebenen Problems darstellt. Wird ein VHDL-Prozess als `postponed` deklariert, wird dieser erst ausgeführt, wenn alle Signale in seiner Sensitivity List stabil sind. Leider unterstützt das *FLI-Interface* von *ModelSIM* keine `postponed processes`, wodurch folgender Umweg erforderlich wird.

Es wird mit *VHDL* ein auf den Eingangsvektor sensitiver `postponed process` erzeugt, der somit sicher erst nach Stabilisierung der Eingangssignale ausgeführt wird. Innerhalb dieses Prozesses wird dann mittels *FLI* eine C-Funktion aus einem Paket (vgl. Abschnitt 5.1) aufgerufen, welche den Signalabgleich durchführt. Somit wird garantiert, dass der Abgleich genau einmal erfolgt.

Im vorliegenden Fall wurde dieses Prinzip folgendermaßen implementiert: Innerhalb der *VHDL*-Deklaration des Synchronisationsmodells wird die Entity `trigger` eingebunden. Diese erwartet am Eingang den gesamten Eingangsvektor des Syncmodells. Innerhalb der Architecture `Trigger_arch` existiert ein auf den Eingangsvektor sensitiver `postponed` Prozess, welcher somit erst aufgerufen wird, wenn der Eingangsvektor stabil anliegt. Sobald dieser Prozess nun aktiv wird, wird die Funktion `Trigger_Procedure()` aufgerufen, welche ihrerseits die Funktion `do_exchange()` aufruft (siehe Bild 5.10).

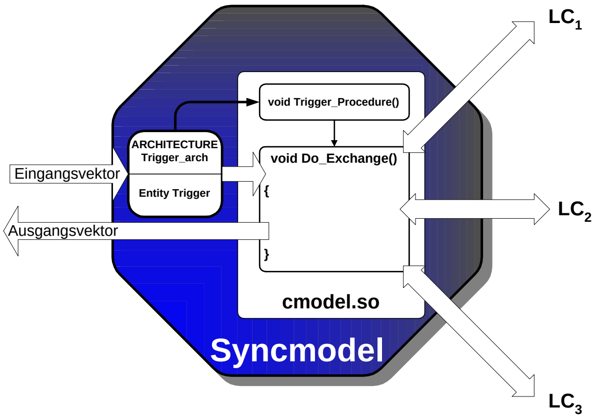


Abbildung 5.10: Interne Arbeitsweise des Syncmodels

Wichtig ist, dass sich diese beiden Funktionen in der selben Shared Objekt-Datei (hier `cmodel.so`) wie die restlichen Funktionen des Synchronisationsmodells befinden, damit alle von der Initalisierungsfunktion gespeicherten Zeiger auch zur Verfügung stehen.

Die Arbeitsweise der Funktion `do_exchange()` zeigt Algorithmus 5.3.2, dem folgendes Prinzip zugrunde liegt:

Vor dem Versenden einer Message wird geprüft, ob sich das betreffende Signal gegenüber der letzten Taktflanke überhaupt verändert hat. Wenn keine neue Signalflanke vorliegt, braucht auch keine Message versendet werden. Zu diesem Zweck müssen am Ende der Funktion aber alle Signalwerte in einem Feld gespeichert werden, um beim nächsten Funktionsaufruf als Referenz zur Verfügung zu stehen. Die notwendigen Signale werden in der Reihenfolge ihres Auftretens im Eingangsvektor abgearbeitet. Um die Synchronisierung zu ermöglichen muss auf jedem Kanal das letzte [6] Signal in jedem Falle übertragen werden. Die Anzahl der zu übertragenden Signale wird aus der in Abschnitt 5.3.3 beschriebenen Konfigurationstabelle ermittelt.

---

[6] entsprechend der Position im Eingangsvektor

```
Process do_exchange(clock)
    CONNECTIONS          // Anzahl der Verbindungen zum Modell
    INPUTS[CONNECTIONS]  // Anzahl der Signale am Eingang
    OUTPUTS[CONNECTIONS] // Anzahl der Signale am Ausgang

    for n = 0 to CONNECTIONS DO
    {
      for i = 1 to INPUTS-1   // senden der nötigen Signale
        if (signal has changed) send signal[i] to connection n
      send signal[INPUTS] to connection n // senden des
    }                                     // letzten Signals

    For n = 0 to CONNECTIONS DO
    {
    i=0
    repeat
      receive signal
      i=ID of the received Signal
    until i is OUTPUTS[n]
    }

    For all Signals DO
    {
        SetSignalValue()
        Save SignalValues
    }

end Process do_exchange
```

Algorithmus 5.3.2: Algorithmus zum Signalabgleich

Kapitel 6

Integration des Synchronisationsmodells

Im Folgenden soll nun die Einbindung des vorgestellten Synchronisationsmodells in bestehende Entwürfe dargestellt werden. Dieses Kapitel stellt auch eine Zusammenfassung des praktischen Teils der Arbeit dar. Die im vorigen Abschnitt präsentierte Lösung wurde erst im Laufe der Tests am IBUS (s. Abschnitt 4.3) vollständig implementiert und laufend verbessert. Danach erfolgte die Aufspaltung der vorhandenen MBD (s. Abschnitt 4.2) und CP113E (s. Abschnitt 4.4) Testbenches. Alle präsentierten Simulationen und Messungen wurden im firmeninternen Netzwerk während normaler Systemauslastung durchgeführt.

6.1 Test mit dem IBUS-Modell

Die Funktionsweise des IBUS wurde bereits in Abschnitt 4.3 erörtert. An dieser Stelle soll nun, die Aufspaltung zweier sehr einfacher Testbenches, bestehend aus zwei IBUS-Interfaces und einem Taktgenerator, beschrieben werden. Dieser Test soll in erster Linie der Sicherstellung der Korrektheit des Simulationsergebnisses dienen. Eine Aussage über einen Performancegewinn ist auf Grund der geringen Komplexität der ursprünglichen Testbench nicht zu erwarten. Wahrscheinlicher ist, dass der durch den zusätzlichen Netzwerkverkehr verursachte Overhead die Simulation sogar verlangsamt.

6.1.1 Das IBUS-Interface

Das IBUS-Interface ist die Schnittstelle jedes ASICs zum IBUS. Die Aufgabe dieses Interfaces ist es, Nutzdaten aus dem SDRAM über den IBUS zu versenden, bzw. über den IBUS empfangene Daten in das SDRAM zu schreiben. Die genaue Funktionsweise dieses Interfaces kann in [13] nachgelesen werden, ist aber für die folgenden Betrachtungen nicht von Bedeutung. Für die Simulation wurde ein bestehendes VHDL-Modell des IBUS-Interfaces verwendet, welches über eine ASCII-Datei gesteuert werden kann. Mit Hilfe dieses VHDL-Modells kann somit alleine der Verkehr über den IBUS simuliert werden, ohne dass das gesamte MBD-System dahinter steht.

6.1.2 Der IBUS-Tracer

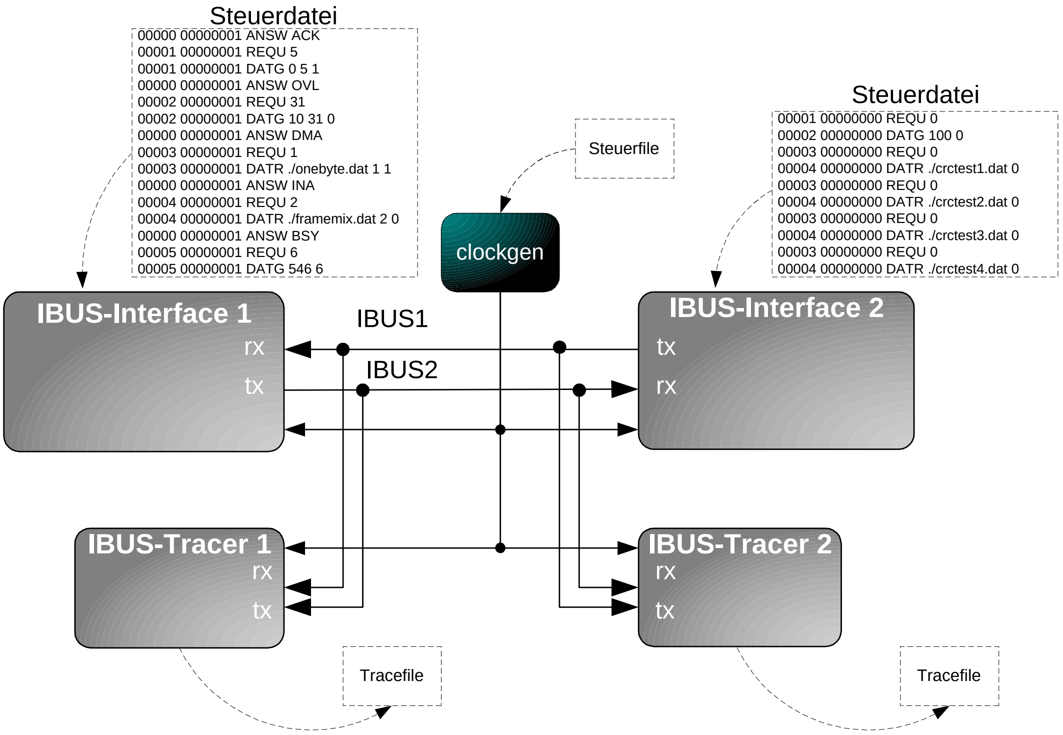
Um den Verkehr auf dem IBUS leichter verfolgen zu können, wird ein VHDL-Modell verwendet, welches mittels `TEXTIO`-Funktionen die verschiedenen Rahmen, die über den Bus laufen, in eine Datei schreibt. Dieses Modell wird als IBUS-Tracer bezeichnet. Die erzeugten Ausgabedateien werden IBUS-Tracefiles genannt.

6.1.3 Testfall mit zwei IBUS-Interfaces

Ausgangspunkt für den ersten Einsatz des Synchronisationsmodells war die in Abbildung 6.1 schematisch dargestellte Testbench. Zwei IBUS-Interfaces, die jeweils von einer eigenen Textdatei gesteuert [1] werden, versenden über zwei Leitungen IBUS-Frames. Parallel zu jedem IBUS-Interface ist ein IBUS-Tracer geschalten, der den Verkehr auf den Leitungen aufzeichnet und in einer eigenen Datei abspeichert. Alle vier Entities werden von einem gemeinsamen Taktgenerator mit einem 50 MHz Takt versorgt.

---

[1] Die genaue Bedeutung der Steuerbefehle kann in [13] nachgelesen werden



```
00000 00000001 ANSW ACK
00001 00000001 REQU 5
00001 00000001 DATG 0 5 1
00000 00000001 ANSW OVL
00002 00000001 REQU 31
00002 00000001 DATG 10 31 0
00000 00000001 ANSW DMA
00003 00000001 REQU 1
00003 00000001 DATR ./onebyte.dat 1 1
00000 00000001 ANSW INA
00004 00000001 REQU 2
00004 00000001 DATR ./framemix.dat 2 0
00000 00000001 ANSW BSY
00005 00000001 REQU 6
00005 00000001 DATG 546 6
```

```
00001 00000000 REQU 0
00002 00000000 DATG 100 0
00003 00000000 REQU 0
00004 00000000 DATR ./crctest1.dat 0
00003 00000000 REQU 0
00004 00000000 DATR ./crctest2.dat 0
00003 00000000 REQU 0
00004 00000000 DATR ./crctest3.dat 0
00003 00000000 REQU 0
00004 00000000 DATR ./crctest4.dat 0
```

Abbildung 6.1: ursprüngliche Testbench mit zwei IBUS-Interfaces, zwei IBUS-Tracers und einem Taktgenerator

Die dargestellte Testbench wurde mit Hilfe der Synchronisationsmodelle in folgender Weise aufgetrennt (siehe auch Abbildung 6.2): Dabei wurden zwei neue Testbenches mit jeweils einem IBUS-Interface und einem Tracer erstellt. Die ursprüngliche Verbindung über die beiden Signale (IBUS1, IBUS2) wurde aufgetrennt, und an die entstandene Schnittstelle wurde jeweils ein Syncmodel angeschlossen. Diese beiden Testbenches wurden auf zwei eigenen Workstations (WS1,WS2) mit ModelSIM simuliert. Der Takt wurde, wie in Abschnitt 5.3.2 gezeigt, von einem eigenen Taktgenerator, der ebenfalls auf einer eigenen Workstation (WS0) gestartet wurde, geliefert.

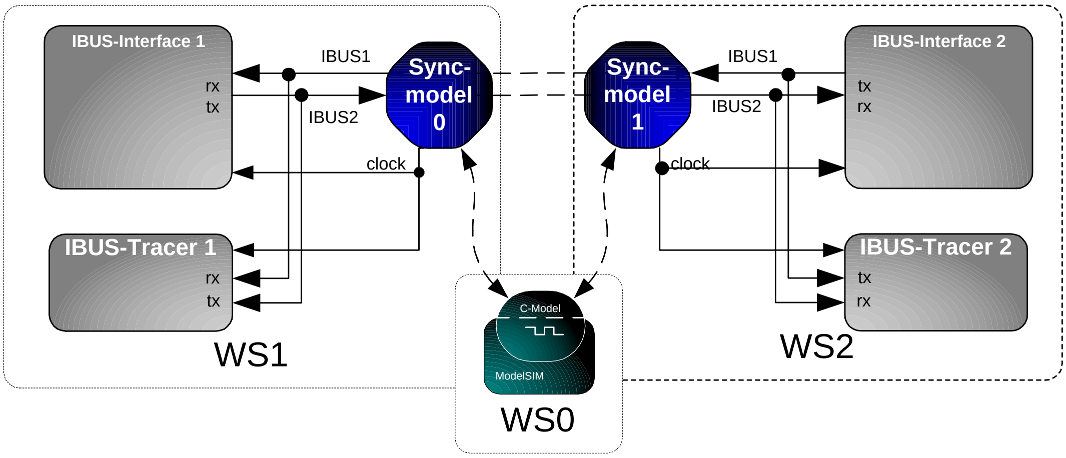


Abbildung 6.2: Verteilte Simulation der ursprünglichen Testbench

6.1.3.1 Ergebnis des Testfalls IBUS-Interfaces

Durch Vergleich der von den Tracern gelieferten Tracefiles konnte gezeigt werden, dass die verteilte Simulation zu demselben Ergebnis führt wie die Simulation des Gesamtsystems auf einer Workstation. Die gewählte Methode eignet sich somit zur “Umleitung” des IBUS über TCP/IP. Alle Steuerfiles, und die zugehörigen Tracefiles sind im Anhang zu finden. Wie erwartet war im vorliegenden Falle die verteilte Simulation aber deutlich langsamer. Der Performanceverlust war so deutlich, dass auf genaue Messungen verzichtet werden konnte.

Der Grund für die Verlangsamung ist wie schon erwähnt in der Einfachheit des ursprünglichen Entwurfes zu suchen, was sich darin äußert, dass die einzelnen Simulatoren zwischen den Taktflanken kaum Rechenzeit konsumieren. Zu jeder Taktflanke erfolgt nun aber auch ein Signalabgleich, für welchen nun zusätzliche Rechenzeit benötigt wird. Daraus ergibt sich, dass die einzelnen Modelle relativ lange mit dem Signalabgleich beschäftigt sind, was den Performanceverlust erklärt.

Aus dem Gesagten kann man die Schlussfolgerung ableiten, dass der Performancegewinn proportional der Komplexität des Entwurfes ist. Genauer gesagt hängt der Geschwindigkeitszuwachs in hohem Maße davon ab, wie viel Rechenzeit die Gesamtsimulation zwischen zwei Taktflanken benötigt.

Um diese Vermutung zu bestätigen, wurden im nächsten Schritt noch zwei IBUS-Interfaces mit den zugehörigen Tracern integriert. Entgegen den Erwartungen konnte aber noch kein merklicher Gewinn erzielt werden. Der Entwurf war somit immer noch viel zu “einfach”. Nachdem somit gezeigt werden konnte, dass die gewählte Methode praktisch einsetzbar ist, wurde nun schrittweise versucht, komplexere Entwürfe aufzuteilen.

6.1.4 Testfall mit sechs IBUS-Interfaces

Der nächste Testfall beschäftigt sich mit der Aufspaltung einer Testbench in drei Teile. Zu diesem Zweck musste das Synchronisationsmodell für den Betrieb in dieser Dreifachkonstellation erweitert werden. Wie im ersten Testfall wurden wieder mehrere IBUS-Interfaces mit zugehörigen Tracern verwendet. In Abbildung 6.3 wird bereits das partitionierte Gesamtsystem dargestellt. Die einzelnen IBUS-Interfaces und der Taktgenerator wurden mit den gleichen Steuerdateien wie im vorigen Testfall betrieben.

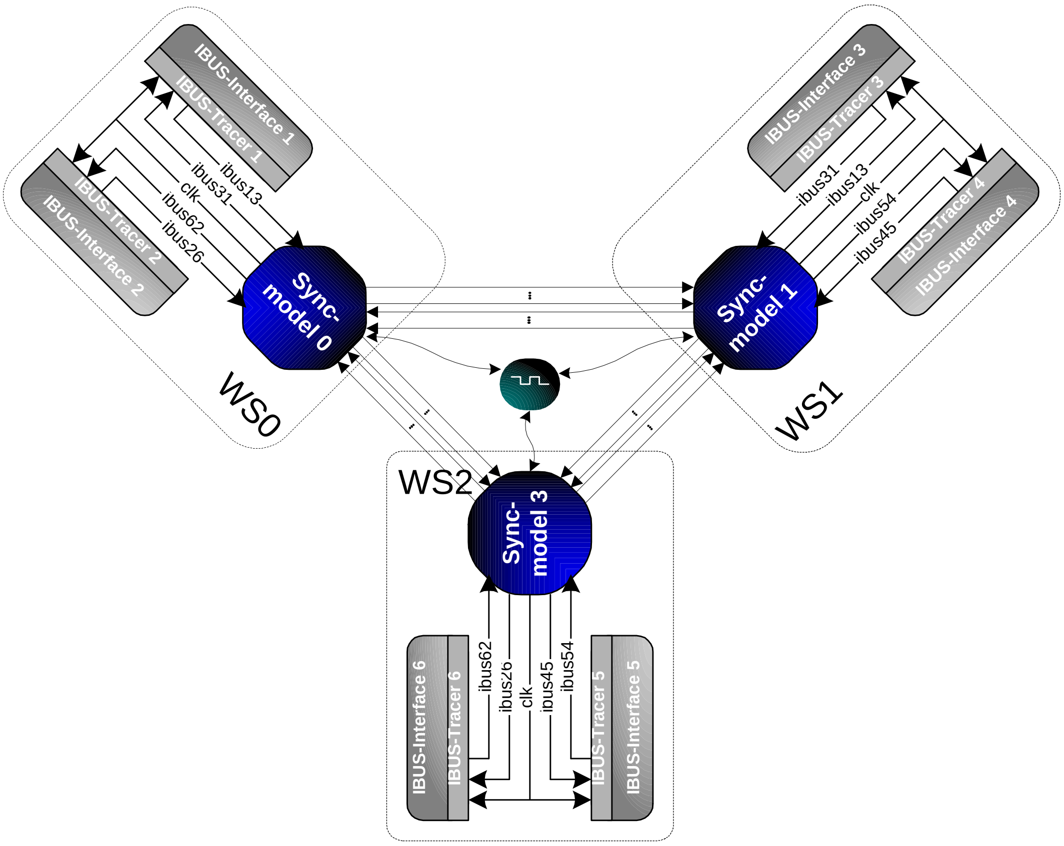


Abbildung 6.3: Testfall mit sechs IBUS-Interfaces

Wie erwartet lieferte die Simulation wieder das korrekte Ergebnis, was durch Vergleich der Tracefiles bestätigt werden konnte. Ein direkter Geschwindigkeitsvergleich zeigte wieder, dass die Aufspaltung dieser einfachen Testbench keine Beschleunigung brachte.

6.2 Integration in die MBD-Testbench

Im folgenden Abschitt wird das eigentliche Hauptziel der Tests in Angriff genommen, nämlich die Aufteilung der Systemtestbench, des in Abschnitt 4.2 beschriebenen, Message Buffer Typ D. Dieser Test lässt aufgrund der Komplexität des Gesamtsystems erstmals einen Geschwindigkeitsschub erwarten.

6.2.1 Die MBD-Testbench

Für die Simulation des MBD wurde eine einzige hierarchische Testbench erstellt, welche den gesamten MBD mit allen seinen externen Schnittstellen umfasst. Diese Testbench wurde für alle Bereiche der Simulation verwendet, von der Simulation der ASICS bis zum Gesamtsystem. Dabei wurden je nach Testanforderung mittels VHDL-Konfigurationen nur die notwendigen Baugruppen bestückt. [18]

Die Struktur der betreffenden Testbench ist in Abbildung 6.4 dargestellt. Die oberste Ebene der Testbench besteht dabei aus zwei Backplane-Stromläufen(BP0, BP1) für die beiden MBD-Hälften. Innerhalb der Backplanes befinden sich die Baugruppensteckplätze (z. B.: MBDH_0_I.....), die bei Bedarf mit entprechenden Baugruppen (z. B.: MBDH_SHELL2) bestückt werden können. Die lokalen Modelle der ASICS sowie die Tracer sind innerhalb der sogenannten Baugruppenshells angeordnet.

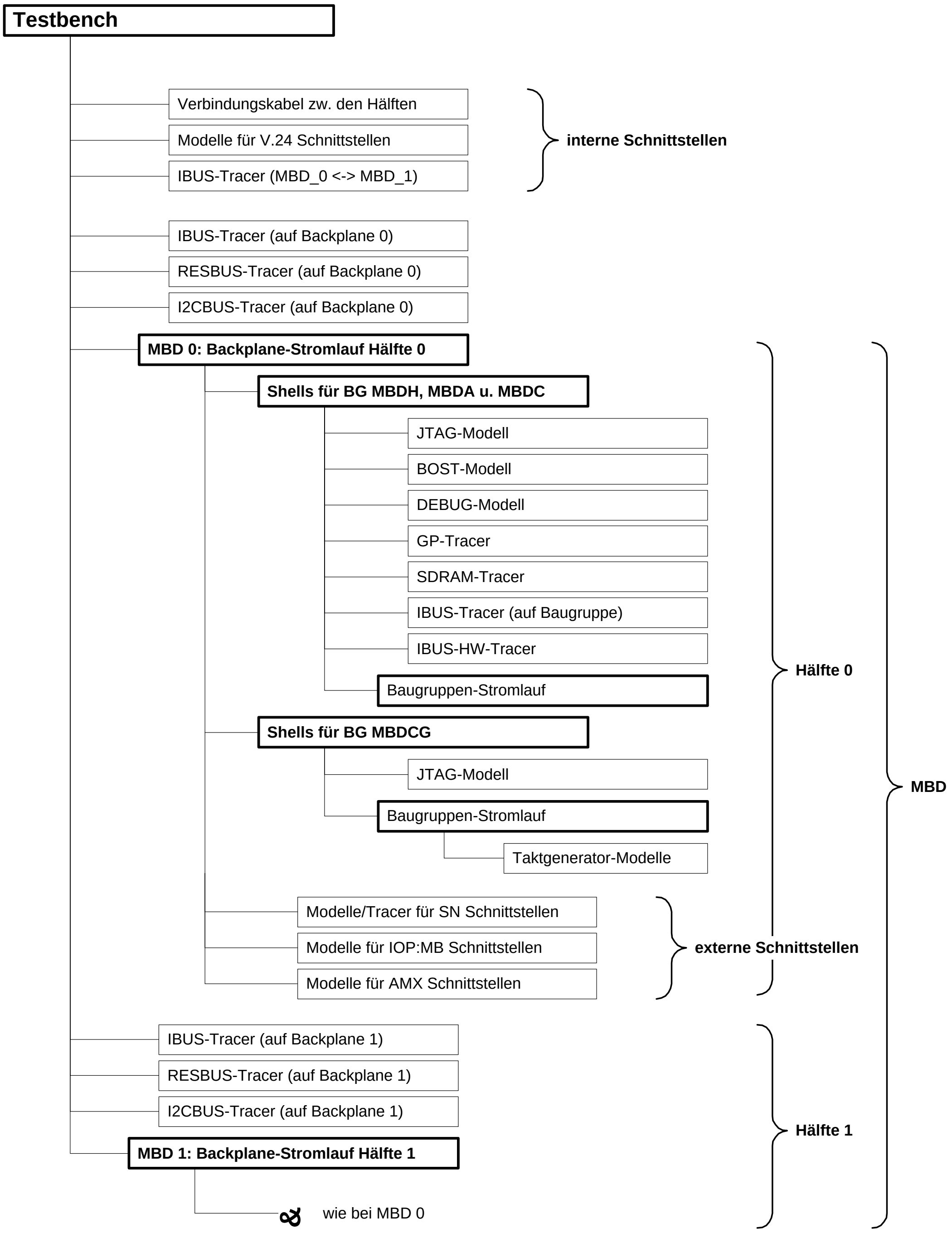


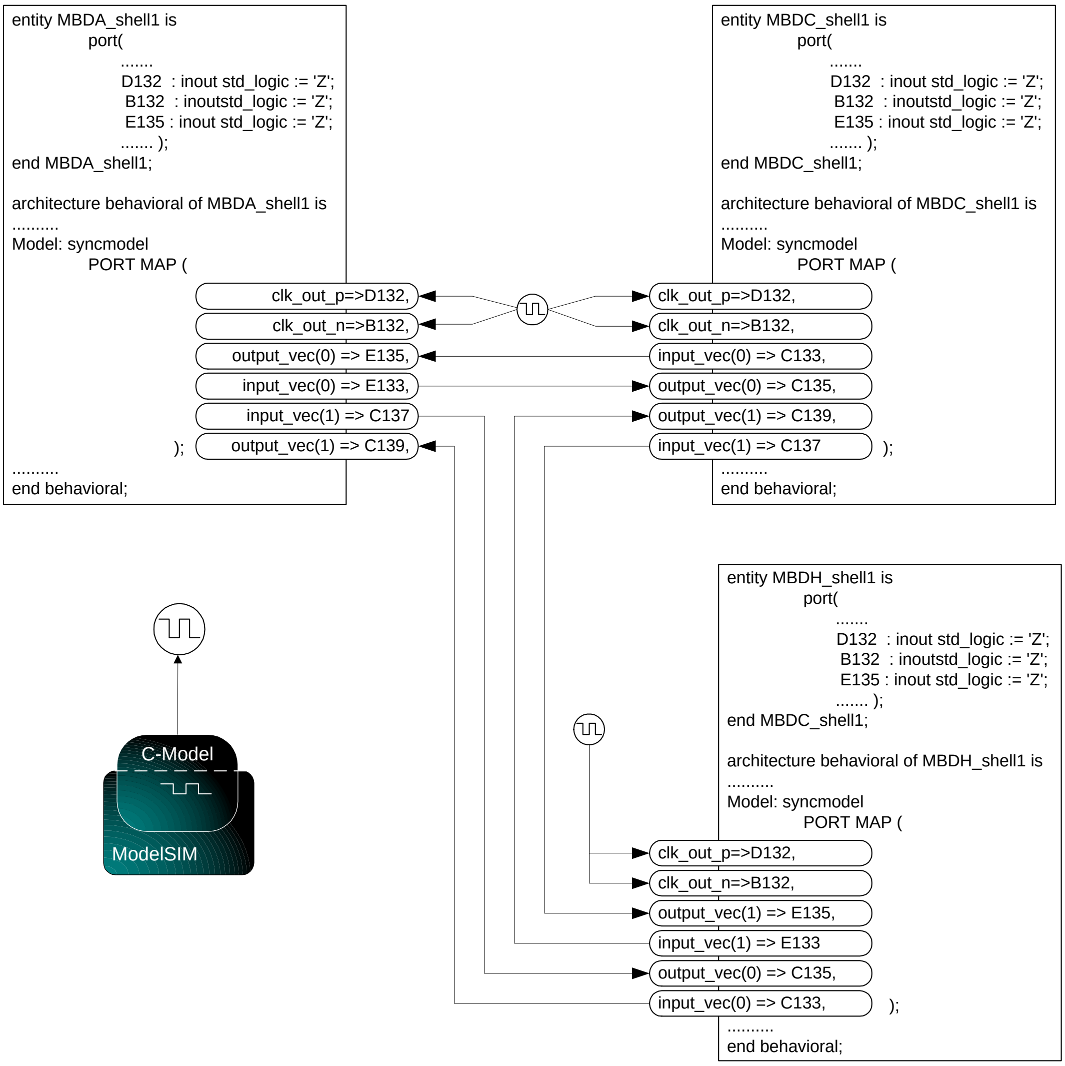
Abbildung 6.4: Struktur der MBD-Testbench

6.2.2 Aufteilung der MBD-Testbench mit vier unterschiedlichen konfigurierten Baugruppen

Für die nun folgenden Betrachtungen wurde ein Testfall für vier Baugruppen(MBDA, MBDH, MBDC und MBDCG) herangezogen. Dabei werden alle Baugruppen von der MBDCG mit einem differenziellen Takt versorgt. Die MBDC kommuniziert über jeweils zwei IBUS-Leitungen mit den anderen beiden Baugruppen. An den IBUS-Leitungen hängen IBUS-Tracer, welche die Kommunikation aufzeichnen. Auf einer Sun Solaris Workstation mit 4 CPUs und 4 GB Arbeitsspeicher dauerte die Simulation des 2 ms langen Testfalls 36 Stunden.

Die Aufteilung der MBD-Testbench erfolgte in folgender Weise: Es wurden ausgehend von der ursprünglichen Testbench drei neue Testbenches erstellt, wobei in jeder jeweils nur eine Baugruppe konfiguriert wurde. Die MBDCG wurde vollständig durch den in 5.3.2 beschriebenen Taktgenerator ersetzt. Im Vergleich zu den anderen drei Baugruppen ist die MBDCG sehr einfach und trägt somit kaum zur Systemauslastung bei. Es ist somit keine wesentliche Verfälschung der Ergebnisse wenn die MBDCG im in der verteilten Simulation nicht mehr auftritt.

Innerhalb der Baugruppenshells von MBDA, MBDH und MBDC wurde zusätzlich ein Synchronisationsmodell integriert. Die baugruppeninternen Signale für den IBUS und die Taktsignale wurden direkt innerhalb der Shell an das Synchronisationsmodell angeschlossen. Die Tracer wurden in der Shell belassen. In Abbildung 6.5 sind die betreffenden Code-Fragmente aus den Baugruppen-Shells dargestellt. Innerhalb der jeweiligen Portdefinition sind dabei die Steckersignale zu erkennen, über die die Shell mit der Backplane verbunden wird. Die Pfeile im Diagramm symbolisieren die logischen Verbindungen, über TCP/IP.



```
entity MBDA_shell1 is
            port(
                  .......
                  D132  : inout std_logic := 'Z';
                   B132  : inoutstd_logic := 'Z';
                   E135 : inout std_logic := 'Z';
                  ....... );
end MBDA_shell1;

architecture behavioral of MBDA_shell1 is
..........
Model: syncmodel
            PORT MAP (
                        clk_out_p=>D132,
                        clk_out_n=>B132,
                     output_vec(0) => E135,
                      input_vec(0) => E133,
                       input_vec(1) => C137
                  );  output_vec(1) => C139,
..........
end behavioral;
```

```
entity MBDC_shell1 is
            port(
                  .......
                  D132  : inout std_logic := 'Z';
                   B132  : inoutstd_logic := 'Z';
                   E135 : inout std_logic := 'Z';
                  ....... );
end MBDC_shell1;

architecture behavioral of MBDC_shell1 is
..........
Model: syncmodel
            PORT MAP (
     clk_out_p=>D132,
     clk_out_n=>B132,
     input_vec(0) => C133,
     output_vec(0) => C135,
     output_vec(1) => C139,
     input_vec(1) => C137       );
..........
end behavioral;
```

```
entity MBDH_shell1 is
            port(
                  .......
                  D132  : inout std_logic := 'Z';
                   B132  : inoutstd_logic := 'Z';
                   E135 : inout std_logic := 'Z';
                  ....... );
end MBDC_shell1;

architecture behavioral of MBDH_shell1 is
..........
Model: syncmodel
            PORT MAP (
     clk_out_p=>D132,
     clk_out_n=>B132,
     output_vec(1) => E135,
     input_vec(1) => E133
     output_vec(0) => C135,
     input_vec(0) => C133,       );
..........
end behavioral;
```

Abbildung 6.5: Aufteilung der MBD-Testbench mit vier konfigurierten Baugruppen

6.2.2.1 Simulation des Testfalls

Nach Fertigstellung der neuen Testbenches wurde das Gesamtsystem auf vier gleichwertigen Sun Solaris Workstations simuliert, und die Performance gemessen. Parallel dazu wurde das ursprüngliche System auf einer eigenen Workstation, sowie auch jede Baugruppe für sich alleine, simuliert.

Durch direkten Vergleich der von den am IBUS angeschlossenen Tracern gelieferten Tracefiles konnte am Ende wieder die Korrektheit des Simulationsergebnisses gezeigt werden.

Um genaue Aussagen über Performance machen zu können wurde im Laufe der Simulation ständig der Fortschritt aufgezeichnet. Dies geschah mit einem einfachen Script, welches immer nach 20 μs Simulationsfortschritt die dazu benötigte Zeit in eine Datei ausgab. Die gesammelten Daten wurden dann mit einem Tabellenkalkulationsprogramm weiterverarbeitet und auch grafisch aufbereitet.

6.2.2.2 Ergebnisse

Die genauen Auswertungen sind in den folgenden Abbildungen dargestellt. In den Diagrammen ist dabei immer die relative Simulationsgeschwindigkeit, gemessen in simulierten Taktzyklen pro Sekunde (clk/s) über der simulierten Zeit (μs) aufgetragen. Zur Beschriftung der Kurven ist zu sagen, dass z. B.: C_mori bedeutet, dass es sich um den Verlauf der Simulation der Baugruppe MBDC auf der Workstation mit dem Namen "mori" handelt. Die recht massiven Unregelmäßigkeiten sind zum Teil durch Spitzen in der Netzwerkbelastung oder Workstationauslastung bedingt.

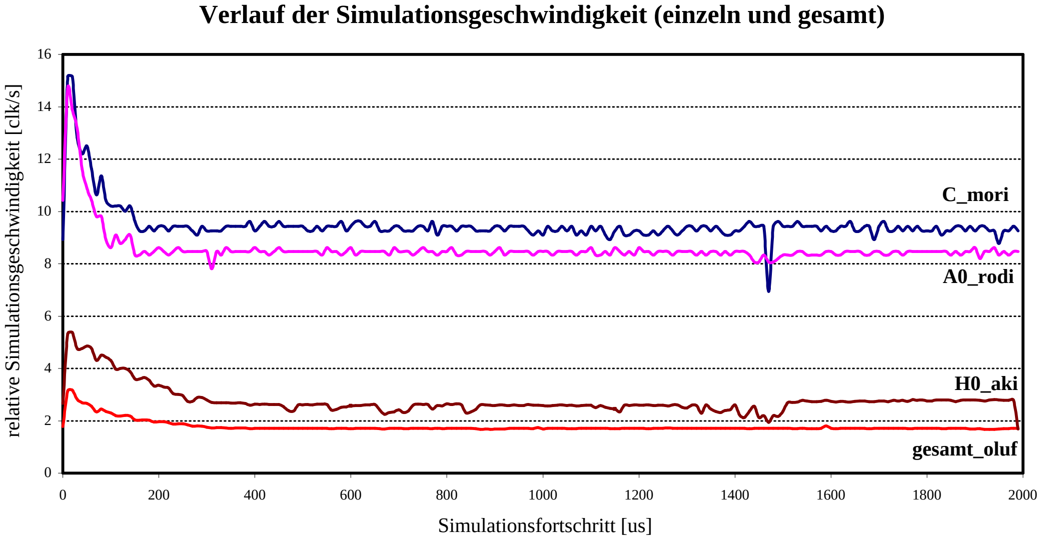


Abbildung 6.6: Simulation des Gesamtsystems sowie der einzelnen Baugruppen

Bild 6.6 zeigt die Verläufe der verschiedenen Simulationen des Testfalls. Zu sehen sind die zu den Simulationen der losgelösten Baugruppen(**C_mori, A0_rodi, H0_aki**) gehörigen Kurven, sowie jene der gesamten (**gesamt_oluf**) Simulation.

Dabei ist sehr deutlich zu erkennen, dass die Simulationsgeschwindigkeit bis zum Zeitpunkt 45 μs abfällt und danach konstant bleibt. Der Grund dafür ist, dass die Baugruppen erst nach 45 μs vollständig initialisiert sind. Weiters ist zu erkennen, dass bei den Einzelsimulationen die MBDH-Baugruppe (**H0_aki**) deutlich langsamer ist.

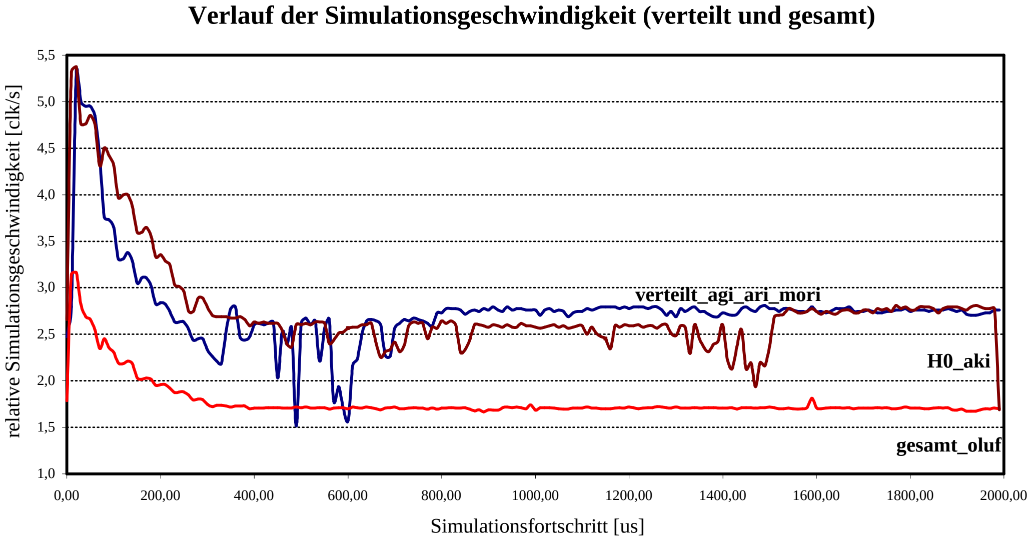


Abbildung 6.7: Geschwindigkeitsverlauf der verteilten Simulation

In Abbildung 6.7 ist nun zusätzlich der Verlauf der auf die drei Workstations aufgeteilten Simulation aufgetragen (**verteilt_agi_ari_mori**). Man sieht dabei, dass der ursprüngliche Verlauf qualitativ gleich geblieben ist, und im wesentlichen dem Zeitverlauf der MBDH entspricht.

*Absolut betrachtet ergibt sich für die verteilte Simulation in dieser Konstellation ein Performance-Zuwachs von **153 Prozent.***

Es zeigt sich somit recht deutlich, dass die Gesamtperformance fast nur durch die langsamste Baugruppe bestimmt wird. Eine Schräglast beeinträchtigt somit den Geschwindigkeitsgewinn massiv. Umgekehrt kann man daraus aber schließen, dass bei symmetrischer Belastung bei n Partitionen der Performancefaktor gegen n streben sollte. Eine relativ einfache Möglichkeit zur Untermauerung dieser These ist Folgende: Der Grund für die langen Simulationszeiten der MBDH liegt darin dass sich auf dieser insgesamt acht ASICs befinden, während die anderen beiden mit nur jeweils einem ASIC bestückt sind. Es ist nun möglich auf der MBDH die ASICs 1 bis 7, die für den betrachteten Testfall nicht benötigt werden, zu deaktivieren. Dazu müssen die ENABLE-Signale der betreffenden ASICs auf 0 gesetzt werden. Dies kann durch geeignete FORCE Anweisungen im Initialisierungs-Script erfolgen.

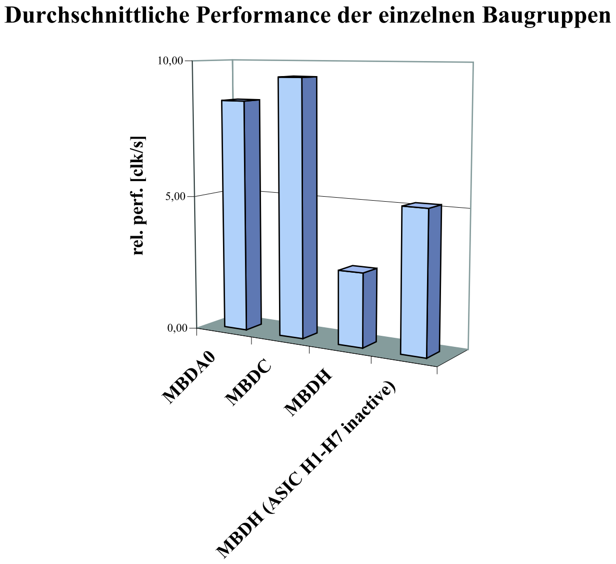


Abbildung 6.8: Geschwindigkeitsgewinn der verteilten Simulation

Im Balkendiagramm in Abbildung 6.8 werden nun die Baugruppen MBDC, MBDA und MBDH mit der MBDH mit deaktivierten ASICS hinsichtlich ihrer durchschnittlichen Simulationsgeschwindigkeit verglichen. Werden nun auch in der verteilten Simulation die betreffenden ASICs auf der MBDH deaktiviert, erhält man das in Bild 6.9 gezeigte Ergebnis.

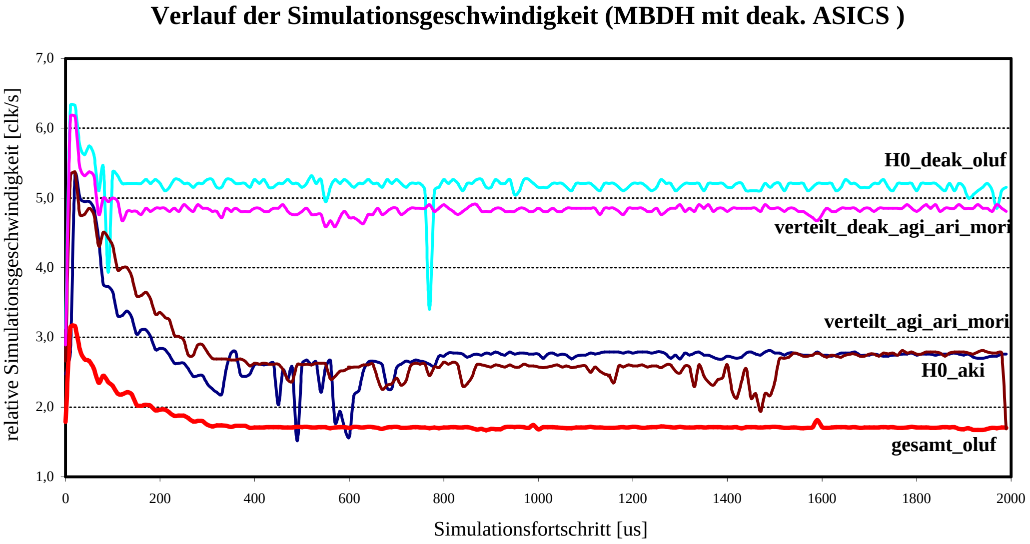


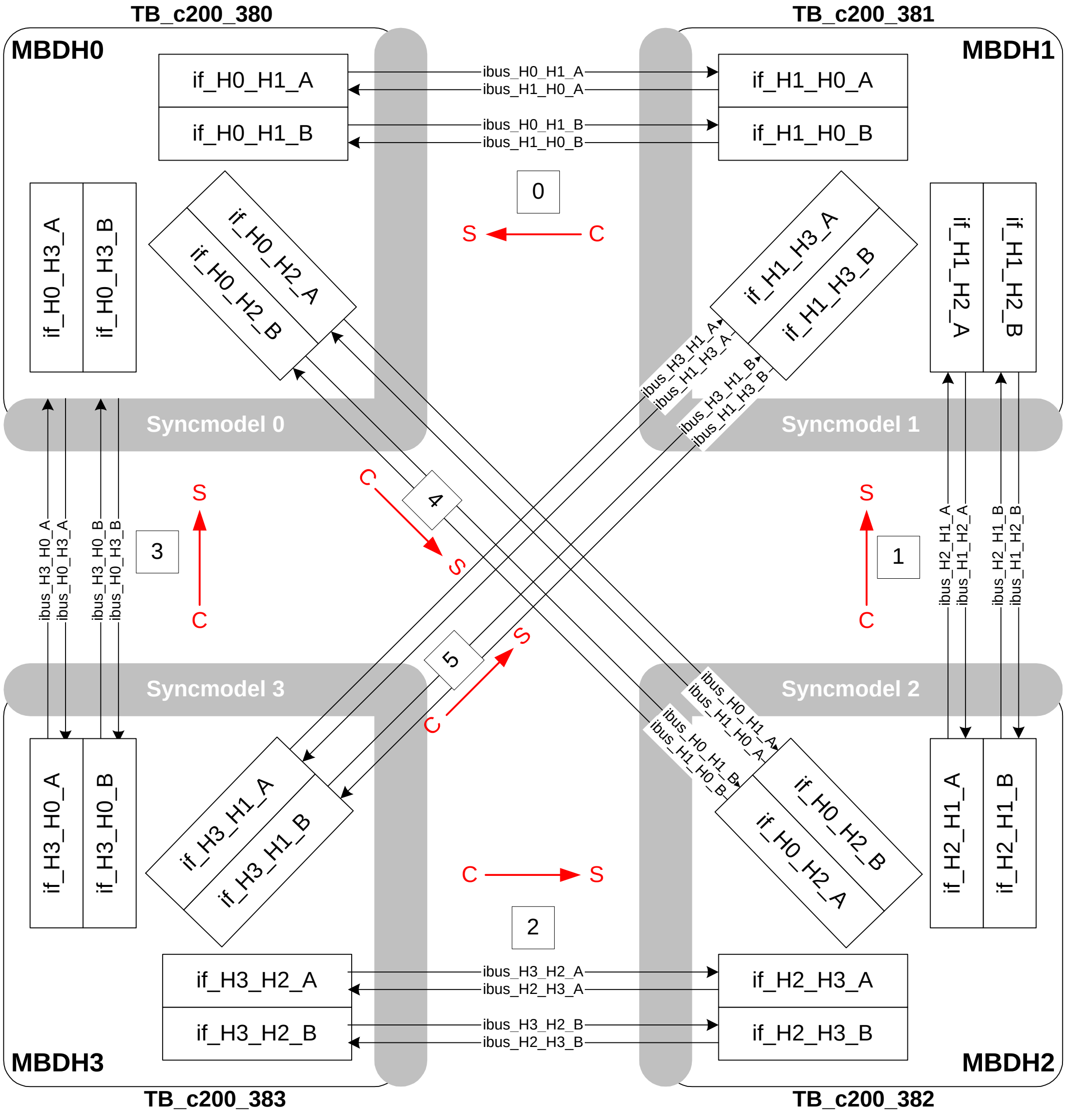
Abbildung 6.9: Geschwindigkeitsgewinn mit geringerer Schräglast

Darin erkennt man, dass sich der Verlauf der verteilten Simulation (**verteilt_deak_agi_mori_ari**) nun an den Simulationsverlauf der MBDH mit deaktivierten ASICs annähert (**H0_deak_oluf**), da nun diese das langsamste Teilsystem darstellt. *Eine neue Messung zeigte einen Geschwindigkeitsgewinn von **211 Prozent!*** Das erzielte Ergebnis ist zwar deutlich besser aber der Einfluss der noch immer vorhandenen Schräglast ist deutlich zu erkennen.

6.2.3 Aufteilung der MBD-Testbench mit vier gleichen Baugruppen und einer MBDCG

Aufgrund der Betrachtungen im vorigen Abschnitt soll nun die Aufspaltung eines Entwurfes mit vier identischen Baugruppen in Angriff genommen werden. Zu diesem Zweck wurde eine Konfiguration mit vier MBDH und einer MBDCG Baugruppe erstellt. Da für diese Konfiguration kein eigener Testfall zur Verfügung stand, wurde folgender Weg eingeschlagen:

Es wurden in jeder Baugruppen-Shell sechs weitere IBUS-Interfaces und ein Synchronisationsmodell integriert. Über die Synchronisationsmodelle können somit alle Baugruppen miteinander kommunizieren. In Abbildung 6.10 ist nun die vorgestellte Konfiguration dargestellt.



```
connections connection[]={
  /*serverID,clientID,port,in,out*/
  /*0*/{0,1,58066,2,2},
  /*1*/{1,2,64460,2,2},
  /*2*/{2,3,64461,2,2},
  /*3*/{0,3,70000,2,2},
  /*4*/{2,0,57944,2,2},
  /*5*/{1,3,64470,2,2},
       {0,0,0,0,0}  //don't remove!
};
```

Abbildung 6.10: Aufspaltung einer Simulation mit 4 MBDH-Baugruppen

Darin ist zu erkennen, dass nun über jedes Synchronisationsmodell bereits zwölf Signale über drei logische Kanäle laufen. Zusätzlich wurde wieder ein eigener Taktgenerator eingebunden, welcher aber der Übersicht wegen im Bild nicht eingezeichnet ist. In der Abbildung ist ebenfalls die Konfigurationstabelle für diese Konstellation direkt als C-Quellcode zu erkennen. Die IBUS-Interfaces wurden mit den bereits in Abschnitt 6.1.3 beschriebenen Steuerdateien betrieben. Wie im vorigen Beispiel wurde das beschriebene System auf vier gleichwertigen SUN Solaris Workstations betrieben und der Simulationsverlauf aufgezeichnet. Wie zu erwarten war, lieferten auch die IBUS-Tracer wieder korrekte Trace-Dateien.

6.2.3.1 Ergebnisse

Nach Auswertung der Messdaten zeigte sich, dass tatsächlich ein bedeutender Geschwindigkeitszuwachs zu verzeichnen war. (s. Bild 6.11)

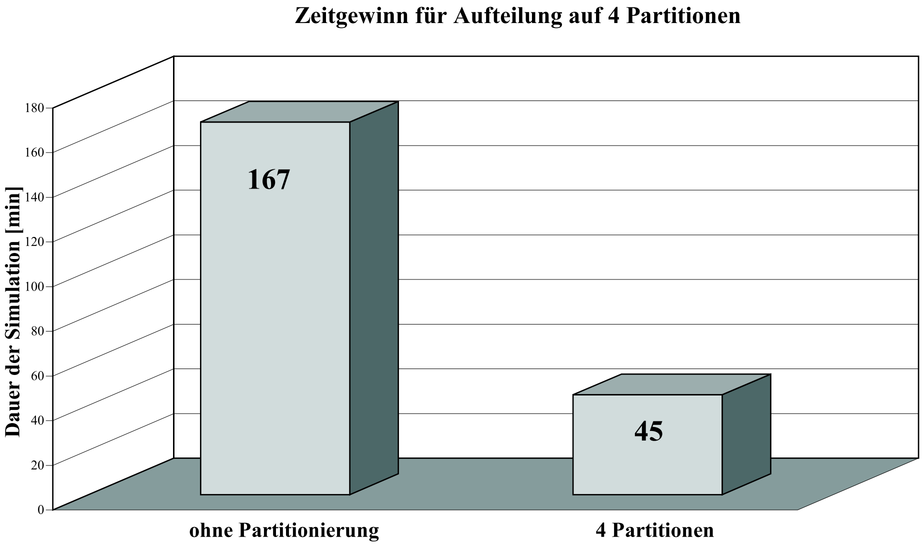


Abbildung 6.11: Absoluter Zeitgewinn bei vierfacher Partitionierung

*Für die beschriebene Konstellation konnte die Simulationsgeschwindigkeit um **380 Prozent** gesteigert werden!*

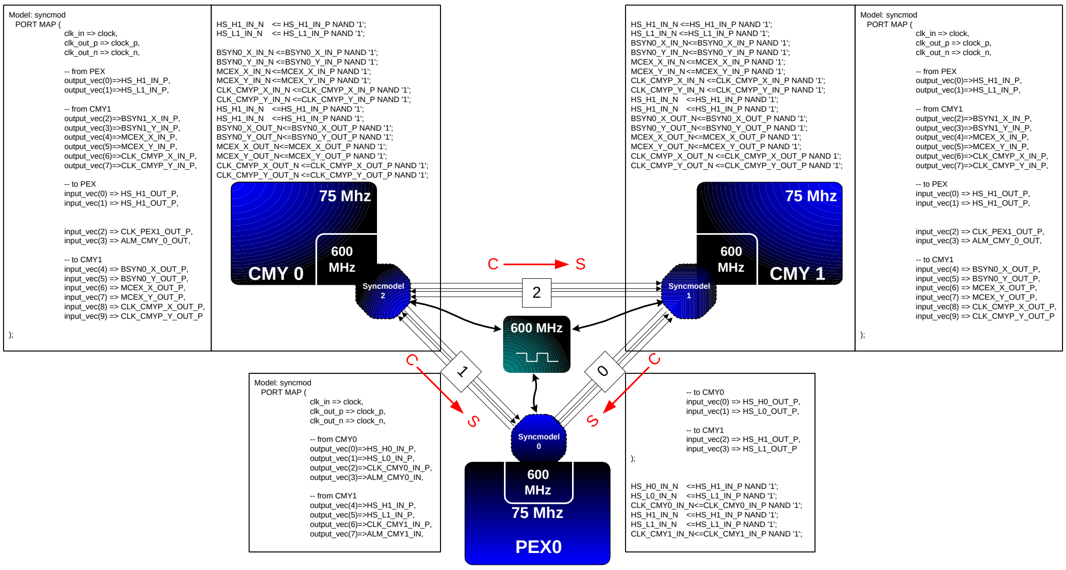
Es konnte somit auch gezeigt werden, dass die Simulationszeit fast nur von der gleichmäßigen Verteilung der Simulationslast abhängt und der, durch die Kommunikation der Modelle untereinander, verursachte Overhead kaum ins Gewicht fällt.

6.3 Integration in die CP113E-Testbench

Im folgenden Abschnitt wird die Aufteilung der CP-Testbench beschrieben.

6.3.1 Aufteilung der CP113E-Testbench

Für die weiteren Betrachtungen wurde die CP-Testbench mit den Baugruppen CMY0, CMY1 und PEX0 herangezogen. Die entstandene Aufteilung ist in Bild 6.12 dargestellt. Bemerkenswert ist dabei, dass die Synchronisationsmodelle vom zentralen Taktgenerator mit einem Takt von 600 MHz betrieben werden, und die Baugruppen über eigene 75 MHz Taktgeneratoren verfügen. Das Taktsignal vom zentralen Taktgenerator dient nur der Abtastung der Signale auf den HSSL[2]-Links und wird nicht auf die restliche Baugruppen übertragen. Weiters ist zu erkennen, dass nicht alle HSSL-Signale über das Synchronisationsmodell geleitet wurden. Um die Anzahl der zu übertragenden Signale gering zu halten, wurde bei den differtentiellen HSSL-Leitungen immer nur das positive Signal übertagen und das negative dann daraus durch Invertieren gewonnen.



```
Model: syncmod
   PORT MAP (
                  clk_in => clock,
                  clk_out_p => clock_p,
                  clk_out_n => clock_n,

                  -- from PEX
                  output_vec(0)=>HS_H1_IN_P,
                  output_vec(1)=>HS_L1_IN_P,

                  -- from CMY1
                  output_vec(2)=>BSYN1_X_IN_P,
                  output_vec(3)=>BSYN1_Y_IN_P,
                  output_vec(4)=>MCEX_X_IN_P,
                  output_vec(5)=>MCEX_Y_IN_P,
                  output_vec(6)=>CLK_CMYP_X_IN_P,
                  output_vec(7)=>CLK_CMYP_Y_IN_P,

                  -- to PEX
                  input_vec(0) => HS_H1_OUT_P,
                  input_vec(1) => HS_H1_OUT_P,


                  input_vec(2) => CLK_PEX1_OUT_P,
                  input_vec(3) => ALM_CMY_0_OUT,

                  -- to CMY1
                  input_vec(4) => BSYN0_X_OUT_P,
                  input_vec(5) => BSYN0_Y_OUT_P,
                  input_vec(6) => MCEX_X_OUT_P,
                  input_vec(7) => MCEX_Y_OUT_P,
                  input_vec(8) => CLK_CMYP_X_OUT_P,
                  input_vec(9) => CLK_CMYP_Y_OUT_P

);
```

```
HS_H1_IN_N    <= HS_H1_IN_P NAND '1';
HS_L1_IN_N    <= HS_L1_IN_P NAND '1';

BSYN0_X_IN_N <=BSYN0_X_IN_P NAND '1';
BSYN0_Y_IN_N <=BSYN0_Y_IN_P NAND '1';
MCEX_X_IN_N<=MCEX_X_IN_P NAND '1';
MCEX_Y_IN_N<=MCEX_Y_IN_P NAND '1';
CLK_CMYP_X_IN_N <=CLK_CMYP_X_IN_P NAND '1';
CLK_CMYP_Y_IN_N <=CLK_CMYP_Y_IN_P NAND '1';
HS_H1_IN_N    <=HS_H1_IN_P NAND '1';
HS_H1_IN_N    <=HS_H1_IN_P NAND '1';
BSYN0_X_OUT_N<=BSYN0_X_OUT_P NAND '1';
BSYN0_Y_OUT_N<=BSYN0_Y_OUT_P NAND '1';
MCEX_X_OUT_N<=MCEX_X_OUT_P NAND '1';
MCEX_Y_OUT_N<=MCEX_Y_OUT_P NAND '1';
CLK_CMYP_X_OUT_N <=CLK_CMYP_X_OUT_P NAND '1';
CLK_CMYP_Y_OUT_N <=CLK_CMYP_Y_OUT_P NAND '1';
```

```
HS_H1_IN_N <=HS_H1_IN_P NAND '1';
HS_L1_IN_N <=HS_L1_IN_P NAND '1';
BSYN0_X_IN_N<=BSYN0_X_IN_P NAND '1';
BSYN0_Y_IN_N<=BSYN0_Y_IN_P NAND '1';
MCEX_X_IN_N<=MCEX_X_IN_P NAND '1';
MCEX_Y_IN_N<=MCEX_Y_IN_P NAND '1';
CLK_CMYP_X_IN_N <=CLK_CMYP_X_IN_P NAND '1';
CLK_CMYP_Y_IN_N <=CLK_CMYP_Y_IN_P NAND '1';
HS_H1_IN_N    <=HS_H1_IN_P NAND '1';
HS_H1_IN_N    <=HS_H1_IN_P NAND '1';
BSYN0_X_OUT_N<=BSYN0_X_OUT_P NAND '1';
BSYN0_Y_OUT_N<=BSYN0_Y_OUT_P NAND '1';
MCEX_X_OUT_N<=MCEX_X_OUT_P NAND '1';
MCEX_Y_OUT_N<=MCEX_Y_OUT_P NAND '1';
CLK_CMYP_X_OUT_N <=CLK_CMYP_X_OUT_P NAND 1';
CLK_CMYP_Y_OUT_N <=CLK_CMYP_Y_OUT_P NAND '1';
```

```
Model: syncmod
   PORT MAP (
                  clk_in => clock,
                  clk_out_p => clock_p,
                  clk_out_n => clock_n,

                  -- from PEX
                  output_vec(0)=>HS_H1_IN_P,
                  output_vec(1)=>HS_L1_IN_P,

                  -- from CMY1
                  output_vec(2)=>BSYN1_X_IN_P,
                  output_vec(3)=>BSYN1_Y_IN_P,
                  output_vec(4)=>MCEX_X_IN_P,
                  output_vec(5)=>MCEX_Y_IN_P,
                  output_vec(6)=>CLK_CMYP_X_IN_P,
                  output_vec(7)=>CLK_CMYP_Y_IN_P,

                  -- to PEX
                  input_vec(0) => HS_H1_OUT_P,
                  input_vec(1) => HS_H1_OUT_P,


                  input_vec(2) => CLK_PEX1_OUT_P,
                  input_vec(3) => ALM_CMY_0_OUT,

                  -- to CMY1
                  input_vec(4) => BSYN0_X_OUT_P,
                  input_vec(5) => BSYN0_Y_OUT_P,
                  input_vec(6) => MCEX_X_OUT_P,
                  input_vec(7) => MCEX_Y_OUT_P,
                  input_vec(8) => CLK_CMYP_X_OUT_P,
                  input_vec(9) => CLK_CMYP_Y_OUT_P

);
```

```
Model: syncmod
   PORT MAP (
                  clk_in => clock,
                  clk_out_p => clock_p,
                  clk_out_n => clock_n,

                  -- from CMY0
                  output_vec(0)=>HS_H0_IN_P,
                  output_vec(1)=>HS_L0_IN_P,
                  output_vec(2)=>CLK_CMY0_IN_P,
                  output_vec(3)=>ALM_CMY0_IN,

                  -- from CMY1
                  output_vec(4)=>HS_H1_IN_P,
                  output_vec(5)=>HS_L1_IN_P,
                  output_vec(6)=>CLK_CMY1_IN_P,
                  output_vec(7)=>ALM_CMY1_IN,
```

```
                  -- to CMY0
                  input_vec(0) => HS_H0_OUT_P,
                  input_vec(1) => HS_L0_OUT_P,

                  -- to CMY1
                  input_vec(2) => HS_H1_OUT_P,
                  input_vec(3) => HS_L1_OUT_P
);


HS_H0_IN_N    <=HS_H1_IN_P NAND '1';
HS_L0_IN_N    <=HS_L1_IN_P NAND '1';
CLK_CMY0_IN_N<=CLK_CMY0_IN_P NAND '1';
HS_H1_IN_N    <=HS_H1_IN_P NAND '1';
HS_L1_IN_N    <=HS_L1_IN_P NAND '1';
CLK_CMY1_IN_N<=CLK_CMY1_IN_P NAND '1';
```

Abbildung 6.12: Aufteilung der CP-Testbench

---

[2] High Speed Serial Links

6.3.2 Ergebnisse

Leider zeigte sich im Zuge der Untersuchungen am aufgeteilten CP-Modell sehr bald, dass in diesem Falle durch die Parallelisierung kein Geschwindigkeitsgewinn zu erzielen war. Der Grund dafür ist in der CP-Architektur zu suchen. Innerhalb des CP113E werden die einzelnen Baugruppen (CMY, CMY und PEX) über HSSL-Links verbunden. Über diese *High Speed Serial Links* werden Signale mit einer Taktfrequenz von 600 MHz übertragen. Die Prozessoren auf den Baugruppen selbst arbeiten hingegen nur mit einer Frequenz von 75 MHz. Dies bedeutet, dass innerhalb jedes Prozessortaktes acht mal Signale zwischen den Baugruppen ausgetauscht werden müssen. Für die Parallelisierung hat diese Tatsache fatale Auswirkungen. Es tritt nämlich leider der Fall ein, dass die einzelnen Simulatoren nicht mit voller Geschwindigkeit arbeiten können, da sie sich die meiste Zeit im Wartezustand befinden. Für jeden Taktzyklus den eine Baugruppe abarbeitet, müssen auf jedem *LC* acht Signalabgleiche ausgeführt werden.

Im konkreten Fall liefen die einzelnen Workstations nur mit einer Auslastung von knapp 20 Prozent. Mit derartig geringer Rechenleistung konnte natürlich keine Beschleunigung der Gesamtsimulation erreicht werden. Wegen der ursprünglich schon sehr langen Simulationszeiten und da nun schon klar war, dass in diesem Falle keine Beschleunigung zu erzielen war, wurde auf Aufzeichnungen des Verlaufes der Simulationsgeschwindigkeit verzichtet.

Der CP-Testfall lieferte aber die wertvolle Erkenntnis, dass der erfolgreiche Einsatz des Synchronisationsmodelle, sehr stark von der Eignung der zugrundeliegenden Architektur abhängt. *Keinesfalls darf die Signalübertragung über die LCs mit höherer Frequenz als der Baugruppengrundfrequenz erfolgen.*

Kapitel 7

Schlussfolgerungen

Im Rahmen dieser Diplomarbeit konnte erreicht werden, die existierende Systemtestbench des *MBD* [1] auf mehrere Workstations aufzuteilen und unter Verwendung eines ***konservativen** Verfahrens* erfolgreich parallel zu simulieren. Dadurch konnte ein beachtlicher Geschwindigkeitsgewinn erzielt werden. Für die Simulationen wurde der Simulator *ModelSIM 5.4* mit seinem *FLI-Interface* erfolgreich eingesetzt. Bevor die eigentliche MBD Simulation in Angriff genommen wurde, wurde die erste Implementierung der Simulationsumgebung mit Hilfe des *IBUS* Modells getestet und weiter verbessert.

Die nachfolgenden Ziele konnten im Projekt leider nicht vollständig verwirklicht werden.

1. Es wurde keine eigenständige Simulationsumgebung entwickelt, sondern vielmehr ein *"intelligentes" C Modell*, welches händisch in die VHDL Testbench eingebunden wird.

2. Die gefundene Lösung ist nicht vollkommen herstellerunabhängig, sondern auf *Modelsim* zugeschnitten. Da aber in absehbarer Zukunft zu erwarten ist, dass die *VHDL zu C* Schnittstelle durch einen Standard spezifiziert wird, wird es ab diesem Zeitpunkt möglich sein, Herstellerunabhängigkeit zu erreichen.

3. Die Parallisierung der CP113E [2] Testbench konnte architekturbedingt leider nicht erfolgreich durchgeführt werden. Diese lieferte aber wertvolle Erkenntnisse für zukünftige Projekte.

Weiters wurden im Betrieb folgende Probleme erkannt

---

[1] Message Buffer Typ D

[2] Coordination Processor 113E

1. Die Gesamt-Performance ist extrem abhängig von der Auslastung der einzelnen Workstations. Für optimale Performance muss daher den Simulatoren vom Betriebssystem höchste Priorität zugeordnet werden.
2. Die Integration in eine bestehende Testbench erfordert vom Anwender eine genaue Kenntnis der Architektur, da alle erforderlichen Verbindungssignale identifiziert werden müssen, bevor das Synchronisationsmodell dann händisch in die Testbenches integriert werden kann.
3. Da der Start der einzelnen Simulationen per Hand erfolgen muss, ist dieser sehr zeitaufwendig, da die Simulatoren der Reihe nach ihr gesamtes Design laden müssen.

Dadurch konnten folgende am Beginn aufgestellen Thesen bestätigt werden:

1. Der Geschwindigkeitsgewinn wird fast nur durch das Verhältnis der Komplexität der Partitionen bestimmt.
2. Die Geschwindigkeitseinbußen durch den Netzwerkverkehr gehen mit wachsender Komplexität des Gesamtsystems zurück, soferne die Übertragung nicht mit höherem Takt als der Baugruppengrundtakt erfolgt.
3. Die Anzahl der Verbindungssignale beeinflusst die Performance beträchtlich und soll möglichst gering gehalten werden.

Als konkretes Ergebnis wurde bei einer Konfiguration mit drei unterschiedlichen Boards *(MBDA, MBDC, MBDH)* ein *Geschwindigkeitszuwachs um den Faktor* ***2.11*** erreicht. Eine Konfiguration mit vier gleichen Baugruppen *(MBDH)* brachte sogar eine *Beschleunigung um den Faktor* ***3.8***.

Diese Zahlen beweisen, dass bei geeigneten Architekturen ein beachtliches Beschleunigungspotential vorhanden ist, welches mit der gezeigten Lösung auf relativ einfache Weise ausgeschöpft werden kann.

Die gefundene Methode stellt sicher eine praktikable Möglichkeit dar, in zukünftigen Projekten die Laufzeit von Einzelsimulationen deutlich zu verkürzen.

Literaturverzeichnis

[1] Request for comment: http://www.rfc-editor.org/rfc.html (2000).

[2] R. E. Bryan. Simulation of Packet Communications Architecture Computer System. Technical report, Massechusetts Institute of Technology, 1977.

[3] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study and Verification of Distributed Programms. *IEEE Transactions on Software Engineering*, 1979.

[4] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 1981.

[5] Moon Jung Chung, Jinsheng Xu, and Hee Chul Kim. Parallel VHDL Simulation Engine. Department of Computer Science, Michigan State University.

[6] Alois Ferscha. Parallel and Distributed Simulation of Discrete Event Systems. *Institut für Angewandte Informatik, University of Vienna*, 1995.

[7] R. M. Fujimoto. Performance Measurements of Distributed Simulation Strategie. *Transactions of the Society for Computer Simulation*, 1989.

[8] R. M. Fujimoto. Parallel Discret Event Simulation. *Communications of the ACM*, 33:30–53, October 1990.

[9] Peter R. Gerke. *Digitale Kommunikationsnetze - Prinzipien, Einrichtungen, Systeme.* Springer Verlag, 1991. ISBN 3-540-52330-8.

[10] John L. Hennessy and David A. Petterson. *Rechnerarchitektur, Analyse, Entwurf, Implementierung, Bewertung.* Vieweg, 1994.

[11] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 1985.

[12] D. R. Jefferson and H. Sowizral. Fast Concurrent Simulation using the Time Warp Mechanism, Part I: Local Control. Technical report, RAND Corporation, 1982.

[13] Oliver Krause. Konzeptvalidierung mit Verhaltenssimulation. Diplomarbeit, Fachhochschule München, Fachbereich 04, 1997.

[14] Manfred Linder. Datenkommunikation und Netzwerke. Skriptum zur Vorlesung VO 384.560 an der Technischen Universität Wien, 1999.

[15] LSI Logic. *G10-p Cell-Based ASIC, Product Design Manual, Databook*, October 1996.

[16] Gerd Meister. A Survey on Parallel Logic Simulation, September 1993. Department of Computer Science, University of Saarland.

[17] Siemens. EWSD - die generische Plattform für alle Applikationen. Technische Systembeschreibung.

[18] Siemens. Simulationsablaufspezifikation für das System MBD (Message Buffer Typ D), 1997. Testspezifikation P30303-A7893-T001-03-0024.

[19] Franz Josef Simmet. *Entwurf und Implementierung eines verteilten VHDL-Simulators.* PhD thesis, Universität Saarbrücken, Fachbereich Informatik, April 1994. Teil 1.

[20] Steffen Straßburger, Thomas Schulze, Ulrich Klein, and James O. Henriksen. Internet-Based Simulation Using Off-The-Shelf Simulation Tools and HLA. *Proceedings of the 1998 Winter Simulation Conference*, 1998.

[21] Model Technology. *Modelsim EE/PLUS Reference Manual, Version 5.4b*, 2000.

[22] V. Yarmolik and I. Kachan. *Self-Testing VLSI-Designs.* Elsevier Science Publischers,Amsterdam, 1993.

Abbildungsverzeichnis