Novel Method of ASIC interface IP development using HLS

Anestis Athanasiadis
an7644at-s@student.lu.se
Chandranshu Mishra
ch3843mi-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Academic Supervisor:
Liang Liu

Supervisor: Mattias Sönnerup (Ericsson)

Examiner: Erik Larsson

August 23, 2023

© 2023
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

High-Level Synthesis(HLS) is a design methodology that enables designers to implement hardware from high-level coding languages, such as C, C++, or System C. It provides designers with the ability to convey their design at a higher level of abstraction, which allows more emphasis on an algorithm and functional aspects of design instead on low-level hardware details. As precise control of signal timing in HLS is not a straightforward task, for this reason, it has not a preferred method for control logic designs.
The objective of this master's thesis is to investigate the learning opportunities associated with making use of HLS for the development of an Application Specific Integrated Circuit(ASIC) interface module. To achieve this goal, an Improved Inter-Integrated Circuit(I3C) controller module was built by utilizing the Catapult HLS platform. After completing the design, the module was synthesized in a sub-10nm technology process, to allow a comparison with an Intellectual Property(IP) with the same functionality, developed in traditional Register Transfer Level(RTL).
Furthermore, any challenges that were presented during the implementation stage are identified and possible ways to overcome them are proposed. Consequently, the produced design was functional, but clock accuracy was limited due to increased latency. A 26% increase in the total area was noted, although this difference can be reduced with further optimizations.

Popular Science Summary

With each passing day, more advanced digital technologies are emerging, which bring about higher clock frequencies, improved energy efficiency, and an increased number of transistors over an area. With such advancements in design capabilities, there is a need for better designing techniques that can keep up with this advancement. There is a need for faster methods of design and verification of designs. There used to be a time when designers used to place each individual transistor manually, but that was replaced with hardware description languages such as VHDL and Verilog, which increased the frequency of designing hardware. The time to market for these products is small, so there is always a rush to finish and verify the design to meet market requirements. Also, the designs are becoming more and more complex day by day and all this culminates in a need for a quicker way to design and verification.
To solve this issue, new technology has been emerging for some time, but not very widely used. If this technology is found to be better than existing hardware description languages, it could mark a shift in the industry in how development is taking place. This new technology is called High-Level Synthesis. It has been existing since 1994, but it is gaining traction in recent times. Today, there are many vendors such as Siemens, Cadence, etc. that provide high-level synthesis on their platforms. In this thesis, we are going to expand into how HLS can be used to design ASIC and what challenges we face in doing so.

List Of Abbreviations

HLS	High-Level Synthesis
ASIC	Application Specific Integrated Circuit
I3C	Improved Inter-Integrated Circuit
IP	Intellectual Property
RTL	Register Transfer Level
VLSI	Very Large Scale Integration
FPGA	Field Programmable Gate Array
MIPI	Mobile Industry Processor Interface
IoT	Internet of Things
I2C	Inter-Integrated Circuit
SPI	System Peripheral Interface
FSM	Finite State Machine
HDR	High Data Rate
SCL	Serial Clock Line
SDA	Serial Data Line
CCC	Common Command Code
IBI	In-Band Interrupt
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
CPU	Central Processing Unit
RAM	Random Access Memory
GCC	GNU Compiler Collection
GDB	GNU project Debugger
FIFO	First In First Out
DRAM	Dynamic Random Access Memory
SRAM	Static Random Access Memory

Table of Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

For a long time, RTL has been the predominant approach for describing Very Large Scale Integration(VLSI) systems and their IPs. The advancements in tools describing RTL designs have experienced growth, yet the complexity of VLSI designs has concurrently escalated. This has resulted in a bottleneck in the design process[1]. HLS is a design methodology which can be explored to tackle this challenge, as it enables designers to implement designs at a more abstract level. This makes the development process simpler [2][3]. HLS is predominantly used in designs focusing on data flow, not so much in clock-accurate control and control-oriented designs. This project aims to explore how HLS can be used to design a clock-accurate design[4]. An I3C controller module is designed in untimed C++ language having the same specifications as a pre-existing design in Verilog. Both designs are synthesized using similar technology library files and compared on the basis of area and power.
There were many challenges encountered throughout course of the project. Untimed C++ does not have concept of time during the execution time and clock and enable is automatically generated by the HLS tool[5]. The inability to probe the clock and reset pin while designing a clock-accurate design is a major constraint. HLS has been explored to a lesser extend for clock-accurate designs so there is a limited number of related publications. Thus, many challenges which can occur in implementation are still undocumented.
Firstly, the HLS platform allows for numerous optimizations that can be applied to different parts of the design, and for each case, the most optimal configuration must be selected. Therefore, an extensive understanding of the tool's available parameters must be acquired, before proceeding to implementation. Subsequently, due to the nature of the comparison, it is difficult to acquire an accurate area and power difference, as both designs need to have 100% similar functionality. Also, a specific coding style needs to be followed and the hardware design approach still needs to be adhered to.
In this project, it was discovered that HLS reduces the time to design but also there are many factors which effect the quality of it. The coding approach differs from traditional software development practices, and a hardware-oriented mindset should be maintained. Many issues surfaced during the implementation process and some solutions are proposed. The produced design has an increased area by 26%, but this margin can be decreased with further optimizations.

Chapter 2

Background

2.1 Catapult High Level Synthesis

In this section we are explaining more about the tool used for developing our HLS design and some of its features.

2.1.1 Brief description

Catapult HLS, developed by Siemens EDA, is an advanced HLS toolkit that enables the transformation of high-level language description into hardware designs. High-Level Languages such as C, C++ and SystemC are directly translated into RTL and are subsequently synthesized for use in Field Programmable Gate Arrays (FPGA) or ASIC. Additionally, the platform provides an HLS Verification Flow that can be imitated before the design's completion, and allows for higher simulation speeds [6].

2.1.2 Untimed C++

Untimed C++ or Algorithm C refers to a subset of object-oriented programming in C in which the concept of time is not included in the execution model. It prioritizes algorithmic and functional aspects over consideration of timing or synchronization of operations.
Subsequently, the language diverges from timed or real-timed programming, which tightly associates code execution with specific time intervals or synchronization requirements, by enabling a higher level of abstraction for representing algorithms and system behavior. It makes the design process simpler by removing the need to define timing constraints at the time of designing [5].

2.1.3 Clock accurate designs with HLS

Due to the automatic scheduling of untimed C++ processes, no clock signal is present on a code level. Designers cannot access the system's clock status, but have to rely on a set of directives and timing configurations implemented via the HLS tool. Subsequently, control and reset enable signals cannot be probed or controlled manually [5]. The automatic generation and configuration of these signals by the software, limit design flexibility, by creating a constant dependence on the project's configuration parameters. Therefore, this increases the difficulty of creating applications where clock accuracy and high responsiveness are important, such as communication protocols [4].

2.2 The I3C Protocol

I3C is an abbreviation of Improved Inter-Integrated Circuit protocol which has been developed by Mobile Industry Processor Interface(MIPI) Alliance. In the realm of mobile devices and Internet of Things(IoT) applications, I3C offers a streamlined and efficient approach for facilitating communication between sensors and peripherals. It presets an optimized solution that enhances performance, extends capabilities, and incorporates advanced features tailored to meet specific communication requirements within these domains. Because of these features, I3C provides seamless integration and enables reliable data transfer between sensors and controller.
By utilizing the existing Inter-Integrated Circuit(I2C) and System Peripheral Interface(SPI) communication protocols as a starting point, I3C expands upon their features and capabilities to create a comprehensive and adaptable communication solutions. It has back-compatibility with I2C, which lets legacy devices to operate over I3C bus [7].
The key features of I3C are:

- increased data transfer rates
- support for multiple controllers
- hot join and dynamic address assignment
- utilizes low power

2.3 The I3C Controller

2.3.1 Functionality

In this section, the I3C module designed for the project is clarified. An I3C module can act either as a "target" or a "controller" [7], the module described here provides the latter functionality. To obtain an improved understanding of the applicability of HLS in complex clock-accurate systems, we based our module on an already existing commercial design, thus increasing the code complexity. By doing this, it was possible to locate a broader range of issues that would not be visible in a design of lesser scale. Figure 2.1 shows the top Finite State Machine(FSM) of the module, although some functions have not been implemented as they were not present in the design made in traditional RTL. The excluded features are High Data Rate(HDR) mode support, hot-join capability and controller role request. Consequently, a brief explanation for the key features of the design is given below.

- Private read/write

- Dynamic address assignment
- Ability to broadcast "Common Command Code" commands.
- In-band interrupt support
- Adjustable Serial Clock Line(SCL) and Serial Data Line(SDA) timing frequency and offset delays.
- Backward compatibility with legacy I2C target devices.

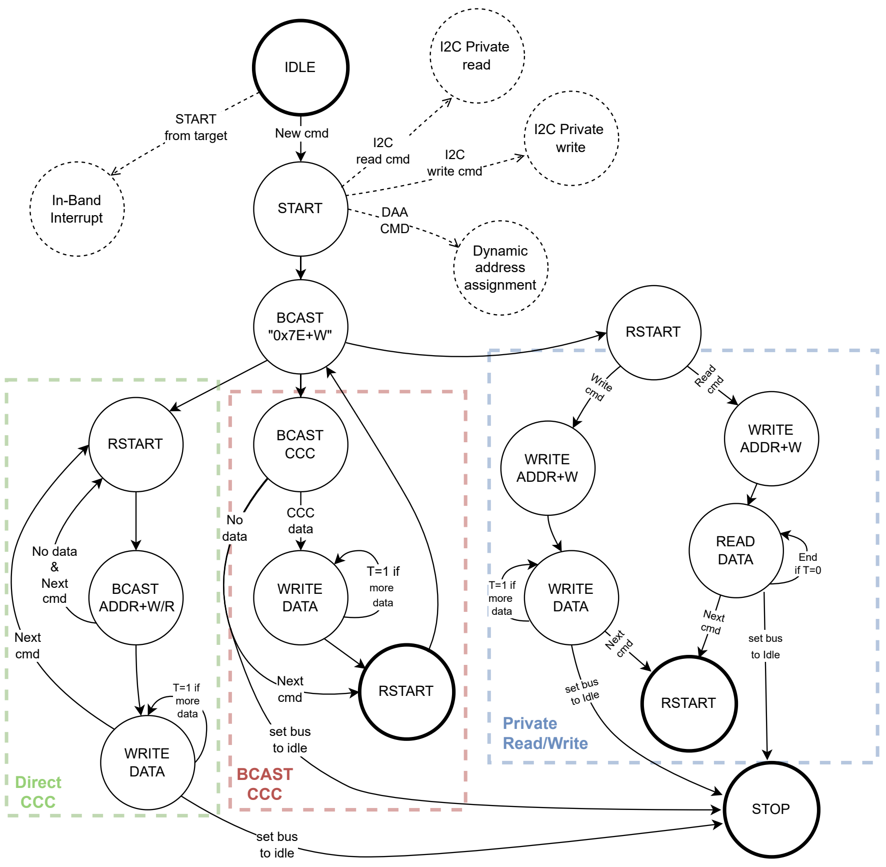


Figure 2.1: Simplified I3C Primary Controller FSM

Private read/Private write

Private read refers to the process where the I3C controller can read from one of the target devices. Private write refers to the process where the controller can transmit data on one of the target devices through SDA line. This can be done by selecting the respective target by their static or dynamic address and read or write data [7].

Dynamic address assignment

Dynamic address assignment is the process where the controller discovers and assigns addresses to the target devices present on I3C bus. When a target device is connected on I3C bus, the controller assigns an address to the target from a predetermined address table. The controller has a predefined set of addresses which are assigned to targets when needed. This allows a better management of the addressing memory because the dynamic address is only assigned to the targets whenever it is required [7]. This can be seen in Figure 2.2.

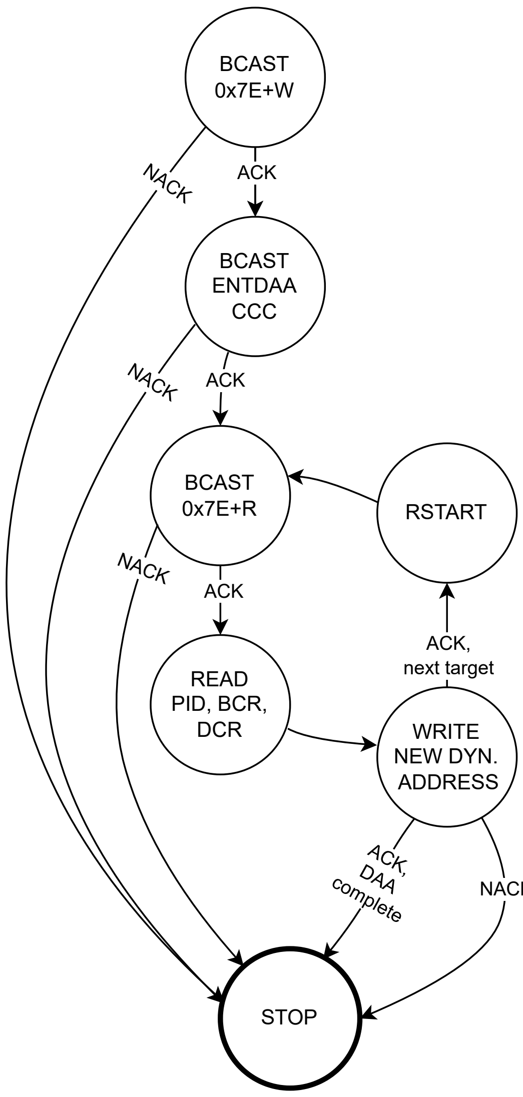


Figure 2.2: Simplified Dynamic Address Assignment FSM

Broadcast Common Command Codes

Common Command Codes(CCC) is one of the features in I3C protocol which are used to control and configure the target devices present on the line. There are a number of functions for which CCCs are used like requesting configuration information from the target and controlling which target gets bus access in order to maintain reliable configuration [7].

In-band interrupt support

This feature allows the target device to initiate a transfer without being prompted by the controller. The target device can issue a "start" command along with its address when the bus is in an idle state. Subsequently, the controller verifies if the transmitting device is registered in its device address table and that it is allowed to generate an interrupt request. If all requirements are fulfilled, then the controller acknowledges the request and reads one byte of data from the target. Otherwise, the request is denied, and the controller then broadcasts a dedicated command that disables the In-band interrupt(IBI) request capability for that specific device or all of them, depending on the controller's configuration [7]. These actions are also illustrated in Figure 2.3.

Adjustable SCL and SDA timing frequency and offset delays

Due to its commercial nature, the module provides some flexibility in terms of the bus timing characteristics. The user is able to define the bus frequency by adjusting the high and low time of the SCL waveform. Furthermore, the switching SDA line can be manually offset by a set amount of clock cycles, as seen in Figure 2.4. These settings can be configured via the APB bus.

Backward compatibility with legacy I2C target devices

The MIPI I3C specification dictates that I2C target devices can be interfaced in the I3C bus. To achieve this, the FSM presented in Figure 2.5 was implemented in the design. As I3C speeds can be much higher that the ones used in the i2c protocol, the devices are required to have a 50ns notch filter, so all I3C transactions can be ignored [7]. Additionally, the controller must set the bus frequency to values specified in the I2C standard so communication can be possible.

2.3.2 Block description

The block diagram of the design can be seen in Figure 2.6. Communication between the module and the associated CPU application is achieved through the utilization of the Advanced Microcontroller Bus Architecture(AMBA) Advanced Peripheral Bus(APB) protocol [8], the related signals are placed on the left side of the module, with the addition of the interrupt signal that is used to indicate the occurrence of specified events on the module.

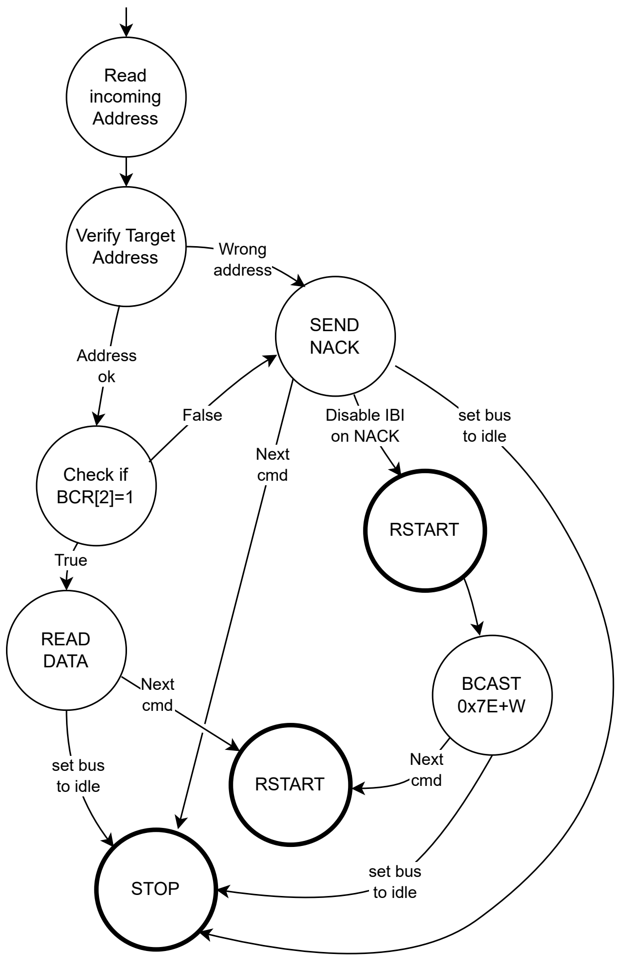


Figure 2.3: Simplified In Band Interrupt FSM

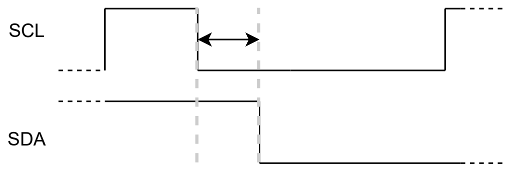


Figure 2.4: SDA offset compared to SCL

The I/O signals that are on the right side are related to the I3C bus. To control and probe each line three signals are needed, "tribuf enable", "state" and input. The first two are connected to a tri-state buffer circuit as seen in Figure 2.7, and the latter is set directly on the bus. Lastly, two configurable pull-up resistors must be added on the bus, to be able to switch between push-pull and open-drain

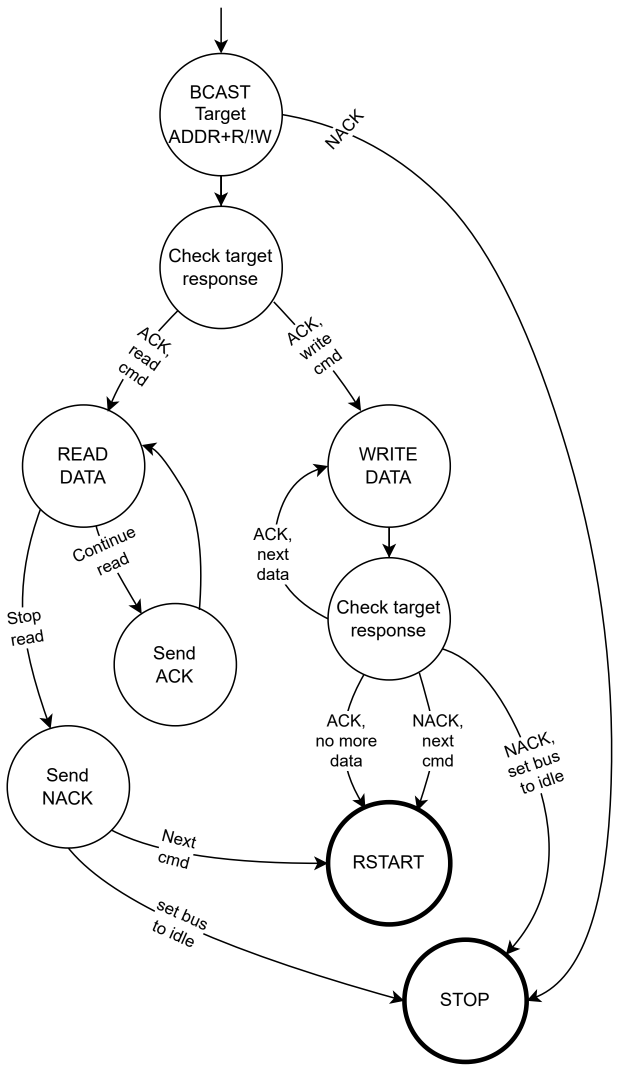


Figure 2.5: I2C communications FSM

modes.

To provide more detail in the designs operation, it's sub-modules are described below.

Bus driver block

The main function of this block is to directly regulate the bus. It manages the switching of both SCL and SDA lines, with the latter changing its value when instructed by the higher sub-module. To control each line, a separate inline block was created, as seen in Figure 2.8. Additionally, it is able to switch from open-drain

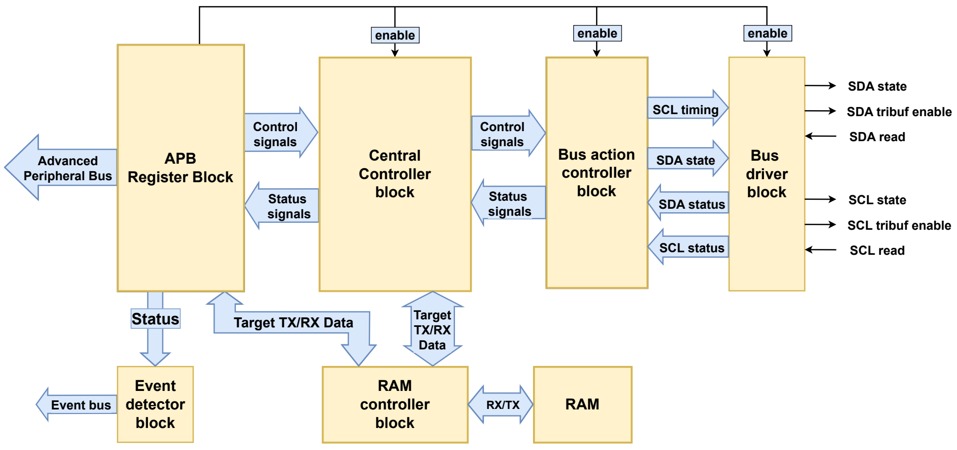


Figure 2.6: I3C controller top view

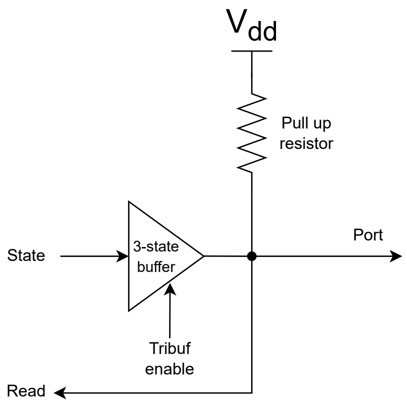


Figure 2.7: SDA/SCL port control

to push-pull mode and vice-versa on demand.

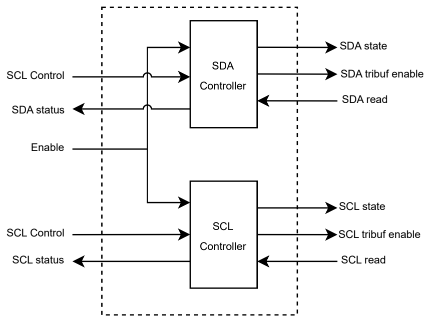


Figure 2.8: Bus driver block diagram

Bus action controller block

By analyzing the I3C and I2C protocols it is noted that the majority of the operations are repetitive. To reduce area, these were broken down into different functions such as "Transmit Start", "Write Byte + T bit", "Read byte", etc. The logic behind this, is that any process can be chosen by the master controller block when required. When the selected action is complete, a report signal is activated so the system can move to the next one. In Figure 2.9 the bus action controller is illustrated. To simplify the diagram some of the designed actions are phased out.

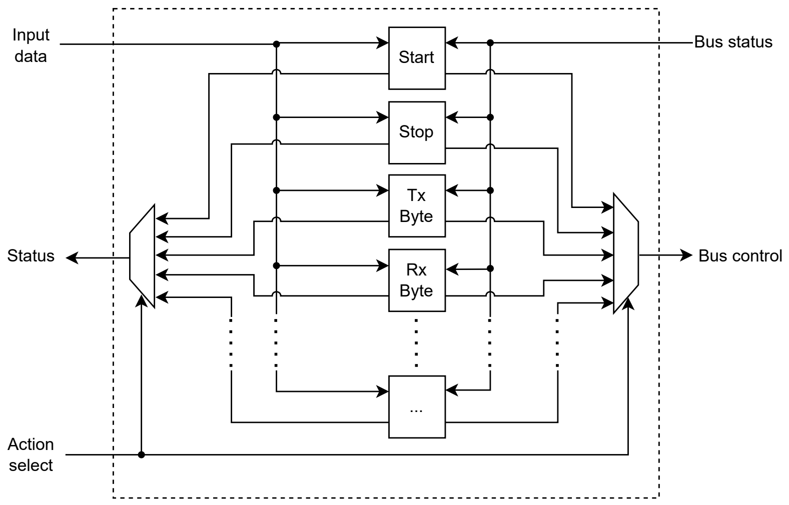


Figure 2.9: Bus action controller block diagram

Central controller block

This is the larger sub-module in terms of complexity and area. Its purpose is to probe the setup registers and ram for new commands, and execute them. It is comprised of a number of FSMs that are used to perform different operations, such as private read/write, dynamic address assignment, IBI handling, etc. A top FSM is used for deciding which process must be selected according to the input from the APB register block and RAM, this is seen in Figure 2.10

Ram controller block

To achieve higher design flexibility, a Random Access Memory(RAM) was added to the system, it is used to buffer communication data before these are transmitted to a device or read from the Central Processing Unit(CPU). The memory can be accessed by both the Master controller and APB register block, but it will always give priority to the latter as it is more critical to the system's operation.

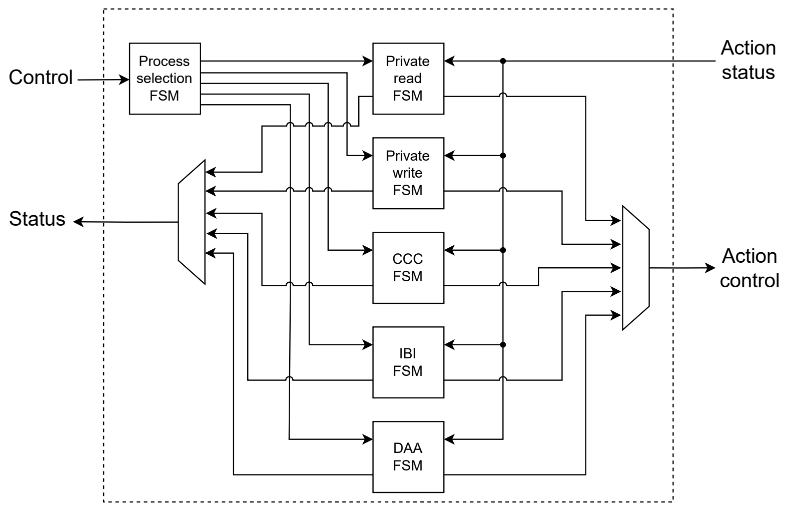


Figure 2.10: Central controller block diagram

APB register block

The main role of this sub-module is to handle read and write accesses to the register array from the APB bus, Central controller, and Interrupt block.

Event detection block

This module's purpose is to produce an interrupt signal whenever certain system conditions are met. These could be an incoming IBI request, receive/transmit buffer overflow, communication errors, etc.

Chapter 3

Catapult HLS flow

3.1 Workflow

A typical design flow would start by creating and verifying a model of the specified system in Matlab or other high-level languages, as seen in Figure 3.1. Subsequently, the system is re-implemented with a hardware description language such as VHDL or Verilog [9].
After the previous steps have been completed, the design undergoes synthesis using the specified technology library. The synthesized model is then reviewed for any timing violations. If such violations are found, the RTL coding phase is reassessed with the aim of either reducing the complexity of the design or introducing additional pipeline stages. Thus, a loop between synthesis and RTL coding is created. When the design is clear of errors, it can proceed to layout or additional testing [9].
With the use of the HLS platform, the specified design is implemented exclusively using C++, although certain elements of the language, such as unions or linked lists, are not supported [5]. The resulting code can be compiled using any appropriate tool depending on the operating system in use. In this project, the GNU Compiler Collection(GCC) tool was utilized for compilation. The verification and fault-finding process was achieved with the use of the GNU project Debugger(GDB), along with assertions and print debugging techniques.
In the subsequent stage, the equivalent RTL version of the code is automatically generated from Catapult HLS, followed by synthesis. In contrast to the aforementioned flow, timing violations are handled by the HLS tool [2]. Consequently, the generated design may have increased latency or pipeline stages to meet the timing constraints.
Afterward, the synthesized model is examined for differences in output response, as well as potential pipeline stalls and deadlocks. This process involves utilizing QuestaSim and SCVerify, with the latter enabling the design's simulation with the C++ testbench[10]. Any output mismatches are identified in the resulting waveform, as depicted in Figure 3.2. If such an error occurs, the C++ code needs to be modified, and the process is repeated.
For pipelining, the designer is able to instruct Catapult HLS to pipeline specific parts of the design while defining the exact iteration interval and throughput. Furthermore, any created loops can be unrolled as needed[5]. In our project, we opted

for an iteration interval of 1 for all blocks. Lastly, a sampling-based programming style was adopted to minimize the design latency without requiring loop unrolling.

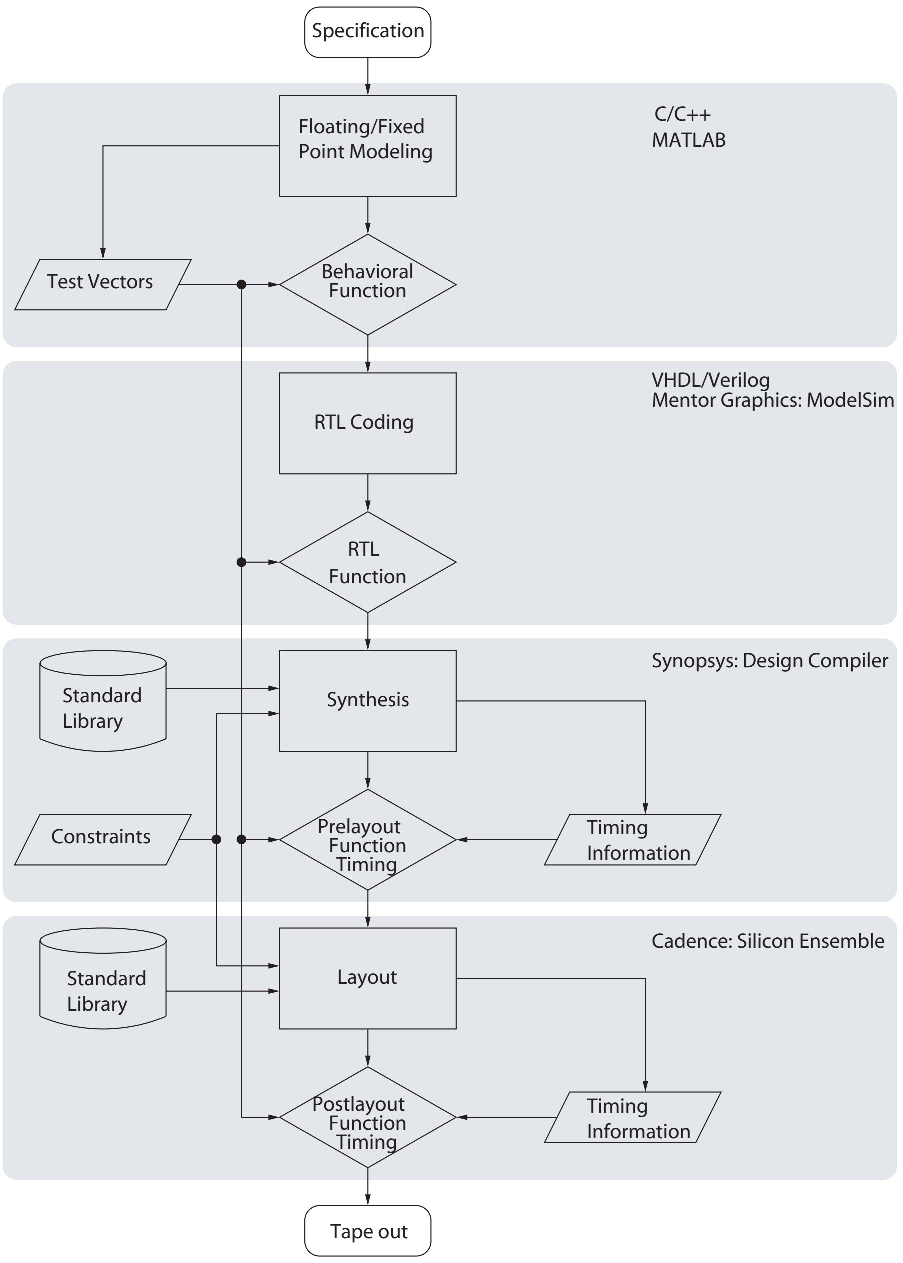


Figure 3.1: .
Design flow steps, taken from [9].

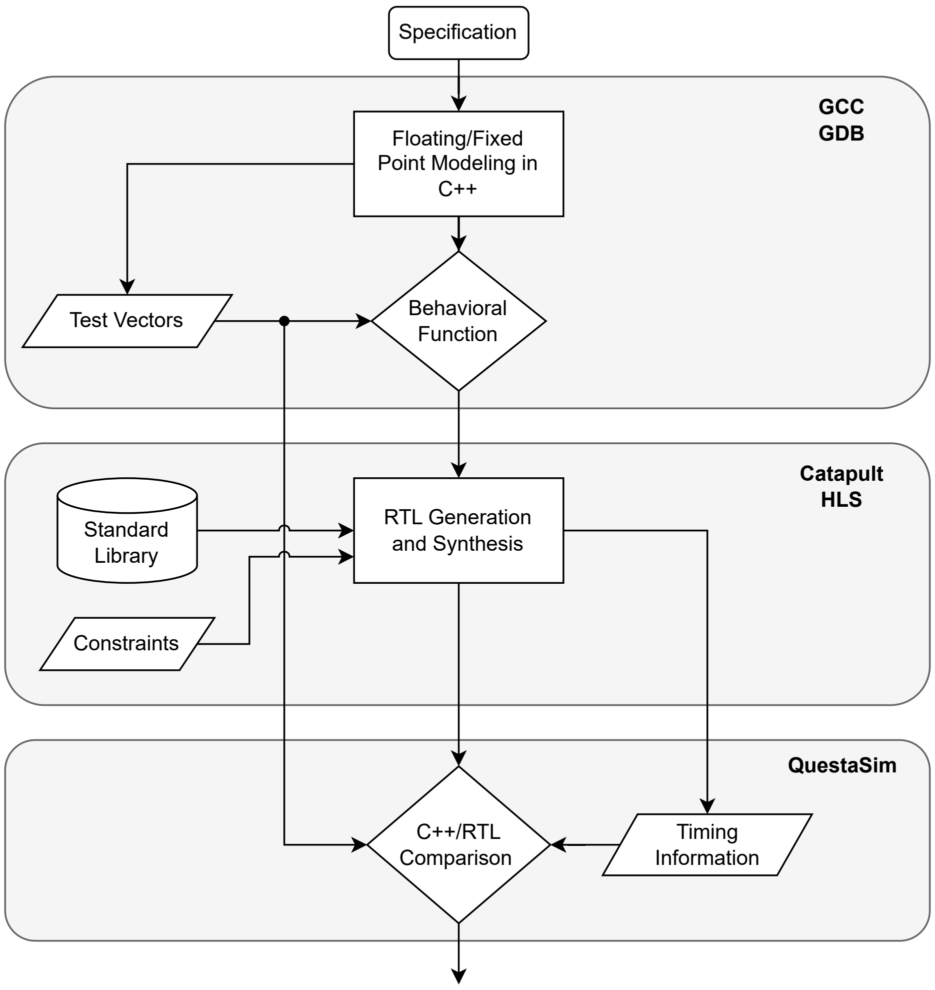


Figure 3.2: Netlist design flow using HLS

3.2 Design partitioning

As with any typical RTL design planning, large projects need to be partitioned into distinct components. These components can consist of classes encompassing multiple functions, with a top function encapsulating all others[11]. This top function is invoked from a higher level where the class is utilized, which can be either the testbench or some other top wrapper. A simple example of a top class hierarchy is shown in Listing 3.1.
In this example, the function "run" acts as the top wrapper, and functions `"write_1"` and `"write_2"` are considered "Inline". An inline function is not considered a separate entity but a part of the block from which it is called.
A project can consist of multiple classes under a top class that acts as a wrapper.

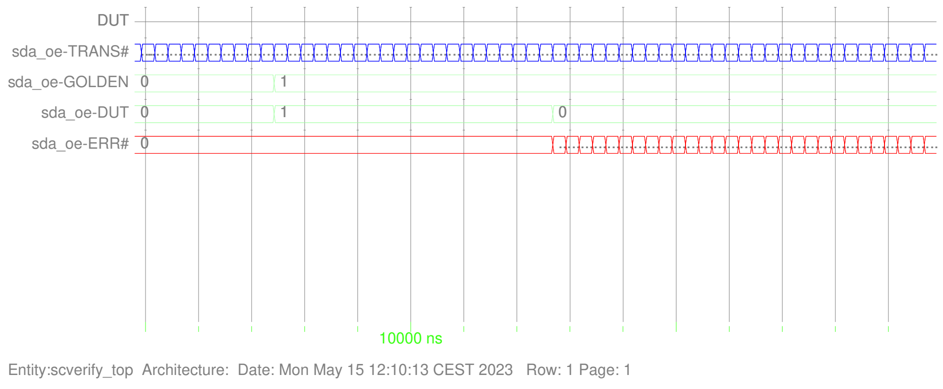
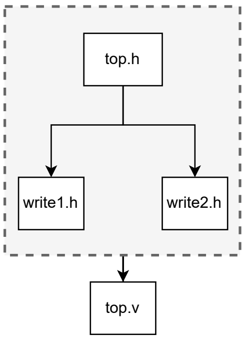


Figure 3.3: Mismatch error on QuestaSim

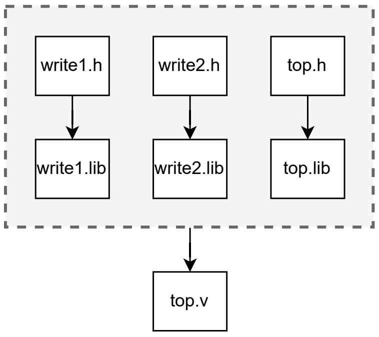
In that case, each class's main function is characterized as a block, and all blocks are assembled in the top class. By following this partitioning strategy, each block can have different attributes, such as pipeline stages, loop unrolling, and clock frequency.
To facilitate effective handshaking with other blocks and prevent the loss of information in case of conditional reads, all interconnected signals are interfaced to First In First Out(FIFO) registers acting as buffers between modules, with the exception of the wires connected to the top-level interface[12].
The registers provide non-blocking write capability, although reads are blocking. The library provides a solution for the latter by probing the register for existing data before reading [5]. By following this configuration, the system becomes comparable to a Kahn process network [13]. Catapult HLS provides the "`ac_channel`" library to be used for this purpose. Listing 3.2 provides an example of block interconnection and hierarchy.
Synthesis of a hierarchical design can be accomplished in two different ways: "top-down" and "bottom-up" hierarchy. With the former approach, one block is set as the top file, and the rest of the blocks reside under it. The drawback of this approach is that during each synthesis iteration, the entire design needs to be re-synthesized.
A "bottom-up" hierarchy implies that each block is synthesized separately and converted into a technology library file. Subsequently, these files are used in the top wrapper block. This provides the advantage of not having to synthesize the complete design when a sub-block is changed but only the blocks that were altered. Therefore, the development process becomes more efficient [14].

```
class write_example{
  public:
  write_example(){}

  #pragma hls_design top
  CCS_BLOCK(run)(int &data_1, int &data_2, int &out_1,int &
    out_2){
```



(a) Top down



(b) Bottom up

Figure 3.4: Hierarchy types

```
    /*write 1 and 2 are inline functions*/
    write_1(data_1,out_1);
    write_2(data_2,out_2);
  }
  private:
  void write_1(int &write_data, int &output){
    output=write_data+1;
  }
  void write_2(int &write_data, int &output){
    output=write_data+2;
  }
};
```

Listing 3.1: Class hierarchy example

```
class write_example{
  public:
  write_texample(){}

  #pragma hls_design top
  CCS_BLOCK(run)(ac_channel<int> &data_1, ac_channel<int> &
    data_2, ac_channel<int> &out_1, ac_channel<int> &out_2){
    /*write 1 and 2 are seperate blocks*/
    write_1.run(data_1,out_1);
    write_2.run(data_2,out_2);
  }
};

class write{
  public:
  write(){}

  #pragma hls_design interface
  CCS_BLOCK(run)(ac_channel<int> &data, ac_channel<int> &out){
    write_inline(data, out);
  }
}
  private:
  void write_inline(int &write_data, int &output){
    output=write_data+1;
  }
};
```

Listing 3.2: Multiple class hierarchy example

3.3 Task division and project development

To increase efficiency in the design's implementation time, a concurrent development approach was followed [15]. Initially, a skeleton project was created and pushed into a Git repository. The repository was split into two different branches, one for each person. Both designers were assigned different blocks to work on as seen in Figure 3.5. Subsequently, one person acted as a forerunner and focused on working with blocks higher up in the design hierarchy, while the other designer worked on the block adjacent to the two previously completed blocks. These steps were repeated until the completion of the design.
In order to simplify fault detection and troubleshooting, a dedicated testbench was created for each individual block. When two blocks are merged, the testbench of the higher-level block is modified to accommodate the integrated system.
An issue that did occur at certain points in this process, was that the merging procedure took longer time due to unforeseen problems, or limitations that were not anticipated during the specification stage. In that case, only one person moved on to the next block while the other focused to complete the merge. Lastly, at certain points in the implementation process, it was deemed necessary to make modifications to the design specification. Consequently, continuous communication was essential throughout the entire process.

3.4 Scheduling

This part of the development process is where the Catapult platform automatically sets the design's latency and throughput according to the set timing constraints and the user's directives [6]. In HLS terminology, throughput denotes the frequency at which a process can be executed and completed within a specific number of clock cycles [5]. Subsequently, latency refers to the time between the first input and the first output.
Since the design in this project is mainly comprised of control logic, the latency is kept as low as possible since it allows for higher clock accuracy.
Initially, the design had a throughput value of 3, which was high for its field of application. During the scheduling stage, it was possible to pipeline the design, by configuring its iteration interval of the system. To elaborate, the iteration interval is the time required for a new iteration of the design to start. Thus, by setting

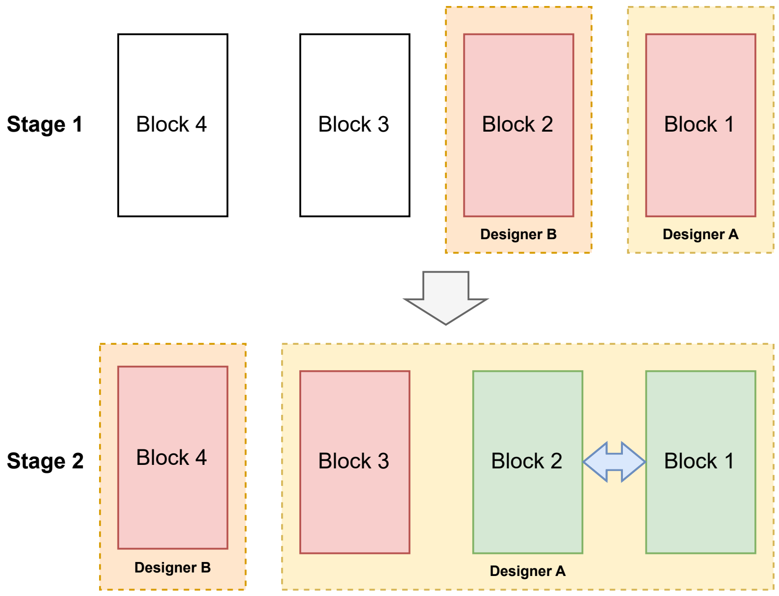


Figure 3.5: Parallel development of a modular design

this value to 1 one clock cycle, a throughput of 1 was achieved. The scheduling of the design is shown in Figure 3.6.

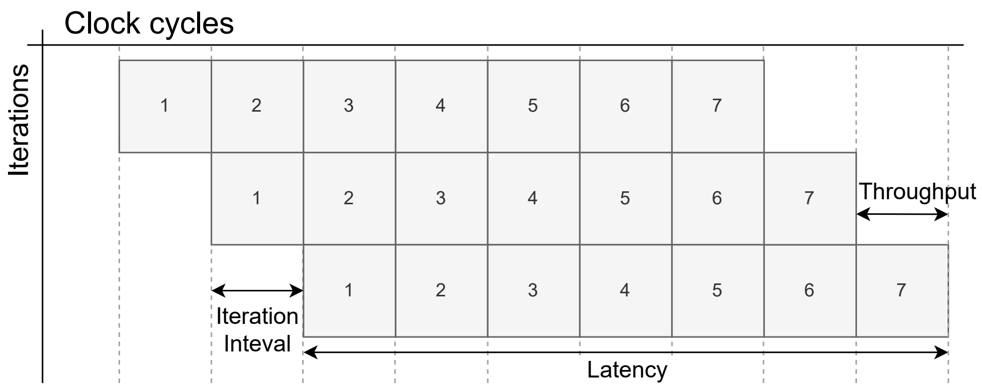


Figure 3.6: Resulting timing of the HLS design

3.5 Synthesis

To achieve a more precise comparison, the synthesis had to be implemented with the same process node as the pre-existing RTL design. For the implementation of

this project, a sub-10nm technology library was utilized. However, due to technical difficulties, the HLS design had to be synthesized outside the platform. This was made possible because Catapult extracts the finalized design in a single Verilog or VHDL file when synthesis is complete. Initially, the design was synthesized using the default 45nm library provided by Catapult. A low frequency of 10 MHz was selected, to avoid increased latency due to timing constraints. Thereafter, the RTL file was synthesized with the same medium as the RTL design.
One side-effect of following this approach, was that the inclusion of the RAM module was no longer possible. To to achieve more clarity in the results, two versions of the design were synthesized. One with the RAM converted to a register-based memory and another with an externalised memory. However, the latter version did not include a RAM module interface during the synthesis process. Lastly, the resulting area, static, and dynamic power results were used to achieve the comparison with the traditional RTL design.

3.6 Verification

To ensure that the device is behaving as expected, a functional verification environment was created in C++. As seen in Figure 3.7(a), the testbench consisted of a continuous loop containing the device under test and other elements. These are explained below.

Target Devices

To provide automated feedback to the controller, two classes were created, one representing an I2C device and the other an I3C device. Although, they have limited functionalities and fixed configurations. Specifically, the register bank block was not implemented in them, as it was deemed unnecessary. When a written request is received, the devices always accept, read, and store the data. Subsequently, when a read request occurs, the stored data are broadcast back on the bus. Additionally, the I3C device can accept new dynamic addresses and initiate IBI requests automatically after a specific series of events.

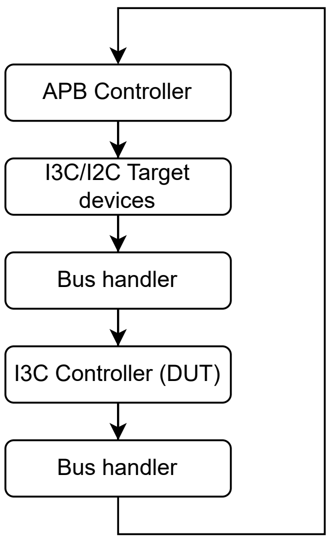
Bus handler

The bus handler function is a process designed with the goal of determining the SDA and SCL bus states, following any changes in the outputs of all I3C and I2C devices.

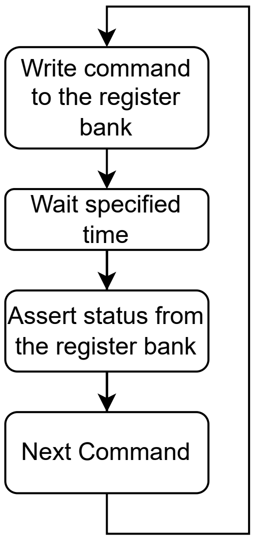
APB Controller

The APB Controller function is responsible for reading data, issuing new commands, and configuring the I3C controller. As mentioned in the previous chapter, the AMBA APB protocol is used for communication between the controller and other IPs in the system.
Figure 3.7(b) illustrates the implemented logic for verifying different aspects of the

device. The input stream consisted primarily of two distinct components: input and assertion data. The former contained a collection of configuration data, commands, and register bank addresses that these where to be written to. The latter was comprised of numerical values of registers and their equivalent addresses that were to be asserted.
For each test, the required data would be loaded into the register bank via the APB protocol and after some fixed time, the designated registers would be probed to verify if the value contained in them was correct. This delay was necessary because the register bank cannot be simultaneously read from and written to. To elaborate, by consistently probing one register, other modules had to wait in a queue to perform their write operations.



(a) The main loop used in the testbench


Write command to the register bank
Wait specified time
Assert status from the register bank
Next Command

(b) The assertion loop

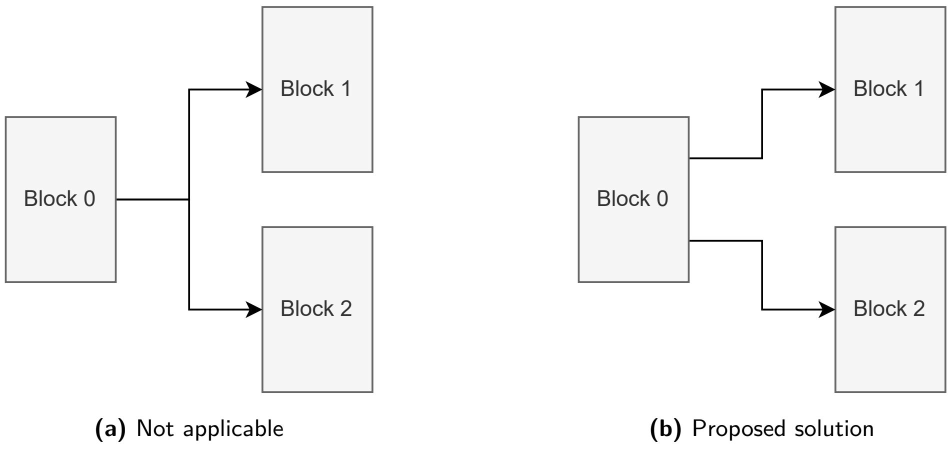
Figure 3.7: Verification process

Chapter 4

Issues and resolutions

4.1 Block design

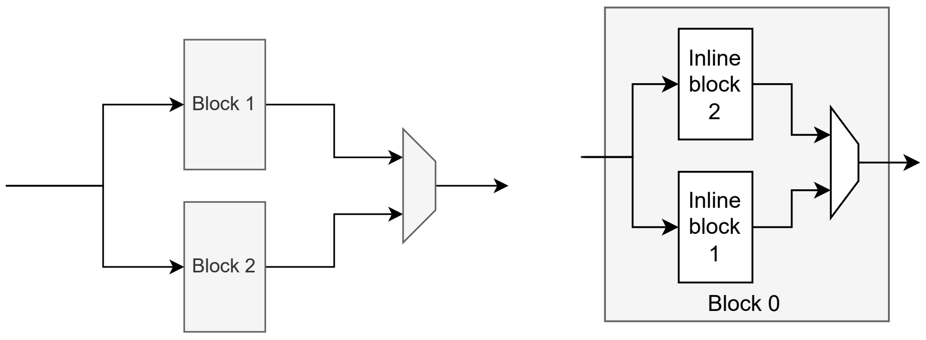
The way block interconnection and signal routing is implemented is different from traditional RTL. One of the initial steps of the project's implementation was overcoming this challenge. As mentioned in section 3.2, all the blocks in a design are interconnected with the use of FIFO registers named `"ac_channels"`. The main limitation of `"ac_channels"` is that a single channel can only be written in one block and read by another. Thus, a connection like the one in Figure (a) is not possible. A solution to this issue is creating two different channels for the same data, as seen in Figure (b).



(a) Not applicable **(b)** Proposed solution

Figure 4.1: Connecting an output signal to two different blocks

Furthermore, an additional problem that emerged was the incapability to multiplex the outputs of distinct blocks, due to the fact that top modules can only contain interconnect elements. A solution for this problem is presented in Figure 4.2 these modules can be set to inline elements of a single block. Nevertheless, this can lead to excessively large block sizes, although it won't pose a problem unless a different pipelining or running frequency is required for each block.



(a) Not applicable **(b)** Proposed solution

Figure 4.2: Connecting an output signal to two different blocks

4.2 Latency

A key contributing factor to increased latency is the timing constraints set by the clock frequency. To avoid any timing violations, Catapult breaks down the design into separate stages. Each stage takes one clock cycle to complete its process, therefore the design's latency is increased. An additional contributor, is dependencies between different blocks. Due to the blocking I/O provided by the `"ac_channel"` library, one block's process must be completed before another can commence, resulting in additional latency. Although this doesn't pose an issue in feed-forward architectures, the functionality of systems that include feedback channels is affected. Such a case is illustrated in Figure 4.3, in this example the design has a latency of 3. Since the feedback data is generated in the last block, it will require 3 clock cycles to reach block 0.
To mitigate this issue, a few solutions are proposed. In the case of latency due to dependencies, the blocking reads can be replaced by a non-blocking type, seen in listing 4.1 . The `"nb_read"` will return a Boolean value indicating that data were read. If no data are in the channel, then the read is skipped. The drawback of this approach is that some mismatches might be present in the simulation, as SCverify might not be able to simulate the non-blocking reads in both software and hardware.
Another solution is to convert all blocks into `"inline"` and merged a single module. Therefore, the need for the `"ac_channel"` library is alleviated. However, this module will be less flexible to optimizations due to it's size. Additionally, it is not possible to apply different scheduling and timing parameters for each inline block.

Finally, if the high latency is attributed to the HLS tool's efforts to meet the timing constraints, one possible approach to reduce it is to introduce pipelining into the design.

```
void Block(ac_channel<int> &data1, ac_channel<int> &data2,
    ac_chanel<int> &out){
     int data1_tmp, data2_tmp;
     if (data1.nb_read(data1_tmp))
         out.write(data1_tmp);
     else if (data2.nb_read(data2_tmp))
         out.write(data2_tmp);
}
```

Listing 4.1: Use of non-blocking reads

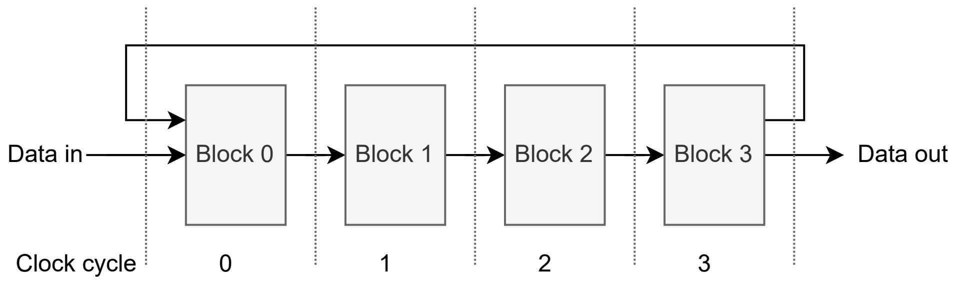


Figure 4.3: A design with feedback channel and a latency of 3 clock cycles

4.3 Counters/Timers

One of the most important limitations in this project was counter latency. As it was mentioned in section 3.4, the iteration time of a design can be increased due to dependencies between blocks. In Figure 4.6, the iteration interval of the HLS design's blocks can bee seen. These waveforms show that the majority of the block's have some dependency, and will be active every 5 clock cycles. Additionally, the `"cpp_testbench_active"` signal indicates the start of a new iteration.
To demonstrate the issue, an counter is also presented in the Figure. This signal exists inside the "Bus driver block", it can be noted that its value will increase every time the block is active. Therefore, the counter's minimum and maximum values are limited to multiples of the block's iteration time. For example, if the top value is set to 2, the counter will reset after 10 clock cycles. If a latency of 1 cannot be achieved, then the additional delay must be taken into consideration when creating systems that measure time.
To reduce this delay, two different approaches are proposed. The first solution is to merge all the system's components into a single block, as this will remove all the dependencies caused from blocking I/O and reduce the iteration time of the module. The drawback of this approach is that optimisation of the design becomes more difficult. The latter solution is the use of non-blocking reads from the `"ac_channels"` library. This will remove any latency due to dependencies between blocks, but might cause problems with the simulation of the design [12].

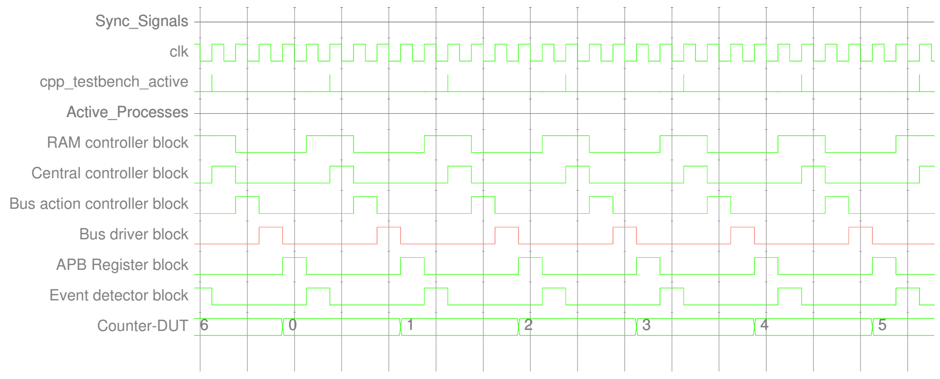


Figure 4.4: Incremental counter with a latency of 5

4.4 Hardware - Software mismatch

The occurrence of signal discrepancies between hardware and software simulations was initially discussed in section 3.1. The most likely reason for such discrepancies is the presence of feedback FIFO registers that have not been pre-filled. This problem is shown in listing 4.2, where the design of Figure 3.3 is described. In this example, the function of `"Block_0"` is called before `"Block_1"`, therefore, the channel `"data_out_1"` is empty on the first iteration of `"Block_0"`. To avoid any segmentation faults in the software simulation, the channel read command is skipped if data are not available. This causes a delay on the software side, as one or more loop iterations are required for the register to receive data. Consequently, the signal offset affects other elements that depend on it, leading to further delays. As a result, tracing the source of the fault can be time-consuming. A practical solution to investigate the origin of such errors is to replace the read of input channel signals with fixed value assignment in the affected block. For the example of Figure 4.5, a proposed solution is presented in listing 4.3, where the channel is pre-filed with one data token that has a value of 0.

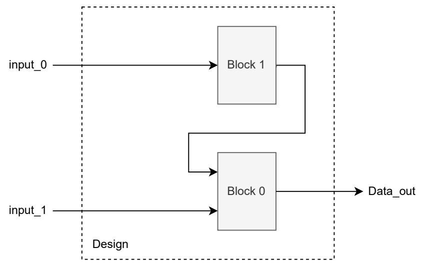


Figure 4.5: Design prone to signal mismatches

```
class mismatch_ex {
    ac_channel<int> data_out_1;
public:
    mismatch_ex() {}
    #pragma hls_design interface top
    void CCS_BLOCK(mismatch_ex, run)(int data_in_0, int
   data_in_1, int& data_out) {
        Block_0(data_in_0, data_out_1, data_out);
        Block_1(data_in_1,data_out_1);
    }
private:
    #pragma hls_design interface
    void Block_0(int data_in, ac_channel<int> &feedback_in,int
     &data_out){
        int feedback_tmp = 0;
        int data_out_tmp =0;
        #ifndef __SYNTHESIS__
        //This part of the code is excluded from synthesis
        while (feedback_in.available(1))
        #endif
            feedback_tmp = feedback_in.read();
        }
        data_out_tmp = (feedback_tmp + data_in);
        data_out = data_out_tmp;
    }
    #pragma hls_design interface
    void Block_1(int data_in, ac_channel<int>& data_out) {
        data_out.write(data_in * 2);
    }
};
```

Listing 4.2: Example of a module where a mismatch is present

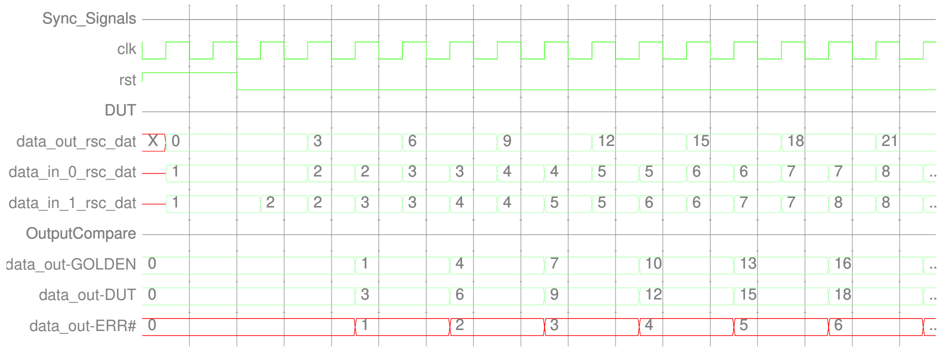


Figure 4.6: Simulation of listing 4.2

```
class mismatch_ex {
    ac_channel<int> data_out_1;
public:
    mismatch_ex() data_out_1(1, 0){} //channel is pre-filled
    ...
```

Listing 4.3: Proposed fix for the mismatch present in listing 4.2

4.5 Deadlocks

A deadlock is a phenomenon that occurs when the system cannot output any data because one or multiple parts of the network are unable to proceed due to a certain condition that is not met [16]. This condition is usually satisfied by other blocks that are mutually dependent on the latter. Thus creating a loop that does not allow the system to continue its operation. Although a deadlock might appear in a hardware simulation, it is likely that the software simulation produces the expected result.
One situation where a deadlock might occur is during the re-convergence of data in the system. Reconvergence happens when data traverses through separate blocks and then converges back together. This can be observed in Figure 4.7. The system can operate continuously only if the latencies of blocks 1 and 2 are equal. When this condition is not met, the system would have to stall for the module with higher latency to produce its output. Subsequently, block 0 will also cease its operation, as its output is now blocked. Therefore, the entire system will fall into a state of inactivity.
A way to solve this issue is by adding additional FIFO registers to the block with the lower latency to align the data sent to block 3. In Catapult HLS this can be done by creating a directive that manually adjusts the size of the `"ac_channel"` that connects to the final block [12].

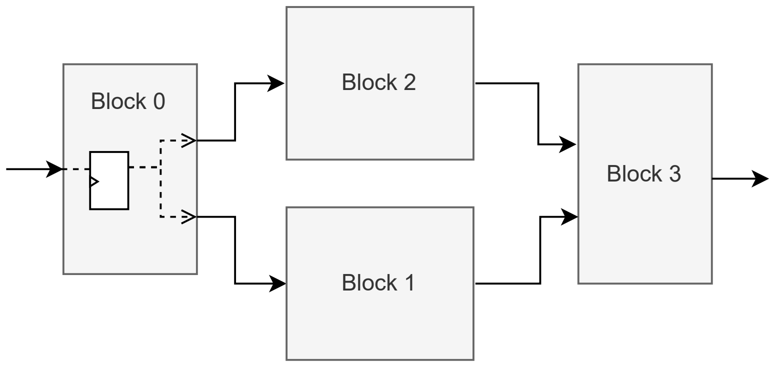


Figure 4.7: Reconvergence of a data stream

Another occasion that deadlocks might be experienced is when feedback registers are utilized, seen in Figure 4.8. As mentioned in section 4.4, feedback channels must always be pre-filled; otherwise, a hardware-software mismatch is caused. If the design's latency is larger than 1, it needs to be taken into consideration. The number of pre-filled data must be equal to the feed-forward latency. For example, when it takes 3 clock cycles to produce an output, the FIFO needs to be filled with 3 different sets of data. If that is not the case, then the feedback FIFO will not have enough data for the loop to complete.
During the implementation of the design, the problems described above did occur and a way to trace the issue was required. SCVerify provides a signal for each

block that displays its operational status; these waveforms can be examined in Questasim. By doing this, it is possible to see which block stalls first, thus finding where the issue resides. However, in pipelined systems, all blocks may cease simultaneously. One additional approach that can be used to identify deadlock sources, is by commenting out suspected parts of the code and re-synthesising the design. Although, this is a very time-consuming process and should only be used as a last resort.

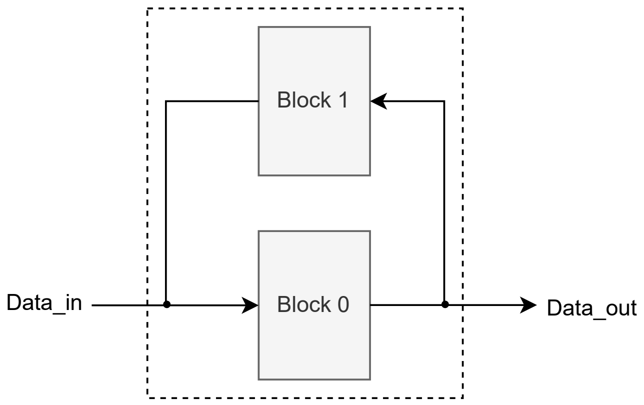


Figure 4.8: System with feedback where a deadlock might occur

4.6 Fault tracing

Most troubleshooting primarily occurs during the software implementation stage. However, in certain situations, there is a need to visualize a signal within the hardware simulation environment. This can be necessary either to gain a better understanding of the signal's status or to investigate a suspected mismatch. One limitation that arises, is that during RTL generation Catapult HLS optimizes all functions inside the blocks. As a result, any internal variable declared in C++ is not accessible in Questasim. To address this issue, Catapult provides the `"ac_probes"` library. This tool enables the preservation of a variable after synthesis, making it possible to observe it in the Questasim simulation. Finally, a schematic view of the design is also available, via the Design Analyzer module contained in the platform.

Chapter 5

Results

5.1 Area from both designs

After synthesizing the design and the RTL design the area which was observed is stated in this section. The synthesis tool provides information about the total area of design, including the sequential, combinational and memory components. The last is the size of all the external memory modules that may have been added to the design. Moreover, a section called "Total area excluding Macros" is included, which states the size of the design without any external memory IPs. Lastly, the total number of cells, flip-flops and gates is also provided.
As stated in Section 3.5, two different versions of the design underwent synthesis. A comparison was made between both versions and the RTL design. The first version utilized an embedded register-based memory, as indicated in Table 5.1. The second one featured an external memory port, but no memory IP was connected to it, as shown in Table 5.2.

Table 5.1: Area report of the RTL and HLS design with register based memory

	RTL	HLS
Total area	1309 um^2	1736 um^2
Number of cells	21078	9535
Total area excluding Macros	1309 um^2	611 um^2
Combination block area	367 um^2	181 um^2
Sequential block area	834 um^2	359 um^2
Number of edge triggered flip-flop cell	5083	2177
Memory area	1125 um^2	0 um^2
Combinational gatecount	14009	6922
Complete gatecount (comb. + seq.)	49925	66232

The distribution and comparison of the areas within the HLS design with externalized memory and the traditional RTL design can be seen in Figure 5.1. To gain a clearer understanding, the RTL design's RAM area is also shown in the

Table 5.2: Area report of the RTL and HLS design with externalized memory

	RTL	HLS
Total area	1309 um^2	775 um^2
Number of cells	21078	14210
Total area excluding Macros	1309 um^2	775 um^2
Combination block area	367 um^2	264 um^2
Sequential block area	834 um^2	449 um^2
Number of edge triggered flip-flop cell	5083	2755
Memory area	1125 um^2	0 um^2
Combinational gatecount	14009	10062
Complete gatecount (comb. + seq.)	49925	29558

chart. Subsequently, Figure 5.2 shows the comparison between the RTL design and the HLS design with externalized memory. In this chart the total area comparison does not include the RAM section.

5.2 Power analysis

After synthesizing the HLS design with the included register based memory and the RTL design, the power utilization which was observed is stated in this section. This data is presented in Table 5.3. The HLS design version with the externalized memory is not included, as it was not deemed possible to obtain power measurements with a connected RAM module to the design.

Table 5.3: Power comparison of RTL and HLS design with register based memory.

	RTL	HLS
Cell Internal Power	0.2916 mW	2.0217 mW
Driven Net Switching Power	0.00481 mW	0.1752 mW
Cell Leakage Power	392.6474 nW	130.1461 nW
Total Dynamic Power	0.2964 mW	2.1968 mW

It is noted that there is internal, switching, and leakage power utilized in the HLS design and the traditional RTL design separately. As seen in the Table 5.3, cell interval power refers to the power which is being used in the cells in the ASIC over an interval of time. Driver net switching power refers to the power being consumed by the design for switching the nets present in the design. Both are combined and presented as the Dynamic power consumed by the design. This refers to the consumption of total power when the design is active. Another measurement that can be noted in the results is "Cell leakage power" which is also called static power, it refers to the power consumed when the design is not active and in a static state. The comparison of these values is also visualised in Figure 5.3.

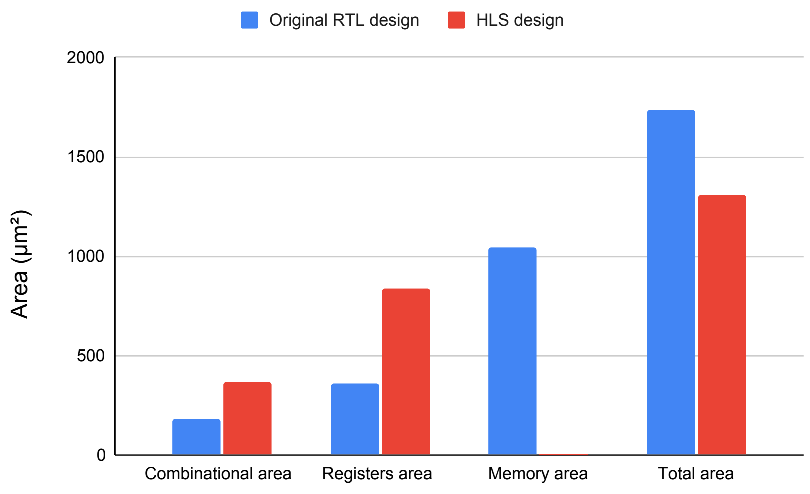


Figure 5.1: HLS design with register based memory and RTL design area comparison

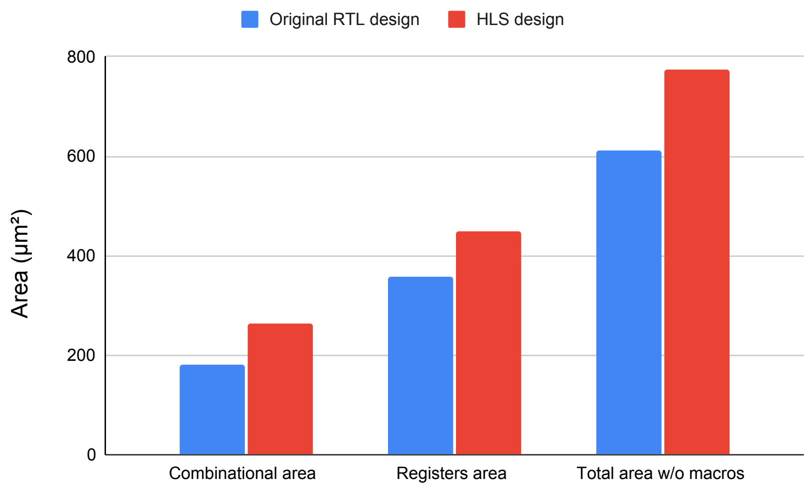
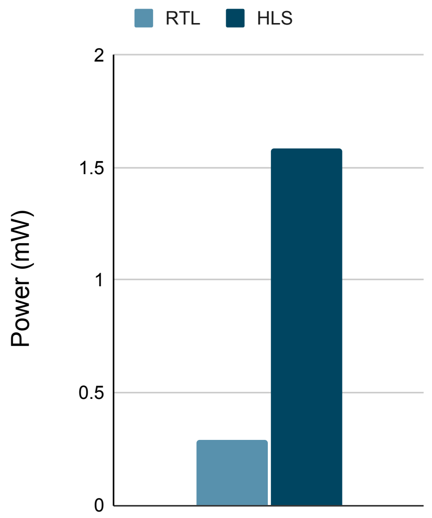
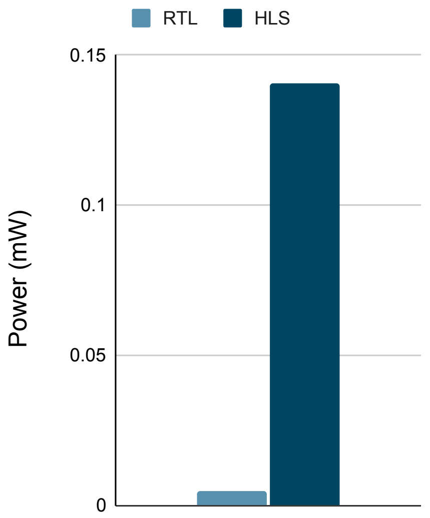


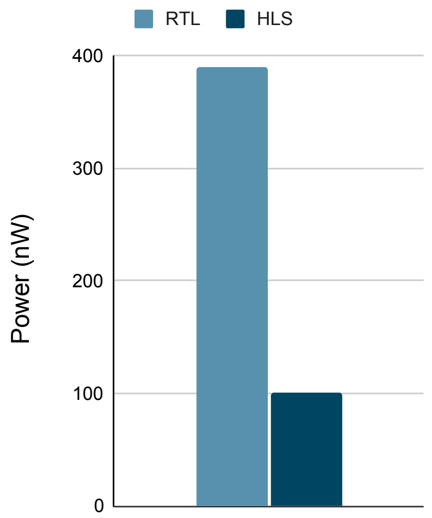
Figure 5.2: HLS design with external memory and RTL design area comparison



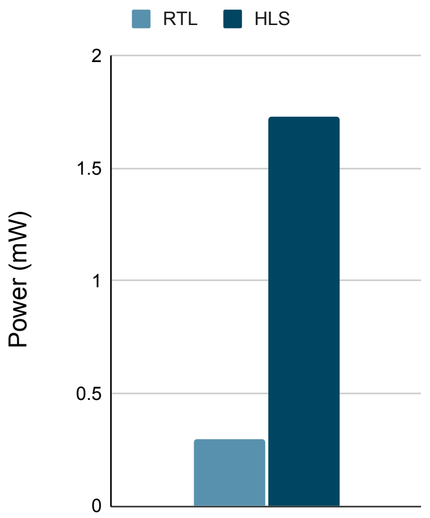
(a) Internal power



(b) [Switching power



(c) Leakage power



(d) Total power

Figure 5.3: Power comparison of HLS design with register based memory and RTL design

5.3 Interpretation

From the synthesis results in the Table 5.2, and Figure 5.2 it is noted that without the inclusion of the RAM section, the HLS design covers approximately 26% more area than the original RTL module. The combinational part of the HLS circuit was larger by 36% and the sequential part was 25% greater. This can be attributed to the fact that the functionality of the design was prioritised over area, thus the design is not as optimised as the RTL module. Additionally, a contributing factor for the increased sequential area is the added pipelining registers.
Furthermore, the area comparison in the Table 5.1 and Figure 5.1 indicates that the HLS design exhibits an increased quantity of flip-flops, this is due to the addition of pipelining registers and the substitution of the RAM module with register-based memory, leading to the addition of more sequential elements. Although the RAM module used in the RTL design is greater in area that the register based memory, despite having the same width and number or words.
It is also important to acknowledge, that the verification goals did not aim to achieve 100% functional coverage, as this process is highly time-consuming and falls beyond the scope of this thesis. Consequently, some minor variations between the functionalities of the two designs are expected. These discrepancies can also contribute to the differences in area and power consumption.
The comparison of power characteristics also indicates large differences, in both static and dynamic power. The latter can be attributed to the presence of the "Dynamic Random Access Memory(DRAM) type" memory module, as DRAM cells have higher leakage than Static Random Access Memory(SRAM) cells [17]. Furthermore, it should be noted that the module produced from the Catapult platform has not gone through any power optimizations.

Chapter 6
Discussions

6.1 Conclusion

The objective of this thesis was the implementation of an ASIC interface module in a High-level synthesis platform. The design chosen for this purpose was an I3C controller used for commercial applications, as it is mainly comprised of complex control logic elements, and its actions are highly time-dependent.
To achieve this, a workflow encompassing the transition from C++ design to verification was established. Moreover, the project presented an opportunity to identify numerous challenges that are likely to arise during the creation of such designs, and potential solutions were proposed to overcome them. The synthesized I3C controller was later compared with an already existing module on where it was based on. The goal of this comparison was to observe any notable differences in terms of area and power consumption.
As it was not feasible to synthesize the design within the HLS platform, any external elements such as the RAM could not be included during synthesis. Thus, two versions of the design were created. One were the RAM was replaced with register-based memory, and another with an externalized memory port. Additionally, the modules created in HLS did not pass through the same verification stages as the original design as this was considered outside the scope of this study and would require much more additional time and resources. Consequently, these factors should be taken under consideration when comparing the area and power results. Nevertheless, it was proven that the creation of such modules in an HLS environment is possible and most of the issues presented can be overcome.

6.2 Future scope

To expand on this study, a few proposals are presented, which cam be the focus of future investigations.
A SystemC design might be more suitable for clock-accurate modules. This high-level language allows the designer to probe the status of the clock. Therefore this approach might provide higher timing accuracy when implementing clock cycle counters [18].
For power consumption studies, the Catapult HLS platform is capable of integrat-

ing power optimization tools such as PowerPro or Catapult Ultra [6], this can offer improved power characteristics than a simple synthesized design. Subsequently, a dynamic power analysis could be utilized.
Both designs can be tested on a similar scenario over time, to capture the variations in power consumption based on the circuit's activity and timing characteristics. Therefore, a more precise power estimation will be acquired.
To conclude, by porting the used technology library to Catapult HLS, it is possible to include memory in the design, have more precise control over the timing constraints, and reduce implementation time.

Reference

[1] Siemens Digital Industries. (2021). High-Level Synthesis (HLS): status, trends and future directions [White paper]. Retrieved from https://resources.sw.siemens.com/en-US/white-paper-high-level-synthesis-hls-status-trends-and-future-directions.

[2] A. Takach, "Design and verification using high-level synthesis," 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), Macao, China, 2016, pp. 198-203, doi: 10.1109/ASPDAC.2016.7428011.

[3] S. Lahti, P. Sjövall, J. Vanne and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 5, pp. 898-911, May 2019, doi: 10.1109/TCAD.2018.2834439

[4] Lahti, Sakari & Vanne, Jarno & Hämäläinen, Timo. (2016). "Designing a clock cycle accurate application with high-level synthesis. 4756-4761. 10.1109/IECON.2016.7793783.

[5] Michael Fingeroff. 2010. High-Level Synthesis Blue Book. Xlibris Corporation.

[6] Catapult High-Level Synthesis and Verification, Siemens Digital Industries Software, 2020 [Online]. Available: https://resources.sw.siemens.com/en-US/fact-sheet-catapult-high-level-synthesis-and-verification

[7] MIPI I3C Basic Specification, Version 1.1.1, MIPI Alliance, June 2022. [Online],Available: https://www.mipi.org/mipi-i3c-basic-download

[8] AMBA APB Protocol Specification, ARM IHI 0024E, ARM Limited, Feb. 2023.[Online], Available: https://developer.arm.com/documentation/ihi0024/e/?lang=en

[9] Lund university Digital ASIC Group, A Tutorial on the Design Flow (2005). Accessed: Apr.19, 2023. [Online]. Available: https://www.eit.lth.se/fileadmin/eit/courses/etin01/manual$_{e}$*tc/dasic.pdf*

[10] ChipsMedia: design and verification of deep learning object detection IP, Siemens Digital Industries Software, 2021 [Online]. Available: https://resources.sw.siemens.com/en-US/white-paper-chips-and-media-design-and-verification-of-deep-learning-object-detection-ip

[11] P. P. Chu, RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability. Hoboken, NJ: Wiley-Interscience, 2006.

[12] Algorithmic C (AC) Datatypes Reference Manual, v4.6.1 , SIEMENS, 2022

[13] G. Kahn, “The Semantics of a Simple Language for Parallel Programming.,” pp. 471–475, Jan. 1974.

[14] H. Vuopio, "C++ CODING PRINCIPLES FOR HIGH-LEVEL SYNTHESIS," Master’s thesis, Faculty of Information Technology and Electrical Engineering, University of Oulu, Oulu, Finland, 2021. [Online]. Available: http://jultika.oulu.fi/files/nbnfioulu-202109098961.pdf

[15] B. Prasad, “Sequential versus concurrent engineering—an analogy,” Concurrent Engineering, vol. 3, no. 4, pp. 250–255, 1995. doi:10.1177/1063293x9500300401

[16] J. L. Hennessy, D. A. Patterson, and K. Asanovic, Computer Architecture: A Quantitative Approach, 6th ed. Cambridge (Estados Unidos): Morgan Kaufmann, 2019.

[17] J.M. Rabaey,Anatha Chandrakasan, Borivoje Nicolic, “DIGITAL INTEGRATED CIRCUITS ,” A DESIGN PRESPECTIVE, 2nd ed. London, United Kingdom:Pearson Education inc., 2016.

[18] Systemc Golden Reference Guide. Hampshire, Angleterre: Doulos, 2005.