Fachhochschule
Dortmund
University of Applied Sciences and Arts

FB3 - Energiesysteme

Masterarbeit

Entwurf von seriellen Schnittstellen zur Konfiguration und Test integrierter Schaltkreise

Design of serial interfaces for configuration and test of integrated circuits

Achraf Drissi El Bouzaidi

Erstprüfer: Prof. Dr. Michael Karagounis

Zweitprüfer: Herr Alexander Walsemann

Dortmund, den 30.03.2023

Inhaltsverzeichnis

Eigenständigkeitserklärung

„Hiermit versichere ich, dass die von mir vorgelegte Prüfungsleistung selbständig und ohne unzulässige fremde Hilfe erstellt worden ist. Alle verwendeten Quellen sind in der Arbeit so aufgeführt, dass Art und Umfang der Verwendung nachvollziehbar sind.“

Dortmund 30.03.2023

Ort/Datum **Achraf Drissi El Bouzaidi**

KURZZUSAMMENFASSUNG

Diese Masterarbeit befasst sich mit der Entwicklung von serieller Schnittstelle zur Konfiguration und Überprüfung von integrierten Schaltungen. Das Projekt behandelt zum einen die Umsetzung eines I2C-Master-Interfaces in Verilog und die Optimiereung und Erweiterung der Schaltung. Der Hauptfokus liegt jedoch auf der Implementierung des JTAG (Joint Test Action Group) Protokolls in Verilog.

Der Bericht gliedert sich in zwei Teile. Der erste Teil befasst sich mit den grundlegenden Funktionen des I2C-Master gemäß der NXP-UM10204 Spezifikation. Hier wird dargestellt, wie die Grundschaltung implementiert wurde und wie die implementierten Module genutzt werden können. Der Hauptbestandteil beschäftigt sich mit den grundlegenden Konzepten des JTAG-Standards und seiner praktischen Anwendung. Es wird demonstriert, wie das JTAG-Protokoll in Verilog umgesetzt wurde und wie es zur Überprüfung und Konfiguration des Zustands eines integrierten Schaltkreises genutzt werden kann. Der Bericht schließt mit der Simulation von Testfällen und einer Zusammenfassung der Ergebnisse.

ABSTRACT

This master thesis examines the design of a serial interface for configuring and monitoring integrated circuits. The project deals with the implementation of an I2C master interface in Verilog and the optimization and extension of the circuit. The main focus however, ist he implementation of the Joint Test Action Group (JTAG) protocol in Verilog.

The report is divided into two parts. The first part deals with the basic functions of the I2C-master according to the NXP-UM10204 specification. Here, the basic circuit is shown and how the implemented modules can be used. The main part deals with the basic concepts of the JTAG standard and its practical application. It demonstrates how the JTAG protocol was implemented in Verilog and how it can be used to test and configure the state of an integrated circuit. The report concludes with the simulation of test cases and a summary of the results.

1. Einleitung

Serielle Schnittstellen werden verwendet, um integrierte Schaltkreise (ICs) zu konfigurieren und zu testen. Diese Schnittstellen ermöglichen es, Daten zwischen dem IC und einem externen Gerät, wie einem Computer oder einem anderen IC, zu übertragen.

Der serielle Datentransfer kann über ein Kabel oder drahtlos über eine Funkverbindung erfolgen. Das für die Kommunikation verwendete Protokoll hängt von der spezifischen Anwendung ab. Einige der gängigen seriellen Schnittstellenprotokolle sind RS-232, RS-485,I^2C, SPI und JTAG.

Die Verwendung von seriellen Schnittstellen zur Konfiguration und Test von ICs ermöglicht es, dass Änderungen an den ICs vorgenommen werden können, ohne dass diese physisch aus dem System entfernt werden müssen, in dem sie integriert sind. Dies kann bei der Fehlerbehebung und der Optimierung von Schaltkreisen sehr nützlich sein.

Insgesamt sind serielle Schnittstellen ein wichtiges Werkzeug für den Umgang mit integrierten Schaltkreisen, da sie es ermöglichen, diese zu programmieren, zu testen und zu verwalten.

Joint Test Action Group (JTAG) ist ein Standard, der vom Institute of Electrical and Electronics Engineers (IEEE) entwickelt wurde und in der Regel als IEEE-Standard 1149.1 bezeichnet wird. JTAG ermöglicht den Zugriff auf die spezielle integrierte Logik von integrierten Schaltungen (ICs), die in vielen elektronischen Geräten verwendet werden.

Über JTAG können verschiedene Funktionen aufgerufen werden, darunter Testlogik zum Testen von Verbindungen zwischen Bausteinen ohne externe Zugriffe, Programmierlogik in CPLDs und FPGAs zur On-Board-Programmierung dieser Bausteine und Debug-Logik in Mikroprozessoren und Mikrocontrollern, die für das Software-Debugging verwendet werden. JTAG kann auch verwendet werden, um Verbindungen mit Peripheriebausteinen mit hoher Geschwindigkeit zu testen, ohne die im Baustein vorhandene Software zu verwenden, oder um den integrierten Speicher in einem Mikrocontroller zu programmieren. [1]

Als Beispiel ist in Abb.1 eine JTAG-Kette mit mehreren Geräten dargestellt, welche eine Anordnung von Schaltkreisen dargestellt. Im Falle eines PC Computers könnten die Schaltkreise die CPU, das RAM, das BIOS und andere Komponenten des Computers umfassen. Die Schaltkreise werden über die JTAG-Schnittstelle miteinander verbunden und können über eine Testumgebung überprüft werden, um sicherzustellen, dass sie richtig funktionieren.

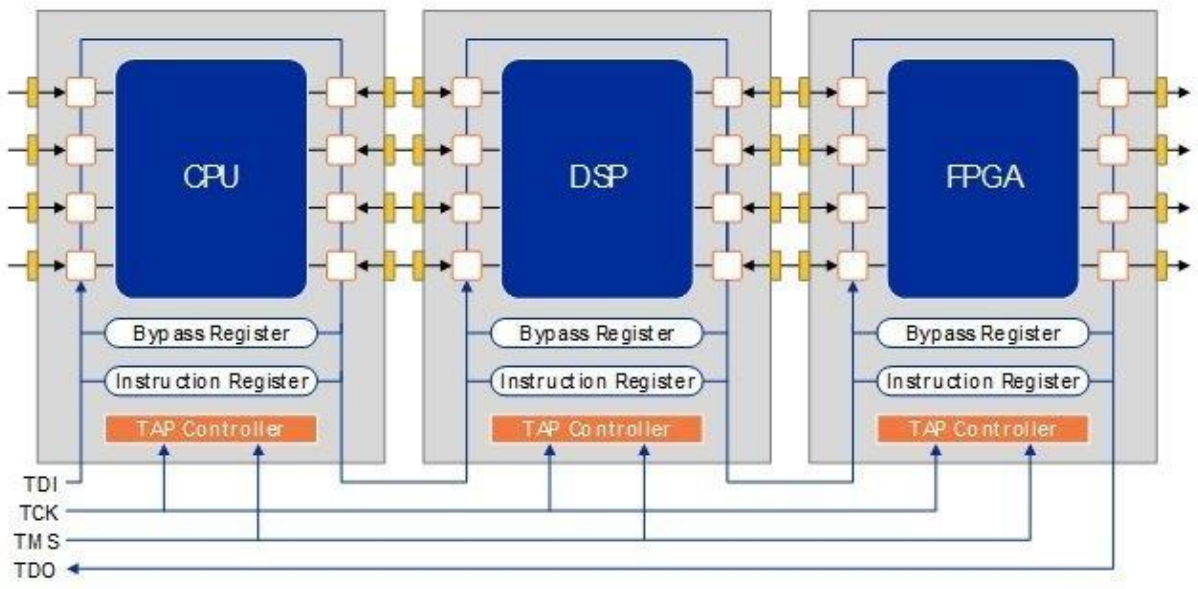

Abbildung 1: Beispiel für eine JTAG-Kette mit mehreren Geräten [1]

Bausteine mit einer JTAG/Boundary-Scan-Schnittstelle und -Logik sind auf vielen heutigen elektronischen Baugruppen (PCBs) vorhanden. Diese Bausteine werden oft seriell (in Form einer Daisy Chain) zu einer sogenannten "Scan-Kette" auf der Baugruppe verbunden. Ein externer JTAG/Boundary-Scan-Controller wird verwendet, um die Logik anzusteuern und Tests der Baugruppe sowie die Programmierung der Bausteine durchzuführen.

Der Einsatz von JTAG/Boundary-Scan minimiert den Umfang und die Komplexität des erforderlichen Equipments zum Testen und Programmieren der Baugruppe in Entwicklung, Fertigung und Service, da die Notwendigkeit externer Kontaktierung und komplexer Adaptionen entfällt. Auf diese Weise können Zeit und Kosten für den Bordtest und die Programmierung eingespart werden.

Zusammenfassend lässt sich sagen, dass JTAG eine nützliche Technologie ist, die den Test und die Programmierung von elektronischen Baugruppen erleichtert und beschleunigt.

In diesem Projekt wird ein JTAG-Interface in Verilog implementiert. Der IEEE Standard 1149.1 wird hierbei als Referenz verwendet, um die Funktion der JTAG Schnittstelle nachzuvollziehen und die korrekte Implementation sicherzustellen.

Dieser Standard beschreibt verschiedene Betriebsarten von JTAG und bietet eine umfassende Einführung in die Datenübertragung, das Boundary Scan und die Standard-Zustandsmaschine.

2. I²C-Protokoll

2.1 Elektrische Spezifikation

Der I²C -Bus benötigt neben der Versorgungsspannung zunächst nur zwei Verbindungen. Eine Verbindung entspricht der SCL (Serial Clock), die andere der SDA (Serial Data). Auf der SDA Leitung können Daten sowohl vom Master, als auch vom Slave gesendet und empfangen werden. Über die SCL Leitung wird das vom Master gesendete Taktsignal zur Synchronisation an die Slaves übermittelt. Die Busleitungen sind im Grundzustand HIGH. Der Zustand LOW kann durch jeden einzelnen Busteilnehmer erzeugt werden, indem dieser die Leitung auf Masse zieht. Dies wird erreicht, indem die Busleitungen über Pull-Up-Widerstände an die positive Spannungsversorgung angeschlossen werden. [2]

Die Bus-Spezifikation sieht für die Dimensionierung der Pegel zwei Varianten vor:

1. Fixed: Das Spannungsniveau für logisch Null soll im Intervall von -0,5 V bis 1,5 V liegen. Logisch Eins entspricht dann einem Pegel von mindesten 3V. In diesem Fall müssen die beiden Pull-Up-Widerstände an einer Spannung von 5 V liegen (±10%). So lassen sich auch Teilnehmer mit anderer Versorgungsspannung einfach anschließen.
2. Related: Das Spannungsniveau für logisch Null soll im Intervall von -0,5V bis 0,3*VDD liegen. Logisch Eins entspricht dann einem Pegel von mindestens 0,5*VDD V.

Beide Varianten lassen sich auch kombinieren Die Werte für die jeweiligen Pull- Up-Widerstände berechnen sich unter Berücksichtigung:

1. der Versorgungsspannung
2. der Anzahl der teilnehmenden Instanzen und
3. der resultierenden Kapazität der Busleitung.

Kennlinien dazu finden sich im Spezifikationsdokument [NXP-UM10204].

2.2 Datenübertragung

Die Datenübertragung erfolgt in Byte-Paketen. Ein Transfer kann nur von einem Master initialisiert werden. Die Slaves Teilnehmer können in der drauf folgenden Kommunikation sowohl Empfänger als auch Sender sein. Der Master kann ebenfalls Sender oder Empfänger sein.

Der Empfänger muss nach jedem Paket eine Bestätigung zurücksenden, dass er die Daten korrekt empfangen hat. Dies geschieht, indem nach dem Senden des 8. Bits als 9.Bit vom Empfänger eine logische Null auf SDA gelegt wird. Dieser Vorgang wird als Acknowledge in der

Spezifikation bezeichnet. In Zeichnungen findet sich dazu häufig das Kürzel ACK. Erfolgt dies nicht, wird der Transfer beendet. In vielen Fällen teilt der Empfänger dadurch mit, dass momentan keine weiteren Daten mehr entgegengenommen werden können. Bevor jedoch Daten übertragen werden, muss zunächst jedem Bus-Teilnehmer signalisiert werden, dass ein Transfer initialisiert wird. Dies geschieht mit dem Senden einer Startbedingung. [2]

2.2.1 Datenvalidität

In der I2C-Spezifikation ist festgelegt, dass Daten nur dann gültig sind, wenn die Takt-Leitung SCL auf logisch Eins liegt. Während dieses Hoch-Pegels darf sich der Wert auf SDA nicht ändern andernfalls sind die Daten ungültig. Der Fall der Daten-Invalidität stellt jedoch gleichzeitig eine Start- oder Stopp-Bedingung dar. Daher erfährt die gesamte Logik einen Reset im Sinne der unten skizzierten Start- /Stopp-Bedingungen. Über die Definition dieser Bedingungen ist ein relativ einfacher Mechanismus zur impliziten Fehlererkennung vorhanden. Eine Fehlerbehebung ist in diesem Fall jedoch nicht möglich. [2]

2.2.2 Startbedingung

Die Übertragung beginnt immer mit der Signalisierung des Starts an alle Teilnehmer(siehe Abb.2). Eine Startbedingung tritt dann auf, wenn auf SDA ein Übergang von HIGH auf LOW stattfindet, während SCL auf HIGH ist. Jeder Busteilnehmer muss daraufhin seine Logik unabhängig von seinem vorherigen Zustand zurücksetzen. [2]

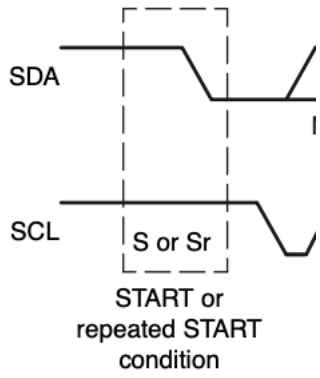

Abbildung 2 : Start-Bedingung [2]

2.2.3 Stoppbedingung

Die Stoppbedingung liegt dann vor, wenn auf SDA ein Übergang von LOW auf HIGH stattfindet, während SCL auf HIGH ist(siehe Abb.3). der Master kann Stoppsignal auf den BUS legen. Damit wird allen anderen Instanzen signalisiert, dass der Bus nun wieder frei ist. Alle am Bus angeschlossenen Instanzen müssen beim Empfang einer Stoppbedingung ihre Logik zurücksetzen. Die Slaves erwarten als nächstes eine Startbedingung. [2]

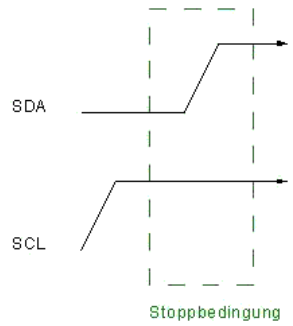

Abbildung 3: Stopp Bedingung [2]

2.3 Adressierung

Um zu ermitteln, an wen die Pakete gerichtet sind, besitzt jeder Teilnehmer eine eindeutige Adresse. Ursprünglich waren nur 7-Bits Adressen vorgesehen, was die Anzahl möglicher Teilnehmer theoretisch auf 127 begrenzte. Die Anzahl wurde allerdings durch reservierte Adressbereiche auf tatsächlich 112 Teilnehmer reduziert. Da dies in moderneren Systemen nicht ausreicht, erweiterte man das Protokoll in der Spezifikation 1.0 auf 10-Bit-Adressen. Damit können bis zu 1024 zusätzliche Adressen geschaffen werden. So wird die Anzahl der tatsächlich verfügbaren Adressen auf insgesamt 1136 erhöht. Die Adresse selbst wird immer am Anfang des Transfers übertragen. Unmittelbar nach der Startbedingung wird mit der nächsten High-Phase der Clock direkt das erste Bit der Adresse gesendet.

2.3.1 7-Bit-Adressierung

Die Adressierung von 7Bit Slaves erfolgt nach dem in Abb. 5 dargestellten Diagramm. Die Kommunikation wird von einem Master mit der Erzeugung einer Startbedingung eingeleitet. Daraufhin wird die 7-Bit lange Slave-Adresse versendet. Vervollständigt wird das erste Byte mit dem Read-Write-Bit.

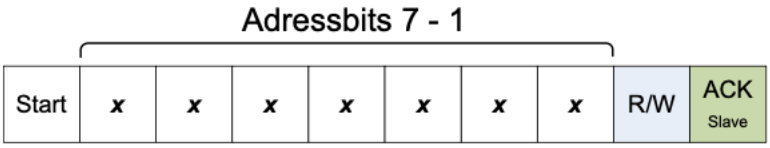

Abbildung 4: 7-Bits Adress-Register

An neunter Stelle folgt nun das Acknowledge-Bit, das in diesem Fall immer vom Empfänger gesetzt werden muss. Die Bits müssen wie bereits erwähnt vor der steigenden Flanke auf den Bus gelegt und bis nach der fallenden Flanke von SCL gehalten werden. Geschieht dies nicht,

würde dann eine Start-bzw. Stoppbedingung vorliegen. Ist der Empfang der korrekten Adresse bestätigt worden, folgt unmittelbar die Datenübertragung.

Für das Senden des Acknowledge-Bits der Daten-Pakete gibt es nun (in Abhängigkeit vom Read-Write-Bit) zwei Möglichkeiten:

1. Das Read-Write-Bit ist Null.

   In diesem Fall wartet der Master und damit der Initiator der Kommunikation auf das Acknowledge-Bit, das der Empfänger jedes Mal setzen muss.

2. Das Read-Write-Bit ist Eins.

   Durch das Vertauschen der Rollen von Sender und Empfänger muss jetzt der Master jedes Paket bestätigen.

In beiden Fällen gilt: Wird nicht bestätigt, gilt das zuletzt gesendete Paket für den Sender als nicht übermittelt. Der Sender stellt daraufhin sofort die Übermittlung von Daten ein. Jetzt darf die SDA-Leitung durch Slave-Instanzen nicht nach unten gezogen werden, damit auf dem Bus eine Stopp- oder wiederholte Start- Bedingung erzeugt werden kann. Würde keine der Bedingungen signalisiert, wäre der Bus bis auf Weiteres blockiert.

2.3.2 10-Bit-Adressierung

Bei der 10-Bit-Adressierung wird zunächst im 7-Bit Adressfeld ein 5 Bit breites reserviertes Datenwort versendet, welches die Übermittlung einer 10-Bit Adresse kenntlich macht. Danach folgen die ersten zwei Bits der 10-Bit-Slave- Adresse sowie das Read-Write-Bit. Erkennt mindestens ein Slave das reservierte 5-Bit Datenwort und stimmen die ersten zwei gesendeten Adressbits mit der eigenen Adresse überein, so wird das ACK vom Slave gesetzt. Als Nächstes wird von allen 10-Bit Instanzen, die ihre ersten zwei Adressbits erkannt haben, ein weiteres Adressbyte erwartet. [4]

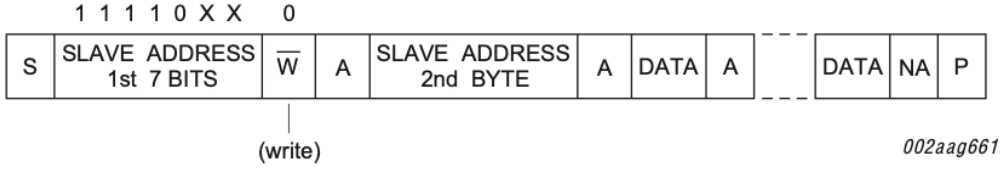

Abbildung 5: 10Bit Adressierung (Write)

Entspricht der Inhalt des zweiten Adressbytes zusammen mit den ersten zwei Adressbit einer vorhandenen Slave-Adresse, so setzt der adressierte Slave das 2. ACK wie in Abb.6 dargestellt. In Abb.5 ist ein Schreibvorgang und in Abb.6 ein Lesevorgang dargestellt. Beim Schreibvorgang ist es ähnlich wie bei der 7-Bit Adressierung. Nach der Übermittlung der Adresse und des R/W-Bits, folgt unmittelbar die Datenübertragung. Der Auslesevorgang von Daten mit 10-Bit

Adressierung ist jedoch einiges komplexer. Zunächst wird die 10-Bit Adresse mit einem R/W-Bit von Null übermittelt. Damit unterscheidet sich die Adressierung zunächst nicht von der eines Schreibvorgangs. Daraufhin folgt jedoch ein Repeated-Start (Abbildung 6) und es wird wieder das erste Byte der 10-Bit Adressierung verschickt, und zwar diesmal mit gesetztem R/W-Bit. Danach gibt der Master den Bus frei und der Slave setzt die Inhalte des Registers auf den Bus. Soweit der Master keine weiteren Daten braucht, setzt dieser ein NACK und beendet die Kommunikation mit einem Stopp-Signal. [4]

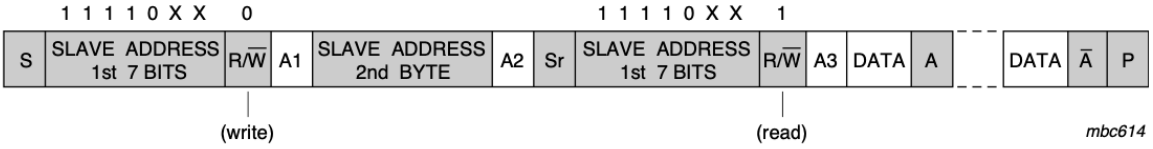

Abbildung 6: 10Bit Adressierung Read[2]

2.4 Synchronisation

Jeder Master erzeugt für den Kommunikationsvorgang auf SCL ein Taktsignal. Da mehrere Master am Bus beteiligt sein können, ergibt sich das Problem der Clock-Synchronisation. Auch das Taktungsintervall müsste in einem Multi-Master- System vorher festgelegt werden. Zur Lösung dieses Problems wird wieder die elektrische Spezifizierung ausgenutzt.

Der erste Teilnehmer, der seiner Clock-Periode folgend SCL auf null legt, zieht auch für alle anderen Master diese Leitung auf Null. Ab diesem Zeitpunkt zählt jeder Master die Zeit bis zur nächsten High-Periode herunter. Der erste Master, der diesen Zeitpunkt erreicht, versucht nun, SCL hochzuziehen. Hält jedoch ein anderer Master dies Leitung auf Null, kann der erste den Vorgang nicht erfolgreich abschließen und muss warten. In dieser Zeit befindet er sich in einem High-Wartezustand, bis der letzte Master die Leitung wieder auf Eins anhebt. Die Zeit, in der die Taktleitung den Wert Eins trägt, wird so durch den Master mit der kürzesten Periode vorgegeben. Umgekehrt verhält es sich mit der Periodenzeit des Wertes Null. Diese wird vom Master mit der größten Periodendauer vorgegeben.[2]

2.4.1 Clock-Stretching

Analog zur Verwendung bei der Synchronisation macht man sich die elektrische Spezifizierung zu Nutze, um Unterschiede in der Verarbeitungsgeschwindigkeit auszugleichen. Wird es während eines Übertragungsvorganges erforderlich, diesen für eine definierte Zeit zu unterbrechen, kann dies durch das Clock-Stretching erreicht werden. Dabei legt der Teilnehmer, der die Kommunikation unterbrechen möchte, die Taktleitung SCL so lange auf null, bis er weiterarbeiten kann.

2.5 Erweiterungen der Standardmodi

In diesem Abschnitt werden die weitere Ausbaumöglichkeiten des I2C-Busses angesprochen. Neben speziellen Adressierungsverfahren sind besonders zwei erweiterte Übertragungsmodi zu nennen, die zu einer erheblichen Steigerung der Datenübertragungsrate führen.

2.5.1 Der Fast-Mode

Ermöglicht der Standard-Mode Übertragungsgeschwindigkeiten von bis zu 100kBit/s, wird die Geschwindigkeit durch Verwendung des Fast-Mode auf bis zu 400 kBit/s heraufgesetzt.
Die Notwendigkeit einer solchen Erweiterung wurde bereits im Jahr 1992 in der Version 1.0 der I2C-Spezifikation erkannt und umgesetzt. Geräte, die den Fast- Mode unterstützen, sind immer auch zu Standard-Mode-Geräten kompatibel, können also mit diesen kommunizieren. Umgekehrt gilt dies nicht. Deswegen wird weitgehend darauf verzichtet, Standard-Mode- und Fast-Mode-Geräte zu kombinieren, gerade weil durch die höhere Geschwindigkeit undefinierte Zustände in Standard-Mode-Geräten auftreten können. Die Spezifikationen der Adressierung und Datenübertragung bleiben von der höheren Geschwindigkeit unberührt. Lediglich die elektrischen Spezifikationen müssen leicht abgewandelt werden. So ist nun ein Input-Filter vorgesehen, der mögliche Spikes herausfiltert. Außerdem darf in Fast-Mode-Geräten der Pegel der logischen Zustände nicht mehr fest vorgegeben werden, sondern muss in Relation zu einer Versorgungsspannung stehen.
Der Fast-Mode erfuhr mit der Spezifikation 3.0 im Jahr 2007 eine Überarbeitung. Es wurde der Fast-Mode-Plus (Fm+) eingeführt, der abwärtskompatibel zu Fast-Mode und Standard-Mode ist. Im Vergleich zum Fast-Mode hat sich die Geschwindigkeit im Fast-Mode-Plus verzehnfacht. Dazu benötigt er, im Gegensatz zum HS-Mode, keine zusätzlichen Leitungen. [2]

2.5.2 Der HS-Mode

Der HS-Mode ermöglicht Übertragungsgeschwindigkeiten von bis zu 3,4 MBit/s, benötigt dafür aber zwei zusätzliche Leitungen: SCLH und SDAH. Da gegenüber dem Standard-Mode eine um den Faktor 30 erhöhte Geschwindigkeit erreicht werden kann, mussten stringentere Regeln für erlaubte parasitäre Effekte auf den Leitungen eingeführt werden. Es wurde erforderlich, zwei zusätzliche Leitungen zu definieren, auf denen der Highspeed-Datentransferstattfindet. Die in Standard- und Fast-Mode-Systemen eingesetzten Leitungen SDA und SCL werden nur dann am HS-Mode-Gerät benötigt, wenn Instanzen beteiligt sind, die nur mit Standard- oder Fast-Mode betrieben werden können. Ansonsten kann jegliche Kommunikation über SCLH und SDAH stattfinden.
Damit ein HS-Mode-Transfer beginnen kann, sendet der HS-Master zunächst einen speziellen Mastercode. Dieser signalisiert allen Busteilnehmern, dass nach diesem Byte eine Highspeed-Kommunikation stattfindet. Aufgebaut ist dieses Byte aus vier Nullen und einer Eins, darauf folgen drei Bits, die den HS-Master identifizieren. Durch den Vorgang der Bus-Arbitrierung

kann dieser Code letztendlich nur von einem HS-Master komplett gesendet werden, der Master mit der niedrigsten Adresse ist dabei immer der Gewinner. Anschließend darf kein Busteilnehmer dieses Byte bestätigen, das Acknowledge-Bit bleibt auf Eins gesetzt. Unmittelbar danach wird die Kommunikation auf den Leitungen SCLH und SDAH fortgesetzt. Dabei folgen die Adressierung und Datenübertragung den Regeln im ursprünglichen I2C-System. [2]

2.5.3 Spezielle Adressierungsverfahren

Neben den vorgestellten Codes für die 10-Bit-Adressierung, den General-Call und den HS-Mode existieren noch eine Reihe weiterer Codes, die für spätere Erweiterungen reserviert sind. Sie dürfen nicht als reguläre Adresse verwendet werden. [2] Die reservierten Codes im ersten Paket sind in der folgenden Tabelle.1 aufgelistet, ein x steht dabei für frei wählbar:

Tabelle 1 : Weitere reservierte Codes im ersten Byte [2]

Erstes Byte	R/W	Bedeutung
0000000	1	Start-Byte
0000001	x	CBUS Adresse
0000010	x	Reserviert für andere Busformate
0000011	x	Reserviert für zukünftige Erweiterungen
11111xx	x	Reserviert für zukünftige Erweiterungen

2.5.4 Das Start-Byte

Das Start-Byte wurde für Mikrocontroller eingeführt, die nicht nur eine I2C Instanz beherbergen, sondern nebenher auch noch andere Funktionen bedienen müssen. Damit diese Controller nicht ununterbrochen den Status von SDA abfragen müssen (Polling), wird nach einer Startbedingung ein Byte versendet, das bis auf die achte Stelle nur Nullen enthält. Das achte Bit ist als einziges im gesamten Paket auf Eins gesetzt. Nach dem Senden des Start-Bytes darf keinerlei Bestätigung erfolgen; das Acknowledge-Bit bleibt auf Eins. Unmittelbar danach wird eine wiederholte Startbedingung gesendet.
Der pollende Mikrocontroller kann nun die SDA-Leitung in längeren Abständen abfragen und sich primär um seine anderen Aufgaben kümmern. Stellt er eine Null auf SDA fest, verkürzt er sofort die Polling-Intervalle und wartet auf eine Eins auf SDA. Tritt diese auf, muss unmittelbar eine weitere Eins folgen (das nicht gesetzte Acknowledge-Bit). Danach erwartet er die wiederholte Startbedingung. Sollte der Bus nicht genau diesem Ablauf folgen, kann der Mikrocontroller sein Polling-Intervall wieder vergrößern. [2]

2.5.5 Die CBUS-Adresse

Die CBUS-Adresse wurde eingeführt, damit I2C-Instanzen und CBUS-Instanzen durch dieselben Leitungen verbunden werden können. Ein Master, der eine CBUS-Instanz adressieren möchte, schickt zunächst die CBUS-Adresse. Auf dieses Paket darf keine I2C-Instanz reagieren. Der weitere Datentransfer kann nun im CBUS-System erfolgen. Ist der Transfer beendet, wird ein Stopp-Signal auf die SDA-Leitung gelegt. Damit wird allen anderen I2C-Instanzen die Freigabe des Busses signalisiert. Jetzt können wieder normale I2C-Pakete versendet werden. [2]

3. Joint Test Action Group (JTAG)

Ende der 1970er Jahre war der Integrationsgrad der Mikroelektronik soweit gestiegen (zeitgenössische komplexe ICs sind Intel 4004, Intel 8008 oder Zilog Z80), dass ICs mit tausenden Flipflops bzw. Registern in einem Chip arbeiteten. Die Zustände dieser internen Flipflops sind bei einem IC nicht mehr zugänglich. Es entstand die Forderung zunächst der IC-Hersteller selbst, dass zum Test der Struktur eines komplexen Bausteins dessen Gatter und Leitungen steuerbar und zum Test der Funktion die Zustände aller Register und Flipflops beobachtbar sein sollten. Eichelberger veröffentlichte 1977[1] einen als Scan-Path bezeichneten Lösungsansatz, bei dem jedes Flipflop im IC einen zusätzlichen Multiplexer (Transfergate) am Eingang erhält. Auf diese Weise können nun alle Flipflops des IC wahlweise zu einem langen Schieberegister zusammengeschaltet werden, über das jeder Zustand jedes Flipflops von außen beobachtbar und steuerbar wird.

Der JTAG-Standard entstand durch einen Zusammenschluss von Halbleiterherstellern im Jahr 1985/86. Es wurde ein Standard erarbeitet, der in der Norm IEEE 1149.1–1990 festgehalten wurde. Seit der Überarbeitung IEEE 1149.1–1994 ist die Boundary Scan Description Language Teil des Standards. Die aktuelle Version des Standards ist 1149.1-2001 IEEE Standard Test Access Port and Boundary-Scan Architecture. [8]

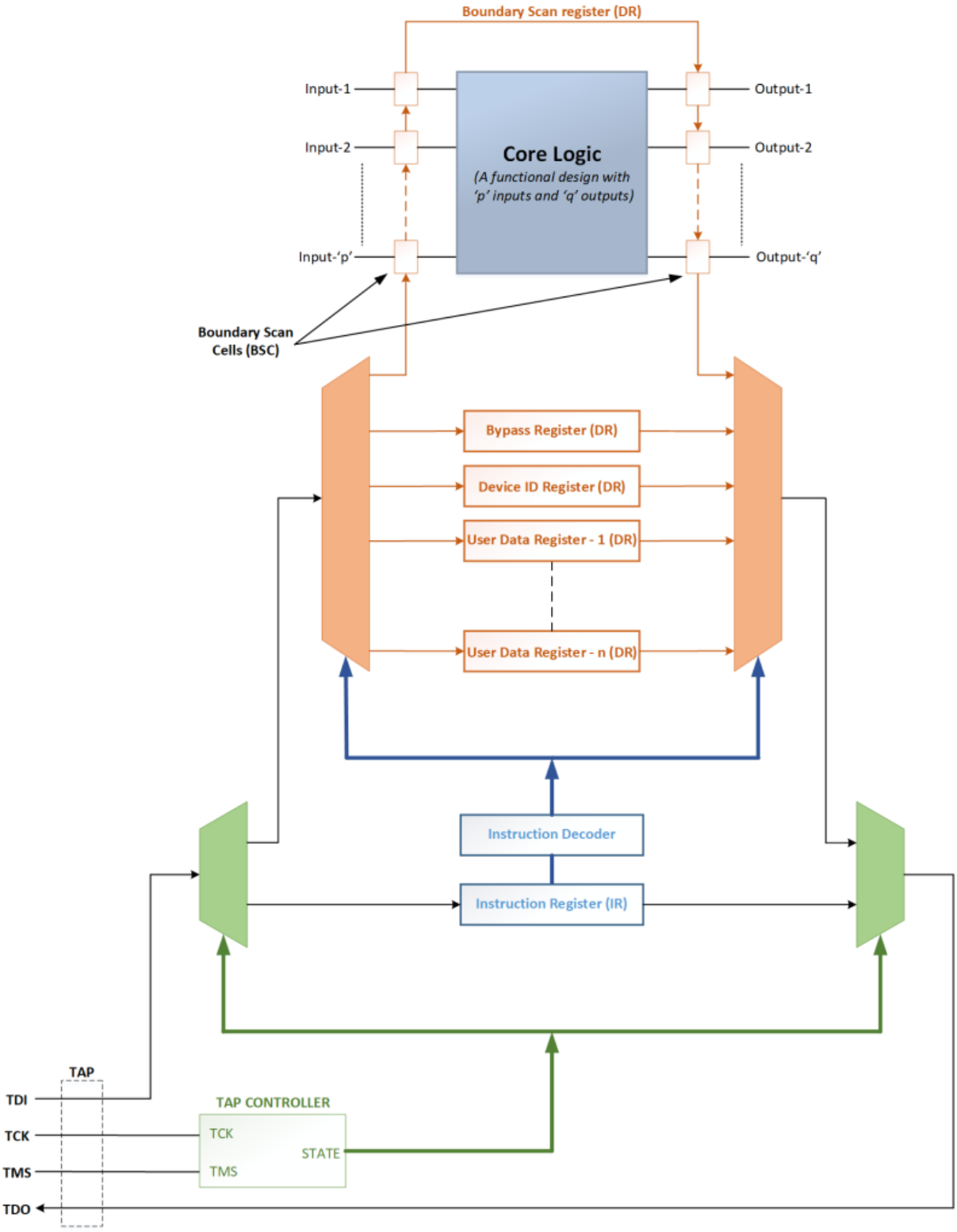

Abbildung 7 :Eine Top-Level-Ansicht der JTAG-Architektur [9]

3.1 JTAG-Test-Access-Port (TAP)

Der in Abbildung 7 dargestellte Test Access Port (TAP) stellt die Schnittstelle zwischen der Boundary Scan Logik im Baustein und der Außenwelt dar. Es besteht aus dem TAP-Controller, einem Befehlsregister und mehreren Testdatenregistern, sowie einiger koppellogik. Der TAP-Controller enthält eine Zustandsmaschine und ist für die Interpretation der TCK- und TMS-Signale verantwortlich.

Der Data Input Pin (TDI) wird verwendet, um Daten in die Grenzzellen (Boundary Scan Cells) zwischen den physikalischen Pins und dem IC-Kern zu laden, sowie um Daten in das Befehlsregister oder eines der Datenregister zu laden. Der Data Output Pin (TDO) wird verwendet, um Daten aus den Grenzzellen zu lesen oder um Daten aus dem Befehlsregister oder den Datenregistern zu lesen. Diese Pins und Register ermöglichen es, über den TAP auf die integrierte Logik im Baustein zuzugreifen und sie zu testen und zu programmieren. [9]

Der TAP verwendet folgende Signale, um den Betrieb des Boundary-Scans zu unterstützen:

1. TDI (Test Data In) – dieses Signal repräsentiert die Daten, die in die Test- oder Programmierlogik des Gerätes verschoben wurden. Es wird an der steigenden Flanke des TCKs abgetastet, wenn die interne Zustandsmaschine im korrekten Zustand ist.
2. TDO (Test Data Out) – dieses Signal repräsentiert die Daten, die aus der Test- oder Programmierlogik des Gerätes verschoben werden. Die Daten werden mit der fallenden Flanke des TCKs Signals angelegt, wenn sich die interne Zustandsmaschine im korrekten Zustand befindet.
3. TCK (Testclock) – dieses Signal synchronisiert die Ausführung der internen Zustandsmaschine.
4. TMS (Test Mode Select) – Steuert die Zustandsübergänge des TAP-Controllers.
5. [Optional] TRST (Test Reset) – dieser Pin ist optional und kann, falls verfügbar, die Zustandsmaschine des TAP-Controllers zurücksetzen.

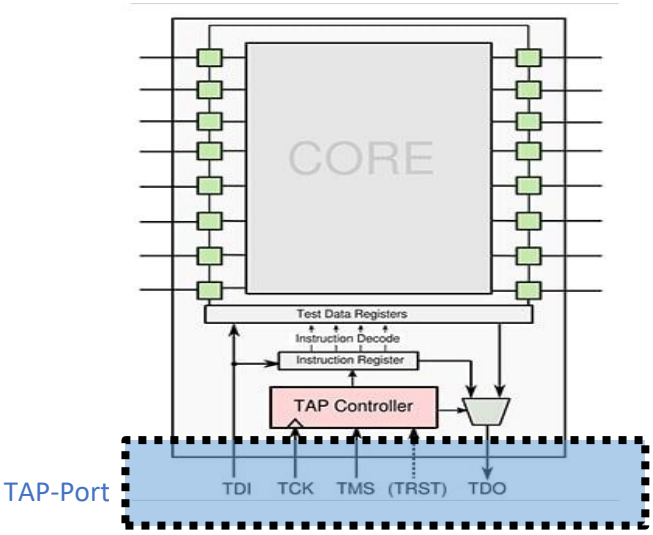

Abbildung 8: JTAG-Architekturschema mit TAP-Ports [11]

3.2 TAP-Controller

Die Zustandsmaschine des TAP-Controllers, welche in Abbildung 9 dargestellt, wird mithilfe des Modus-Auswahlsignals TMS gesteuert, das von dem Taktsignal TCK synchronisiert wird. Es gibt nur zwei mögliche "Pfade", die die Zustandsmaschine einschlagen kann, die zwei verschiedene Modi darstellen: den Befehlsmodus und den Datenmodus. Der Modus wird durch das Eintakten von TMS-HIGH oder TMS-LOW ausgewählt. Wenn sich die Zustandsmaschine in einem bestimmten Modus befindet, kann sie entweder fortgesetzt (TMS-HIGH) oder in Richtung des Rücksetzzustands (TMS-LOW) bewegt werden.

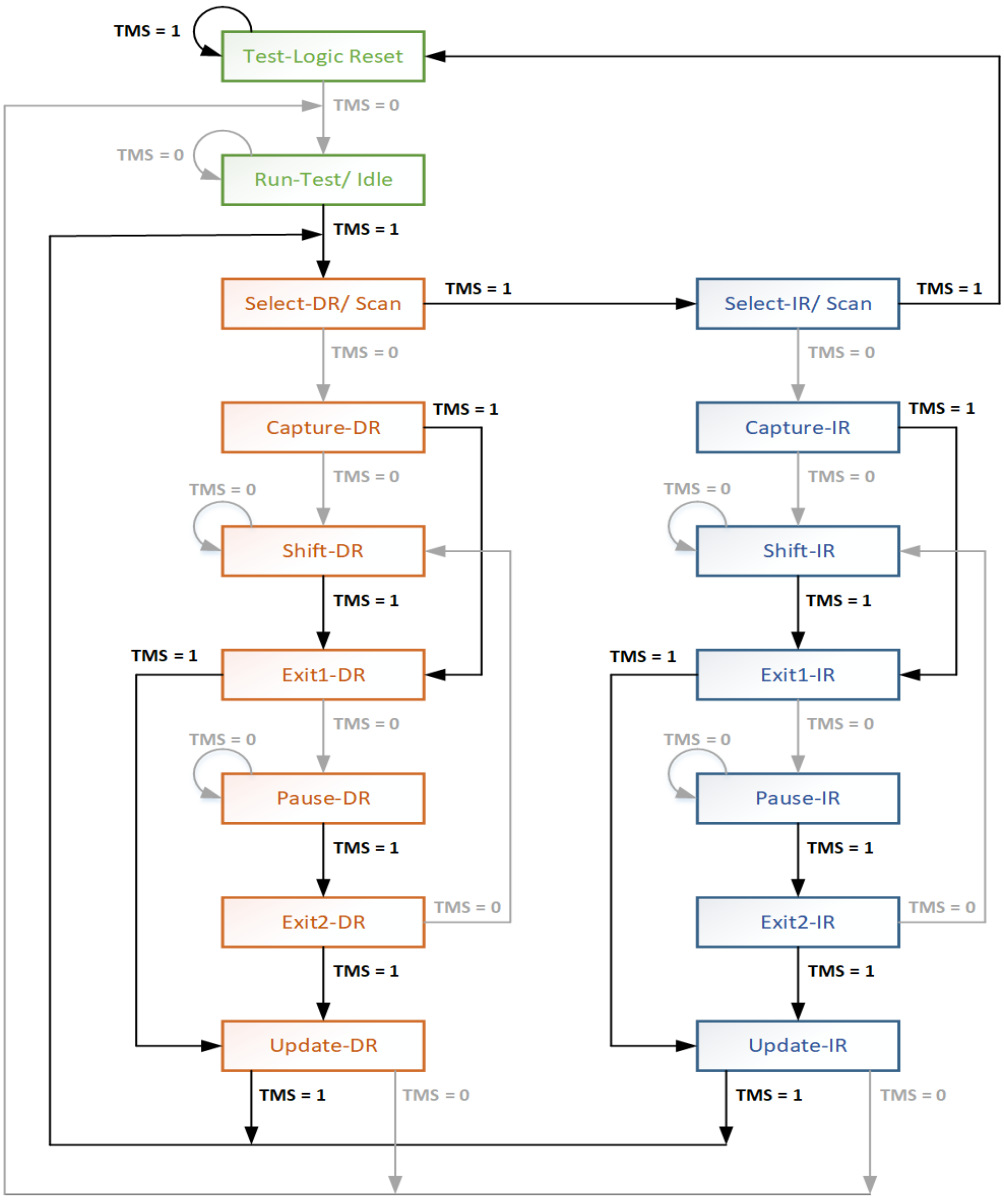

Abbildung 9: Zustandsübergangsdiagramm des TAP-Controllers FSM [9]

Im folgenden wird eine kurze Beschreibung über die verschiedenen Zustände des TAP-Controllers gegeben.

- **Reset der Testlogik**: Setzt die JTAG-Schaltungen zurück. Immer wenn das (optionale) TRST-Signal aktiviert wird, geht es in diesen Zustand zurück. Zu beachten ist auch, dass der TAP-Controller von jedem Zustand, in dem er sich befindet, in diesen Zustand zurückkehrt, wenn TMS für 5 aufeinanderfolgende TCK-Zyklen auf 1 gesetzt ist. Wenn also das TRST-Signal nicht vorhanden ist, kann die Schaltung trotzdem auf diese Weise zurückgesetzt werden.

- **Run-Test/Idle**: Dies ist ein Zustand, in dem die FSM auf den Abschluss einiger Testoperationen wartet.

• **Select-DR/Scan** und **Select-IR/Scan** werden verwendet, um zwischen die Verzweigung in den Daten und den Instruktionszweig der Zustandsmaschine wählen zu können.

• **Capture-DR und Capture-IR**: In diesem Zustand können Daten parallel in das entsprechende Register des Controllers geladen werden

• **Shift-DR und Shift-IR**: In diesem Zustand werden die neue Daten über die TDI Leitung in das Schieberegister geschoben, während die bereits vorhandenen Daten im Schieberegister gleichzeitig über die TDO Leitung herausgeschoben werden. Das bedeutet, dass sowohl das Ein- als auch das Auslesen von Daten über das Schieberegister gleichzeitig erfolgen

• **Exit1-DR und Exit1-IR**: Alle aus den Zuständen Capture-DR und Capture-IR oder aus den Zuständen Shift-DR und Shift-IR seriell geladenen Daten werden in diesem Zustand im Register gehalten.

• **Pause-DR und Pause-IR**: Die FSM hält hier ihre Funktion an, um auf eine externe Operation zu warten.

• **Exit2-DR und Exit2-IR**: Dieser Zustand stellt das Ende des Pause-DR- oder Pause-IR-Vorgangs dar und ermöglicht dem TAP-Controller, in den Shift-DR- oder Shift-IR-Zustand zurückzukehren, damit weitere Daten hineingeschoben werden können (oder verschoben).

• **Update-DR und Update-IR**: Die im Schiebregister gespeicherten Testdaten werden je nach Zustand in das Instruktionsregister oder das zuvor adressierte Datenregister übernommen.

3.3 Befehlsregister (Instruction-Register)

Der Zweck des Befehlsregisters besteht darin, Befehle zu speichern und zu übertragen und die Kommunikation mit der integrierten Logik im Baustein zu ermöglichen. Typischerweise besteht ein Befehlsregister (IR) aus zwei Register, wie in Abb. 10 dargestellt. Das Halteregister speichert den vorherigen Befehl, während das Schieberegister verwendet wird, um den nächsten Befehl einzufügen, ohne die Ausführung des vorherigen Befehls zu beeinflussen.

Die Steuersignale, die an das Befehlsregister gesendet werden, stammen von der TAP-Steuerung und können je nach FSM-Zustand entweder die Einfügung oder Extraktion von Daten über das Schieberegister bei seriellen Aktualisierungsoperationen im Shift-IR-Zustand oder eine Übertragung des Inhalts des Schieberegisters an das Halteregister bei parallelen Aktualisierungsoperationen im Zustand Update-IR bewirken. Diese Steuersignale werden verwendet, um die Kommunikation mit dem Befehlsregister zu steuern und die gewünschten Operationen auszuführen.

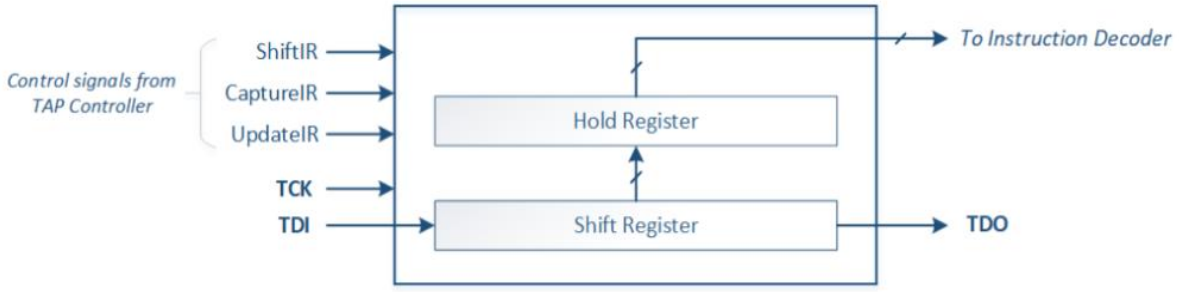

Abbildung 10: Eine Top-Level-Ansicht des Instruktions-Register (IR) [9]

Das Befehlsregister bestimmt unter anderem den Betriebsmodus des Boundary Scan Bausteins, der wiederum Auswirkungen auf die Steuerung der Boundary Scan Zellen und die Auswahl des aktuellen Datenregisters in der Scankette hat. Der IEEE Standard 1149.1 definiert drei Befehle, die zwingend erforderlich sind:

- BYPASS: Dieser Befehl schaltet alle Boundary Scan Zellen aus und leitet die Daten direkt von den Eingangs- zu den Ausgangs-Pins weiter. Er wird verwendet, um Tests zu überspringen, wenn keine Boundary Scan Zellen vorhanden sind oder wenn sie nicht angeschlossen sind.
- EXTEST: Dieser Befehl schaltet alle Boundary Scan Zellen ein und ermöglicht es, Tests auf Verbindungen zwischen den Eingangs- und Ausgangs-Pins des Bausteins durchzuführen.
- SAMPLE/PRELOAD: Dieser Befehl schaltet alle Boundary Scan Zellen ein und ermöglicht es, die Werte der Eingangs-Pins aufzunehmen und in den Boundary Scan Zellen zu speichern oder die Werte der Boundary Scan Zellen auf die Ausgangs-Pins zu übertragen. Er wird verwendet, um Tests auf Verbindungen innerhalb des Bausteins durchzuführen. [4]

Die Befehle, die vom Befehlsregister verarbeitet werden, sind wichtig, um den Betriebsmodus des Boundary Scan Bausteins zu bestimmen und somit die Steuerung der Boundary Scan Zellen und die Auswahl des aktuell in die Scankette geschalteten Datenregisters zu beeinflussen. Durch diese Befehle können Tests auf Verbindungen zwischen den Eingangs- und Ausgangs-Pins des Bausteins durchgeführt werden, Tests auf Verbindungen innerhalb des Bausteins werden ermöglicht und die Werte der Eingangs-Pins können aufgezeichnet und in den Boundary Scan Zellen gespeichert oder die Werte der Boundary Scan Zellen auf die Ausgangs-Pins übertragen werden. Die Verarbeitung dieser Befehle durch das Befehlsregister ist daher wichtig, um die Funktionalität des Boundary Scan Bausteins zu ermöglichen. [9]

Andere häufig verfügbare Anweisungen (Optional) sind:

- IDCODE – diese Anweisung bewirkt, dass TDI und TDO mit dem IDCODE-Register verbunden sind.
- INTEST – diese Anweisung bewirkt, dass die TDI- und TDO-Leitungen mit dem Boundary-Scan-Register (BSR) verbunden sind. Während die EXTEST-Anweisung es dem Benutzer erlaubt, Pin-Pegel einzustellen und zu lesen, bezieht sich die INTEST-Anweisung auf die Core-Logiksignale eines Gerätes.

Jedem Befehl ist ein spezifischer Befehlscode zugeordnet. Dieser kann von jedem Chip-Hersteller frei definiert werden. Ausgenommen ist der BYPASS-Befehl, der sich vollständig aus Einsen zusammensetzen muss. Auch die Länge des Befehlsregisters kann frei gewählt werden wobei das Minimum zwei Bits beträgt. Eine beispielhafte Zuordnung ist in Tabelle 2 dargestellt, wobei hier die Länge des Befehlsregisters auf vier Bits festgelegt wurde. Die Zuordnung von Befehlscodes zu Befehlen hilft, den Betriebsmodus des Boundary Scan Bausteins zu bestimmen und die gewünschten Operationen auszuführen. [4]

Tabelle 2: spezifische Befehlscode des JTAGs [11]

BETRIEBSMODE / BEFEHL	BEFEHLSCODE (BINÄR)
BYPASS	1111
SAMPLE/PRELOAD	0001
EXTEST	0000
IDCODE	0010

3.4 Befehls-Decoder (Instruction Decoder)

Die Anweisung aus dem Befehlsregister (IR) wird an eine Decoder-Logik weitergeleitet, die das entsprechende Datenregister für den JTAG-Betrieb auswählt. Jedes Datenregister im JTAG wird mit einem eindeutigen Wert bzw. Opcode belegt. Um ein bestimmtes Datenregister auszuwählen, wird das IR mit dem entsprechenden Opcode geladen und der Befehlsdecoder dekodiert diesen Wert und richtet einen Zugriffspfad zwischen TDI/TDO und dem benötigten Datenregister ein. Durch die Verwendung von Opcodes und der Decoder-Logik kann das JTAG-System einfach zwischen verschiedenen Datenregistern und Operationen umschalten und somit die Kommunikation mit der integrierten Logik im Baustein unterstützen. Abb. 11.

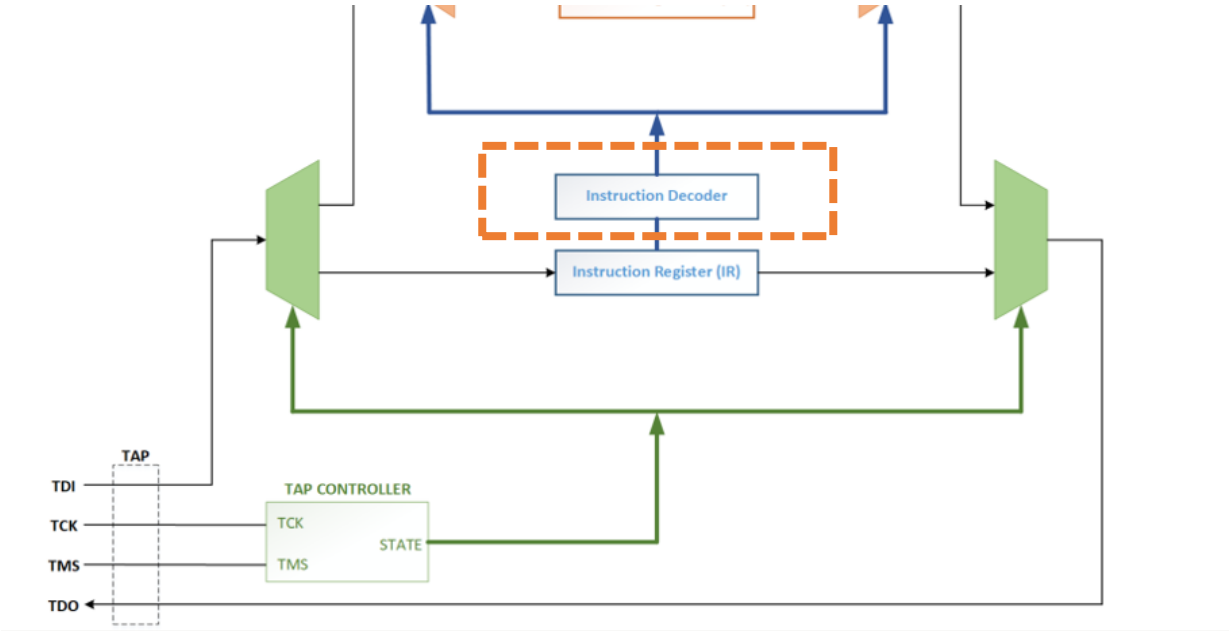

Abbildung 11: Befehlsregister in JTAG Architektur [9]

3.5 Datenregisters (Data-Registers)

Ein JTAG-Baustein enthält mehrere Datenregister, die zur Speicherung oder zum Auslesen von Informationen im Baustein verwendet werden können.

Der Standard IEEE 1149.1 definiert als Minimum zwei unbedingt erforderliche Datenregister:

- BYPASS
- Boudary-Scan

Auch hier sind wieder zusätzliche Register möglich, wie das "Device Identification"- Register welches umgangssprachlich auch "idcode"-Register gennant wird.

3.5.1 BYPASS-Register

Das Bypass-Register verbindet die TDI- und TDO-Leitungen miteinander und ermöglicht somit das Testen anderer Geräte in der JTAG-Kette, ohne unnötig Zeit zu verlieren. Um einen Chip zu umgehen, muss der Operation-Code des Bypass-Registers in das Befehlsregister geladen werden, wodurch das Bypass-Register zwischen TDI und TDO bereitgestellt wird. [9]

In Abb. 12 sind mehrere Chips in Serie geschaltet. Ohne das Bypass-Register müssten die Daten durch die BSCs von Chip-1 und Chip-3 geschoben werden, um die Daten in den BSCs von Chip-2 zu laden oder auszulesen, was die Lade- und Auslesungszeit verlängern würde. Um diese Verzögerung zu vermeiden, wird ein Bypass-Register zwischen TDI und TDO von Chip-1

und Chip-2 bereitgestellt, welches nur eine Verzögerung von einem Takt pro JTAG-Gerät verursacht.

Aus dem gegebenen Beispiel ergibt sich für die Anzahl benötigte Takte zur Auslesung von Chip 2:

- Ohne Bypass werden alle 3 Chips durchlaufen und es ergeben sich: 12 + 8 + 12 = 32 Takte
- Mit Bypass werden Chip-1 und Chip-3 überbrückt: 1 + 8 + 1 = 10 Takte

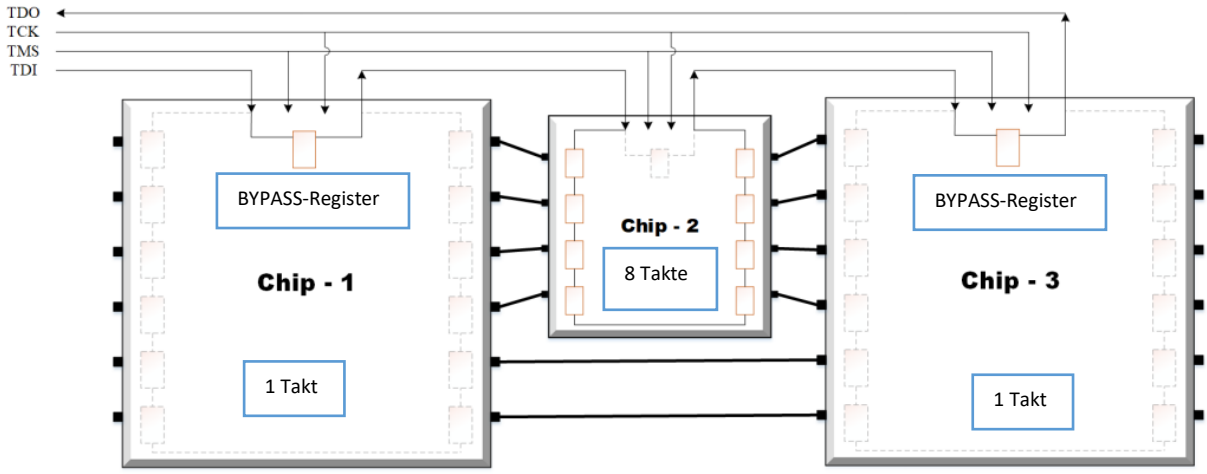

Abbildung 12: Ein Beispiel, wie das Bypass-Register verwendet wird [9]

3.5.2 Boundary Scan Register

Die Boundary-Scan-Technik ermöglicht es, die Funktionalität von Schaltungen zu testen, indem Signale von außen über vordefinierte Pfade in die Schaltung injiziert werden und die Signale, die an den Pins des integrierten Schaltkreises (ICs) anliegen, über den Scan-Pfad erfasst werden. Dies geschieht mithilfe von zusätzlichen Zellen, die als Boundary-Scan-Register (BSR) bezeichnet werden. Wenn mehrere Chips JTAG und Boundary-Scan unterstützen, können die Pins jedes Geräts gesteuert und überwacht werden, indem der BSR auf den erforderlichen Wert geladen wird und die Antwort über den BSR erfasst wird, um so die Verbindung zwischen den Chips zu testen.

Das Boundary-Scan-Testverfahren nutzt Boundary-Scan-Zellen Abbidlung 13, um die Pins von Bauteilen unabhängig von deren normaler Funktion zu kontrollieren und zu überwachen. Die

Boundary-Scan-Zellen befinden sich zwischen der Kernlogik des Bausteins und dessen Ausgangstreiber bzw. Eingangstreiber. Sie ermöglichen es, bestimmte Pegel an die Bauteilpins zu treiben oder zu messen. Die BSR-Zelle ist der Hauptbestandteil des Boundary-Scan-Testverfahrens und alle anderen Konstrukte dienen der korrekten Ansteuerung der einzelnen Boundary-Scan-Zellen. [9]

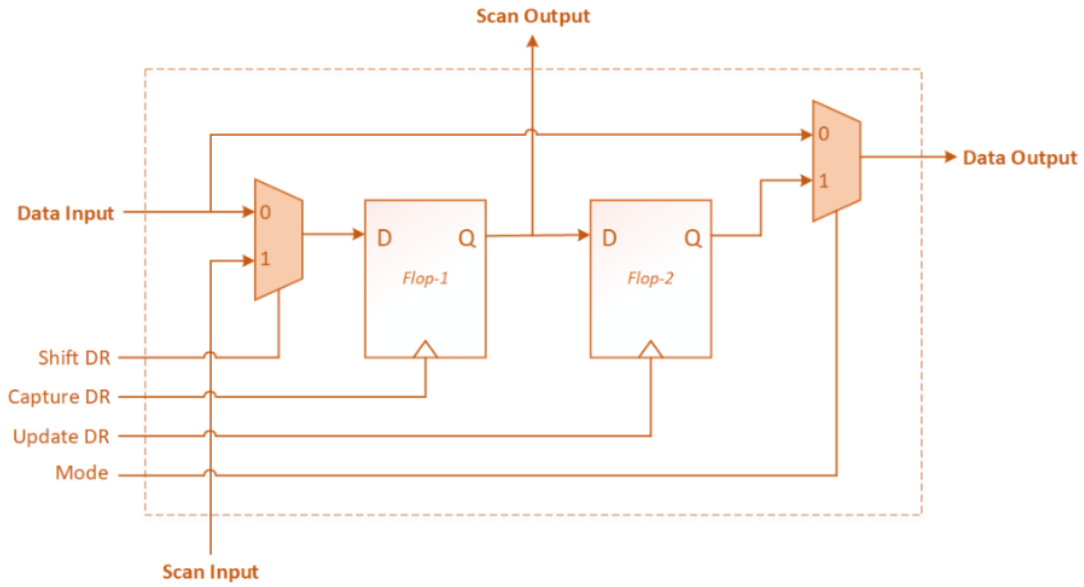

Abbildung 13: Eine einfache Boundary-Scan-Zelle (BSC) [9]

Die BSC unterstützt vier Operationen, wie unten in der Tabelle 3 gezeigt:

Tabelle 3: verschiedene Operationen der BSC [9]

Betrieb	Steuersignale		Clock	Beschreibung
	Mode	Shift Dr		
Normal	0	X	N/A	Der Funktionsmodus, somit ist BSC transparent
Scan	X	1	Clock DR (or Capture Dr)	Daten werden durch Verwendung des FF1 von einer BSC zur anderen verschoben
Update	1	X	Update DR	verschobene Daten werden von Flop-1 zu Flop-2 übernommen
Capture	X	0	Clock DR (or Capture Dr)	Daten werden vom Dateneingang in Flop-1 übernommen

3.5.2.1 Externen Test (Extest)

Dies Extest Anweisung steuert die Verbindung von TDI und TDO mit dem Boundary-Scan-Register (BSR). Der Clock_DR-JTAG-Zustand wird verwendet, um die Pin-Zustände des Geräts zu scannen, während der Shift_DR-Zustand dazu verwendet wird, neue Werte in das BSR zu schreiben. Mit dem Update_DR-Zustand werden diese Werte schließlich auf die Pins des Geräts angewendet. Auf diese Weise können die Pin-Zustände des Geräts gesteuert und überwacht werden.

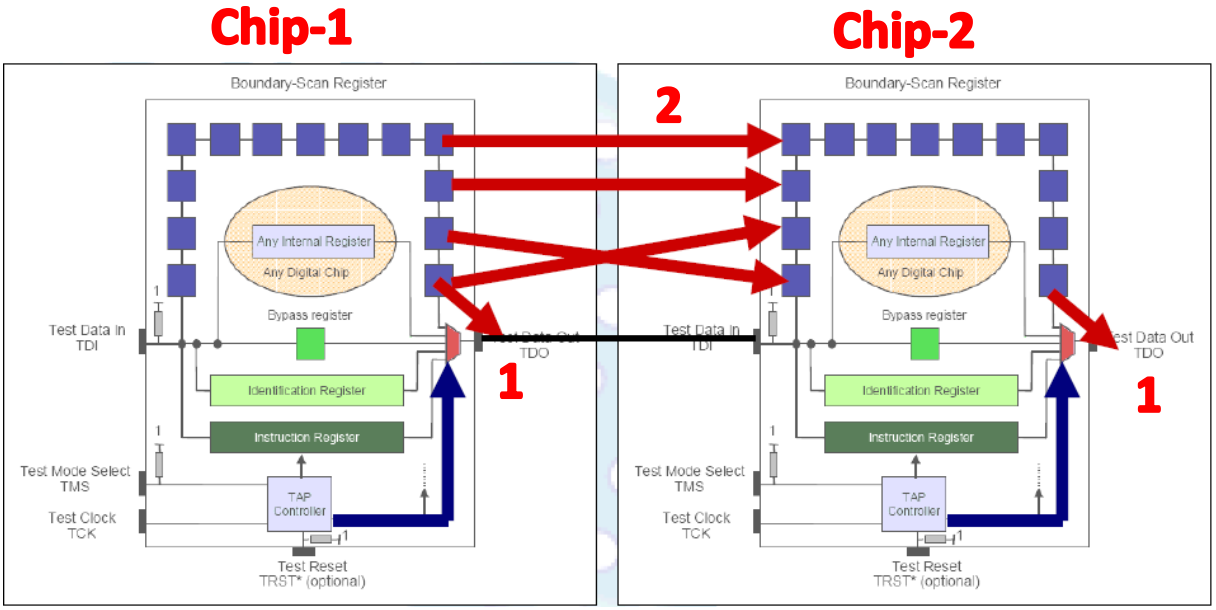

Abbildung 14: Aufbau des Pins-ExternalTest

Die Abbildung 14 veranschaulicht ein Beispiel für einen externen Test der Verbindung zwischen zwei Chips. Bei diesem Test werden die folgenden Schritte durchgeführt:

- Der Befehlscode besteht ausschließlich aus logischen Nullen.
- Das Boundary-Scan-Register wird als Ausgang ausgewählt (siehe Abbildung 14, Nummer 1).
- Die Zellen werden auf den externen Test vorbereitet (siehe Abbildung 14, Nummer 2).

3.5.2.2 SAMPLE/PRELOAD

Diese Anweisung sorgt dafür, dass TDI und TDO mit dem Boundary-Scan-Register (BSR) verbunden sind, während das Gerät weiterhin in seinem normalen Funktionsmodus bleibt.

Während dieser Anweisung kann auf das BSR durch einen Datenabtastvorgang zugegriffen werden, um eine Probe der Funktionsdaten zu erfassen, die in das Gerät eingehen und es verlassen. Die Anweisung wird auch verwendet, um Testdaten in das BSR zu laden, bevor eine EXTEST-Anweisung ausgeführt wird. Auf diese Weise können Funktionalität und Leistung des Geräts getestet werden, ohne dass das Gerät seinen normalen Betriebsmodus verlässt.

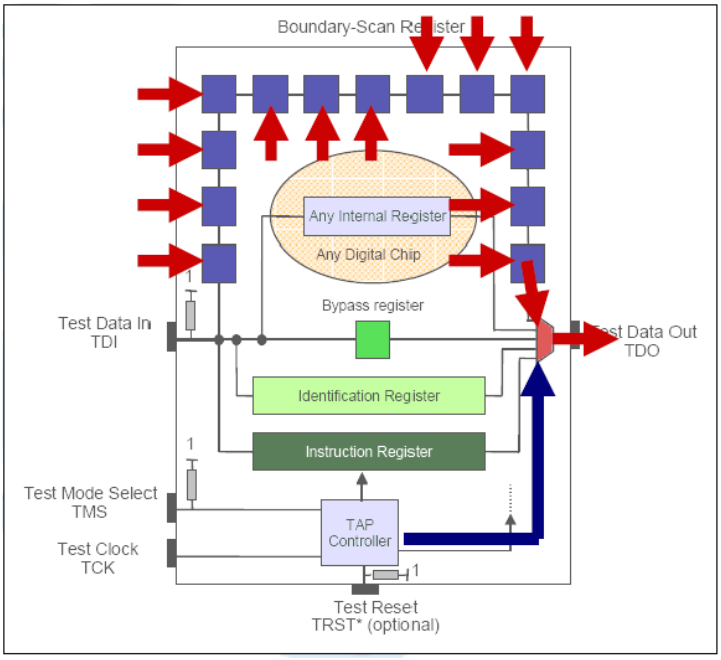

Abbildung 15: Sample/Preload-Funktionalität

Die Abbildung 15 zeigt ein Beispiel für eine Sample/Preload Vorgang. Dabei werden folgende Schritte durchgeführt:

- Befehlscode ist vom Designer zu definieren
- Das Boudary-Scan-Register wird als Ausgang ausgewählt
- Einlesen von Daten in die Boundary-Scan-Zellen wird aktiviert

3.5.3 Device Identifikation (ID-Code)

Ein Device Identifikation (ID) Code ist eine eindeutige Kennung, die einem Gerät oder Bauteil zugeordnet wird, um es von anderen Geräten oder Bauteilen zu unterscheiden. Dieser Code wird in der Regel in Form einer Binärzahl oder als Hexadezimalzahl dargestellt und kann in Hardware oder in Software implementiert sein. Er wird häufig verwendet, um den Typ oder die Funktionalität eines Geräts oder Bauteils zu bestimmen und kommt in vielen Bereichen zum Einsatz, wie zum Beispiel in der Elektronik, in der Computertechnik oder in der Automatisierungstechnik. Der ID Code dient dazu, Geräte oder Bauteile eindeutig zu identifizieren. [3]

Dieses Register enthält den Identifikation-Code und die Revisionsnummer für das Gerät. Mit dieser Information kann der Debugger, die unterschiedliche verbundene Debug-Ports identifizieren.

Dieses ID-Code-Register ist 32bits breit und enthält die folgende Informationen an den folgenden Bitpositionen:

- 31-28: steht für die Versionierung & Revision
- 27-12: steht für Artikelnummer/Produktnummer
- 11-1: steht für Hersteller
- 0: ist immer mit 1 gesetzt

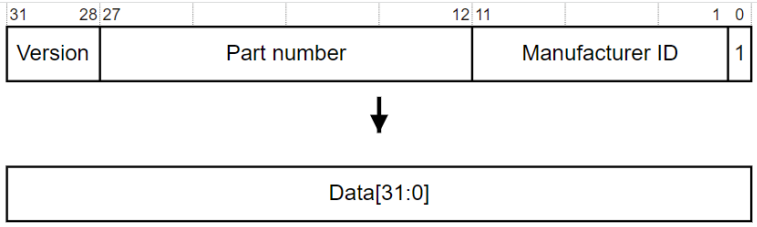

Abbildung 16: Inhalt des ID-CODE (Identifikationscode) [11]

4. Verwendete Software-Pakete

Beim Design der seriellen Schnittstellen werden verschiedene Software-Tools verwendet. In diesem Kapitel werden diese Software-Pakete vorgestellt.

4.1 HDL-Designer

HDL-Designer ist eine leistungsstarke, HDL-basierte Entwicklungsumgebung, die dazu dient, komplexe FPGAs und ASICs zu designen und zu analysieren. Die Software wird von Ingenieuren und Ingenieurteams weltweit verwendet, um Designs von Bauteilen zu erstellen und zu verwalten. HDL-Designer bietet viele Möglichkeiten, um die Produktivität und Vorhersehbarkeit von Projekten zu verbessern, indem es Abläufe und Aufgaben automatisiert. Durch die Automatisierung von Regelprüfungen, Registergenerierung und Dokumentation sowie die Verwendung von Editoren für die textuelle, tabellarische und grafische Erstellung von funktionatlität wird die Entwicklungszeit reduziert und manuell eingeführte Fehler minimiert. Die Integration von Tools und die Versionsverwaltung des gesamten Projekts tragen dazu bei, den Team-, Tool- und Designprozess strukturiert zu halten, bleiben aber durch eine API flexibel genug, um vorhandene Designflüsse zu erweitern. Durch diese Automatisierung und das Projektmanagement wird die Qualität des Projekts und damit auch der endgültige Chip verbessert und das Projektrisiko reduziert.

Durch die Nutzung von HDL-Designer können durch Automatisierung und bei zukünftigen Projekten durch bessere Designwiederverwendung, Konsistenz der Codierung und verbesserte Dokumentation Kosten gesenkt werden und zusätzliche Kosten vermieden werden. Für Sicherheits- und unternehmenskritische Projekte unterstützen Eigenschaften des HDL-Designers wie Design-Checking, Versionsverwaltung, Registergenerierung und -dokumentation die Einhaltung von Vorschriften wie DO-254, ISO 26262. [12]

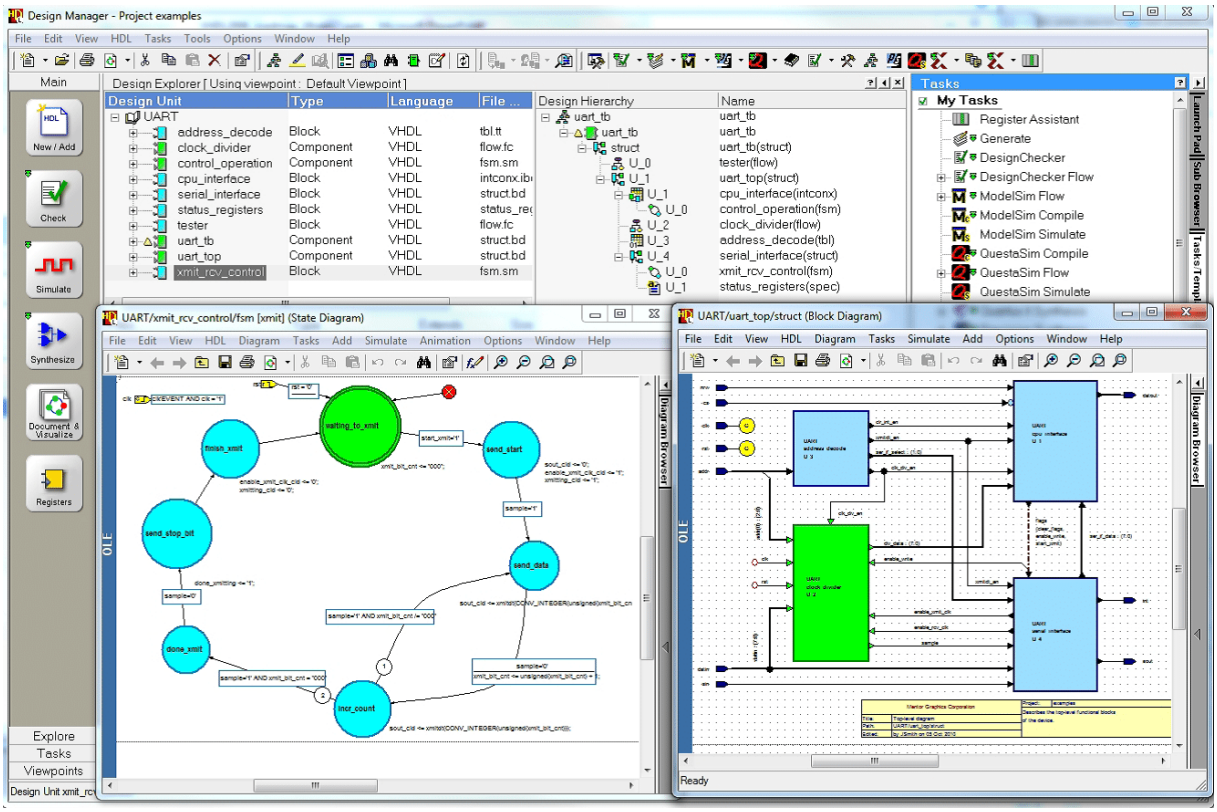

Abbildung 17: Überblick von HDL-Designer [12]

4.2 QuestaSim

Questa Sim ist ein Simulationswerkzeug für elektronische Schaltungen und Systeme, das von der Firma Siemens EDA ehemals Mentor Graphics entwickelt wurde. Es wird hauptsächlich in der Industrie und in akademischen Einrichtungen verwendet, um die Funktion und Leistung von Schaltungen und Systemen zu verstehen und zu optimieren. Questa Sim bietet eine Vielzahl von Funktionen, darunter die Simulation von Schaltungen auf verschiedenen Ebenen des Designs, die Integration von Hardware-Beschreibungssprachen (HDL) wie VHDL und Verilog, die Möglichkeit, Schaltungen zu verifizieren und zu validieren, und die Unterstützung von Co-Simulation und Hardware-in-the-Loop-Simulation. Es wird häufig in Kombination mit anderen Werkzeugen wie Synthese- und Layout-Tools verwendet, um einen vollständigen Design-Flow zu implementieren. [12]

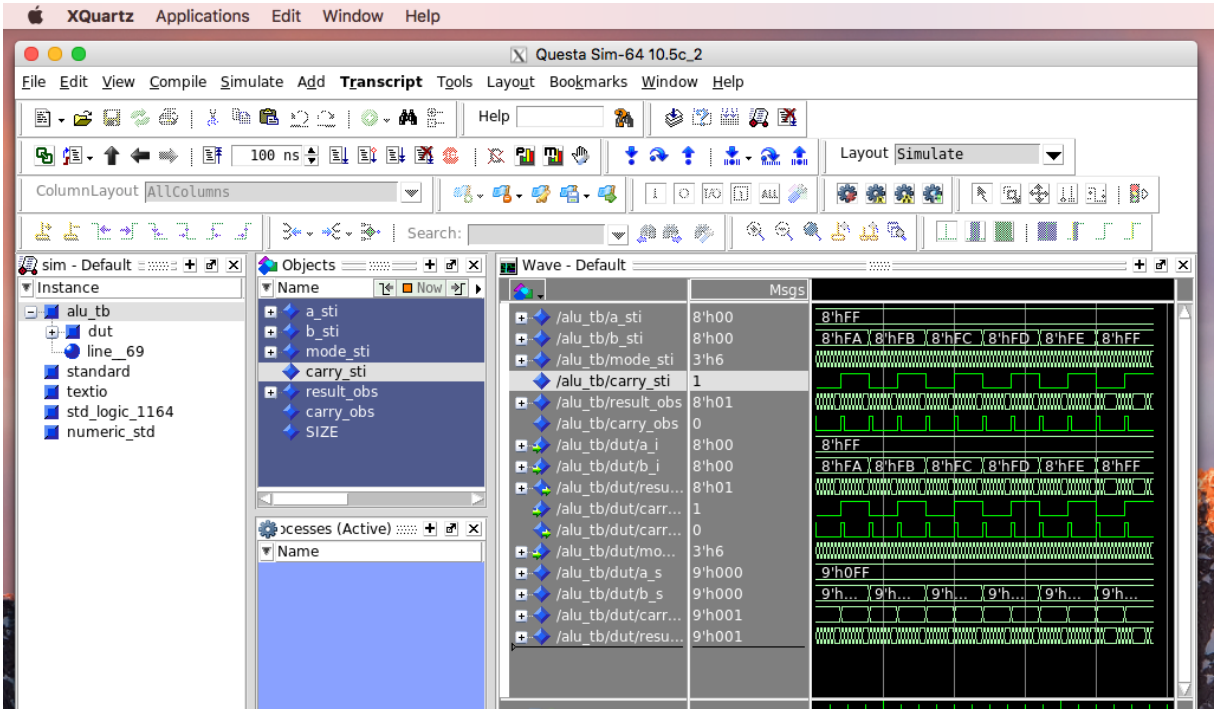

Abbildung 18: QuestaSim Overview [12]

4.3 Verilog

Verilog ist eine Hardwarebeschreibungssprache, die von Ingenieuren verwendet wird, um digitale Schaltungen und Systeme zu entwerfen. Die Verwendung von Verilog ermöglicht es, das System auf verschiedenen Abstraktionsebenen zu spezifizieren, von niedrige ebenen, die sich mit Details von Schaltungen befassten, bis hin zu höheren Ebenen, die sich mit den abstrakten Funktionen und Systemeigenschaften befassen. Verilog wird häufig in Kombination mit anderen Werkzeugen wie Simulationstools verwendet, um das Verhalten des Systems zu testen und zu optimieren. [13]

4.3.1 Geschichte

Verilog wurde in den 1980er Jahren als proprietäres Werkzeug zur Verifikation und Simulation von digitaler Logik entwickelt. Es wurde später von dem Institute of Electrical and Electronics Engineers (IEEE) standardisiert und in die Standards IEEE 1364-1995, IEEE 1364-2001 und IEEE 1364-2005 überführt. Nach Veröffentlichung des letzten Standards hat die IEEE P1364 Arbeitsgruppe ihre Arbeit beendet und die Pflege des Standards der IEEE P1800 Arbeitsgruppe übergeben, die sich mit der Standardisierung von System Verilog befasst. [13]

4.3.2 Sprachübersicht

Verilog hat eine Syntax, die den Sprachen C oder Java ähnelt, obwohl die Semantik, das heißt die Bedeutung der Sprachelemente, ein völlig andere ist. Verilog verfügt über zahlreiche Mechanismen, die für die Modellierung von Hardware geeignet sind und die Parallelität ermöglichen. Diese Mechanismen werden verwendet, um parallele Prozesse in der Hardwarebeschreibung auszudrücken und damit die Entwicklung von Schaltungen und Systemen zu beschleunigen. [13]

Verilog

```
module counter (data, clk, clrn, ena, ld, count);

  input [7:0] data;
  input clk, clrn, ena, ld;
  output [7:0] count;

  reg  [7:0] count_tmp;

  always @(posedge clk or posedge clrn)
   begin
     if (!clrn)
        count_tmp = 'b0;
     else if (ld)
        count_tmp = data;
     else if (ena)
        count_tmp = count_tmp + 'b1;
   end

  assign count = count_tmp;

endmodule
```

Abbildung 19: Verilog BeispiEL-Code

4.3.2.1 Module

Die grundlegende Gliederung in Verilog erfolgt durch Module durchgeführt.

```
module MeinModulName ( input a, output b, inout data);

/* Mehrzeiliger Kommentar

  Wie von C bekannt
*/
endmodule </verilog>
```

Abbildung 20: Aufbau eines Modules in Verilog

Signale werden durch Ports dem Modul übergeben. Für Signale in das Modul gibt es den input-Port. Ausgänge werden durch output spezifiziert und ein bidirektionaler Bus durch inout.

4.3.2.2 Datentypen: wire, reg

Die grundlegenden Datentypen, mit denen in Verilog modelliert wird, sind wire und reg. Analog zum elektrischen Draht können mit dem Wire Verbindungen durchgeführt werden. Wie ein Draht kann ein Wire aber keinen Signalzustand speichern. Für die Modellierung digitaler Logik ist es aber auch nötig, Signaltreiber zu haben. Hier kommt der Datentyp reg ins Spiel. Mit ihm können Signalzustände gespeichert werden, und er findet damit als Signaltreiber Verwendung. [13]

Verilog unterstützt unter anderem die Signalzustände:

- 0 --> logisch null
- 1 --> logisch eins
- z --> hochohmig
- x --> undefiniert

4.3.2.3 Always- und Initial-Blöcke

Die always- und initial-Blöcke werden verwendet, um die parallele Natur von Hardware zu beschreiben. Jeder dieser Blöcke wird parallel ausgeführt, wobei ein initial-Block nur einmal durchlaufen, während ein always-Block, wie eine Endlosschleife ständig wiederholt wird.

Das initial-Konstrukt wird vorwiegend in der Simulation für Testbenches verwendet. In der Beschreibung synthetisierbarer Logik findet es nur in FPGAs Verwendung.

```
module;

 reg a, b, c, d;

initial begin
   a = 0;
   b = 1;
   c = z;
   d = x;
 end
endmodule
```

Abbildung 21: InitialBlock-Block Beispiel

Das „always“ Konstrukt kann sowohl in Testbenches als auch in synthetisierbarer Logik verwendet werden.

Die Abbildung 21 zeigt, wie ein „always“ innerhalb einer synthetisierbarern Logik verwendet werden kann.

```
module BeispielName(
input wire clk
input wire rst
);

reg a;
always @(posedge Clk,negedge rst)
begin
    if(!rst) a=0;

    else   //do something

end
endmodule
```

Abbildung 22: Verilog Modul mit Always-Block Besipiel

5. Der I2C-Master: Beschreibung des Schaltungsentwurfs

In diesem Kapitel wird der entwickelte I²C -Master vorgestellt, der die im vorherigen Kapitel beschrieben Spezifikationen vollständig erfüllt.

5.1 Grundlegender Aufbau

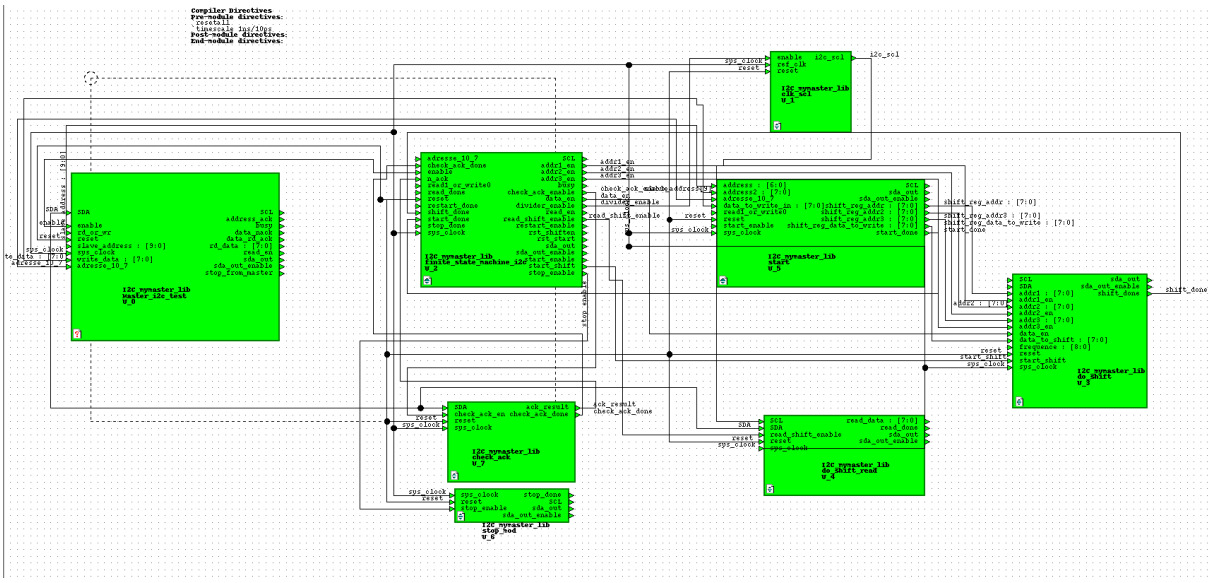

Abbildung 23: Block-Diagramm des Sytems

Die Oberste Hierarchie des Master-Moduls ist in Abbildung 23 dargestellt und besitzt den Namen Master_i2c_TOP. Seine Aufgabe besteht darin selbständig Sende- und Empfangsprozesse auf dem I2C-Bus durchzuführen und dabei die Vorgaben des I2C Protokolls zu beachten. Der vollständige Verilog-Code zu allen Modulen befindet sich in Anhang A1.

Der Datenaustausch mit dem Master_i2c_TOP erfolgt Register basiert. Die Einheit kann über mehrere Signale Daten austauschen, um Registerinhalte auf dem I2C-Bus zu versenden und empfangene Daten in Registern zu speichern.

Der Master_i2c_TOP enthält alle notwendigen Komponenten, um eine reibungslose und protokollkonforme I2C-Funktionalität zu gewährleisten. Alle Module werden mit der Clock des I2C-Busses SCL getaktet. Damit werden einerseits Synchronisationsprobleme vermieden. Auf der anderen Seite kann so unmittelbar auf Clock-Stretching reagiert werden. Zu beachten ist unbedingt, auf welche Flanke von SCL reagiert wird, da dies von Modul zu Modul unterschiedlich ist.

Jedes Modul besitzt zusätzlich ein asynchrones active-low Reset Signal.

Der Master_i2c_TOP besteht aus den folgenden Sub-Modulen, die im folgenden näher erläutert werden :

- Clk_Scl(Clock Divider/ Erzeugung der SCL)

- Start (Start Modul)
- Restart_md (Restart Modul)
- Do_shift_read (Read from Testbench)
- Check_ack (Acknowledge)
- Do_shift (Shift Adresse/Data auf SDA)
- Stop_mod (Stopp Modul)
- Finite_state_machine_i2c (Zustandsmaschine)

5.2 Master_I2C_TOP

Die Schnittstelle des Master_I2C_TOP besteht aus den folgenden Signalen.

Tabelle 4: Master_I2C_Top Signale

Signale	Bemerkungen
SDA	Input wire für SDA
SCL	Output wire for SCL
Sys-Clock	Clock Signal
Enable	Enable erlaubt den Start des Transfers
Slave_Adresse[9:0]	Input wire für Slave Adresse (bis 10-Bit)
Write_Data[7:0]	Input Data, welche zu übertragen sind
Rd_or_wr	Gibt bekannt, ob es sich um ein Read oder Write Befehl handelt.
Adresse_10_7	Gibt bekannt, ob es sich um eine 7Bit oder 10Bit Adresse handelt.
Rd_Data[7:0]	Gibt die Daten aus, die vom Slave ausgelesen wurden.
Busy	Zustand des I2C-Busses. ist gesetzt, wenn der der Bus sich im Transfer befindet.
Sda_out	SDA Output
Sda_out_enable	SDA Output Enable
Stop_From_master	Output wire. Wird gesetzt, wenn der Master den Transfer stoppen will.
Address_ack	Output wire. Wird gesetzt, wenn der Master keine Adressbestätigung vom Slave erhält.
Read_en	Der Master ist bereit, Daten vom Slave zu empfangen.

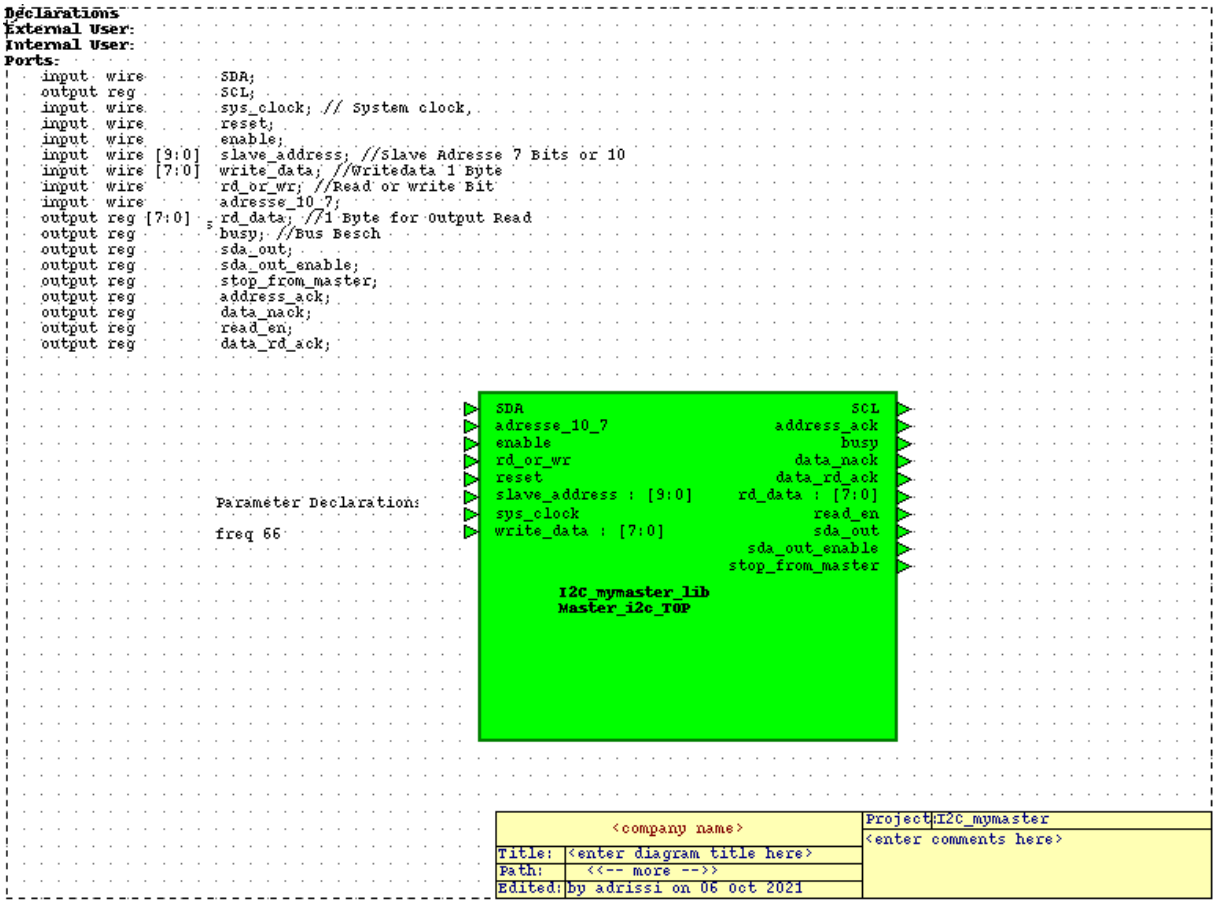

Abbildung 24: Master-I2C-TOP Diagramm

5.2.1 Start/Stopp-Module

Die richtige Erkennung der Start/ Stopp -Bedingungen gehört zu den wichtigen Aufgaben des I2C Masters. In diesem Modul werden die Adressen sowie die zusendenden Daten bei gesetztem „Enable“ in verschiedenen Registern gespeichert und für den Transfer vorbereitet. Nach dem Start dürfen weder die Slave Adressen noch die Data geändert werden.

Zur Signalisierung der durchzuführenden Schritte besitzt das Modul mehrere Inputs/Outputs, die aus Abbildung 24 entnommen werden können.

	A	B	C	D	E	F	G	H
M	Group	Name	Mode	Type	Signed	Bounds	Value	Comment
1		SCL	output	reg				
2		address	input	wire		[6:0]		
3		address2	input	wire		[7:0]		
4		adresse_10_7	input	wire				
5		data_to_write_in	input	wire		[7:0]		
6		read1_or_write0	input	wire				
7		reset	input	wire				
8		sda_out	output	reg				
9		sda_out_enable	output	reg				
10		shift_reg_addr	output	reg		[7:0]		
11		shift_reg_addr2	output	reg		[7:0]		
12		shift_reg_addr3	output	reg		[7:0]		
13		ft_reg_data_to_wr	output	reg		[7:0]		
14		start_done	output	reg				
15		start_enable	input	wire				
16		sys_clock	input	wire				' System clock
17								

Abbildung 25: Start In/Outputs

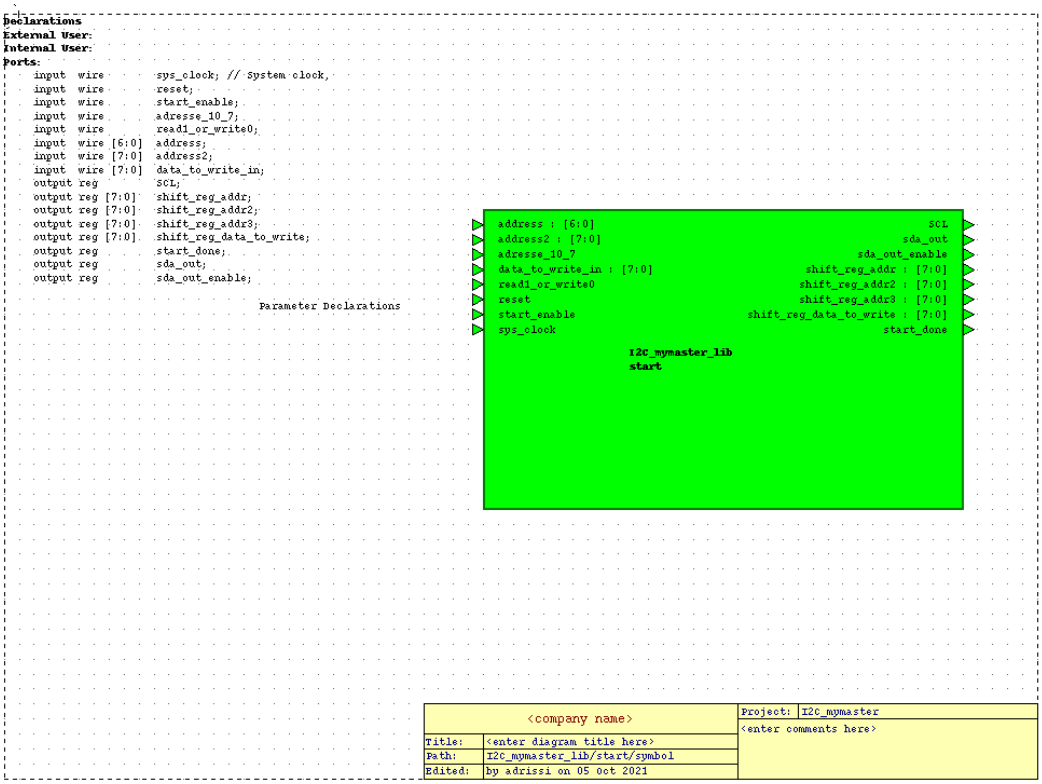

Abbildung 26: Start Diagramm

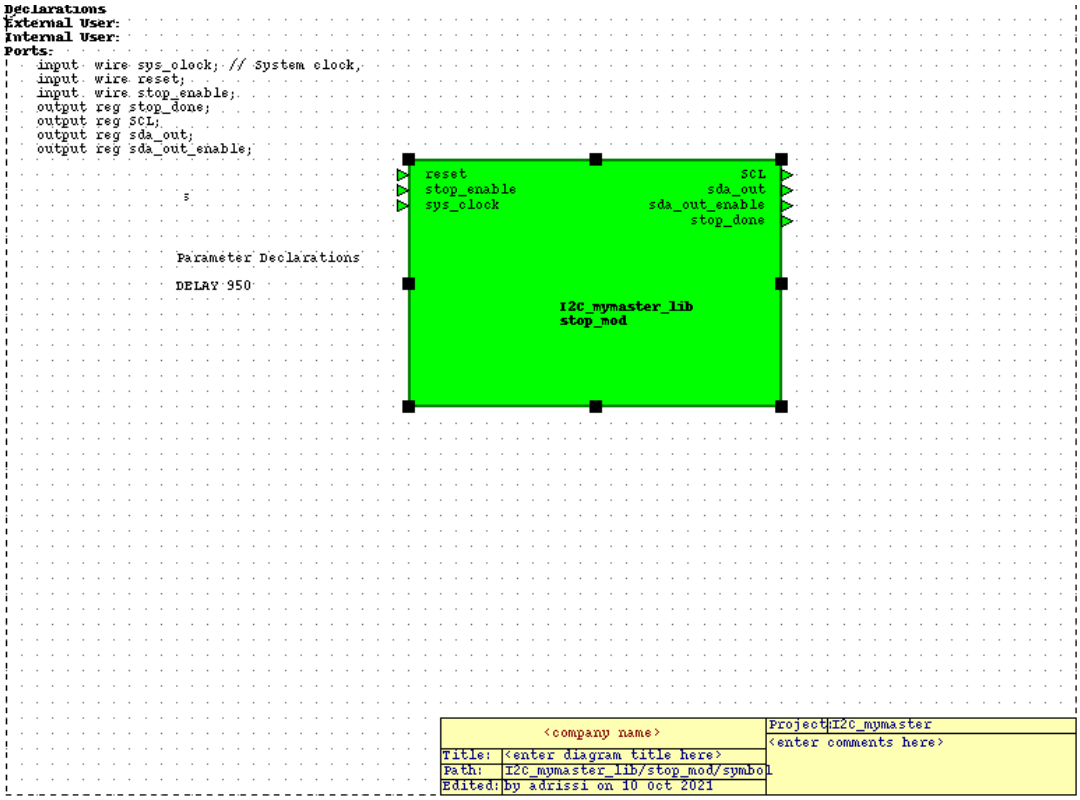

Abbildung 27: Stopp Diagramm

5.2.2 Clk_SCL (Clock Divider/ Erzeugung der SCL)

Das Taktsignal SCL wird in einem externen Modul generiert. Um erkennen zu können, wann eine Zeit von 4,0us (Minimale High Periode des SCL Clock) durchlaufen worden ist, benötigt der Master_i2c_TOP einen einfachen Zähler. Er setzt das Signal I2c_SCL, wenn das Signal Counter den Wert 500 erreicht hat. Zur Initialisierung lässt sich der Zähler mit dem Reset Signal auf den Anfangszählerstand zurücksetzen.

```
//           by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//           at - 12:11:54 01/22/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/1ps
module clk_scl(

input wire reset,
input wire ref_clk,
input wire enable,
output reg i2c_scl
);

reg [9:0]DELAY;
reg [9:0] count;

always @(posedge ref_clk, posedge reset)
begin
  if(reset)
  begin
    count=0;
    i2c_scl<=0;
    DELAY=1000;
  end
  else if(enable)
  begin
    if(count==(((DELAY)/2)-1))
      begin
      i2c_scl = !i2c_scl;
      count=0;
    end
    else
    begin
      count = count +1;
    end
   end
end

endmodule
```

Abbildung 28: Clock Devider Verilog-Code

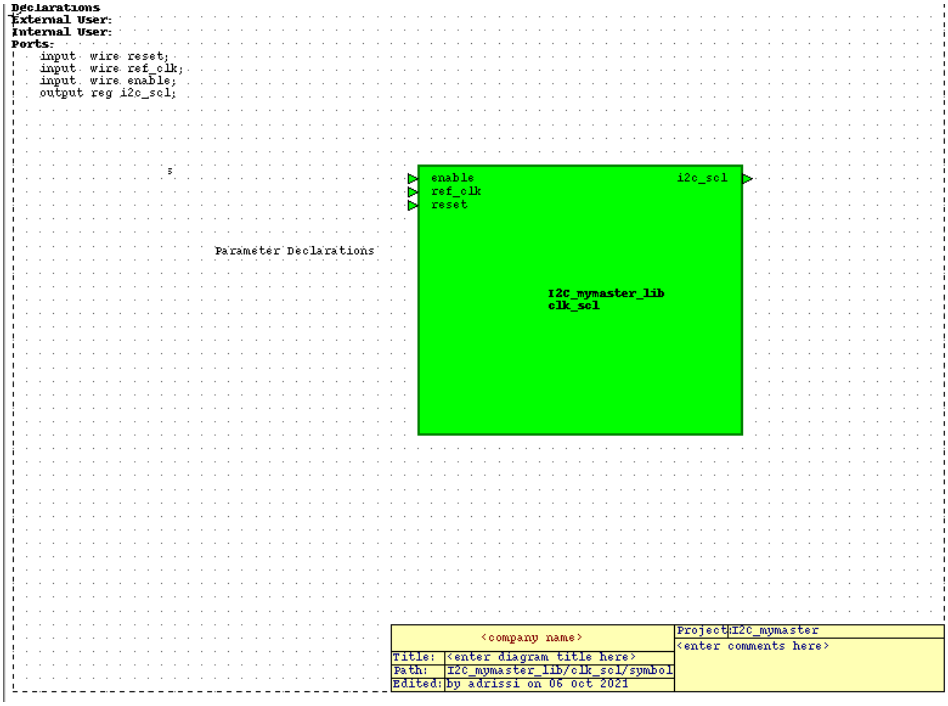

Abbildung 29: Clock-Divider Diagramm

5.2.3 Do_shift_read(Schiebe-Register Eingang)

Direkt an SDA angeschlossen, schiebt dieses Modul den aktuellen Wert von SDA mit der positiven Flanke von SCL in ein 8-Bit bereits Register. Geschoben wird der aktuell empfangene Wert in das LSB (Least Significant Bit) d.h das niedrigstwertiges Bit. Dadurch wird das bisherige MSB (Most Significant Bit)d.h das höchstwertiges Bit aus dem Register befördert. Für die korrekte Weiterverarbeitung der Daten reagiert der Prozess auf SCL. Auf diese Weise können Daten länger im Register gehalten werden, bis alle Verarbeitungsschritte durchgeführt worden sind. Das Signal Read_Done wird gesetzt, wenn acht Taktzyklen durchlaufen worden sind.

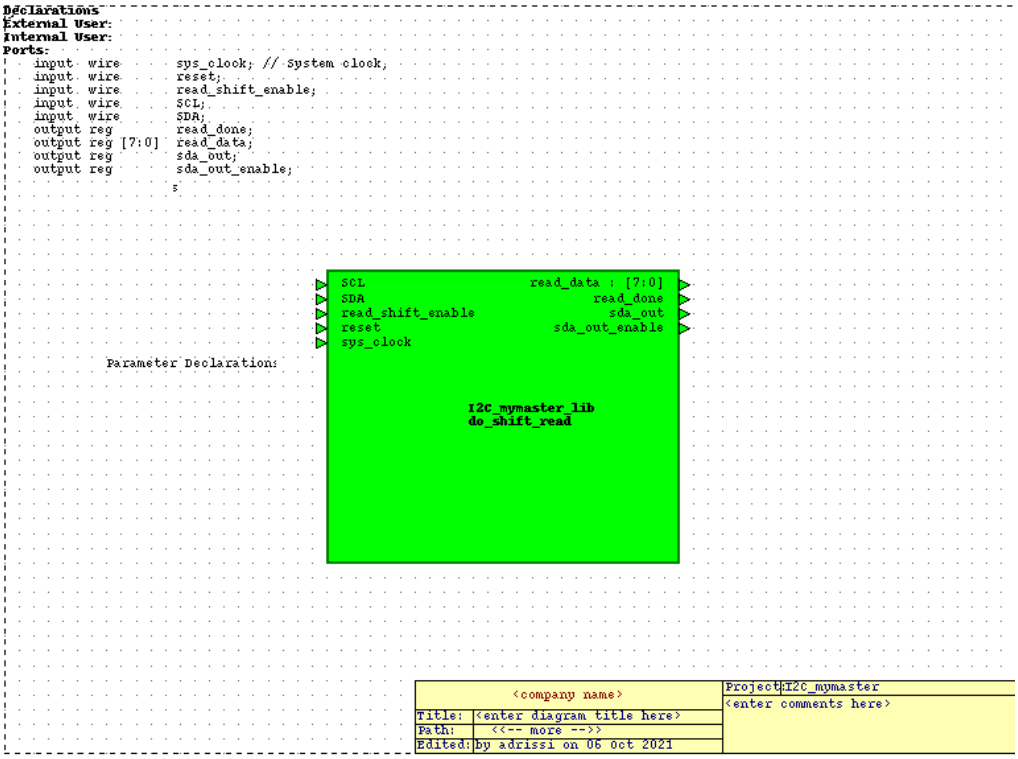

Abbildung 30: Do_Shift-Read Module Diagramm

5.2.4 Do_shift (Schiebe-Register Ausgang)

Das Modul DO_shift (Schiebe-Register Ausgang) übernimmt die Funktion der Datenübermittlung auf SDA. Aus einem 8-Bit-Register legt es dazu das MSB an den SDA Ausgang. Damit die Datenvalidität auf SDA nicht beeinflusst wird, schiebt es mit jeder negativen Flanke eine Null in das Register, an die Stelle des LSB. Während des Takt-Intervalls für das Acknowledge-Bit (9. Taktzyklus) dürfen keine Daten versendet werden. Dazu kann der Schiebvorgang mit dem Signal start_shift=0 unterbrochen werden. Gleichzeitig wird das Signal shift_done gesetzt und somit endet der Schiebvorgang.

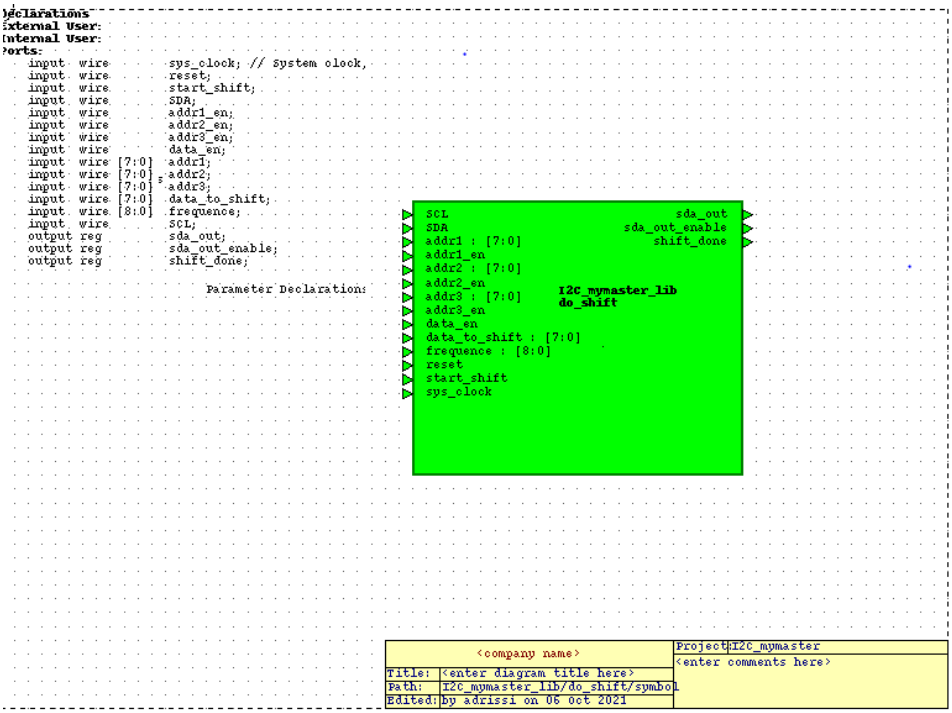

Abbildung 31: Shift Module für Write

5.2.5 Check_Ack

Das Modul check_ack übernimmt die Aufgabe, das Acknowledge-Bit (9. Taktzyklus) auszulesen und als Output an die Zustandsmachine zu übergeben.

Zur Signalisierung der durchzuführenden Schritte besitzt das Modul mehrere Inputs/Outputs, welche dem Bild 17 entnommen werden können:

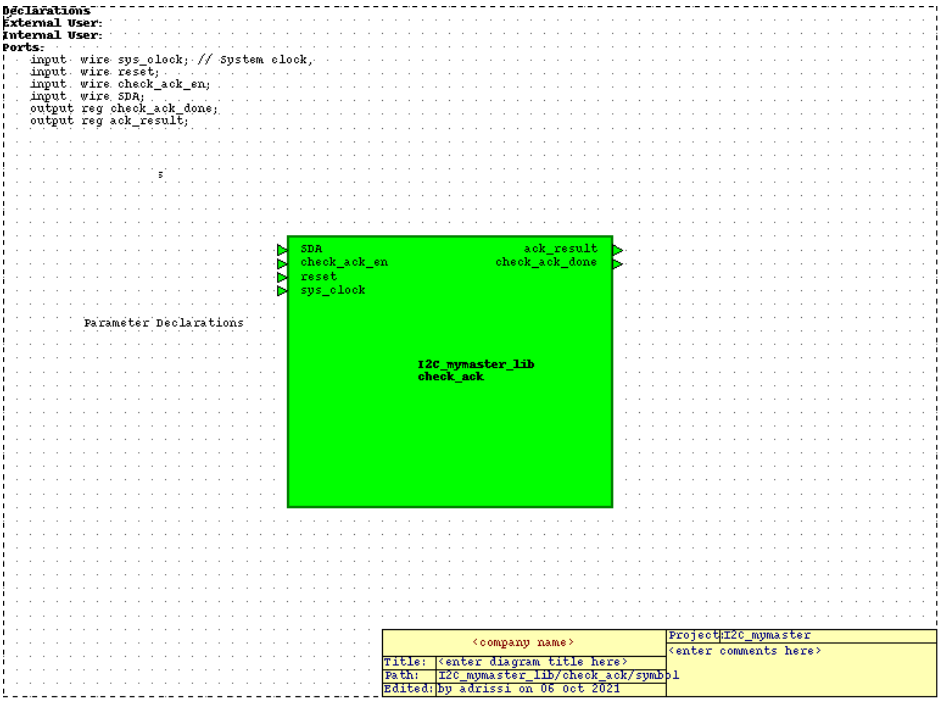

Abbildung 32: Check ACK Diagramm

5.3 Zustandsmaschine

Die Zustandsmaschine ist das umfangsreichste Modul im gesamten System. Sie übernimmt die Steuer- und Kontroll-Vorgänge für die Kommunikation auf dem I2C-Bus.

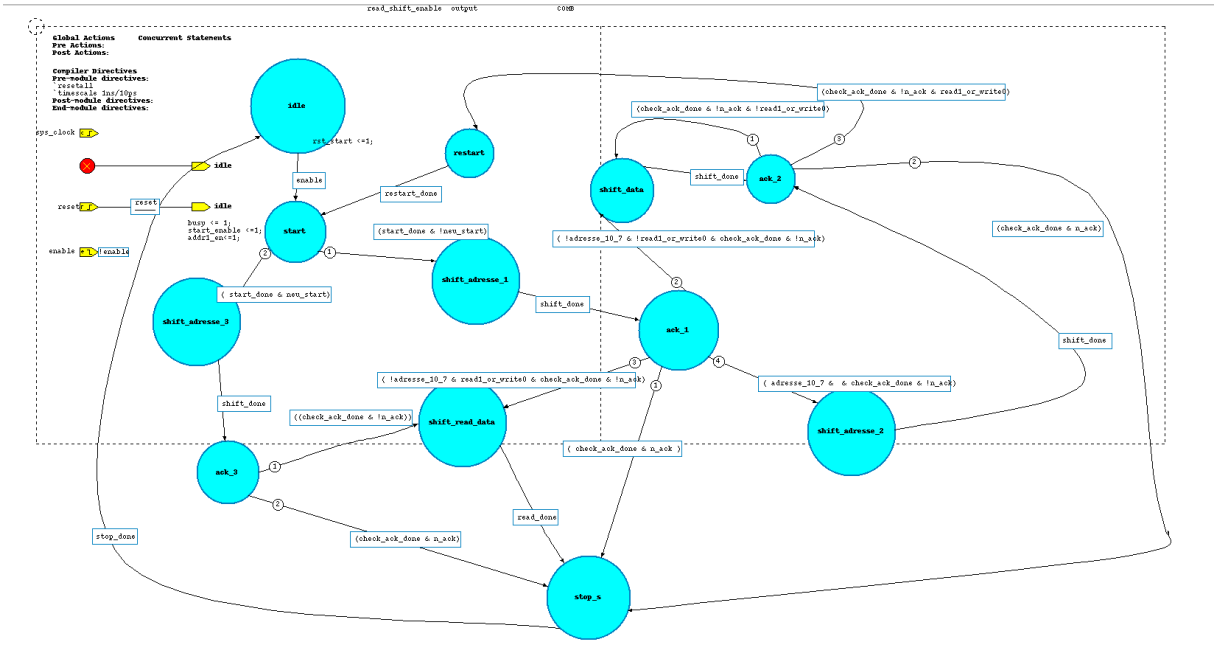

Abbildung 33: Finite State Machine

5.3.1 Allgemeiner Ablauf

Durch das asynchrone Reset-Signal ***Reset*** kann aus den nachfolgend beschriebenen Zuständen jederzeit in den Zustand ***IDLE*** gewechselt werden. Sobald das Start-Signal (Enable) auftritt, wechselt die Zustandsmaschine in den Zustand ***Start***. Das kann zu jedem Zeitpunkt einer Übertragung passieren. Auf den Zustand Start folgt der Wechsel entweder nach Shift_Adresse_1 oder nach shift_adresse_3. Wenn das Signal **neu_start** gesetzt ist, handelt es sich um einen wiederholten Start. Ist das Signal **neu_start** nicht gesetzt, wechselt die Zustandsmaschine in den Zustand Shift_Adresse_1. In diesem Zustand wird das erste Byte der Slave´s Adresse gesendet. Dort wartet die Zustandsmaschine, bis das Signal **shift_done** gesetzt ist, und wechselt dann in den Zustand **ack_1**. In diesem Zustand wird Ack (9.Bit) vom Bus gelesen. Ist das Ack-Bit gesetzt, folgt ein Stop Zustand und damit ein Stop des Transfers. Ist das Ack nicht gesetzt, wird in den entsprechenden Zustand gewechselt.

Bei einer 7-Bit-Adresse wird bei nicht gesetztem Signal **read1_or_write0** der Zustand **shift_data** erreicht. In diesem Fall möchte der Master Daten an den Slave schicken. Daher wird in den Zustand **Ack_2** gewechselt, in dem die Bestätigung der gesendeten Daten erfolgt.

Ist **read1_or_write0** gesetzt, wird bei Empfang eines nicht gesetzten n_ack in den Zustand **shift_read_data** gewechselt. In diesem Fall möchte der sendende Slave Daten an den Master schicken. Hier wird gewartet, bis das erste Byte vom Master empfangen wurde.

Handelt es um eine 10-Bit-Adresse**,** wird in den Zustand bei **shit_adresse2** gewechselt. In diesem Fall wird das zweite Teil der 10-Bit-Adresse gesendet. Ist das Signal ***shift_done*** gesetzt, wird in den Zustand **Ack_2** gewechselt, in dem die Bestätigung der gesendeten Daten erfolgt. Ist das Signal n_ack im 9. Zyklus gesetzt, wird der Transfer sofort gestoppt. Andersfalls wird in den entsprechenden Zustand gewechselt.

Für eine 10-Bit-Adresse ist das bei nicht gesetztem **read1_or_write0** der Zustand **shift_data**. In diesem Fall möchte der Master Daten an den Slave schicken. Daher wird in den Zustand **Ack_2** gewechselt, in dem die Bestätigung der gesendeten Daten erfolgt.

Ist das Signal **read1_or_write0** gesetzt, wird bei Empfang eines nicht gesetzten n_ack in den Zustand ***Restart*** gewechselt. Nach dem neuen Start wechselt die Zustandsmaschine in den Zustand ***Shift_Adresse3***. In diesem Fall wird gewartet, bis der dritte Teil der 10-Bit-Adresse gesendet und bestätigt wurde. Bei nicht gesetztem ***n_Ack*** im Zustand Ack_3, wird in den Zustand ***Shift_read_Data*** gewechselt. In diesem Fall möchte der sendende Slave Daten an den Master schicken. Hier wird gewartet, bis das erste Byte vom Master empfangen wurde.

Nach jedem Transfer der Daten/Adressen, empfängt der Master beim 9.Zyklus ein Ack vom Slave als Bestätigung. Sollte das Ack_Signal nicht gesetzt sein, wird ein Stopp des Transfers gefordert und die Zustandsmaschine wechselt in den Zustand ***IDLE***.

5.3.2 Eingangssignale der Zustandsmaschine

Zur Signalisierung der durchzuführenden Schritte besitzt die Zustandsmaschine mehrere Inputs. Mit ihnen können die Bedienungen der Zustandsübergänge gestaltet werden. Die Inputs können der folgenden Tabelle 5 entnommen werden:

Tabelle 5: Inputs der Zustandsmaschine

Input-Port	Bemerkungen
Sys_clock	Clock-Signal
reset	Reset-Signal, asynchron
Start_done	Das Signal bestätigt, dass der Transfer gestartet wurde.
Enable	Das Signal erlaubt den Start des Transfers.
Shift_done	Gibt bekannt, dass der Shift erfolgt ist.
Stop_done	Transfer beendet.
Read_done	Wird gesetzt, wenn das Byte aus dem Slave ausgelesen wurde.
Restart_done	Restart bestätigt.
Check_ack_done	Ack wurde empfangen.
Adresse_10_7	Gibt bekannt, ob es sich um eine 10 oder 7 Bit Adresse handelt.

n_ack	Wert des Ack-Signal
Read1_or_write0	Gibt bekannt, ob es sich um ein Read oder Write handelt.

5.3.3 Ausgangssignale der Zustandsmaschine

Damit zu keinem Zeitpunkt ein nicht definierter Zustand an einem Ausgang der Zustandsmaschine entsteht, lassen sich Werte für die Ausgänge vordefinieren. Einen Überblick auf die Ausgangssignale liefert auf die Ausgangsignale die Tabelle 6.

Tabelle 6: Outputs der Zustandsmaschine

Output-Port	Bemerkungen
Start_enable	Enable Start Module
Busy	Transfer ist gestartet. Die Adresse/Data dürfen nicht geändert werden.
Restart_enable	Restart Enable.
Rst_shift	Reset für Shift Module.
Addr1_en	Erlaubt das Versenden von Adresse 1.
Addr2_en	Erlaubt das Versenden von Adresse 2.
Addr3_en	Erlaubt das Versenden von Adresse 3.
Data_en	Erlaubt das Versenden von Daten.
Start_shift	Transfer darf starten.
Check_ack_enable	Acknowledgement prüfen.
Read_shift_enable	Erlaubt die Datenauslesung des Slaves.
Stop_enable	Transfer wird gestoppt.
Sda_out	SDA Output.
Sda_out_enable	SDA Output Enable.

6. JTAG: Beschreibung des Schaltungsentwurfs

In diesem Kapitel wird das JTAG-Design vorgestellt, das alle in früheren Abschnitten beschriebenen Spezifikationen erfüllt. Das JTAG-Design stellt sicher, dass alle notwendigen Funktionen und Eigenschaften des Systems vorliegen, um die JTAG-Spezifikationen umzusetzen. Es ist wichtig, dass das JTAG-Design alle Anforderungen des Standards erfüllt, um sicherzustellen, dass das System ordnungsgemäß funktioniert und alle notwendigen Funktionen bereitstellt.

6.1 Grundlegender Aufbau

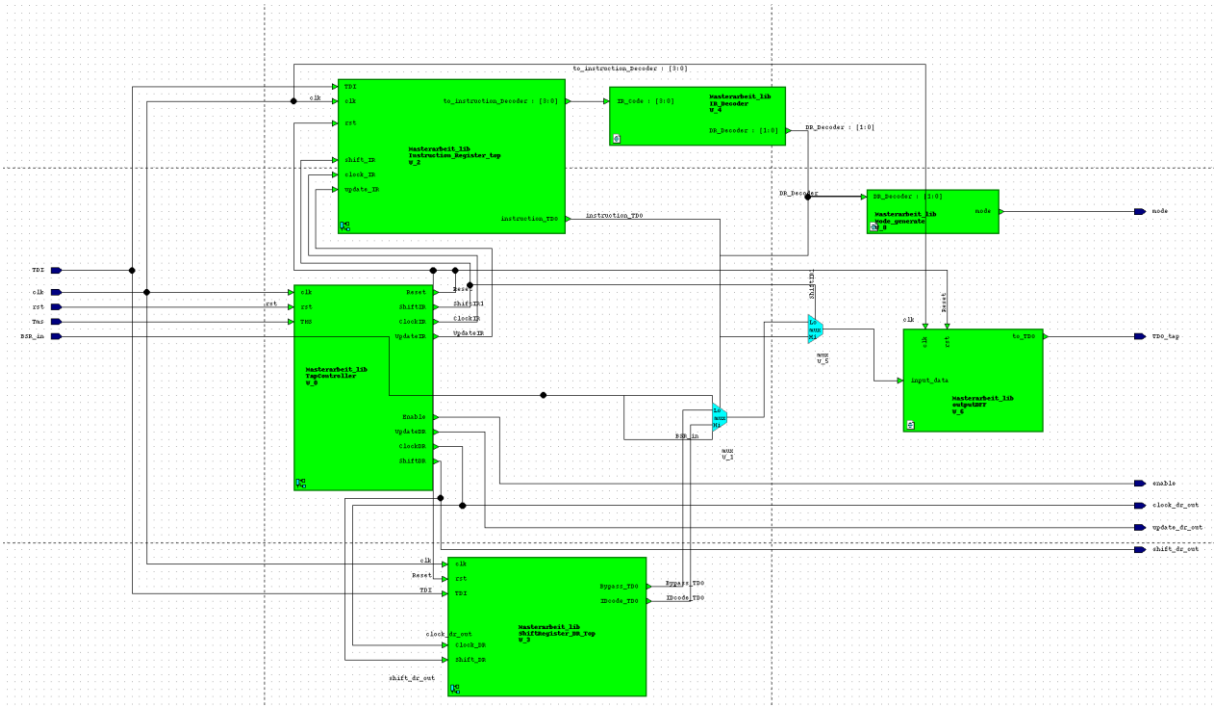

Abbildung 34: TAP_Top-Modul des Implementierten JTAG-Standards

Das TAP_TOP-Modul ist das oberste Element in der Hierarchie des JTAG-Standards und ist in Abbildung 34 dargestellt. Es ist für den Datenaustausch zwischen Sende- und Empfangsmodulen auf dem Chip verantwortlich und muss dabei die Regeln des JTAG-Protokolls beachten. Der Verilog-Code für alle Module in diesem Design befindet sich im Anhang A1.

Das TAP_TOP-Modul enthält alle Komponenten, die für eine ordnungsgemäße und protokollkonforme Funktion des JTAG-Systems erforderlich sind, mit Ausnahme des Boundary-Scan-Registers, das in einem anderen Modul implementiert wird. Alle Module im TAP_TOP-Modul werden von einem gemeinsamen Taktsignal gesteuert, um Synchronisationsprobleme zu vermeiden. Es ist wichtig zu beachten, auf welche Flanke des Taktsignals (Clock) jedes Modul reagiert, da dies von Modul zu Modul unterschiedlich ist. Diese Unterschiede können sich auf die Funktion der Module auswirken und müssen daher sorgfältig beachtet werden, um sicherzustellen, dass das JTAG-System ordnungsgemäß funktioniert.

Jedes Modul besitzt zusätzlich ein asynchrones active-low Reset (nRST) Signal.

Die Folgende Tabelle 7 zeigt die In/Output des TAP_TOP Moduls:

Tabelle 7: Ein- /Ausgangssignale des TAP_TOP-Modul

Signal	Beschreibung
Input wire clk	Clock signal
Input wire rst	Reset Signal
Input wire TDI	Test Daten Input
Input wire TMS	Testmodusauswahl
Input wire BSR_in	Empfangenes Signal aus Boundary-Scan-Register
Output wire TDO_tap	Output signal des JTAG-Standards
Output wire Clock_dr_out	Clock-Dr, wird an BSR-Zellen versendet.
Output wire Shift_dr_out	Shift-DR, wird an BSR-Zellen versendet.
Output wire update_dr_out	Update-DR, wird an BSR-Zellen versendet.
Mode	Wird vom Instruktionsdecoder gesetzt und an BSR-Zellen versendet.

Der TAP_TOP Modul besteht aus den folgenden Sub-Modulen/, die im Folgenden näher erläutert werden:

- TapController
- Instruction_Register_top (Befehlsregister)
- IR_Decoder (Instruktionsdecoder)
- ShiftRegister_DR_TOP (Datenregister)
- Mode_generate (generiert Modi für das BSR)
- outputDFF (Output Flipflop)
- Multiplexer für Datenregister (BSR & Datenregistern)
- Multiplexer für Datenregister & Instruktionsregister

6.2 TapController

Der TAP-Controller ist ein wichtiger Bestandteil des JTAG (Joint Test Action Group) Standards, der für die Diagnose und das Testen von integrierten Schaltkreisen verwendet wird. Er ist eine Zustandsmaschine, die das Verhalten des JTAG-Systems steuert und sicherstellt, dass die Übertragung von Daten und Kommandos zwischen dem integrierten Schaltkreis und einem externen Gerät, wie einem Computer oder einem anderen integrierten Schaltkreis, korrekt funktioniert. Der TAP-Controller koordiniert die Kommunikation zwischen den verschiedenen Komponenten des JTAG-Systems und sorgt dafür, dass alle Anweisungen und Daten korrekt

übertragen werden. Er wird mittels des TMS-Signals (Test Mode Select) gesteuert, das verwendet wird, um den Zustand des TAP-Controllers zu ändern und den Übergang zwischen den verschiedenen Zuständen zu steuern. Der TAP-Controller spielt eine zentrale Rolle bei der Funktionsweise des JTAG-Systems und ist von entscheidender Bedeutung für die korrekte Übertragung von Daten und Kommandos.

6.2.1 TAP-Blockdiagramm

Abbildung 35 zeigt das Modul "TapFSM" mit den Eingang- und Ausgangssignalen der Zustandsmaschine. Dieses Modul stellt die Zustandsmaschine des TAP-Controllers dar und verarbeitet die Eingangs- und Ausgangssignale, um das Verhalten des JTAG-Systems zu steuern und sicherzustellen, dass die Übertragung von Daten und Kommandos zwischen dem integrierten Schaltkreis und einem externen Gerät korrekt funktioniert. Die Eingangssignale werden verwendet, um den Zustand der Zustandsmaschine zu ändern und den Übergang zwischen den verschiedenen Zuständen zu steuern, während die Ausgangssignale verwendet werden, um das Verhalten des JTAG-Systems zu kontrollieren und sicherzustellen, dass alle Anweisungen und Daten korrekt übertragen werden.

Die Zustandsmaschine der TapController besteht ausschließlich ausfolgenden zwei Eingang-Signalen:

- TMS: Test Mode Select
- TCK: Clock

Ein Optional Signal ist das nRST (Reset).

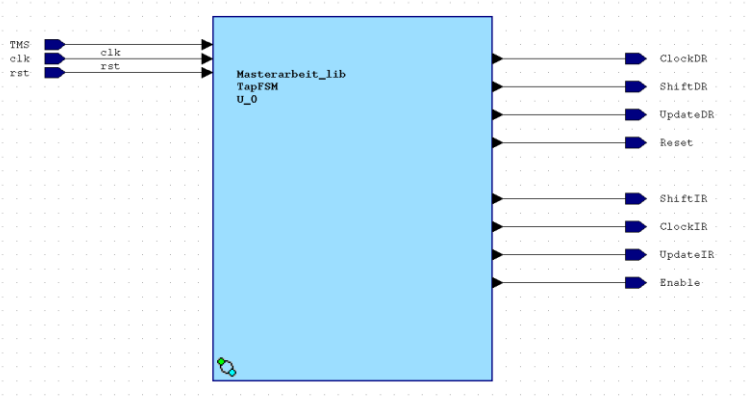

Abbildung 35: TapController Diagramm in Hdl-Designer

Die folgende Tabelle 8 zeigt die Ausgang-Signale der Zustandsmaschine und deren Funktionalitäten:

Tabelle 8: Ein- /Ausgangssignale der Zustandsmaschine

Signale	Bemerkungen
ClockDR	Die drei Signale steuern das Daten-Register/Boundary-Scan-Register
ShiftDR	
UpdateDR	
Enable	Erlaubt das Setzender Output-Signale
Reset	Setzt das System zurück
ClockIR	Die IR-Signale Steuern das Befehlsregister
ShiftIR	
UpdateIR	

6.2.2 Zustandsmaschine

Die Zustandsmaschine in Abbildung 36 ist ein wichtiges Element des JTAG-Systems und ist für die Steuerung und das Verhalten des Systems verantwortlich. Sie verfügt über zwei Ausgänge für jeden Zustand, wodurch alle Übergänge über das TMS-Signal gesteuert werden können, das auf die steigende Flanke des Taktsignals (Clock) abgetastet wird. Die beiden Hauptpfade ermöglichen das Setzen oder Abrufen von Informationen aus dem Datenregister oder dem Befehlsregister des Geräts. Der betriebene Datenregister hängt vom Wert ab, der in das Befehlsregister geladen worden ist. Die Zustandsmaschine ist das umfangreichste Modul im gesamten JTAG-System und ist von entscheidender Bedeutung für die Funktion des Systems.

Die implementierte Zustandsmaschine ist ein endlicher Automat, der nach dem Moore-Modell funktioniert. Dies bedeutet, dass die Ausgaben der Maschine ausschließlich vom aktuellen Zustand abhängen und nicht von den Eingangsdaten. Der Automat hat festgelegte Zustände und Übergänge zwischen diesen Zuständen, die von den Eingangsdaten gesteuert werden.

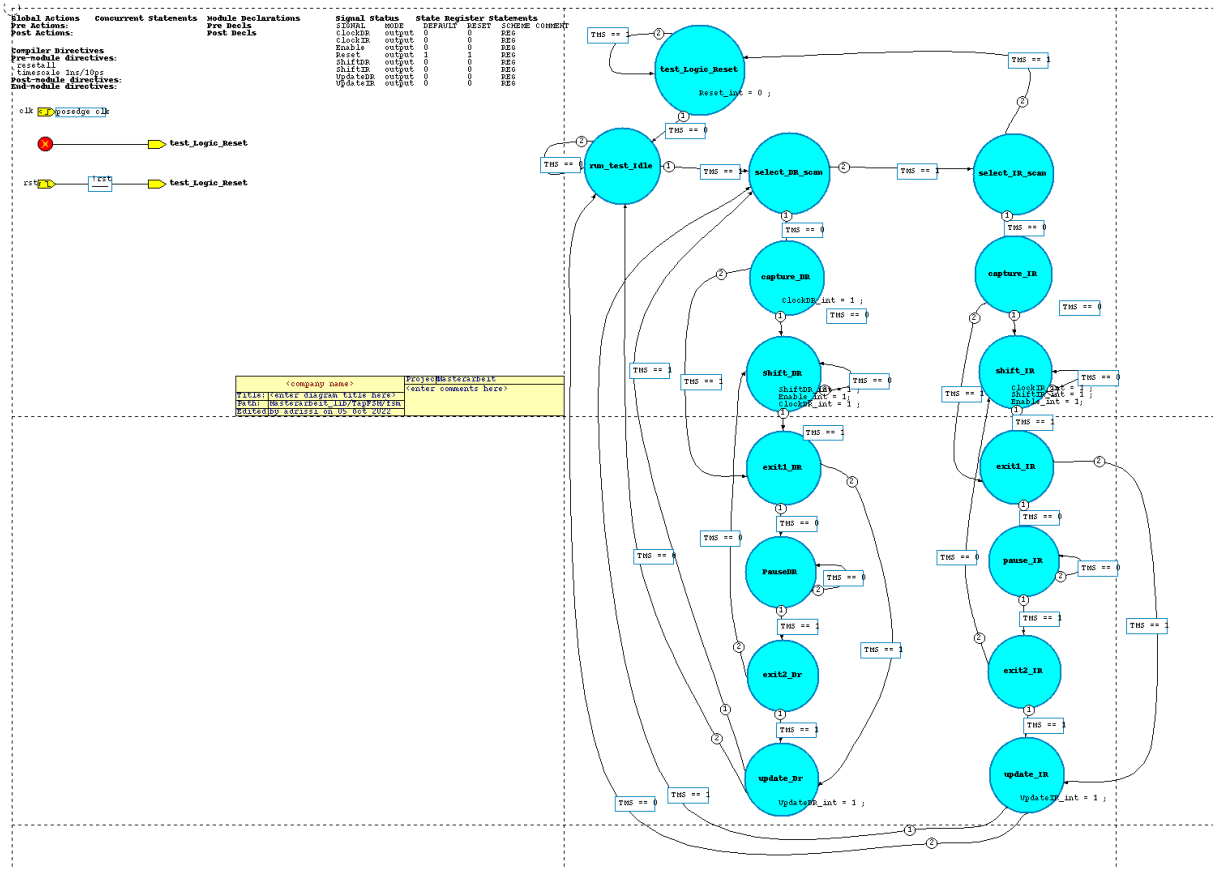

Abbildung 36: Zustandsmaschine des JTAG-Systems

Das asynchrone Reset-Signal (rst) ermöglicht es, aus jedem Zustand des TAP-Controllers in den Zustand test_Logic_Reset zurückzukehren. Es ist wichtig zu beachten, dass der TAP-Controller in jedem Zustand, in dem er sich befindet, zum Zustand test_Logic_Reset zurückkehren muss, wenn das TMS-Signal für fünf aufeinanderfolgende Taktzyklen (TCK) auf 1 gesetzt wird. Wenn das TMS-Signal auf 0 gesetzt wird, wechselt der TAP-Controller in den Zustand run_test_idle und bleibt dort, solange das TMS-Signal auf 0 gesetzt bleibt. Wenn das TMS-Signal auf 1 gesetzt wird, wechselt der TAP-Controller in den Zustand select_DR_scan. Wenn sich der TAP-Controller im Zustand select_DR_scan oder select_IR_scan befindet und das TMS-Signal auf 0 gesetzt wird, wechselt er beim nächsten Taktzyklus entweder in den Zustand capture_DR oder capture_IR, je nachdem, in welchem Zweig er sich befindet. Wenn sich der TAP-Controller in einem dieser Zustände befindet, bleibt er dort, solange das TMS-Signal auf 0 gesetzt bleibt. Nach einem weiteren Taktzyklus wechselt er dann in den Zustand shift_DR oder shift_IR, je nachdem, in welchem Zweig er sich befindet. Während sich der TAP-Controller in einem dieser Zustände befindet, bleibt er dort, solange das TMS-Signal auf 0 gesetzt bleibt und wird durch das TCK-Signal getaktet.

Während sich der TAP-Controller im Zustand shift_DR oder shift_IR befindet, wird für jeden Taktzyklus beim Schreiben ein Datenbit über TDI oder beim Lesen über TDO in das ausgewählte Register geschoben. Nachdem der Shift der Daten abgeschlossen ist, wechselt der TAP-Controller in den Zustand exit1_DR oder exit1_IR. In diesem Zustand werden alle

parallel geladenen Daten im Register gehalten. Der nächste Zustand ist update_DR oder update_IR, in dem die Testdaten in Halteregister geladen werden. Danach kann der TAP-Controller entweder wieder in den Zustand run_test_idle wechseln oder zurück in den Zustand select_DR_scan gehen, je nachdem, welche Aktion als nächstes ausgeführt werden soll.

6.3 Instruction_Register_top (Befehlsregister)

Das Befehlsregister ist ein spezielles Register, das im JTAG-System verwendet wird, um die aktuellen Anweisungen zu speichern, die vom TAP-Controller genutzt werden, um zu entscheiden, wie mit empfangenen Signalen verfahren werden soll. Es wird hauptsächlich herangezogen, um festzulegen, an welches Datenregister die Signale weitergeleitet werden sollen.

Das Modul verfügt über mehrere Eingänge und Ausgänge, die zur Signalisierung der Instruktionen verwendet werden und in Abbildung 37 dargestellt sind.

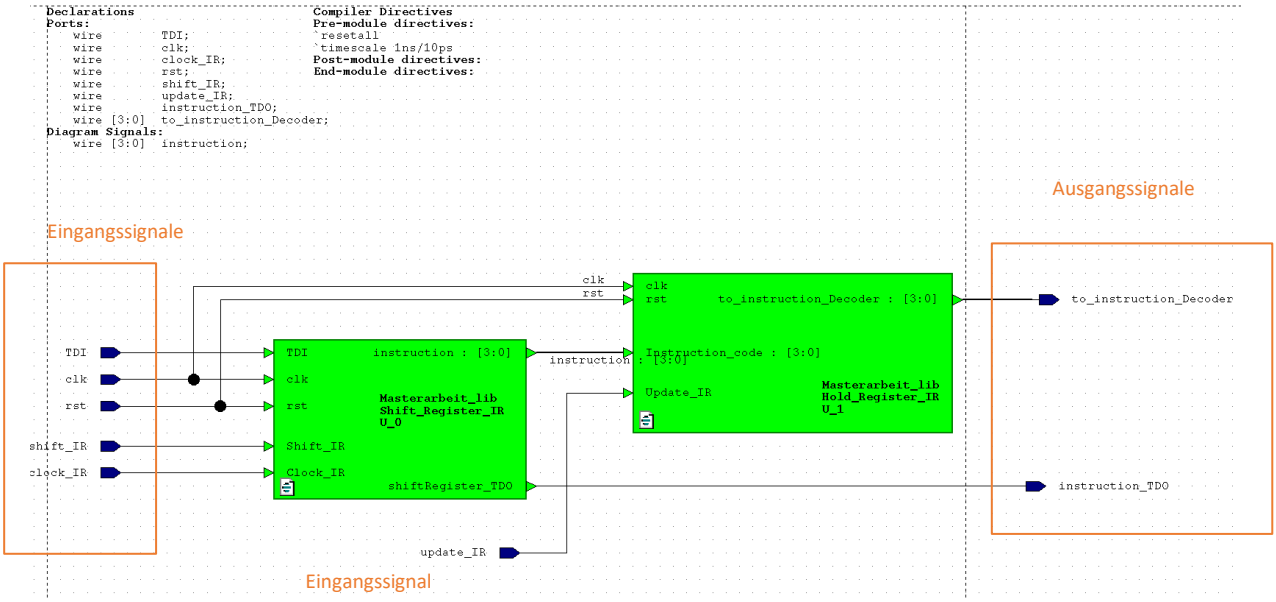

Abbildung 37: Überblick über das implementierte Instruktionsregister

Der Modul Instruction_Register_top besteht aus zwei Komponenten:

- **Shift_Register_IR**: in diesem Modul werden die Eingangsdaten über TDI solange die in das Instruktion-Register (4Bits Register) geschoben wie die Signale Shift_IR & Clock_IR gesetzt sind. Dabei wird das LSB (Niedrigstwertige Bit instruction [0]) dem Output-Signal instruction_TDO zugewiesen.

- **Hold_Register_IR**: in diesem Modul wird der aktuelle Befehl in einem Halteregister gespeichert. Ist das Update_IR gesetzt, wird der vorgeladene Befehl über das Ausgangregister to_instruction_Decoder an den Befehlsdecoder weitergegeben.

6.4 IR_Decoder

Nachdem der Test-Befehl geladen wurde, soll der Befehls-Dekoder ihn dekodieren und das Ergebnis an den Multiplexer weiterleiten.

```
//
// Verilog Module Masterarbeit_lib.IR_Decoder
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 17:07:08 08/04/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps

module IR_Decoder (
  // Port Declarations
   input   reg[3:0]  IR_Code,
   output  reg [1:0] DR_Decoder
   );


reg [1:0] current_Register;
reg TempOut;
parameter
          Extest_Enable= 4'b0000,
          Bypass_Enable = 4'b1111,
          Sample_Preload = 4'b0001,
          IDCODE_Enable = 4'b0010;


always@(IR_Code)
begin

  case(IR_Code)
    Extest_Enable : current_Register =2'b00;
    Bypass_Enable : current_Register =2'b01;
    IDCODE_Enable : current_Register = 2'b10;
    Sample_Preload : current_Register = 2'b11;
    default: current_Register = 1'bx; //Bypass
  endcase
end

assign DR_Decoder = current_Register;


// ### Please start your Verilog code here ###

endmodule
```

Abbildung 38: Verilog-Code des Implementierten IR-DEKODER

Abbildung 38 zeigt ein Modul, das einen Befehls-Dekoder (IR) implementiert. Es besitzt zwei Eingang- und Ausgangsports:

- IR_Code: ein 4-Bit-Register, das den IR-Code enthält.
- DR_Decoder: ein 2-Bit-Register, das die decodierte Anweisung enthält

Das Modul hat vier Parameter:

- Extest_Enable: eine 4-Bit-Konstante, die den IR-Code für die Anweisung "Extest Enable" darstellt.
- Bypass_Enable: eine 4-Bit-Konstante, die den IR-Code für die Anweisung "Bypass Enable" darstellt.
- Sample_Preload: eine 4-Bit-Konstante, die den IR-Code für die Anweisung "Sample Preload" darstellt.
- IDCODE_Enable: eine 4-Bit-Konstante, die den IR-Code für die Anweisung "IDCODE Enable" darstellt.

Innerhalb des Moduls gibt es einen Always-Block (Prozess), der empfindlich auf Änderungen am IR_Code-Eingang reagiert. Innerhalb des Prozesses gibt es eine Fallunterscheidung, die den Wert von IR_Code mit den in den Parametern definierten Konstanten vergleicht. Je nach Wert von IR_Code wird der current_Register-Variable ein anderer 2-Bit-Wert zugewiesen.

Schließlich wird der DR_Decoder-Ausgangsport dem Wert von current_Register zugewiesen.

6.5 ShiftRegister_DR_Top (Datenregister)

Es gibt hauptsächlich drei Datenregister: das Boundary-Scan-Register (BSR), das BYPASS-Register und das IDCODES-Register. Es können weitere Datenregister vorhanden sein, die jedoch nicht als Teil des JTAG-Standards erforderlich sind.

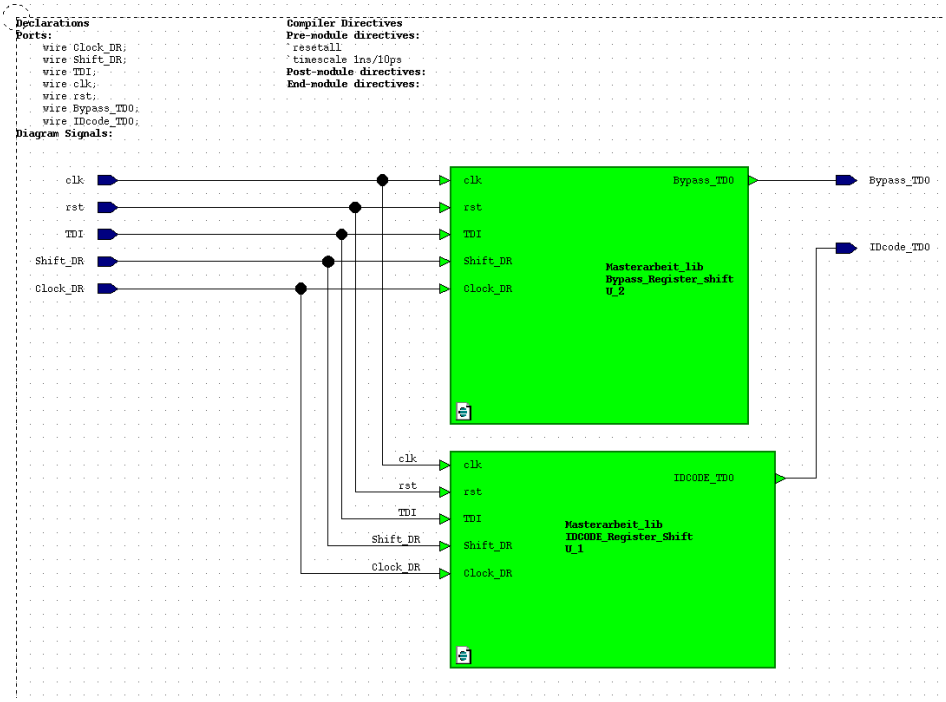

Abbildung 39: Überblick über das implementierte Daten-Register

Das ShiftRegister_DR_Top besteht aus zwei Modulen (Abbildung 39):

- **Bypass_Register_Shift**: Dies ist ein Ein-Bit-Register, das zwischen TDI und TDO platziert wird und ermöglicht, andere Bauteile auf dem gleichen PCB mit minimalem Zeitaufwand zu testen. Das Modul besteht aus fünf Eingangssignalen (Systemclock, Reset, TDI, Shift_DR und Clock_DR) und einem Ausgang (Bypass_TDO). Der Prozess reagiert auf CLK und Rst. Die Schieboperation kann erst starten, wenn die Signale Clock_DR und Shift_DR von der Zustandsmaschine gesetzt sind.

- **IDCODE_Register_Shift**: Dieses Register enthält den Identifikationscode des Geräts. Das Modul besitzt fünf Eingangssignale *Systemclock, Reset, TDI, Shift_DR und Clock_DR* und ein Ausgangssignal *IDCODE_TDO*. Der im Modul verwendete Prozess reagiert auf *CLK* und *Rst*. Die ID-Code-Ausgabe kann erst starten, wenn die Signale Clock_DR und Shift_DR von der Zustandsmaschine gesetzt sind.

Es ist wichtig zu beachten, dass das Boundary-Scan-Register zwar Teil der Datenregister ist, aber nicht Bestandteil des JTAG-darstellt.

6.6 Die Boundary Scan Registers (BSR)

Ein Boundary Scan Register (BSR) ist eine Funktion des Joint Test Action Group (JTAG) - Interfaces, die es ermöglicht, elektronische Schaltungen zu testen und zu debuggen. Der BSR besteht aus einer Reihe von Registern, die über das JTAG-Interface zugänglich sind und zum Steuern und Beobachten von Ein-/Ausgangs-Pins (I/O-Pins) von elektronischen Geräten verwendet werden.

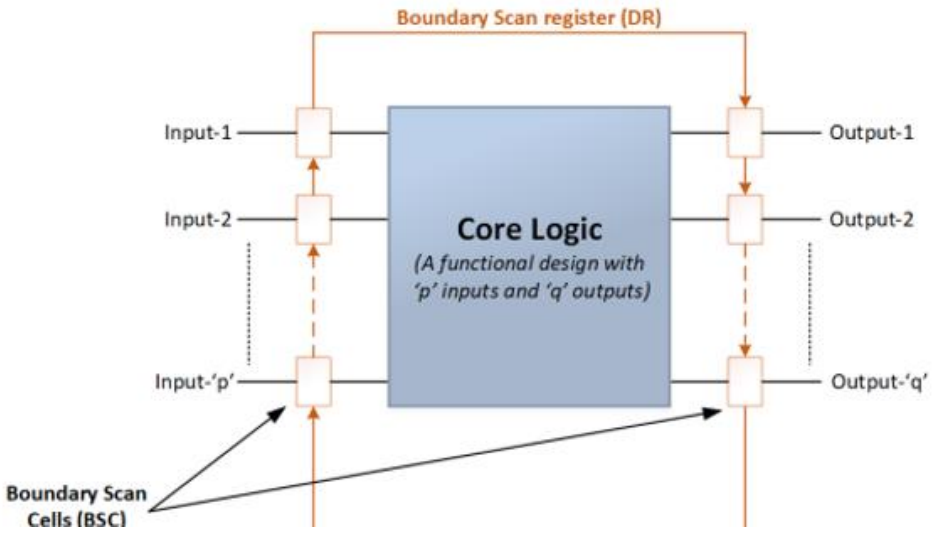

Abbildung 40: Architektur des Boudary-Scan-Register [3]

Ein der Hauptvorteil bei der Verwendung von BSR besteht darin, dass möglicht ist, Schaltungen zu testen und zu debuggen, auch wenn die I/O-Pins der Geräte nicht sichtbar oder leicht zugänglich sind. Dies ist besonders nützlich beim Testen von Schaltungen, die aus mehreren Chips oder Geräte bestehen, da der Test von Verbindungen zwischen den Chips möglich ist.

Neben dem Test und des Debuggings kann der BSR auch zum Fertigungstest und zur In-System-Programmierung von Schaltungen verwendet werden.

6.6.1 Boundary-Scan-Zelle

Der BSR wird in der Regel mit Boundary Scan Register Cells (BSRCs) implementiert, die aus Flip-Flops und Multiplexer bestehen. Das Flip-Flop wird verwendet, um den Wert des BSR für einen bestimmten I/O-Pin zu halten, und der Multiplexer ermöglicht es, den Wert entweder vom I/O-Pin zu erfassen oder den I/O-Pin zu treiben. Das Flip-Flop kann über das JTAG-Interface gesteuert und beobachtet werden, was den Test und das Debugging der Schaltung ermöglicht. Abbildung 41.

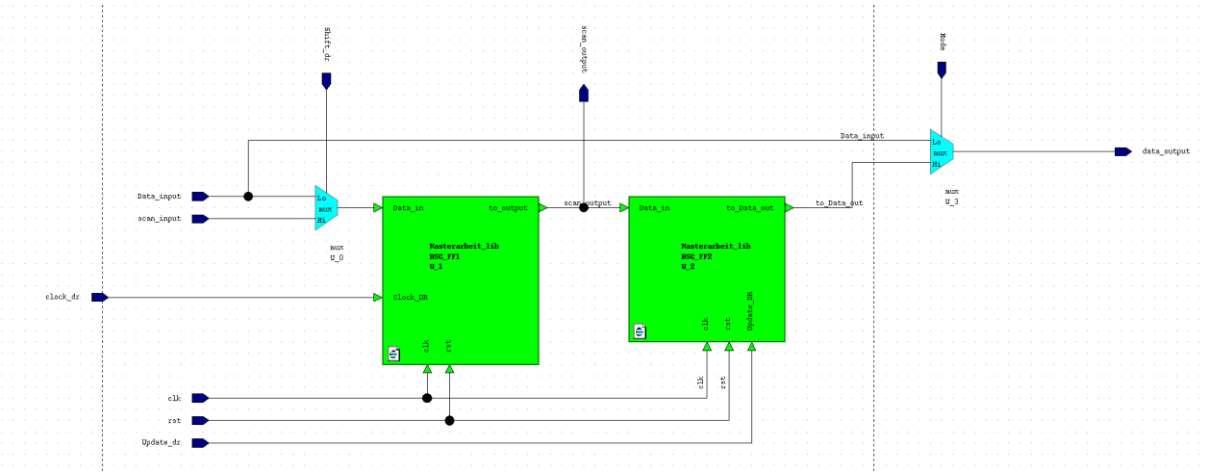

Abbildung 41: Überblick über das implementierte Boundary-Scan-Zelle

Das Modul verfügt über mehrere Eingangs- und Ausgangssignale:

- *Data_input*: Dies ist der Eingabedatensatz für die BSC.
- Mode: Dies ist ein Steuersignal, das bestimmt, ob die BSC im Normalmodus (wenn Mode 0 ist) oder im Testmodus (wenn Mode 1 ist) arbeitet. Im Normalmodus leitet die BSC die *Data_input* direkt an die *data_output* weiter. Im Testmodus verwendet die BSC den im Flip-Flop gespeicherten Wert, um den *data_output* zu treiben.
- *Shift_dr*: Dies ist ein Steuersignal, das bestimmt, ob die BSC den im Flip-Flop gespeicherten Wert verschiebt (wenn Shift_dr 1 ist) oder nicht (wenn Shift_dr 0 ist).
- *Update_dr*: Dies ist ein Steuersignal, das bestimmt, ob der im Flip-Flop gespeicherte Wert von scan_input aktualisiert wird (wenn Update_dr 1 ist) oder nicht (wenn Update_dr 0 ist).
- *clock_dr*: Dies ist ein Steuersignal, das bestimmt, ob der im Flip-Flop gespeicherte Wert mit der steigenden Flanke des *clk*-Signals aktualisiert wird (wenn *clock_dr* 1 ist) oder nicht (wenn *clock_dr* 0 ist).
- rst: Dies ist das Reset-Signal für die BSC. Wenn rst aktiviert wird, setzt die BSC den im Flip-Flop gespeicherten Wert auf 0 zurück.
- *scan_input*: Dies ist das Eingangssignal für den Scan-Datensatz.

- *data_output*: Dies ist der Ausgang der BSC. Das Signal entspricht entweder dem *Data_input* (im Normalmodus) oder dem im Flip-Flop gespeicherte Wert (im Testmodus).
- *scan_output*: Dies ist der Ausgang für den Scan-Datensatz. Das Signal entspricht entweder dem im Flip-Flop gespeicherten Wert (wenn Shift_dr 1 ist) oder dem scan_input (wenn Shift_dr 0 ist).

Innerhalb des Moduls gibt es zwei Instanzen von Untermodulen (Flip-Flops): BSC_FF1 und BSC_FF2. Diese Untermodule implementieren die Flip-Flop- und Multiplexerfunktionalität der BSC. Das Modul verfügt auch über zwei always-Blöcke, welche die Multiplexerfunktionalität mithilfe einer Fallunterscheidung implementieren. Der erste always-Block bestimmt den Wert des dout-Signals, das je nach Wert von *Shift_dr* entweder dem *Data_input* oder der *scan_input* entspricht. Der zweite always-Block bestimmt den Wert von data_output, der je nach Wert von Mode entweder dem Data_input oder dem to_Data_out-Signal entspricht.

Insgesamt implementiert dieses Modul eine BSC, die zum Test und Debugging von elektronischen Schaltkreisen über die JTAG-Schnittstelle verwendet werden kann.

6.7 Mode_generate

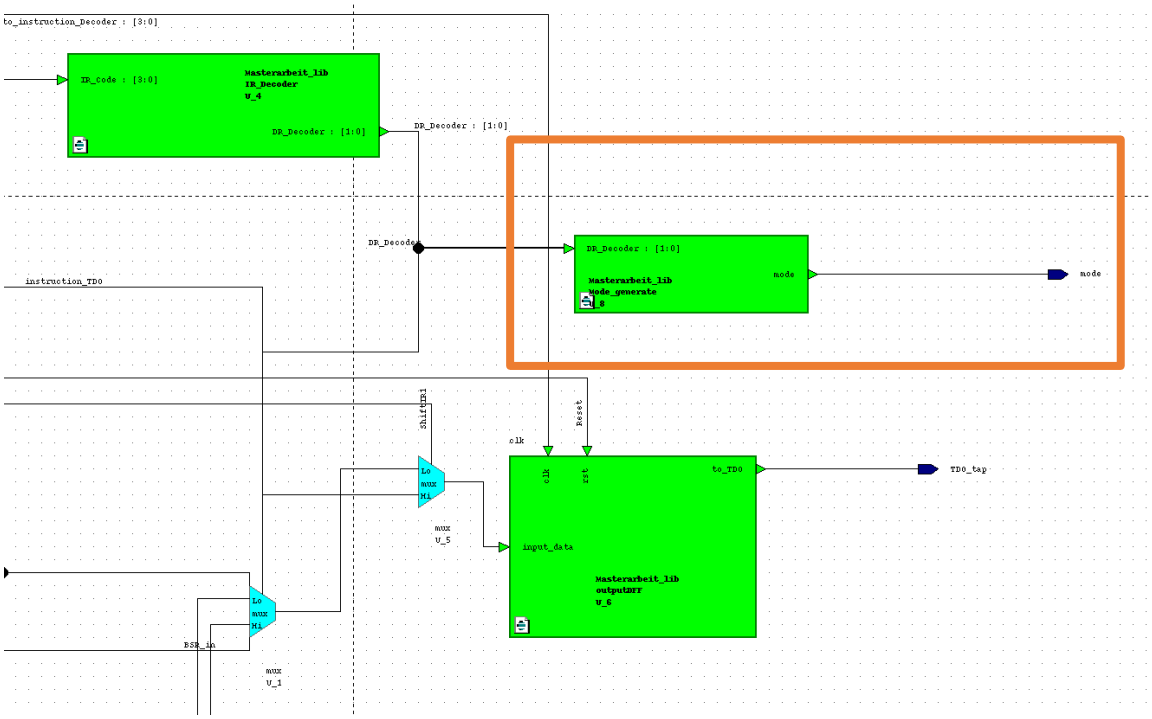

Abbildung 42: Überblick über das implementierte Modul Mode_Generate

Diese Komponente definiert ein Modul mit dem Namen Mode_generate, das einen Eingang und einen Ausgang besitzt. Der Eingang entspricht dem Signal *DR_Decoder* einem 2-Bit-Register, während der Ausgang dem 1-Bit Registersignal *mode entspricht.*

Das Modul enthält auch eine parameter-Deklaration, die zwei 4-Bit-Konstanten definiert, welche als Referenzen in der case-Anweisung verwendet werden, die im always-Block folgt.

Im always-Block wird eine case-Anweisung verwendet, um den Ausgang *TempOut* basierend auf dem Wert des Eingangs *DR_Decoder* festzulegen. Wenn *DR_Decoder* den Wert *Extest_Enable* annimmt, wird *TempOut* auf 1'b1 gesetzt. Wenn DR_Decoder den Wert sample_Enable besitzt, wird TempOut auf 1'b0 gesetzt. In allen anderen Fällen wird TempOut auf 1'bx gesetzt, was einen ungültigen Zustand darstellt.

Schließlich wird der Wert von TempOut dem Ausgang *mode* zugewiesen, indem der Ausdruck *assign mode = TempOut*; verwendet wird.

6.8 IR & DR Multiplexer

Im Tap_Top_Modul werden die zwei Multiplexer, "U_1" und "U_5" implementiert. Ein Multiplexer, oder "Mux" ist eine digitale Schaltung, die eines von mehreren Eingangssignalen auswählt und es an den einzigen vorhandenen Ausgang leitet(siehe Abbildung 43).

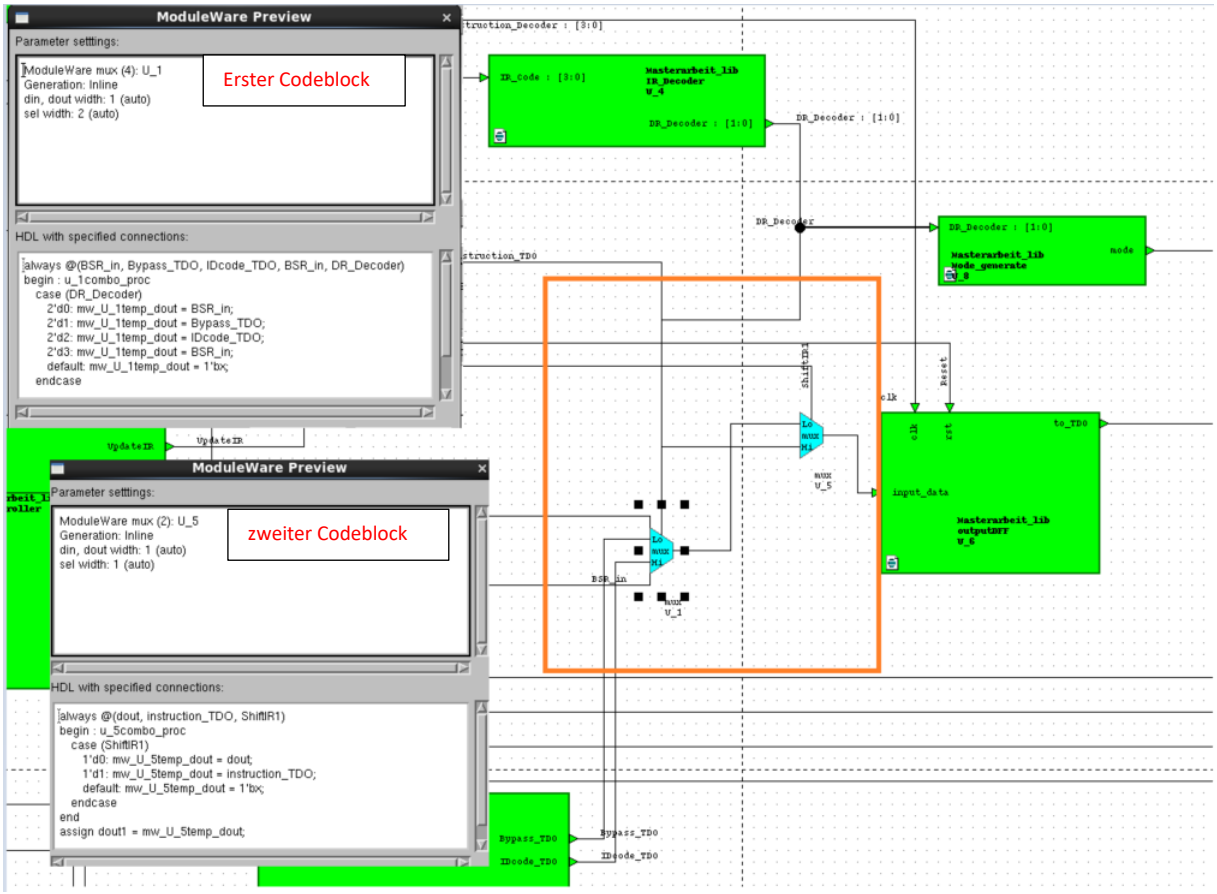

Abbildung 43: Überblick auf die implementierten Multiplexer

Im ersten Codeblock hat der Multiplexer "U_1" fünf Eingangssignale: *BSR_in*, *Bypass_TDO*, *IDcode_TDO*, *BSR_in* und *DR_Decoder*. Der "case"-Befehl innerhalb des "always"-Blocks gibt an, dass der Ausgang des Multiplexers, *mw_U_1temp_dout*, je nach Wert von *DR_Decoder* auf eines der Eingangssignale gesetzt wird. Wenn *DR_Decoder* 0 ist, dann wird *mw_U_1temp_dout* auf *BSR_in* gesetzt. Wenn *DR_Decoder* 1 ist, dann wird *mw_U_1temp_dout* auf *Bypass_TDO* gesetzt und so weiter. Der "default"-Fall am Ende gibt an, dass *mw_U_1temp_dout* auf den unbekannten Wert "1'bx" gesetzt wird, wenn *DR_Decoder* einen anderen Wert hat. Der Ausgang des Multiplexers, "dout", wird dann auf den Wert von *mw_U_1temp_dout* gesetzt.

Der zweite Codeblock für den Multiplexer "U_5" funktioniert auf ähnliche Weise. Die Eingangssignale für diesen Multiplexer sind *dout*, *instruction_TDO* und *ShiftIR1*. Der Ausgang des Multiplexers, *mw_U_5temp_dout*, wird aufgrund des Werts von *ShiftIR1* ausgewählt. Wenn *ShiftIR1* Null ist, dann wird *mw_U_5temp_dout* auf *dout* gesetzt. Wenn *ShiftIR1* Eins ist, dann wird *mw_U_5temp_dout* auf *instruction_TDO* gesetzt. Der "default"-Fall am Ende gibt an, dass *mw_U_5temp_dout* auf den unbekannten Wert "1'bx" gesetzt wird, wenn *ShiftIR1* einen anderen Wert besitzt. Der Ausgang des Multiplexers, *dout*, wird in diesem Codeblock nicht neu zugewiesen.

7. Testbench und Simulation des I2C-Master-Interface

7.1 Grundlegender Aufbau der Testbench

Am I2C-Master ist die Testbench als Teilnehmer angeschlossen. Einige Komponenten des I2C_Masters fanden auch Eingang in die Testbench. So ist der Zähler (Wait-of-Timer) verwendet und auch der Clock-Teiler eingebaut worden. Daneben tauchen auch die Schiebe-Register (Read/Write) wieder auf. Zusammenfassend kann gesagt werden, dass es sich um einzelne einfach strukturierte Module handelt und nicht um den Zusammenschluss vieler Module zu einer komplexen Einheit, die auf Grund der Unübersichtlichkeit fehlerbehaftet sein könnten. Natürlich lassen sich auch in dieser Anordnung Fehler nicht ausschließen, doch würden sie im Entwicklungsprozess der Testbench eher auffallen. Schließlich wird die Testbench mit dem Wissen enworfen, wie die Vorgänge korrekt ablaufen müssten. Sollte das beobachtete Verhalten dem nicht entsprechen, lassen sich Fehler in diesen Komponenten einfacher entdecken. Die Tatsache, dass es sich bei der Testbench um Module handelt, die nicht synthetisiert werden müssen, schlägt sich in den verwendeten Verilog Sprachkonstrukten nieder. Es werden nun einige zusätzliche Statements verwendet, die nicht synthetisierbar sind. Die wichtigsten sind:

- initial-Block

Legt definierte Signalpegel zu Beginn einer Simulation an die Ausgänge eines Moduls. Diese Werte können im späteren Verlauf von Zuweisungen überschrieben werden.

```
initial begin

  // Initialize Inputs
  sys_clock = 0;
  SDA_en_tb =0;
  reset = 1;
  SDA_en_tb =0;
  SDA_out_tb = 1;
  //SDA_en =1;
  read_en =0;
  //slave_address =7'b1010101;
  slave_address =10'b1001010110;
  write_data =8'b10100010;
  rd_or_wr = 1;  //read 1 / write 0
  adresse_10_7 =1;  // 0=7 // 1=10bits
  enable = 0;
  //Wait 100ns for global reset to finish
  #101;
  reset = 0;
  // Add stimulus here
  #30000
  @ (posedge sys_clock) enable <= #1 1;   //enable für Start  wird/soll Synchron zum Clock
  @ (posedge sys_clock) enable <= #1 0; // beim nächsten clock direkt wiedr auf 0
```

Abbildung 44: Initial Begin des Testbenches

- timescale-Direktive

Definiert die verwendeten Zeiteinheiten und steht zu Beginn der Verilog-Datei. In diesem Projekt wurde die Direktive folgendermaßen verwendet: timescale 1ns/1ps.

Dabei gibt die erste Zahl nach dem Schlüsselwort *timescale* die Zeiteinheit der Simulation an, und die zweite bestimmt die Auflösung. Wird die Auflösung erhöht, werden beispielsweise Verzögerungen exakter berechnet.

- Verzögerungsanweisungen

Diese werden durch eine Raute (#) mit nachfolgender Zahl eingeleitet. Die Zahl gibt dabei die Zeit der Verzögerung, bezogen auf die timescale-Direktive, an. Der Simulationsprozess legt die nachfolgenden Signale erst nach Ablauf dieser Zeit an. Durch die Angabe von „#1“ lässt sich z.B. eine Verzögerung von einer Zeiteinheit erreichen. Das bedeutet, dass alle nachfolgenden Anweisungen eine Zeiteinheit später ausgeführt werden. Die Verzögerungsanweisungen lassen sich auch hintereinander anordnen. Damit kann ein Modul einfach Signale zu unterschiedlichen Zeitpunkten setzen. Natürlich lassen sich diese Werte auch zur Laufzeit der Simulation mit Hilfe mathematischer Ausdrücke bestimmen.

7.2 Tri-State Buffer

Tri-State-Puffer können sich in einem von drei Zuständen befinden: Logisch 0, Logisch 1 und Z (hohe Impedanz). Ihre Verwendung ermöglicht es mehreren Treibern, sich eine gemeinsame Leitung zu teilen. In dem Projekt teilen der I2C_Master_Top und der Testbench die In/Out Leitung SDA.

Die Abbildung 45 zeigt ein Prinzipdiagramm, wie ein Chip-Ausgangsport eine Null, eine Eins oder Z erzeugt. In Wirklichkeit handelt es sich bei den Schaltern um MOSFETs.

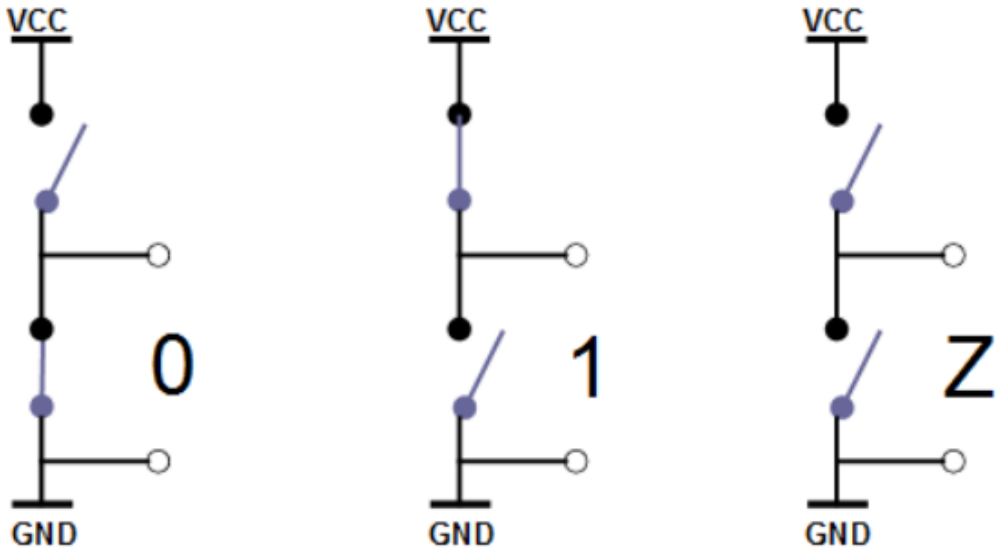

Abbildung 45: Tri-State Zustände

Die Tri-State-Signale werden nicht im Chip oder im FPGA verwendet, sondern in der Testbench, um Signale miteinander zu verbinden. Die Abbildung 45 zeigt zwei entworfene Tri-State-Gatter in der Testbench. Ein Tristate-Gatter kontrolliert der Zugriff der Testbench auf den SDA Bus. Das andere Tristate-Gatter wird für das Modul I2C_Master_Top benötigt.

```
assign SDA_in = SDA;
assign SDA = sda_out_enable ? sda_out : 1'bZ; //I2c_Master
assign SDA = SDA_en_tb ? SDA_out_tb : 1'bZ; //TestBench Tri-State

pullup (SDA);
```

Abbildung 46: Aufgebaute Tri-State in Testbench

7.3 I2c_Master_Top Instanzblock

Um die Komplexität der Testbench zu reduzieren, wurden die meisten externen Modulen im Modul I2c_Master_Top instanziiert. Der I2c_Master_Top erhält die Eingangssignale der Testbench. Diese Signale werden innerhalb des Moduls an die anderen externen Module weitergeleitet.

```
// Instantiate the Unit Under Test (UUT)
Master_i2c_TOP
#(
   .freq(100)
)
 uut
(
   .SDA            (SDA),
   .adresse_10_7(adresse_10_7),
   .sda_out_enable (sda_out_enable),
   .sda_out (sda_out),
   .enable(enable),
   .SCL            (SCL),
   .sys_clock      (sys_clock),
   .reset          (reset),
   .read_en(read_en),
   .slave_address(slave_address),
   .write_data  (write_data),
   .rd_or_wr    (rd_or_wr),
   .rd_data    (rd_data),
   .busy     (busy),
   .stop_from_master(stop_from_master),
   .address_ack (address_ack),
   .data_nack (data_nack),
   .data_rd_ack (data_rd_ack)

);
```

Abbildung 47: Instanz des I2c_Master_Top in Testbench

Die Hilfssignale und Register werden im Initial Block verwendet. Die wichtigen Signale sind:

Tabelle 9: Wichtige Signale in Testbench

Signale in Testbench	Bemerkungen
Sys_clock	Clock
Reset	Async Reset
SDA_en_TB	Aktiviert Testbench Zugriff auf SDA.
SDA_out_TB	Signal mit dem die Testbench auf SDA zugreift.
Read_en	Das Signal des Masters. Bei 1 ist der Master bereit Daten zu empfangen.
Slave_adresse	Slave Adresse (Entweder 7 oder 10 Bits)
Write_Data	Zu sendende Daten
Rd_or_wr	Bei einer Read-Operation ist eine 1 und bei einer Write-Operation eine 0.
Adresse_7_10	Bei einer 7Bit-Adresse ist eine 0 zu setzen. Eine 1 steht für eine 10Bit-Adresse.
Enable	Aktivierung des Transfers

7.4 Der Simulationsvorgang

Die fertiggestellte Testbench kann nun simuliert werden. Aus dem HDL-Designer kann ModelSim unmittelbar mit den Modulen der Testbench gestartet werden. Es bieten sich nun in dem Projekt die folgende Möglichkeit an:

- Grafische Analyse der Daten

  Dazu können die Verläufe bestimmter Signale in einer Waveform dargestellt werden. Die Signale lassen sich frei auswählen, so dass jeder einzelne Prozess genau verfolgt werden kann. Für eine längere Laufzeit ist dies jedoch ungeeignet, da die Waveform schnell unübersichtlich wird.

7.5 Simulation und Ergebnisse

Um die Funktionalität des Busses zu prüfen, wurden mehrere Fälle mit ModelSim getestet.

Die Tabelle 10, zeigt die wichtigen Signale, die in der Simulation verwendet werden:

Tabelle 10: Simulationssignle

Signale in	Bemerkungen
Sys_clock	Clock
Reset	Async Reset
Enable	Aktivierung des Transfers
Read_en	Wird gesetzt, wenn der Master bereit ist, die Daten des Slaves zu empfangen.
Write_Data	Zu sendende Daten
Slave_adresse	Slave Adresse (Entweder 7 oder 10 Bits)
Rd_or_wr	Bei einer Read-Operation ist eine 1 und bei einer Write-Operation eine 0.
Rd_data	In diesem Register werden die Daten, die aus dem Slave empfangen wurden, abgelegt.
Busy	Wird während des Transfers auf „1“ gesetzt .
SDA_in	SDA input
Sda_out	SDA Ausgang des Masters
Sda_out_enable	Erlaubt dem Master den Zugriff auf SDA.
SDA_en_TB	Erlaubt der Testbench den Zugriff auf SDA.
SDA_out_TB	Testbench schreibt ein Wert auf SDA.
Adresse_7_10	Bei einer 7Bit-Adresse ist eine 0 zu setzen. Eine 1 steht für eine 10Bit-Adresse.
SCL	SCL Signal
SDA	SDA Signal

1. Testdurchlauf: 7Bit Adresse Write, Ack-Adresse = 0

Bei diesem Test handelt es sich um einen Schreibfall bei dem der Master die Daten an den Slave sendet.Das Signal *rd_or_wr ist dabei auf 0 gesetzt.* Der Slave hat eine 7-Bit Adresse, daher ist das Signal *adresse_10_7* auf Null gesetzt ist. Nach dem das *Reset* zurückgenommen und das *Enable* Signal gesetzt wurde, wird der Transfer gestartet. Das Signal „SDA“ geht auf 0, während das Signal SCL auf 1 ist. Nach dem erfolgreichen Start des Transfers, wird das Signal *Busy* auf 1 gesetzt. Das zeigt an, dass der Master beschäftigt ist und aktuell keine neuen Daten sofort annehmen oder ändern kann. Beim 9. Taktzyklus erwartet der Master eine Adressbestätigung vom angesprochenen Slave. In diesem Fall wird eine Null auf SDA geschrieben. Dieses bedeutet, dass der Slave eine Adressbestätigung an den Master gesendet hat. Daraufhin folgt die Zusendung der Daten, die ebenfalls beim 9. Taktzyklus vom Slave bestätigt werden müssen. In diesem Fall schreibt der Slave ein High Pegel auf SDA. Dadurch wird der Transfer vom Master sofort gestoppt.

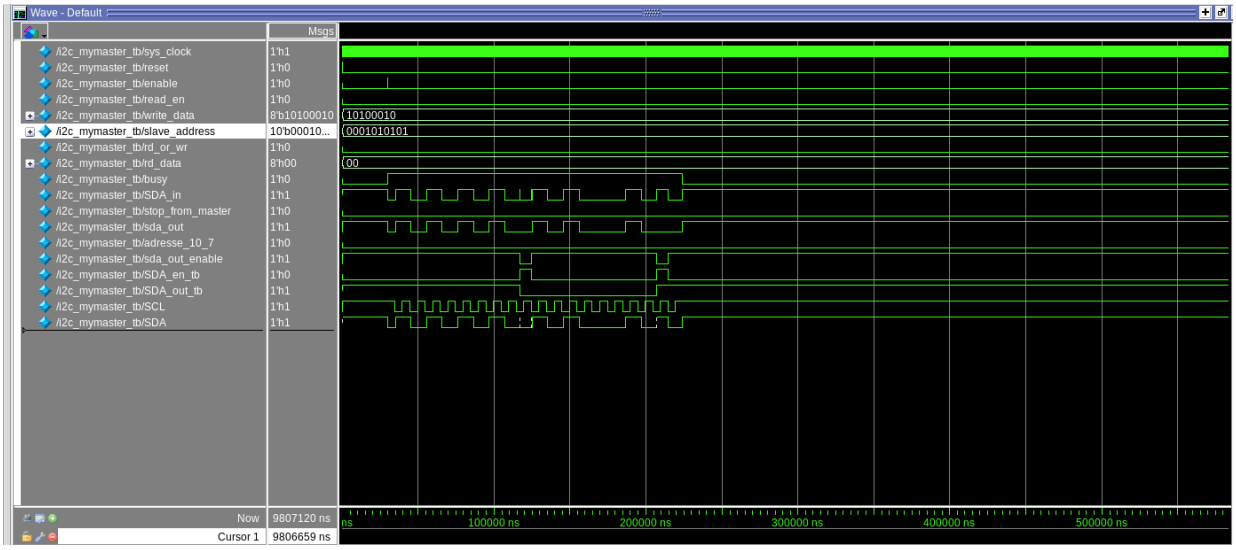

Abbildung 48: 7Bit-Adresse Transfer

2. Testdurchlauf: 7Bit Adresse Write, Ack-Adresse = 1

Bei diesem Test handelt es sich wieder um einen Schreibfall bei dem der Master die Daten an den Slave sendet. Das Signal *rd_or_wr* ist ebenfalls wieder auf 0 gesetzt. Der Slave hat ein 7-Bit Adresse, daher wird das Signal *adresse_10_7* auf Null gesetzt. Nach dem das *Reset* zurückgesetzt und das *Enable* gesetzt wurde, wird der Transfer gestartet. Das Signal *SDA* geht auf 0, während das Signal *SCL* auf 1 ist. Nach dem erfolgreichen Start des Transfers, wird das Signal *Busy* auf 1 gesetzt. Das zeigt an, dass der Master beschäftigt ist und keine neuen Daten

annehmen oder ändern kann. Beim 9. Taktzyklus erwartet der Master eine Adressbestätigung vom angesprochenen Slave. In diesem Fall wird ein High Pegel auf SDA geschrieben. Dieses bedeutet, dass der Slave die zugesendete Adresse nicht erkannt hat. Deswegen wird der Transfer sofort vom Master gestoppt.

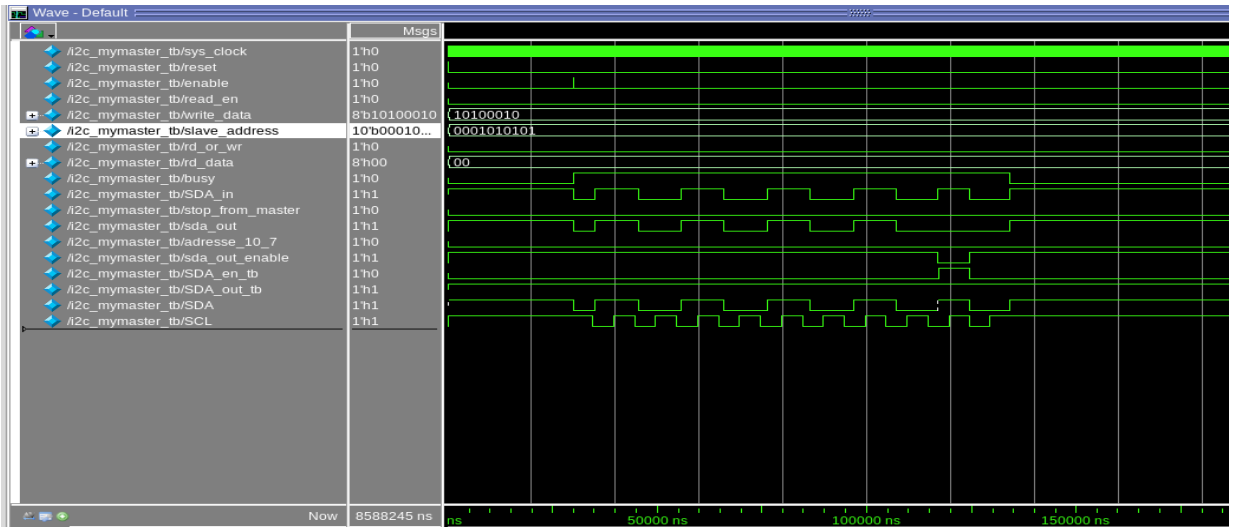

Abbildung 49: 7Bit-Adresse, Ack =1

3. Testdurchlauf: 7Bit Adresse Read

Bei diesem Test handelt es sich um einen Lesen-Fall bei dem der Master die Daten des Slaves empfängt. Das Signal *rd_or_wr* ist daher auf 1 gesetzt. Der Slave hat eine 7-Bit Adresse, daher ist das Signal „adresse_10_7“ auf Null gesetzt. Nach dem das „Reset“ zurückgesetzt und das „Enable“ Signal auf 1 gesetzt wurde, wird der Transfer gestartet. Das Signal „SDA“ geht auf 0, während das Signal SCL auf 1 ist. Nach dem erfolgreichen Start des Transfers, wird das Signal „Busy“ auf 1 gesetzt. Das zeigt an, dass der Master beschäftigt ist und nicht sofort neue Daten annehmen oder ändern kann. Beim 9. Taktzyklus erwartet der Master eine Adressbestätigung vom angesprochenen Slave. In diesem Fall wird eine Null auf SDA geschrieben. Dies bedeutet, dass der Master eine Adressbestätigung vom Slave erhalten hat. Daraufhin wird das Signal „Read_en“ auf 1 gesetzt. Der Slave beginnt dann mit der Versendung der Daten an den Master. Beim 9. Taktzyklus entscheidet der Master, ob er noch Daten vom Slave erhalten möchte, sonst fordert er den Stopp des Transfers.

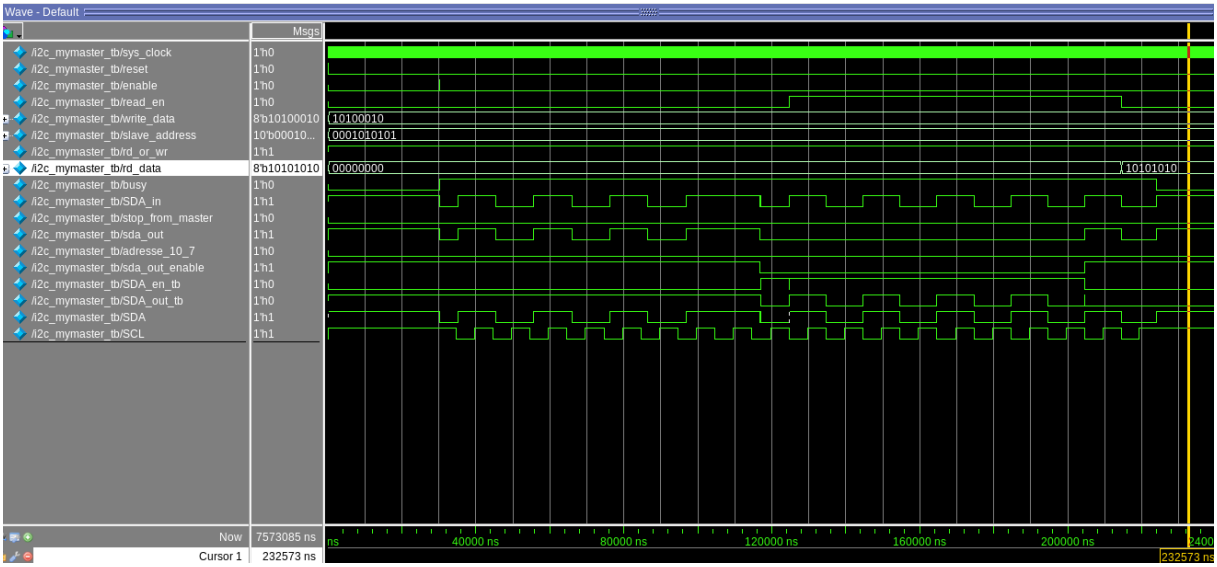

Abbildung 50: 7Bit Adresse, Read

4. Testdurchlauf: 10Bit-Adresse, Write

Bei diesem Test handelt es sich um einen Schreibfall bei dem der Master die Daten an den Slave sendet. Das Signal *rd_or_wr* ist auf 0 gesetzt. Der Slave hat eine 10-Bit Adresse, daher ist das Signal *adresse_10_7* auf 1 gesetzt. Nach dem das *Reset* zurückgesetzt und das Signal *Enable* auf 1 gesetzt wurde, wird der Transfer gestartet. Das Signal SDA geht auf 0, während das Signal SCL auf 1 ist. Nach dem erfolgreichen Start des Transfers, wird das Signal *Busy* auf 1 gesetzt. Das zeigt an, dass der Master beschäftigt ist und nicht sofort neue Daten annehmen oder ändern kann. Beim 9. Taktzyklus erwartet der Master eine Adressbestätigung vom angesprochenen Slave. In diesem Fall wird eine Null auf SDA geschrieben. Dieses bedeutet, dass der Slave eine Adresse-Bestätigung an den Master gesendet hat. Daraufhin folgt die Zusendung des zweiten Teils der Adresse, die ebenfalls beim 9. Taktzyklus vom Slave bestätigt werden soll. In diesem Fall schreibt der Slave ein Low Pegel auf SDA. Dies hat die Bedeutung, dass der Slave die Adresse erkannt hat und der Transfer somit weiterlaufen kann. Der Master schreibt nun die zusendenden Daten auf *SDA*.

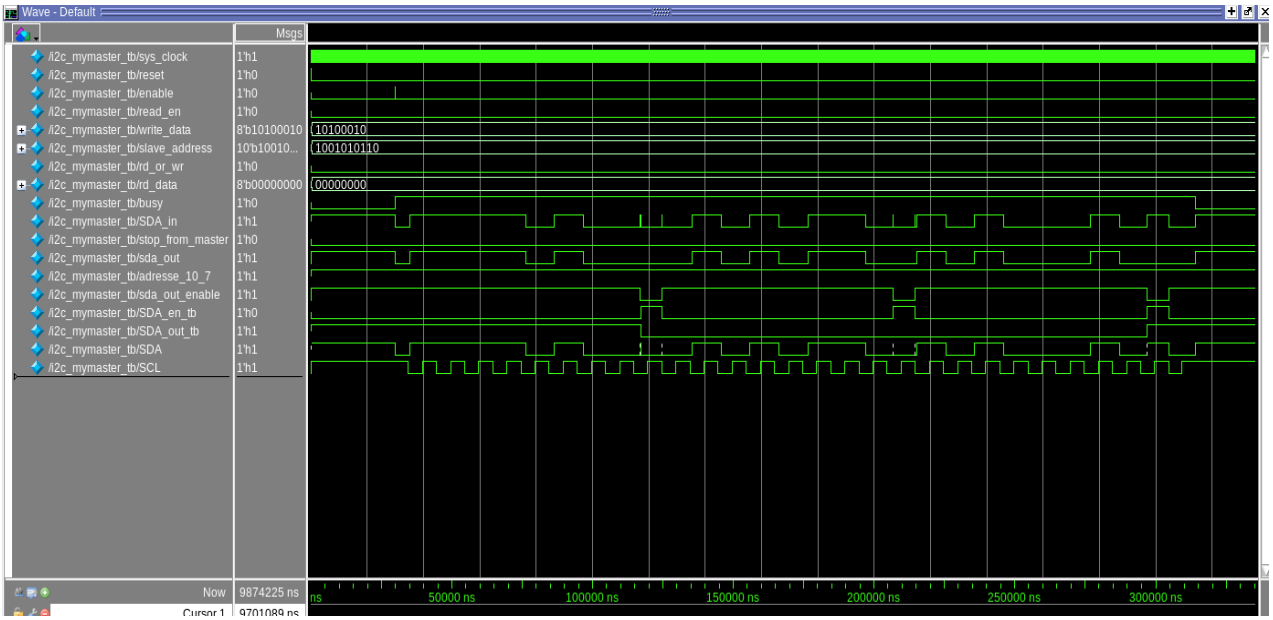

Abbildung 51: 10Bit-Adresse, Write

5. Testdurchlauf: 10Bit-Adresse, Read

Bei diesem Test handelt es sich um einen Lesen-Fall bei dem der Master die Daten vom Slave empfängt. Das Signal *rd_or_wr* ist auf 1 gesetzt. Der Slave hat ein 10-Bit Adresse, daher ist das Signal *adresse_10_7* auf 1 gesetzt. Nach dem das *Reset* zurückgesetzt und das Signal *Enable* auf 1 gesetzt wurde, wird der Transfer gestartet. Das Signal *SDA* geht auf 0, während das Signal SCL auf 1 ist. Nach dem erfolgreichen Start des Transfers, wird das Signal *Busy* auf 1 gesetzt. Das zeigt an, dass der Master beschäftigt ist und nicht sofort neue Daten annehmen oder ändern kann. Beim 9. Taktzyklus erwartet der Master eine Adressbestätigung vom angesprochenen Slave. In diesem Fall wird eine Null auf SDA geschrieben. Dieses bedeutet, dass der Slave eine Adresse-Bestätigung an dem Master gesendet hat. Daraufhin folgt die Zusendung des zweiten Teils der Adresse, die ebenfalls beim 9. Taktzyklus vom Slave bestätigt werden soll. In diesem Fall schreibt der Slave ein Low Pegel auf SDA. Dieses hat die Bedeutung, dass der Slave die Adresse erkannt hat und der Transfer somit weiterlaufen kann. Nun wird ein Restart vom Master gefordert um den dritten Teil der Adresse zu zusenden. Der dritte Teil der Adresse wird ebenfalls vom Slave bestätigt. Daraufhin wird das Signal *Read_en* auf 1 gesetzt. Der Slave kann danach anfangen die Daten an den Master zu versenden. Beim 9. Taktzyklus entscheidet der Master, ob er noch Daten vom Slave erhalten möchte. In diesem Fall fordert er einen Stopp des Transfers.

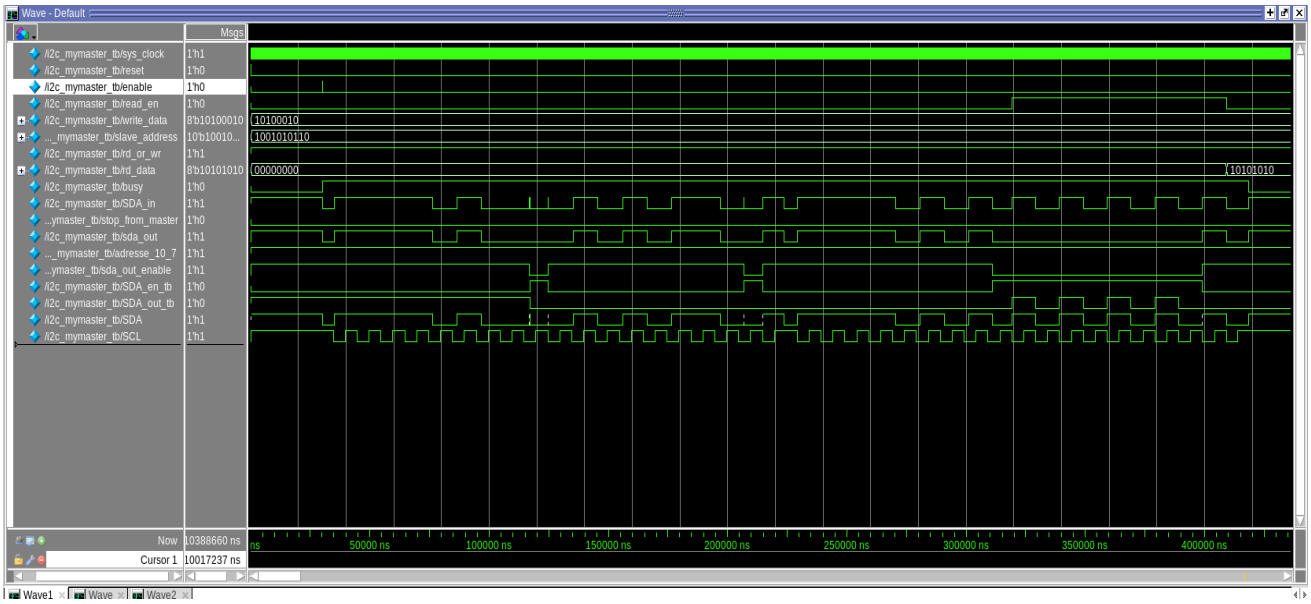

Abbildung 52: 10Bit-Adresse, Read

8. Testbench & Simulation des JTAGs

8.1 Testbench für ExternalTest

Um die Funktionen des JTAG-Extest zu überprüfen, wird ein Testbench mit zwei Chips erstellt. Jeder Chip enthält eine Instanziierung des JTAG-Standards und eine des Boundary-Scan-Registers. Diese Testbench dient dazu, sicherzustellen, dass der ExternalTest korrekt implementiert ist und die gewünschten Funktionen bereitstellt.

8.1.1 Grundlegender Aufbau der ExternalTest-Testbench

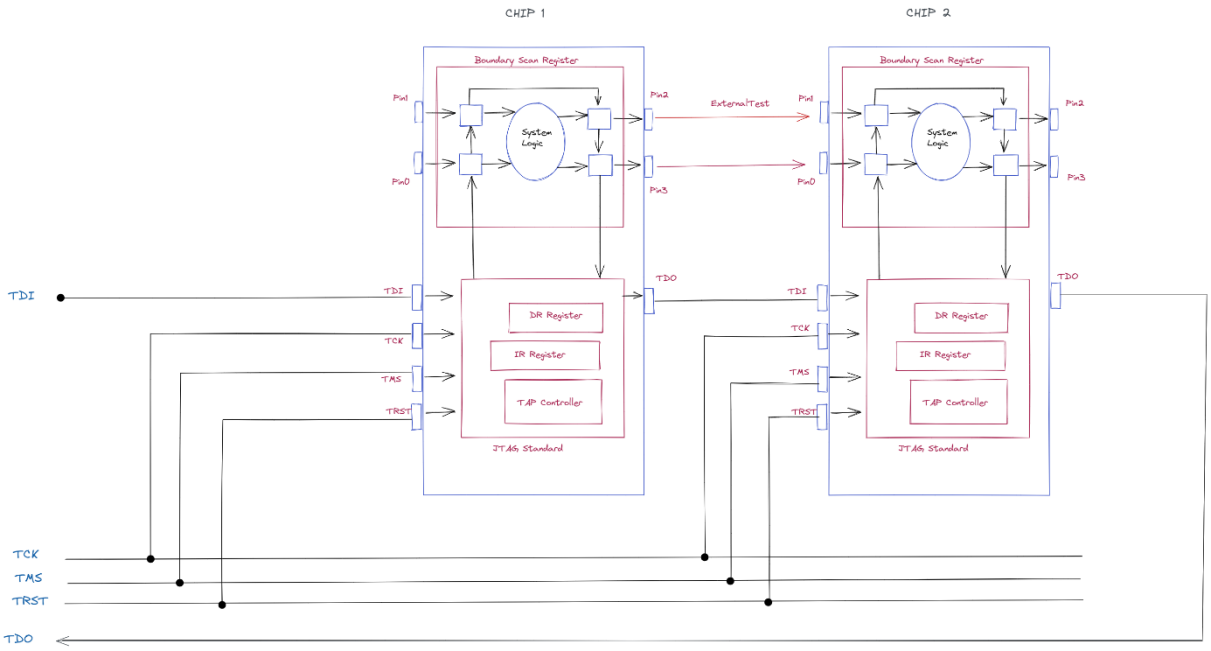

Abbildung 53: Konzept der Testbench JTAG-Test

In diesem Test wird die Verbindung zwischen zwei Bausteinen bzw. Chips, wie in Abbildung 53 gezeigt, betrachtet. Dabei ist der Ausgangspin-2 von Chip-1 mit dem Eingangspin-1 von Chip-2, und der Ausgangspin-3 von Chip-1 mit dem Eingangspin-0 von Chip-2 verbunden.

Um die Funktionsfähigkeit dieser Verbindung zu überprüfen, soll ein Wert am Ausgangspins von Chip-1 festgelegt und dann überprüft werden, ob dieser Wert auch am Eingangspins von Chip-2 gelesen werden kann. Diese Art des Tests kann für alle Verbindungen auf einer bestückten Leiterplatte (PCB) durchgeführt werden und wird als ExternalTest bezeichnet.

Während der Durchführung der Simulation werden die Ausgangsignale und Inhalt der Instruktionsregister aller Bausteine betrachtet.

8.1.2 Implementierung der Testbench

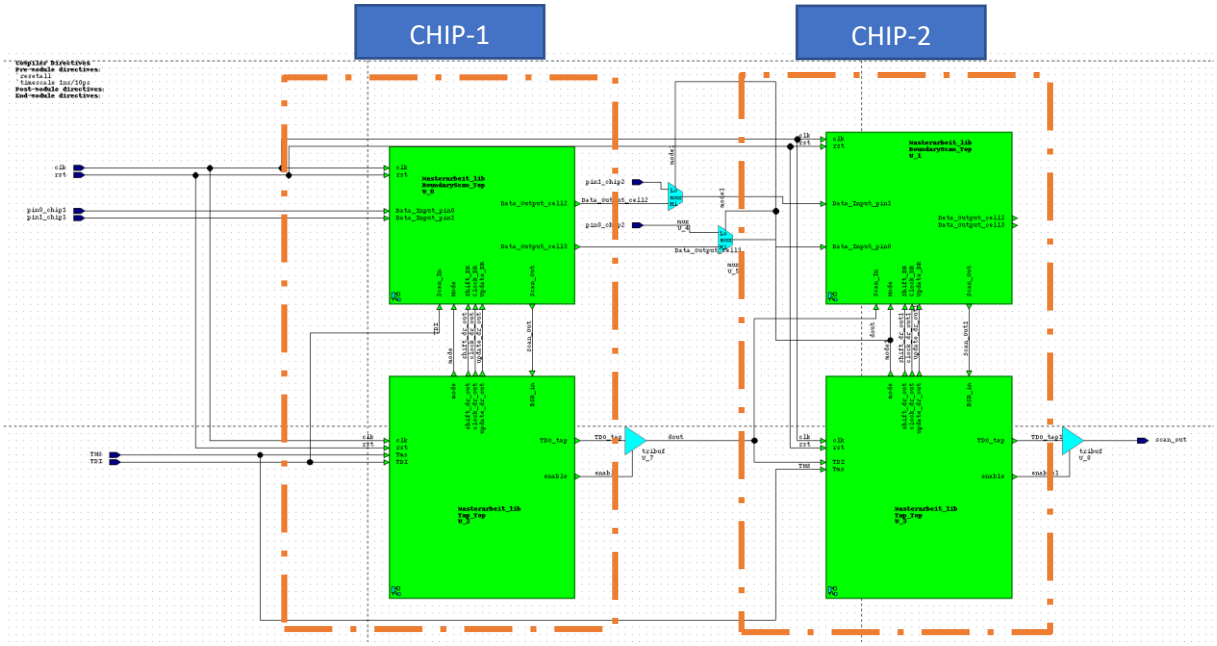

Abbildung 54: Aufbau der Testbench in HDL-Designer

Die Verilog-Testbench, wie in Abbildung 54 dargestellt, besteht aus zwei Instanzen: eine für CHIP-1 und eine für CHIP-2. Jeder Chip umfasst eine Instanz des JTAG-Standards, die einen TAP-Controller, ein Instruktionsregister und ein Datensregister enthält, sowie ein Boundary-Scan-Register, das aus vier Boundary-Scan-Zellen besteht.

Die Signale TCK, TMS, TDI und TDO bilden zusammen die JTAG-Schnittstelle oder TAP (Test Access Port), die als Testschnittstelle des Chips dient. Der TAP-Controller steuert den Zugriff auf das IR (Instruction Register) oder das DR (Data Register). Wenn ein Befehl in das IR geladen wird, wählt der TAP-Controller das DR aus, das vom Befehl im IR angesprochen wird. Anschließend werden entsprechende Daten in und aus dem ausgewählten DR geschoben. Der Ablauf lässt sich wie folgt zusammenfassen:

1. Laden eines Befehles/Instruktion in das IR.
2. Auswählen des DR und Übertragen von Daten in und aus dem DR anhand des Befehles im IR.

8.1.3 Simulation & Ergebnisse

In diesem Abschnitt wird die Durchführung der Tests erläutert, indem Eingangssignale festgelegt werden und die Ausgangssignale überprüft und interpretiert werden

1. **Testdurchlauf 1 (Überprüfung des Instruktionsregisters):**

Nach dem *Reset* werden die folgenden Daten über TDI in das IR-Register geschoben.

- Input-Daten: „0000 0000“. (0000 entsprechen ExternalTest)
- Es wird erwartet, dass die IR-Register von Chip-1 und Chip-2 mit dem Wert „0000“ gefüllt sind.

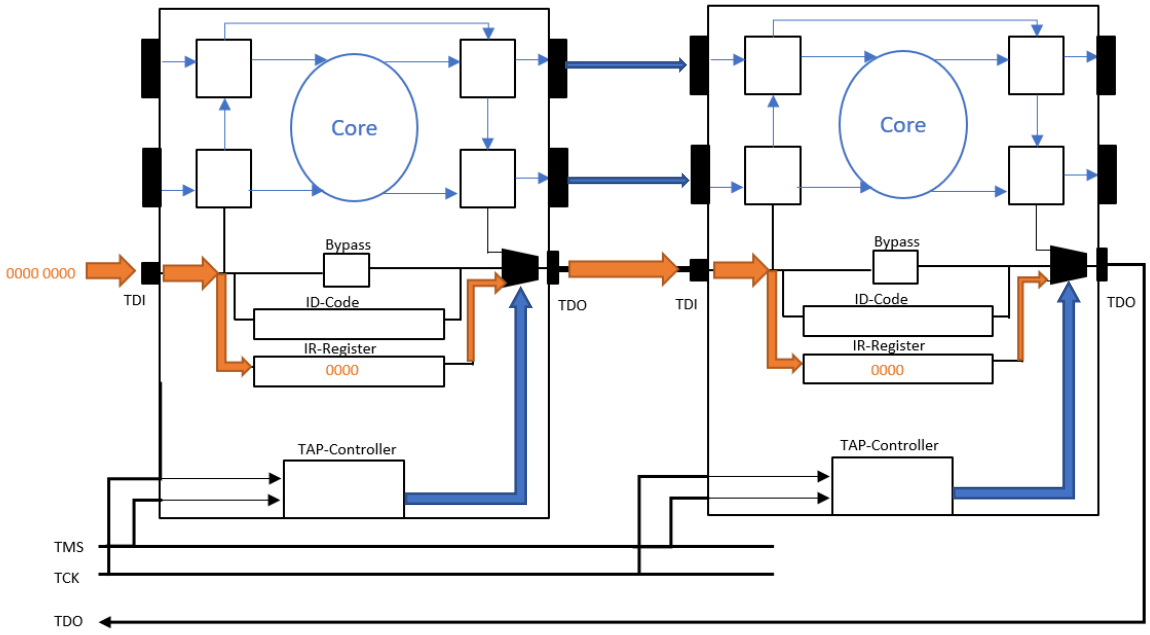

Abbildung 55: JTAG ExternalTest Logik

Chip-1

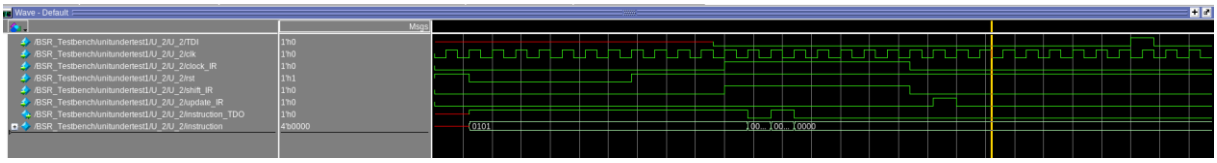

Abbildung 56: Simulationsergebnis des Instruktionsregisters von Chip-1

Chip-2

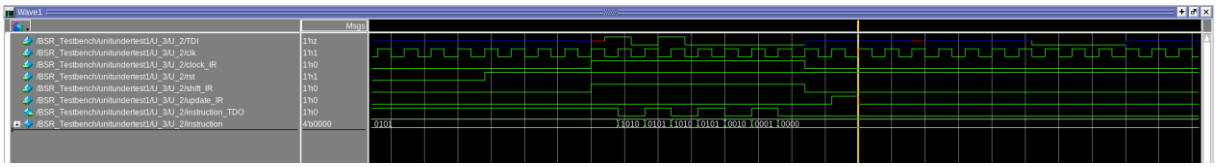

Abbildung 57: Simulationsergebnis des Instruktionsregisters von Chip-2

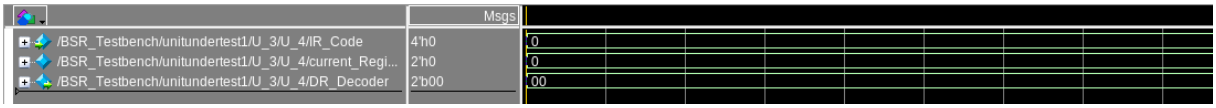

Abbildung 58: Simulationsergebnis des Instruktion-Decoder

Das Ergebnis der Prüfung der Instruktionsregister lautet wie folgt:

- Instruktionsregisters werden mit „0000“ korrekt befüllt.
- Der Instruktion-Decoder decodiert den eingegebenen Wert und gibt den Wert „00“ weiter.

2. Testdurchlauf 2 (Überprüfung der Boundary-Scan-Zellen):

Nach dem Laden des Befehls in das Instruktionsregister, soll der TAP-Controller das „Boundary-Scan-Register“ als Datenregister auswählen.
Mit dem seriellen Dateneingangspin, der TDI (Test Data In) genannt wird, und einem seriellen Datenausgangspin, dem TDO (Test Data Out), können wir nun alle Pins eines Chips über das Boundary-ScanRegister erreicht werden.

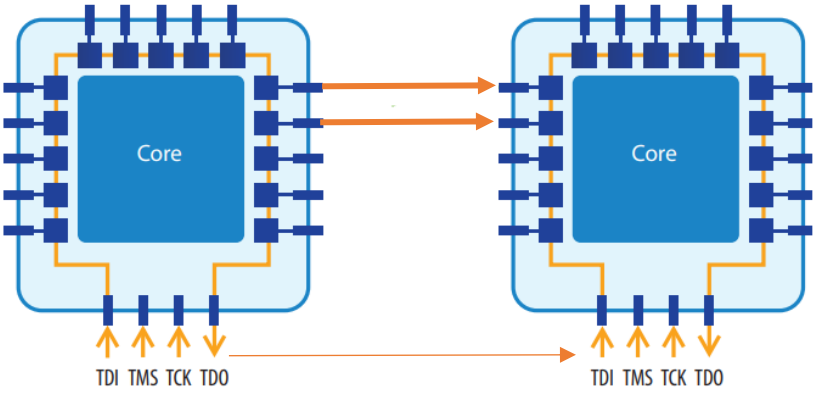

Abbildung 59: Schieberegister entlang der Boundary (Pins) eines Chips [2]

Es wird der Datenstrom „0000 0100“ über TDI in den TAP Controller geschoben.
Wenn die Daten über TDI in den TAP-Controller gerschoben werden, wird erwartet, dass die Werte in den Zellen, wie in Abbildung 59 dargestellt, übertragen werden.

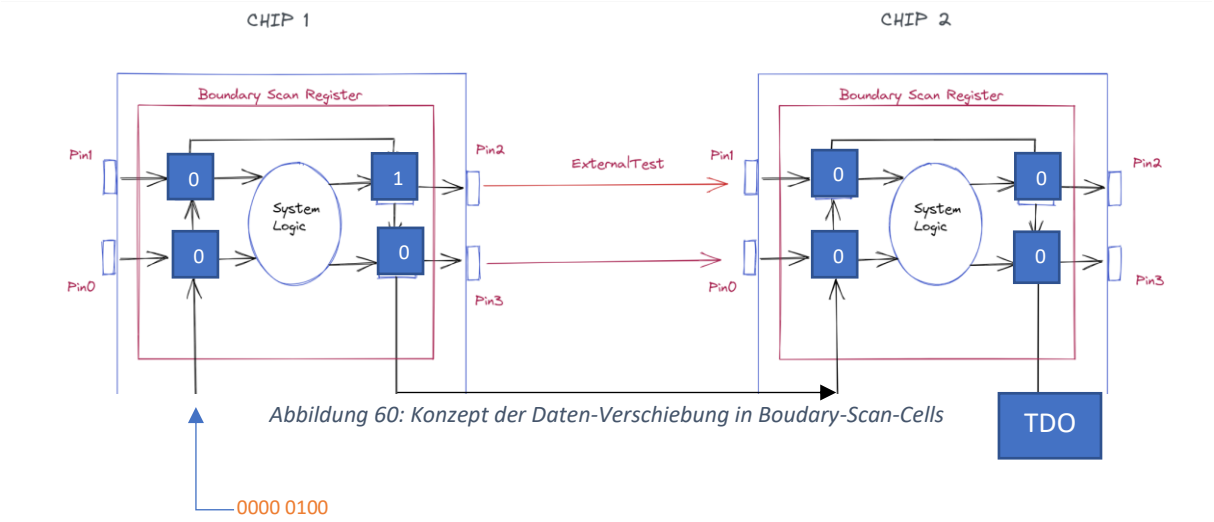

Abbildung 60: Konzept der Daten-Verschiebung in Boudary-Scan-Cells

Die beiden Abbildungen 61/62 zeigen, dass die Boundary-Scan-Zellen die richtigen Werte erhalten haben. Dies deutet darauf hin, dass die serielle Datenübertragung über TDI an BSR erfolgreich durchgeführt wurde.

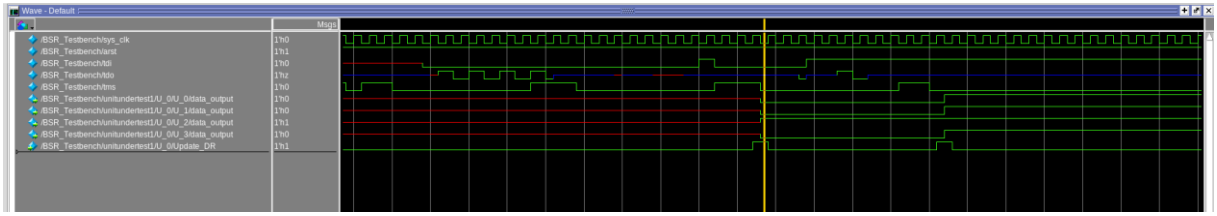

Abbildung 61: Simulationsergebnis des BSC von Chip-1

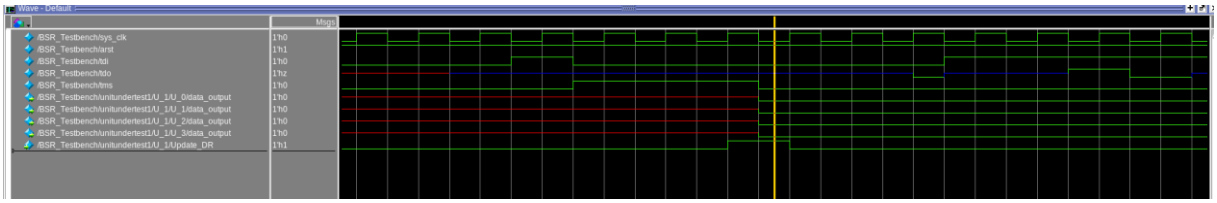

Abbildung 62: Simulationsergebnis des BSC von Chip-2

Nach dem Laden der Daten in die Boundary-Scan-Zellen, selektiert der Tap-Controller wieder das Boundary-Scan-Register als Datenregister.
Folgende Fälle werden betrachtet, um die Verbindung zwischen den Pins zu überprüfen:

- Wenn Capture_DR gesetzt ist, soll jede Zelle die neuen Werte des Pins aufnehmen, mit dem sie verbunden ist.
- Da Pin2 von Chip-1 mit dem Pin1 von Chip-2 verbunden ist, und Pin3 von Chip-1 mit Pin0 von Chip-2, wird erwartet, dass die Pins 0 und 1 die Werte von Pins 2 und 3 übernehmen werden.

- Wenn der *Shift_DR* von Tap-Controller gesetzt ist, werden neue Daten über den TDI geschoben, und somit werden die aktuellen Daten, die in den Pins gespeichert sind, am Ausgang sichtbar.

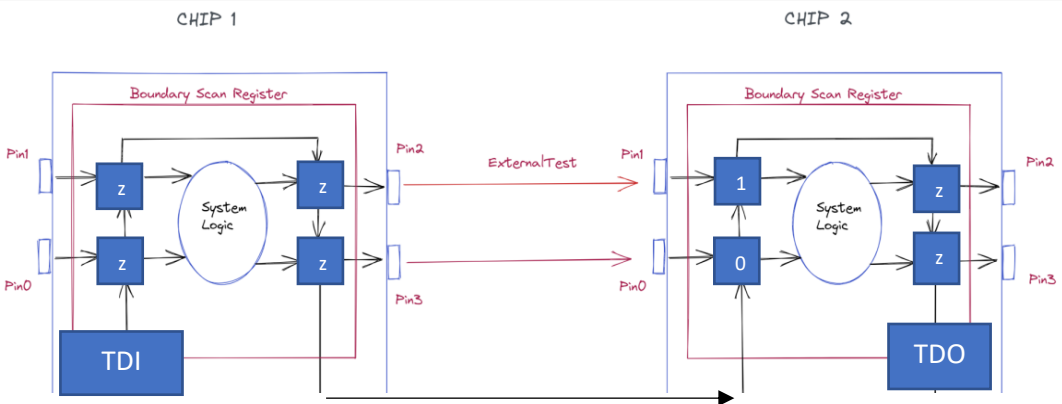

Abbildung 63: Konzept der Datenaustausch zwischen der verbundenen Pins nach Capture_DR

Ergebnisse des Testlaufs:

- Abbildung 65 zeigt, dass die Pins 0 und 1 die Werte von Pins 2 und 3 übernommen haben, wenn der Tap_Controller sich im Zustand Capture_Dr befindet (Clock_DR 1 & Shift_Dr 0). (siehe Abbildung 64).
- Im Ausgangsregister ist der erwartete Wert "zzzz01zz" sichtbar. Abbildung 65.

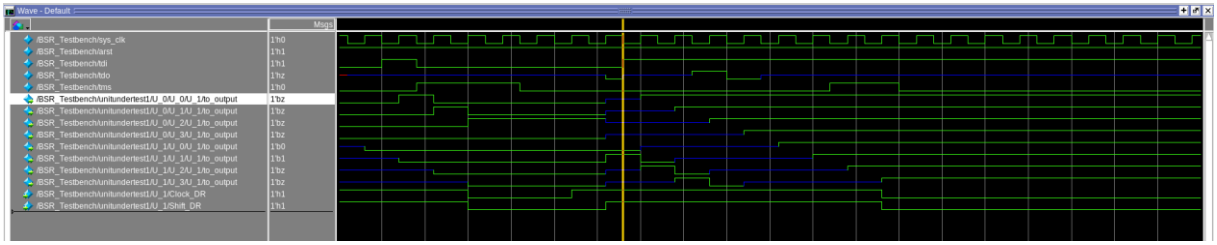

Abbildung 64: Signalverläufe der Ausgang-Pins

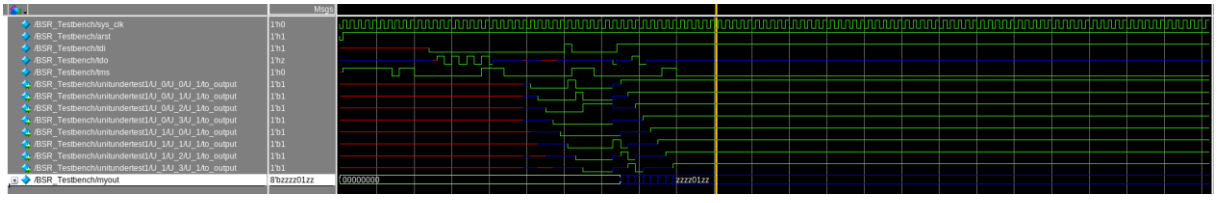

Abbildung 65: Ergebnis des ExternalTest (myout-Register)

8.2 Testbench für Sample/Preload

Um die Funktionen des JTAG Sample/Preload zu überprüfen, wird ein Testbench mit zwei Chips erstellt. Jeder Chip enthält eine Instanz des JTAG-Standards und eine des Boundary-Scan-Registers. Dieser Testbench dient dazu, sicherzustellen, dass die Sample/Preload Logik korrekt implementiert ist und die gewünschten Funktionen bereitstellt.

8.2.1 Grundlegender Aufbau und Ziel des Tests

In diesem Test wird das Verhalten des Systems untersucht, wenn das Sample/Preload-Datenregister vom Befehlsregister ausgewählt wird. Der Test besteht aus zwei Chips: Chip-1 und Chip-2. Die beiden Chips sind in Reihe geschaltet, indem das TDO des Chip-1 mit TDI des Chip-2 verbunden wird. TCK und TMS werden parallel an alle Chips angeschlossen.

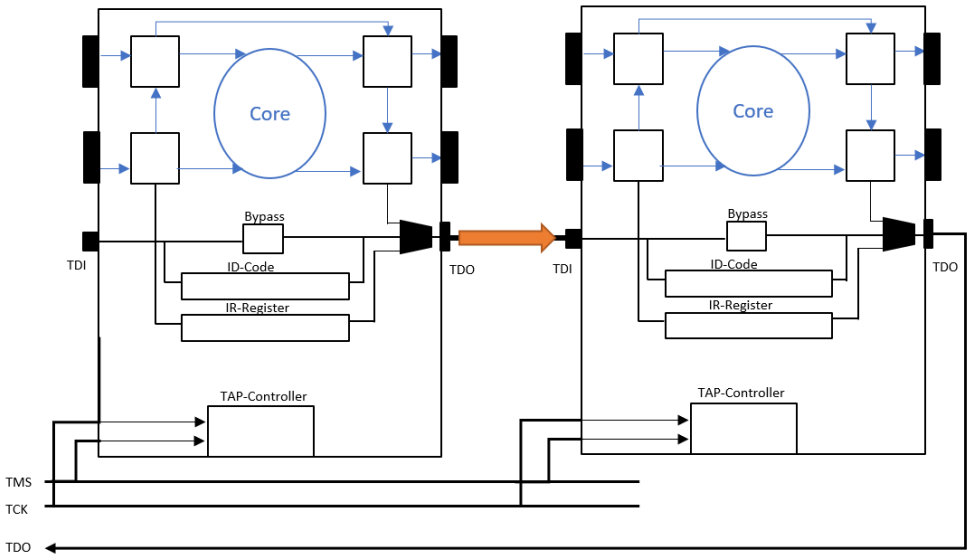

Abbildung 66: Grundlegender Aufbau des Sample/Preload Testes

8.2.2 Simulation & Ergebnisse

1. Testdurchlauf: Test der Samples-intruktion

Um das Verhalten der Sample-Funktion zu untersuchen, werden die folgenden Schritte durchgeführt:

1. Befehl in das Befehlsregister laden: „0001“

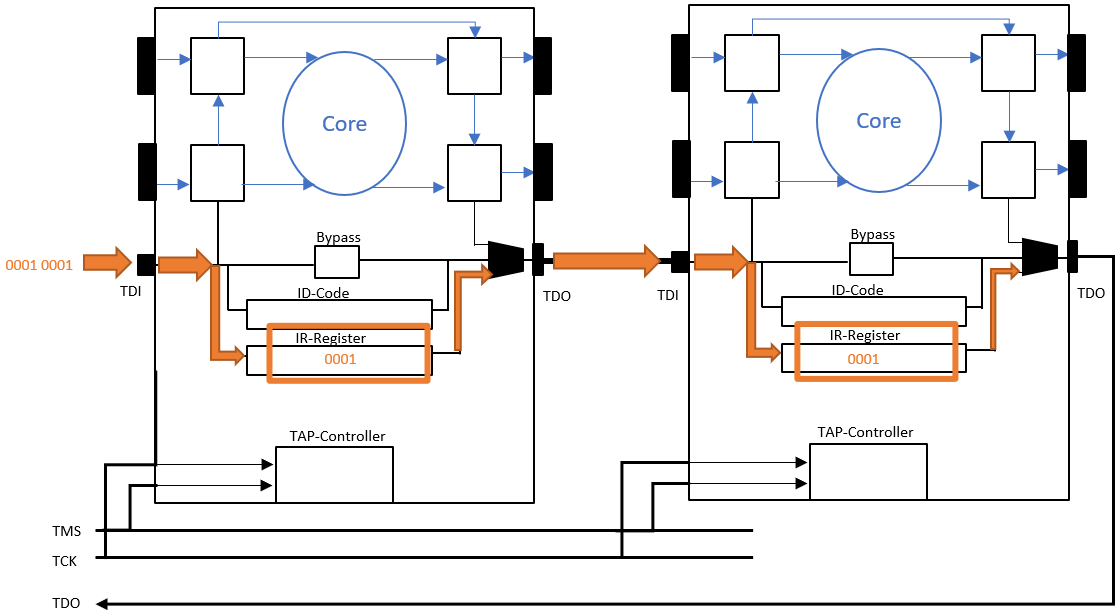

Abbildung 67: Verlauf des Sample-Tests

2. Rückkehr in den Zustand Run-test/Idle
3. Daten an die Pins anlegen.(siehe Abbildung 68).

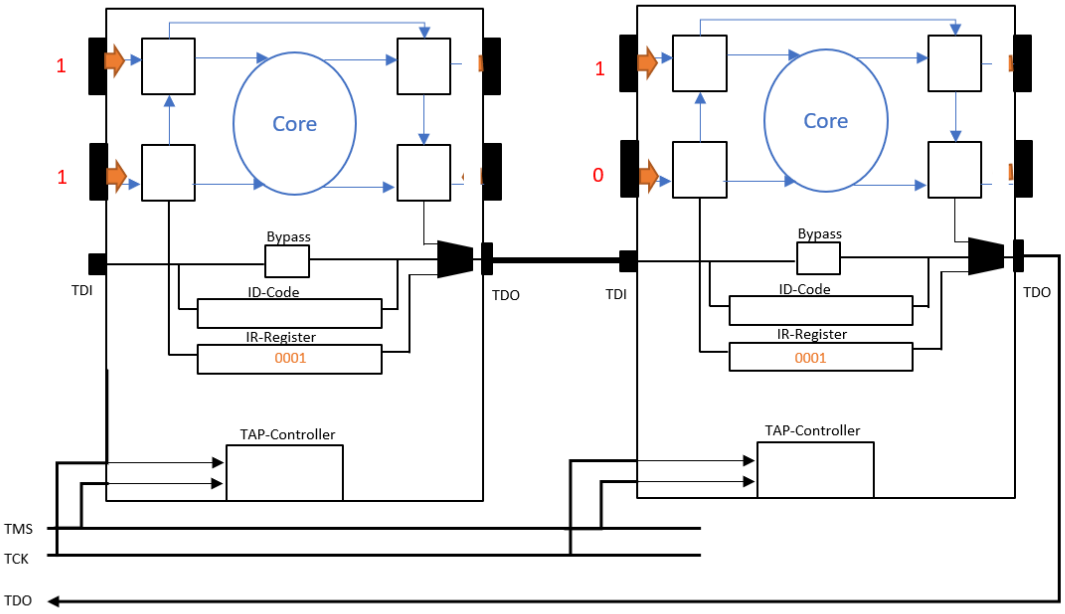

Abbildung 68: Konzept und Verlauf des Sample-Testes

4. Beliebige Daten über TDI schieben, um die anliegenden Daten am Ausgang sichtbar zu machen. (siehe Abbildung 69).

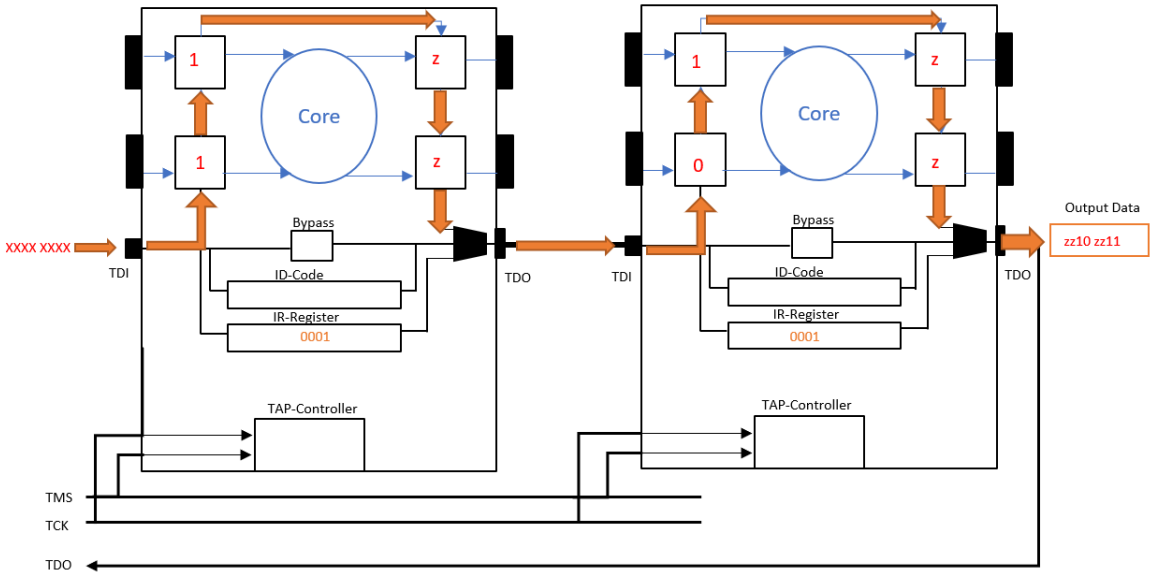

Abbildung 69: Konzept und Verlauf des Testes

Die Simulation liefert die in den: Abbildungen 70/71/72 dargestellten Ergebnisse:

- Die Instruktionsregister von Chip 1 und Chip 2 sind beide mit dem Wert „0001" gefüllt, was den Code der Sample/Preload Instruktion repräsentiert(siehe Abbildung 70).
- Bei *Capture_DR* (Clock_DR=1 & Shift_DR=0) übernehmen die Boudary-Scan-Zellen die Werte von den Chip-Pins(siehe Abbildung 71).
- Wenn neue Daten über den seriellen Eingang TDI übertragen werden, werden die alten Daten aus den Flip-Flops über den seriellen Ausgang TDO ausgegeben und im Register der Testbench als Codewort „11zz01zz“ sichtbar gemacht (siehe Abbildung 72).

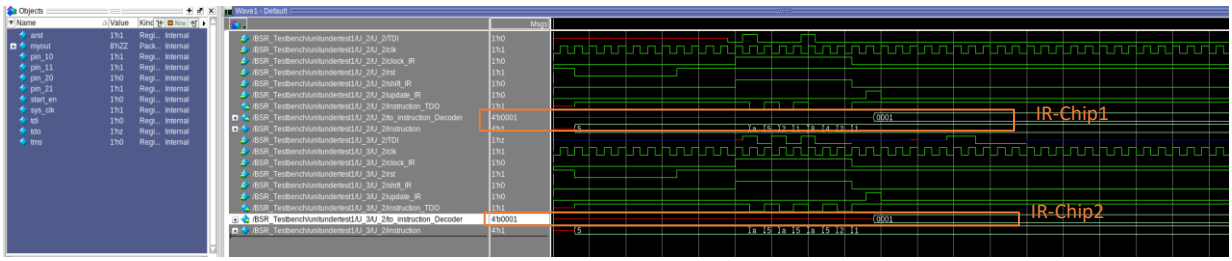

Abbildung 70: Signalverläufe des IR-Register

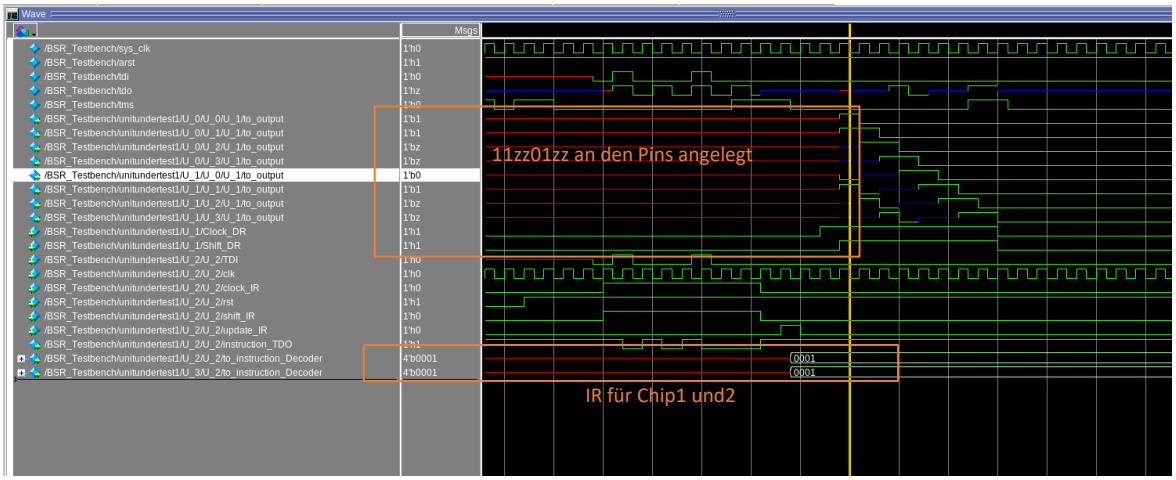

Abbildung 71: Signalverläufe der BS-Zellen

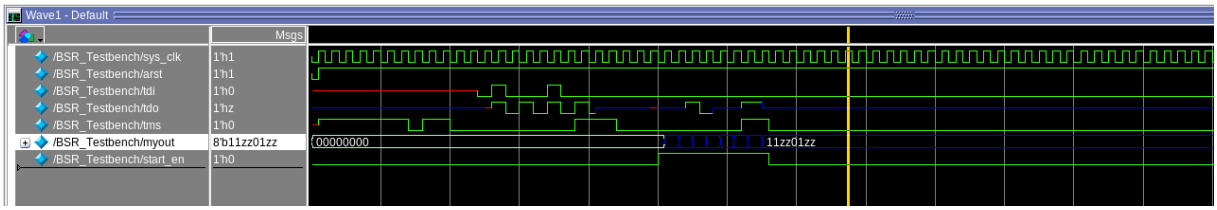

Abbildung 72: Signalverläufe und Ergebnisse des Sample-Testes

2. Testdurchlauf: Test der Preload-Instruktion

Im vorherigen Test wurden die Daten über Chip-Pins eingegeben und dann am *TDO* Signal sichtbar gemacht, was der Sample-Funktionalität entspricht.
In diesem Test werden die Daten erneut an die Boundary-Scan-Zellen übertragen, jedoch nicht über die Chip-Pins, sondern über TDI. Mit anderen Worten werden die Boundary-Scan-Zellen in diesem Test mit spezifischen Werten vorbereitet oder "vorgeladen", was der Preload Funktion enstrpicht.
Um das Verhalten des Preloads zu untersuchen, werden die folgenden Schritte durchgeführt:

1. Reset des Systems durch das Setzen des TMS-Signals auf Null für fünf Takte von TCK.

2. Befehl „0001“ in das Befehlsregister laden (siehe Testdurchlauf 1)

3. Beispiel Daten „1010 0100“ über TDI in den TAP-Controller hineinschieben. Die Zustandsmaschine soll für 8-Takte im Zustand *Shift_DR* 8 der Anzahl der Boundary-Scan-Zellen entspricht. (siehe Abbildung 73).

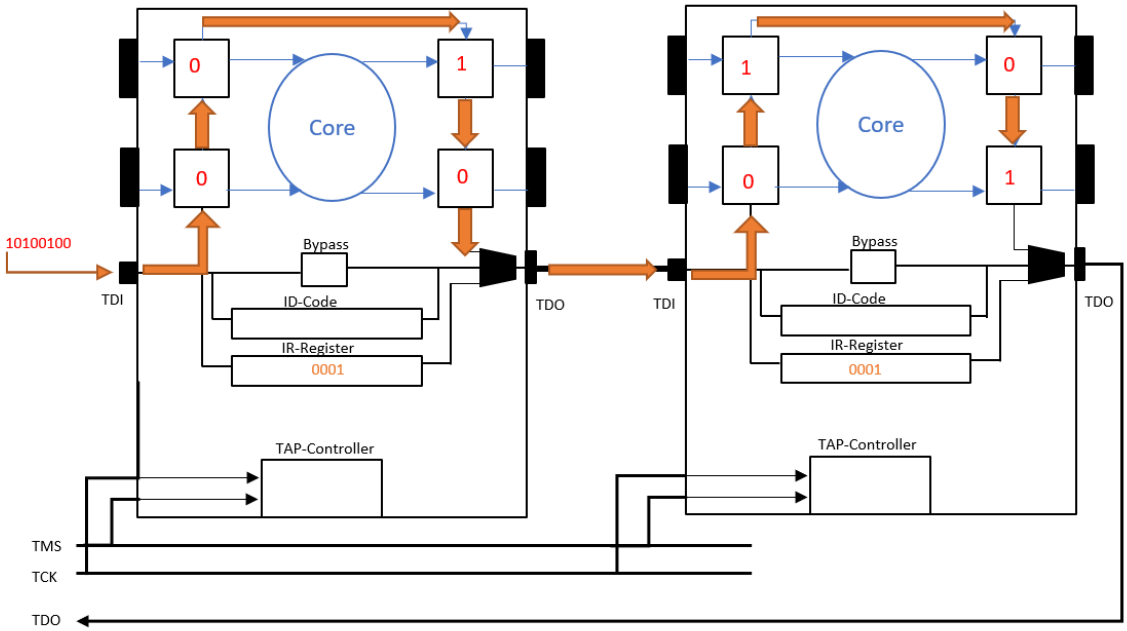

Abbildung 73: Konzept und Verlauf des Preloads-Test

Die Simulation liefert die in Abbildung dargestellten Ergebnisse:

- Die Daten wurden über den seriellen Eingang TDI übertragen und die Zellen haben diese Daten erfolgreich empfangen. Nachdem das Update_DR gesetzt wurde, wurden die Werte an die Pins übertragen, wie in der Abbildung 74 dargestellt, an die Pins der Chip angelegt

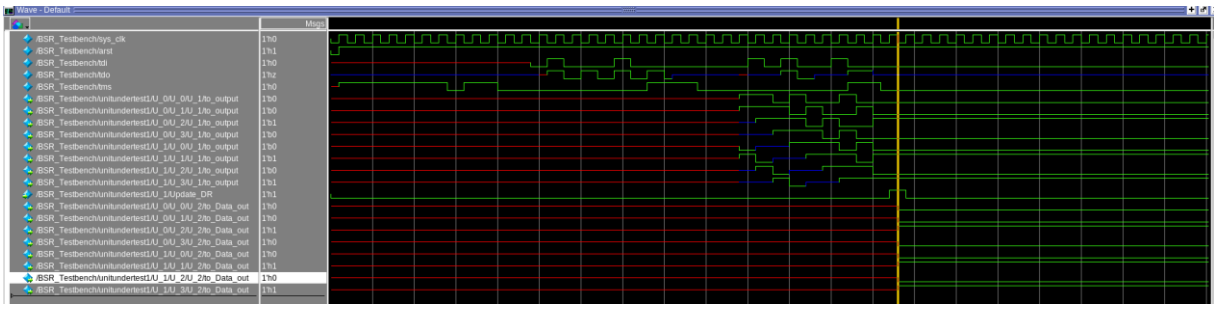

Abbildung 74: Signalverläufe der BoudaryScan-Zellen nach Update_DR im Preload Test

8.3 Test des JTAG-Standards

In diesem Kapitel werden die Hauptfunktionalitäten des JTAG-Standards, wie das Bypass-Register und das ID-Code-Register (Identification Code), getestet und die Ergebnisse der Simulationen bewertet.

8.3.1 ID-Code-Register (Identification Code)

Das ID-Code-Register ist ein spezielles Register im JTAG-Standard, das von der zu testenden Hardware zur Verfügung gestellt wird. Es enthält Informationen über die Hardware, wie den Hersteller, das Produkt und die Version. Das ID-Code-Register kann von einem JTAG-Tester ausgelesen werden, um die Hardware zu identifizieren und sicherzustellen, dass der Tester mit der richtigen Hardware kommuniziert.

Um die Komplexität des Tests zu verringern, wurde das ID-Code-Register auf eine 8-Bit-Version reduziert, anstatt wie ursprünglich vorgesehen ein 32-Bit-Register zu verwenden.

1. **Testdurchlauf:**

In dieser Testbench gibt es zwei Instanzen des JTAG-Standards, die in Reihe geschaltet sind. Dies wird erreicht, indem das *TDO* Signal der ersten Instanz mit dem *TDI* Signal der zweiten Instanz verbunden wird. Die Signale *TCK* und *TMS* werden parallel an alle Chips angeschlossen.

Um das Verhalten des ID-Code-Register zu untersuchen, sollten die folgenden Schritte durchgeführt werden:

1. Reset des Systems durch das Setzten des TMS Signal auf Null für fünf Takte.
2. Den Befehl „0010“ in das Befehlsregister jede Instanz laden, der dem ID-Code entspricht.
3. Der ID-Code (Identification Code) wird vom System-Designer festgelegt und besitzt in diesem Fall den Wert „1111 0000". Nachdem beliebige Daten über *TDI* übertragen wurden, sind die ID-Codes beider Instanzen am Ausgang sichtbar.

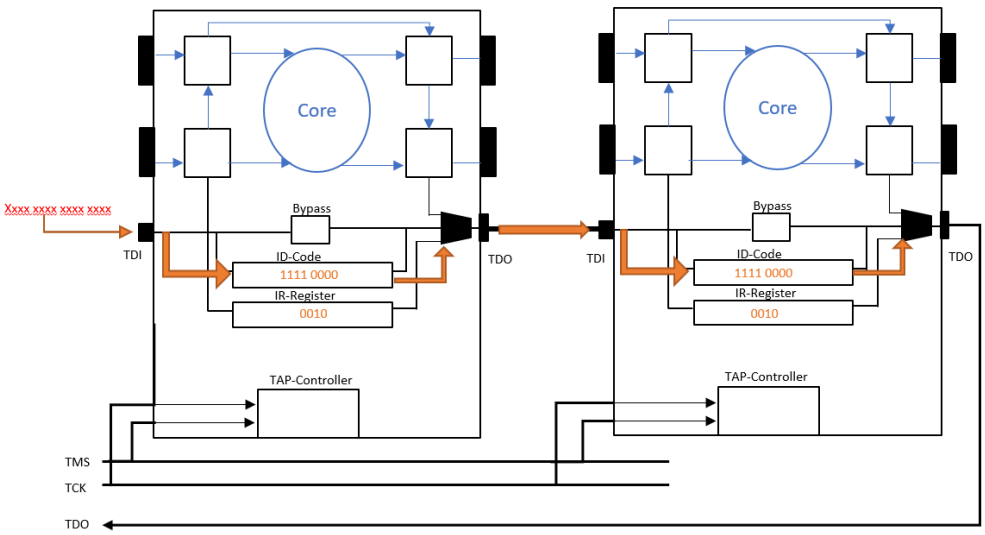

Abbildung 75: Konzept und Verlauf des Testes

Die Simulation liefert in Abbildung 76 dargestellte Ergebnisse:

- Die Befehlsregister beider Instanzen werden mit dem Wert „0010“ gefüllt.
- Die ID-Codes beider Instanzen sind am Ausgang TDO sichtbar.

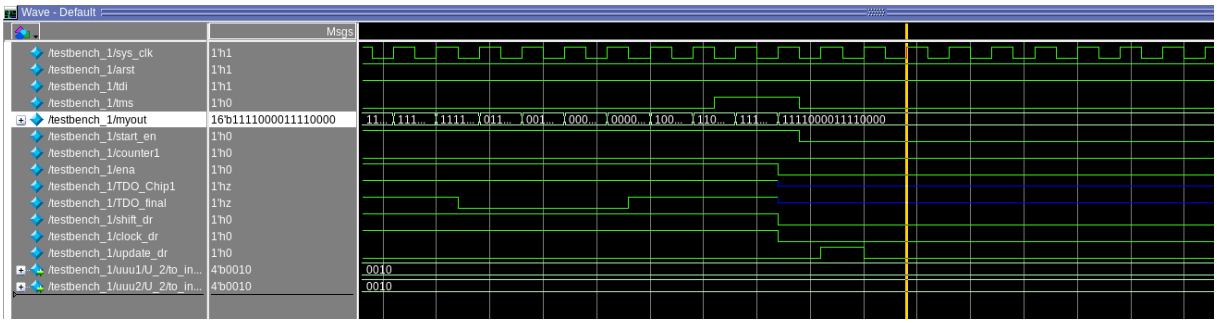

Abbildung 76: Signalverläufe und Ergebnisse des ID-Code Test

8.3.2 Bypass-Register-Test

Im vorherigen Test wurde der Zugriff auf das ID-Code-Register simuliert und die Ergebnisse interpretiert. Nun wird das ID-Code-Register erneut in eine Simulation eingebracht, wobei bei der ersten JTAG-Instanz das BYPASS-Register ausgewählt und bei der zweiten Instanz das ID-Code-Register ausgewählt wird. Auf diese Weise kann die Funktionalität der Bypass und der ID-Code Instruktion in einem Test simuliert werden.

1. **Testdurchlauf:**

   Diese Testbench besteht aus zwei Instanzen des JTAG-Standards, die in Reihe geschaltet sind. Dies wird erreicht, indem das *TDO* Signal der ersten Instanz mit dem *TDI Signal* der zweiten Instanz verbunden wird. Die Signale *TCK* und *TMS* werden parallel an alle Chips angeschlossen.

Um das Verhalten des Bypasses und ID-Code-Register zu untersuchen, werden die folgenden Schritte durchgeführt:

1. Ein Reset des Systems durch das Setzten des TMS Signal auf Null für fünf Takte.
2. Der entsprechende Befehl wird in das Befehlsregister jeder Instanz geladen: die erste Instanz erhält den Bypass-Befehl „1111“ und die zweite Instanz den ID-Befehl „0010“.

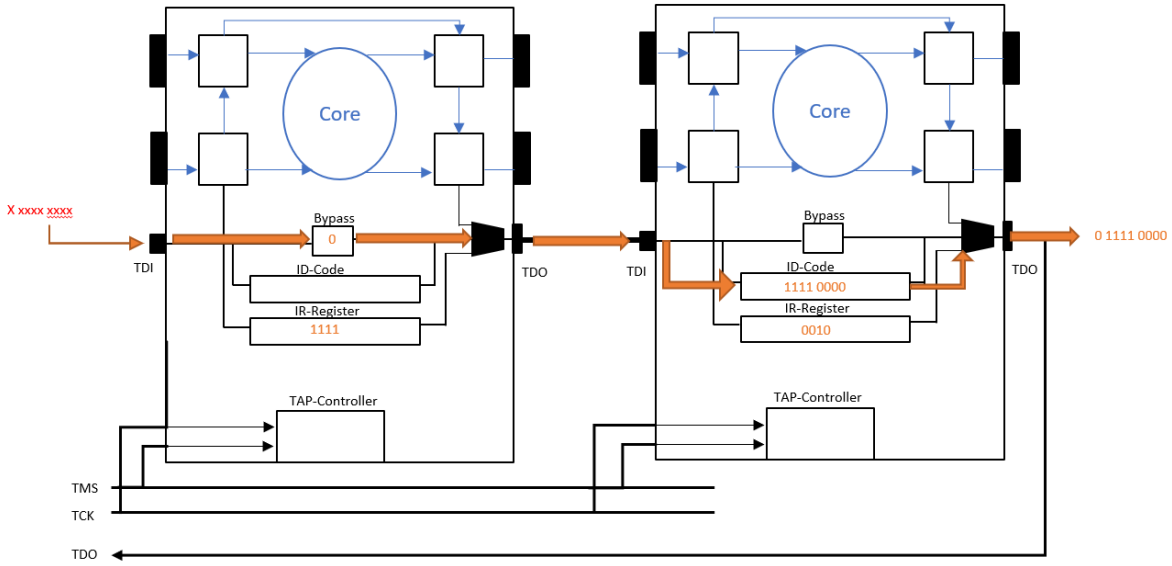

Abbildung 77: Konzept und Verlauf des Bypass-Testes

3. Für die zweite Instanz wurde vom System-Designer der ID-Code „1111 0000“ festgelegt. Nachdem beliebige Daten über *TDI* Signal übertragen werden, werden die „0 1111 0000“ am Ausgang erwartet.

Die Simulation liefert die in Abbildung 78 dargestellten Ergebnisse:

- Das Befehlsregister der ersten Instanz wird mit dem Wert „1111“ gefüllt.
- Das Befehlsregister der zweiten Instanz wird mit dem Wert „0010“ gefüllt.
- Die ID-Codes sind am *TDO* Ausgangssignal sichtbar.
- Das Bypass Bit ist am Ausgang sichtbar und ist zusammen mit dem ID-Code kombiniert. „0“ und „1111 0000“.

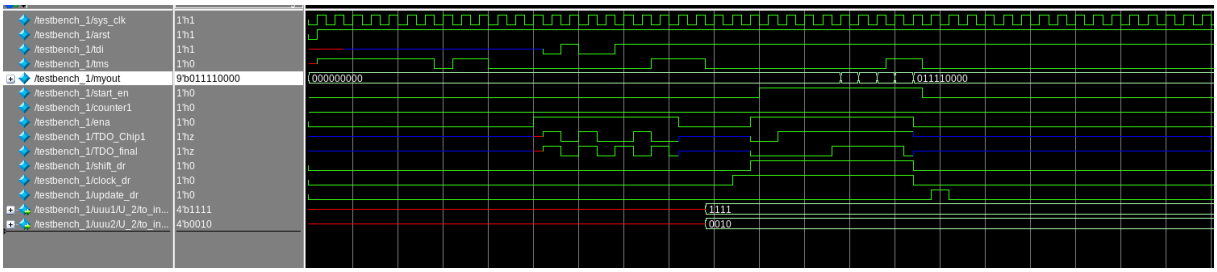

Abbildung 78: Signalverläufe und Ergebnisse des ID-Code / Bypass-Testdurchlaufs

9. Zusammenfassung

Die Aufgabenstellung bestand darin, serielle Schnittstellen zur Konfiguration und Test von integrierten Schaltkreisen zu entwerfen. Dabei wurde das Projekt in zwei Schwerpunkte unterteilt: Den Entwurf eines I2C-Master-Interface in Verilog und die Implementierung einer JTAG-Protokolleinheit. Der Hauptfokus lag auf der Erstellung einer Testumgebung mit dem implementierten JTAG-Standard Block und den Boundary-Scan-Registern/Zellen sowie auf der Durchführung einer Simulation der Funktionsweise des JTAG-Standard Blcoks mit QuestaSim. Die Aufgabenstellung wurde erfolgreich umgesetzt und es ist nun möglich, eine integrierte Schaltung über die entwickelte JTAG-Schnittstelle zu testen.

Im ersten Teil des Projekts wurde zunächst die I2C-Master-Instanz in Verilog entwickelt und ersten Testdurchläufen während der Entwicklung unterzogen. Auftretende Fehler konnten so unmittelbar be-hoben werden. Zu Beginn des Entwicklungsprozesses wurde der Master in zwei Teile gegliedert. So konnten Anforderungen des Protokolls und anwendungsspezifische Belange einfach abgegrenzt werden.

Nach der Fertigstellung eines ersten lauffähigen I2C-Masters wurde mit der Entwicklung einer umfassenden Testbench für den Gesamtentwurf begonnen. Dabei wurden ein Initial-Block, die Timescale-Direktive und Verzögerungsan-weisungen verwendet, damit benötigte Signale zu unterschiedlichen Zeitpunkten gesetzt werden können.

Ein Tri-State Gatter wurde ebenfalls in die Testbench integriert, um den Zugriff auf den SDA Bus zwischen I2c_Master und der Testbench zu arbitrieren.
Abschließend wurde eine Simulation für verschiedene Fälle durchgeüfhrt:

- Transfer mit 7Bit-Adresse, Write, Ack=1 (dabei sollte der Transfer abbrechen).
- Transfer mit 7Bit-Adresse, Write, mit Ack-Bestätigung Ack=0.
- Transfer mit 7Bit-Adresse, Read.
- Transfer mit 10Bit-Adresse, Write
- Transfer mit 10Bit-Adresse, Read

Im zweiten Teil des Projekts wurde die JTAG-Instanz in Verilog implementiert. Während des Entwicklungsprozesses wurden die Subkomponenten mithilfe von Testläufen mit überschaubarem Umfang überprüft, um auftretende Fehler unmittelbar zu beheben. Der JTAG Block wurde ebenfalls zu Beginn des Entwicklungsprozesses in zwei Teile aufgeteilt: einem JTAG-Standard Block und einem Boundary-Scan-Register mit Zellen.

Der JTAG-Standard Block besteht aus den erforderlichen Komponenten wie dem TAP-Controller, dem Instruktions-Register und dem Daten-Register. Das Boundary-Scan-Register ist zwar Teil des Daten-Registers, wird aber außerhalb des JTAG-Standard Blocks implementiert. Der

JTAG-Standard Block und das Boundary-Scan-Register mit Zellen bilden zusammen eine JTAG-Schnittstelle und wurden gemeinsam in einer Testumgebung instanziiert und getestet.

Nach der Fertigstellung eines ersten lauffähigen JTAG Blocks wurde mit der Entwicklung einer Testbench begonnen. Dabei wurde ein Initial-Block, die Timescale-Direktive und Verzögerungsanweisungen verwendet, um Signale zu unterschiedlichen Zeitpunkten zu setzen.

Um die Funktionen des JTAGs zu überprüfen, wurde ein Testbench mit zwei Chips erstellt. Jeder Chip enthält eine Instanz des JTAG-Standards und eine des Boundary-Scan-Registers

Abschließend wurde eine Simulation für verschiedene Fälle durchgeführt:

- Test der ExternalPins: Dabei wurde der Ausgangspin-2 von Chip-1 mit dem Eingangspin-1 von Chip-2 verbunden, und der Ausgangspin-3 von Chip-1 mit dem Eingangspin-0 von Chip-2.
- Test des Sample/Preload: Der Sample Vorgang wird vor dem Preload-Register aktiviert. Es erfasst den Zustand des Pins und speichert diesen im Register. Dann wird das Preload-Register aktiviert und der gespeicherte Zustand an den Pin übertragen. Anschließend wird der Zustand des Pins erneut im Sample-Register erfasst und verglichen, um festzustellen, ob der Pin den erwarteten Zustand hatte.
- Test des JTAG-Standard Bypass-Register
- Test des JTAG-Standard Instruktion-Register
- Test des JTAG-Standard ID-Code-Register (Identifikation Code)

Der JTAG hat alle Testfälle erfolgreich durchlaufen und die korrekte Funktion des Entwurfs festgestellt werden konnte.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

N	Number Of Bits
Verilog	Verilog Hardware Description Language
CPU	Central Procesing Unit
Ack	Acknowledgement
Nack	Not- Acknowledgement
MSB	Most significant bit
LSB	Least significant bit
Z	Hochomig
JTAG	Joint Test Action Group
BSR	Boundary-Scan-Register
BSZ/BSC	Boundary-Scan-Zelle/ Boundary-Scan-Cell
IR	Instruktionsregister
DR	Daten-Register
ID-CODE	Identifikationscode
TMS	Test Mode Select/Testmoduswahl
TCK	Test Clock/Prüftakt
TDI	Test Data In
TDO	Test Data Out
TRST	Test Reset
TAP	Test-Access-Port

Literaturverzeichnis

[1] https://www.digikey.at/de/articles/why-the-inter-integrated-circuit-bus-makes-connecting-ics-so-easy

[2] NXP-I2c, UM10204, I2C-Bus specification and user Manual, Rev. 6- 4 April 2014

[3] I2C Primer: What is I2C? (Part 1) | Analog Devices

[4] https://www.i2c-bus.org/highspeed [Online]

[5] M. Halbach; Verilog-Übersicht; TU Darmstadt, FB Informatik; Juni 2006

[6] J. Bhasker; Verilog HDL Synthesis - A Practical Primer;

[7] https://www.corelis.com/education/tutorials/jtag-tutorial/jtag-technical-primer

[8] https://www.jtag.com/

[9] https://vlsitutorials.com/jtag-test-access-port-and-tap-controller/

[10] https://www.allaboutcircuits.com/technical-articles/introduction-to-jtag-test-access-port-tap/

[11] https://trias-mikro.de/wp-content/uploads/2018/07/hds_datasheet.pdf

[12] https://sus.ziti.uni-heidelberg.de/Lehre/WS1920_DST/DST_Fischer_08_Verilog.pdf

[13] J. Bhasker; Verilog HDL Synthesis - A Practical Primer;

[14] Stephan Günther; JTAG-Interface; Stephan Günther, Informationssystemtechnik, TU Dresden

Anhang A: I2C Verilog-Code

A.1 Master_I2C_Top

```
`timescale 1 ns / 1 ps

`default_nettype none

module I2c_master_top
#(
  parameter freq = 66,
  parameter read_from10b_en = 0
  //parameter integer adresse_7_10 = 10
)
(
  input wire        SDA,
  output reg        SCL,


  input wire        sys_clock,  // System clock,
  input wire        reset,
  input wire enable,
  input wire [9:0] slave_address,  //Slave Adresse 7 Bits or 10

  input wire [7:0] write_data,      //Writedata 1 Byte
  input wire rd_or_wr,                  //Read or write Bit
  input wire [3:0] adresse_7_10,

  output reg [7:0]rd_data,              //1 Byte for Output Read

  output reg busy,                  //Bus Besch闌igt
  output reg sda_out,
  output reg sda_out_enable,

  output reg stop_from_master,
  output reg address_ack,
  output reg data_nack,
  output reg read_en,
  output reg data_rd_ack


);



localparam  t_hd_sta = 4 * freq,      //400 = 4us
            t_low = 5 * freq,
            t_high = 5 * freq,
            t_su_sta = 5 * freq,
            t_hold = (freq >> 1), //
                    t data = 4 * freq,
            t_su_sto = 4 * freq;


reg nexte;

reg [8:0] timer;            //Max 511
reg [2:0] restart_count;
reg [6:0] address;              //Hilfsregister fr Slave Adresse
reg [7:0] address2;

reg shit_data7_en;
reg [7:0] data_to_write;        //Data zu schreiben
reg read1_or_write0;

reg adresse10teil2;

reg restart_en;
reg [1:0]cnt_data_10b_en;
```

```
reg [7:0] shift_reg_addr;           //Shift Register fr Slave Adresse


reg adresse7_enable;
reg adresse10_enable;
reg adresse10_read_enable;
reg start_enable;
reg data_enable;


reg rst_data;
reg rst_7b_adresse;
reg rst_10b_adresse;
reg rst_10b_read_adresse;
reg rst_start;

reg [15:0] shift_reg_addr10;            //Shift Register fr Slave Adresse

reg [7:0] shift_addr_10bits_read1;
reg [7:0] shift_addr_10bits_read2;

reg [7:0] shift_reg_data;           //Shift fr Write data

reg[1:0] neustrt_en;
reg [4:0] bit_count;                // Counts bits during shift states.
reg [4:0] bit_count_shit;

reg [4:0] state;      // Main state machine state variable
reg [4:0] rtn_state;  // Return state fr Subroutine

reg [3:0] states;
reg inter_Adress;
reg inter;
reg read_hold;

reg [7:0] read_data;  //  shift register for reads

reg [3:0] scl_startup_count;  // Clock SCL at least 3 times after SDA is detected high

//for Timer
reg timer_enable;
reg t_hd_sta_en;
reg t_low_en;
reg t_high_en;
reg t_su_sta_en;
reg t_hold_en;
reg t_su_sto_en;
reg timer_ready;
reg start_timer;


start do_start(
.sys_clock(sys_clock),
.reset(rst_start),
.start_enable(start_enable),
.adresse_7_10(adresse_7_10),
.read1_or_write0(read1_or_write0),
.address(address),
.address2(address2),
.data_to_write_in(data_to_write),
.shift_reg_addr(shift_reg_addr),
.shift_reg_addr10(shift_reg_addr10),
.shift_reg_data_to_write(shift_reg_data),
.SCL(SCL),
.sda_out(sda_out),
.sda out enable(sda out enable)


);

shift_adresse adresse7bits(
.sys clock(sys clock),
.reset(rst_7b_adresse),
.adresse_enable(adresse7_enable),
.adresse_7_10(adresse_7_10),
.shift_reg_addr(shift_reg_addr),
```

```
.adresse10teil2(adresse10teil2),
.sda_out(sda_out),
.sda_out_enable(sda_out_enable)
);

shift_adresse adresse10bits(
.sys_clock(sys_clock),
.reset(rst_10b_adresse),
.adresse_enable(adresse10_enable),
.adresse_7_10(adresse_7_10),
.shift_reg_addr(shift_reg_addr10),
.adresse10teil2(adresse10teil2),
.sda_out(sda_out),
.sda_out_enable(sda_out_enable)
);

shift_adresse adresse10bits4read(
.sys_clock(sys_clock),
.reset(rst_10b_read_adresse),
.adresse_enable(adresse10_read_enable),
.adresse_7_10(adresse_7_10),
.shift_reg_addr(shift_reg_addr),
.adresse10teil2(adresse10teil2),
.sda_out(sda_out),
.sda_out_enable(sda_out_enable)
);


shift_data Data_shift(
.sys_clock(sys_clock),
.reset(rst_data),
.data_enable(data_enable),
.shift_reg_data(shift_reg_data),
.sda_out(sda_out),
.sda_out_enable(sda_out_enable)
);
// Define states:

  localparam
            idle = 1,
            start = 2,
            clock_low = 3,
            shift_data = 4,
            clock_high = 5,
            stop = 6,
                    ack=7,
                    ack_low =8,
                    ack_stop = 9,
                    shift_read_data_high =10,
                    shift_read_data_low =11,
                    shift_adresse_7bits =12,
                    shift_adresse_10bits =13,
                    ack_for_7b_adresse_high=14,
                    ack_for_7b_adresse_low =15,
                    shift_data_7bits=16,
                    clock_low_7w_data =17,
                    ack_write=18,
                    ack_for_10b_adresse_low =19,
                    shift_data_10bits =20,
                    clock_low_10b_data=21,
                    ack_for_10b_adresse_high =22,
                    pre_restart = 23,
                    restart = 24,
                    shift_adresse_10bits_read=25,
                    clock_low_10b_read =26,
                    clock high 10b read=27,
                    after_start =28,
            wait_for_timer = 30;




always @ (posedge sys_clock or posedge reset)
if (reset)
  begin
```

```
    sda_out_enable <=1;
    sda_out <= 1;
    restart count <=0;
    read_data <= 1'b00000000;
  timer <= t_low;
  state <= idle;
  rtn_state <= idle;
    stop_from_master <=0;
    shit data7 en <=0;
    address <= 0;
    address2 <=0;
    restart_en<=0;
    data_to_write <= 0;
    read1_or_write0 = 0;
    shift addr 10bits read1 <=0;
  shift_addr_10bits_read2 <=0;
  cnt_data_10b_en <=0;
  bit_count <= 0;
    bit_count_shit <=0;
  adresse10teil2<=0;
  SCL <= 1;
    inter <=0;

   read_data <= 0;
   scl_startup_count <= 0;
 end
else
 begin
   if (enable)
     begin

         sda_out_enable <=1;

       if(adresse_7_10 == 7)
        begin
          address = slave_address;
        end
     else
        begin
          address = {4'b1111,1'b0,slave_address[9:8]};
          address2 ={slave_address[7:0]};
        end

       read1_or_write0 = rd_or_wr;
       data_to_write = write_data;

   end     //end else (if reset)

//-------------------------------------------------------------
   case (state)
//-------------------------------------------------------------
     idle:
       begin

         rst_start <=1;

         bit_count <= 0;


               sda_out_enable <=1;
         sda_out <= 1;
         SCL <= 1;

         busy <= 0;  // Not busy

         if (enable & !busy)  //folgelogik
           begin
             state <= start;

             timer_enable <=1;
                     t hd sta en <=1;

             busy <= 1;  // go busy
           end
       end
```

```
//-------------------------------------------------------------
   start:
     begin
       neustrt_en<=0;
       rst_start<=0;
       start_enable <=1;



       timer_enable <=0;
              t_hd_sta_en <=0;
       start_timer <=1;



              if(timer_ready)
                begin

                start_timer <=0;
                //timer_ready <=0;
                //timer_enable <=1;
                //t_hold_en <=1;
                state <=after_start;

                end
       //timer <= t_hd_sta;  // 4.0us from Spezifikation
       //rtn_state <= after_start;
       //state <= wait_for_timer; //zustandfolgelogik
     end
   //---------------------------------------------
   after_start:
   begin
     start_enable <=0;
     rst_7b_adresse <=1;
     rst_10b_adresse <=1;
     rst_10b_read_adresse<=1;
     rst_data <=1;
     state <= clock_low; //zustandfolgelogik
   end

   //---------------------------------------------
   clock_low :
     begin
     rst_data <=0;
     SCL <= 0;
           sda_out_enable <=1;

           timer_enable <=0;
           t_hd_sta_en <=0;
     start_timer <=1;

           if(adresse_7_10 == 7)
             begin
                        //rtn_state <= shift_adresse_7bits;

                        if(timer_ready)
                       begin
                         start_timer <=0;
                         timer_ready <=0;
                         //timer_enable <=1;
                         //t_hold_en <=1;
                         state <=shift_adresse_7bits;
                       end

                        end
           else
             begin
            rtn_state <= shift_adresse_10bits;
             end

          // timer <= t_hold;
//                    state <= wait_for_timer;
     end
```

```
//-------------------------------------------------------
     shift_adresse_7bits :
      begin
        if(bit count==8)
          begin
           sda_out_enable <=0;
             rtn_state <= ack_for_7b_adresse_high;
          end
      else
         begin
          sda_out_enable <=1;
          rst_7b_adresse <=0;
          adresse7_enable <=1;
            rtn_state <= clock_high;
         end

         timer <= t_low; // 4.7us Minimum aus Spezifikation
            state <= wait_for_timer;
      end


//-------------------------------------------------------
        ack_for_7b_adresse_high :
        begin
         sda_out_enable <=0;
         SCL <= 1;
         address_ack = SDA;  //Extern
         timer <= t_low;  // 4.0us from spec
         rtn_state <= ack_for_7b_adresse_low;
         state <= wait_for_timer;
        end

//-------------------------------------------------------
        ack_for_7b_adresse_low:
        begin
         SCL <= 0;
         if(address_ack ==1)
         begin
           timer <= t_low;  // 4.0us from spec
           rtn_state <= ack_stop;

         end
         else if(read1_or_write0)
         begin
           bit_count =0;
           read_en <=1;
           timer <= t_low;  // 4.0us from spec
           rtn_state <= shift_read_data_high;
         end
         else if(read1_or_write0 == 0)
         begin
           bit_count =0;
           timer <= t_hold;  // 4.0us from spec
           rtn_state <= shift_data_7bits;
         end

         state <= wait_for_timer;
        end


//-------------------------------------------------------
   shift_data_7bits :
   begin
    if(bit_count == 8)
    begin
      sda out enable <=0;
          rtn_state <= ack_for_7b_adresse_high;
    end
    else
    begin
   data_enable <=1;
         shit data7 en <=1;
         rtn_state <= clock_high;
    end
   timer <= t_low; // 4.7us min from spec
         state <= wait_for_timer;
```

```
      end
   //-------------------------------------------------------
    clock low 7w data :
    begin
       SCL <= 0;
             sda_out_enable <=1;
       timer <= t_hold;
               rtn_state <= shift_data_7bits;
             state <= wait for timer;
     end
//-------------------------------------------------------
  ack_write:
   begin

     SCL <=1;
     sda_out_enable <=1;
           sda_out =1;
           address_ack=1; //ack_write
           data_rd_ack =1;
           timer <= t_low;  // 4.0us from spec
           rtn_state <= ack_for_7b_adresse_low;
           state <= wait_for_timer;

   end

//-------------------------------------------------------
   shift_read_data_high:
        begin
        sda_out_enable <=0;
        SCL <= 1;
        read_data <= {read_data[6:0],SDA};
        timer <= t_high; // 4.7us min from spec
        rtn_state <= shift_read_data_low;
        state <= wait_for_timer;
        end
//-------------------------------------------------------
       shift_read_data_low:
           begin
           SCL <= 0;
           if(bit_count == 7)
               begin
                  sda_out_enable <=1;
                  rtn_state <= ack_write;
               end
           else
               begin
                    rtn_state <= shift_read_data_high;
               end
       timer <= t_low;
           state <= wait_for_timer;
           bit_count <= bit_count + 1;
           end
//-------------------------------------------------------
  shift_adresse_10bits:
        begin
          if((bit_count==8))
          begin
            sda_out_enable <=0;
            rtn_state <= ack_for_10b_adresse_high;
            cnt_data_10b_en <= cnt_data_10b_en +1;
            restart_en <= 1;
          end
          else
          begin
            sda out enable <=1;
            rst_10b_adresse <=0;
            adresse10_enable <=1;
            rtn_state <= clock_high;
          end

            timer <= t low; // 4.7us Minimum aus Spezifikation
            state <= wait_for_timer;
        end
//-------------------------------------------------------
  ack_for_10b_adresse_high :
```

```
        begin
            sda_out_enable <=0;
            SCL <= 1;
            address ack = SDA;
            timer <= t_low;  // 4.0us from spec
            rtn_state <= ack_for_10b_adresse_low;
            state <= wait_for_timer;
        end
//-------------------------------------------------------
 ack for 10b adresse low :
  begin
    SCL <= 0;
            if(address_ack ==1)
            begin
              rtn_state <= ack_stop;
              timer <= t low;  // 4.0us from spec
            end
            else if((cnt_data_10b_en == 2)&(read1_or_write0 ==1))
            begin
                if(restart_en == 1)
                begin
                rtn_state <= pre_restart;
          timer <= t_low;  // 4.0us from spec
                end

                bit_count =0;
            end
            else if((cnt_data_10b_en == 2)&(read1_or_write0 ==0))
            begin
              rtn_state <= shift_data_10bits;
              bit_count =0;
              timer <= t_hold;  // 4.0us from spec
            end
            else
            begin
              bit_count =0;
              rtn_state <= shift_adresse_10bits;
              timer <= t_hold;  // 4.0us from spec
            end

            state <= wait_for_timer;
  end

//-------------------------------------------------------
    shift_data_10bits:
    begin
       if(bit_count == 8)
       begin
         if(read1_or_write0)
         restart_en <= 1;
         sda_out_enable <=0;
             rtn_state <= ack_for_10b_adresse_high;
       end
       else
       begin
        sda_out_enable <=1;
        data_enable <=1;
            rtn_state <= clock_high;
       end
      timer <= t_low; // 4.7us min from spec
            state <= wait_for_timer;

    end
//-------------------------------------------------------
   clock_low_10b_data :
     begin
       SCL <= 0;
             sda_out_enable <=1;
       timer <= t_hold;
               rtn_state <= shift_data_10bits;
             state <= wait_for_timer;
     end
//-------------------------------------------------------
   pre_restart :
     begin
       sda_out_enable <=1;
```

```
       SCL<=1;
       timer <= t_low;
              rtn_state <= restart;
            state <= wait for timer;
     end
//-------------------------------------------------------
   restart :
     begin
       SCL<=1;
       sda out enable <=1;
              sda_out=0;
              timer <= t_hd_sta;  // 4.0us from Spezifikation
       rtn_state <= clock_low_10b_read;
       state <= wait_for_timer;
     end


//-------------------------------------------------------
   shift_adresse_10bits_read:
     begin
      if((bit_count==8))
          begin
            read1_or_write0 <=1;
            sda_out_enable <=0;
            rtn_state <= ack_for_7b_adresse_high;
          end
          else
          begin
            sda_out_enable <=1;
            rst_10b_read_adresse<=0;
            adresse10teil2 <=1;
            adresse10_read_enable <=1;
            rtn_state <= clock_high_10b_read;
          end
            timer <= t_low; // 4.7us Minimum aus Spezifikation
            state <= wait_for_timer;
     end


//-------------------------------------------------------
   clock_high_10b_read:
   begin
     SCL <= 1;
     bit_count <= bit_count + 1;
     timer <= t_high;
     adresse10_read_enable <=0;
     rtn_state <= clock_low_10b_read;
     state <= wait_for_timer;
   end
//-------------------------------------------------------
   clock_low_10b_read:
   begin
     SCL <= 0;
           sda_out_enable <=1;
     timer <= t_hold;
           rtn_state <= shift_adresse_10bits_read;
           state <= wait_for_timer;

   end

//-------------------------------------------------------

     clock_high:
       begin

         SCL <= 1;
         if (SCL)
           begin
               bit_count <= bit_count + 1;

                 if((inter==1)|(address_ack==1))
                 begin
                  timer <= t_su_sto;  // 4.0us from spec
          rtn_state <= stop;
         state <= wait_for_timer;
                 end
```

```
              else if(shit_data7_en ==1)
              begin
              data enable <=0;
              timer <= t_high;
      rtn_state <= clock_low_7w_data;
      state <= wait_for_timer;
      end
      else if(cnt_data_10b_en == 2)
      begin
      timer <= t_high;
      data_enable <=0;
      rtn_state <= clock_low_10b_data;
      state <= wait_for_timer;
      end
              else
              begin
              adresse7_enable <=0;
              adresse10_enable <=0;
              timer <= t_high;
      rtn_state <= clock_low;
      state <= wait_for_timer;
              end

        end
    end

//-----------------------------------------------------------

    ack_stop : //Ack Stop
    begin
    if(!SCL)
    begin
    sda_out_enable <=1;
    sda_out <= 0;

    timer <= t_low;  // 4.0us from spec
    rtn_state <=clock_high ;
    state <= wait_for_timer;
    end
    end
//-----------------------------------------------
stop:
    begin

      sda_out_enable <=1;
        sda_out <= 1;

      if (SDA)
        begin
          if (read1_or_write0) // reading
            begin
              rd_data <= read_data;
              timer <= t_su_sta;  // Time to start 4.0 us aus Spezif PDF
              rtn_state <= idle;  // Auslesen is done dann to IDLE
            end
          else              // writing
            begin
                    if(inter)
                    begin
                    timer <= t_su_sta;
        rtn_state <= idle;
        state <= wait_for_timer;
                    end
                    else
                    begin
          timer <= t_su_sta;
          rtn_state <= idle;
          state <= wait_for_timer;
            end
                  end
        end
    end
//-----------------------------------------------------------

    //this Statemaschine haltet ein state bis der Timer auf Null geht
```

```
      wait_for_timer:   //Extern
        begin

          if (timer > 0)
            begin
              timer <= timer - 1;
            end
          else
            begin
              state <= rtn state;
            end

        end
    endcase

end
endmodule


`default_nettype wire
```

A.2 Do_shift

```
//
// Verilog Module I2C_mymaster_lib.do_shift
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 18:01:08 01/21/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/1ps
module do_shift (
input wire        sys clock,  // System clock,
input wire        reset,
input wire        start_shift,
input wire        SDA,
input wire   addr1_en,
input wire   addr2_en,
input wire   addr3 en,
input wire   data_en,

input reg[7:0]    addr1,
input reg[7:0]    addr2,
input reg[7:0]    addr3,
input reg[7:0]    data_to_shift,

input reg[8:0] frequence,

input wire SCL,
output reg sda_out,
output reg sda_out_enable,

output reg shift_done




);

reg [8:0] timer;            //Max 511
reg [7:0] register_toshift =0;

reg [7:0]addr1_reg;
reg [7:0]addr2_reg;
reg [7:0]addr3_reg;

reg [2:0]st_case ;
reg [2:0]rtn_st_case ;
```

```
reg [3:0]bit_count;

reg addr1_en_intern;
reg addr2 en intern;
reg addr3_en_intern;
reg data_en_intern;



//localparam  t hd sta = 4 * 100,     //400 = 4us
//            t_low = 5 * 100,
//            t_high = 5 * 100,
//            t_su_sta = 5 * 100,
//            t_hold = 50, //
//            t_su_sto = 400;



//
//localparam  clock_low =1,
//            vor_clock_low=2,
//            shift =3,
//            clock_high=4,
//            stop_shift =5,
//            wait_for=6;


//for Timer
reg timer_enable;
reg t_hd_sta_en;
reg t_low_en;
reg t_high_en;
reg t_su_sta_en;
reg t_hold_en;
reg t_su_sto_en;
reg timer_ready;
reg start_timer=0;
reg rst_done;
reg divider_enable;
reg scl_intern;
reg check_ack_en;
reg check_ack_done;
reg ack_result;

reg go_check_ack;


reg wait_for_0or1;

//assign SCL = scl_intern;

wait_for_timer shift_timer(
.sys_clock(sys_clock),
.reset(reset),
.wait_for_0or1(wait_for_0or1),
.timer_enable(timer_enable),
.ready(timer_ready)

);

//
//always@(reset )
//begin
//    //if(reset)
////    begin
////    shift done<=0;
////
////    if(addr1_en)
////        register_toshift<=addr1;
////    else if(addr2_en)
////        register_toshift<=addr2;
////    else if(addr3 en)
////        register_toshift<=addr3;
////    else if (data_en)
////        register_toshift<=data_to_shift;
////
```

```
////
////    bit_count <=0;
////    go_check_ack<=0;
////    timer enable <=1;
////    end
//end

localparam start=1,
           shifft=2,
           wait for=3;


reg [2:0]mode;

always@(posedge sys_clock, posedge reset)   //negedge SCL or posedge timer_ready or posedge
reset
begin

 if(reset)
    begin
    shift_done<=0;

    if(addr1_en)begin
        register_toshift<=addr1;end
    else if(addr2_en)begin
        register_toshift<=addr2;end
    else if(addr3_en)begin
        register_toshift<=addr3; end
    else if (data_en) begin
        register_toshift<=data_to_shift;end


    bit_count <=0;
    go_check_ack<=0;
    mode <= start;
     wait_for_0or1<=0;
    //timer_enable <=1;
    end

else
begin
 if(start_shift)
  begin


    //switch
    case(mode)
      //fsf

    start:
    begin
      if(!SCL)
      begin
       timer_ready <=0;

       wait_for_0or1 <=0;
       timer_enable <=1;
       if(timer_ready)
       begin
         sda_out_enable <=1;
         mode <=shifft;
       end
     end
    end

    shifft:
    begin
      timer_enable <=0;
        timer_ready <=0;

        sda_out_enable <=1;

        sda_out <= register_toshift[7];
        register_toshift <= {register_toshift[6:0],1'b0};
        bit_count <= bit_count+1;
        mode<=wait_for;
```

```
    end

    wait_for:
    begin
       timer_ready <=0;
       sda_out_enable <=1;
       wait_for_0or1 <=1;
       timer_enable <=1;

       if(timer ready)
       begin
         if(bit_count==8)
           begin
        shift_done<=1;
        timer_enable <=0;
        timer ready<=0;
        sda_out_enable <=0;
        sda_out<=0;
        bit_count<=0;
        //mode <=wait_for;
          end
          else
          begin

         mode <=start;
         timer_enable <=0;end
       end

    end

 endcase

 end //start_shift

end//end else reset

end

endmodule
```

A.3 Start

```
//
// Verilog Module I2C_mymaster_lib.start
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 11:24:29 01/17/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/1ps

module start(

  input wire        sys_clock,  // System clock,
  input wire        reset,
  input wire        start_enable,
  input wire  adresse_10_7,
  input wire read1_or_write0,
  input reg [6:0] address,
  input reg [7:0] address2,
  input reg [7:0]data_to_write_in,
  output reg SCL,
  output reg [7:0]shift_reg_addr,
  output reg [7:0]shift_reg_addr2,
  output reg [7:0]shift_reg_addr3,
  output reg [7:0] shift_reg_data_to_write,
  output reg start_done,
  output reg sda_out,
  output reg sda_out_enable
);

localparam start=1,
```

```
          wait_for_timer=2;


reg [1:0]mode;
reg [8:0]timer;

always @ (posedge sys_clock or posedge reset)
begin
  if (reset)
  begin
    mode <= start;
    timer <= 450;
  end

  else
    begin
      if(start_enable)
      begin

      //--------------------
      case(mode)
      //--------------------
      start:
      begin

      //sda_out_enable <=1;
            //sda_out <= 0;
            //SCL <= 1;

        if(adresse_10_7 == 0)
            begin
              if (read1_or_write0)
                        begin
                         shift_reg_addr <= {address,read1_or_write0};
                         shift_reg_addr2<=0;
                         shift_reg_addr3 <=0;
                         shift_reg_data_to_write<=0;

                        end
                       else
                        begin
                          shift_reg_addr <= {address,read1_or_write0};
                          shift_reg_addr2<=0;
                          shift_reg_addr3 <=0;
                          shift_reg_data_to_write <={data_to_write_in};

                        end

         end //end if 7
         else
            begin //adresse = 10

              if(read1_or_write0==0) //Schreibvorgang in 10bits slave
               begin

                shift_reg_addr <= {address,read1_or_write0};
                shift_reg_addr2 <={address2};
                shift_reg_addr3<=0;
                          shift_reg_data_to_write <={data_to_write_in};


                        end //schreibvorgang in 10bits

                        else //lesevorgang from 10bits slave
                        begin
                          shift_reg_addr <= {address,1'b0};
                          shift_reg_addr2 <={address2};
                          shift_reg_addr3 <= {address,read1_or_write0};
                          shift_reg_data_to_write <=0;
                        end

          end


      mode <= wait_for_timer;
```

```
      end //end start

      wait for timer :
      begin

      if (timer > 0)
      begin
        timer <= timer - 1;
        start done <= 0;
      end
      else
      begin
        start_done <= 1;
      end

      end

      endcase

      //
      end
  end
end //always's end
endmodule
```

A.4 Stopp

```
//
// Verilog Module I2C_mymaster_lib.stop_mod
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 15:04:33 01/24/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module stop_mod
(
input wire        sys_clock,  // System clock,
input wire        reset,
input wire        stop_enable,

output reg stop_done,

output reg SCL,
output reg sda_out,
output reg sda_out_enable

);

parameter DELAY =950;
reg [9:0]count =8;

always @(posedge sys_clock, posedge reset)
begin
  if(reset)
  begin
     SCL <=0;
     sda_out <=0;
     sda_out_enable<=0;
  end

  else if(stop_enable)
  begin

   if(count==DELAY)
    begin
      count <=0;
      stop_done<=1;
    end
    else if(count==(DELAY)/2)
    begin


      SCL <=1;
      count <=count+1;
    end
    else
    begin
```

```
     sda_out <=0;
     sda_out_enable<=1;
      //stop_done<=0;
      count <=count+1;
    end
  end
end

endmodule
```

A.5 Wait_For_Timer

```
//
// Verilog Module I2C_mymaster_lib.wait_for_timer
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 20:57:02 01/20/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/1ps
module wait_for_timer
(
  input wire        sys_clock,  // System clock,
  input wire        reset,
  input wire        timer_enable,
  input wire        wait_for_0or1,
  output reg        ready

);

reg[20:0] DELAY =100;

reg [20:0] count;
reg [1:0] waitfor01;
initial ready=0;

always @(posedge sys_clock, negedge reset)
begin
  if(reset)
  begin
    count=0;
    waitfor01<=0;
  end

  else if(timer_enable)
  begin
    waitfor01 <= wait for 0or1;
    if(waitfor01==0)
    begin
      DELAY=100;
    end
    else if(waitfor01)begin
      DELAY =1950;
    end

    if(count==(((DELAY)/2)-1))
      begin
      ready <=1;
      count=0;
    end
    else
    begin
      ready <=0;
      count = count +1;
    end
   end
end
endmodule
```

A.6 Restart

```
//
// Verilog Module I2C_mymaster_lib.restart_md
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 19:30:02 01/26/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/1ps
module restart_md(
  input wire        sys_clock,  // System clock,
  input wire        reset,
  input wire        restart_enable,

  output reg     restart_done

);
parameter DLEAY=900;

reg [9:0]count =0;

always @(posedge sys_clock)
begin

  if(restart_enable)
  begin
      if(count==DLEAY)
      begin
        count<=0;
        restart_done<=1;
      end
      else
      begin
        count <=count+1;
      end

  end
end


endmodule
```

A.7 Do_Shift_Read

```
//
// Verilog Module I2C_mymaster_lib.do_shift_read
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 11:06:43 01/25/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/1ps
module do_shift_read
(
input wire        sys_clock,  // System clock,
```

```
input wire        reset,
input wire        read_shift_enable,
input wire SCL,
input wire SDA,
output reg read_done,
output reg [7:0] read_data,
output reg sda_out,
output reg sda_out_enable
 );

reg[7:0] read_data_shift;
reg [3:0]bit_count;
reg ack_mode;

always @(posedge SCL, posedge reset)
begin
  if(reset)
  begin
    read_done<=0;
    read_data<=0;
    read_data_shift=0;
    bit_count =0;
    ack_mode=0;
  end
  if(read_shift_enable)
  begin
      if(!ack_mode)
        begin
          sda_out_enable <=0;
          read_data_shift <={read_data_shift[6:0],SDA};
          bit_count <=bit_count+1;
        end
       else
        begin
          bit_count <=bit_count+1;
          sda_out_enable <=1;
          sda_out <=1;
        end
  end
end

always @(negedge SCL)
begin

  if(bit_count==8)
  begin
    sda_out_enable <=1;
    sda_out <=1;
    ack_mode <=1;
  end
  else if(bit_count==9)
  begin
       sda_out_enable <=1;
       sda_out <=0;
       read_data <=read_data_shift;
       read_done<=1;
  end

end

endmodule
```

A.8 Clk_scl (Clock Teiler)

```
//
// Verilog Module I2C_mymaster_lib.clk_scl
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 12:11:54 01/22/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/1ps
module clk_scl(

input wire reset,
input wire ref_clk,
input wire enable,
output reg i2c_scl
);

reg [9:0]DELAY;
reg [9:0] count;

always @(posedge ref_clk, posedge reset)
begin
  if(reset)
  begin
    count=0;
    i2c_scl<=0;
    DELAY=1000;
  end
  else if(enable)
  begin
    if(count==(((DELAY)/2)-1))
      begin
      i2c_scl = !i2c_scl;
      count=0;
    end
    else
    begin
      count = count +1;
    end
   end
end

endmodule
```

A.9 Check_Ack

```
//
// Verilog Module I2C_mymaster_lib.check_ack
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 20:30:46 01/21/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/1ps
module check_ack (
input wire          sys_clock,  // System clock,
input wire          reset,
input wire          check_ack_en,
input wire          SDA,

output reg          check_ack_done,
output reg          ack_result



);

reg go_to_out ;
reg ack_n;
always@(posedge sys_clock, posedge reset)
begin
  if(reset)
  begin
    go_to_out=0;
    ack_n=0;
  end
  else if(check_ack_en)
  begin
    ack_n <= SDA;
    go_to_out <=1;
  end
  else
  begin
    check_ack_done <=0;
  end

end

always@(negedge sys_clock)
begin
   if(go_to_out)
   begin
     check_ack_done <=1;
     ack_result <= ack_n;
     go_to_out <=0;
```

```
    end
end

endmodule
```

A.10 Finite State Machine

```
//
// Verilog Module I2C_mymaster_lib.finite_state_machine_i2c
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 20:59:12 01/29/21
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module finite_state_machine_i2c(

input wire        sys_clock,  // System clock,
input wire        reset,

input wire  start_done,
input wire enable,
input wire shift_done,
input wire stop_done,
input wire read_done,
input wire restart_done,
input wire check_ack_done,
input wire adresse_10_7,
input wire n_ack,
input wire read1_or_write0,

output reg stop_enable,
output reg divider_enable,
output reg sda_out_enable,
output reg sda_out,
output reg SCL,
output reg rst_start,
output reg start_enable,
output reg busy,
output reg restart_enable,
output reg rst_shiften,
output reg addr1_en,
output reg addr2_en,
output reg addr3_en,
output reg data_en,
output reg start_shift,
output reg check_ack_enable,
output reg read_en,
output reg read_shift_enable



) ;

localparam
            idle = 1,
            start = 2,
            shift_adressse_1 =3,
            shift_adressse_2=4,
            shift_adressse_3 =5,
            shift_data=6,
            shift_read_data=7,
            write_ack=8,
            restart =9,
            ack_1=10,
            ack_2=11,
            ack_3=12,
            stop_s=13;


reg [4:0]next_state;
reg [4:0]current_state;
reg neu_start=0;
//getakteter Block

always @ (posedge sys_clock or posedge reset)
begin
  if (reset)
      current_state <= idle;
```

```
  else
      current_state <= next_state;

end

//state machine

always @(current_state or start_done or enable or shift_done or stop_done or read_done or restart_done or
check_ack_done)
begin

   case (current_state)
 //-------------------------------------------------------------
      idle:
        begin
         if(enable)
          next_state <=start;
          else
          next_state <=idle;
        end

 //-------------------------------------------------------------
    start:
      begin
        if(start_done & !neu_start)
          next_state <=shift_adressse_1;
        else if(start_done & neu_start)
          next_state <=shift_adressse_3;
        else
          next_state <=start;
      end
 //-------------------------------------------------------------
    shift_adressse_1:
      begin

        if(shift_done)
           next_state <=ack_1;
        else
           next_state <=shift_adressse_1;
      end
//-------------------------------------------------------------
    ack_1:
    begin
      if(check_ack_done & n_ack)
        next_state <=stop_s;
      else if((adresse_10_7==0)&(!read1_or_write0)&((check_ack_done & !n_ack)))
        next_state<=shift_data;
      else if((adresse_10_7==0)&(read1_or_write0)&(check_ack_done & !n_ack))
         next_state<=shift_read_data;
      else if((adresse_10_7==1)&(check_ack_done & !n_ack))
          next_state<=shift_adressse_2;
      else
          next_state <=ack_1;

    end
//-------------------------------------------------------------



    shift_adressse_2:
    begin
    if(shift_done)
      next_state <= ack_2;
    else
      next_state <= shift_adressse_2;
    end


    ack_2:
    begin
    if(check_ack_done & n_ack)
        next_state <= stop_s;
    else if(check_ack_done & !n_ack & !read1_or_write0)
        next_state <= shift_data;
    else if(check_ack_done & !n_ack & read1_or_write0)
        next_state <= restart;
    else
      next_state <= ack_2;
    end



    shift_data:
    begin
      if(shift_done)
      next_state <= ack_2;
      else
```

```
      next_state <= shift_data;
    end


    shift_read_data :
    begin
      if(read_done)
       next_state <= stop_s;
      else
      next_state <= shift_read_data;

    end

    restart:
    begin
      if(restart_done)
      next_state <= start;
      else
      next_state <= restart;
    end


    shift_adressse_3:
    begin
      if(shift_done)
      next_state <= ack_3;
      else
      next_state <= shift_adressse_3;
    end

    ack_3:
    begin
      if((check_ack_done & !n_ack))
        next_state <= shift_read_data;
      else if((check_ack_done & n_ack))
       next_state <= stop_s;
      else
        next_state <= ack_3;

    end

    stop_s:
    begin
      if(stop_done)
        next_state <=idle;
      else
        next_state<=stop_s;
    end



    endcase

end //end always 2

// current state

always @(current_state)
begin


  rst_shiften <=0;

  start_shift <=0;
  addr1_en<=0;
  addr2_en <=0;
  addr3_en<=0;
  data_en<=0;
  start_enable<=0;


  case(current_state)

     idle:
        begin
          stop_enable<=0;
          rst_start <=1;
          //bit_count <= 0;
                sda_out_enable <=1;
          sda_out <= 1;
          SCL <= 1;
          busy <= 0;  // Not busy

       end
```

```
start:
 begin
 rst_start <=0;
 divider_enable<=0;
 restart_enable<=0;

 sda_out_enable<=1;
 sda_out<=0;

 rst_shiften <=1;
 if(neu_start)
 begin
   addr3_en<=1;
   //divider_enable<=1;
 end
 else
 begin
   addr1_en<=1;
 end


 busy <= 1;
 start_enable <=1;

end

//-------------

shift_adressse_1:
 begin
 SCL <=0;
 divider_enable<=1;
 start_enable<=0;


 start_shift<=1;
 rst_shiften<=0;
 end


 ack_1:
 begin
  start_shift<=0;
  sda_out_enable <=0;
  check_ack_enable <=1;
  if(adresse_10_7==0 & !read1_or_write0 )
  begin
    data_en<=1;
  end
  else
  begin
    addr2_en <=1;
  end

  rst_shiften<=1;

 end
 //------

 shift_adressse_2:
 begin
 sda_out_enable <=1;
 check_ack_enable <=0;
  rst_shiften <=0;
 start_shift<=1;
 end

 ack_2:
 begin
  sda_out_enable <=0;
  start_shift<=0;
  check_ack_enable <=1;

  if(adresse_10_7==1 & !read1_or_write0 )
  begin
    rst_shiften<=1;
    data_en<=1;
  end
  else if(adresse_10_7==1 & read1_or_write0 )
  begin
    divider_enable<=1;
  end
 end

 ack_3:
 begin
  sda_out_enable <=0;
```

```
        start_shift<=0;
        check_ack_enable <=1;
      end


      shift_data:
      begin
      sda_out_enable <=1;
      check_ack_enable <=0;
      //data_en<=1;
      //rst_shiften<=1;
      start_shift<=1;
      end


      shift_read_data :
      begin
        check_ack_enable <=0;
        start_shift<=0;
        read_en<=1;
        read_shift_enable <=1;
      end


      restart:
      begin

        check_ack_enable <=0;
        start_shift<=0;
        sda_out_enable <=1;
        sda_out <= 1;
        neu_start <=1;
        rst_start <=1;
        restart_enable<=1;
        addr3_en<=1;
        rst_shiften <=1;
      end


      shift_adressse_3:
      begin

        rst_shiften <=0;
        check_ack_enable <=0;
        start_enable<=0;

        divider_enable<=1;
        //sda_out_enable <=0;


        start_shift<=1;

      end

      stop_s:
      begin
        //start_done<=0;
        check_ack_enable <=0;
        read_en<=0;
        read_shift_enable <=0;
        stop_enable<=1;
        start_shift<=0;
        divider_enable<=0;
        sda_out_enable <=1;
        sda_out <= 0;

      end


endcase;

end

endmodule
```

A.11 Testbench

```
`timescale 1ns / 1ps


module i2c_mymaster_tb;

// Inputs
reg sys_clock;
reg reset;

reg enable;
reg read_en;

reg [7:0]write_data;
reg [9:0]slave_address;

reg rd_or_wr;

wire [7:0]rd_data;
wire busy;

wire SDA_in;
reg SDA_en;
reg go_to_read;

reg stop_from_master;

reg goto;
reg goto2;
reg goto3;
reg goto4;
reg goto5;


reg sda_out;
reg sda_out_enable;

reg adresse_10_7;
reg SDA_en_tb ;
reg SDA_out_tb;
// Outputs

 wire address_ack;
 wire data_nack;
 wire data_rd_ack;


wire SDA;
wire SCL;


assign SDA_in = SDA;

assign SDA = sda_out_enable ? sda_out : 1'bZ; //I2c_Master

assign SDA = SDA_en_tb ? SDA_out_tb : 1'bZ; //TestBench Tri-State


pullup (SDA);


// Instantiate the Unit Under Test (UUT)
Master_i2c_TOP
#(
  .freq(100)
)
 uut
(
  .SDA           (SDA),
  .adresse_10_7(adresse_10_7),
  .sda_out_enable (sda_out_enable),
  .sda_out (sda_out),
  .enable(enable),
  .SCL           (SCL),
  .sys_clock     (sys_clock),
  .reset         (reset),
  .read_en(read_en),
```

```
    .slave_address(slave_address),
    .write_data    (write_data),
    .rd_or_wr      (rd_or_wr),
    .rd_data       (rd_data),
    .busy          (busy),
    .stop_from_master(stop_from_master),
    .address_ack (address_ack),
    .data_nack (data_nack),
    .data_rd_ack (data_rd_ack)

);

initial begin

    // Initialize Inputs
    sys_clock = 0;
    SDA_en_tb =0;
    reset = 1;
    SDA_en_tb =0;
    SDA_out_tb = 1;
    //SDA_en =1;
    read_en =0;
    //slave_address =7'b1010101;
    slave_address =10'b1001010110;
    write_data =8'b10100010;
    rd_or_wr = 1;      //read 1 / write 0
    adresse_10_7 =1;  // 0=7 // 1=10bits
    enable = 0;
    //Wait 100ns for global reset to finish
    #101;
    reset = 0;
    // Add stimulus here
    #30000
    @ (posedge sys_clock) enable <= #1 1;   //enable fr Start  wird/soll Synchron zum Clock
    @ (posedge sys_clock) enable <= #1 0; // beim nächsten clock direkt wiedr auf 0

    #87000                //in dieser Zeit wird der Master Receiver / Testbench Sender damit der Master den
ACK empfangen kann
    //goto =1;
       //Adresse ACK
     SDA_en_tb = 1;
     SDA_out_tb = 0;
    #7540
    //#24319              //warte eine Periodendauer 10us und wird dann der Testbench receiver

    SDA_en_tb=0;

    //Write for 7-bits
    if((!rd_or_wr)&(adresse_10_7 ==0))
    begin
    #82200
    goto2 =1;
    SDA_en_tb = 1;
    SDA_out_tb = 1;
   // write_data =8'b01000001;
//    @ (posedge sys_clock) enable <= #1 1;   //enable fr Start  wird/soll Synchron zum Clock
//    @ (posedge sys_clock) enable <= #1 0;
    #7800
    SDA_en_tb =0;
    #80000
    goto3 =1;
    #10000
    SDA_en_tb =0;
    end


    //Write for 10 Bits

    if((!rd_or_wr)&(adresse_10_7 ==1))
    begin
    #82200
    goto2 =1;
    SDA_en_tb = 1;
    SDA_out_tb = 0;
    #7790
    SDA_en_tb =0;

    #82200
    goto3 =1;
    SDA_en_tb = 1;
    SDA_out_tb = 1;
    #7800
    SDA_en_tb =0;
    end


    if((rd_or_wr)&(adresse_10_7 ==1))
```

```
  begin
  #82200
  goto2 =1;
   SDA_en_tb = 1;
  SDA_out_tb = 0;

  #7790
  SDA_en_tb =0;

  #96900
  goto3 =1;
  SDA_en_tb = 1;
  SDA_out_tb = 0;
  #7600
  goto2<=0;
//  SDA_en_tb =0;
  #100
  if (read_en)    //wird gepfft und entschieden ob der Master ein Read oder Write anfordert.  beim 1 wird
ein Read
  begin
  SDA_en_tb =1;
  SDA_out_tb = 1;
  go_to_read =1;

  //#78000
  #79700
  go_to_read =0;
  SDA_en_tb = 0;
  SDA_out_tb = 0;
  end
  end

  if((rd_or_wr)&(adresse_10_7 ==0))
  begin
  #6
  if (read_en)    //wird gepfft und entschieden ob der Master ein Read oder Write anfordert.  beim 1 wird
ein Read
  begin
  SDA_en_tb =1;
  SDA_out_tb = 1;
  go_to_read =1;

  //#78000
  #79985
  go_to_read =0;
  SDA_en_tb = 0;
  SDA_out_tb = 0;
  end
  end

end

  always sys_clock = #5 !sys_clock;


  always @ (negedge SCL)
  begin
    if(go_to_read)
    begin
    SDA_en_tb = 1;
    SDA_out_tb =!SDA_out_tb;   //Adresse ACK
    end

  end


  always @ (negedge sda_out_enable)
  begin

    if(goto)
    begin
    SDA_en_tb = 1;
    SDA_out_tb = 1;    //Adresse ACK
    goto =0;
    end

    if(goto2)
    begin

    SDA_en_tb = 1;
    SDA_out_tb = 0;    //Adresse ACK
    goto2 =0;
    end

    else if(goto3)
    begin
    SDA_en_tb = 1;
```

```
    SDA_out_tb = 0;   //Adresse ACK
    goto3 =0;
    end


   end

  endmodul
```

Anhang B: JTAG Verilog-Code

B.1 TAP_Top

```
//
// Module Masterarbeit_lib.Tap_Top.struct
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 14:33:23 12/24/22
//
// Generated by Mentor Graphics' HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module Tap_Top(
   // Port Declarations
   input   wire      BSR_in,
   input   wire      TDI,
   input   wire      Tms,
   input   wire      clk,
   input   wire      rst,
   output  wire      TDO_tap,
   output  wire      clock_dr_out,
   output  wire      enable,
   output  wire      mode,
   output  wire      shift_dr_out,
   output  wire      update_dr_out
);


// Internal Declarations


// Local declarations

// Internal signal declarations
wire        Bypass_TDO;
wire        ClockIR;
wire  [1:0] DR_Decoder;
wire        IDcode_TDO;
wire        Reset;
wire        ShiftIR1;
wire        UpdateIR;
wire        dout;
wire        dout1;
wire        instruction_TDO;
wire  [3:0] to_instruction_Decoder;

// ModuleWare signal declarations(v1.12) for instance 'U_1' of 'mux'
reg mw_U_1temp_dout;
```

```
// ModuleWare signal declarations(v1.12) for instance 'U_5' of 'mux'
reg mw_U_5temp_dout;


// Instances
IR_Decoder U_4(
    .IR_Code     (to_instruction_Decoder),
    .DR_Decoder  (DR_Decoder)
);

Instruction_Register_top U_2(
    .TDI                    (TDI),
    .clk                    (clk),
    .clock_IR               (ClockIR),
    .rst                    (Reset),
    .shift_IR               (ShiftIR1),
    .update_IR              (UpdateIR),
    .instruction_TDO        (instruction_TDO),
    .to_instruction_Decoder (to_instruction_Decoder)
);

Mode_generate U_8(
    .DR_Decoder (DR_Decoder),
    .mode       (mode)
);

ShiftRegister_DR_Top U_3(
    .Clock_DR   (clock_dr_out),
    .Shift_DR   (shift_dr_out),
    .TDI        (TDI),
    .clk        (clk),
    .rst        (Reset),
    .Bypass_TDO (Bypass_TDO),
    .IDcode_TDO (IDcode_TDO)
);

TapController U_0(
    .TMS      (Tms),
    .clk      (clk),
    .rst      (rst),
    .ClockDR  (clock_dr_out),
    .ClockIR  (ClockIR),
    .Enable   (enable),
    .Reset    (Reset),
    .ShiftDR  (shift_dr_out),
    .ShiftIR  (ShiftIR1),
    .UpdateDR (update_dr_out),
    .UpdateIR (UpdateIR)
);

outputDFF U_6(
    .clk        (clk),
    .rst        (Reset),
    .input_data (dout1),
    .to_TDO     (TDO_tap)
);


// ModuleWare code(v1.12) for instance 'U_1' of 'mux'
always @(BSR_in, Bypass_TDO, IDcode_TDO, BSR_in, DR_Decoder)
```

```
begin : u_1combo_proc
   case (DR_Decoder)
      2'd0: mw_U_1temp_dout = BSR_in;
      2'd1: mw_U_1temp_dout = Bypass_TDO;
      2'd2: mw_U_1temp_dout = IDcode_TDO;
      2'd3: mw_U_1temp_dout = BSR_in;
      default: mw_U_1temp_dout = 1'bx;
   endcase
end
assign dout = mw_U_1temp_dout;

// ModuleWare code(v1.12) for instance 'U_5' of 'mux'
always @(dout, instruction_TDO, ShiftIR1)
begin : u_5combo_proc
   case (ShiftIR1)
      1'd0: mw_U_5temp_dout = dout;
      1'd1: mw_U_5temp_dout = instruction_TDO;
      default: mw_U_5temp_dout = 1'bx;
   endcase
end
assign dout1 = mw_U_5temp_dout;

endmodule // Tap_Top
```

B.2 Instruction_Register_Top

```
//
// Module Masterarbeit_lib.Instruction_Register_top.struct
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 18:13:28 10/05/22
//
// Generated by Mentor Graphics' HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module Instruction_Register_top(
   // Port Declarations
   input   wire           TDI,
   input   wire           clk,
   input   wire           clock_IR,
   input   wire           rst,
   input   wire           shift_IR,
   input   wire           update_IR,
   output  wire           instruction_TDO,
   output  wire    [3:0]  to_instruction_Decoder
);


// Internal Declarations


// Local declarations

// Internal signal declarations
```

```
wire  [3:0] instruction;


// Instances
Hold_Register_IR U_1(
    .clk                    (clk),
    .rst                    (rst),
    .Update_IR              (update_IR),
    .Instruction_code       (instruction),
    .to_instruction_Decoder (to_instruction_Decoder)
);

Shift_Register_IR U_0(
    .TDI                (TDI),
    .clk                (clk),
    .rst                (rst),
    .Shift_IR           (shift_IR),
    .Clock_IR           (clock_IR),
    .instruction        (instruction),
    .shiftRegister_TDO  (instruction_TDO)
);
endmodule // Instruction_Register_top
```

B.3 Shift_Register_IR

```
//
// Verilog Module Masterarbeit_lib.Shift_Register_IR
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 13:27:30 07/23/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module Shift_Register_IR (
  // Port Declarations
    input   wire      TDI,
    input   wire      clk,
    input   wire      rst,
    input   wire      Shift_IR,
    input   wire      Clock_IR,
    output  reg[3:0]  instruction,
    output  reg       shiftRegister_TDO
    );
always @(
   posedge clk,
   negedge rst
)
begin
  if (!rst) begin
    instruction <=4'b0101;
  end
  else
  begin
    if(Clock_IR & Shift_IR)
    begin
      instruction[3:0] <={TDI,instruction[3:1]};
    end
```

```
  end
end
assign  shiftRegister_TDO = instruction[0] ;

// ### Please start your Verilog code here ###
Endmodule
```

B.4 Hold_Register_IR

```
//
// Verilog Module Masterarbeit_lib.Hold_Register_IR
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 14:03:23 07/23/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module Hold_Register_IR(
  // Port Declarations
   input   wire      clk,
   input   wire      rst,
   input   wire      Update_IR,
   input   reg [3:0] Instruction_code,
   output  reg [3:0] to_instruction_Decoder

   );

reg [3:0] Hold_instruction ;
always @(
   negedge clk,
   negedge rst
)
begin
  if (!rst) begin
    Hold_instruction <= 1'b0;

  end
  else
  begin
    if(Update_IR)
    begin

      to_instruction_Decoder <= Instruction_code;
      Hold_instruction <= Instruction_code;
    end


  end
end

endmodule
```

B.5 Shiftregister_dr_top

```
//
// Module Masterarbeit_lib.ShiftRegister_DR_Top.struct
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 14:33:23 12/24/22
//
// Generated by Mentor Graphics' HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module ShiftRegister_DR_Top(
   // Port Declarations
   input   wire      Clock_DR,
   input   wire      Shift_DR,
   input   wire      TDI,
   input   wire      clk,
   input   wire      rst,
   output  wire      Bypass_TDO,
   output  wire      IDcode_TDO
);


// Internal Declarations


// Local declarations

// Internal signal declarations


// Instances
Bypass_Register_shift U_2(
   .TDI        (TDI),
   .clk        (clk),
   .rst        (rst),
   .Shift_DR   (Shift_DR),
   .Clock_DR   (Clock_DR),
   .Bypass_TDO (Bypass_TDO)
);

IDCODE_Register_Shift U_1(
   .TDI        (TDI),
   .clk        (clk),
   .rst        (rst),
   .Shift_DR   (Shift_DR),
   .Clock_DR   (Clock_DR),
   .IDCODE_TDO (IDcode_TDO)
);


endmodule // ShiftRegister_DR_Top
```

B.6 Bypass_Register_shift

```
//
// Verilog Module Masterarbeit_lib.Bypass_Register_shift
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 12:21:17 08/13/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module Bypass_Register_shift (
  // Port Declarations
   input   wire      TDI,
   input   wire      clk,
   input   wire      rst,
   input   wire      Shift_DR,
   input   wire      Clock_DR,
   output  reg       Bypass_TDO
   );

// ### Please start your Verilog code here ###
reg bypassIntern;

always @(
   posedge clk,
   negedge rst
)
begin
  if (!rst) begin
    bypassIntern <= 1'b0;
  end
  else
  begin
    if(Clock_DR & Shift_DR)
    begin
       bypassIntern <= TDI;
    end
  end
end

assign  Bypass_TDO = bypassIntern ;

endmodule
```

B.7 IDCODE_Register_Shift

```
//
// Verilog Module Masterarbeit_lib.IDCODE_Register_Shift
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 12:54:41 08/13/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module IDCODE_Register_Shift (
  // Port Declarations
   input   wire      TDI,
   input   wire      clk,
   input   wire      rst,
   input   wire      Shift_DR,
   input   wire      Clock_DR,
   output  reg       IDCODE_TDO
   );


// ### Please start your Verilog code here ###

reg[7:0]  IDCODE;

always @(
   posedge clk,
   negedge rst
)
begin
  if (!rst) begin
    IDCODE <= 8'b11110000;

  end
  else
  begin
    if(Clock_DR & Shift_DR)
    begin
      IDCODE[7:0] <= {TDI,IDCODE[7:1]};
    end
  end
end

assign  IDCODE_TDO = IDCODE[0] ;

endmodule
```

B.8 IR_Decoder

```
//
// Verilog Module Masterarbeit_lib.IR_Decoder
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 17:07:08 08/04/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps

module IR_Decoder (
  // Port Declarations
   input   reg[3:0]  IR_Code,
   output  reg [1:0] DR_Decoder
   );


reg [1:0] current_Register;
reg TempOut;
parameter
          Extest_Enable= 4'b0000,
          Bypass_Enable = 4'b1111,
          Sample_Preload = 4'b0001,
          IDCODE_Enable = 4'b0010;


always@(IR_Code)
begin

  case(IR_Code)
    Extest_Enable : current_Register =2'b00;
    Bypass_Enable : current_Register =2'b01;
    IDCODE_Enable : current_Register = 2'b10;
    Sample_Preload : current_Register = 2'b11;
    default: current_Register = 1'bx; //Bypass
  endcase
end

assign DR_Decoder = current_Register;


// ### Please start your Verilog code here ###

endmodule
```

B.9 Mode_generate

```
//
// Verilog Module Masterarbeit_lib.Mode_generate
//
// Created:
```

```
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 13:08:22 09/04/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module Mode_generate (
  // Port Declarations

   input  reg [1:0] DR_Decoder,
   output  reg  mode

   );

reg TempOut;
// ### Please start your Verilog code here ###
parameter
          Extest_Enable= 4'b00,
          sample_Enable   = 4'b11;
          //preload_Enable = 4'b10;

always@(DR_Decoder)
begin

  case(DR_Decoder)
    Extest_Enable : TempOut =1'b1;
    sample_Enable : TempOut =1'b0;
    default: TempOut = 1'bx; //Bypass
  endcase
end

assign mode = TempOut;

endmodule
```

B.10 outputDFF

```
//
// Verilog Module Masterarbeit_lib.outputDFF
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 19:40:41 08/22/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module outputDFF (
  // Port Declarations
   input   wire      clk,
```

```
   input   wire      rst,
   input   wire input_data,
   output  wire to_TDO

   );


// ### Please start your Verilog code here ###
reg  Data;

assign  to_TDO = Data ;

always @(
   negedge clk,
   negedge rst
)
begin
  if (!rst) begin
    Data = 0;
  end
  else
  begin
     Data= input_data;
  end
end

endmodule
```

B.11 tapfsm_fsm (Zustandsmaschine)

```
//
// Module Masterarbeit_lib.TapFSM.fsm
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 14:33:20 12/24/22
//
// Generated by Mentor Graphics' HDL Designer(TM) 2019.4 (Build 4)
//
`resetall
`timescale 1ns/10ps
module TapFSM(
   // Port Declarations
   output  reg       ClockDR,
   output  reg       ClockIR,
   output  reg       Enable,
   output  reg       Reset,
   output  reg       ShiftDR,
   output  reg       ShiftIR,
   input   wire      TMS,
   output  reg       UpdateDR,
   output  reg       UpdateIR,
   input   wire      clk,
   input   wire      rst
);


// Internal Declarations
```

```
// Declare any pre-registered internal signals
reg ClockDR_int;
reg ClockIR_int;
reg Enable_int;
reg Reset_int;
reg ShiftDR_int;
reg ShiftIR_int;
reg UpdateDR_int;
reg UpdateIR_int;

// Module Declarations

// State encoding
parameter
          test_Logic_Reset = 4'd0,
          run_test_Idle    = 4'd1,
          select_DR_scan   = 4'd2,
          capture_DR       = 4'd3,
          Shift_DR         = 4'd4,
          exit1_DR         = 4'd5,
          PauseDR          = 4'd6,
          exit2_Dr         = 4'd7,
          update_Dr        = 4'd8,
          select_IR_scan   = 4'd9,
          capture_IR       = 4'd10,
          shift_IR         = 4'd11,
          exit1_IR         = 4'd12,
          pause_IR         = 4'd13,
          exit2_IR         = 4'd14,
          update_IR        = 4'd15;

reg [3:0] current_state, next_state;

//------------------------------------------------------------------
// Next State Block for machine csm
//------------------------------------------------------------------
always @(
   TMS,
   current_state
)
begin : next_state_block_proc
   case (current_state)
      test_Logic_Reset: begin
         if (TMS == 0)
            next_state = run_test_Idle;
         else if (TMS == 1)
            next_state = test_Logic_Reset;
         else
            next_state = test_Logic_Reset;
      end
      run_test_Idle: begin
         if (TMS == 1)
            next_state = select_DR_scan;
         else if (TMS == 0)
            next_state = run_test_Idle;
         else
            next_state = run_test_Idle;
      end
      select_DR_scan: begin
```

```
   if (TMS == 0)
      next_state = capture_DR;
   else if (TMS == 1)
      next_state = select_IR_scan;
   else
      next_state = select_DR_scan;
end
capture_DR: begin
   if (TMS == 0)
      next_state = Shift_DR;
   else if (TMS == 1)
      next_state = exit1_DR;
   else
      next_state = capture_DR;
end
Shift_DR: begin
   if (TMS == 1)
      next_state = exit1_DR;
   else if (TMS == 0)
      next_state = Shift_DR;
   else
      next_state = Shift_DR;
end
exit1_DR: begin
   if (TMS == 0)
      next_state = PauseDR;
   else if (TMS == 1)
      next_state = update_Dr;
   else
      next_state = exit1_DR;
end
PauseDR: begin
   if (TMS == 1)
      next_state = exit2_Dr;
   else if (TMS == 0)
      next_state = PauseDR;
   else
      next_state = PauseDR;
end
exit2_Dr: begin
   if (TMS == 1)
      next_state = update_Dr;
   else if (TMS == 0)
      next_state = Shift_DR;
   else
      next_state = exit2_Dr;
end
update_Dr: begin
   if (TMS == 1)
      next_state = select_DR_scan;
   else if (TMS == 0)
      next_state = run_test_Idle;
   else
      next_state = update_Dr;
end
select_IR_scan: begin
   if (TMS == 0)
      next_state = capture_IR;
   else if (TMS == 1)
      next_state = test_Logic_Reset;
   else
```

```
            next_state = select_IR_scan;
      end
      capture_IR: begin
         if (TMS == 0)
            next_state = shift_IR;
         else if (TMS == 1)
            next_state = exit1_IR;
         else
            next_state = capture_IR;
      end
      shift_IR: begin
         if (TMS == 1)
            next_state = exit1_IR;
         else if (TMS == 0)
            next_state = shift_IR;
         else
            next_state = shift_IR;
      end
      exit1_IR: begin
         if (TMS == 0)
            next_state = pause_IR;
         else if (TMS == 1)
            next_state = update_IR;
         else
            next_state = exit1_IR;
      end
      pause_IR: begin
         if (TMS == 1)
            next_state = exit2_IR;
         else if (TMS == 0)
            next_state = pause_IR;
         else
            next_state = pause_IR;
      end
      exit2_IR: begin
         if (TMS == 1)
            next_state = update_IR;
         else if (TMS == 0)
            next_state = shift_IR;
         else
            next_state = exit2_IR;
      end
      update_IR: begin
         if (TMS == 1)
            next_state = select_DR_scan;
         else if (TMS == 0)
            next_state = run_test_Idle;
         else
            next_state = update_IR;
      end
      default:
         next_state = test_Logic_Reset;
   endcase
end // Next State Block

//-----------------------------------------------------------------
// Output Block for machine csm
//-----------------------------------------------------------------
always @(
   current_state
)
```

```
begin : output_block_proc
   // Default Assignment
   ClockDR_int = 0;
   ClockIR_int = 0;
   Enable_int = 0;
   Reset_int = 1;
   ShiftDR_int = 0;
   ShiftIR_int = 0;
   UpdateDR_int = 0;
   UpdateIR_int = 0;

   // Combined Actions
   case (current_state)
      test_Logic_Reset: begin
         Reset_int = 0 ;
      end
      capture_DR: begin
         ClockDR_int = 1 ;
      end
      Shift_DR: begin
         ShiftDR_int = 1 ;
         Enable_int = 1;
         ClockDR_int = 1 ;
      end
      update_Dr: begin
         UpdateDR_int = 1 ;
      end
      shift_IR: begin
         ClockIR_int = 1 ;
         ShiftIR_int = 1 ;
         Enable_int = 1;
      end
      update_IR: begin
         UpdateIR_int = 1 ;
      end
   endcase
end // Output Block

//---------------------------------------------------------------------
// Clocked Block for machine csm
//---------------------------------------------------------------------
always @(
   posedge clk,
   negedge rst
)
begin : clocked_block_proc
   if (!rst) begin
      current_state <= test_Logic_Reset;
      // Reset Values
      ClockDR <= 0;
      ClockIR <= 0;
      Enable <= 0;
      Reset <= 1;
      ShiftDR <= 0;
      ShiftIR <= 0;
      UpdateDR <= 0;
      UpdateIR <= 0;
   end
   else
   begin
      current_state <= next_state;
```

```
      // Registered output assignments
      ClockDR <= ClockDR_int;
      ClockIR <= ClockIR_int;
      Enable <= Enable_int;
      Reset <= Reset_int;
      ShiftDR <= ShiftDR_int;
      ShiftIR <= ShiftIR_int;
      UpdateDR <= UpdateDR_int;
      UpdateIR <= UpdateIR_int;
   end
end // Clocked Block

endmodule // TapFSM
```

B.12 boundaryscan_top_struct

```
//
// Module Masterarbeit_lib.BoundaryScan_Top.struct
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 15:21:49 09/18/22
//
// Generated by Mentor Graphics' HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module BoundaryScan_Top(
   // Port Declarations
   input   wire      Clock_DR,
   input   wire      Data_Input_pin0,
   input   wire      Data_Input_pin1,
   input   wire      Mode,
   input   wire      Scan_In,
   input   wire      Shift_DR,
   input   wire      Update_DR,
   input   wire      clk,
   input   wire      rst,
   output  wire      Data_Output_cell2,
   output  wire      Data_Output_cell3,
   output  wire      Scan_Out
);


// Internal Declarations


// Local declarations

// Internal signal declarations
wire   scan_output;
wire   scan_output1;
wire   scan_output2;


// Instances
BS_CELL U_0(
```

```
    .Data_input   (Data_Input_pin0),
    .Mode         (Mode),
    .Shift_dr     (Shift_DR),
    .Update_dr    (Update_DR),
    .clk          (clk),
    .clock_dr     (Clock_DR),
    .rst          (rst),
    .scan_input   (Scan_In),
    .data_output  (),
    .scan_output  (scan_output)
);

BS_CELL U_1(
    .Data_input   (Data_Input_pin1),
    .Mode         (Mode),
    .Shift_dr     (Shift_DR),
    .Update_dr    (Update_DR),
    .clk          (clk),
    .clock_dr     (Clock_DR),
    .rst          (rst),
    .scan_input   (scan_output),
    .data_output  (),
    .scan_output  (scan_output1)
);

BS_CELL U_2(
    .Data_input   (),
    .Mode         (Mode),
    .Shift_dr     (Shift_DR),
    .Update_dr    (Update_DR),
    .clk          (clk),
    .clock_dr     (Clock_DR),
    .rst          (rst),
    .scan_input   (scan_output1),
    .data_output  (Data_Output_cell2),
    .scan_output  (scan_output2)
);

BS_CELL U_3(
    .Data_input   (),
    .Mode         (Mode),
    .Shift_dr     (Shift_DR),
    .Update_dr    (Update_DR),
    .clk          (clk),
    .clock_dr     (Clock_DR),
    .rst          (rst),
    .scan_input   (scan_output2),
    .data_output  (Data_Output_cell3),
    .scan_output  (Scan_Out)
);


endmodule // BoundaryScan_Top
```

B.13 bs_cell_struct

```
//
// Module Masterarbeit_lib.BS_CELL.struct
//
// Created:
```

```
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 18:16:40 10/05/22
//
// Generated by Mentor Graphics' HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module BS_CELL(
   // Port Declarations
   input   wire      Data_input,
   input   wire      Mode,
   input   wire      Shift_dr,
   input   wire      Update_dr,
   input   wire      clk,
   input   wire      clock_dr,
   input   wire      rst,
   input   wire      scan_input,
   output  wire      data_output,
   output  wire      scan_output
);


// Internal Declarations


// Local declarations

// Internal signal declarations
wire   dout;
wire   to_Data_out;

// ModuleWare signal declarations(v1.12) for instance 'U_0' of 'mux'
reg mw_U_0temp_dout;

// ModuleWare signal declarations(v1.12) for instance 'U_3' of 'mux'
reg mw_U_3temp_dout;


// Instances
BSC_FF1 U_1(
   .clk       (clk),
   .rst       (rst),
   .Clock_DR  (clock_dr),
   .Data_in   (dout),
   .to_output (scan_output)
);

BSC_FF2 U_2(
   .clk         (clk),
   .rst         (rst),
   .Update_DR   (Update_dr),
   .Data_in     (scan_output),
   .to_Data_out (to_Data_out)
);


// ModuleWare code(v1.12) for instance 'U_0' of 'mux'
always @(Data_input, scan_input, Shift_dr)
begin : u_0combo_proc
   case (Shift_dr)
```

```
      1'd0: mw_U_0temp_dout = Data_input;
      1'd1: mw_U_0temp_dout = scan_input;
      default: mw_U_0temp_dout = 1'bx;
   endcase
end
assign dout = mw_U_0temp_dout;

// ModuleWare code(v1.12) for instance 'U_3' of 'mux'
always @(Data_input, to_Data_out, Mode)
begin : u_3combo_proc
   case (Mode)
      1'd0: mw_U_3temp_dout = Data_input;
      1'd1: mw_U_3temp_dout = to_Data_out;
      default: mw_U_3temp_dout = 1'bx;
   endcase
end
assign data_output = mw_U_3temp_dout;

endmodule // BS_CELL
```

B.14 BSC_FF1

```
//
// Verilog Module Masterarbeit_lib.BSC_FF1
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 16:51:20 08/19/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module BSC_FF1 (
  // Port Declarations
   input   wire      clk,
   input   wire      rst,
   input   wire      Clock_DR,
   input   wire      Data_in,
   output  reg       to_output
   );

reg data;
// ### Please start your Verilog code here ###


always @(
   posedge clk,
   negedge rst
)
begin
  if (!rst) begin

  end
  else
  begin
    if(Clock_DR)
    begin
```

```
      data <=Data_in;

    end
  end
end

assign to_output =data;

endmodule
```

B.15 BSC_FF2

```
//
// Verilog Module Masterarbeit_lib.BSC_FF2
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 12:01:13 08/21/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module BSC_FF2(
  // Port Declarations
   input   wire      clk,
   input   wire      rst,
   input   wire      Update_DR,
   input   wire      Data_in,
   output  reg       to_Data_out
   );


// ### Please start your Verilog code here ###

always @(
   negedge clk,
   negedge rst
)
begin
  if (!rst) begin

  end
  else
  begin
    if(Update_DR)
    begin
      to_Data_out <= Data_in ;

    end
  end
end


endmodule
```

B.16 extestboundaryscan_struct

```
//
// Module Masterarbeit_lib.ExtestBoundaryScan.struct
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 15:03:02 12/26/22
//
// Generated by Mentor Graphics' HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module ExtestBoundaryScan(
   // Port Declarations
   input   wire      TDI,
   input   wire      TMS,
   input   wire      clk,
   input   wire      pin0_chip1,
   input   wire      pin0_chip2,
   input   wire      pin1_chip1,
   input   wire      pin1_chip2,
   input   wire      rst,
   output  wire      scan_out
);


// Internal Declarations


// Local declarations

// Internal signal declarations
wire   Data_Output_cell2;
wire   Data_Output_cell3;
wire   Scan_Out;
wire   Scan_Out1;
wire   TDO_tap;
wire   TDO_tap1;
wire   clock_dr_out;
wire   clock_dr_out1;
wire   dout;
wire   dout1;
wire   dout2;
wire   enable;
wire   enable1;
wire   mode;
wire   mode1;
wire   shift_dr_out;
wire   shift_dr_out1;
wire   update_dr_out;
wire   update_dr_out1;

// ModuleWare signal declarations(v1.12) for instance 'U_4' of 'mux'
reg mw_U_4temp_dout;

// ModuleWare signal declarations(v1.12) for instance 'U_5' of 'mux'
reg mw_U_5temp_dout;
```

```
// Instances
BoundaryScan_Top U_0(
    .Clock_DR           (clock_dr_out),
    .Data_Input_pin0    (pin1_chip1),
    .Data_Input_pin1    (pin0_chip1),
    .Mode               (mode),
    .Scan_In            (TDI),
    .Shift_DR           (shift_dr_out),
    .Update_DR          (update_dr_out),
    .clk                (clk),
    .rst                (rst),
    .Data_Output_cell2  (Data_Output_cell2),
    .Data_Output_cell3  (Data_Output_cell3),
    .Scan_Out           (Scan_Out)
);

BoundaryScan_Top U_1(
    .Clock_DR           (clock_dr_out1),
    .Data_Input_pin0    (dout2),
    .Data_Input_pin1    (dout1),
    .Mode               (mode1),
    .Scan_In            (dout),
    .Shift_DR           (shift_dr_out1),
    .Update_DR          (update_dr_out1),
    .clk                (clk),
    .rst                (rst),
    .Data_Output_cell2  (),
    .Data_Output_cell3  (),
    .Scan_Out           (Scan_Out1)
);

Tap_Top U_2(
    .BSR_in         (Scan_Out),
    .TDI            (TDI),
    .Tms            (TMS),
    .clk            (clk),
    .rst            (rst),
    .TDO_tap        (TDO_tap),
    .clock_dr_out   (clock_dr_out),
    .enable         (enable),
    .mode           (mode),
    .shift_dr_out   (shift_dr_out),
    .update_dr_out  (update_dr_out)
);

Tap_Top U_3(
    .BSR_in         (Scan_Out1),
    .TDI            (dout),
    .Tms            (TMS),
    .clk            (clk),
    .rst            (rst),
    .TDO_tap        (TDO_tap1),
    .clock_dr_out   (clock_dr_out1),
    .enable         (enable1),
    .mode           (mode1),
    .shift_dr_out   (shift_dr_out1),
    .update_dr_out  (update_dr_out1)
);


// ModuleWare code(v1.12) for instance 'U_4' of 'mux'
```

```
always @(pin1_chip2, Data_Output_cell2, mode1)
begin : u_4combo_proc
   case (mode1)
      1'd0: mw_U_4temp_dout = pin1_chip2;
      1'd1: mw_U_4temp_dout = Data_Output_cell2;
      default: mw_U_4temp_dout = 1'bx;
   endcase
end
assign dout1 = mw_U_4temp_dout;

// ModuleWare code(v1.12) for instance 'U_5' of 'mux'
always @(pin0_chip2, Data_Output_cell3, mode1)
begin : u_5combo_proc
   case (mode1)
      1'd0: mw_U_5temp_dout = pin0_chip2;
      1'd1: mw_U_5temp_dout = Data_Output_cell3;
      default: mw_U_5temp_dout = 1'bx;
   endcase
end
assign dout2 = mw_U_5temp_dout;

// ModuleWare code(v1.12) for instance 'U_7' of 'tribuf'
assign dout = (enable) ? TDO_tap : 1'bz;

// ModuleWare code(v1.12) for instance 'U_8' of 'tribuf'
assign scan_out = (enable1) ? TDO_tap1 : 1'bz;

endmodule // ExtestBoundaryScan
```

B.17 BSR_Testbench

```
//
// Verilog Module Masterarbeit_lib.BSR_Testbench
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 16:15:00 09/09/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//
`resetall
`timescale 1ns/10ps
module BSR_Testbench ;


// ### Please start your Verilog code here ###


reg sys_clk;
reg arst;
reg tdi;
reg tdo;
reg tms;
reg  pin_10 = 1;
reg  pin_11 = 1;
reg  pin_20 = 0;
reg  pin_21 = 1;
reg [7:0]myout;
reg start_en =0 ;
```

```
ExtestBoundaryScan unitundertest1(

.TDI(tdi),
.TMS(tms),
.clk(sys_clk),
.rst(arst),
.pin0_chip1(pin_10),
.pin1_chip1(pin_11),
.pin0_chip2(pin_20),
.pin1_chip2(pin_21),
.scan_out(tdo)

);

initial
begin
  sys_clk = 0;
  arst = 0;
  myout[7:0]=0;
  @(posedge sys_clk) arst =1;
  tms = 1;
  #2
  //Reset 5 TMS = 1
  @(negedge sys_clk) tms=1;tdi=1'bx;
  @(negedge sys_clk) tms=1;tdi=1'bx;
  @(negedge sys_clk) tms=1;tdi=1'bx;
  @(negedge sys_clk) tms=1;tdi=1'bx;
  @(negedge sys_clk) tms=1;tdi=1'bx;
  //to IR
  @(negedge sys_clk) tms=0;tdi=1'bx; //run test idle
  @(negedge sys_clk) tms=1;tdi=1'bx; //select dr scan
  @(negedge sys_clk) tms=1;tdi=1'bx; //select IRscan
  @(negedge sys_clk) tms=0;tdi=1'bx;  //capture IR
  @(negedge sys_clk) tms=0;tdi=1'bx;  //shift IR
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=1;   //IR
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=1;tdi=0;              //exit IR
  @(negedge sys_clk) tms=1;tdi=0;                   //Update IR

  //DR bsr ext1
  @(negedge sys_clk) tms=1;                    //select DR
  @(negedge sys_clk) tms=0;                    //capture DR
  @(negedge sys_clk) tms=0;                    //Shift DR
  @(negedge sys_clk) tms=0;
  @(negedge sys_clk) tms=0;tdi=1;start_en=1;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=1;tdi=0;       //exit dr
  @(negedge sys_clk) tms=1;tdi=0;       //update dr
  @(negedge sys_clk) tms=0;tdi=0;start_en=0; //for Sample preload
//
```

```
//  //DR bsr ext2
//  @(negedge sys_clk) tms=1;              //select dr
//  @(negedge sys_clk) tms=0;               //capture DR
//  @(negedge sys_clk) tms=0;                //Shift DR
//  @(negedge sys_clk) tms=0;
//  @(negedge sys_clk) tms=0;tdi=1;start_en=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=1;tdi=1;        //exit dr
//  @(negedge sys_clk) tms=1;              //update dr
//  @(negedge sys_clk) tms=0;start_en=0;

end

always sys_clk = #2 !sys_clk;

always @(posedge sys_clk)
begin
  if(start_en)
     // myout[15:0] <= {tdo,myout[15:1]};
     myout[7:0] <= {tdo, myout[7:1]};
end
endmodule
```

B.18 testbench_1

```
//
// Verilog Module Masterarbeit_lib.testbench_1
//
// Created:
//          by - adrissi.adrissi (kilby.telcom.fh-dortmund.de)
//          at - 11:50:43 08/14/22
//
// using Mentor Graphics HDL Designer(TM) 2019.4 (Build 4)
//

`resetall
`timescale 1ns/10ps
module testbench_1 ;

// ### Please start your Verilog code here ###

reg sys_clk;
reg arst;
reg tdi;
reg tdo;
reg tms;
reg [8:0]myout;
reg start_en =0 ;
reg counter1=0;
reg tdo_chip2;
reg ena;

reg TDO_Chip1;
reg TDO_final;
reg shift_dr;
reg clock_dr;
reg update_dr;
```

```
reg Mode;

Tap_Top uuu1(

.TDI(tdi),
.Tms(tms),
.clk(sys_clk),
.rst(arst),
.enable(ena),
.shift_dr_out(shift_dr),
.clock_dr_out(clock_dr),
.update_dr_out(update_dr),
.mode(Mode),
.TDO_tap(tdo)

);

Tap_Top uuu2(

.TDI(TDO_Chip1),
.Tms(tms),
.clk(sys_clk),
.rst(arst),
.enable(ena),
.shift_dr_out(shift_dr),
.clock_dr_out(clock_dr),
.update_dr_out(update_dr),
.mode(Mode),
.TDO_tap(tdo_chip2)

);

assign TDO_Chip1 = (ena)? tdo : 1'bz;
assign TDO_final = (ena)? tdo_chip2 : 1'bz;

initial
begin
  sys_clk = 0;
  arst = 0;
  myout[8:0]=0;
  @(posedge sys_clk) arst =1;
  tms = 1;
  #2
  //Reset 5 TMS = 1
  @(negedge sys_clk) tms=1;tdi=1'bZ;
  @(negedge sys_clk) tms=1;tdi=1'bZ;
  @(negedge sys_clk) tms=1;tdi=1'bZ;
  @(negedge sys_clk) tms=1;tdi=1'bZ;
  @(negedge sys_clk) tms=1;tdi=1'bZ;
  //to IR
  @(negedge sys_clk) tms=0;tdi=1'bZ;
  @(negedge sys_clk) tms=1;tdi=1'bZ;
  @(negedge sys_clk) tms=1;tdi=1'bZ;
  @(negedge sys_clk) tms=0;tdi=1'bZ;
  @(negedge sys_clk) tms=0;tdi=1'bZ;
  @(negedge sys_clk) tms=0;tdi=1'bZ;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=0;
  @(negedge sys_clk) tms=0;tdi=1;
```

```
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=1;tdi=1;
  @(negedge sys_clk) tms=1;tdi=1;
  @(negedge sys_clk) tms=1;
  @(negedge sys_clk) tms=0;
  @(negedge sys_clk) tms=0;
  @(negedge sys_clk) tms=0;
  @(negedge sys_clk) tms=0;tdi=1;start_en=1;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=0;tdi=1;
 // @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
//  @(negedge sys_clk) tms=0;tdi=1;
  @(negedge sys_clk) tms=1;tdi=1;
  @(negedge sys_clk) tms=1;
  @(negedge sys_clk) tms=0;start_en=0;

end

always sys_clk = #2 !sys_clk;

always @(posedge sys_clk)
begin
  if(start_en)
      myout[8:0] <= {tdo_chip2,myout[8:1]};
end

endmodule
```