Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik

Datenflussoptimierung in rekonfigurierbarer Hardware durch Pipelining

Studienarbeit

Studiengang Informatik

von

Christian Behler
Lützer Weg 18
34439 Willebadessen-Niesen

betreut durch

Dipl.-Inform. Stefan Ihmor

vorgelegt bei

Prof. Dr. rer. nat. Franz Josef Rammig

im

Mai 2005

Dank und Erklärung

Diese Studienarbeit entstand in der Arbeitsgruppe von Prof. Dr. rer. nat. Franz Josef Rammig (HNI) der Universität Paderborn.

Ich bedanke mich bei Prof. Rammig und vor allem bei Stefan Ihmor, der mich in allen Phasen engagiert unterstützt und betreut hat. Außerdem danke ich Marion Pauly sowie Britta und Markus Göner für das Korrekturlesen.

Hiermit erkläre ich, dass die vorliegende Arbeit selbstständig von mir verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht wurden.

Niesen, im Mai 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

1.1 Motivation

Kommunikation ist in vielen Bereichen ein wichtiger Faktor. Neu entwickelte und bereits bestehende Komponenten müssen in der Lage sein, Informationen untereinander auszutauschen. Dazu ist es wichtig, dass Komponenten, die miteinander kommunizieren, die gleiche Kommunikationsstruktur verwenden, damit die Informationen korrekt interpretiert und verarbeitet werden können.

Im Bereich der Informationstechnik wird häufig auf bereits bestehende Komponenten zurückgegriffen, um neue Funktionalitäten zu implementieren. Somit ist es oft schwierig, eine gemeinsame Kommunikationstruktur zu finden, so dass die einzelnen Teile des Gesamtsystems miteinander kommunizieren können. Um dieses Problem zu umgehen, gibt es zwei mögliche Ansätze [7]:

- Die verwendeten Komponenten benutzen die gleiche Kommunikationsstruktur und eine Anpassung ist nicht nötig. Dadurch wird das System unflexibel, da nicht beliebige Komponenten eingebunden werden können.

- Jede Komponente muss ein Interface bieten, dass mit einem standardisierten Bussystem kommunizieren kann. Diese Standardisierung kann aber dazu führen, dass Komponenten, die von der Funktionalität her recht klein sind, ein recht großes Interface implementieren müssen, um in das vorhandene System integriert werden zu können. Somit kann ein unverhältnismäßig hoher Aufwand entstehen.

Betrachtet man diese Problematik auf der Ebene der Eingebetteten Systeme, so existiert dort ein weiterer Lösungsansatz. Durch die sogenannte *Interface Synthesis* (IFS), einer Methodik für die Modellierung und automatisierte Synthese von Schnittstellen, kann ein Modul generiert werden, welches zwei Kommunikationspartner, zwischen denen eine direkte Kommunikation nicht möglich ist, verbindet. Dieses Modul wird *Interface Block*(IFB) genannt und fungiert als "Übersetzer" zwischen den inkompatiblen Komponenten. So wird der Austausch von Informationen ermöglicht, ohne dass die Kommunikationspartner modifiziert werden müssten.

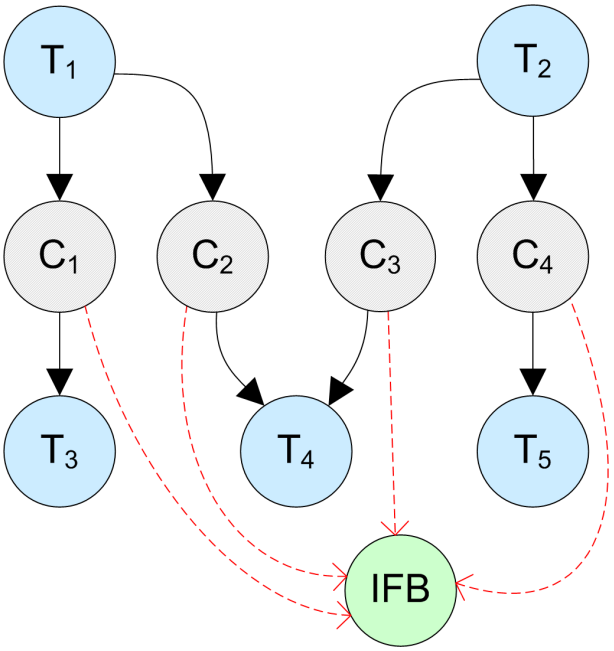

Abbildung 1.1: Kommunikationsgraph

Abbildung 1.1 zeigt einen möglichen Einsatz für den IFB. Es existieren fünf Kommunikationspartner (hier als Task bezeichnet), zwischen denen eine direkte Kommunikation nicht möglich ist. Task_1 will mit Task_3 und Task_4 kommunizieren, Task_2 will Daten an Task_4 und Task_5 senden. Zwischen jedes Kommunikationspaar beschreibt eine Abbildungsvorschrift (hier als Kommunikationsknoten $C_{1..4}$ bezeichnet) wie die gelesenen Daten zu transformieren sind, so dass der Empfänger diese korrekt interpretieren kann. Eine Abbildungsvorschrift wird auf einen IFB abgebildet, d.h. sie wird dort implementiert. In Kapitel 2 wird der genaue Aufbau des IFBs und dessen Funktionalität erklärt. Es wird auch darauf eingegangen, welche Hardware verwendet werden kann, um die Funktionalität des IFBs zu realisieren.

Die Transformation der Daten im IFB verursacht eine *Latenz*, da das Senden und Empfangen der Daten sowie die Verarbeitung (also die "Übersetzung") eine gewisse Zeit benötigt. Latenz bezeichnet die Zeit, die vom Senden der Daten bis zum Empfangen der ersten Information vergeht. Daten, die zuerst von einem IFB verarbeitet werden, kommen also zu einem späteren Zeitpunkt bei den empfangenden Tasks, an als wenn die Daten direkt gesendet würden.

Ein weiterer Aspekt bei der Betrachtung des IFBs ist der physikalische Platz auf dem Chip, der benötigt wird, um die Komponenten des IFBs zu implementieren. Je größer ein IFB wird, desto mehr Platz benötigt dieser. Große Chips sind allerdings mit hohen Kosten verbunden. Es ist ein allgemeines Ziel, diese Kosten so gering wie möglich zu halten.

1.2 Aufgabenstellung

Ausgehend von der existierenden IFB-Technologie sollen im Rahmen dieser Studienarbeit zwei Verfahren zur Optimierung der Aspekte Latenz und Fläche untersucht werden.

Datenflussoptimierung In Kapitel 3 wird die Latenz durch Datenflussoptimierung reduziert. Dazu wird versucht, die maximal mögliche Anzahl an Eingabe-, Verarbeitungs- und Ausgabeprozessen parallel laufen zu lassen, damit die empfangenen Daten früher verarbeitet und zum Empfänger gesendet werden können. Um dies zu erreichen, werden die eingehenden Datenpakete betrachtet. Die tatsächliche Größe, die der IFB auf dem Chip, auf dem er implementiert ist, benötigt, wird nicht berücksichtigt.

Flächenoptimierung Der zweite Ansatz beschäftigt sich mit der optimalen Ausnutzung der Fläche auf dem Chip, auf dem der IFB implementiert ist. Die Grundlage ist ein Chip, der zu klein ist, um alle gewünschten Funktionalitäten gleichzeitig unterzubringen und es somit erforderlich macht, Funktionalitäten zur Laufzeit zu laden. Der Ansatz dazu wird in Kapitel 4 vorgestellt.

Diese beiden Ansätze sind *nicht* gleichzeitig realisierbar und zwar aus folgendem Grund: Der erste Ansatz berücksichtigt keine Restriktionen bezüglich der Größe des IFBs (und somit des Chips). Es wird davon ausgegangen, dass die vollständige IFB-Funktionalität zu jeder Zeit vorhanden ist. Der zweite Optimierungsansatz verletzt diese Restriktion: Die Flächenoptimierung setzt voraus, dass *nicht* die komplette Funktionalitäten zu jeder Zeit vorhanden sein muss. Das führt dazu, dass ungenutzte Funktionalitäten verdrängt und durch solche Funktionalitäten ersetzt werden müssen, die gerade benötigt werden. Eine Optimierung des Datenflusses, wie sie in Kapitel 3 beschrieben wird, ist dabei nicht möglich, da die Optimierung das Verdrängen und Laden von Funktionalitäten nicht berücksichtigt. Beide Verfahren werden anhand von ausgewählten Beispielen bewertet. Abschließend wird in Kapitel 5 eine kurze Zusammenfassung gegeben.

2 Grundlagen

In diesem Kapitel werden grundlegende Begriffe erklärt sowie eine Einführung in die verwendete Hardware und Software gegeben. Dabei wird insbesondere auf Field Programmable Gate Arrays (siehe Kapitel 2.1) als eine mögliche Zielplattform für IFBs eingegangen. Die partielle Rekonfigurierbarkeit dieser Hardwareplattform ist die Grundlage für die Flächenoptimierung, die in Kapitel 4 vorgestellt wird.

2.1 Das FPGA

Ein *Field Programmable Gate Array* (FPGA) ist ein programmierbarer Chip, der eine geeignete Ausführungsplattform für den IFB darstellt. Der Chip kann rekonfiguriert werden, d.h. Funktionalitäten können zu Laufzeit ausgetauscht werden. Somit kann das FPGA auch als Plattform für die rekonfigurierbare Version des IFBs [4] genutzt werden. Im Folgenden wird ein kurzer Einblick in die Funktionalität von FPGAs gegeben. Eine detailliertere Darstellung findet sich in Kapitel 4.

Ein FPGA besteht aus rekonfigurierbaren Einheiten, sogenannten *slices*. Mehrere *slices* ergeben einen Rekonfigurierungsblock (RB) oder *Slot*. In diese Slots können die gewünschten Funktionen (Programme) geladen werden. Allerdings benötigt jeder Slot sogenannte *Bus Macros*, die ihn mit den anderen Slots verbindet. Die Trennung durch Bus Macros ist für die Rekonfigurierung notwendig und technisch bedingt.

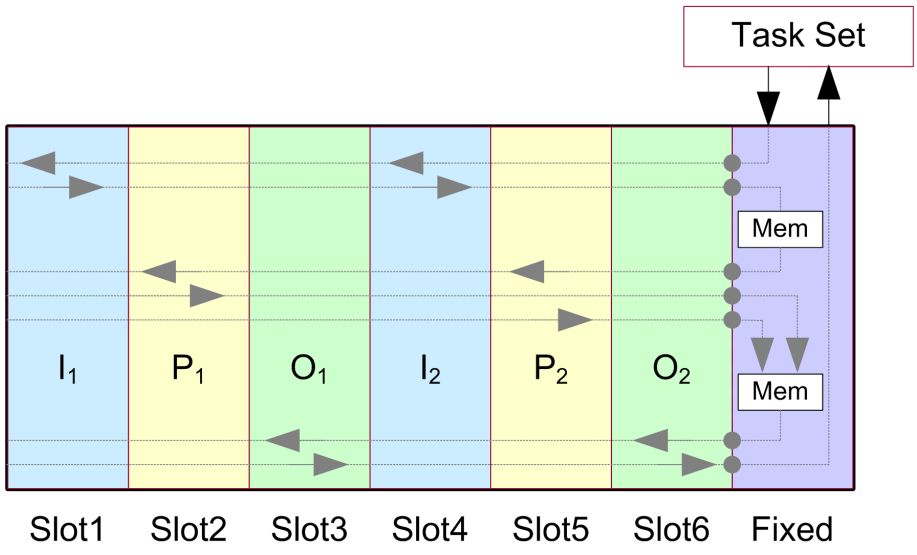

Abbildung 2.1: FPGA mit Slots

Somit ist es möglich, neue Funktionalitäten zur Laufzeit hinzuzufügen. Ist die neue Funktionalität komplett geladen, kann diese einfach hinzugeschaltet werden. Ein weiterer Vorteil ist die Möglichkeit, vorhandene Funktionalitäten zu verdrängen. Soll eine Anwendung verdrängt werden, werden die Bereiche, die von der auszutauschenden Task benutzt werden, deaktiviert. Dann wird die neue Funktionalität geladen und die Bereiche wieder aktiviert, so dass die neue Funktionalität genutzt werden kann.

In Abbildung 2.1 ist ein partiell rekonfigurierbares FPGA mit einer geladene Task abgebildet. Der rechte Slot ist eine nicht austauschbare Komponente (als 'fixed' gekennzeichnet). Über diesen Slot, der die zentralen und nicht austauschbaren Teile eines IFBs beinhaltet, wird die Verbindung zu externen Anwendungen hergestellt. Zusätzlich fungiert dieser Slot als Speicher. Dieser Speicher steht den Anwendungen in allen Slots zur Verfügung, da der Inhalt dieses Slots niemals ausgetauscht (also rekonfiguriert) wird.

Der fest installierte Slot enthält im Falle des IFB neben dem Speicher im Wesentlichen eine Kontrolleinheit. Diese wird u.a. für die Steuerung und Überwachung der Rekonfigurierung benötigt. Denn bevor eine Rekonfigurierung durchgeführt werden kann, müssen einige Bedingungen erfüllt sein. Beispielsweise muss sichergestellt sein, dass zum Zeitpunkt der Rekonfigurierung keine zu rekonfigurierende Komponente aktiv ist. Details zu der Kontrolleinheit werden in Kapitel 4 genauer erläutert.

2.2 Interface Block (IFB)

Im Folgenden wird die eigentliche Anwendung, die auf dem FPGA implementiert werden soll, näher erläutert. Dazu siehe auch [6] und [5].

Ein *Interface Block* (IFB) fungiert, wie bereits in der Einleitung beschrieben, als "Übersetzungskomponente". Existieren zwei Kommunikationspartner A und B (im Folgenden als Task bezeichnet), deren Kommunikationsstruktur (Protokolle) inkompatibel sind, so ist generell eine direkte Kommunikation zwischen diesen beiden Tasks nicht möglich. Der IFB ist in der Lage, mit beiden Komponenten zu kommunizieren und die Daten, die zwischen A und B ausgetauscht werden, so zu transformieren, dass die empfangende Task die Daten vom sendenden Task verstehen kann. Der IFB ist dabei für A und B transparent.

Wie in Abbildung 2.2 dargestellt, besteht der IFB generell aus drei Hauptkomponenten: dem *Protocol Handler*(PH), dem *Sequence Handler*(SH) und der *Control Unit*(CU). Der Aufbau und die Funktionsweise dieser Komponenten werden im Folgenden erklärt.

Der **Protocol Handler** ist die Schnittstelle zwischen den externen Tasks und den anderen Komponenten des IFB. Der PH besteht aus einzelnen PH_{Modes}, die mit jeweils der Schnittstelle einer Task verbunden sind und die Kommunikation mit dieser Task realisieren. Um die Daten korrekt zu empfangen, muss der PH_{Mode} das Protokoll der externen Task verarbeiten können. Dazu besteht der Mode aus einer *Finite State Machine* (FSM), die das komplementäre Protokoll dieser Task implementiert. Dabei werden aus dem Protokoll, das von der Task empfangen wird, Kontrollbits ausgefiltert, die einen korrekten Kommunikationsablauf sicherstellen sollen. Ein PH_{Mode} kann ausschließlich mit der Task kommunizieren, dessen komplementäres Protokoll implementiert wird. Es

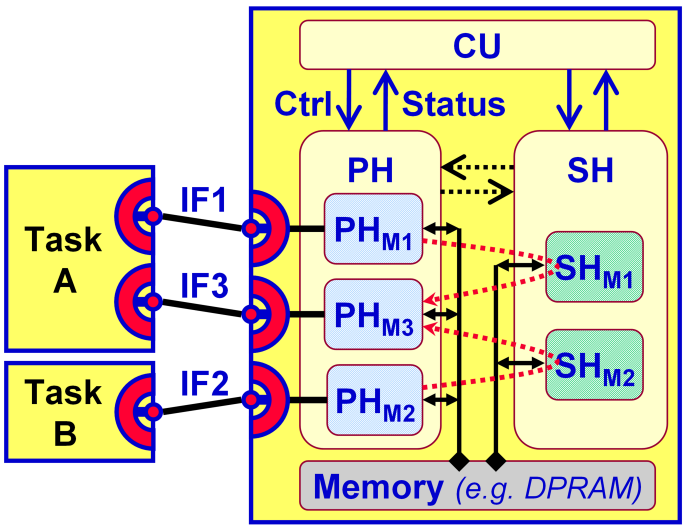

Abbildung 2.2: IFB Makrostruktur

wird also für jede Task, die mit dem IFB verbunden sind, ein PH_{Mode} benötigt. Die FSM innerhalb des Modes benötigt noch zusätzliche Zustände, um die Interaktion mit den anderen Komponenten des IFBs zu realisieren.

Der **Sequence Handler** ist die Komponente, die die "Übersetzung" der Daten vornimmt. Er besteht aus einer Anzahl von SH_{Modes} und zwei Speichern (Data Reader und Data Writer). Die SH_{Modes} bestehen, wie die PH_{Modes}, aus Automaten und modifizieren die Daten. Das sogenannte *IFD-Mapping*(Kapitel 2.4) gibt dabei an, wie die Daten aus dem Data Reader auf die Daten im Data Writer abzubilden sind. Die beiden Speicher sind über Datenbusse mit den PH_{Modes} und den SH_{Modes} verbunden und dienen als Zwischenspeicher für eingehende bzw. ausgehende Datenpakete. Die Größe des Speichers entspricht genau der Menge an Daten, die empfangen bzw. durch die Transformation generiert und nachfolgend versendet werden. Jedes Bit hat dabei eine genau festgelegte Position in dem Speicher, der speziell für dieses Szenario synthetisiert wurde. Liest ein PH_{Mode} ein Bit ein, so wird die vorhandene Information an der Stelle dieses Bits überschrieben. Die Synthese des Speichers erfolgt aufgrund der Informationen, die in dem IFD-Mapping enthalten sind.

SH und PH werden durch die **Control Unit** koordiniert. Die CU ist bspw. für die Arbitrierung der Datenbusse zuständig. Da allgemein davon ausgegangen wird, dass ein IFB mit mehreren sendenden und mehreren empfangenden Tasks verbunden ist (Multi-Task IFB), müssen die Zugriffe zeitlich gesteuert werden. Dazu existieren die $CTRL_{In}$ Komponente und die $CTRL_{Out}$ Komponente (beide im Folgenden als *Scheduler* bezeichnet). Erstere vergibt den Datenbus für eingehende Daten, letztere den Datenbus für ausgehende Daten. Somit wird sichergestellt, dass immer nur ein PH_{Mode} Daten in den Data Reader schreibt und ein PH_{Mode} Daten aus dem Data Writer liest. Des weiteren steuert die CU die Sequence Handler Modes. Erst wenn die Modes ein entsprechendes Signal von der CU erhalten, beginnen sie mit der Verarbeitung (Mapping) der Daten. Dieses Signal wird vom *Scoreboard* gesendet. Erst wenn durch das Scoreboard entsprechende

Signale gesendet wurden, kann ein Scheduler einen Datenbus freigeben oder ein SH_{Mode} mit der Transformation der Daten beginnen. In Kapitel 2.5 wird das Scoreboard detaillierter erklärt, da es eine wichtige Komponente zur Optimierung darstellt. Die letzte Komponente in der CU ist die *Reconfiguration Unit*. Sie überwacht die Rekonfigurierung und ist eine zentrale Komponente bei der Flächenoptimierung (Kapitel 4).

Die Kommunikation zwischen diesen Komponenten erfolgt über ein *fully interlocked protocol*. Die Funktionsweise der Komponenten wird im folgenden Abschnitt anhand eines Beispiels verdeutlicht

2.3 Datenfluss im IFB

Es wird von zwei Tasks A und B ausgegangen. Die Tasks besitzen inkompatible Protokolle, müssen aber Daten austauschen. Zwischen beiden Tasks wird ein IFB eingesetzt und mit den beiden Tasks verbunden. Sendet nun Task A Daten, werden sie vom IFB empfangen.

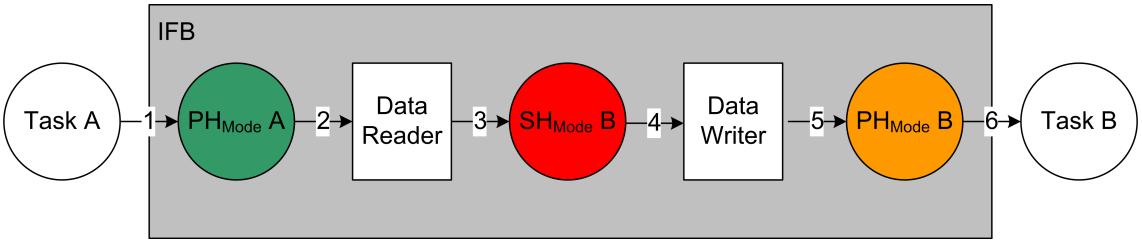

Abbildung 2.3: IFB Datenfluss

Schritt 1 Der PH_{Mode} A bewirbt sich um den Zugriff auf den Datenbus zum Data Reader. Nachdem der Scheduler den Bus zugeteilt hat, werden die Daten vom PH_{Mode} gelesen.

Schritt 2 Der PH_{Mode} leitet die Nutzdaten zum Data Reader, die dort gespeichert werden.

Schritt 3 Wurden die Daten komplett eingelesen, startet das Scoreboard den entsprechenden SH_{Mode}, der die Daten aus dem Data Reader liest und modifiziert.

Schritt 4 Die transformierten Daten werden in den Data Writer geschrieben.

Schritt 5 Hat der SH_{Mode} B seine Arbeit beendet, kann sich der PH_{Mode} B erfolgreich um den Datenbus zum Data Writer bewerben. Darf der PH_{Mode} den Datenbus belegen, liest er die Daten aus dem Data Writer, integriert diese in das ausgehende Protokoll.

Schritt 6 Darauf sendet der PH_{Mode} B die Daten an Task B.

2.4 IFD-Mapping

Das IFD-Mapping (*Interface Definition* Mapping) beinhaltet eine Menge von Regeln, die beschreiben, wie eingehende auf ausgehende Daten abgebildet werden. Eine solche Regel wird *Mapping Equation* (MapEqn) genannt und hat die Form

$$MapEqn : data_{Out} <= f_{Map}(data_{In,1}, ..., data_{In,k});$$

Ein ausgehendes Datenpaket kann also von einem oder mehreren eingehenden Datenpaketen abhängig sein. Die Umwandlung der Daten aus den Datenpaketen erfolgt über vier mögliche Grundoperationen:

- Zuweisung konstanter Werte
- Zuweisung eingehender Bits (ohne diese zu verändern)
- Anwenden von boolschen Funktionen auf eingehende Bits
- Verwenden eines endlichen Zustandsautomaten (FSM) zum Erzeugen von Bits

Bevor hier ein kurzes Beispiel zur Verdeutlichung gegeben wird, wird erklärt, wie die Daten einer Task empfangen und vorbereitend verarbeitet werden, bevor sie abgebildet werden können.

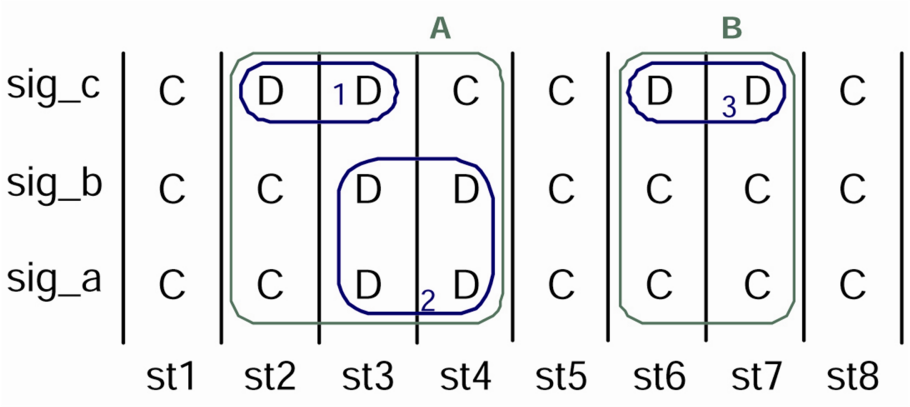

Abbildung 2.4: Ein Basic Block aus einem Protokoll mit Kontroll- und Nutzdaten

Die Daten einer Task sind in einem Protokoll (siehe Abbildung 2.4) enthalten. Dieses Protokoll wird in drei Schritten zergliedert.

Der erste Schritt analysiert das Protokoll und zerteilt es in ***Basic Blocks***. Basic Blocks sind Teile des Protokolls und bestehen aus einer Menge von Zuständen und Transitionen. Innerhalb dieses Basic Blocks darf es keine Verzweigung geben, d.h. es ist eine sequentielle Folge von Zuständen. Ausführliche Informationen zu Basic Blocks und deren Eigenschaften sind in [7] zu finden.

Abbildung 2.4 ist die Darstellung eines Basic Blocks als Matrix. Diese Matrix besteht aus Zuständen (Spalten) und den Signalen (Zeilen). In den einzelnen Zellen der Matrix sind die Bits eines Protokolls als Redundanz oder Informationen vorhanden. Die Redundanz liegt in Form von Kontrollbits (**C**) vor. Die Informationen enthalten die eigentlichen Datenbits (**D**). Um das Einlesen bzw. Abspeichern der Daten im Data Reader zu vereinfachen, werden zusammenhängende Bits (also Daten, die sich in angrenzenden Zellen befinden) zu Datenpaketen zusammengefasst. Hierbei werden zuerst die Zeilen nach zusammenhängenden Bits durchlaufen, dann erst die Spalten (deswegen ergibt sich im obigen Beispiel das Datenpaket 1 in der oberen Zeile und das Datenpaket 2 in den beiden darunterliegenden). Im nächsten Schritt werden sogenannte *Frames* gebildet. Frames enthalten mindestens ein Datenpaket und haben eine eindeutige ID. Überlappen sich zwei Datenpakete im gleichen Zustand, werden diese zu einem einzigen Frame zusammengefasst. Im obigen Beispiel überlappen sich die Datenpakete 1 und 2 und ergeben den Frame A, Datenpaket 3 hat keine Überschneidung mit anderen Datenpaketen und bildet so den Frame B. Frames kapseln die Daten, die ohne Unterbrechung zwischen PH und SH übertragen werden. Für diese Übertragung muss der entsprechende Datenbus im IFB zugeteilt werden. Im Folgenden soll in dieser Arbeit davon ausgegangen werden, dass jeder Frame immer exakt ein Datenpaket beinhalte.

Das folgende Beispiel beinhaltet drei der vier Grundoperationen des Mappings und soll die Funktionsweise des IFD-Mappings verdeutlichen:

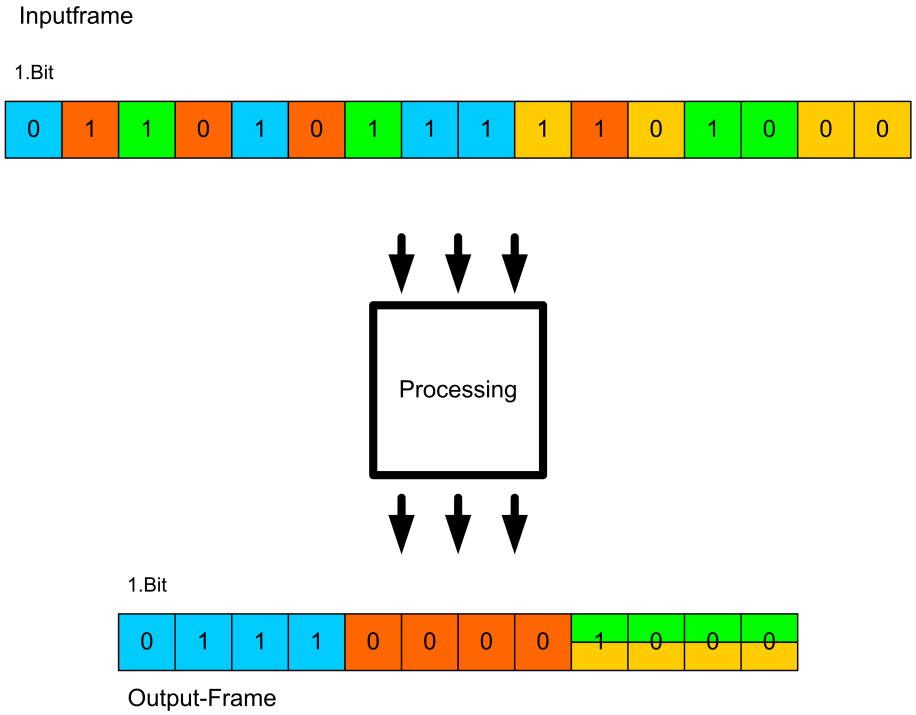

Abbildung 2.5: Funktion des IFD-Mappings

Die einzelnen Bits in Abbildung 2.5 wurden farblich markiert, um zu verdeutlichen, was während des Mappings geschieht. Die hellblau gefärbten Bits werden auf die ersten vier Bits des Output-Frames abgebildet. Der Inhalt der nächsten vier Bits wird durch eine '0' überschrieben und auf die mittleren vier Bitpositionen abgebildet. Die gelben und grünen Bits wurden bitweise '&' verknüpft und die Ergebnisse auf die letzte vier Bits abgebildet.

Die entsprechenden MappingEquations zu diesem Beispiel sehen folgendermaßen aus. Dabei steht 'P' für ein Datenpaket.

- $P_{Out}[1:1] <= P_{In}[1:1];$
- $P_{Out}[2:2] <= P_{In}[5:5];$
- $P_{Out}[3:4] <= P_{In}[8:9];$
- $P_{Out}[5:8] <= "0000";$
- $P_{Out}[9:9] <= (P_{In}[3:3]$ & $P_{In}[10:10])$
- $P_{Out}[10:10] <= (P_{In}[7:7]$ & $P_{In}[12:12])$
- $P_{Out}[11:11] <= (P_{In}[13:13]$ & $P_{In}[15:15])$
- $P_{Out}[12:12] <= (P_{In}[14:14]$ & $P_{In}[16:16])$

Aus diesen Mapping Equations kann also die Information extrahiert werden, welches Bit aus dem Input-Frame für welches Bit des Output-Frames benötigt wird. Einzelne Bits können dabei mehrfach genutzt werden.

2.5 Ausführung als Pipeline

Eine weitere grundlegende Eigenschaft des IFBs besteht darin, dass man die Arbeitsschritte aus Kapitel 2.3 in *Stufen* zusammenfassen kann. In der ersten Stufe werden Daten eingelesen, in der zweiten werden diese verarbeitet und in der dritten und letzten Stufe werden sie dann versendet. Diese einzelnen Stufen können zu einer Pipeline zusammengefasst werden und ergeben dann das so genannte *I-P-O Schema*: Input-Processing-Output. Wie bei einem Fließband gilt, dass eine Stufe erst dann begonnen werden kann, wenn die vorige Stufe beendet ist. Für die Einhaltung des I-P-O Schemas im IFB ist das *Scoreboard* zuständig. Es kann anhand von Bedingungen feststellen, ob und welche Stufe als nächstes begonnen wird. Diese Bedingungen werden im Folgenden aufgeführt und erklärt [1]:

Eingabe Diese Bedingungen beziehen sich auf die Daten, die von sendenden Tasks empfangen werden.

- Ein Frame enthält n Datenpakete.
- Ein Ausgangsdatenpaket kann nicht als Eingangsdatenpaket genutzt werden.
- Wurden alle Daten im Data Reader von einem SH_{Mode} gelesen und verarbeitet, kann ein neuer Frame gelesen werden.
- Nachdem der PH_{Mode} ein Frame vollständig eingelesen hat, wird dies dem Scoreboard gemeldet.

Verarbeitung Diese Bedingungen beziehen sich auf den Start der Verarbeitung. Diese darf beginnen, wenn alle Daten vorhanden sind.

- Sobald alle für einen Sequence Handler Mode notwendigen Datenpakete vollständig im Data Reader sind und alle entsprechenden Daten im Data Writer gesendet wurden, startet das Scoreboard den entsprechenden SH_{Mode}.

Ausgabe Hier prüft das Scoreboard, ob Daten versendet werden dürfen.

- Ein Frame kann ausgegeben werden, wenn der SH_{Mode} beendet ist und alle abgebildeten Daten im Data Writer abgespeichert wurden.

Das Scoreboard ist damit eine zentrale Komponente, die die Befolgung des I-P-O Schemas und damit die Kausalität des Datenverkehrs kontrolliert. Es wertet die Kontrollsignale der beteiligten Komponenten aus und verhindert oder lässt zu, dass ein PH_{Mode} oder SH_{Mode} mit seiner Ausführung fortfahren kann. Es existieren allerdings Szenarien, in denen das I-P-O Schema *nicht* eingehalten werden kann. Um die Problematik und deren Lösung zu verdeutlichen, wird zunächst der allgemeine Aufbau eines PH_{Modes} gezeigt.

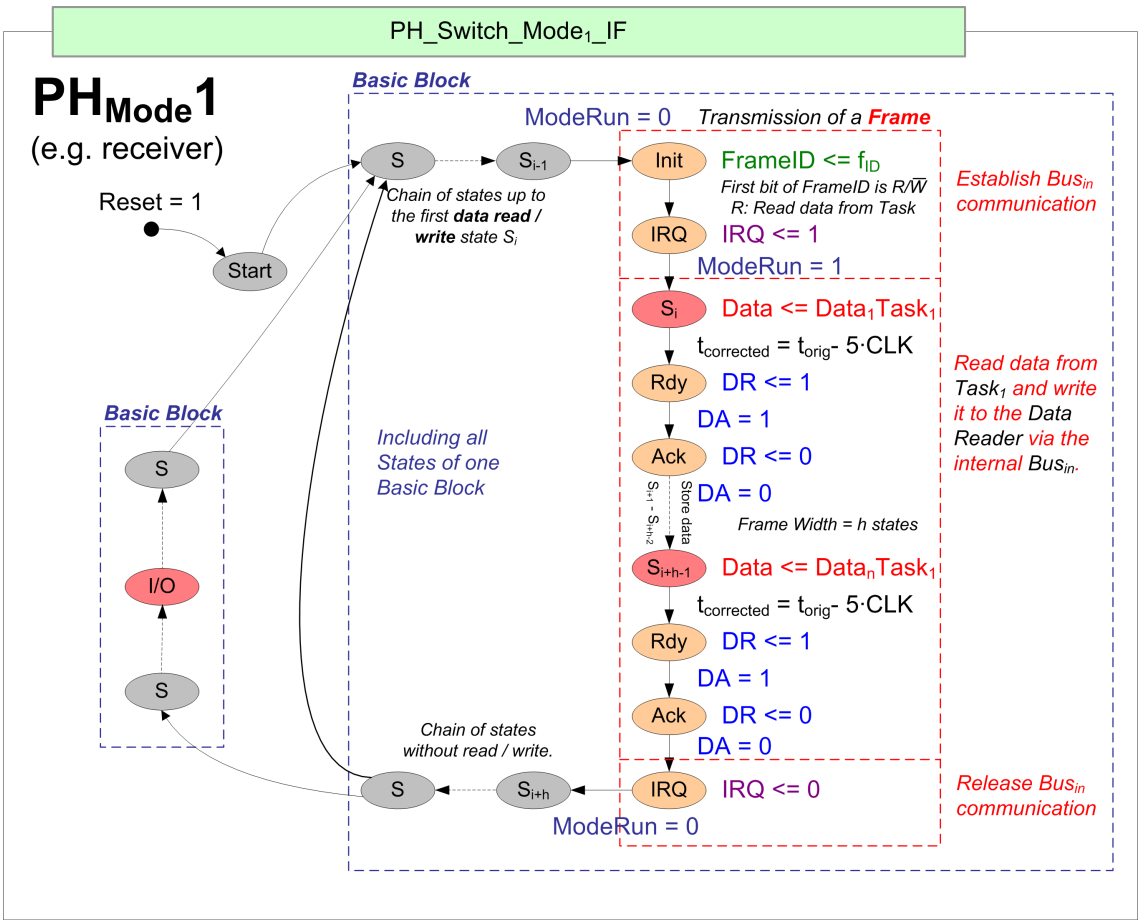

Abbildung 2.6: Aufbau eines PH_{Modes}

Abbildung 2.6 zeigt einen beispielhaften PH_{Mode}, dessen Automat einen `Start` Zustand hat, aus dem in den eigentlichen Lesevorgang gewechselt wird. In *Establish* Bus_{In} *communication* fordert der PH_{Mode} den Buszugriff an. Mit einem einmaligen Durchlauf kann der Input-Frame des blau umrandeten Basic Blocks genau einmal gelesen werden. Nach dem Lesen wird der Datenbus in dem Zustand *Release* Bus_{In} *communication* wieder frei gegeben.

Problematisch wäre nun ein Szenario, in dem ein Input-Frame zweimal gelesen werden muss, um alle benötigten Daten für einen Output-Frame einzulesen. Somit müsste der Basic Block, der den Input-Frame beinhaltet, zweimal ausgeführt werden.

Ebenso wäre ein Szenario denkbar, in dem für einen Input-Frame mehrere Instanzen eines Output-Frames notwendig wären. Das würde implizieren, dass auch das Processing mehrfach ausgeführt werden müsste. Der Scheduler im IFB lässt nur die einmalige Ausführung des Inputs, Processings und Outputs pro Kommunikationszyklus zu. Wenn also innerhalb einer Abbildungsvorschrift mehrere Instanzen eines Frames benötigt werden, erfordert dies eine Modifikation der PH_{Modes}.

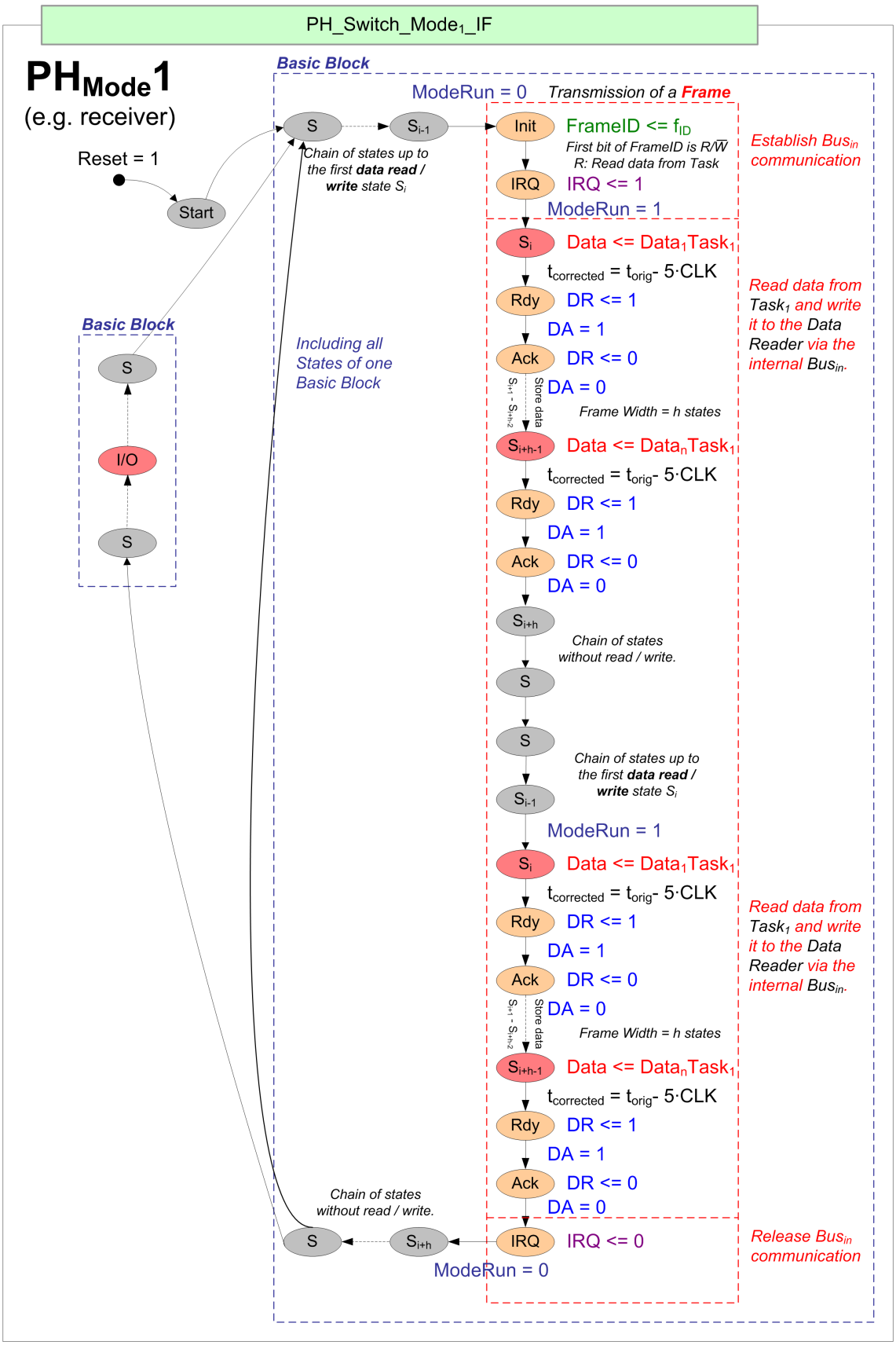

Abbildung 2.7: Ein modifizierter PH_{Modes}

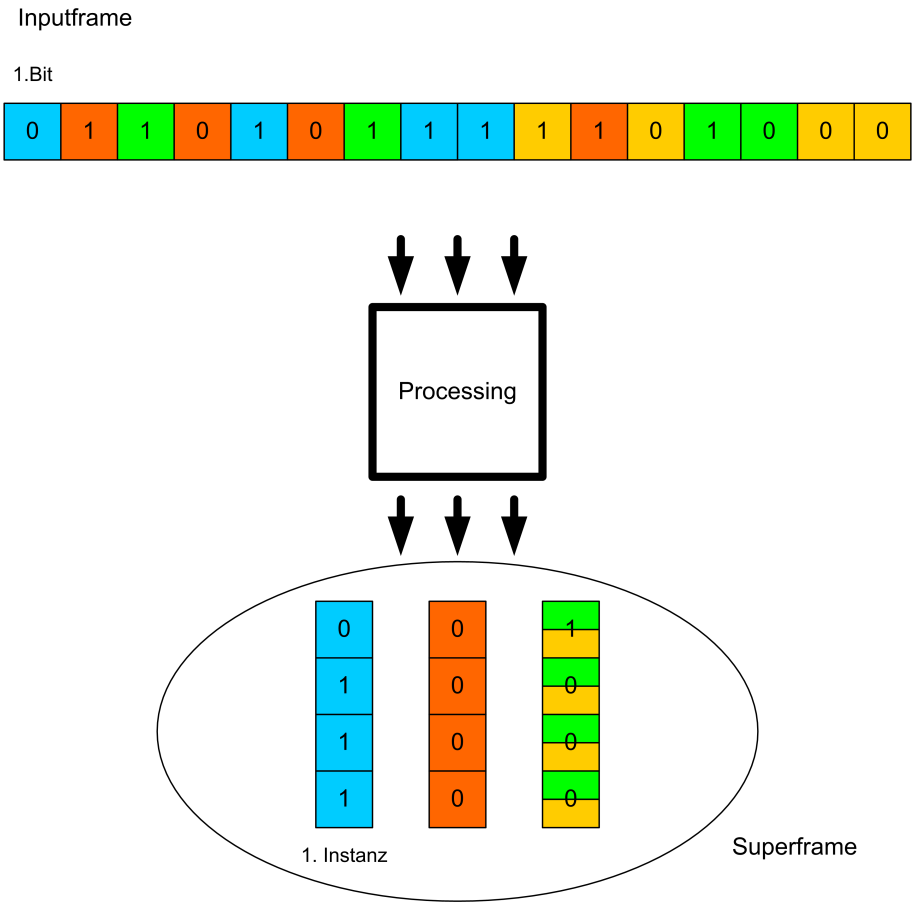

Abbildung 2.8: Nutzung von Instanzen und Superframes

In Abbildung 2.7 ist ein modifizierter PH_{Mode} zu sehen. Dieser PH_{Mode} besitzt einen Automaten, in dem ein Basic Block "verdoppelt" wurde. Das führt dazu, dass der ursprüngliche Basic Block *zweimal* direkt hintereinander ausgeführt wird. Die "Vervielfachung" eines Basic Blocks ist immer dann erlaubt, wenn ein Basic Block eine Selbsttransition besitzt. Der durch diese Transformation entstandene Frame wird als *Superframe* bezeichnet und erlaubt es, mehrere Instanzen eines Pakets zu lesen bzw. zu schreiben.

Abbildung 2.8 zeigt ein Szenario, in dem durch die Verwendung von Superframes das I-P-O Schema eingehalten werden kann. Das Beispiel zeigt einen relativ großen Input-Frame, der in drei Instanzen des Output-Frames versendet werden soll. Hierbei wurde der Superframe durch die Verdreifachung des Basic Blocks gebildet.

2.6 Detaillierte Darstellung des IFB Datenflusses

Um die nachfolgende Optimierung der Latenz besser verstehen zu können, wird in diesem Abschnitt noch einmal ausführlich der Datenfluss des IFB betrachtet. Dazu werden die einzelnen Aktionen aufgeführt, um die Interaktion zwischen den einzelnen Komponenten inklusive des *fully interlocked protocol* aufzuzeigen.

Takt	Phase	Aktion	Komponente
1	1	SetFrameId	PH_{Mode}
2	1	SetIRQ	PH_{Mode}
3	1	GrantBus	$CTRL_{In}$
4	2	Daten lesen	PH_{Mode}
5	2	SetDR	PH_{Mode}
6	2	Daten in Speicher übernehmen	Data Reader
7	2	SetDA	Data Reader
8	2	ResetDR	PH_{Mode}
9	2	ResetDA	Data Reader
10	3	ResetIRQ	PH_{Mode}
11	3	AssignFrame or ReleaseBus	$CTRL_{In}$
12	3	Frame wird in P_{in} Register übernommen	CU
13	4	ModifyData	SH_{Mode}
14	4	Set SH_{Mode} Complete	SH_{Mode}
15	4	Set SH_{Mode} Complete	CU
16	5	SetFrameId	PH_{Mode}
17	5	SetIRQ	PH_{Mode}
18	5	GrantBus	$CTRL_{Out}$
19	6	SetDR	PH_{Mode}
20	6	Daten auf Datenbus	DataWriter
21	6	SetDA	Data Writer
22	6	Daten schreiben	PH_{Mode}
23	6	ResetDR	PH_{Mode}
24	6	ResetDA	Data Writer
25	7	ResetIRQ	PH_{Mode}
26	7	AssignFrame or ReleaseBus	CU
27	7	Reset SH_{Mode} Complete	Scoreboard

Tabelle 2.1: Detaillierte Darstellung des Datenflusses im IFB

Die Tabelle 2.1 beschreibt Taktzyklen-genau das Einlesen, das Verarbeiten und das Senden eines Frames mit jeweils einem Bit. Dabei können sieben Phasen unterschieden werden. Die 27 Schritte wurden direkt aus den kommunizierenden Automaten des IFBs abgeleitet und beschreiben die präzise Übertragung eines Frames. Die Schritte stellen eine Verfeinerung des in Kapitel 2.3 angegebenen Datenflusses dar.

Phase 1 – `Establish` Die erste Phase (Takt eins bis drei) beinhaltet das Festlegen der Frame ID und die Bewerbung um den Datenbus.

Phase 2 – `Read` In der zweiten Phase (Takt vier bis neun) läuft der eigentliche Lesevorgang der Daten ab. Diese Phase vervielfacht sich je nach Anzahl der einzulesenden Bits.

Phase 3 – `Release` Das Einlesen wurde abgeschlossen und der Datenbus wird wieder freigegeben (Takt zehn bis zwölf).

Phase 4 – `Process` In den Takten 13 bis 15 werden die Daten modifiziert und abgespeichert.

Phase 5 – `Establish` Nun sollen die Daten an die empfangsbereite Task gesendet werden. Dazu wird der Datenbus benötigt. Dazu werden (wie beim Einlesen) drei Takte benötigt.

Phase 6 – `Write` Takte 19 bis 24 beschreiben das Schreiben eines Bits zu einer empfangsbereiten Task. Werden mehrere Bits geschrieben, müssen diese sechs Takte für jedes Bit wiederholt werden.

Phase 7 – `Release` Das Senden der Daten ist abgeschlossen und die letzten drei Takte geben u.a. den Datenbus wieder frei.

Die ersten drei Phasen `Establish`, `Read` und `Release` sind Teil jeder Input Stage, die Phase `Process` steht für die gleichnamige Stage im I-P-O Schema. Die letzten drei Phasen werden der Output Stage zugeordnet.

3 Datenflussoptimierung

In diesem Kapitel wird erläutert, wie eine Optimierung des IFB-internen Datenflusses erreicht werden kann. Dazu wird zunächst die nicht optimierte Ausführung des IFB vorgestellt. Anschließend werden zwei Verfahren zur Verkürzung der Latenz beschrieben und die erzielten Ergebnisse miteinander verglichen.

3.1 Ausführung des IFB als Pipeline

In diesem Abschnitt wird zunächst das ursprüngliche Ausführungsmodell des IFBs ohne Optimierung beschrieben. Dieses Modell dient als Basis für die beiden in diesem Kapitel vorgestellten Optimierungsverfahren. In Kapitel 2.5 wurde bereits die Ausführung eines Kommunikationszyklus in den Schritten Input-Processing-Output vorgestellt. Dort wurde bereits angesprochen, dass diese Schritte als Stages einer Pipeline angesehen werden können, deren Organisation durch das Scoreboard sichergestellt wird. Da Protokolle zyklisch ausgeführt werden, wiederholen sich die Stages Input, Processing und Output bei jeder Ausführung eines Kommunikationszyklus. Als Teil des Schedulings repräsentieren die Input und Output Stages Frames, die vom Protocol Handler empfangen bzw. versendet werden. Entsprechend existieren Processing Stages für jede Datentransformation (Mapping Equation) im Sequence Handler.

In Abbildung 3.1 wird das Scheduling für das einfachste Szenario einer Kommunikation bestehend aus einem Input-Frame, der Transformation und einem Output-Frame dargestellt. Gegeben sind die ersten beiden Zyklen; alle weiteren Zyklen können entsprechend konstruiert werden. Dieses Scheduling wird im Folgenden als simple Schedule bezeichnet.

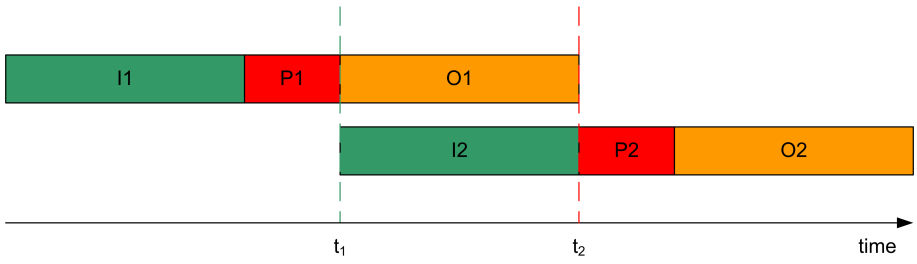

Abbildung 3.1: simple Schedule I

Bevor eine Stage ausgeführt werden kann, müssen bestimmte Bedingungen erfüllt sein, wie sie in Kapitel 2.5 informal angegeben wurden. Dabei existieren je nach Schedulingverfahren charakteristische Bedingungen für die Input, Processing und Output-Stages:

Input Bedingung

Die Input Stage eines Kommunikationszyklus darf erst dann ausgeführt werden, wenn die Processing Stage des vorigen Kommunikationszyklus beendet ist (hier durch die grüne Linie zum Zeitpunkt t_1 gekennzeichnet), denn erst nachdem alle Daten aus dem Data Reader verarbeitet wurden, dürfen sie durch neue Daten überschrieben werden.

- Input(x)[1] nach Process(x-1)

Processing Bedingungen

Die Processing Stage darf erst dann beginnen, wenn alle Daten der Input Stage eingelesen und im Data Reader gespeichert sind (t_2, rote Linie). Diese Bedingung wird durch das I-P-O Schema vorgegeben und vom Scoreboard kontrolliert. Außerdem muss im vorigen Kommunikationszyklus das Versenden der Daten abgeschlossen sein (in diesem Beispiel ebenfalls zum Zeitpunkt t_2). Würde die Processing Stage früher beginnen, könnten noch nicht versendete Daten aus dem vorigen Zyklus durch neu berechnete Daten des aktuellen Kommunikationszyklus überschrieben werden.

- Process(x) nach Input(x)
- Process(x) nach Output(x-1)

Output Bedingung

Die Output Stage muss auf die Beendigung der Processing Stage warten, da erst dann alle Daten verarbeitet sind. Diese Bedingung ist ebenfalls durch die Kausalität im I-P-O Schema vorgegeben.

- Output(x) nach Process(x)

In dem beschriebenen Beispiel werden genau so viele Daten eingelesen wie versendet werden. In Abbildung 3.2 ist ein modifiziertes Beispiel angegeben, in dem der Input-Frame kleiner als der Output-Frame ist. Die Input Stage zum Zeitpunkt t_2 beendet, aber die Output Stage des vorigen Zyklus erst zum Zeitpunkt t_3. Die Processing Stage kann also erst zum Zeitpunkt t_3 gestartet werden.

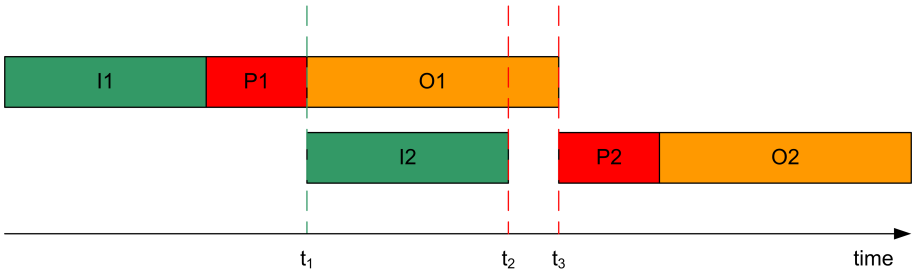

Abbildung 3.2: simple Schedule II

[1] das x bezeichnet hier die Nummer des Kommunikationszyklus

Das Scheduling in den Abbildungen 3.1 und 3.2 vorgestellten Kommunikationszyklen lässt sich nicht weiter optimieren, wenn man davon ausgeht, dass die repräsentierten Frames keine Superframes sind. Das bedeutet, dass innerhalb der Input und Output Stage jeweils genau eine Instanz der ursprünglichen Frames enthalten ist.

3.2 Optimierungskonzept

Die Datenflussoptimierung zur Reduzierung der Latenz nutzt eine Besonderheit des I–P–O Schemas aus. Eine Optimierung der Latenz des IFB ist möglich, wenn bedingt durch das IFD-Mapping ein Superframe für die Output Stage und damit auch für die Processing Stage vorliegt. Wie in Kapitel 2.4 beschrieben, werden in einem solchen Fall eine große Menge eingehender Daten eines einzigen Frames auf mehrere Instanzen kleinerer ausgehender Frames abgebildet, die zu einem Superframe zusammengefasst werden. In Abbildung 3.3 ist ein solches Szenario dargestellt.

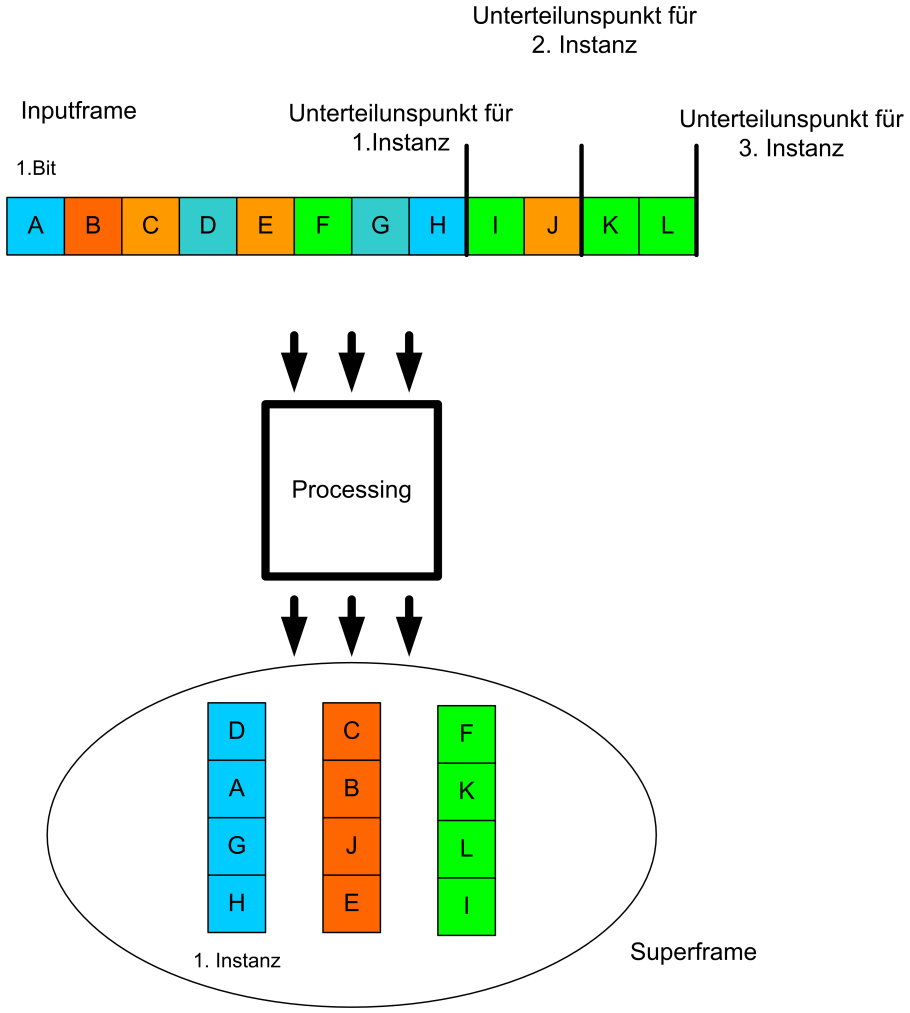

Abbildung 3.3: Einführung von Unterteilungspunkten innerhalb des Input-Frames

Das Beispiel in Abbildung 3.3 zeigt, dass die Möglichkeit besteht, dass die Daten der ersten Instanz im Output-Superframe bereits vollständig eingelesen wurden (beim ersten Unterteilungspunkt), bevor das letzte Bit des Input-Frames im IFB eintrifft.

Es werden in diesem Beispiel 16 Bits sequentiell eingelesen und ein Superframe bestehend aus drei Instanzen mit je vier parallelen Bits versendet. Die erste Instanz benötigt die Bits A, D, G und H des Input-Frames. Der SH_{Mode} kann die Verarbeitung dieser vier Bits starten, sobald diese im Speicher abgelegt sind (beim ersten Unterteilungspunkt). Dann versendet der entsprechende PH_{Mode} die erste Instanz des Superframes. Sind die Bits B, C, E und J komplett eingelesen (zweiter Unterteilungspunkt), können diese verarbeitet und die zweite Instanz kann versendet werden. Ist der Input-Frame komplett eingelesen (letzter Unterteilungspunkt), sind auch die Informationen für die letzte Instanz vorhanden, so dass diese transformiert und dann in das ausgehende Protokoll integriert werden können.

Um festzustellen, wann alle benötigten Daten für eine Instanz im Output-Superframe vollständig eingelesen sind, wird das IFD-Mapping analysiert, das in Kapitel 2.4 vorgestellt wurde. Da das IFD-Mapping die Informationen beinhaltet, welche Bits aus dem Input-Frame auf welche Bitpositionen des Output-Superframes gemappt werden, kann daraus abgeleitet werden, wann frühestens mit dem Processing der Daten einer Instanz innerhalb des Output-Superframes begonnen werden kann. Aus diesen Informationen ergeben sich die Unterteilungspunkte.

Somit lässt sich für jede Instanz innerhalb des Output-Superframes ableiten, wann mit der Verarbeitung der eingehenden Daten begonnen werden kann. Die Datenflussoptimierung besteht also darin, einen "großen" eingehenden Frame in mehrere Abschnitte zu unterteilen und diese dann frühzeitig zur berechnen und auszugeben. Diese Abschnitte werden im Folgenden als *Subframes* bezeichnet. Diese Subframes stellen einen genau spezifizierten Teil eines Frames dar, d.h. jeder Subframe ist genau einem Frame zugeordnet und beinhaltet einen Abschnitt zwischen zwei Unterteilungspunkten. Da ein Subframe bei einem Unterteilungspunkt beginnt und beim nächsten aufhört, ist es nicht möglich, dass Subframes sich überlappen.

$$\bigcap \text{Subframes} = \emptyset$$

Damit ergibt sich als alternative Beschreibung für einen Frame

$$\text{Frame} = \bigcup \text{Subframes}.$$

Allerdings müssen zwei Bedingungen eingehalten werden, damit die Unterteilung des Frames in Subframes erfolgen kann:

- Der Input-Frame beinhaltet Informationen für mehr als einen Output-Frame, so dass mehrere Instanzen benötigt werden. Besteht die Ausgabe dagegen aus nur einem Frame, müssen alle Daten eingelesen werden und es ergeben sich keine Unterteilungspunkte.
- Der Input-Frame muss eine serielle Folge von Datenbits beinhalten. Ist diese Bedingung nicht erfüllt, d.h. sind alle Eingangsdaten parallel, werden sie gleichzeitig eingelesen. Eine Unterteilung ist somit nicht möglich.

Sind diese Bedingungen erfüllt, können anhand der Unterteilungspunkte die Subframes definiert werden: Der erste Input-Subframe beginnt am Anfang des Input-Frames und reicht bis zum ersten Unterteilungspunkt (siehe Abbildung 3.3). Daran schließt sich der zweite Subframe bis zum nächsten Unterteilungspunkt an, daran wiederum der dritte usw.

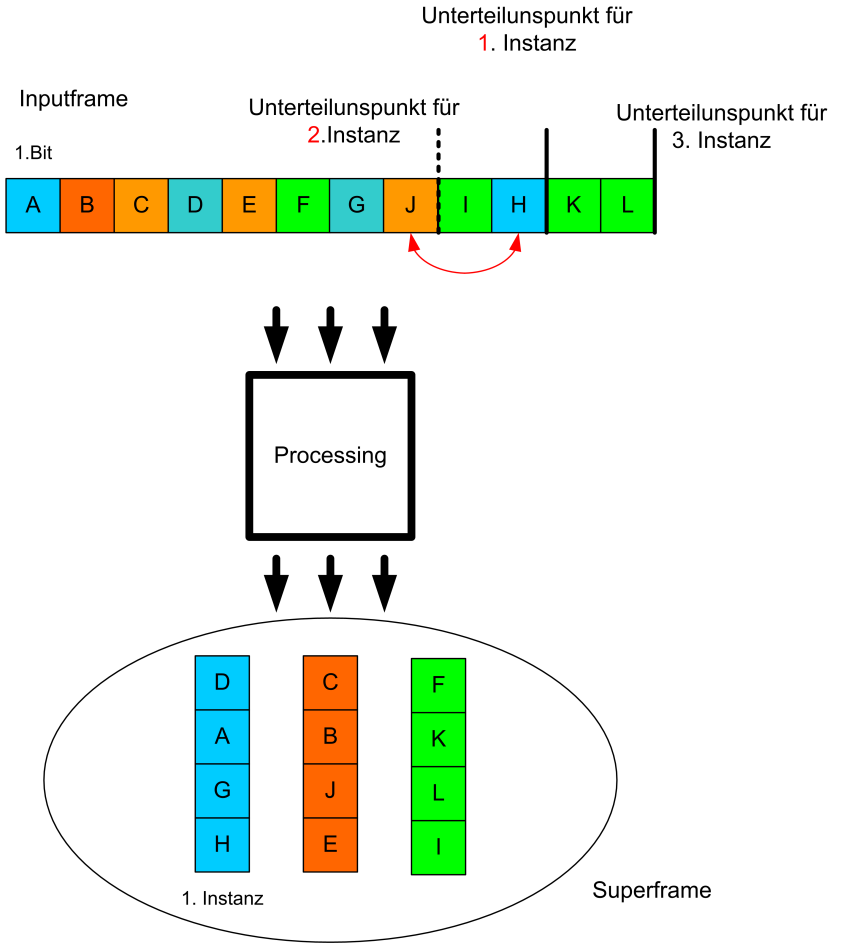

Abbildung 3.4: Beispiel für den Wegfall von Unterteilungspunkten

Dabei kann es zu folgender Situation kommen: Abbildung 3.4 ist eine Modifikation der Abbildung 3.3. In diesem Beispiel sind die Daten für die zweite Instanz bereits vor dem Unterteilungspunkt der ersten Instanz eingelesen. Die Reihenfolge, in der Subframes verarbeitet werden, ist aber fest vorgegeben. Dadurch entfällt der Unterteilungspunkt für die zweite Instanz. Erst nachdem der erste Unterteilungspunkt erreicht ist, kann die erste Instanz ausgeführt werden. Da die Daten für die zweite Instanz bereits vollständig vorliegen, können diese direkt im Anschluss an die erste Instanz transformiert werden.

Ein Nachteil dieses Optimierungsansatzes besteht im Folgenden: Benötigt eine Instanz Daten, die erst am Ende des Input-Frames eingelesen werden, so reduziert sich der Optimierungseffekt, da alle nachfolgenden Instanzen erst dann ausgeführt werden dürfen, wenn die Informationen für den Vorgänger komplett eingelesen und transformiert sind. Der schlimmste anzunehmende Fall ist der, dass die erste Output-Frame-Instanz das letzte einzulesende Bit benötigt. In diesem Fall ist der Optimierungseffekt gleich Null, da die Verarbeitung erst nach dem Einlesen des letzten Bits beginnt.

Um die Aufteilung der Input-Frames in der automatischen Generierung zu realisieren, existieren zwei Möglichkeiten. Die erste Möglichkeit beinhaltet das Einführen von Subframe IDs. Eine Subframe ID wird genau einem Subframe zugeordnet. Somit sind die Subframes eindeutig identifizierbar und unterscheidbar. Diese Verwendung der Subframe IDs kann dazu genutzt werden, um festzustellen, wann ein Subframe komplett eingelesen wurde. Wird das Ende eines Subframes beim Einlesen erreicht, wird die Subframe ID inkrementiert. Das ist das Signal, dass die Daten für die nächste Output-Frame-Instanz vorhanden sind und transformiert werden können.

Die zweite Möglichkeit besteht darin, dass der PH_{Mode} nach dem Einlesen eines Subframes den Datenbus wieder freigibt (`Release`) und sich dann für den neuen Subframe erneut darum bewirbt. Ein Frame wird somit nicht als Menge von Subframes verarbeitet, sondern als Menge von kleinen Frames angesehen. Dabei geht die eindeutige Zuordnung eines Subframes zu einem Frame nicht verloren!

3.3 Optimierungsverfahren

In den beiden folgenden Abschnitten werden zwei Optimierungsansätze für das Beispiel aus Abbildung 3.3 präsentiert. Die beiden Ansätze resultieren aus den beiden Möglichkeiten, die Subframes zu realisieren.

3.3.1 framebased schedule

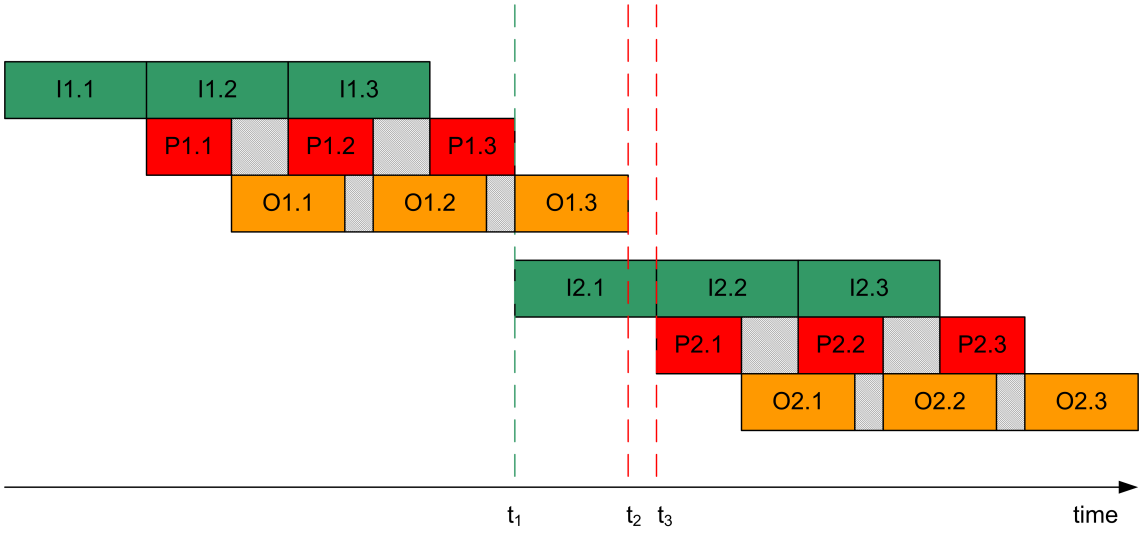

Abbildung 3.5: framebased Schedule

Dieses Verfahren verwendet die Subframe IDs. Wie bereits beschrieben, wird jedem Subframe eine solche ID zugewiesen, um die einzelnen Subframes voneinander zu unterscheiden. Kontrolliert und verwaltet werden die Subframe IDs vom PH_{Mode}, der an den Unterteilungspunkten die Subframe ID inkrementiert. Es ergibt sich folgende Struktur:

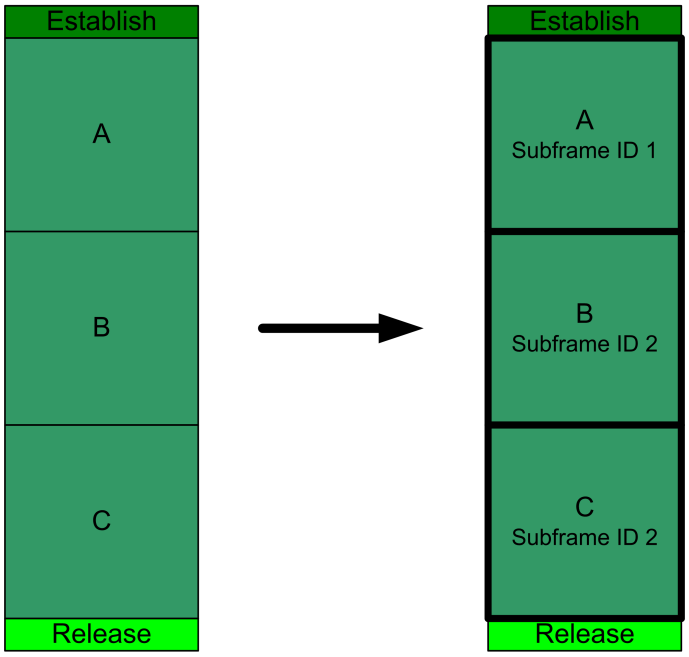

Abbildung 3.6: Links der ursprüngliche Frame, rechts der Frame mit der internen Aufteilung in die drei Subframes

Wie in Abbildung 3.6 sichtbar, bleibt der Frame als solcher erhalten, es werden nur Abschnitte (die Subframes) in diesem Frame definiert.

Hat ein PH_{Mode} einen Subframe komplett eingelesen, muss der SH_{Mode}, der die Daten aus diesem Subframe transformiert, darüber informiert werden. Dazu sendet der PH_{Mode} über eine neue Leitung, die zwischen dem PH_{Mode} und dem SH_{Mode} eingerichtet wird, ein entsprechendes Signal an den SH_{Mode}. Diese zusätzliche Leitung fungiert als *Bypass* zur CU und ist notwendig, da das Scoreboard einen Start des SH_{Modes} erst dann zulassen würde, wenn der komplette Frame eingelesen ist. Zusätzlich benötigt der PH_{Mode} eine Erweiterung der Zustände, um die Subframe IDs zu verwalten und den Subframes zuzuweisen. Dabei kann die bereits vorhandene Frame ID erweitert werden, so dass sie auch die Subframe ID enthält. Die CU muss derart modifiziert werden, dass der SH_{Mode} bereits dann aktiviert wird, sobald der Datenbus vom PH_{Mode} arbitriert wurde, damit der SH_{Mode} rechtzeitig schon während des Einlesens mit der Verarbeitung beginnen kann.

Der SH_{Mode} benötigt zusätzliche Zustände, um die Subframe ID auszuwerten. Nach der Verarbeitung eines Subframes wartet der SH_{Mode} mit der weiteren Verarbeitung darauf, dass der PH_{Mode} die Subframe ID inkrementiert.

Input Bedingung

Wie im simple Schedule müssen auch bei diesem Optimierungsansatz bestimmte Bedingungen eingehalten werden, die durch die Linien in Abbildung 3.5 verdeutlicht werden. Die erste Input Stage eines Kommunikationszyklus darf erst dann begonnen werden, wenn *alle* Verarbeitungsphasen aus dem vorigen Zyklus beendet wurden. In Abbildung 3.5 ist diese Bedingung zum Zeitpunkt t_1 erfüllt. Ein verfrühtes Einlesen könnte Daten im Data Reader überschreiben, die noch nicht von einem SH_{Mode} verarbeitet wurden.

- Input(x.1)[2] nach Process(x-1.L)[3]

Processing Bedingungen

Eine Processing Stage darf – gemäß dem I-P-O Schema – erst dann ausgeführt werden, wenn die dazugehörige Input Stage beendet wurde. Bspw. muss P2.1 auf I2.1 warten (t_3). Analog zum simple Schedule gilt außerdem die Bedingung, dass die Output Stage des vorigen Zyklus beendet sein muss, um das Überschreiben nicht gesendeter Daten zu vermeiden (t_2). Beide Bedingungen sind in Abbildung 3.5 durch die roten Striche gekennzeichnet.

- Process(x.1) nach Output(x-1.L)
- Process(x.y) nach Input(x.y)

[2]die erste Ziffer beschreibt wieder die Nummer des Kommunikationszyklus, die zweite Ziffer die Nummer des Subframes

[3]das L kennzeichnet den letzten Subframe einer Stage, in diesem Fall wäre dies Process(x-1.3)

Output Bedingung

Die Ausführung der Output Stage richtet sich wiederum nach dem Ende der zugehörigen Processing Stage. O2.1 darf nicht gestartet werden, bevor P2.1 beendet wurde. Diese Bedingung wird durch das Scoreboard kontrolliert, da sie durch die Kausalität im I-P-O Schema vorgegeben ist.

- Output(x.y) nach Process(x.y)

Somit kann eine Verbesserung erreicht werden. Die Verarbeitung in einem Kommunikationszyklus kann früher begonnen werden (nämlich nach dem ersten Subframe), was dazu führt, dass die komplette Verarbeitung früher beendet wird. Das wiederum wirkt sich auf den nächsten Kommunikationszyklus aus, der ja genau diesen Zeitpunkt abwarten muss, bevor das Input beginnen kann. Allerdings müssen dafür Subframe IDs dem IFB hinzugefügt werden, was eine Modifikation der PH_{Modes} und der SH_{Modes} erfordert. Zusätzlich muss die CU in geringfügigem Maße geändert werden, so dass der SH_{Mode} aktiviert wird, sobald Daten vom PH_{Mode} an den Data Reader gesendet werden. Eine qualitative Auswertung des Ansatzes ist in Kapitel 3.4 gegeben.

3.3.2 subframebased schedule

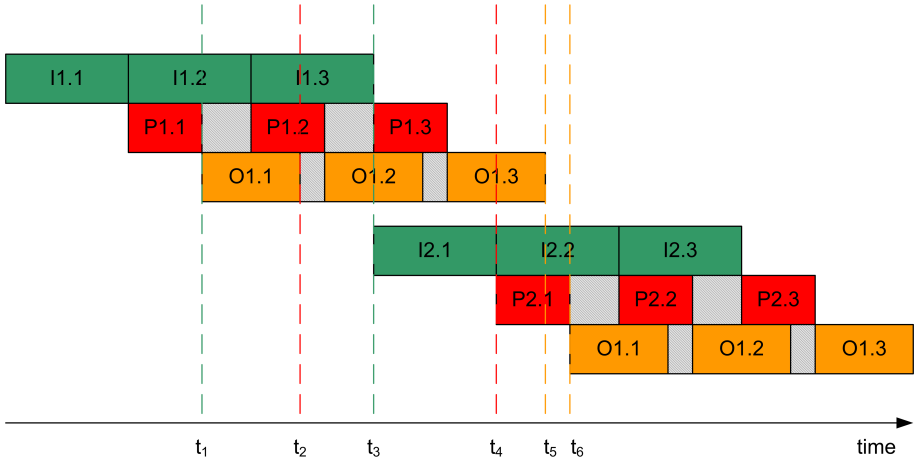

Abbildung 3.7: subframebased Schedule

Im Gegensatz zum framebased Schedule, der Subframe IDs verwendet, wird in diesem Ansatz wird die eigentliche Struktur des Frames geändert. Die Subframes werden nun zu "unabhängigen" Frames erweitert, indem jeder Subframe um eine `Establish` und eine `Release` Phase erweitert wird. Das führt dazu, dass jeder Subframe des eingehenden Protokolls nun eine Input Stage darstellt (siehe Abbildung 3.8).

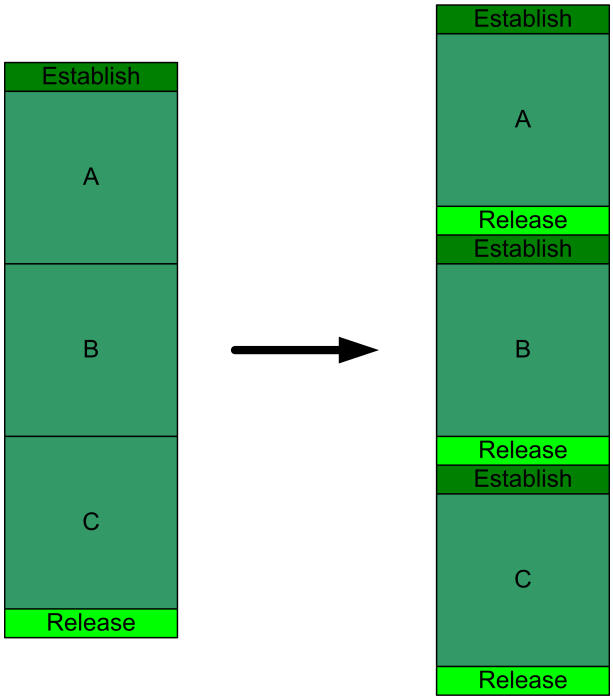

Abbildung 3.8: Links der ursprüngliche Frame, rechts die Subframes als 'unabhängige' Frames

Dies hat den Effekt, dass sich der PH_{Mode} für jeden Subframe um die Belegung des Datenbusses bewirbt und nach jedem Subframe den Datenbus wieder freigibt. Dann bewirbt er sich erneut um den Bus. Diese Vorgehensweise ermöglicht es anderen, höher priorisierten Prozessen, den Datenbus zwischenzeitlich zu belegen, was in zeitbehafteten Protokollen zu Verletzung von Zeitschranken führen kann.

Für dieses Verfahren muss der IFB jedoch nicht modifiziert werden. Die nun unabhängigen Subframes werden als normale Frames eingelesen und behandelt. Es werden auch keine Subframe IDs mehr verwendet, sondern jeder Subframe ist jetzt durch eine normale Frame ID gekennzeichnet.

Input Bedingungen

Die Bedingungen, die in diesem Scheduling gelten, sind vergleichbar mit denen im simple Schedule (da ja nichts anderes als das Einlesen von Frames gemacht wird). Die erste Input Stage eines Kommunikationszyklus (bspw. I2.1) darf dann begonnen werden, wenn sowohl die letzte Input Stage des vorigen Kommunikationszyklus beendet ist (I1.3) als auch das Processing des vorigen Kommunikationszyklus mit der gleichen ID (hier P1.**1**). Im Beispiel in Abbildung 3.7 ist letztere Bedingung bereits zum Zeitpunkt t_1 erfüllt, aber die erste Input Stage muss noch bis zum Zeitpunkt t_3 warten, da erst die Input Stage aus dem ersten Kommunikationszyklus beendet sein muss. Alle weiteren Subframes (hier I2.2 und I2.3) müssen ebenfalls auf die Beendigung der Processing Stages aus der vorigen Stufe (also P2.**2** bzw. P1.**3**) warten. Zusätzlich muss der vorige Subframe eingelesen und der Datenbus zum Data Reader wieder freigegeben sein.

- Input(x.1) nach (Input(x-1.L) und Process(x-1.1))
- Input(x.y) nach (Input(x.y-1) und Process(x-1.y)) für y > 1

Processing Bedingungen
Für die Processing Stages gelten die folgenden Bedingungen: zum Einen müssen die Daten, die in dieser Stage verarbeitet werden sollen, vorhanden sein. Diese Bedingung ist zum Zeitpunkt t_4 erfüllt. Zum Anderen muss die Output Stage aus dem vorigen Zyklus mit der gleichen Subframe Nummer beendet sein (bspw. muss P2.**1** auf O1.**1** warten). Dies ist in Abbildung 3.7 zum Zeitpunkt t_2 erfüllt. Beide Bedingungen sind durch die roten Striche kenntlich gemacht.

- Process(x.y) nach (Input(x.y) und Output(x-1.y))

Output Bedingungen
Auch in diesem Verfahren gelten unterschiedliche Bedingungen für den ersten und für die restlichen Subframes. Der Subframe darf erst dann ausgeführt werden, wenn die Daten, die in dieser Output Stage versendet werden sollen, alle vorliegen (O2.1 muss also auf P2.1 warten) und wenn die letzte Output Stage aus dem vorigen Kommunikationszyklus beendet ist (was zum Zeitpunkt t_5 erfüllt ist). Alle anderen Output Stages müssen auf die zugehörige Processing Stage warten und auf das Ende der vorigen Output Stage (bspw. muss O2.2 auf P2.2 und O2.1 warten).

- Output(x.1) nach (Process(x.1) und Output(x-1.L))
- Output(x.y) nach (Process(x.y) und Output(x.y-1)) für y > 1

Auch durch das subframebased Schedule kann eine Verbesserung der Latenz erzielt werden. Durch das Umwandeln von Subframes in Frames konnte erreicht werden, dass die Verarbeitung noch früher begonnen werden konnte. Es musste nicht das Ende der letzten Processing Stage (bspw. P1.3) abgewartet werden, sondern das Ende von P1.1 und die Freigabe des Datenbusses (I1.3).

In diesem Verfahren ist eine Veränderung des IFBs, wie sie im framebased Schedule notwendig ist, nicht notwendig. Die Verbesserung wird durch das Hinzufügen zusätzlicher `Establish` und `Release` Phasen erreicht.

Im nächsten Abschnitt werden einige konkrete Beispiele präsentiert, um die Verbesserungen, die durch die einzelnen Schedules gewonnen werden, qualitativ abzuschätzen.

3.4 Ergebnisse

Die Ergebnisse, die in diesem Abschnitt präsentiert werden, wurden durch eine Erweiterung des IFS-Editors generiert, die im Rahmen dieser Studienarbeit entwickelt wurde. Dabei handelt es sich um eine Visualisierung von Scheduling Diagrammen, wie sie in Abbildung 3.9 dargestellt ist.

Um die Beispiele zur Auswertung der beiden vorgestellten Optimierungsverfahren zu generieren, müssen zwei Annahmen bezüglich der Daten im Superframe gemacht werden. Die erste Annahme legt fest, dass die Menge der eingelesenen und versendeten Daten gleich ist. Daraus folgt, dass die Anzahl der Output Instanzen, die ursprünglich den Output-Supeframe gebildet haben, konstant ist. Die jeweilige Verteilung der Bits und damit der Anordnung der Unterteilungspunkte resultiert ausschließlich in einer bestimmten Gruppierung der Output Instanzen. Die zweite Annahme bezieht sich auf eben diese Gruppierungen der Output Instanzen. Dabei wird in den Beispielen davon ausgegangen, dass sich die Instanzen in zwei, vier und acht Gruppen äquivalenter Größe unterteilen lassen.

Die Beispiele sind folgendermaßen aufgebaut:

- es werden zwei Kommunikationszyklen betrachtet
- pro Kommunikationszyklus werden acht Bits vom IFB eingelesen und acht Bits gesendet
- im framebased Schedule und im subframebased Schedule werden diese acht Bits nacheinander in Subframes zu vier, zwei und einem Bit aufgeteilt

Abbildung 3.9 zeigt einen Ausschnitt des simple Schedules, das mit der implementierten Erweiterung des IFS-Editors visualisiert wird.

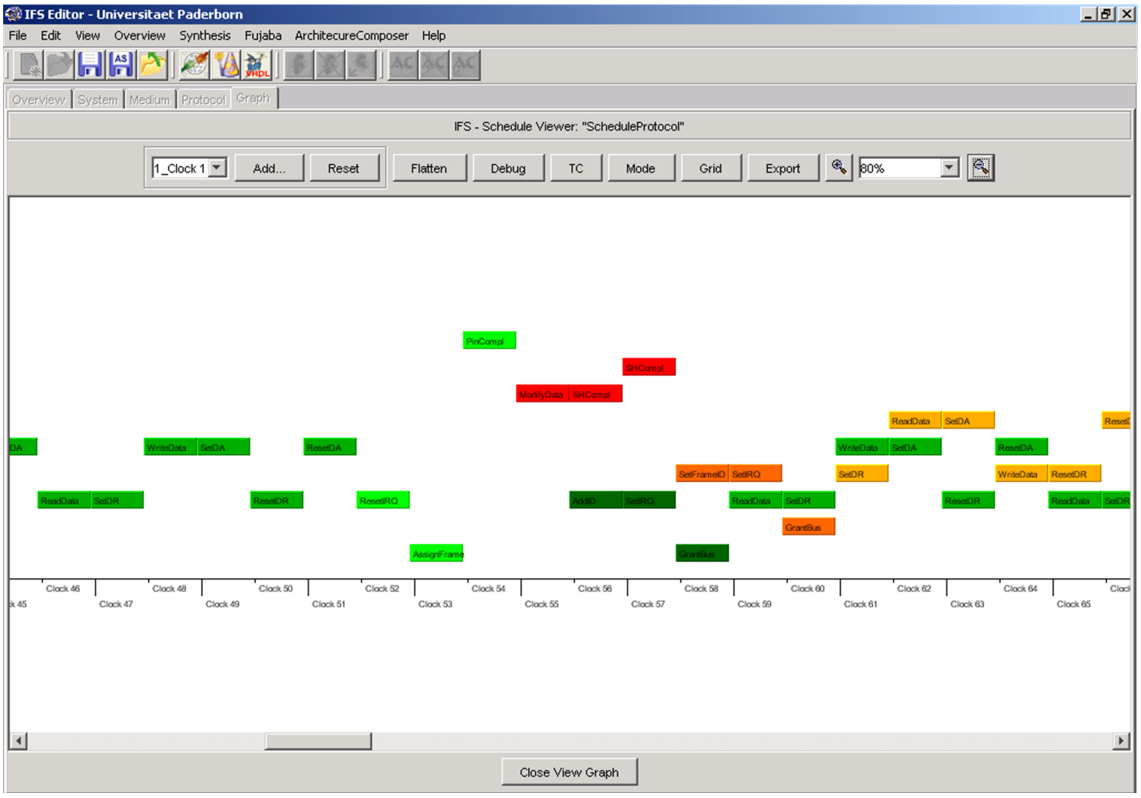

Abbildung 3.9: simple Schedule mit 8 Bits pro Frame

Eine Betrachtung der zyklengenauen Ausführung (Tabelle 2.1) erlaubt vorweg eine einfache Optimierung. Wie in Abbildung 3.9 sichtbar, konnte noch eine geringfügige Verbesserung implementiert werden: die ersten beiden Schritte der `Establish` Phase im zweiten Zyklus (hier dunkelgrün dargestellt) können bereits *während* des Processings (rot) ausgeführt werden (obwohl die Input Bedingung in Kapitel 3.3.1 besagt, dass das `Establish` erst nach dem Processing ausgeführt werden dürfte). Dies ist dadurch möglich, dass die beiden ersten Schritte nur die Frame ID und den IRQ setzen (siehe Tabelle 2.1 in Kapitel 2.6). Diese Schritte beeinflussen nicht die Verarbeitung oder den Inhalt des Speichers. Der letzte Schritt, die Anforderung des Datenbusses kann erst dann ausgeführt werden, wenn das Processing komplett beendet ist. Mit dieser zusätzlichen Verbesserung kann der zweite Kommunikationszyklus bereits im 56. Takt begonnen werden. Nach 166 Takten sind beide Zyklen beendet. Diese 166 Takte sind der Referenzwert für die beiden Schedulings.

In Abbildung 3.10 wird das **framebased Schedule** dargestellt. Die Input-Frames wurden in zwei Subframes mit je vier Bits aufgeteilt. Durch die Aufteilung kann der erste Zyklus bereits nach 82 Takten beendet werden, der zweite Kommunikationszyklus ist in Takt 139 abgearbeitet. Es liegt also eine Verbesserung von 17% gegenüber dem simple Schedule vor.

Im **subframebased Schedule** mit zwei Subframes a vier Bits (Abbildung 3.11) kann der erste Kommunikationszyklus nach 91, der zweite nach 151 Takten beendet werden.

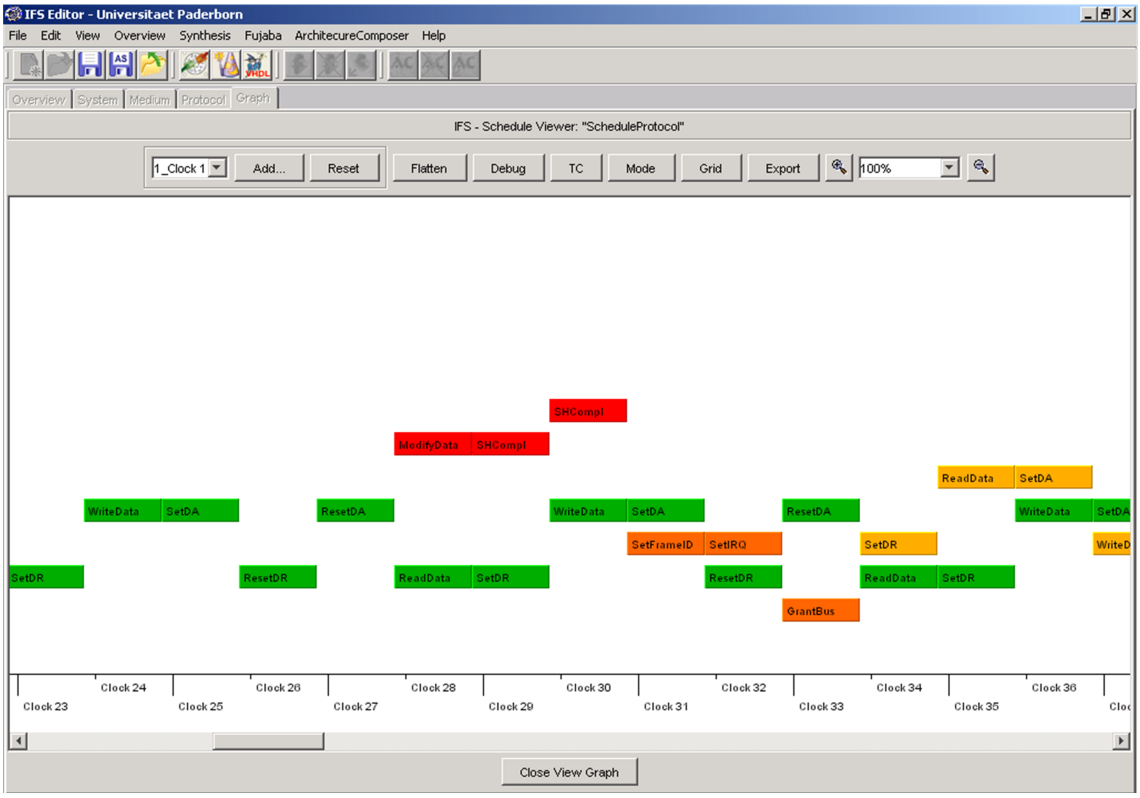

Abbildung 3.10: framebased Schedule mit 2 Subframes mit vier Bits

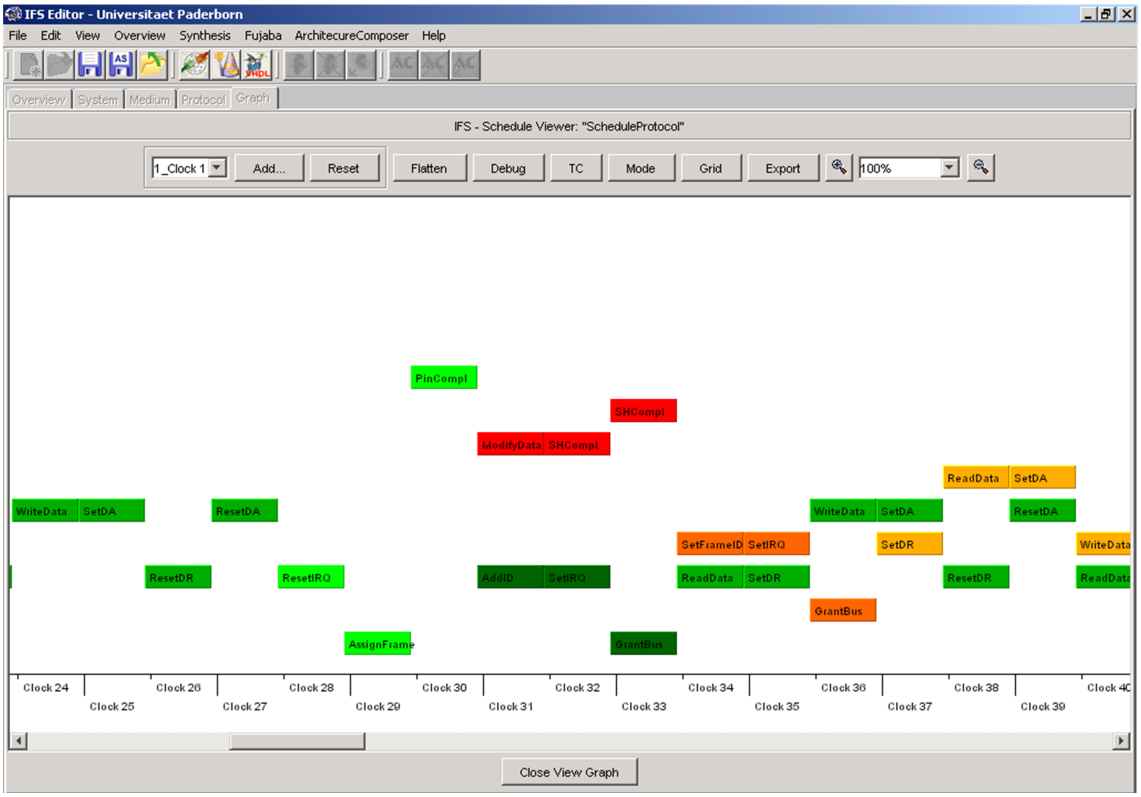

Abbildung 3.11: subframebased Schedule mit 2 Subframes mit vier Bits

Der subframebased Schedule ist in diesem Fall immer noch besser als der simple Schedule (um 10%), aber er benötigt 12 Takte mehr als der framebased Schedule.

Wird die Größe der Subframes und der Instanzen auf jeweils ein Bit reduziert, ergeben sich noch deutlichere Unterschiede. Das framebased Schedule benötigt für das Einlesen der acht Subframes und dem Senden der acht Instanzen 118 Takte. Gegenüber dem simple Schedule ist das eine Verbesserung von 28%. Der subframebased Schedule hingegen benötigt für die gleiche Anzahl Subframes und Instanzen 205 Takte. Er ist also um 39 Takte (das entspricht 24%) langsamer als der simple Schedule.

Folgende Tabelle stellt diese und weitere Werte gegenüber.

Verfahren	8 Bits	2x4 Bits	4x2 Bits	8x1 Bit
simple Schedule	166	–	–	–
framebased Schedule	166	139	124	118
subframebased Schedule	166	151	199	205

Tabelle 3.1: Vergleich der Ausführungszeiten der Optimierungsverfahren in IFB-Takten für einen Frame mit 8 Bit

Das framebased Schedule erzielt also bessere Ergebnisse als das simple Schedule. Je kleiner die Subframes und die Instanzen, desto schneller können sie abgearbeitet werden. Diese Verbesserungen erfordern aber eine Modifikation des IFB (siehe Kapitel 3.3.1).

Das subframebased Schedule erzielt nur bis zu einer Subframegröße von vier Bits ein besseres Ergebnis als der simple Schedule. Sind die Subframes kleiner, benötigt das Durchlaufen der zusätzlichen `Establish` und `Release` Phasen benötigt mehr Zeit, als durch die Verbesserung gewonnen werden kann. Eine Modifikation des IFB ist in diesem Scheduling nicht erforderlich, allerdings sind die Verbesserungen geringer als im framebased Schedule. Betrachtet man große Input-Frames, deren Daten auf wenige Gruppen abgebildet werden, so ist das Ergebnis nur geringfügig schlechter als das des framebased Schedules. Ein Beispiel mit 1024 Input Bits ist in Tabelle 3.2 dargestellt und belegt diesen Effekt.

Verfahren	1024 Bits	2x512 Bits	4x256 Bits	8x128 Bit
simple Schedule	18.455	–	–	–
framebased Schedule	18.455	15.377	13.840	13.060
subframebased Schedule	18.455	15.391	13.878	13.150

Tabelle 3.2: Vergleich der Ausführungszeiten der Optimierungsverfahren in IFB-Takten für einen Frame mit 1024 Bit

Wie gezeigt wurde, kann durch beide Verfahren eine deutliche Verbesserung der Latenz erreicht werden. Die vorgezogene Verarbeitung der Daten ist somit ein effektiver Optimierungsansatz. Die Ergebnisse beider Verfahren hängen dabei von dem Anwendungsszenario ab.

4 Rekonfigurierte Ausführung

Im vorigen Kapitel wurde gezeigt, wie durch geschicktes Pipelining im IFB die Latenz des IFBs reduziert werden konnte. Dies geschah unter der Annahme, dass alle Modes, die benötigt werden, immer auf dem Chip vorhanden sind. Geht man nun davon aus, dass ein IFB mit vielen Tasks verbunden ist und viele Mapping Equations implementiert, so werden entsprechend viele Modes benötigt. Somit benötigt die Implementierung des IFB viel Platz. Der Platz stellt auf einem Chip aber eine teure Ressource dar und Chips können nicht beliebig gross konstruiert werden. Die damit verbundenen Kosten und die Größe der Chips würden enorm steigen, was in dem meisten Fällen unerwünscht ist. Man möchte eher "kleine" Chips verwenden.

Hier setzt der zweite Optimierungsansatz an. Es wird davon ausgegangen, dass nicht ausreichend Platz für alle Modes vorhanden ist, die von einem IFB benötigt werden. Somit ergibt sich die Notwendigkeit, Modes zu verdrängen und andere Modes zu laden. Das wiederum erfordert Hardware, die zur Laufzeit rekonfiguriert werden kann. Es werden also multifunktionale Ausführungseinheiten benötigt, in die dynamisch die benötigten Komponenten geladen und dann ausgeführt werden können. Dazu wird ein FPGA verwendet.

Wie bereits im Abschnitt 2.1 kurz erläutert, besteht das FPGA aus *slices*, die wiederum gruppiert und zu Slots zusammengefasst werden können (siehe Abbildung 2.1). Diese Slots sind dynamisch rekonfigurierbar, d.h., dass zur Laufzeit vorhandene Modes dadurch verdrängt werden, indem neue Modes in diese Slots geladen werden. Hierbei unterscheidet man zwischen *multi reconfigurable* (es können mehrere Modes gleichzeitig geladen werden) und *single reconfigurable* (es kann nur ein Mode gleichzeitig geladen werden) FPGAs.

Zur Rekonfigurierung wird eine Kontrollkomponente benötigt, die den Vorgang der Rekonfigurierung ausführt und kontrolliert. Denn bevor eine Rekonfigurierung ausgeführt werden kann, muss die Einhaltung bestimmter Bedingungen sichergestellt werden. Es ergeben sich folgende Aufgaben für die Kontrolleinheit:

- Auswahl des Modes, der verdrängt wird
- Sicherstellung, dass zu Beginn der Rekonfigurierung kein Mode aktiv ist und somit keine Kommunikation oder Verarbeitung im Gange ist
- Das korrekte Einbinden des neuen Modes in den Slot (u.a. die Verbindungen)
- Das Anpassen des Interfaces an die neuen Funktionalitäten (wenn nötig)
- Das Sicherstellen eines reibungslosen Ablaufes ohne Beeinflussung der anderen Modes (ohne dass bei diesen Datenverlust auftritt)

Diese Kontrollkomponente ist für das FPGA notwendig und darf nicht ausgetauscht werden. Deswegen ist sie auf dem *fixed slot* implementiert. Dieser Slot ist nicht rekonfigurierbar, d.h. diese Slots können nicht durch andere Funktionalitäten überschrieben werden.

Im Falle des IFBs werden auf dem nicht rekonfigurierbaren Slot neben der Kontrolleinheit (der *Reconfiguration Unit*) weitere Komponenten untergebracht, die für die Funktionalität des IFBs unabdingbar sind und somit nicht ausgetauscht werden dürfen:

Die **Control Unit** mit ihren Unterkomponenten (dazu gehört die *Reconfiguration* Einheit) sind wichtige Steuerinstrumente zur Kontrolle des IFBs.

Der **Speicher** (Data Reader und Data Writer) dient zur Zwischenablage der Daten aus den und für die Modes. Alle Komponenten müssen darauf zugreifen können. Ein Austausch des Speichers würde somit zu Datenverlust führen.

Auf dem festen Slot sind außerdem die **Verbindungsleitungen** zu den externen Tasks untergebracht. Letztere sind nicht direkt mit den Slots verbunden, in denen die PH_{Modes} geladen sind, sondern die Verbindung läuft erst über diesen nicht austauschbaren Slot. Von hier aus gehen die Leitungen zum PH_{Mode}. Dies hat den Vorteil, dass das Interface für die externen Tasks immer das gleiche ist und ein Mode in einen beliebigen Slot geladen werden kann. Somit ist eine Anpassung der Tasks bzw. des FPGAs unnötig (und auch nicht wünschenswert), denn der IFB soll für die Tasks ja transparent bleiben.

Somit ergibt sich die bereits bekannte Abbildung 2.1.

Um die Rekonfigurierung von Modes zu beschreiben, wurde das bisher verwendete Modell erweitert. Grundlage war das I-P-O Schema. Es umfasste nur das Ausführen der Input Stage, der Processing Stage und der Output Stage. Ist eine dieser Stages aber nicht vorhanden, muss sie erst in einen Slot geladen werden. Also muss für jede der drei Stufen noch eine weitere Stufe hinzugefügt werden, in der die entsprechende Komponente in einen Slot geladen wird. Es ergibt sich das folgende Schema:

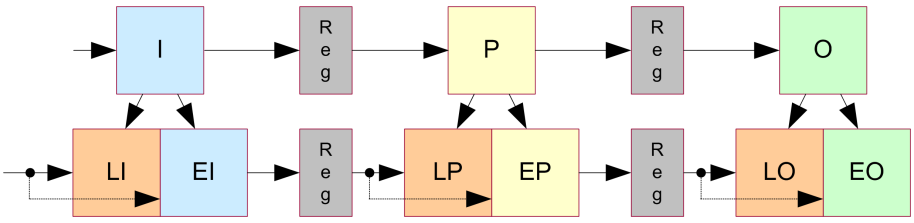

Abbildung 4.1: Das neue I-P-O Schema mit sechs Stufen

Abbildung 4.1 zeigt das daraus resultierende Schema für die Ausführungs-Pipeline. Die bereits bekannten Stufen wurden in eine *Load* Phase und eine *Execute* Phase aufgeteilt. Zu beachten ist, dass diese neuen Stufen durchaus übersprungen werden können (durch den Pfeil gekennzeichnet, der durch die *Load* Phasen geht). Ist eine Komponente bereits geladen, ist es unnötig, diese nochmals zu laden.

Wie bereits in der Einleitung erwähnt, ist dieser Ansatz nicht mit der Datenflussoptimierung kombinierbar. Dieser betrachtet alle benötigten Modes als gegeben an, d.h. sie sind von alle auf dem FPGA geladen und können ohne Verzögerung ausgeführt werden.

Das Laden einer Stage wurde somit nicht berücksichtigt. Diese Optimierung kann also genutzt werden, wenn die Chips, auf denen ein IFB implementiert wird, genügend groß sind, so dass tatsächlich alle Modes bereits zu Beginn geladen werden können. Benötigt der IFB aber viele Modes (da viele Tasks an ihn angeschlossen sind), für die nicht ausreichend Slots zur Verfügung stehen, kann die Flächenoptimierung genutzt werden. Diese Optimierung sorgt für eine optimale Ausnutzung (Utilization) der Slots, so dass Stages frühestmöglich und in optimaler Reihenfolge geladen werden.

Nachdem das erweiterte Modell eingeführt und erklärt wurde, wird gezeigt, wann welche Stage ausgeführt werden kann und welche Bedingungen eingehalten werden müssen, um einen korrekten Ablauf sicherzustellen.

4.1 I-P-O Scheduling eines rekonfigurierbaren IFB

Die grundlegende Idee beim Scheduling des erweiterten I-P-O Schemas ist das Caching. Hat bspw. ein PH_{Mode} die Daten einer Task gelesen, so wird dieser Mode nicht sofort verdrängt. Er bleibt vorerst geladen, denn dieser Mode wird im nächsten Kommunikationszyklus wieder benötigt. Der Mode bleibt also so lange in einem Slot geladen, bis er von einem anderen verdrängt werden muss. Ein Beispiel soll dies verdeutlichen:

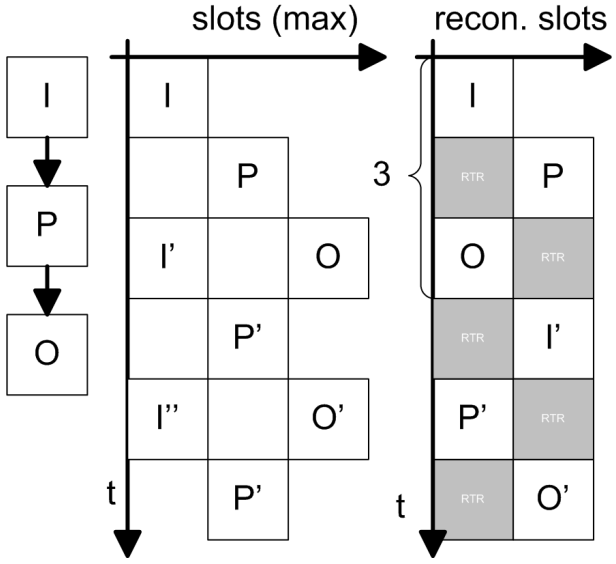

Abbildung 4.2: Scheduling mit beliebig vielen Slots (links) und zwei Slots (rechts) [3].

Im linken Teil von Abbildung 4.2 stehen die maximal benötigte Anzahl an Slots (in diesem Fall drei) zur Verfügung, so dass die Ausführung des IFBs ohne unnötige Verzögerungen stattfindet. Das ist dadurch möglich, dass alle Slots parallel geladen werden und ein Nachladen zur Laufzeit nicht notwendig ist.

Im rechten Teil wird deutlich, dass weniger Slots die Ausführung des IFBs verzögern, da Stages zur Laufzeit nachgeladen werden müssen (RTR = ***R**un**t**ime **R**econfiguration*). Dies führt zu einem Trade-Off zwischen Rechenzeit und benötigtem Platz. Im dritten

Takt kann daher nur die Output Stage des ersten Kommunikationszyklus ausgeführt werden. Im anderen Beispiel war es möglich, parallel zur Output Stage die Input Stage für den nächsten Kommunikationszyklus im zweiten Takt zu laden und im dritten Takt auszuführen. Um eine Verdrängung durchzuführen, wird eine Verdrängungsstrategie benötigt, die im Folgenden erklärt wird [2]:

1. grundsätzlich werden nur Modes verdrängt, die nicht aktiv sind
2. existieren keine inaktive Modes, wird die Verdrängung verzögert
3. existiert mindestens ein inaktiver Mode, so werden folgende Fälle unterschieden:
   a) ist genau ein Mode inaktiv, wird dieser ersetzt
   b) sind mehrere Modes inaktiv, wird der ersetzt, der am weitesten entfernt und von dem verdrängenden Mode in seiner Ausführung abhängig ist
   c) existiert kein abhängiger Mode, so wird einer der Modes mit der größten Entfernung verdrängt

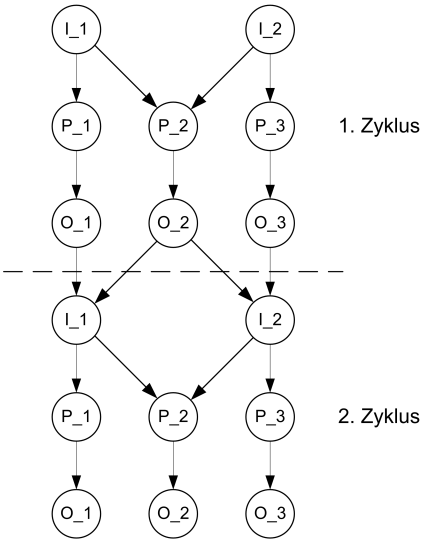

Abbildung 4.3: Die Stages zweier Kommunikationszyklen sowie deren Abhängigkeiten

Diese Verdrängungsstrategie ist optimal. Die Grundlage des Verfahrens ist aus der Caching Theorie abgeleitet, die besagt, dass idealerweise das am längsten nicht mehr benötigte Objekt zu verdrängen ist. Der Graph in Abbildung 4.3 zeigt beispielhaft zwei Kommunkationszyklen. Die Knoten des Graphen repräsentieren die Stages, die ausgeführt werden müssen. Der Abstand zwischen zwei Stages lässt sich in Form der vertikalen Distanz zwischen der Stage, die platziert werden soll, und der Stage, die in einem Slot allokiert ist, messen. Das folgende Kapitel enthält einige Ergebnisse zu dieser Strategie.

4.2 Ergebnisse

Um die Effektivität der oben präsentierten Verdrängungsstrategie aufzuzeigen, wurden verschiedene Beispiele im IFS-Editor simuliert.

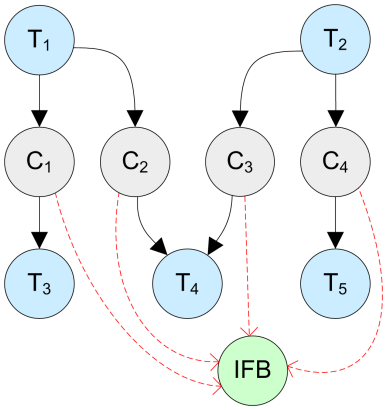

Abbildung 4.4: Der Kommunikationsgraph aus der Einleitung

In Abbildung 4.4 wurde der bereits bekannte Kommunikationsgraph nochmals gezeigt. In diesem Beispiel existieren zwei sendende und drei empfangende Tasks. Zwischen den Tasks sind die Abbildungsregeln, die auf den IFB abgebildet werden. Die folgenden Abbildungen zeigen beispielhaft das I-P-O Scheduling des in Abbildung 4.4 abgebildeten Szenarios. Dabei wurde ein FPGA mit vier Slots simuliert und der Kommunikationszyklus zweimal ausgeführt.

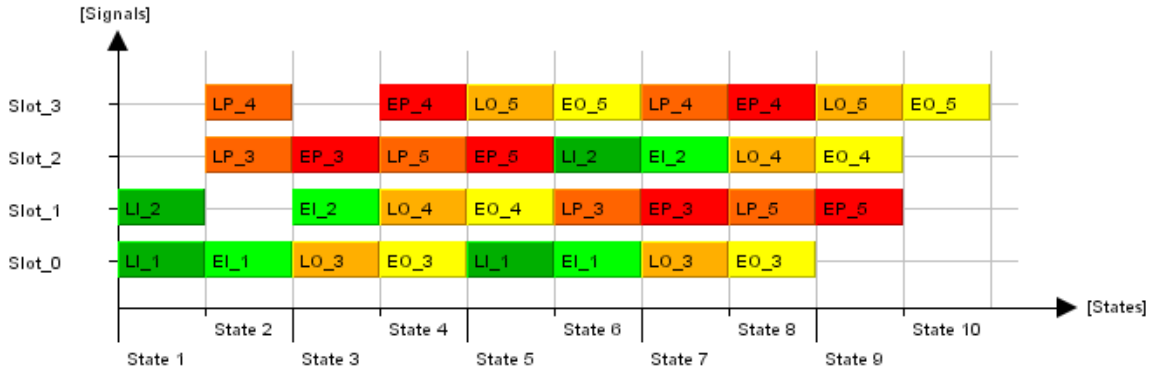

Abbildung 4.5: Zwei Kommunikationszyklen auf einem multi reconfigurable FPGA

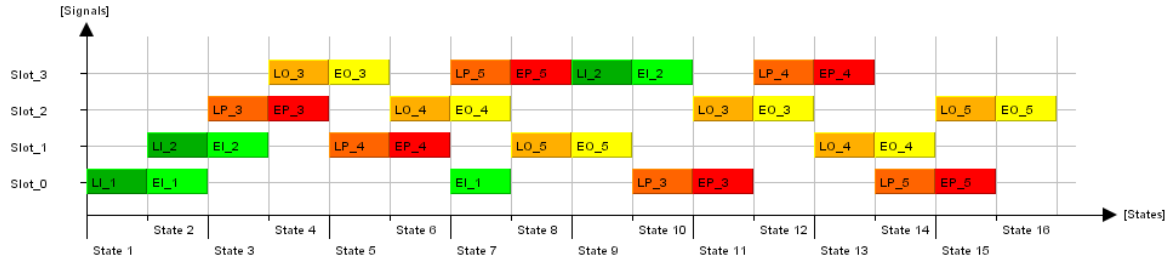

Abbildung 4.6: Zwei Kommunikationszyklen auf einem single reconfigurable FPGA

Abbildung 4.5 zeigt dabei die Ausführung auf einem multi reconfigurable FPGA. In Abbildung 4.6 wurde ein single reconfigurable FPGA verwendet. Für das Beispiel wurden vier Slots verwendet, weil hier die Unterschiede zwischen den beiden Varianten besonders deutlich hervortreten. So benötigt das multi reconfigurable FPGA sechs Takte weniger und weist dabei eine höhere Utilization (siehe Gleichung 4.1) auf.

Im Folgenden werden drei repräsentative Szenarien betrachtet, die eine qualitative Bewertung des vorgestellten Pipelining-Verfahrens erlaubt:

- 1x Input, 3x Output (ein sendender und drei empfangende Tasks)
- 2x Input, 3x Output (siehe Abbildung 4.4)
- 3x Input, 1x Output

Die folgenden drei Abbildungen zeigen die Ergebnisse der berechneten Schedulings der drei Szenarien. Dabei werden sowohl die Anzahl der benötigten Schritte (*Pipeline Clocks*) als auch die Flächenausnutzung auf dem Chip(*Utilization*) für multi reconfigurable und single reconfigurable FPGAs visualisiert. Die Utilitzation berechnet sich mit der Formel

$$Utilization = \frac{\sum_{states} \texttt{Anzahl aktiver Slots}}{\texttt{Anzahl aller Slots} \cdot \texttt{Anzahl der Slots}} \tag{4.1}$$

Jedes Szenario wurde erst mit einem Slot, dann mit zwei Slots, usw. ausgeführt. Sobald in einem Scheduling die Anzahl der Slots größer oder gleich der Anzahl der zu ladenden Stages ist, kann keine Verbesserung der Pipeline clocks erzielt werden. Daher war sind die dargestellten Diagramme auf zehn Slots begrenzt. Zusätzliche Slots würden nur zu einer Verschlechterung der Utilization führen.

Jedes dieser Szenarien wird zehn Mal hintereinander ausgeführt, es ergeben sich also zehn Kommunikationszyklen.

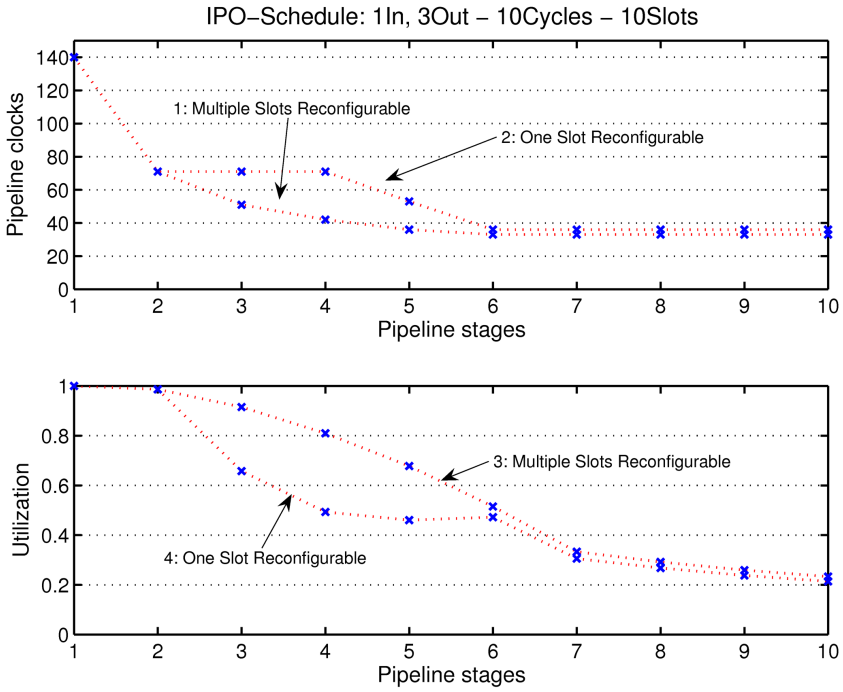

Abbildung 4.7: IPO-Schedule: Ein sendender und drei empfangende Tasks

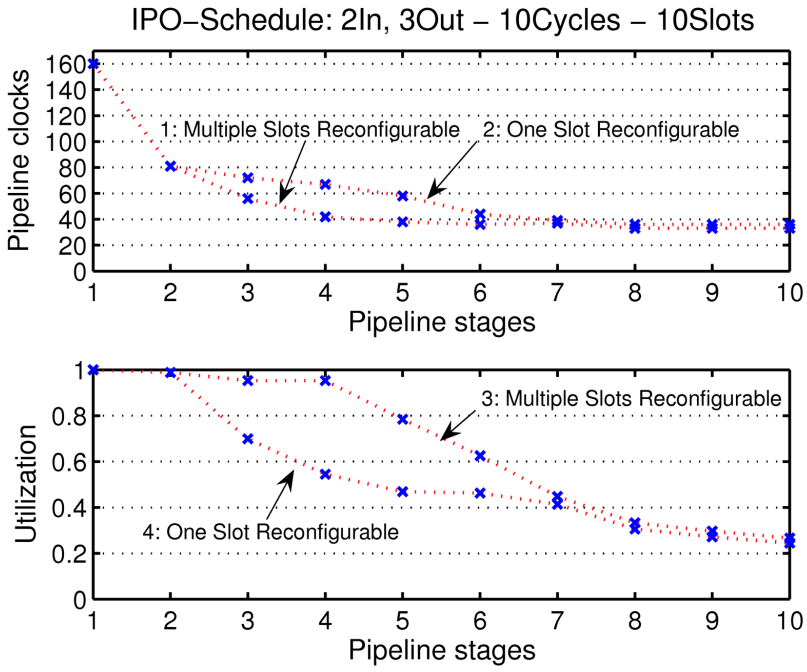

Abbildung 4.8: IPO-Schedule: Zwei sendende und drei empfangende Tasks

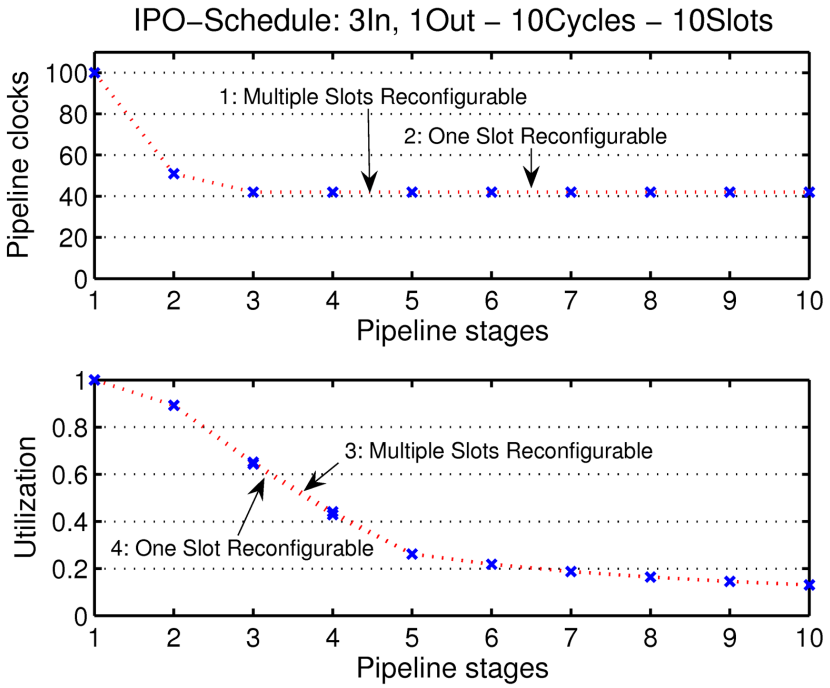

Abbildung 4.9: IPO-Schedule: Drei sendende und ein empfangender Tasks

In den ersten beiden Szenarien (Abbildungen 4.7 und 4.8) ist deutlich zu erkennen, dass es von Vorteil ist, wenn mehrere Slots gleichzeitig rekonfiguriert werden können. Somit wird sowohl die Zahl der benötigten Schritte reduziert als auch der Platz optimaler ausgenutzt (es werden sind also mehr Stages pro Pipeline Clock geladen). Diese Unterschiede treten hauptsächlich auf, wenn zwei bis sechs Slots zur Verfügung standen, um die Pipeline Stages zu laden. Mit mehr oder weniger Slots weichen die erzielten Ergebnisse nicht gravierend voneinander ab. Es ist also von Vorteil, mit zwei bis sechs verfügbaren Slots einen multi reconfigurable FPGA zu verwenden.

In Abbildung 4.9 zeigen sich keine deutlichen Unterschiede zwischen beiden Versionen. In diesem speziellen, sehr einfachen Beispiel spielt es keine Rolle, ob mehrere Slots gleichzeitig rekonfiguriert werden können.

Vergleicht man die drei Szenarien hinsichtlich benötigter Zeit und Flächenausnutzung, so wird deutlich, dass je mehr Slots zur Verfügung stehen, die **Zeiten** verbessert werden können. Allerdings konvergieren die Zeiten für die ersten beiden Szenarion stetig gegen die minimale Anzahl von Pipeline Clocks, sobald mehr als sechs Slots verfügbar sind. Danach können keine weiteren Verbesserungen erzielt werden. Beim dritten Szenario (Abbildung 4.9) pendelt sich dieser Wert bei etwas mehr als 40 Zeiteinheiten ein. Die **Flächenausnutzung** dagegen nimmt unterschiedlich stark ab, erreicht aber bei allen drei Szenarion mit zehn Slots einen Wert zwischen 15% und 25%.

Zusammenfassend kann gesagt werden, dass multi reconfigurable FPGAs mit zwei bis vier Slots deutliche Vorteile besitzen. Damit können die Kommunikationszeiten reduziert

und die Fläche besser ausgenutzt werden, ohne die Kommunikationszeiten drastisch zu erhöhen. Voraussetzung für eine solche Optimierung ist, dass keine Real-Time Protokolle ausgeführt werden können, da sonst ggf. Deadlines verletzt werden könnten.

Diese Optimierung ermöglicht es, die benötigte Fläche für eine IFB-Implementierung zu reduzieren. Das erlaubt die Verwendung kleinerer Chips, was wiederum die Kosten des Designs senkt. Die Rekonfigurierungseigenschaften des Chips beeinflussen dabei die erreichbaren Verbesserungen. Plattformen, auf denen mehrere Teilschaltungen gleichzeitig rekonfiguriert werden können, bieten Vorteile gegenüber solchen Plattformen, die nur die Rekonfigurierung eines Teilstücks unterstützen.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

In dieser Arbeit wurden zwei Verfahren präsentiert, um die Ausführung des IFBs zu beschleunigen bzw. zu optimieren.

Das erste Verfahren zielte auf die Reduzierung der Latenz durch Analyse der eingelesenen Daten. Die Verarbeitung wurde nicht erst nach dem Einlesen *aller* Daten gestartet, sondern ein Teil der Daten konnte bereits vorher modifiziert werden. Dazu wurden die Subframes eingeführt, die auf zwei unterschiedlichen Arten in den IFB integriert wurden. Das erste Verfahren (das framebased Schedule) erwies sich als sehr effizient. Je kleiner die Subframes und die Instanzen wurden, desto schneller konnten die einzelnen Stages durchlaufen und beendet werden. Das zweite Verfahren (subframebased Schedule) stellte erwies sich ebenfalls als effizient. Nur für Subframegrößen unter vier Bit wurden schlechtere Ergebnisse erzielt, was keine wesentliche Einschränkung darstellt.

Das zweite Optimierungsverfahren beschreibt die Implementierung einer effektiven Verdrängungsstrategie, um die benötigte Fläche für die Implementierung eines IFB zu minimieren. Dazu werden die einzelnen Stages des I-P-O Schemas in `Load Stage` und `Execute Stage` aufgeteilt und eine Verdrängungsstrategie verwendet, die diese Aufteilung berücksichtigt. In Kapitel 4.2 wurde gezeigt, dass ein FPGA die besseren Ergebnisse erzielt, wenn mehrere Slots gleichzeitig rekonfiguriert werden können. Außerdem ist diese Strategie dann am effektivsten, wenn weniger Sender als Empfänger vorhanden sind.

Beide Strategien sind aus den bereits dargelegten Gründen nicht miteinander kombinierbar. Das bedeutet, dass man eine Entscheidung hinsichtlich der Bedeutung von Latenz und Fläche treffen muss. Ist eine niedrige Latenz von hoher Bedeutung, so sollte das framebased Schedule implementiert werden. Verfügt man aber nur über wenige Slots (ist also der FPGA relativ klein und kann nicht alle Modes gleichzeitig speichern), so sollte man die Flächenoptimierung mit der Verdrängungsstrategie aus Kapitel 4.1 verwenden.

5.2 Ausblick

In dieser Arbeit wurde für die Datenflussoptimierung nur der einfache Fall mit einer Input Stage, einer Processing Stage und einer Output Stage berücksichtigt. Ein mögliches Szenario besteht darin, dass eine Processing Stage Daten von *zwei* Input Stages benötigt. Das Scheduling für einen solchen Fall könnte durch eine Erweiterung der Scheduling Algorithmen implementiert werden.

Literaturverzeichnis

[1] D. Averberg. Synthese von deadline-konformen protokollkonvertern für heterogene verteilte anwendungen. Studienarbeit, Universität Paderborn, März 2005.

[2] Hennessy, John L. and Patterson, David A. and Goldberg, David und Asanovic, Krste. *Computer Architecture*. Morgan Kaufmann, June 2002.

[3] Ihmor, Stefan and Dittmann, Florian. Optimizing Interface Implementation Costs Using Runtime Reconfigurable Systems . In , 2005.

[4] Ihmor, Stefan and Hardt, Wolfram. Runtime Reconfigurable Interfaces - The RTR-IFB Approach. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW'04)*. IEEE Computer Society, 26 - 27 April 2004.

[5] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. A Consistent Design Methodology for Configurable HW/SW-Interfaces in Embedded Systems. Montreal, Canada, Aug. 2002.

[6] Ihmor, Stefan and Visarius, Markus and Hardt, Wolfram. Modeling of Configurable HW/SW-Interfaces. In *RSS2003*, pages 51 – 60, 2003.

[7] T. Loke. Synthese von kommunikationsstrukturen in eingebetteten systemen. Studienarbeit, Universität Paderborn, März 2005.