Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Microelectronics
Study major: Electronics and Communication

I3C Controller Design for RISC-V Processor

MASTER'S THESIS

Author: Bc. Tomáš Bánok
Supervisor: prof. Ing. Jiří Jakovenko, Ph.D.
Year: 2024

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Bánok Tomáš** Personal ID number: **491843**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Microelectronics**

Study program: **Electronics and Communications**

Specialisation: **Electronics**

II. Master's thesis details

Master's thesis title in English:

I3C Controller Design for RISC-V Processor

Master's thesis title in Czech:

Návrh adi e I3C pro procesor RISC-V

Guidelines:

1. Get familiar with the concept of data transfer over a serial interface using I2C and I3C protocols.
2. Design an I3C bus controller IP block on a system level as a peripheral for a RISC-V processor; implement the protocol according to a freely available interface specification.
3. Implement the block in VHDL on the RTL level; use the chosen FPGA platform for the physical implementation.
4. Verify the correct operation of the block using digital simulation.
5. Discuss the results and compare the complexity of implementing the controller for I2C and I3C interfaces.

Bibliography / sources:

[1] Š ASTNÝ Jakub. FPGA Prakticky, BEN Prague 2010
[2] MIPI I3C Basic v1.1.1 - bus specification
[3] NXP Semiconductors. I2C Bus specification and user manual, [online]. Rev. v.7 (1.10.2021) Available at: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

Name and workplace of master's thesis supervisor:

prof. Ing. Ji í Jakovenko, Ph.D. Department of Microelectronics FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **13.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

prof. Ing. Ji í Jakovenko, Ph.D. Supervisor's signature	prof. Ing. Pavel Hazdra, CSc. Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Declaration

I hereby declare that I have written the submitted thesis independently and all used resources have been cited in bibliography.

Prague, ...

Bc. Tomáš Bánok

Acknowledgement

I would like to thank my supervisor prof. Ing. Jiří Jakovenko, Ph.D. for valuable advice in the development of this thesis. I also wish to thank my ASICentrum s.r.o. colleagues for their patience, guidance and support, in particular Ing. Jakub Šťastný, Ph.D. Lastly, I would like to thank my family for their support during my studies.

Bc. Tomáš Bánok

Title:
I3C Controller Design for RISC-V Processor

Author: Bc. Tomáš Bánok

Study major: Electronics and Communication
Type: Master's thesis

Supervisor: prof. Ing. Jiří Jakovenko, Ph.D.

Abstract: This thesis deals with the RTL design and implementation of an I3C Controller peripheral for a system with a RISC-V processor core. The thesis describes the I3C protocol with its main features, including backward compatibility with its predecessor, I2C, from a freely available specification. From the specification, the supported features of the protocol have been selected and a system peripheral design has been written. The individual blocks of the peripheral were implemented in VHDL and tested with the RISC-V system. To verify the communication, an I3C Target Agent was created, acting as a Target device connected to the I3C bus. For timing verification, the Controller was synthesized and implemented for the FPGA. The generated netlist was used for gate level simulation of the peripheral.

Keywords: VHDL, I3C, Controller, SDR-only, RISC-V, AHB, FPGA

Contents

List of Acronyms **xi**

List of Figures **xiii**

List of Tables **xiv**

1 Introduction **1**

2 I3C Protocol **3**
2.1 Introduction . . . 3
2.2 Start, Stop, and other patterns . . . 4
2.3 Address header . . . 6
2.4 Private transactions . . . 7
2.5 SDA handoffs . . . 7
2.6 CCC bytes . . . 9
2.6.1 Dynamic Addressing . . . 9
2.6.2 In-band Interrupts, Hot-Joins, Resets . . . 11
2.7 Backwards Compatibility . . . 11
2.8 Power Efficiency . . . 12

3 System Level Design **15**
3.1 System design . . . 15
3.2 SW/HW decomposition . . . 16
3.3 Resets . . . 17
3.4 Clocks . . . 17
3.5 I3C Controller interface . . . 18
3.6 Differentiating Addresses from Data . . . 18
3.7 Supported Command Codes . . . 19
3.8 Error recovery . . . 21

4 RTL Design **23**
4.1 Clock gating, resets . . . 23
4.2 Counters . . . 24
4.3 Register map wrapper . . . 24
4.4 FIFO memories . . . 25
4.5 FIFO Address registers . . . 25
4.6 Interrupt controller . . . 26
4.7 I3C Controller registers . . . 27
4.8 Control block . . . 40
4.9 Shift register . . . 42
4.10 IBI shift register . . . 42
4.11 SCL clock generation . . . 43
4.11.1 Transmission Control . . . 43
4.11.2 Read Counter . . . 44
4.11.3 Elemental Use Cases . . . 44

5 Verification **47**
5.1 Verification Plan . . . 47
5.2 I3C Target Agent . . . 48
5.3 Use cases . . . 53
5.3.1 Initial configuration . . . 53
5.3.2 Basic I3C SDR write . . . 54
5.3.3 Basic I3C SDR read . . . 54

5.3.4 Legacy I2C write . . . 55
5.3.5 Delayed I3C SDR read/write transaction . . . 55
5.3.6 Broadcast CCC transfer . . . 56
5.3.7 Dynamic Address Assignment . . . 56
5.4 Tests . . . 57
5.5 Coverage . . . 58

6 Implementation **61**
6.1 Physical design parameters . . . 61
6.2 Gate Level Simulation . . . 62

7 Conclusions, Next Steps **63**

Bibliography **65**

Used software **66**

RTL codes **67**

List of Acronyms

ACK Acknowledge

AHB Advanced High-Performance Bus

AMBA Advanced Microcontroller Bus Architecture

ASIC Application Specific Integrated Circuit

BCR Bus Characteristic Register

BT Bulk Transport

CCC Common Command Code

CEx Controller Error x

CPU Central Processing Unit

DAA Dynamic Address Assignment

DCR Device Characteristic Register

DDR Double Data Rate

DFF D Flip-Flop

FF Flip-Flop

FIFO First In First Out

FPGA Field Programmable Gate Aray

FSM Finite State Machine

Fm Fast Mode

GSL Gate Level Simulation

HDR High Data Rate

High-Z High impedance

I2C Inter Integrated Circuit

I3CC I3C Controller

I3C Improved Inter Integraded Circuit

IBI In-Band Interrupt

LSB Least Significant Bit

MIPI Mobile Industry Processor Interface

ML Multi-Lane

MSB Most Significant Bit

NACK Not Acknowledge

PEx Peripheral Error x

PID Provisioned ID

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

RnW Read / non-Write

Rx Receive

SCL Serial Clock Line

SDA Serial Data Line

SDR Standard Data Rate

SPI Serial Peripheral Interface

T-bit Transition Bit

TB Test Bench

TEx Target Error x

LUT Look Up Table

TSL Ternary Symbol Legacy-inclusive-bus

TSP Ter nary Symbol Pure-bus

Tx Transmit

UART Universal Asynchronous Receiver-Transmitter

UVM Universal Verification Methodology

List of Figures

2.1 I3C mixed bus 4
2.2 I3C Start, Stop conditions 5
2.3 I3C Restart pattern 5
2.4 I3C HDR Restart pattern 6
2.5 I3C HDR Exit pattern 6
2.6 Address arbitration, controller wins and receives ACK 6
2.7 Address arbitration, target wins and receives ACK 6
2.8 Address arbitration, target wins and receives NACK 7
2.9 SDA handoff during address ACK when writing 8
2.10 CCC transfers 9
2.11 Dynamic Address Assignment using ENTDAA 10
2.12 RSTDAA CCC format 10
2.13 SETAASA CCC format 10
2.14 SETDASA CCC format 11
2.15 SETNEWDA CCC format 11
2.16 Typical transaction on the I3C bus. 12
2.17 Bitrates for 12.5 MHz clock 12
2.18 Effective energy range per 1 kB (μJ) 13

3.1 Diagram of the I3C Controller peripheral 15

4.1 Diagram of the I3C Controller peripheral 23
4.2 Clock gating cell 23
4.3 Register map wrapper diagram 24
4.4 FIFO memory diagram 25
4.5 FIFO full and empty conditions 26
4.6 I3CC CR1 register 27
4.7 I3CC CR2 register 28
4.8 I3CC stat1 register 28
4.9 I3CC stat2 register 30
4.10 I3CC DAA register 31
4.11 I3CC IBI register 31
4.12 I3CC TxFIFO data register 32
4.13 I3CC TxFIFO ptr register 33
4.14 I3CC RxFIFO data register 34
4.15 I3CC RxFIFO ptr register 34
4.16 I3CC Timing 0 register 35
4.17 I3CC Timing 1 register 36
4.18 I3CC Timing 2 register 37
4.19 I3CC IRQ fl register 37
4.20 I3CC IRQ en register 38
4.21 START condition timings 39
4.22 STOP condition timing 39
4.23 SCL timings 39
4.24 SDA setup timings 39
4.25 Diagram of the Control block 40
4.26 SCL high delay caused by the Control FSM 40
4.27 State diagram of the Control FSM 41

4.28 I3C shift register . . . 42
4.29 State diagram of the SCL Gen FSM . . . 43
4.30 SCL Generator delays . . . 43
4.31 State diagram of the Transmit FSM . . . 44

5.1 Test bench structure . . . 47
5.2 I3C Target Agent main loop . . . 48
5.3 I3C Target Agent Get Address fork . . . 49
5.4 I3C Target Agent Data Read fork . . . 50
5.5 I3C Target Agent Data Write fork . . . 51
5.6 I3C Target Agent Dynamic Address Assignment fork . . . 52
5.7 Code coverage of the peripheral . . . 59
5.8 Code coverage by type of metric . . . 59
5.9 Code coverage of the peripheral with applied waiver . . . 60
5.10 Code coverage by type of metric with applied waiver . . . 60

6.1 Timing report of the generated FPGA netlist . . . 61
6.2 FPGA utilization by the peripheral . . . 61
6.3 Inputs of a LUT in a gate level simulation . . . 62
6.4 Delay inserted into the ahb_hwdata[31:0] signal for gate level simulations . . . 62

List of Tables

2.1 Roles of I3C compatible devices (copied from MIPI [1]). . . . 4

3.1 I3C Controller interface - generics . . . 18
3.2 I3C Controller interface - signals . . . 18
3.3 Supported CCC commands . . . 19
3.4 Unsupported CCC commands . . . 20
3.5 SDR Controller Error Types . . . 21

4.1 Interrupt sources . . . 26

5.1 Relevant delays of the Pad Model . . . 48

Chapter 1

Introduction

The goal of this master's thesis is to design an I3C Controller peripheral for a RISC-V platform. The peripheral is designed to implement the freely available I3C protocol specification [1] released by the MIPI organization.

First, the I3C protocol specification was studied and a system level design was prepared, which acted as a general guide for designing the peripheral. The system level design contains the functionality the Controller was desired to fulfill. After this step, the Register Transfer Level (RTL) design was written in VHDL. An I3C Target Agent was developed in System Verilog to simulate a Target alongside the peripheral to verify its functioning in RTL simulations. The design was gradually verified during its design phase, as each new feature needed to be debugged. The generic use cases of the peripheral were also finalized at this time.

After all of the necessary features were implemented, the I3C Controller was imported into Xilinx Vivado tool for synthesis and implementation on an FPGA to generate a netlist and an accompanying simulation delay file for a Gate Level Simulation (GLS) of the peripheral to prove that the design can be implemented and works as expected.

Chapter 2

I3C Protocol

2.1 Introduction

The Improved Inter Integrated Circuit (I3C) is a serial half-duplex communication protocol, developed by the MIPI Alliance [1]. The I3C protocol is designed to supersede the widely used I2C protocol developed by Philips Semiconductor in the early 1980s, which is abundant in the realm of embedded systems.

The protocol defines a bus made of one Serial Clock Line (SCL) and up to four Serial DAta lines (SDA) which are shared among all of the connected devices. Backward compatibility with I2C is secured by having a common I3C address, which all I2C devices passively NACK and therefore remain idle for the rest of the transmission as well as the ability to shorten the Serial Clock high duration, which will, in turn, make the communication transparent to every I2C device equipped with a spike filter. The I3C specification defines a mandatory Standard Data Rate (SDR) mode, which follows an I2C-like communication by sending data in 9-bit frames and multiple optional High Data Rate (HDR) modes, which transmit data in up to 256-bit long frames.

Main features

- Transmissions up to 12.9 MHz using push-pull outputs,
- interrupts, hot-joining and reset with no additional wires required,
- higher bandwidth while maintaining lower power consumption,
- dynamic addressing of target devices,
- backwards compatibility with I2C devices[1],
- grouping of multiple targets with a common address,
- optional High Data Rate (HDR) messaging.

Devices on the bus fall into one of two categories – either the device is a Controller or a Target. Controllers manage the SCL clock generation, are in control of addressing Targets, and generally control the flow of data on the bus. Targets listen to the communication on the bus and act on the addresses and instructions sent by Controllers. An example of an I3C bus can be seen in Figure 2.1, where multiple devices are connected to the same bus. The I3C devices can additionally be separated into subcategories listed in Table 2.1 based on their role on the bus.

[1] I3C forbids the use of clock stretching and 10bit addresses.

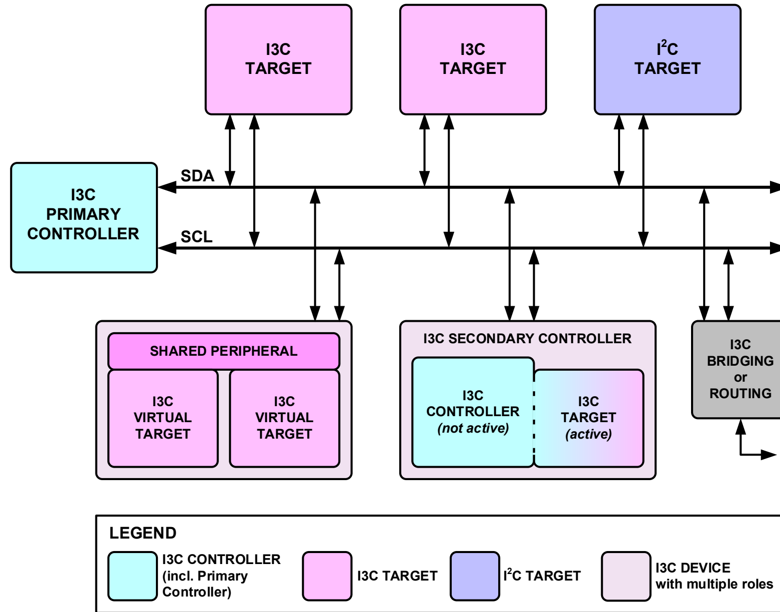

Figure 2.1: Example of an I3C bus in a mixed state (with I2C devices present). (copied from MIPI [1]).

Device Type	Device Role	Description
I3C Controller	I3C Primary Controller	Initially configures I3C Bus, has HDR support
	SDR-Only Primary Controller	Initially configures I3C Bus, no HDR support
	I3C Secondary Controller	Can control the Bus but currently functioning as Target
	SDR-Only Secondary Controller	Can control the Bus but currently functioning as Target, no HDR support
I3C Target	I3C Target	Ordinary I3C Target, no Controller capability
	SDR-Only Target	Ordinary I3C Target, no Controller capability, no HDR support
	I2C Target	No I3C Controller or I3C Target capabilities

Table 2.1: Roles of I3C compatible devices (copied from MIPI [1]).

This paper uses the same address notation as the I3C specification, where addresses are specified as 7 bits in hexadecimal format followed by a *Read* or *Write* indication. Three examples are listed below:

- 7'h7E/W – write to device(s) addressed as 7E (0'b1111110),
- 7'h7E/R – read from device(s) addressed as 7E (0'b1111110),
- 7'h5D/W – write to device(s) addressed as 5D (0b'1011101).

Data uses the usual notation either in binary (0'b1111 1111) or hexadecimal (0xFF) form to differentiate from addresses.

2.2 Start, Stop, and other patterns

The I3C protocol specifies two conditions – Start and Stop. The Start condition signalizes the beginning of communication and is required before any data is transmitted over the bus, while the

Stop pattern indicates the end. Multiple Start patterns can follow after the first one, in which case they are called Repeated Starts.

After the (repeated) Start pattern, an address is sent over the bus specifying which device the following data is intended for. Unlike I2C, the Start pattern can be initiated by a Target device if the bus is idle, generating an In-Band Interrupt.

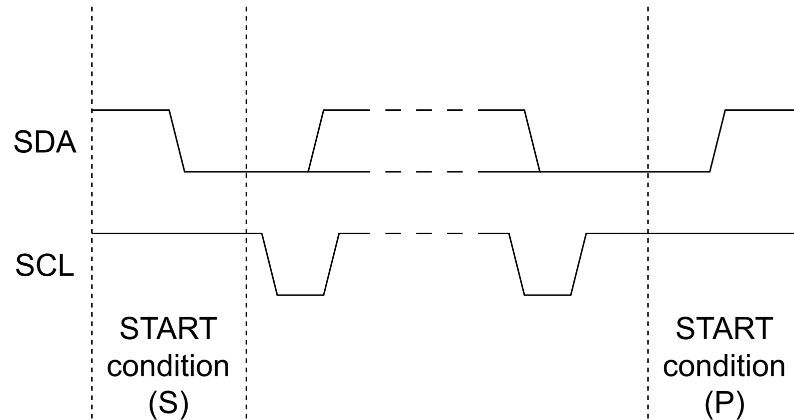

Figure 2.2: The Start and Stop conditions in an I3C transmission.

Additionally, I3C defines three more patterns:

- restart (Figure 2.3),
- HDR Restart (Figure 2.4), and
- HDR Exit (Figure 2.5).

As seen below, all of these patterns are very alike and function similarly. The HDR Restart is similar to a repeated START in SDR mode, as it marks the boundary between two messages in HDR mode while the HDR Exit pattern is similar to a STOP in SDR, exiting the HDR mode and returning to SDR followed by a STOP condition.

The Restart pattern, depending on a configuration of the target, shall either:

a) wake up the target,

b) reset the target's I3C peripheral,

c) reset the whole chip, or

d) take an action configured by the controller[2].

+

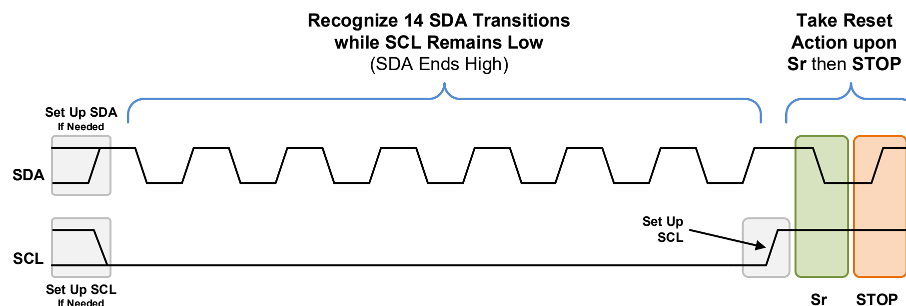

Figure 2.3: I3C Restart pattern (copied from MIPI [1]).

[2]Depends on the RSTACT CCC sent beforehand to the target(s).

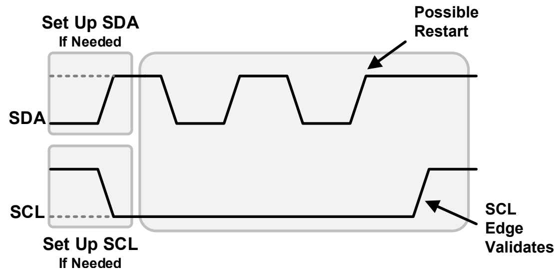

Figure 2.4: I3C HDR Restart pattern (copied from MIPI [1]).

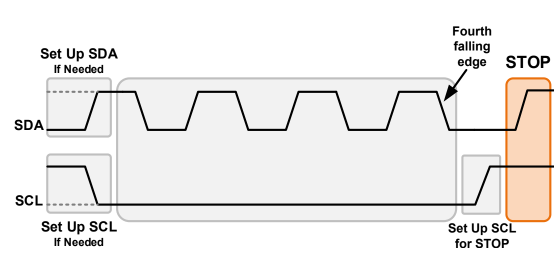

Figure 2.5: I3C HDR Exit pattern (copied from MIPI [1]).

2.3 Address header

I3C address headers follow the same format as in I2C - either a START or a Repeated START, followed by 7 bits of address, 1 bit of RnW, and a (N)ACK bit.

I3C expands on this by having the first address header arbitrated, where all devices connected to the I3C bus can compete over which address is sent onto the bus. Collisions on this address are prevented by using an open drain mode[3], resulting in the lowest address winning.

Depending on which device won the arbitration, the controller can either continue in sending its data (if it won), ACK/NACK an in-band interrupt, and/or send a repeated Start and repeat its transmission.

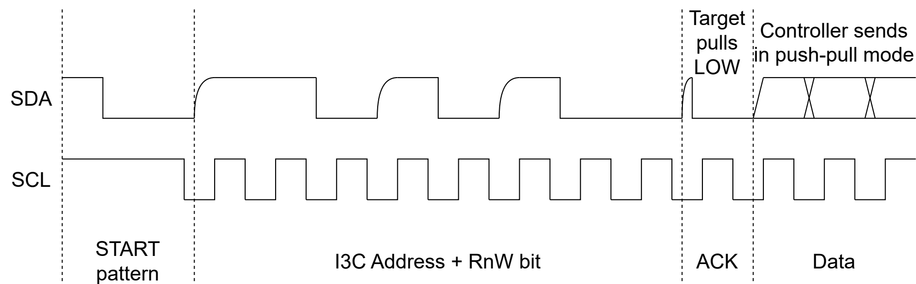

Figure 2.6: Controller sends a target's address, target ACKs and controller starts sending data.

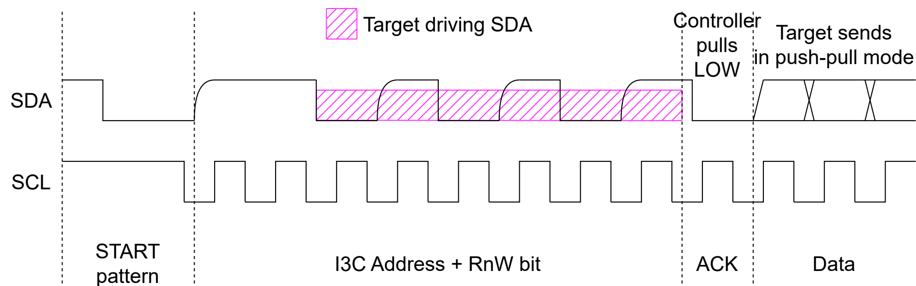

Figure 2.7: Controller attempts to send an address, but a different Target wins the address arbitration. Controller ACKs the In-Band Interrupt and starts receiving data.

[3]The I3C specification also defines the ability to send a part of the address in push-pull if certain conditions are met.

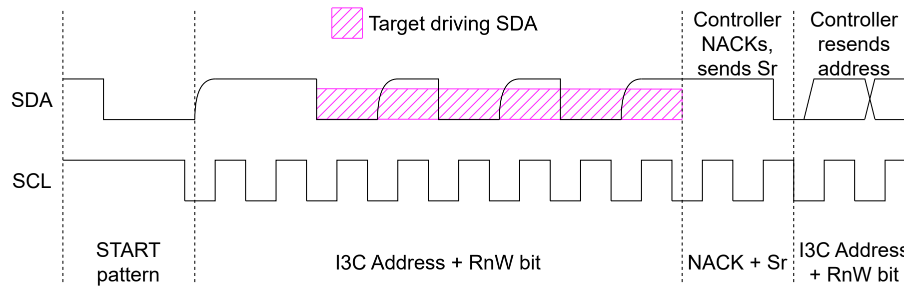

Figure 2.8: Controller attempts to send an address, but a different target wins the address arbitration. Controller NACKs the In-Band Interrupt, sends a repeated START and transmits the address in push-pull mode.

2.4 Private transactions

Once an I3C target device has been assigned its dynamic address, the controller can communicate with the target directly by specifying its address. When a target device has been addressed, the controller can begin writing to or reading from the target depending on the value of the RnW bit of the address header. Similarly to I2C, the data is transmitted in 9-bit frames, consisting of 8 data bits plus a 9th *T-bit*. The T-bit has different functionality depending on whether the target is being read from (RnW=1) or written to (RnW=0). During writes to the target, the T-bit is used as a parity bit to verify whether the received byte is valid, whereas during reads its function is to make sure the target can still manage to send data, and if not, the target can terminate the communication.

2.5 SDA handoffs

Switching between open drain and push-pull outputs creates a problem with handing off control of the SDA line(s) to prevent a collision on the bus. The I3C specification defines where and how a procedure has to be followed to prevent said collisions. Examples of this are:

- transition from address ACK to SDR Controller write data,
- T-bit during SDR read (end-of-data bit),
- preamble in HDR-DDR mode and more.

Generally speaking, these procedures follow one of two ways of ensuring collision free handoffs of the bus, either:

a) two devices which activities are overlapping each other by both actively driving SDA low for a short time (Figure 2.9), or

b) one device driving SDA high and switching to high-Z on the next clock edge, allowing the other device to respond.

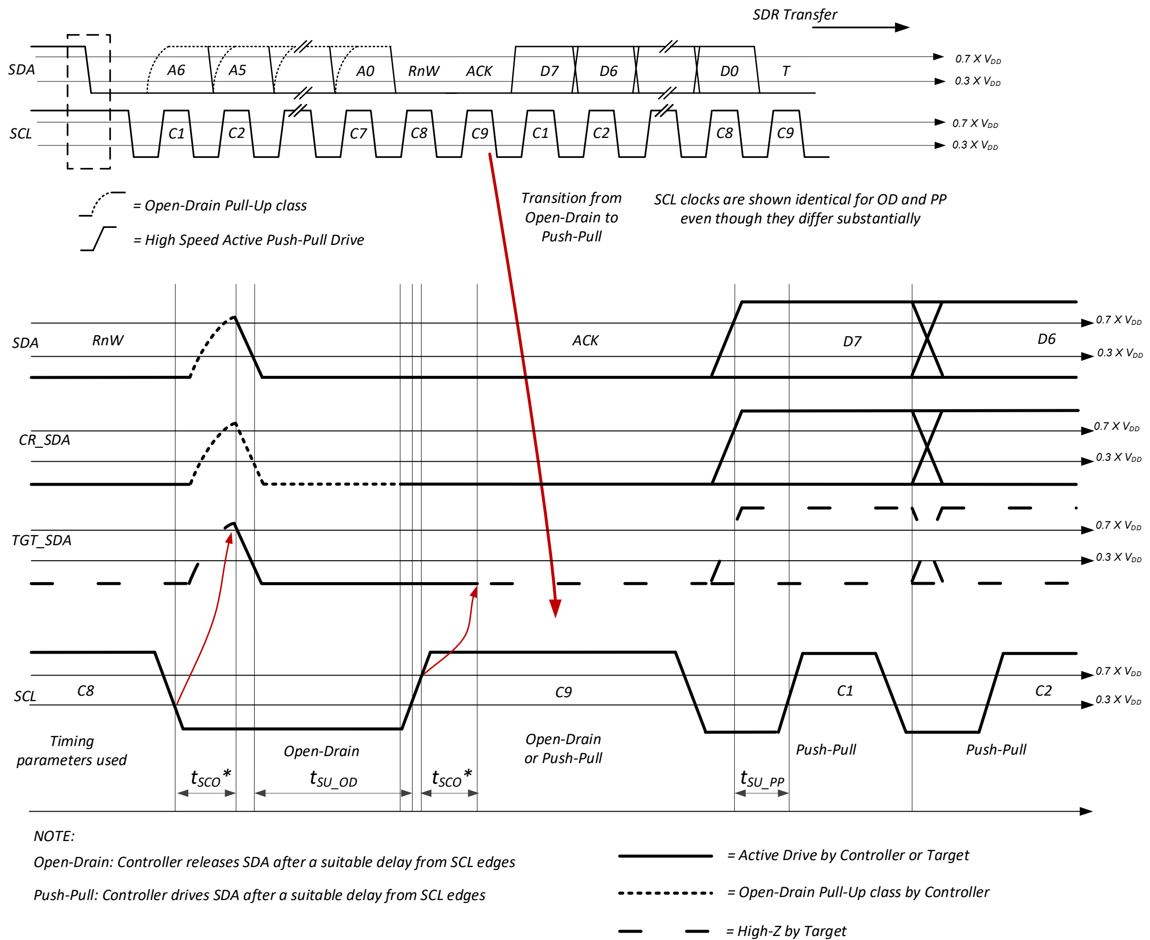

Figure 2.9: SDA handoff during address ACK when writing (copied from MIPI [1]).

2.6 CCC bytes

To manage devices connected to the bus, the I3C specification defines a set of Common Command Codes (CCC), which are 8-bit words that follow the I3C broadcast address (7h'7E/W). Additionally, these commands can be followed by optional data payload and they can be either broadcasted to every target (broadcast CCC) or directed at specific ones (direct CCC) depending on the value of the first bit of the command code.

Vendors can optionally implement their own command code bytes, for which they have dedicated ranges of 0x61-0x7F for broadcast and 0xE0-0xEF for direct command codes.

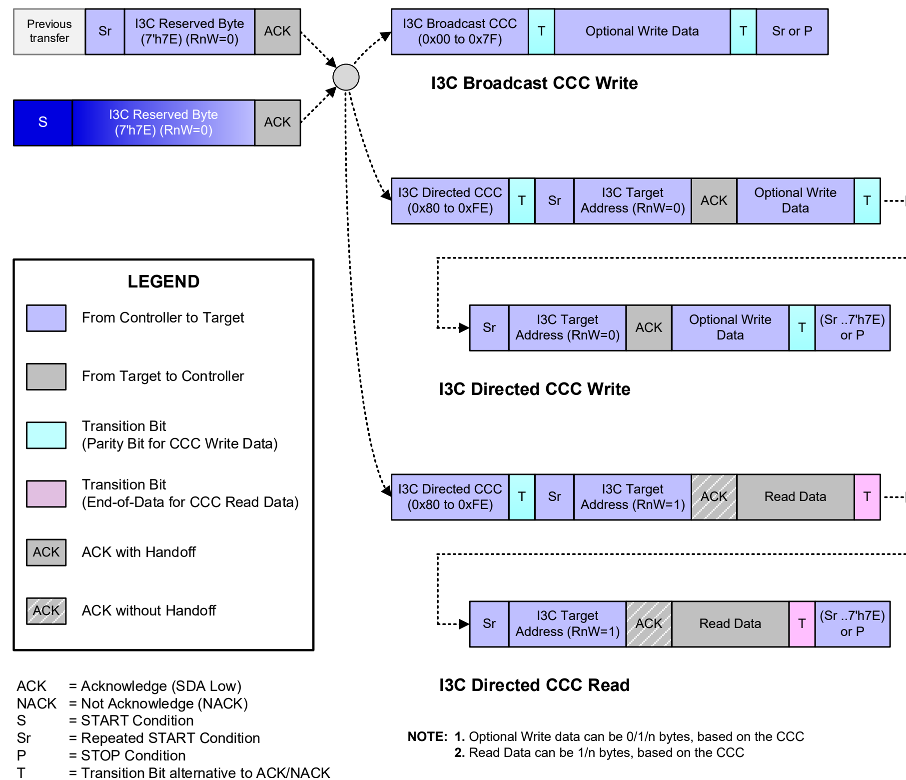

Figure 2.10: CCC transfers (copied from MIPI [1]).

2.6.1 Dynamic Addressing

Instead of using a fixed address, I3C devices use dynamic addresses which are assigned by the controller on the bus. There are multiple benefits to this – multiple devices of the same type can be assigned their own address, the controller sets the in-band interrupt priority between targets, and targets can be grouped together with a common address when supported. There are multiple CCC bytes which can change the target's address:

- **ENTDAA**

Enter Dynamic Address Assignment. A broadcast CCC indicating to all I3C devices that the controller is entering the Dynamic Address Assignment procedure.

During the Dynamic Address Assignment (DAA) procedure (Figure 2.11, the bus switches to an open-drain mode, and all devices compete in an arbitration similar to the address arbitration, but using the values of their Provisioned ID (PID), Bus Characteristic Register (BCR) and Device Characteristic Register (DCR) registers instead of an address. In total, these registers form a 64-bit value which is *likely* to be unique, as the PID can be either set by the manufacturer or have a random value.

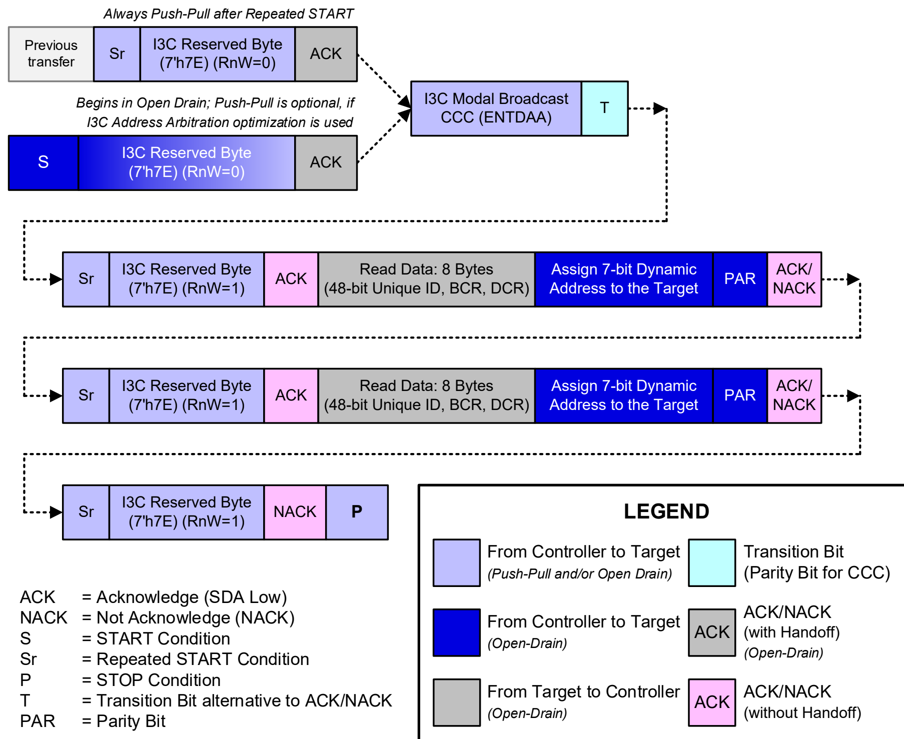

Figure 2.11: Dynamic Address Assignment Transaction using ENTDAA (copied from MIPI [1]).

- **RSTDAA**

  Reset Dynamic Address Assignment. Broadcast CCC for clearing the dynamic addresses assigned to targets.

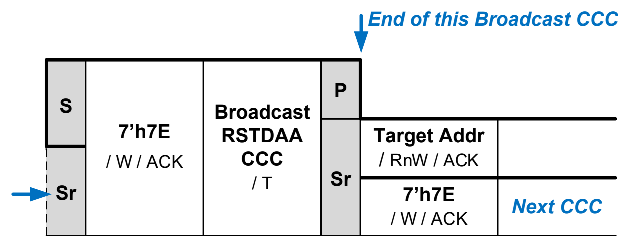

Figure 2.12: RSTDAA CCC format (copied from MIPI [1]).

- **SETAASA**

  Set All Addresses to Static Addresses. Broadcast CCC, sets the dynamic addresses of targets to match their static I2C address *if* they have one.

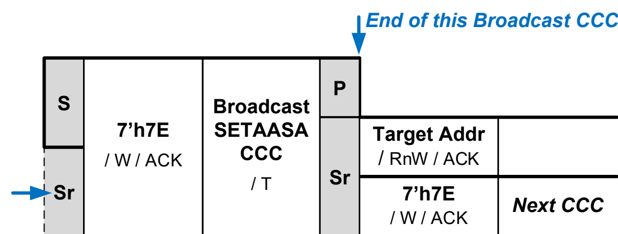

Figure 2.13: SETAASA CCC format (copied from MIPI [1]).

- **SETDASA**

  Set Dynamic Address from Static Address. Directed CCC, sets a new dynamic address of a target by first addressing it by its static I2C address.

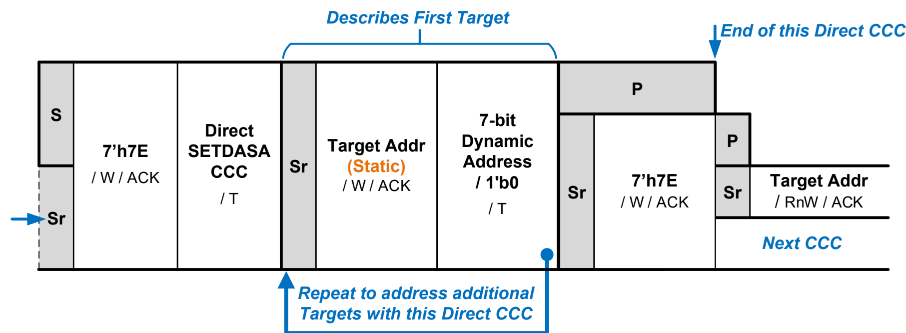

Figure 2.14: SETDASA CCC format (copied from MIPI [1]).

- **SETNEWDA**

  Set New Dynamic Address. Directed CCC, sets a new dynamic address of a target by first addressing it by its old dynamic address.

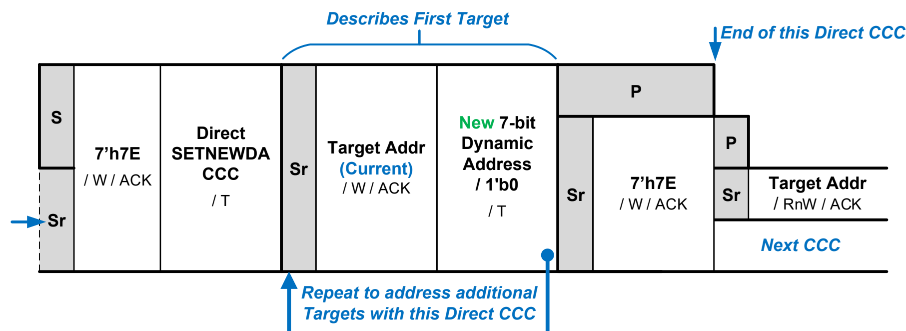

Figure 2.15: SETNEWDA CCC format (copied from MIPI [1]).

2.6.2 In-band Interrupts, Hot-Joins, Resets

Instead of having to route additional lines for interrupts and resets for some or every target, I3C defines a way of transmitting them over the bus. The major benefit of this approach is lower costs associated with PCB manufacturing, as fewer traces need to be routed between chips.

Targets that support In-Band Interrupts (IBI) can compete to send their address onto the bus during address arbitration. If a target wins, the controller knows which device has sent the interrupt and can choose to service it and optionally receive data. Target devices can also pull down the SDA line when the bus is idle, generating a START pattern instead of waiting until a controller sends a START, cutting down the time between an interrupt event happening in a target and the controller being notified.

2.7 Backwards Compatibility

Due to the significant similarity between I3C SDR Mode and the I2C protocol in terms of procedures and conditions, many I3C devices and legacy I2C Target devices can coexist on the same I3C bus. The I3C specification defines that any legacy I2C Target on the I3C bus is:

- *required* to not use clock stretching,
- *required* to support at least I2C Fm speed (400 kbit/s) and
- *desired* to have a 50 ns spike filter.

Clock stretching is a feature in the I3C protocol in which Target devices can pull down the open drain SCL line to slow down communication. The use of this feature is strictly prohibited on the I3C bus due to the SCL line being driven with push-pull outputs by the I3C Controller.

Any communication over the I3C bus is made transparent to all legacy I2C Target by using the *7'h7E* I3C broadcast address, as every I2C Target will see it as an address of another device and wait for the next START condition.

Additionally, if the connected legacy I2C Targets are equipped with a 50 ns SCL spike filter, then the I3C Controller can take advantage of this by reducing its SCL *tHIGH* period to less than 50 ns. This approach will make all I3C transactions transparent to the legacy I2C Target, as it will not see any clocking of the SCL line.

2.8 Power Efficiency

According to a white paper *Achieving Power Efficiency in IoT Devices with MIPI I3C®* released by MIPI ([2]), the largest contributors to power consumption in the I3C bus are:

1. **Shoot-through current** of the push-pull pads, which happens during line transitions. Determined by technology used for the output pads.
2. **Charging/discharging of bus capacitance**, which is a sum of the bus wire capacitance and input/output pads capacitance.
3. **Current through the pull-up resistor**, which is only present when the SDA line is driven low when the pull-up is enabled.

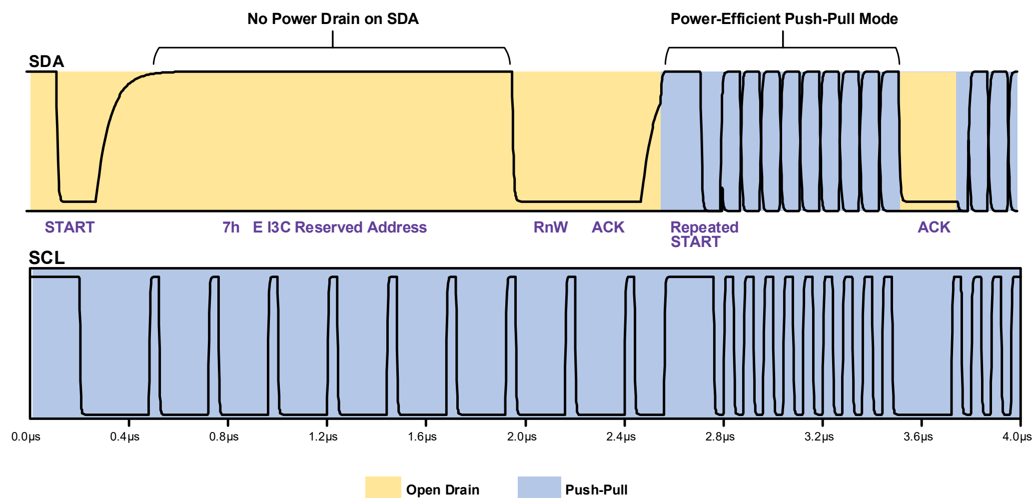

Figure 2.16: Typical I3C data transfer (MIPI, [2])

This shift to push-pull is noticeable in both the throughput of data (Figure 2.17), as well as the energy needed to transmit 1 kB of data on the bus (Figure 2.18).

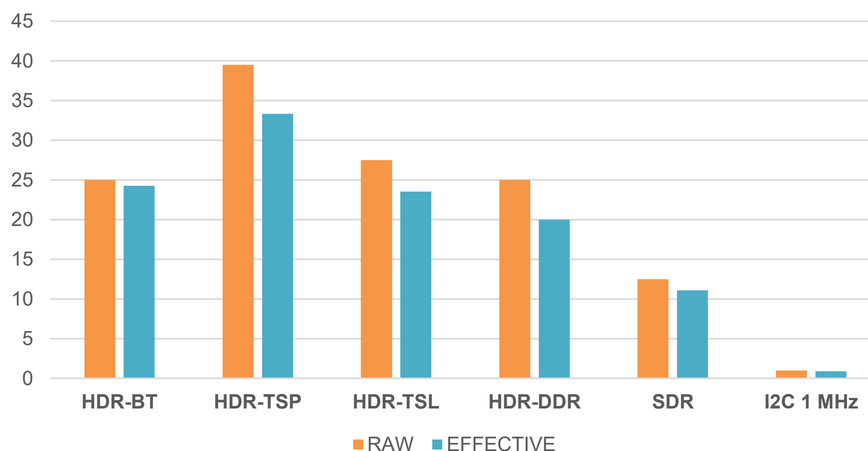

Figure 2.17: Bitrates for 12.5 MHz clock in Mbps compared to I2C (MIPI [2])

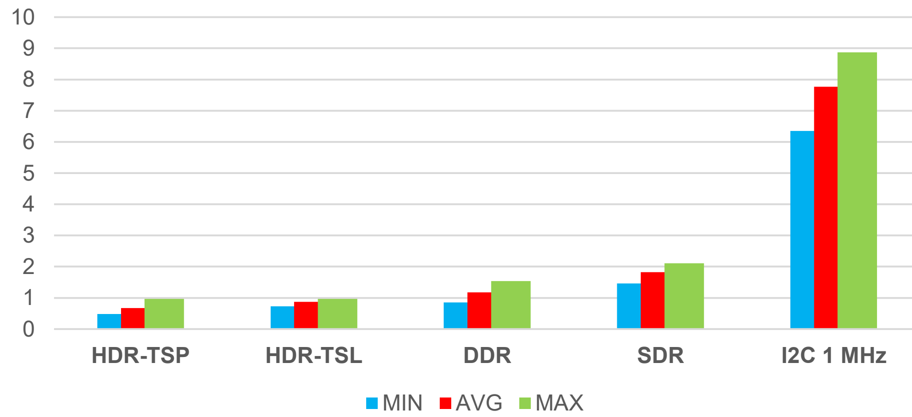

Figure 2.18: Effective energy range per 1 kB (μJ) compared to I2C (MIPI [2])

When the controller sends the I3C Broadcast address, the SDA line must be kept in open drain mode, because the address is arbitrable. This part of the transaction has to run at a slower SCL clock due to the slow open drain rising edge of SDA. After a Target ACKs the I3C Broadcast address, the Controller can then switch to the faster, more efficient push-pull mode. This transition eliminates the current through the pull-up resistor, as it is not used on the bus, with the exception of some SDA line handoffs and the Dynamic Address Assignment procedure. This drastically lowers the power consumption of the bus (Figure 2.18) while also increasing the maximum data throughput (Figure 2.17).

Chapter 3

System Level Design

This chapter outlines the general requirements and functionality of the designed I3C Controller IP core. The core is to be implemented into and tested with a RISC-V platform utilizing the CV32E40P core. The platform had an I2C Master block, whose functionality will be superseded by the I3C Controller block.

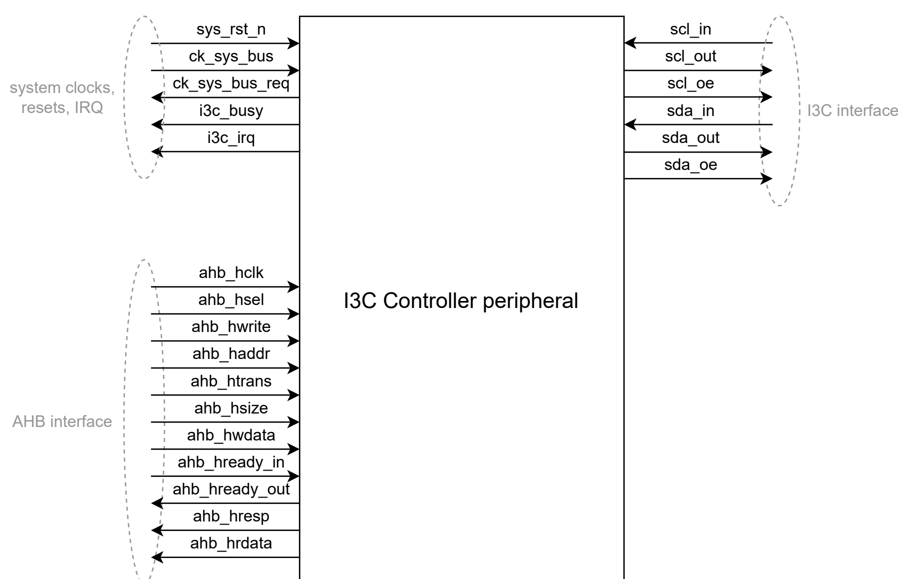

Figure 3.1: Diagram of the I3C Controller peripheral.

3.1 System design

We lay the following requirements of the design:

- The peripheral shall be backwards compatible with I2C devices and support operating in a mixed bus.
- The peripheral shall support I3C SDR transmissions up to 12.9 MHz.
- The peripheral shall support legacy I2C Fast mode+ transmissions up to 1 MHz.
- The designed I3C controller is to be a low-level block capable of handling basic tasks (data Tx, Rx), but reliant on software for higher level functions, such as address assignment, group addressing and responding to In-Band Interrupts.
- The In-Band Interrupts and Hot-Join requests will be handled by the CPU through polling or an interrupt, after which the CPU will instruct the peripheral on which action to take.

- The peripheral shall not support handling over the role of the active controller, as it is intended to be used in a single controller design. This decision was made as implementation would result with the need to incorporate an I3C Target into the peripheral as well.
- The generated Serial Clock signal high and low periods shall be independently configurable to allow for a programmable duty cycle, allowing the I3C communication to be transparent to I2C targets with a 50 ns spike filter.
- The outgoing and incoming data shall be be stored in generically sized FIFO memories, whose contents are to be accessible by the CPU through the register map. These memories shall indicate whether they are full, empty, or have reached a watermark value.
- The peripheral shall be synchronous with the system clock and all necessary timings are to be configurable on the fly by the CPU as multiples of the system clock period.
- Most of the peripheral shall be held in reset while in idle state, such as any Finite State Machines (FSM). The exception to this are parts of the register map, such as the timing registers, FIFO memories or the enable register.
- The peripheral shall use clock gating to lower the power consumption while the peripheral is not used. The peripheral will also gate clocks to its internal blocks that are not directly accessed or used.
- The peripheral shall use the AMBA AHB-lite interface to communicate with the CPU, as it is already present in the RISC-V platform.
- The SDA pad driver shall be weaker or configurable to be weaker than of the connected Target devices to detect collisions on the bus when in push-pull mode. Alternatively, a different method of detecting collisions shall be incorporated into the design.

3.2 SW/HW decomposition

The Controller shall be designed to be a low-level I3C Controller, i.e. handling only the most elemental tasks. The Controller hardware shall be capable of:

- generating patterns and conditions (START, STOP, restart, etc.),
- sending and receiving 8 data bits + (N)ACK bit (addresses and legacy I2C transactions),
- sending and receiving 8 data bits + T-bit (I3C transactions),
- automatic repeated START insertion before addresses,
- enter Dynamic Address Assignment procedure, and
- receiving In-Band Interrupts through Address Arbitration and START condition.

To future-proof the design, it was decided that all higher-level behavior is to be implemented in software. This means that the designed peripheral is capable of handling basic data transfers independently on any higher-level protocol transfers happening on the bus.

This design direction makes the Controller easy to reconfigure or to extend its functions using the embedded software that controls its operation, while also reducing the number of unused dedicated logic when a function is not required.

An example of a higher-level feature is *Group Addressing*, where multiple Targets can be accessed for data writes with a shared group address. This feature uses the **SETGRPA** and **RSTGRPA** Command Codes for configuration. The Controller transfers these Command Codes onto the bus without tracking which Target is in which group, as accessing these Targets is conceptually the same as accessing any Target - by their address. In this case, their Group Address instead of their individual Dynamic Addresses.

The advantages and disadvantages of a low-level peripheral can be summarized as:

+ smaller footprint of the peripheral,

+ higher level functions can be easily updated or extended,

+ quicker RTL development and verification,

- increased software complexity,

- in some cases the reaction time of the controller can be larger than if it were all implemented in dedicated logic.

3.3 Resets

The peripheral has two asynchronous reset signals - rst_n and rst_n_soft. The rst_n is driven from the *sys_rst_n*, which is the RISC-V platform's reset signal and resets the whole peripheral. The peripheral presumes that the sys_rst_n signal is synchronously released and therefore its resynchronization does not take place in the peripheral.

The soft reset signal is driven by a combination of the rst_n signal and the peripheral enable signal stored in the register map. Most of the peripheral uses the soft reset to be held in a known default state while unused with the exception of the register map and FIFO memories, as losing the configuration of the peripheral and stored data upon disabling the peripheral would be counterproductive.

3.4 Clocks

The peripheral is designed to be entirely synchronous to the system's clock in order to prevent difficulties that arise with clock domain crossing. In total, the peripheral contains 5 clock signals, all of which are driven from the system ck_sys_bus clock:

- ck_i3c - the peripheral's main clock signal, input for the clock gen block,
- ahb_hclk - clock used by the AHB interface,
- ck_i3c_regmap - clock required for parts of the register map accessed by the peripheral,
- ck_i3c_ctrl - clock used for the Control FSM, runs while the peripheral is busy,
- ck_i3c_resync - clock used by the SDA and SCL resynchronizers.

Bus mode	Minimum ck_{sys} frequency
Legacy I2C	0 MHz
I3C Mixed	293 MHz
I3C Pure	180 kHz

The I3C Pure mode has its minimum frequency limited by the minimum SCL clock frequency of 10 kHz and the lowest acceptable timings configuration of the peripheral. The I3C Mixed mode has its minimum frequency limited by the maximum SCL high period of 41 ns. These frequencies were calculated as:

$$f_{\text{ck_sys_min}} = \frac{t_{\text{scl_pp_high_min}} + 3 + 6}{t_{\text{high_max}}} = \frac{3+3+6}{41 \cdot 10^{-9}} = 293\text{MHz}$$

The **+3** and **+6** constants are caused by the SCL Gen FSM and Control FSM respectively and are discussed in sections 4.11 *SCL clock generation* and 4.8 *Control block*.

The minimum required frequency of the system clock for a given SCL clock frequency can be calculated as:

$$f_{\text{ck_sys_min}} = f_{\text{SCL}} \cdot (t_{\text{scl_pp_high}} + 3 + 6 + t_{\text{scl_pp_low}} + 3)$$

3.5 I3C Controller interface

Table 3.1: I3C Controller interface - generics

Generic name	Description
G_FIFO_WIDTH_TX	Width of the TxFIFO address, sets the memory size to 2^N bytes.
G_FIFO_WIDTH_RX	Width of the RxFIFO address, sets the memory size to 2^N bytes.

Table 3.2: I3C Controller interface - signals

Signal name	direction	reset value	Description
sys_rst_n	IN		System reset, active low
ck_sys_bus	IN		System clock
ck_sys_bus_req	OUT	0	System clock request
i3c_busy	OUT	0	Peripheral busy
i3c_irq	OUT	0	Interrupt request by the peripheral
ahb_hclk	IN		AHB system clock
ahb_hsel	IN		AHB decoded peripheral select
ahb_hwrite	IN		AHB transfer direction indicator
ahb_haddr[:]	IN		AHB address
ahb_htrans[1:0]	IN		AHB transfer type
ahb_hsize[:]	IN		AHB transfer size
ahb_hwdata[31:0]	IN		AHB write data
ahb_hready_in	IN		AHB bus ready
ahb_hready_out	OUT	1	Target AHB output ready
ahb_hresp	OUT	0	Target AHB bus transfer response
ahb_hrdata[31:0]	OUT	0x0000 0000	Target AHB read data
scl_in	IN		I3C Serial Clock in
scl_out	OUT	1	I3C Serial Clock out
scl_oe	OUT	0	I3C Serial Clock output enable
sda_in	IN		I3C Serial Data in
sda_out	OUT	1	I3C Serial Data out
sda_oe	OUT	0	I3C Serial Data output enable

3.6 Differentiating Addresses from Data

During the conception of the system design of the I3C Controller, it was decided that the Controller shall not decide itself which byte in the FIFO memory is an address or data to send onto the bus. This results in an increased software overhead, as it would have to either:

a) queue command regarding START and STOP conditions, or

b) mark addresses in the TxFIFO memory.

The latter approach was chosen, as adding an extra *Flag bit* to mark addresses is easier than having to prepare the whole transaction sequence in software, especially in case many short transactions are queued in the TxFIFO memory.

The TxFIFO memory was enlarged to 9 bits, allowing to store the Flag bit per every byte in the FIFO memory. During operation, the designed I3C Controller reads the Flag bit and based on it it either:

- sends data if ’0’, or
- sends address if ’1’ and START condition was sent before this byte, or
- sends a START condition followed by the address.

The Flag bit is stored in the MSB position so as to not interfere with normal data when writing into the TxFIFO memory.

3.7 Supported Command Codes

The made I3C Controller is designed to handle Command Codes in a generic manner - after all, it is just data sent to the I3C Broadcast address instead of a specific Target device. This approach makes handling CCCs reliant on software [controlling] the Controller. Almost all CCCs are expected to be supported, as long as they do not require hardware to behave in “special” ways.

Of the supported Command Codes listed in Table 3.3, only one requires special handling by the hardware – **ENTDAA** (Enter Dynamic Address Assignment). This command is detailed in Section 2.6.1 *Dynamic Addressing*. Its implementation requires the hardware to switch its SDA driver to open drain, lower the SCL clock to open drain timings, and read 64 bits without any (N)ACKs or T-bit, whereas other Command Codes consist of “normal” data transfer consisting of 8 data bits followed by a T-bit.

Table 3.3: Supported CCC commands

CCC Code	CCC Type	Command Name	Description
0x00	Broadcast	**ENEC**	Enable Target event driven interrupts
0x01	Broadcast	**DISEC**	Disable Target event driven interrupts
0x02–0x05	Broadcast	**ENTAS0–5**	Set activity state 0–5
0x06	Broadcast	**RSTDAA**	Forget current Dynamic Address and wait for new assignment
0x07	Broadcast	**ENTDAA***	Controller started the Dynamic Address Assignment procedure
0x09	Broadcast	**SETMWL**	Maximum write length in a single command
0x0A	Broadcast	**SETMRL**	Maximum read length in a single command
0x0B	Broadcast	**ENTTM**	Controller has entered Test Mode
0x0C	Broadcast	**SETBUSCON**	Controller specifies a higher-level protocol and/or I3C specification version that the Bus will use
0x29	Broadcast	**SETAASA**	Controller tells every Target with a Static Address to use it as the Dynamic Address
0x2A	Broadcast	**RSTACT**	Configure and query Target Reset action and timing
0x2C	Broadcast	**RSTGRPA**	Controller removes a Target from an indicated Group Address by resetting the assigned Group Address
0x61–7F	Broadcast	Vendor / Standards Extension – Broadcast CCCs	For Vendors or Standards use
0x80	Direct	**ENEC**	Enable Target even driven interrupts
0x81	Direct	**DISEC**	Disable Target even driven interrupts
0x82–85	Direct	**ENTAS0–5**	Set activity state 0–5
0x87	Direct Set	**SETDASA**	Controller assigns a Dynamic Address to a Target with a known Static Address
0x88	Direct Set	**SETNEWDA**	Controller assigns a new Dynamic Address to any I3C Target with an existing Dynamic Address

CCC Code	CCC Type	Command Name	Description
0x89	Direct Set	**SETMWL**	Maximum write length in a single command
0x8A	Direct Set	**SETMRL**	Maximum read length in a single command
0x8B	Direct Get	**GETMWL**	Get Target's maximum possible write length
0x8C	Direct Get	**GETMRL**	Get Target's maximum possible read length
0x8D	Direct Get	**GETPID**	Get Target's Provisioned ID
0x8E	Direct Get	**GETBCR**	Get Target's Bus Characteristic Register (BCR)
0x8F	Direct Get	**GETDCR**	Get a Device's Device Characteristics Register (DCR)
0x90	Direct Get	**GETSTATUS**	Get a Device's operating status
0x93	Direct Set	**SETBRGTGT**	Controller tells Bridge (to/from I2C, SPI, UART, etc.) what endpoints it is talking to (by Dynamic Address and type/ID)
0x94	Direct Get	**GETMXDS**	Controller asks Target for its SDR Mode maximum. Read and Write data speeds (& optionally maximum Read Turnaround time)
0x95	Direct Get	**GETCAPS**	Controller asks Target what optional capabilities it supports
0x96	Direct	**SETROUTE**	Controller tells Routing Device wha Route(s) to enable
0x9A	Direct	**RSTACT**	Configure and query Target Reset action and timing
0x9B	Direct	**SETGRPA**	Controller assigns Group Addresses
0x9C	Direct	**RSTGRPA**	Controller removes a Target from an indicated Group Address by resetting the assigned Group Address
0xE0–FE	Direct	Vendor / Standards Extension – Direct CCCs	For Vendors or Standards use

The **ENTDAA** CCC relies on software to be properly used, as the Dynamic Address Assignment procedure detailed in Section 2.6.1 *Dynamic Addressing* is entered after the transmission of I3C Broadcast + Read address *without* any checks of previously sent data. This means that the Controller can enter the DAA procedure without sending the ENTDAA CCC, which would result in all I3C Targets returning a NACK.

Table 3.4: Unsupported CCC commands

CCC Code	CCC Type	Command Name	Description
0x08	Broadcast	**DEFTGTS**	Controller defines Dynamic Address, DCR Type, and Static Address (or 0) per Target
0x12	Broadcast	**ENDXFER**	Framework for controllers and Targets to exchange set-up parameters for ending data in supported HDR Modes
0x20–0x27	Broadcast	**ENTHDR0–7**	Controller has entered HDR (DDR / TSP / TSL / BT / reserved) Mode
0x28	Broadcast	**SETXTIME**	Framework for exchanging event timing information
0x2B	Broadcast	**DEFGRPA**	Controller tells Secondary controllers details about an indicated Group Address
0x2D	Broadcast	**MLANE**	Control a Multi-Lane Data Transfer

CCC Code	CCC Type	Command Name	Description
0x91	Direct Get	**GETACCCR**	Active Controller is passing the Bus Controller Role to a Secondary Controller and confirming its acceptance
0x92	Direct	**ENDXFER**	Framework for controllers and Targets to exchange set-up parameters for ending data in supported HDR Modes
0x98	Direct	**SETXTIME**	Framework for exchanging event timing information
0x99	Direct	**GETXTIME**	Framework for exchanging event timing information
0x9D	Direct	**MLANE**	Control a Multi-Lane Data Transfer

The software *is able to* transmit unsupported Command Codes listed in Table 3.4, but doing so *is discouraged*, as it could result in erroneous behavior. The peripheral is not equipped to handle the requirements to properly send or respond to these Command Codes.

An example of this is the **GETACCCR** – Get Accept Controller Role. The peripheral is not designed to hand over the role of the Active Controller to another device, as it is incapable of acting as an I3C Target device.

The listed Command Codes were not supported primarily due to time constraints, as the development and testing required more time than anticipated. Additionally, some of the Command Codes are not present in the freely available specification released by the MIPI organization, such as the *ENTHDR2* CCC (Enter HDR-TSL mode).

3.8 Error recovery

Table 3.5 summarizes the error types specified in the I3C Basic Specification (CE0–CE3) [1] as well as additionally implemented error types (PE0 & PE1). The custom error types use the **PEx** (Peripheral Error) descriptor so as to not confuse them with Controller Error (CEx) or Target Error (TEx).

Table 3.5: SDR Controller Error Types

Error Type	Description	Error Detection Method	Error Recovery Method
CE0	Transaction after sending CCC	HW/SW detects illegally formatted CCC	HW/SW stops the transmission, SW sends STOP and retries the transmission
CE1	Monitoring Error	HW detects transmitted data differ from what it intended to transmit	HW stops the transmission, SW sends STOP and retries the transmission
CE2	No response to Broadcast Address	HW detects NACK after Broadcast Address	Stops the transmission, SW sends HDR Exit followed by STOP
CE3	Failed Controller Hand-off	Not implemented	Not implemented
PE0	No response from Target	HW detects NACK after Target address / I2C write transfer	HW/SW stops the transmission, SW sends STOP and retries the transmission
PE1	Target ended read	HW detects End-Of-Data during I3C private read	HW stops the transmission, SW sends STOP and retries the transmission

Error Type CE0

For Command Codes which include reading from a Target, the software specifies the number of data bytes that are expected to be read by the Controller. If a Target terminates the read, then the hardware stops the transmission. An example of this is if the Controller received only one byte from a Target in a GETMWL CCC code since the Controller expects two bytes.

If the Controller receives the correct number of bytes but the received data is not valid in some way, then it is up to the software to detect this error and instruct the Controller to retry the transmission. An example of this is if the Controller received bits[3:0] of 0b'0000 in the second byte of GETCAPS CCC, which would indicate a Target compliant with MIPI I3C Basic v1.0. This would indicate that these bits were read incorrectly, as the second byte of GETCAPS CCC is defined in later MIPI I3C Specifications.

Error Type CE1

Monitoring Error CE1 occurs when the Controller detects that the transmitted data differs from what the Controller intended to transmit. This check is only done after Address Arbitration, since differing transmitted data during Address Arbitration result in an In-Band Interrupt, which is the desired behavior.

An example of this error occurring is if a Target misinterpreted the RnW bit in a private write transfer. This would result in both the Target and Controller driving the SDA line in push-pull mode. When the Controller detects such an error, it switches its SDA driver to open-drain and lets go of the SDA line to prevent further contrary driving. The Controller then finishes the current SDR byte transfer, stops all transmissions and notifies the software. The software is then required to send a STOP and retry the transaction if desired.

Error Type CE2

If the Controller does not receive an ACK upon transmitting the I3C Broadcast Address (7'h7E), then the hardware stops all transmissions and notifies the software. The software is then required to send an HDR Exit Pattern followed by STOP in order to recover any Target after a TE0, TE1, TE2, TE5, and TE6 errors specified in the MIPI I3C Basic Specification [1].

Error Type PE0

Similar to CE2, if the Controller does not receive an ACK upon transmitting an address or does not receive an ACK upon legacy I2C data write, then the hardware stops all transmissions and notifies the software. Further action is then up to the software, which either stops the transaction or retries it.

Error Type PE1

The PE1 error type shares the working principle with CE0, as in it detects that the Target sent End-Of-Data during a private read transaction (Target terminated read) before the expected number of bytes were read. The hardware stops all transmissions and notifies the software, which either stops the transaction or retries it.

Chapter 4

RTL Design

This chapter describes the designed RTL blocks used by the system. The block diagram of the peripheral (Figure 4.1) shows the layout and general connections between the blocks, which are described later on. The peripheral is fully synchronous to the system clock, therefore no resynchronization is needed when communicating with the rest of the RISC-V system. The two resynchronizers used by the design are used to read responses from Target devices and to accommodate delays caused by pads and the physical connections of the I3C bus.

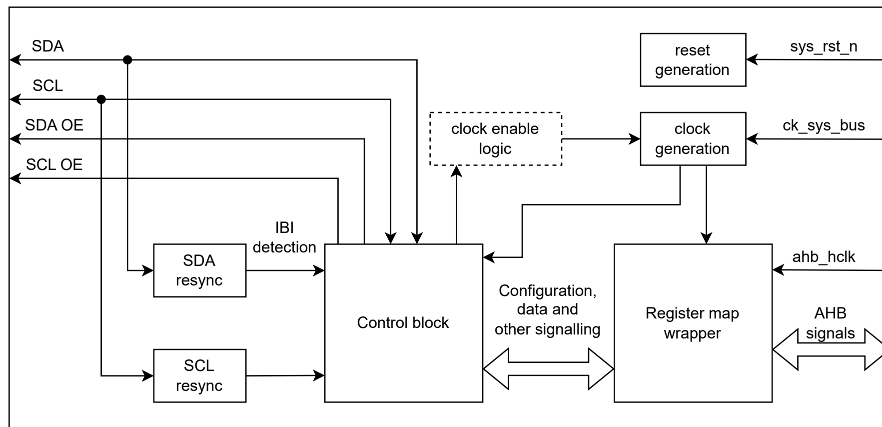

Figure 4.1: Diagram of the I3C Controller peripheral.

4.1 Clock gating, resets

The clock signal used by the peripheral is split into multiple signals, which are gated according to the usage of the Controller. Clock gating is only used in the ASIC design of the peripheral and uses a LATCH+AND cell. The latch is used to prevent glitching on the gated clock, as it only samples the enable signal while the clock signal is low.

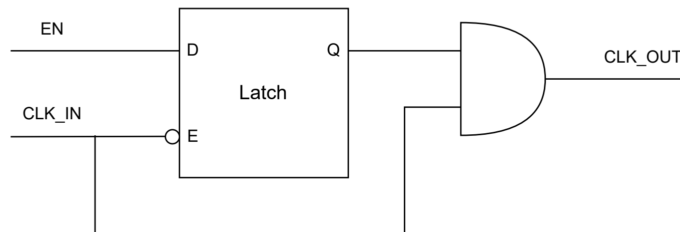

Figure 4.2: Clock gating cell used in the peripheral.

The design of the peripheral assumes and requires the asynchronous *rst_n* signal to be synchronously released by the system. This assumption was made as the I3C Controller is designed

to be used alongside other peripherals that also require this functionality, therefore implementing it in each peripheral would be ineffective.

The peripheral also creates its own soft reset signal, which is made from a logical AND between the *rst_n* and the peripheral's enable signal. The AND gate was instantiated by hand as to prevent glitching if the peripheral was ever expanded with a need for more complex reset behavior.

4.2 Counters

The counter entity is set to count downwards but otherwise is designed to be as simple and as generic as possible. The entity is mentioned as it is used in multiple parts of the peripheral, which will be discussed later.

The entity uses synchronous *load* and *tick* signals and makes its counter value accessible for reading as well as a *done* signal, which is set to '1' when a 'tick' happens when the counter is at zero.

4.3 Register map wrapper

To simplify the design, the AHB register map, the two FIFO memories and some additional registers were wrapped into a single entity, the *register map wrapper*. The register map itself was created using a generator script based on a configuration file.

Some registers needed to be implemented manually, as they either need to be written into by the peripheral (eg. interrupt register) or the peripheral needs to be informed whether a read or a write occurred (FIFO access registers). Accessing the FIFO memories is discussed in Section 4.4 *FIFO memories*.

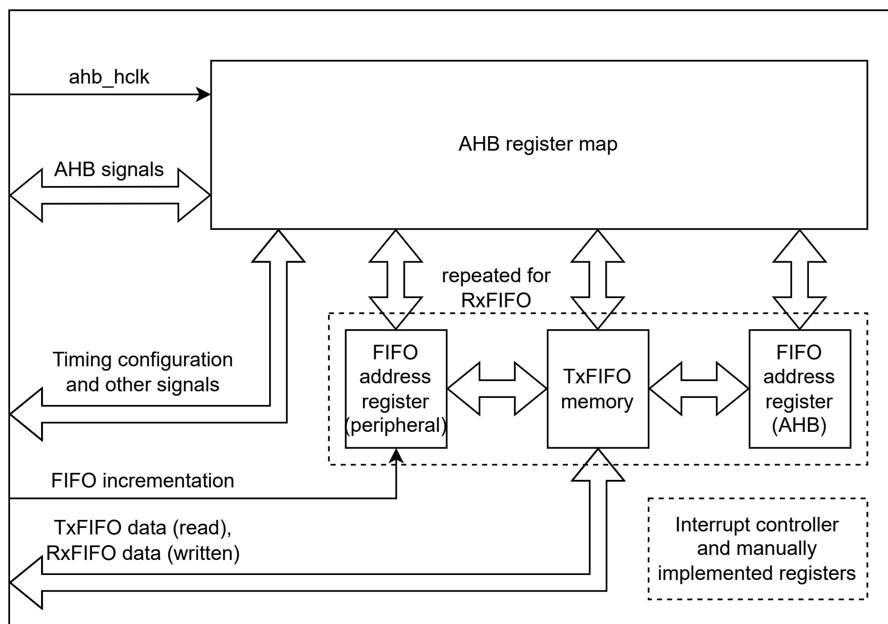

Figure 4.3: Diagram of the register map wrapper.

4.4 FIFO memories

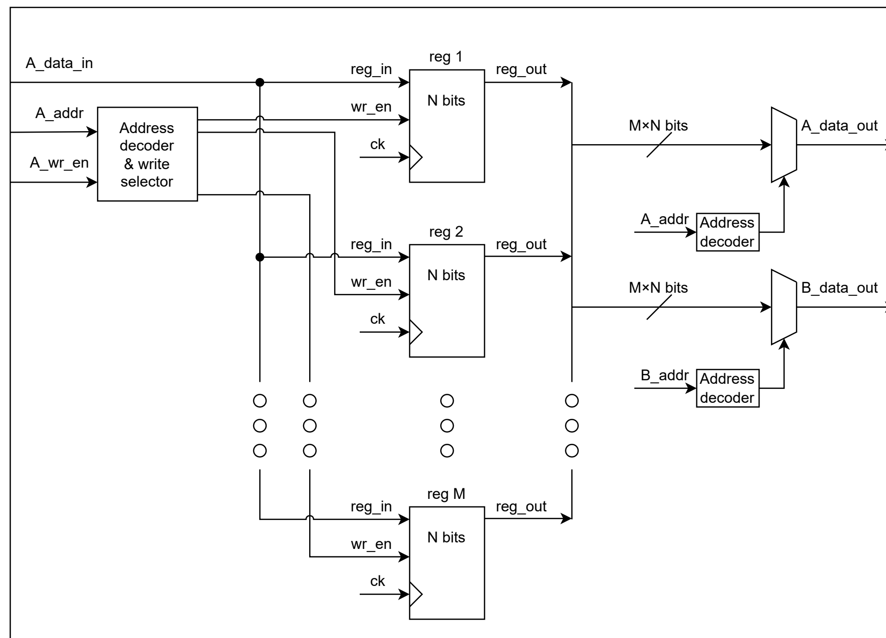

Figure 4.4: Diagram of the FIFO memory. Port A is read/write, port B is read only.

The FIFO memories are designed as synchronous with two interfaces - A and B. One of the interfaces is accessed by the CPU and the other internally by the peripheral. Additionally, the FIFO memory is designed with parametrization in mind, where the memory stores N data words of M bits (where N and M are generic natural integers).

To simplify the design, only one interface is capable of read/write access to the FIFO memory while the other can only read. This design choice was done to simplify the FIFO memories while still allowing read/write access to the TxFIFO memory while only limiting the access to the RxFIFO memory to read-only.

4.5 FIFO Address registers

Each of the two FIFO memories has two pointers stored in the register map wrapper - one for access from the AHB (CPU) and one internal. Both of these pointers are accessible to the CPU through the register map for reading and writing, but special care should be taken when writing the value.

The pointers are 1 bit larger than required for access of the FIFO memories to distinguish between FIFO full and FIFO empty, as when both pointers are *0*, there is no meaningful way of distinguishing between these two states. When the FIFO pointers match and their most significant bits are the same, then the FIFO memory is empty as neither of the pointers has wrapped around yet. The behavior of this extra bit is discussed in Clifford E. Cummings' work on asynchronous FIFO design [3].

The incrementation of the AHB pointers can be either done manually by software or automatically during reading and/or writing to the FIFO memories, which is toggleable by setting the **xFIFO_ptr_incr_read* / **xFIFO_ptr_incr_write* bits to '1'. This incrementation is blocked for the appropriate pointer upon TxFIFO full and RxFIFO empty, so as to not cause either TxFIFO overflow or RxFIFO underflow.

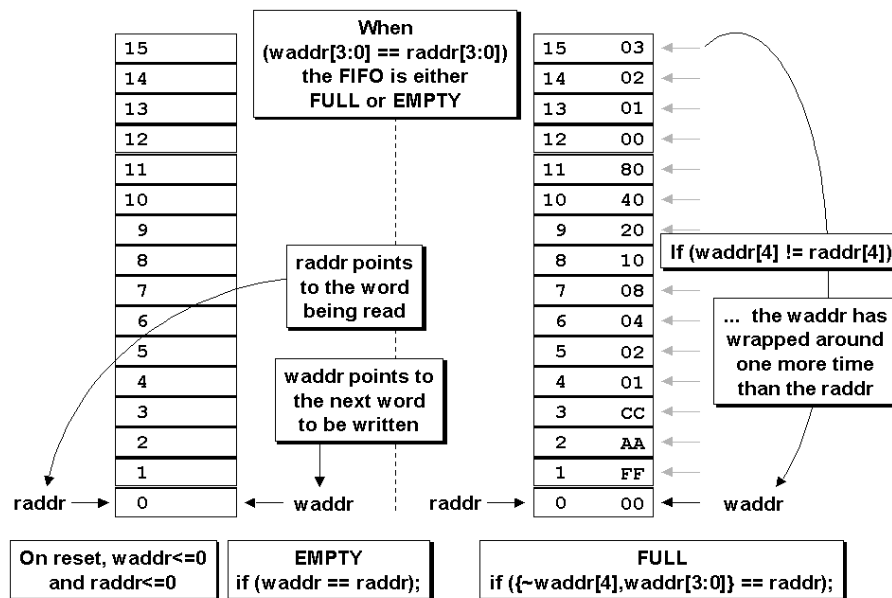

Figure 4.5: FIFO full and empty conditions. (Copied from Simulation and Synthesis Techniques for Asynchronous FIFO Design [3])

4.6 Interrupt controller

The register map wrapper also contains the interrupt controller, used for notifying the system that an interrupt event has occurred in the I3C Controller. The design supports 10 interrupt sources, listed in Table 4.1. Each interrupt source sets its bit in the *IRQfl* register, which is then masked by the *IRQen* register. All of the masked bits then undergo an OR operation and are sampled by a D Flip-Flop (DFF), which then drives the *i3c_irq* signal of the peripheral.

Table 4.1: Interrupt sources

Interrupt name	Description
Transmission finished	The START, STOP, Transmit, etc. bits cleared by the peripheral.
Broadcast NACK	The I3C broadcast address (7'h7E) returned a NACK.
Target NACK	I3C/I2C address or I2C write returned a NACK.
Read terminated	I3C Target terminated read earlier than the controller expected.
Monitoring Error	Collision on the SDA line detected.
IBI	In-Band Interrupt detected. Either a START condition or Controller lost Address Arbitration.
RxFIFO full	RxFIFO memory full.
TxFIFO empty	TxFIFO memory empty.
RxFIFO watermark	RxFIFO memory reached its watermark level.
TxFIFO watermark	TxFIFO memory reached its watermark level.

4.7 I3C Controller registers

This section lists all of the registers of the I3C Controller peripheral, their reset values, and what is stored at each of the bits. The figures are separated into two 16-bit slices and each grouping of bits has its access type listed below, with *reserved* being unused. The used access types are:

- R/W – Read and Write,
- R – Read only, and
- W1 – Write '1' only.

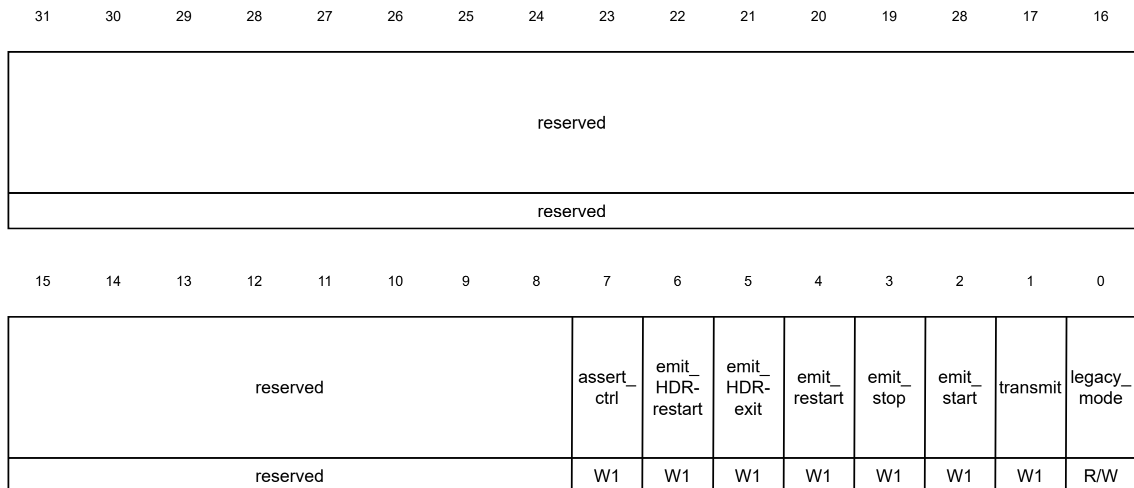

Figure 4.6: I3CC Control Register 1

Reset value: 0x0000 0000

[0] **legacy_mode**
1 - I2C transmission
0 - I3C transmission

[1] **transmit**
W1 - controller starts transmitting data from TxFIFO memory

[2] **emit_start**
W1 - Controller sends (repeated) START pattern

[3] **emit_stop**
W1 - Controller sends STOP pattern

[4] **emit_restart**
W1 - Controller sends Restart pattern

[5] **emit_HDRexit**
W1 - Controller sends HDR-Exit pattern

[6] **emit_HDRrestart**
W1 - Controller sends HDR-Restart pattern

[7] **assert_ctrl**
W1 - Controller attempts to assert control over the SDA line (bus recovery)

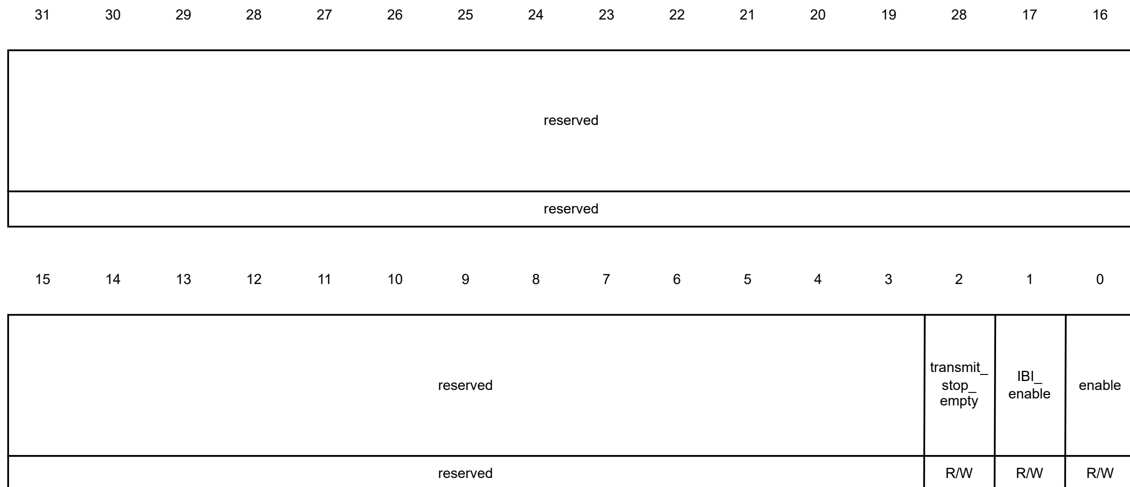

Figure 4.7: I3CC Control Register 2

Reset value: 0x0000 0004

[0] **enable**

1 - the peripheral is enabled and capable of sending and receiving data.

0 - the peripheral is held in a soft reset. Contents of the FIFO memories and registers are still accessible by the system and retain their values.

[1] **ibi_enable**

1 - peripheral monitors the SDA line for a START condition initiated by other I3C devices on the bus.

0 - peripheral does not monitor the SDA line. In-Band Interrupts are still possible, but are only detected upon the loss of the Address Arbitration.

[2] **transmit_stop_empty**

1 - peripheral stops transmitting data once TxFIFO memory is empty. The *transmit_sts* bit of *I3CC stat1* register is cleared upon TxFIFO empty.

0 - peripheral waits for additional data once TxFIFO memory is empty and does not clear the *transmit_sts* bit. This results in the controller resuming a data transfer once the TxFIFO memory has data written into it.

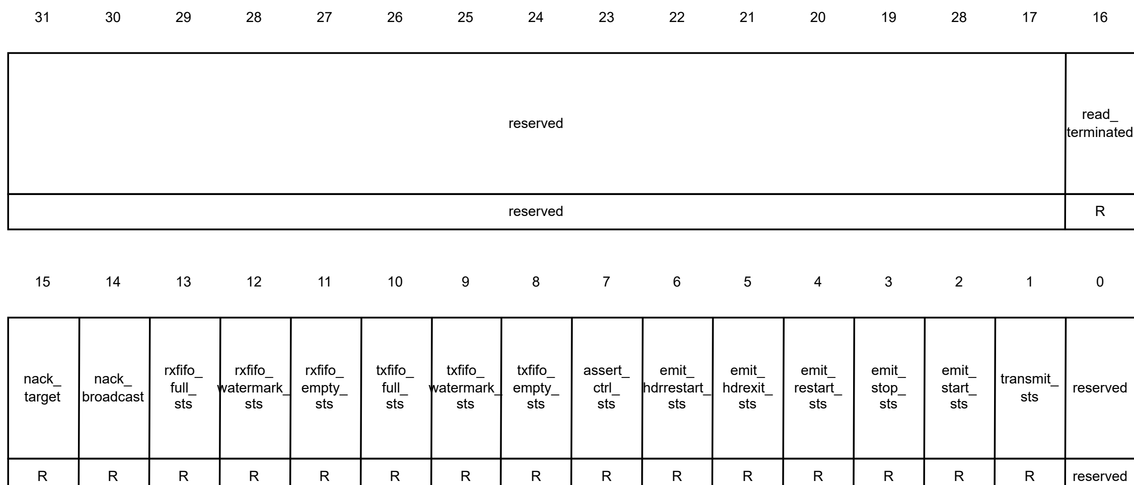

Figure 4.8: I3CC Status 1 register

Reset value: 0x0000 0000

[1] **transmit_sts**
1 - peripheral is transmitting data from TxFIFO memory.

[2] **emit_start_sts**
1 - peripheral is sending a (repeated) START condition.

[3] **emit_stop_sts**
1 - peripheral is sending a STOP condition.

[4] **emit_restart_sts**
1 - peripheral is sending a Restart pattern.

[5] **emit_hdrexit_sts**
1 - peripheral is sending an HDR-Exit pattern.

[6] **emit_hdrrestart_sts**
1 - peripheral is sending an HDR-Restart pattern.

[7] **assert_ctrl_sts**
1 - peripheral is asserting control over the SDA line (bus recovery).

[8] **txfifo_empty_sts**
1 - TxFIFO memory is empty.

[9] **txfifo_watermark_sts**
1 - TxFIFO memory has reached its watermark level.

[10] **txfifo_full_sts**
1 - TxFIFO memory is full.

[11] **rxfifo_empty_sts**
1 - RxFIFO memory is empty.

[12] **rxfifo_watermark_sts**
1 - RxFIFO memory has reached its watermark level.

[13] **rxfifo_full_sts**
1 - RxFIFO memory is full.

[14] **nack_broadcast**
1 - the I3C Broadcast address (7'h7E) returned a NACK.
Cleared upon writing '1' into the **emit_start** bit of the I3CC CR1 register.

[15] **nack_target**
1 - an I3C/I2C address or a legacy I2C write returned a NACK.
Cleared upon writing '1' into the **emit_start** bit of the I3CC CR1 register.

[16] **read_terminated**
1 - an I3C read terminated by a Target earlier than expected (number of received bytes doesn't match the expected amount specified in TxFIFO memory).
Cleared upon writing '1' into the **emit_start** bit of the I3CC CR1 register.

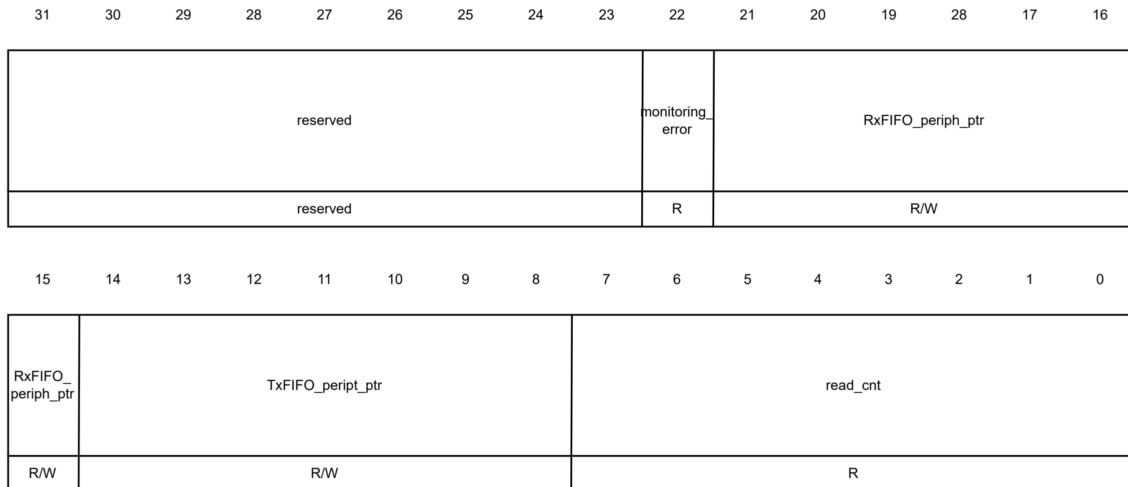

Figure 4.9: I3CC Status 2 register

Reset value: 0x0000 0000

[7:0] **read_cnt**

Current value of the read counter. If a read is terminated early by a Target, this register contains the amount of additional bytes that were expected to be received.

[14:8] **txfifo_periph_ptr**

The inner TxFIFO address pointer of the peripheral. Bits [13:8] contain the current address, whereas bit [14] is used to differentiate between TxFIFO full and TxFIFO empty cases.

[21:15] **rxrfifo_periph_ptr**

The inner RxFIFO address pointer of the peripheral. Bits [13:8] contain the current address, whereas bit [14] is used to differentiate between RxFIFO full and RxFIFO empty cases.

[22] **monitoring_error**

1 - the SDA line state differs from the expected. Signifies error type CE1 - *sda_in* differs from *sda_out* outside of Address Arbitration.

Cleared upon writing '1' into the **emit_start** bit of the I3CC CR1 register.

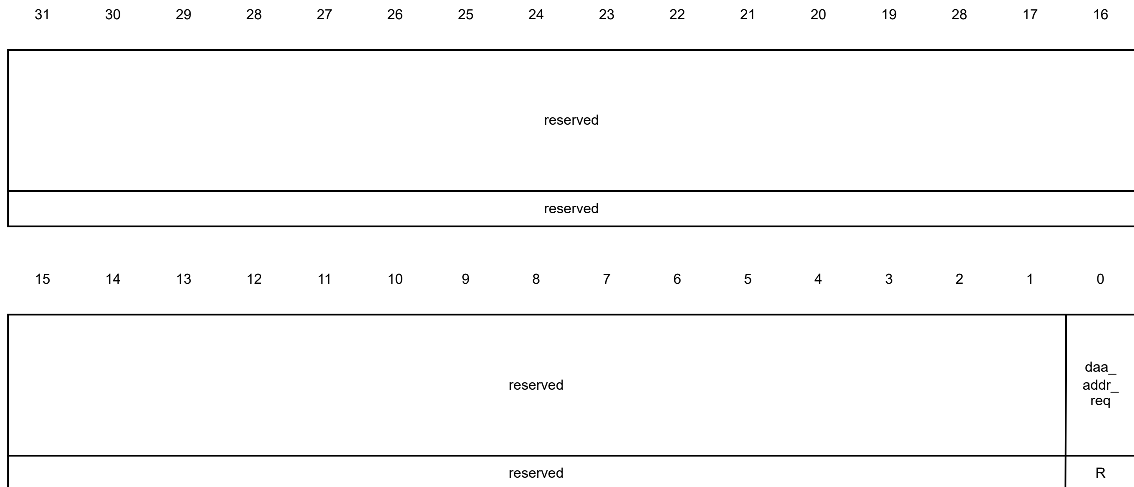

Figure 4.10: I3CC DAA register

Reset value: 0x0000 0000

[0] **daa__addr__req**

1 - the peripheral finished receiving the PID, BCR and DCR registers from a Target which won the arbitration and is requesting a valid address with an accompanying parity bit to be written into the TxFIFO memory.

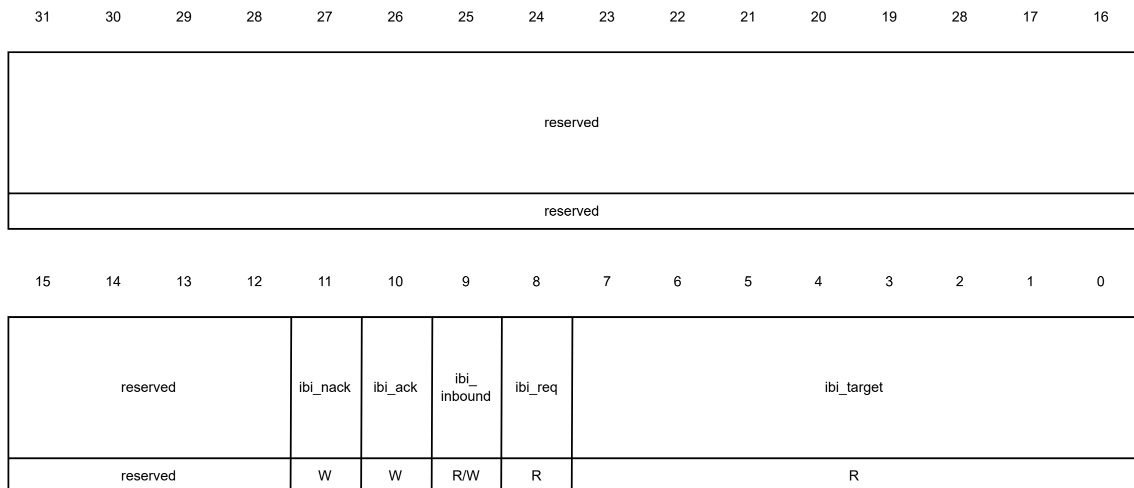

Figure 4.11: I3CC IBI register

Reset value: 0x0000 0000

[7:0] **ibi__target**

bits [7:1] - Address of a device which won the Address Arbitration round,

bit [0] - the received RnW bit.

[8] **ibi__req**

1 - Target generated an In-Band Interrupt request by pulling SDA low, generating a START condition.

[9] **ibi__inbound**
1 - Target won address arbitration.

[10] **ibi__ack**
W1 - ACK the In-Band Interrupt.
If the RnW bit is '1' (READ), then the next TxFIFO word contains the number of bytes to be read from the Target unless the next TxFIFO word is an address.

[11] **ibi__nack**
W1 - NACK the In-Band Interrupt.

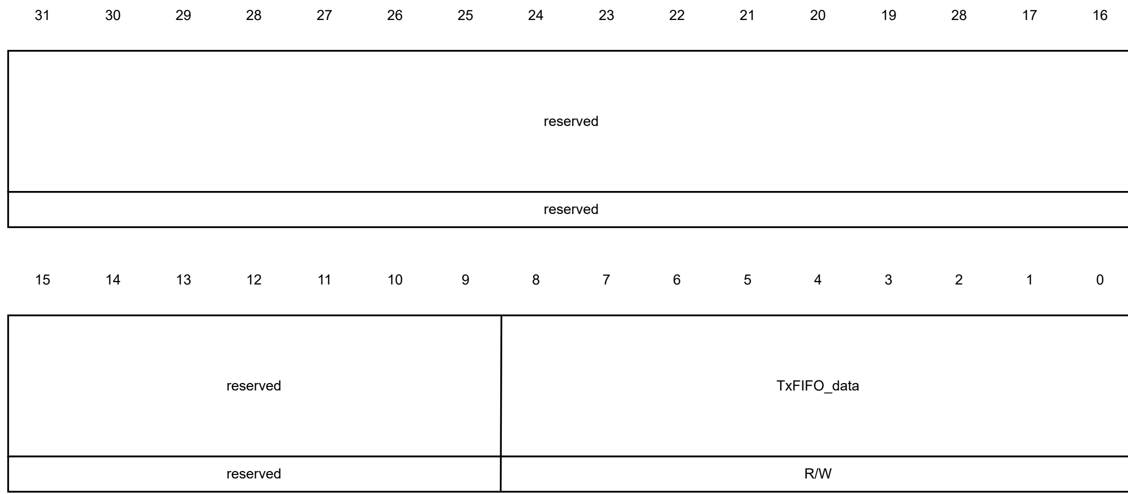

Figure 4.12: I3CC TxFIFO data register

Reset value: 0x0000 0000

[8:0] **txfifo__data**
bit [8] contains a *flag bit*, which marks that the current TxFIFO word contains an I2C/I3C address.
During an I2C/I3C **WRITE** operation, bits [7:0] contain data which will be transmitted onto the I3C bus.
During an I2C/I3C **READ** operation, bits [7:0] contain the number of bytes which is to be read from an addressed device. Value of "0" results in 1 byte being read.

31	30	29	28	27	26	25	24	23	22	21	20	19	28	17	16
reserved														TxFIFO_ ptr_incr_ write	TxFIFO_ ptr_incr_ read
reserved														R/W	R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved		TxFIFO_watermark						reserved	TxFIFO_ptr						
reserved		R/W						reserved	R/W						

Figure 4.13: I3CC TxFIFO ptr register

Reset value: 0x0003 0000

[6:0] **txfifo__ptr**

The TxFIFO address pointer used for access by the system (CPU). Bits [13:8] contain the current address, whereas bit [14] is used to differentiate between TxFIFO full and TxFIFO empty cases.

[13:8] **txfifo__watermark**

The TxFIFO watermark level. When the number of remaining unread words in the TxFIFO match this value, the system is notified of the TxFIFO reaching its watermark level.

[16] **txfifo__ptr__incr__read**

1 - the **txfifo__ptr** is automatically incremented upon a read from the *I3CC TxFIFO data* register.

[17] **txfifo__ptr__incr__write**

1 - the **txfifo__ptr** is automatically incremented upon a write into the *I3CC TxFIFO data* register.

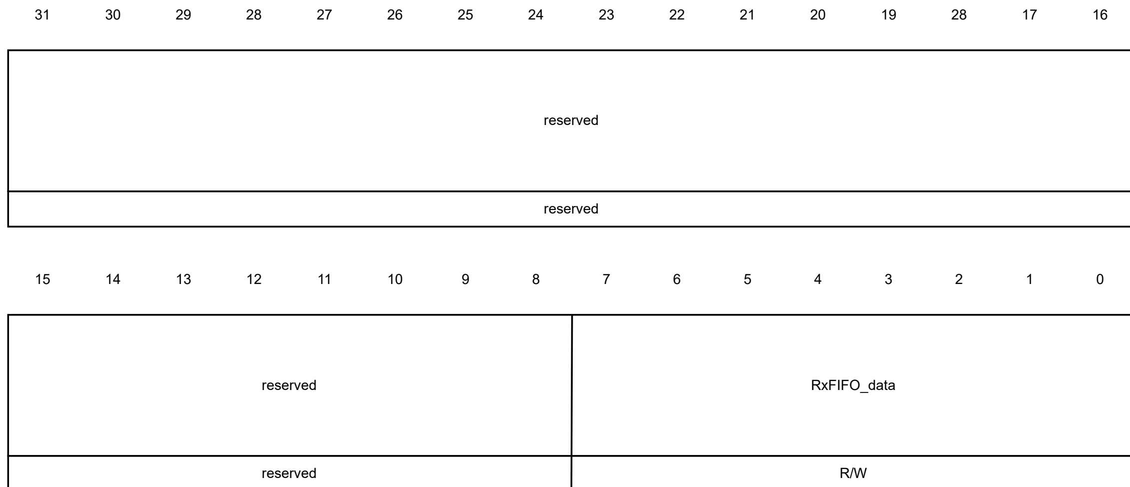

Figure 4.14: I3CC RxFIFO data register

Reset value: 0x0000 0000

[7:0] **rxfifo_data**
data read by the I3C Controller.

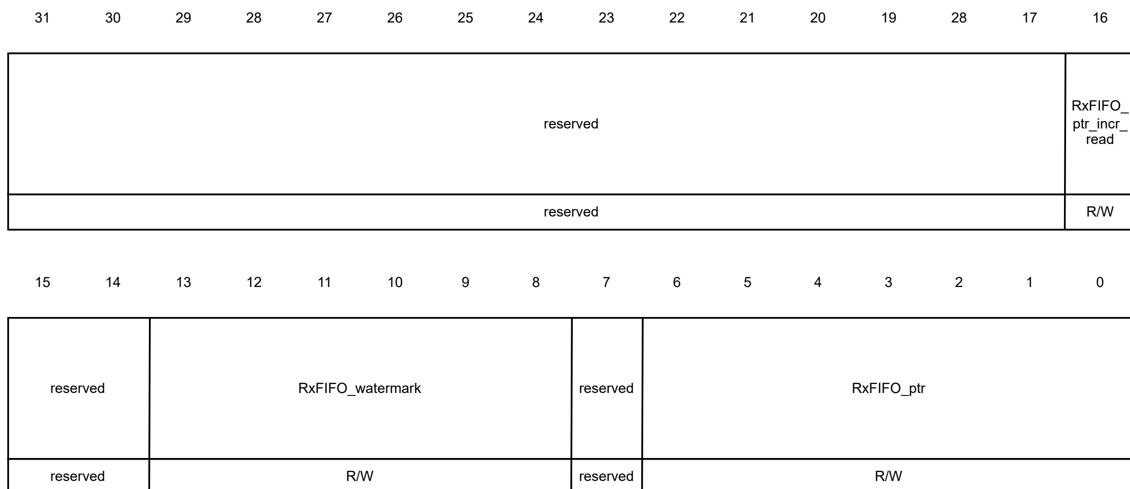

Figure 4.15: I3CC RxFIFO ptr register

Reset value: 0x0001 0000

[6:0] **rxfifo_ptr**
The RxFIFO address pointer used for access by the system (CPU). Bits [13:8] contain the current address, whereas bit [14] is used to differentiate between RxFIFO full and RxFIFO empty cases.

[13:8] **rxfifo_watermark**
The RxFIFO watermark level. When the number of remaining unread bytes in the RxFIFO match this value, the system is notified of the RxFIFO reaching its watermark level.

[16] **rxfifo_ptr_incr_read**
1 - the **rxfifo_ptr** is automatically incremented upon a read from the *I3CC RxFIFO data* register.

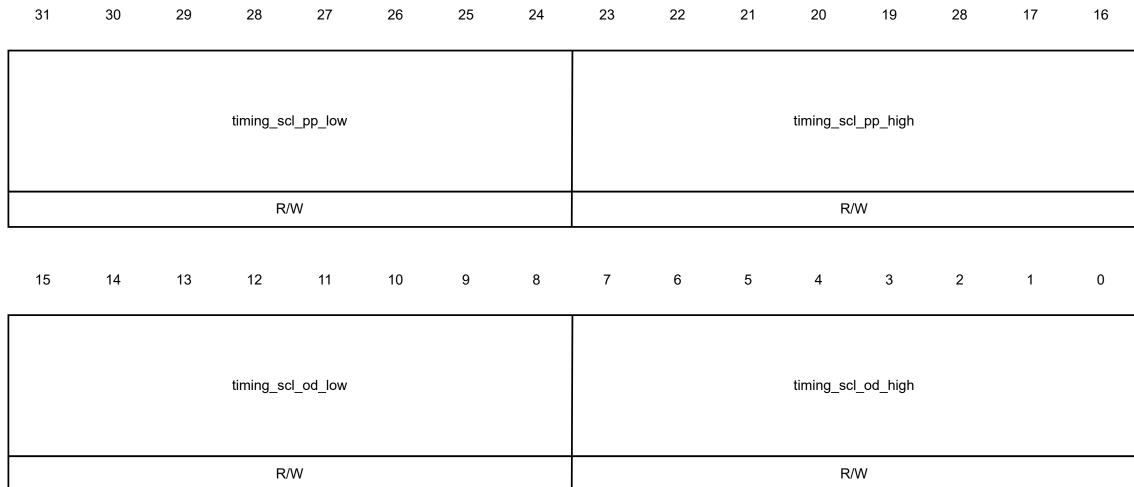

Figure 4.16: I3CC Timing 0 register

Reset value: 0x0000 0000

[7:0] **timing_scl_od_high**
The SCL HIGH period in Open Drain mode, shown on Figure 4.23.
Sets the t_{HIGH} time to $(N + 3) \times t_{\mathrm{SCL}}$.

[15:8] **timing_scl_od_low**
The SCL LOW period in Open Drain mode, shown on Figure 4.23.
Sets the t_{LOW} time to $(N + 3) \times t_{\mathrm{SCL}}$.

[23:16] **timing_scl_pp_high**
The SCL HIGH period in Push-Pull mode, shown on Figure 4.23.
Sets the t_{HIGH} time to $(N + 3) \times t_{\mathrm{SCL}}$.

[31:24] **timing_scl_pp_low**
The SCL LOW period in Push-Pull mode, shown on Figure 4.23.
Sets the t_{LOW} time to $(N + 3) \times t_{\mathrm{SCL}}$.

31	30	29	28	27	26	25	24	23	22	21	20	19	28	17	16
timing_sda_od_setup_falling								timing_sda_od_setup_rising							
R/W								R/W							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
timing_sda_pp_setup_falling								timing_sda_pp_setup_rising							
R/W								R/W							

Figure 4.17: I3CC Timing 1 register

Reset value: 0x0000 0000

[7:0] **timing__sda__pp__setup__rising**

The SDA setup time before SCL rising edge in Push-Pull mode, shown on Figure 4.24.

This value shall be larger than 0, but at maximum $timing_scl_pp_low-2$.

[15:8] **timing__sda__pp__setup__falling**

The SDA setup time before SCL falling edge in Push-Pull mode, shown on Figure 4.24.

This value shall be larger than 0, but at maximum $timing_scl_pp_high-2$.

[23:16] **timing__sda__od__setup__rising**

The SDA setup time before SCL rising edge in Open Drain mode, shown on Figure 4.24.

This value shall be larger than 0, but at maximum $timing_scl_od_low-2$.

[31:24] **timing__sda__od__setup__falling**

The SDA setup time before SCL falling edge in Open Drain mode, shown on Figure 4.24.

This value shall be larger than 0, but at maximum $timing_scl_od_high-2$.

Figure 4.18: I3CC Timing 2 register

Reset value: 0x0000 0000

[7:0] **timing__pattern__start**

Specifies the t_{SETUP} and t_{HOLD} timings for a START condition, shown on Figure 4.21.

t_{SETUP} = timing__pattern__start + 6

t_{HOLD} = timing__pattern__start + 2

[15:8] **timing__pattern__stop**

Specifies the t_{HOLD} timing for a STOP condition, shown on Figure 4.22.

t_{HOLD} = timing__pattern__stop - 1

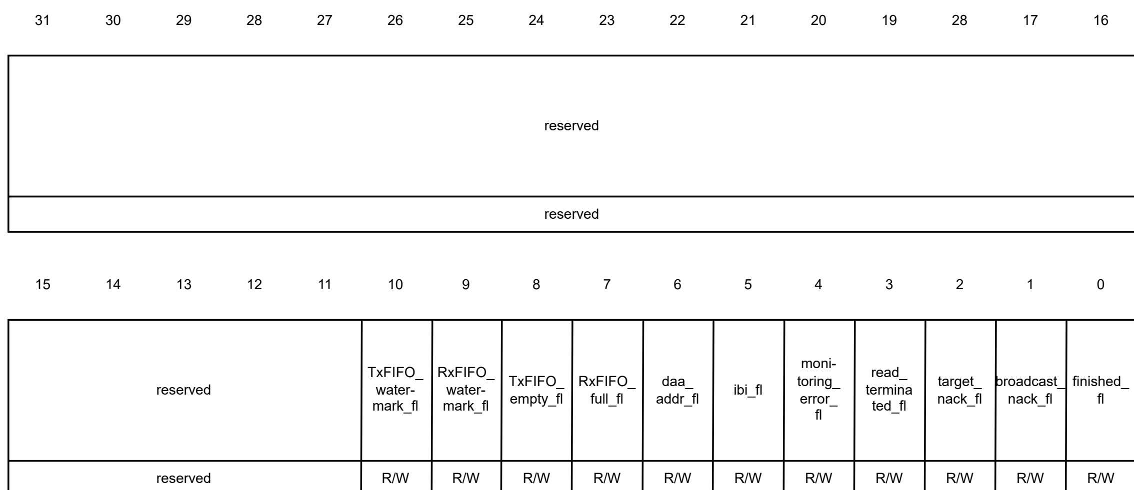

Figure 4.19: I3CC IRQ fl register

Reset value: 0x0000 0000

[0] **finished_fl**
Peripheral finished sending patterns / conditions / transmitting data.

[1] **broadcast_nack_fl**
The I3C Broadcast address (7'h7E) returned a NACK.

[2] **target_nack_fl**
An I3C/I2C address or a legacy I2C write returned a NACK.

[3] **read_terminated_fl**
an I3C read terminated by a Target earlier than expected (number of received bytes doesn't match the expected amount specified in TxFIFO memory).

[4] **monitoring_error_fl**
The SDA line state differs from the expected. Signifies error type CE1 - *sda_in* differs from *sda_out* outside of Address Arbitration.

[5] **ibi_fl**
Target generated an In-Band Interrupt request by pulling SDA low, generating a START condition or Target won Address Arbitration.

[6] **daa_addr_fl**
The peripheral finished receiving the PID, BCR and DCR registers from a Target which won the arbitration and is requesting a valid address with an accompanying parity bit to be written into the TxFIFO memory.

[7] **rxfifo_full_fl**
RxFIFO memory is full.

[8] **txfifo_empty_fl**
TxFIFO memory is empty.

[9] **rxfifo_watermark_fl**
RxFIFO memory has reached its watermark level.

[10] **txfifo_watermark_fl**
TxFIFO memory has reached its watermark level.

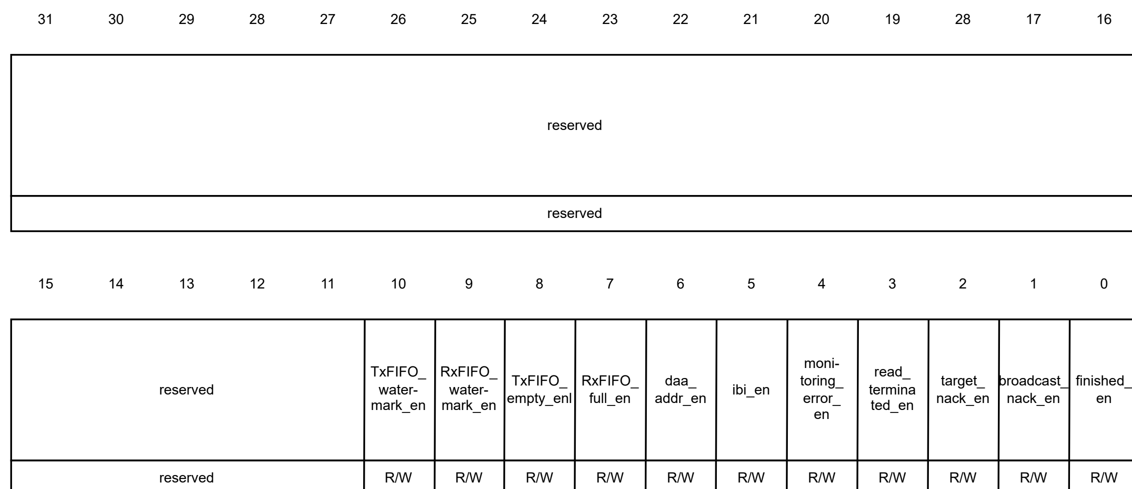

Figure 4.20: I3CC IRQ en register.

Reset value: 0x0000 0000. Uses same bit positions as the **I3CC IRQ fl** register.

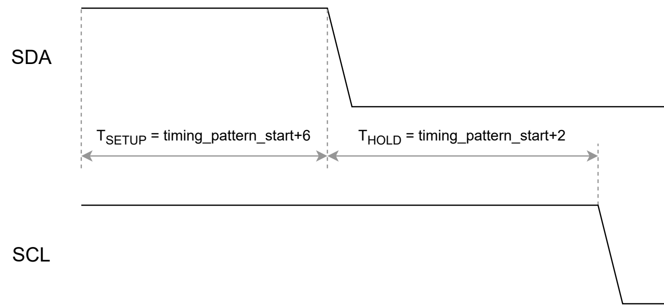

Figure 4.21: START condition timings.

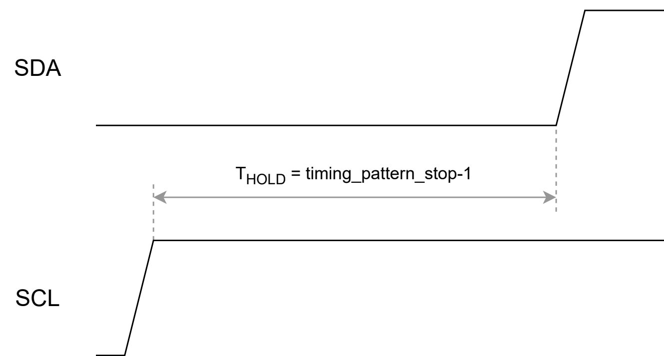

Figure 4.22: STOP condition timing.

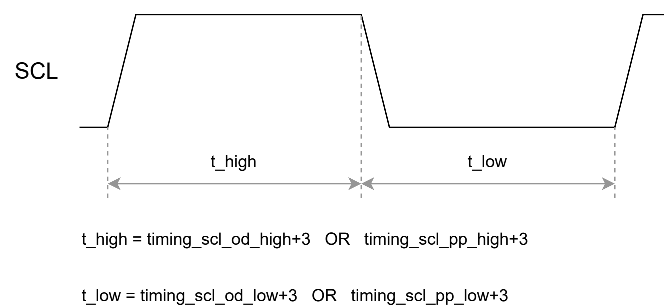

Figure 4.23: SCL timings. Push-Pull or Open Drain timings are selected automatically by the Controller.

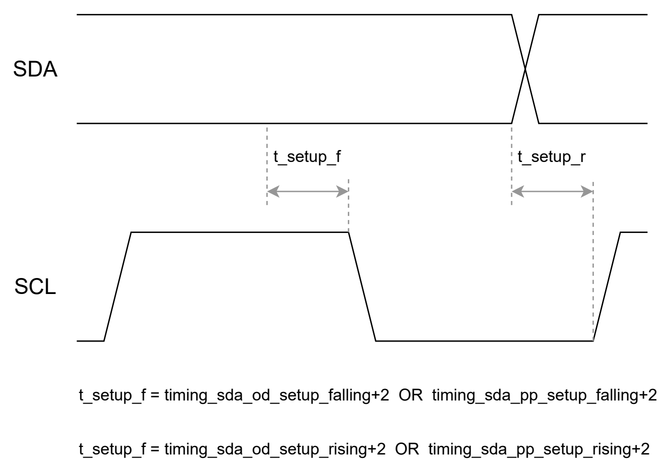

Figure 4.24: SDA setup timings. Push-Pull or Open Drain timings are selected automatically by the Controller.

4.8 Control block

The control block consists of the (main) Control FSM, three additional supportive FSM, and a read counter. The Control FSM oversees all functions of the peripheral and controls other state machines of the peripheral (Figure 4.25).

Figure 4.25: Diagram of the Control block

During the creation of the peripheral, a design decision needed to be made – the complexity of the Control FSM. Due to a lack of experience with creating complex designs and a gradual feature creep, the number of states in the FSM has overgrown, as the balance between the number of states, state machines, and the complexity of their transitions was not worked out (Figure 4.27). Fortunately, the transitions can be followed, as the state machine generally follows three phases per step - setup, execution, and termination.

Another drawback of the Control FSM is the time it retrieves the next TxFIFO word and stores read data bytes into the RxFIFO memory, which happens during *SCL HIGH*. This drastically increases the minimum *ck_sys_bus* frequency required for mixed I3C bus while running at SCL clocks higher than 1 MHz (I2C Fm+ speeds). The Control FSM transitions through 6 states, including its idle state, which causes the **+6** constant in the minimum *ck_sys_bus* calculation in section 3.4 *Clocks*.

Figure 4.26: SCL high delay caused by the Control FSM (the problematic FSM states marked in red).

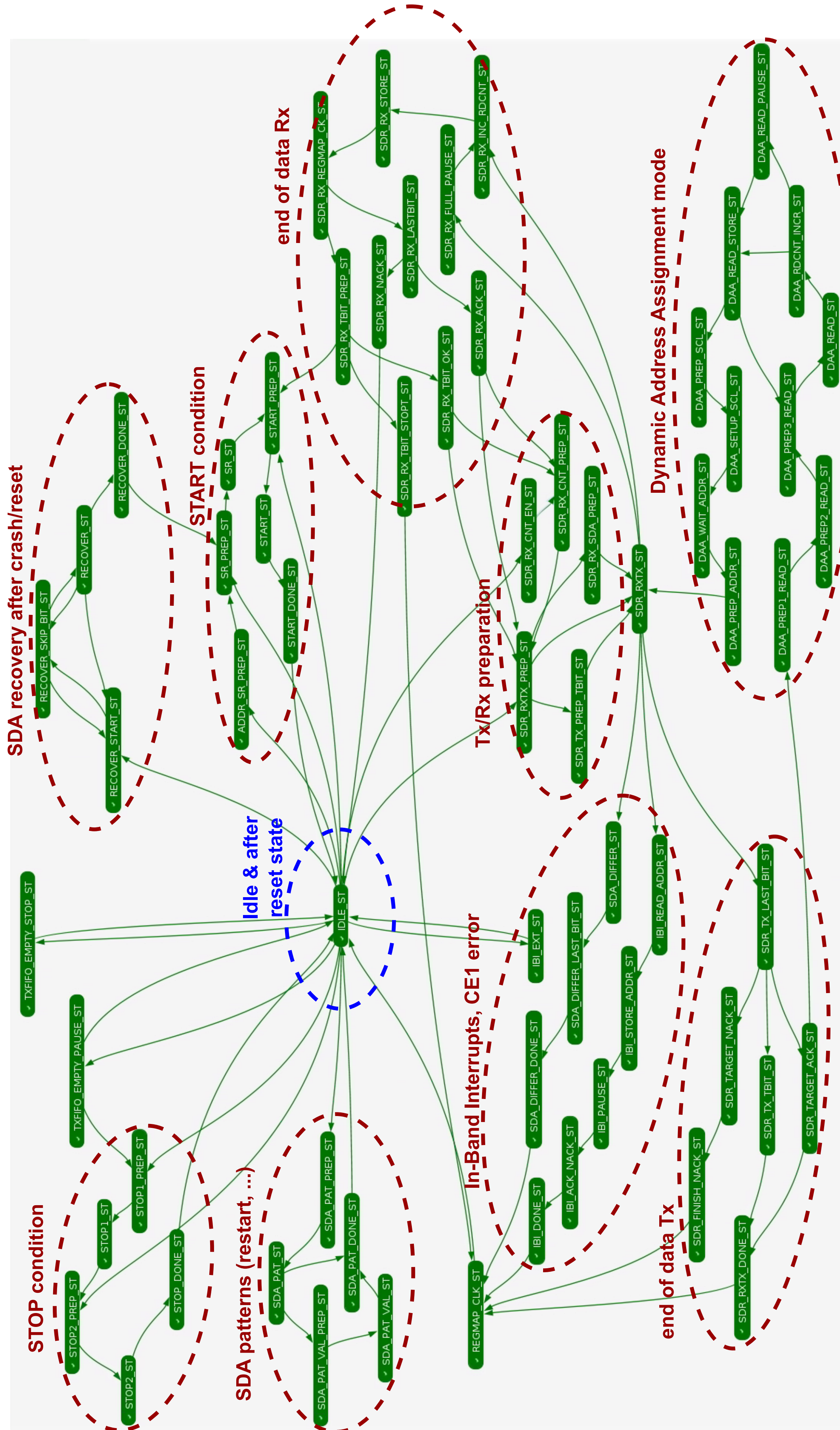

Figure 4.27: State diagram of the Control FSM.

4.9 Shift register

The shift register drives the *sda_out* and samples the *sda_in* signals. Resynchronization was not needed, as the controller actively drives the SCL line, therefore the rest of the I3C bus shall be synchronous to the SCL clock generated by peripheral[1]. During normal operation, the contents of the register are shifted before the rising SCL edge on *writes* and before the falling SCL edge on *reads* from the I3C bus.

The length of the shift register was originally intended to be 20 bits to allow sending a whole HDR-DDR word but was later shortened to 15 bits as the support for HDR-DDR transactions was dropped. The 15-bit length was chosen to allow the Controller to load the whole Restart pattern into the shift register. An alternative approach would be to rely on the CPU to load this pattern into the TxFIFO memory.

The register contains an additional D Flip-Flop (DFF) memory between its MSB bit and the serial output to prevent unwanted transitions of the SDA line while SCL is held high, which would result in unwanted repeated START or STOP conditions. Additionally, the output value of the inserted flip-flop can be changed independently of the shift register stored value using one of these control signals:

- set_out - sets the MSB to '1', therefore driving *sda_out* high,
- clr_out - clears the MSB to '1', therefore driving *sda_out* low,
- cpy_out - sets the MSB to *sda_in* state, copying the state of the bus.

These control signals are used by the main FSM for quick bus handoffs required during some I3C transmissions. An example of this is the SDA handoff during an ACK when writing to a target device, discussed in Section *2.5 SDA Handoffs*.

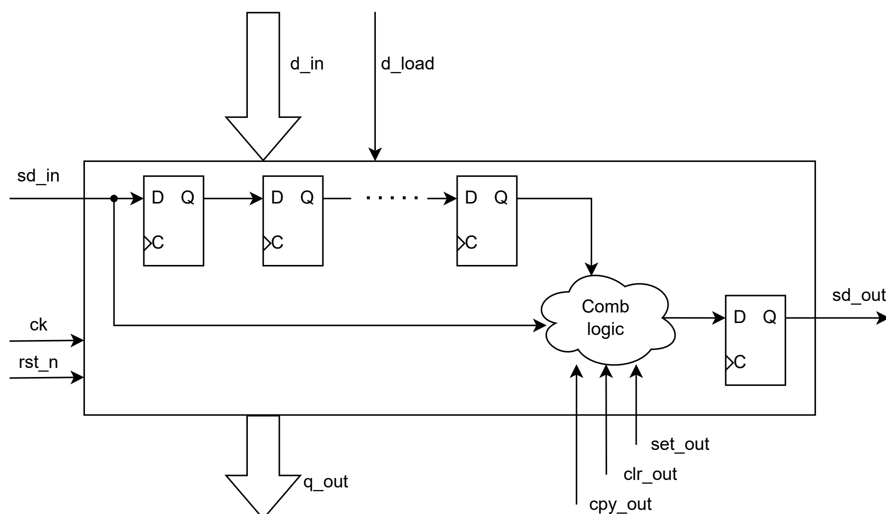

Figure 4.28: Conceptual diagram of the I3C shift register.

4.10 IBI shift register

Due to a limitation of the design, a secondary 8-bit shift register was necessary to read the arbitrated address on the I3C bus. This is caused by the design setting the sda_out bit before the SCL rising edge, but reading the sda_in value before the falling edge. Switching between reading and writing data using the main shift register caused the bit which resulted in the controller losing the address arbitration to be read twice. The loss of the address arbitration can occur on any bit and therefore no simple change to the main shift register would fix this, so a second shift register was incorporated into the design, which only reads the sda_in line.

[1] An exception to this would be a misbehaving device on the bus.

4.11 SCL clock generation

The SCL clock generation is controlled by the *SCLGEN_FSM* block to allow for easier clock stalling and overall supervision by the (main) Control FSM. The SCL generator creates the SCL clock as well as shift pulses for the SDA shift register.

The SCL clock is generated with a variable duty cycle dependent on the *tim_high* and *tim_low* input signals. These signals are managed by the Control FSM, which handles switching between open-drain and push-pull timings. These signals are also used for the START and STOP condition lengths.

The SDA shift pulses are generated in advance of the changing SCL edge to either:

a) allow the SDA line to stabilize when writing or

b) read the SDA line when the SCL was held high.

The clock generator can also be forced to hold the SCL at a constant value while still generating the SDA shift pulses. This behavior is used for generating (repeated) START and STOP conditions or for parking the SCL line when necessary, such as clock stalling during the DAA procedure.

SCL_LOW_ST
IDLE_ST
SCL_HIGH_ST

Figure 4.29: State diagram of the SCL Gen FSM

The designed SCL clock generator adds a delay of three *ck_i3c* clock cycles into every generated SCL timing, as the FSM transitions through the **IDLE_ST** state for every transition of the SCL clock and the internal counter sets its *done* signal one clock cycle after reaching zero (Figure 4.30, turquoise color).

Baseline = 220,560ns
Cursor-Baseline = 120ns
Name | Cursor | 220,620ns | 220,640ns | 220,660ns
rst_n 1
ck_i3c 1
sda_out 0
scl_out 0
next_st SDR_RXTX_ST
curr_st SDR_RXTX_ST
curr_st SCL_LOW_ST IDLE_ST SCL_LOW_ST
next_st SCL_LOW_ST IDLE_ST SCL_LOW_ST
i3c_sclgen_done 0
cnt_i 'h 08 02 01 00 09

Figure 4.30: The delays caused by the SCL Generator FSM. The counter stays at 0 for three cycles before the *scl_out* signal changes.

4.11.1 Transmission Control

The Transmit FSM controls the flow of data between the I3C bus and the shift register, as its function is to make the peripheral transmit and receive a certain number of bits. The entity consists of a small FSM, an accompanying bit counter, and some additional logic for signaling when the bit counter reaches the second to last and the last value, which is used by the Control FSM.

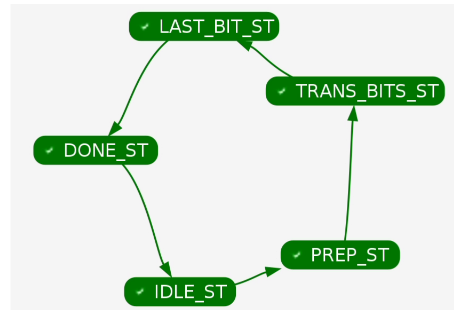

Figure 4.31: State diagram of the Transmit FSM

4.11.2 Read Counter

Whereas the Transmit FSM manages the transmission of bits, the read counter manages the receiving of bytes by the peripheral. When an address is sent onto the I3C bus with the RnW bit set to *read* which is not the broadcast address (7'h7E), then the counter is loaded with a value stored in the next TxFIFO word unless it is an address. This loaded word contains how many bytes are expected to be read by the Controller. The peripheral then attempts to read the number of bytes specified in the TxFIFO memory, with the exception of "0", which reads 1 byte.

4.11.3 Elemental Use Cases

This section shows two short examples of transmitting and receiving 3 bytes of data to/from a Target device. The listed codes utilize the included *i3cc.h* file, which defines the register addresses and the means of accessing them.

```
1  #include "i3cc.h"
2  #define addrFlag  (0b1 << 7)
3  #define RnW_R     0b1
4  #define RnW_W     0b0
5  #define c_broadcastAddr 0x7E
6  #define c_targetAddr    0x4F
7  // Filling the TxFIFO memory
8  // setting MSB to '1' makes the peripheral treat the TxFIFO word as an address
9  I3CC->RegI3CCTxFIFOdata.r32 = (0b1 << 7) | (c_broadcastAddr << 1) | RnW_W;
10 I3CC->RegI3CCTxFIFOdata.r32 = (0b1 << 7) | (c_targetAddr << 1)    | RnW_W;
11 for(int i=0; i<3; i++) {
12   I3CC->RegI3CCTxFIFOdata.r32 = data_to_transmit[i];
13 }
14
15 // send START, transmit data from TxFIFO memory and send STOP
16 I3CC->RegI3CCCR1.r32 |= EMIT_START(1) | TRANSMIT_DATA(1) | EMIT_STOP(1);
17
18 // wait until the STOP has been sent
19 // (also covers waiting for START and data transmitted)
20 while( GET_EMIT_STOP_STS(I3CC->RegI3CCCR1.r32) );
```

Listing 4.1: C code example of transmitting 3 bytes of data to a Target device.

```
1  #include "i3cc.h"
2  #define addrFlag  (0b1 << 7)
3  #define RnW_R     0b1
4  #define RnW_W     0b0
5  #define c_broadcastAddr 0x7E
6  #define c_targetAddr    0x4F
7  // Filling the TxFIFO memory
8  // setting MSB to '1' makes the peripheral treat the TxFIFO word as an address
9  I3CC->RegI3CCTxFIFOdata.r32 = (0b1 << 7) | (c_broadcastAddr << 1) | RnW_W;
10 I3CC->RegI3CCTxFIFOdata.r32 = (0b1 << 7) | (c_targetAddr << 1)    | RnW_R;
11 I3CC->RegI3CCTxFIFOdata.r32 = 3;  // the "3" sets the number of bytes to be
       received
12
13 // send START, transmit data from TxFIFO memory and send STOP
14 I3CC->RegI3CCCR1.r32 |= EMIT_START(1) | TRANSMIT_DATA(1) | EMIT_STOP(1);
15
16 // wait until the STOP has been sent
17 // (also covers waiting for START and data transmitted)
```

```
18 while( GET_EMIT_STOP_STS(I3CC->RegI3CCCR1.r32) );
19
20 // read the received data
21 for(int i=0; i<3; i++) {
22   received_data[i] = I3CC->RegI3CCRxFIFOdata.r32;
23 }
```

Listing 4.2: C code example of receiving 3 bytes of data from a Target device.

Chapter 5

Verification

5.1 Verification Plan

Verification of a design is the second most important task in a design, given that no designer can create the perfect system on his/her first try nor think of every edge case. The phrase “What is not verified does not work” is often true during verification (or its lack of).

The RISC-V environment uses the Universal Verification Methodology (UVM), which is a standardized methodology for verifying integrated circuit designs. The parts that had to be focused on were implementing the *I3C Target Agent*, the *I3C Interface* itself, and their integration. For running the simulations, the Cadence Xcelium simulator was used for compilation and simulation of the design and its test environment.

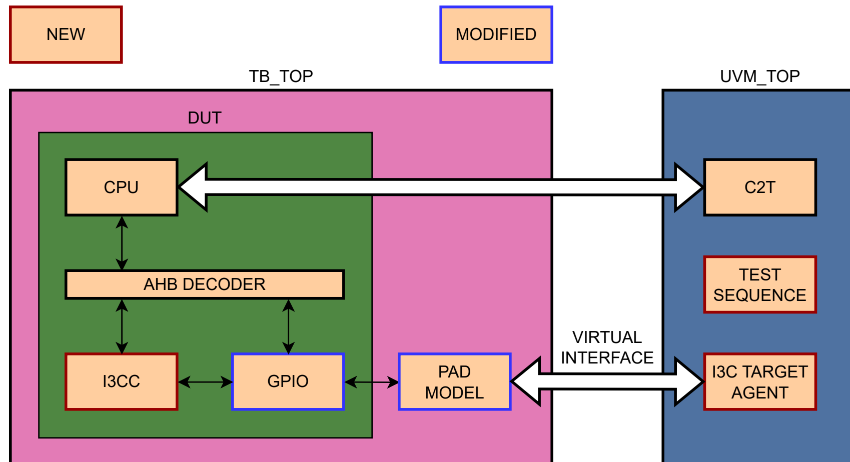

Figure 5.1: Simplified test bench structure with changes compared to the RISC-V system

The test environment consists of the DUT encapsulated in TB_TOP, virtual I3C interface, and UVM_TOP.

The DUT is the RISC-V system with the I3C Controller peripheral inside it. The DUT contains additional blocks and peripherals, but these were omitted in the Test bench structure (Figure 5.1), as they are not relevant for the I3C Controller verification.

The I3C Interface is fairly straightforward, as it models the behavior on the shared SCL and SDA lines, but it was also needed to model the difference between push-pull and open-drain modes of the lines, as one results in a short circuit scenario while the other does not. This is done by separating each device’s actions into *driving LOW* and *driving HIGH* and checking if two (or more) devices create a conflict on the bus.

The pad model contains delays for the connections between the internal circuitry of a chip and its external environment. The applied pad model is designed to be weaker than those of the connected devices in order to detect collisions on the bus. It is capable of switching between open drain and push-pull driving and switching its pull-up structure on or off. Its relevant delay values are listed in Table 5.1, which were chosen from a previously used I2C Controller peripheral, but with modifications to remove glitching when switching between open drain and push-pull outputs.

Table 5.1: Relevant delays of the Pad Model

Delay name	Value	Description
OD_D2PAD	119 ns	Open Drain delay for data out -> pad
D2PAD	10.2 ns	Push-Pull delay for data out -> pad
PAD2Q	7 ns	Delay for pad -> data in
PU_DLAY	10.2 ns	Pull-Up enable/disable delay
OE2PAD	10.2 ns	Delay for switching Open Drain <-> Push-Pull

5.2 I3C Target Agent

Given the relatively complicated nature of I3C transactions, a generic Target was designed, for which the order of transactions and their type does not matter. The Target Agent runs in an infinite loop, where it first checks whether it is active or not. If it is inactive, it then waits until a START condition occurs and starts multiple forks (processes), each responsible for a different type of transaction (Figure 5.2).

This division into multiple forks that run in parallel was done so as to not implicitly set the flow and types of transactions after each other (i.e. addresses, private read/writes, and CCC byte transfers can occur in any order). This approach is not optimal, as multiple processes are spawned per I3C transaction per every Target Agent, but it is fast to develop and each process gets killed after a STOP condition occurs.

The *START* and *STOP* forks are the simplest. *START* waits until a START condition occurs and sets the Target Agent to read an address in the next iteration of the main loop. *STOP* is similar, waiting until a STOP condition occurs, but setting the Target Agent to be inactive.

Each of the forks follows these three steps:

1. check if this fork is to be active. If not - enter an infinite loop,
2. execute,
3. prepare Target Agent for the next iteration of its main loop.

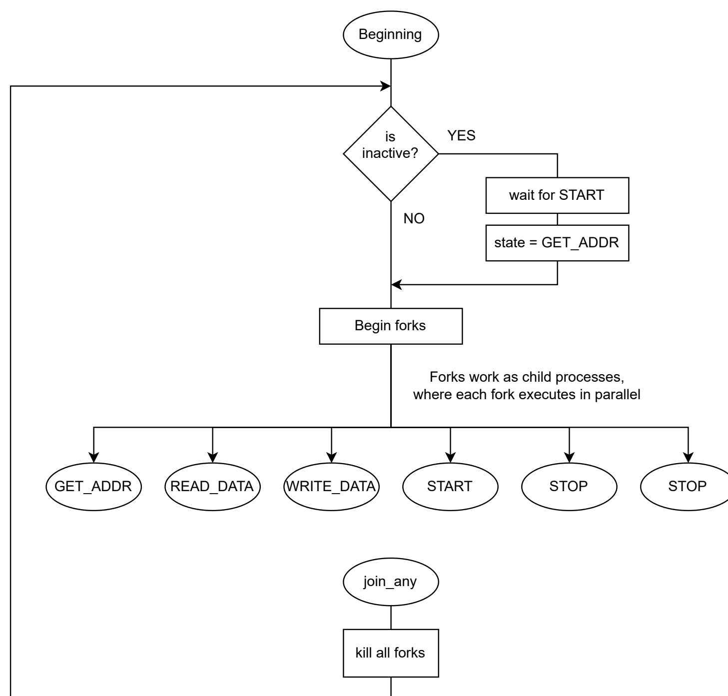

Figure 5.2: I3C Target Agent's main loop.

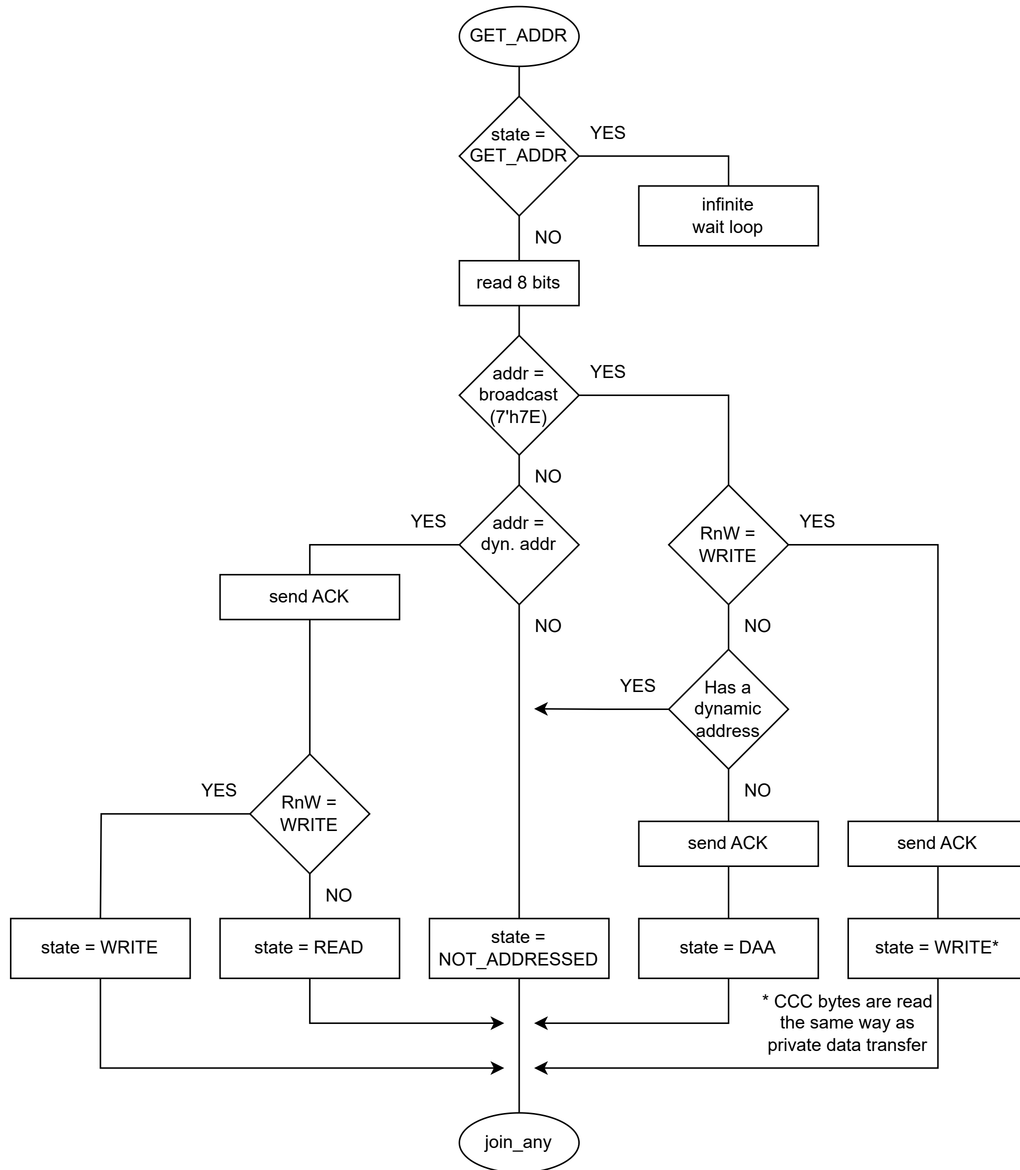

Figure 5.3: I3C Target Agent's Get Address fork.

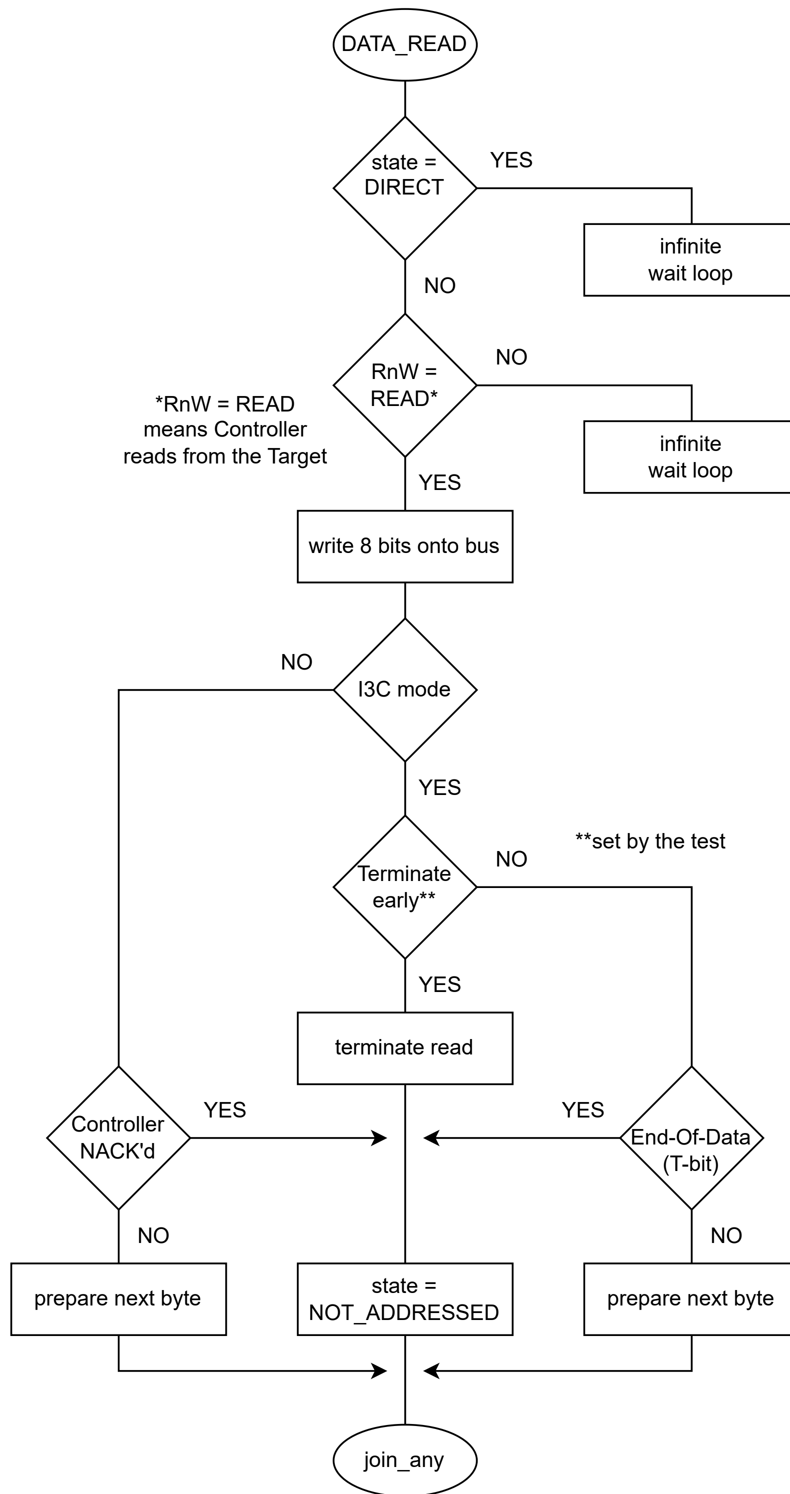

Figure 5.4: I3C Target Agent's Data Read fork (Controller reads from Target).

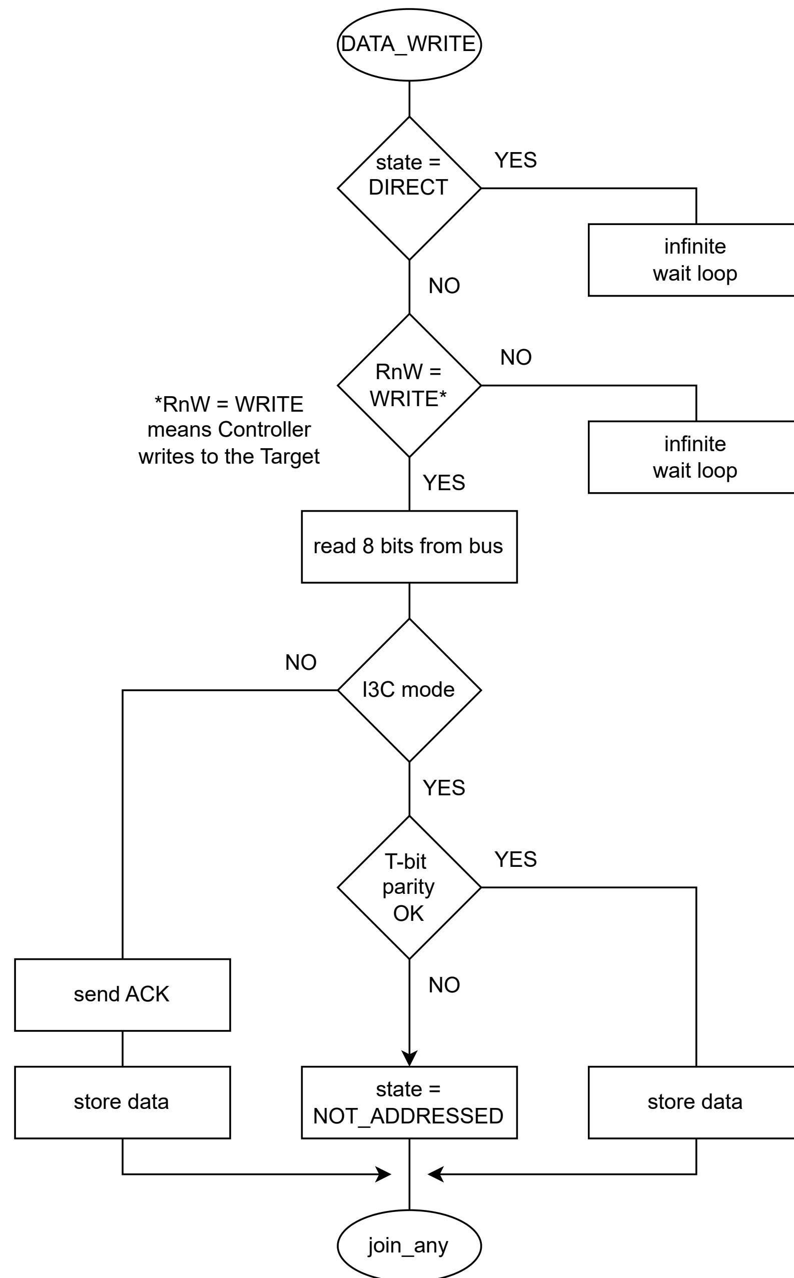

Figure 5.5: I3C Target Agent's Data Write fork (Controller writes to Target).

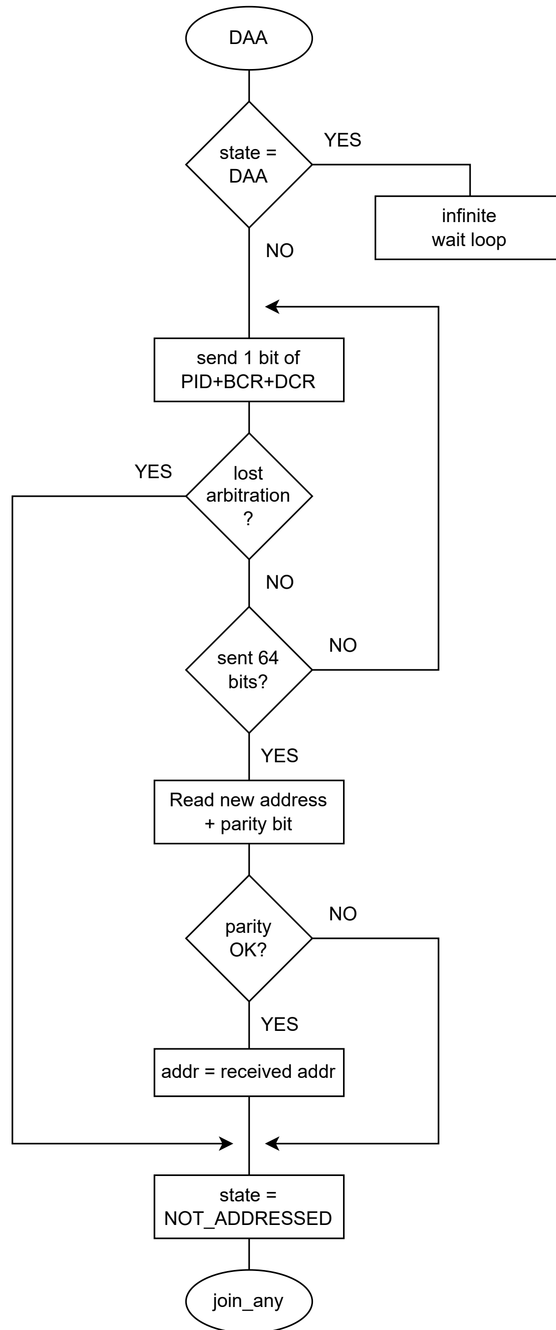

Figure 5.6: I3C Target Agent Dynamic Address Assignment fork

5.3 Use cases

This section outlines how the peripheral is anticipated to be accessed by the system in order to successfully transfer data between the I3C Controller peripheral and Target located on the I3C bus. Listed use cases outline their purpose and preconditions, followed by how the software is expected to control the peripheral.

5.3.1 Initial configuration

Device mode	First configuration & enable
Preconditions	Peripheral was previously disabled
Trigger	Software
Goal	Enable the peripheral
Success end state	I3C controller enabled and in control over the I3C bus
Failed end state	I3C controller disabled and/or not in control over the I3C bus
Notes	Configuration done in every test, SDA recovery tested in **test__i3c__errs**

Step	Hardware	Software
1	Held in soft reset	Configures the I3CC Timing 0, Timing 1 and Timing 2 registers
2	Held in soft reset	Writes '1' into the *enable* bit of the I3CC CR2 register
3	SDA recovery	If the system was unexpectedly reset or the SDA line might be controlled by an another device, then the *assert__ctrl* bit of the I3CC CR1 register shall be set to '1'
4	SDA recovery	Polls *assert__ctrl__sts* bit of I3CC stat1 register or waits for an interrupt if *finished__en* bit of the I3CC IRQ en register is '1'
5a	SDA recovered	Writes '1' into the *emit__stop* bit of the I3CC CR1 register
6	Sending STOP	Polls *emit__stop__sts* bit of I3CC stat1 register or waits for an interrupt if *finished__en* bit of the I3CC IRQ en register is '1'
7	Idle	Controller has control over the SDA line, recovery finished
5b	Idle	Controller writes '0' into the *enable* bit of the I3CC CR2 register if the peripheral has not finished at least 20 times the configured t_{SCL_OD} period, as the peripheral was not able to regain control of the SDA line. Next steps are out of the scope of this document, such as resetting the devices connected to the bus in an another way

5.3.2 Basic I3C SDR write

Device mode	I3C SDR
Preconditions	Peripheral enabled and configured
Trigger	Software
Goal	Send N data bytes to a Target
Success end state	N data bytes transmitted
Failed end state	NACK from broadcast/Target address
Notes	Verified in **test_i3c_rw** and **test_i3c_rw_extra**

Step	Hardware	Software
1	Idle	Writes the I3C broadcast address with a flag bit of '1' into the TxFIFO data register
2	Idle	Writes the Target address with a flag bit of '1' into the TxFIFO data register
3	Idle	Writes N data bytes into the TxFIFO data register
4	Idle	Writes '1' into the *emit_start*, *transmit* and *emit_stop* bits of the I3CC CR1 register
5	Sends START, addresses, data and STOP	Polls *emit_stop_sts* bit of I3CC stat1 register or waits for an interrupt if *finished_en* bit of the I3CC IRQ en register is '1'
6	Idle	Data sent to the Target

5.3.3 Basic I3C SDR read

Device mode	I3C SDR
Preconditions	Peripheral enabled and configured
Trigger	Software
Goal	Receive N data bytes from a Target
Success end state	N data bytes received
Failed end state	NACK from broadcast/Target address or Target ends read earlier than expected
Notes	Verified in **test_i3c_rw** and **test_i3c_rw_extra**

Step	Hardware	Software
1	Idle	Writes the I3C broadcast address with a flag bit of '1' into the TxFIFO data register
2	Idle	Writes the Target address with a flag bit of '1' and into the TxFIFO data register
3	Idle	Writes how many bytes to receive from a Target (N) into the TxFIFO data register
4	Idle	Writes '1' into the *emit_start*, *transmit* and *emit_stop* bits of the I3CC CR1 register
5	Sends START, addresses, data and STOP	Polls *emit_stop_sts* bit of I3CC stat1 register or waits for an interrupt if *finished_en* bit of the I3CC IRQ en register is '1'
6	Idle	Reads received data from the RxFIFO data register

5.3.4 Legacy I2C write

Device mode	Legacy I2C
Preconditions	Peripheral enabled and configured
Trigger	Software
Goal	Send N data bytes to a Target
Success end state	N data bytes received
Failed end state	NACK from Target address or Target returns NACK during data write
Notes	Verified in **test_i3c_rw** and **test_i3c_rw_extra**

Step	Hardware	Software
1	Idle	Writes the Target address with a flag bit of '1' and into the TxFIFO data register
2	Idle	Writes N data bytes into the TxFIFO data register
3	Idle	Writes '1' into the *legacy_mode*, *emit_start*, *transmit* and *emit_stop* bits of the I3CC CR1 register
4	Sends START, addresses, data and STOP	Polls *emit_stop_sts* bit of I3CC stat1 register or waits for an interrupt if *finished_en* bit of the I3CC IRQ en register is '1'
6	Idle	Data sent to the Target

5.3.5 Delayed I3C SDR read/write transaction

Device mode	I3C SDR
Preconditions	Peripheral enabled and configured
Trigger	Software
Goal	Send/receive N bytes to/from a Target
Success end state	N data bytes transmitted
Failed end state	NACK from broadcast/Target address or Target ends read earlier than expected
Notes	Necessary if one of the the **ENTAS1** through **ENTAS5** was previously sent to the addressed Target. Only steps 4, 5 and 6 differ from normal data Tx/Rx.

Step	Hardware	Software
1	Idle	Writes the I3C broadcast address with a flag bit of '1' into the TxFIFO data register
2	Idle	Writes the Target address with a flag bit of '1' into the TxFIFO data register
3a	Idle	Writes N data bytes into the TxFIFO data register
3b	Idle	Writes how many bytes to receive from a Target (N) into the TxFIFO data register

Step	Hardware	Software
4	Idle	Writes '1' into the *emit_start* bit of the I3CC CR1 register
5	Sends START	Polls *emit_stop_sts* bit of I3CC stat1 register or waits for an interrupt if *finished_en* bit of the I3CC IRQ en register is '1'
6	Idle	Waits until sufficient time elapses before the Target device can be accessed
7	Sends addresses, data and STOP	Writes '1' into the *transmit* and *emit_stop* bits of the I3CC CR1 register
8a	Idle	Data sent to the Target or reads received data from the RxFIFO data register

5.3.6 Broadcast CCC transfer

Device mode	I3C SDR
Preconditions	Peripheral enabled and configured
Trigger	Software
Goal	Send a broadcast CCC
Success end state	Broadcast CCC sent
Failed end state	NACK from the broadcast address
Notes	Verified in **test_i3c_daa**

Step	Hardware	Software
1	Idle	Writes the I3C broadcast address with a flag bit of '1' into the TxFIFO data register
2	Idle	Writes the CCC byte into the TxFIFO data register, optionally followed by additional data bytes if required by the CCC
3	Idle	Writes '1' into the *emit_start*, *transmit* and *emit_stop* bits of the I3CC CR1 register
4	Sends START, addresses, data and STOP	Polls *emit_stop_sts* bit of I3CC stat1 register or waits for an interrupt if *finished_en* bit of the I3CC IRQ en register is '1'
5	Idle	Broadcast CCC sent

5.3.7 Dynamic Address Assignment

Device mode	I3C SDR
Preconditions	Peripheral enabled and configured
Trigger	Software
Goal	Assign dynamic addresses to Targets using the **ENTDAA** CCC
Success end state	Assigned addresses to all Targets
Failed end state	Not all Targets have assigned addresses
Notes	Verified in **test_i3c_daa**

Step	Hardware	Software
1	Idle	Clears the *transmit_stop_empty* bit of the I3CC CR2 register
2	Idle	Writes the I3C broadcast address with a flag bit of '1' into the TxFIFO data register
3	Idle	Writes the **ENTDAA** CCC byte into the TxFIFO data register
4	Idle	Writes the I3C broadcast address with a flag bit of '1' and RnW='1' into the TxFIFO data register
5	Idle	Writes '1' into the *emit_start* and *transmit* bits of the I3CC CR1 register
6	Sends START, stored FIFO bytes and enters DAA procedure	Polls *daa_addr_req* bit of I3CC DAA register or waits for an interrupt if *daa_addr_en* bit of the I3CC IRQ en register is '1'
7	Reads PID+BCR+DCR registers	Polls *daa_addr_req* bit of I3CC DAA register or waits for an interrupt if *daa_addr_en* bit of the I3CC IRQ en register is '1'
8	Awaiting address	Reads the PID+BCR+DCR register values from RxFIFO memory and writes the to-be assigned address to the TxFIFO data register with a valid parity bit on the LSB position
9	Sends address	Repeats steps 7,8,9 until the expected number of Targets have their addresses assigned
10	Idle	Writes '1' into the *emit_stop* bit of the I3CC CR1 register
11	Sends STOP	Polls *emit_stop_sts* bit of I3CC stat1 register or waits for an interrupt if *finished_en* bit of the I3CC IRQ en register is '1'
12	Idle	All Targets have their dynamic addresses assigned

5.4 Tests

The goal of these tests is to (ideally) achieve full code coverage, i.e. every line of the RTL code was exercised during a simulation. To achieve this, these tests were prepared:

test_i3c_rw

Basic read/write sequences. The test is divided into four parts:

1. Test sends configuration (Target's address, RW data lengths) to the CPU,
2. test sends write data to the CPU, checks what Target Agent has read,
3. test configures Target Agent for data read,
4. CPU reads data from Target Agent, sends it to the test to validate received data.

test_i3c_rw_extra

The transaction byte length exceeds the lengths of the TxFIFO/RxFIFO memories during read/write sequences. This is achieved by writing/reading into/from the memories while a transaction is ongoing. Additionally disables the peripheral between read/write to test the soft reset and sends multiple I2C or I3C transactions after each other.

test_i3c_errs

Sets the Target Agent to:

1. NACK the I3C Broadcast address,

2. NACK an unrelated address (not assigned to the Target Agent) and

3. end read by Controller in I3C mode using T-bit

while the CPU is attempting to communicate with the Target Agent. After this, the test drives the SDA line while the I3C Controller attempts bus recovery in order to regain control of the SDA line.

test_i3c_daa

The Controller sends ENTDAA CCC and enters the Dynamic Address Assignment procedure. In total, the CPU attempts to assign an address three times:

1. with wrong parity bit (Target NACKs),

2. with correct parity bit (Target ACKs) and

3. with all Targets already having an address assigned (broadcast NACK'd).

test_i3c_patterns

Software manually executes each pattern/condition, such as START, STOP, restart, etc. Used to fulfill code coverage and to check whether the patterns are generated correctly.

test_i3c_ibi

Tests the capability of the I3C Controller to detect an In-Band Interrupt and its ability to send a response based on the instructions sent by the CPU.

5.5 Coverage

The code coverage was collected during the implementation of features, debugging, and final verification of the peripheral in order to check for uncovered parts of the design as well as for unnecessary expressions and signals. To help with managing the code coverage, a few scripts were written to assist with launching multiple tests after one another and merging their collected coverages together into a single coverage database, which was then inspected.

The collected code coverage metrics consist of these metrics, which are combined into a single number:

1. block coverage, measuring which blocks of code, such as begin-end or if-else statements, have been entered,

2. statement coverage, measuring the percentage of RTL code statements (lines) that have been executed,

3. expression coverage, evaluating possible combinations of operands in a expression,

4. toggle coverage, tracking the transitions of signals and

5. FSM coverage, which tracks all FSM states and the transitions between them.

The toggle coverage was only measured on the interface of the I3C Controller peripheral, i.e. its input and output signals. This was done to make sure all of the top interface signals were tested during simulations.

The final code coverage reached a total coverage of 96.62 % (Figure 5.7), with the lowest metrics being expression and toggle (Figure 5.8).

Name	Overall Average Grade	Overall Covered
(no filter)	(no filter)	(no filter)
i_i3c_controller	96.62%	5477 / 5621 (97.44%)
i_rst_gen	100%	3 / 3 (100%)
i_ck_gen	97.92%	58 / 60 (96.67%)
i_ctrl	94.31%	1426 / 1451 (98.28%)
i_regmap_wrapper	87.6%	3813 / 3920 (97.27%)
scl_resync	100%	7 / 7 (100%)
sda_resync	100%	7 / 7 (100%)

Figure 5.7: Code coverage of the peripheral by its blocks.

Details | i_i3c_controller

Metrics Source Attributes

Ex UNR Name	Overall Average Grade	Overall Covered
Overall	96.62%	5477 / 5621 (97.44%)
Code	96.6%	5295 / 5439 (97.35%)
Block	98.45%	2584 / 2602 (99.31%)
Statement	99.06%	2331 / 2346 (99.36%)
Expression	95.82%	2625 / 2741 (95.77%)
Toggle	89.58%	86 / 96 (89.58%)
FSM	100%	182 / 182 (100%)

Figure 5.8: Total code coverage of the peripheral by metric type.

The code coverage had waivers applied to it, which are predefined exceptions to ignore (waive) some of the uncovered parts of the design. The created waivers apply exceptions to:

1. signals or ports that have a constant value assigned to them,
2. impossible to cover expressions,
3. signals or ports not used by the system,
4. by design unreachable expressions,
5. by design unreachable statements, and
6. assertions written in VHDL.

Examples of these exceptions are:

- The *tick* signal of the SCL Generator counter, which has a constant assigned to it ('1') to always run when the SCL Generator is enabled. This results in an expression and a block in the counter not fully covered, as the logic does not encounter the tick signal being low.

- The peripheral clock request signal *ck_sys_bus_req* is set when any of the clock signals of the peripheral are enabled. The problem with this expression metric is that the resynchronizer clock signal is *always* enabled when the control block clock signal is requested, therefore the control block can not set the clock request signal *by itself*. While this makes the expression have a redundant check, it was left in place to improve readability and to ensure a bug would not be added if the resynchronizer clock enable signal had its logic changed.

- The In-Band Interrupt shift register is only used for reading the first address after a (non-repeated) START condition and does not have its serial data output connected. This results in expressions, blocks and statements not covered due to the shift register having its control signals tied to a constant value.

- The *next_state* statement when in the *others* state of an FSM when all of the states were explicitly listed.

Name	Overall Average Grade	Overall Covered
(no filter)	(no filter)	(no filter)
i_i3c_controller	99.53%	5402 / 5464 (98.87%)
i_rst_gen	100%	3 / 3 (100%)
i_ck_gen	100%	47 / 47 (100%)
i_ctrl	100%	1416 / 1416 (100%)
i_regmap_wrapper	98.56%	3759 / 3816 (98.51%)
scl_resync	100%	7 / 7 (100%)
sda_resync	100%	7 / 7 (100%)

Figure 5.9: Code coverage of the peripheral with applied waiver by its blocks.

Details | i_i3c_controller

Metrics Source Attributes

Ex UNR Name	Overall Average Grade	Overall Covered
Overall	99.53%	5402 / 5464 (98.87%)
Code	99.53%	5220 / 5282 (98.83%)
Block	99.98%	2584 / 2588 (99.85%)
Statement	99.97%	2331 / 2336 (99.79%)
Expression	99.62%	2550 / 2603 (97.96%)
Toggle	94.51%	86 / 91 (94.51%)
FSM	100%	182 / 182 (100%)

Figure 5.10: Total code coverage of the peripheral with applied waiver by metric type.

The resulting code coverage with the applied waivers is 99.53 % with most of the uncovered code being in the *regmap_wrapper* (Figure 5.9), specifically the generated AHB register map itself. Additionally, the toggle coverage of the *i3c_controller* interface also does not achieve 100 % coverage. Both of these are related to the AHB signaling, as the C code only accessed the peripheral registers in words (32 bits). To cover this part of the design, the register map and its C header file would have to be slightly modified to allow a half-word (16-bit) and byte (8-bit) access to the registers.

Chapter 6

Implementation

To validate whether the design will work in an FPGA, the peripheral was imported into Xilinx Vivado 2019.1 for synthesis and implementation. The top file of the peripheral needed to be wrapped so as to assign values to the generics listed in Chapter 4 *RTL Design*.

From a design perspective, the implementation on an FPGA and ASIC differ in:

1. limited number of clocking resources,
2. combinatorial logic implemented in Look Up Tables (LUT),
3. timing problems with implementation of latches,
4. finite number of pre-existing structures (RAM blocks, hardware multipliers, PLL, etc), and
5. limited asynchronous set/reset functionality.

These result in the FPGA design being slightly different from the ASIC design.

To ease the implementation on an FPGA, the I3C Controller peripheral uses parametrization to specify which design is to be elaborated and/or synthesized. The *G_TARGET_TECH* generic on top of relevant RTL modules selects if the design is for an FPGA or an ASIC. This results in a "single" change of the design – the clock gating, which was disabled. No further changes were necessary, as the design doesn't use clock multiplexing nor utilizes both asynchronous set and reset for the same registers.

6.1 Physical design parameters

The Xilinx Vivado 2019.1 tool was used for synthesis and implementation on a Xilinx Atrix-7 family model xc7a100tcsg324-2 FPGA.

The FPGA design was constrained to have its clock signals (*ck_sys_bus* and *ahb_hclk*) run at 100 MHz. A timing analysis was run after synthesis and implementation to check whether the designed peripheral was capable of running at those frequencies and whether the design met all timing requirements (Figure 6.1). In total, the implemented design utilized 1144 Look Up Tables (LUT) and 1355 Flip-Flops (FF) of the FPGA.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.462 ns	Worst Hold Slack (WHS):	0.056 ns	Worst Pulse Width Slack (WPWS):	4.600 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1970	Total Number of Endpoints:	1970	Total Number of Endpoints:	1355

Figure 6.1: Timing report of the generated FPGA netlist with clock frequency of 100 MHz.

Resource	Utilization	Available	Utilization %
LUT	1144	41000	2.79
FF	1355	82000	1.65

Figure 6.2: FPGA utilization by the I3C Controller.

6.2 Gate Level Simulation

For running gate level simulations in the Xcelium simulator, the netlist and SDF files needed to be exported from the Xilinx Vivado tool, which was used for synthesis and implementation.

The export of the SDF and verilog netlist from Vivado required the use of the tools TCL console, as the graphical interface lacked the option to export the post-implementation netlist. Additionally, the TCL command allows the designer to include the necessary simulation models directly in the netlist instead of linking to a Xilinx library.

The exported files needed to be slightly modified in order to be used in the Cadence Xcelium Simulator, as the simulator was not able to read a relative path to the annotated SDF file, and one of the entities defined in the netlist shared its name with an entity used elsewhere in the RISC-V system, which resulted in the simulation not launching due to missing connections unless one of them was renamed.

These files were then used to replace the written RTL code for a mixed RTL / gate level simulation (GLS) of the system in the Xcelium simulator, where the RISC-V system was simulated on a behavioral level with the exception of the I3C Controller peripheral, which used the post-implementation netlist with the accompanying delay file for a limited gate level simulation. This simulation validated the overall function of the peripheral and whether the peripheral is capable of meeting its timing requirements.

Whether the Xcelium simulator uses the generated SDF and netlist files correctly was checked by observing the inputs of an arbitrarily chosen LUT in the design, which showed glitching in one of its inputs after a clock edge and differing delays between changes on its inputs (Figure 6.3), which indicated the gate level simulation of the peripheral ran successfully.

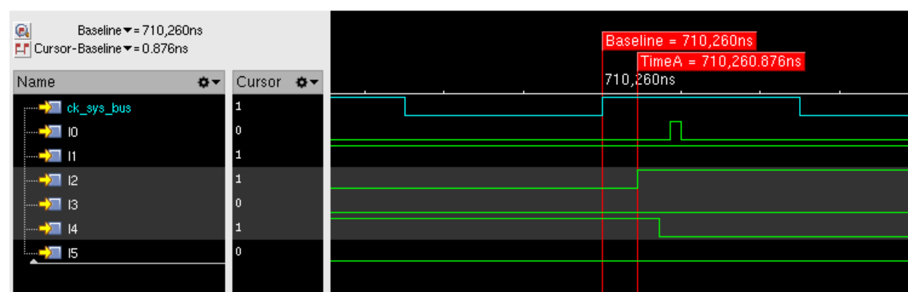

Figure 6.3: Inputs of a LUT in a gate level simulation, which show the Xcelium simulator properly using the generated SDF and netlist files.

Given that the rest of the system uses RTL simulation for the rest of the RISC-V platform, a small modification to the signals between the system and the peripheral needed to be done. The *ahb_hwdata[31:0]* signal, which carries data from the RISC-V system to be written into the peripheral register map needed to be delayed in order to fulfill the hold timing requirement after a clock edge.

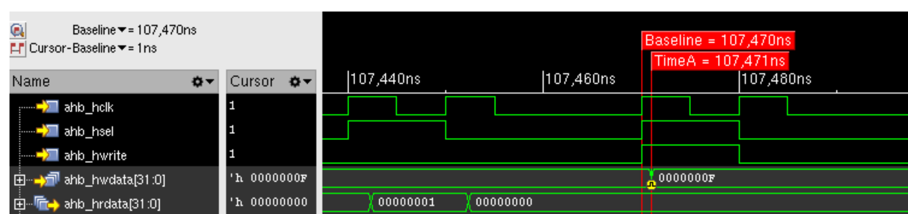

Figure 6.4: Delay inserted into the ahb_hwdata[31:0] signal for gate level simulations.

A 1 ns delay was added to the *ahb_hwdata[31:0]* signal when running gate level simulations of the peripheral (Figure 6.4), which resulted in all of the RTL tests passing. The tests were then repeated for over 40 runs distributed across all of the tests to ensure the generated netlist passed verification and all tests passed.

Chapter 7

Conclusions, Next Steps

The goal of this thesis was to design an I3C Controller peripheral for the RISC-V system platform. The I3C protocol specification was studied and a system level design was written as a general guide for its implementation.

The RTL design of the peripheral was then written in VHDL while testing its basic functionality. To test the Controller, an I3C Target Agent was created to verify that the Controller accesses the I3C bus correctly and is capable of writing and reading data to/from the bus.

The I3C Controller was designed to implement the vast majority of the features of the free version of the I3C protocol specification. The designed I3C Controller supports common usage scenarios, and by decomposing the functions between hardware and software, it can be easily customized according to the requirement of the future application/system in which it will be used. The expected requirement can be for example a modification of the CCC command, this can be easily done by changing the software.

The design was then imported into the Xilinx Vivado 2019.1 tool for synthesis and implementation, which generated a netlist annotated by a simulation delay file necessary for a gate-level simulation of the peripheral. These files were imported back into the RISC-V system and simulations were run to verify that the design behaves as expected after implementation. The goals of this thesis were fulfilled, as the created I3C Controller peripheral passed the listed test cases.

100% code coverage was not achieved during verification; there is room for improvement in the future. Code coverage analysis showed that 16b and 8b wide read/write register map accesses are missing in the verification. In the future, test cases should be extended to cover these use cases.

Bibliography

1. MIPI. *I3C Basic v1.1.1 - bus specification* [online]. 2022. [visited on 2024-05-21]. Available from: https://www.mipi.org/mipi-i3c-basic-download.
2. MIPI. *Achieving Power Efficiency in IoT Devices with MIPI I3C* [online]. 2022. [visited on 2024-05-21]. Available from: https://www.mipi.org/download-mipi-whitepaper-achieving-power-efficiency-in-iot-devices-with-mipi-i3c.
3. CUMMINGS, Clifford E. *Simulation and Synthesis Techniques for Asynchronous FIFO Design* [online]. 2002. [visited on 2024-05-21]. Available from: http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf.
4. ŠŤASTNÝ, Jakub. *FPGA Prakticky*. Ben, 2010.
5. XILINX. *Vivado Design Suite User Guide: Synthesis* [online]. 2020. [visited on 2024-05-21]. Available from: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf.

Used software:
Microsoft Office 365, Draw.io v24.1.0, MiKTeX v4.9, Xilinx Vivado 2019.1, Xcelium Logic Simulator 21.09.007, IMC 23.03-s004, DVT v23.1.18, Overleaf, DeepL.

RTL codes

The RTL codes of the design were uploaded as an attachment and are available in the public/online version of the thesis.