Ludvig Samuelsen Jordet

Creation of a software programmable hardware interface

Master's thesis in Computer Science
Supervisor: Magnus Själander
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Ludvig Samuelsen Jordet

Creation of a software programmable hardware interface

Master's thesis in Computer Science
Supervisor: Magnus Själander
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

During the work with this thesis, two separate hardware designs were created. The first, a configurable component implementing both UART, SPI and I^{2}C, was created to see whether this would gain any advantage over separate components in core size, power usage or pin usage. The suggested design does gain a small improvement in estimated power usage from Vivado reports, however the core size is in fact a bit larger than three separate components combined. In addition, limitations of the design means that I^{2}C is the only interface fully supported, with both UART and SPI having limitations not present in the alternative, separate cores. With this taken into account, as well as somewhat uneven measurement grounds (I/O bus communications were not included in the configurable core reports), the author concludes that any advantage of this specific design is not enough to call this an improvement over using separate components.

The second hardware design is the GenIE, or generic interface engine, a fully programmable interface controller able to implement arbitrary interfaces by reprogramming. The GenIE is implemented as an ASIP, or application specific instruction set processor. This was created to see whether or not a programmable component could gain any further advantages over the configurable component. With regards to the metrics focused on for the configurable component, no definitive answer was arrived at. This was mainly because the programmable core uses an external memory, which was not considered in the scope of this thesis. However, the increased flexibility of the GenIE, with the ability to implement different interfaces by changing its program, is an advantage over more fixed components, as this makes it possible to use interface not originally available on the system.

Sammendrag

Gjennom arbeidet med denne oppgaven ble det laget to forskjellige maskinvaredesign. Den første var en konfigurerbar komponent som implementerte både UART, SPI og I^{2}C. Denne ble laget for å finne ut hvorvidt dette kunne ha noen fordeler over å bruke separate komponenter, med fokus på størrelse, strømforbruk og pin-bruk. Det foreslåtte designet har en liten forbedring i estimert strømforbruk fra Vivado-rapporter, men størrelsen er litt større enn summen av tre separate komponenter. I tillegg gjør begrensninger i designet at kun I^{2}C er støttet i sin helhet. Både UART og SPI har begrensninger som ikke finnes i de alternative, separate komponentene. Basert på dette, samt litt ujevne målinger (I/O-busskommunikasjon var ikke inkludert i rapportene om den konfigurerbare komponenten), konkluderer forfatteren med at eventuelle fordeler med dette spesifike designet ikke er nok til å kalle dette en forbedring fra å bruke separate komponenter.

Det andre designet er GenIE, eller Generic Interface Engine. Dette er en fullt programmerbar grensesnitt-kontroller som kan implementere vilkårlige grensesnitt ved å omprogrammeres. GenIE er implementert som en ASIP, application specific instruction set-prosessor. Den ble laget for å finne ut hvorvidt en programmerbar komponent hadde noen videre fordeler kontra den konfigurerbare komponenten. Med hensyn til fokusområdene for evalueringen av den konfigurerbare komponenten ble det ikke funnet noe definitivt svar. Dette i hovedsak fordi GenIE er avhengig av eksternt minne, som ikke ble regnet med i omfanget av denne oppgaven. Likevel er GenIEs økte fleksibilitet, med mulighetene til å implementere forskjellige grensesnitt ved å endre programmeringen, en fordel over mer låste komponenter, da dette muliggjør bruk av grensesnitt som ikke opprinnelig er støttet av systemet.

Contents

Chapter 1

Introduction

This thesis will attempt to answer the following research questions:

1. Can the OpenCores I2CMST I^{2}C master core be modified to support UART and SPI, in order to gain any significant advantage over using separate cores with regards to power consumption, core size or pin usage?
2. Does a further extension of the core with a flexible, programmable protocol controller to support arbitrary wire-compatible protocols have any significant disadvantages over using more rigid cores? Does this open possibilities not available to more narrow purpose cores?

Answering the second question will constitute the most novel part of this thesis, a fully programmable hardware interface controller.

1.1 Motivation

Communication between different devices in a system is commonly done using a variety of different hardware interfaces. Three of the most popular such interfaces are UART, SPI and I2C. They are very commonly used for communication with internal and external peripherals, both as a part of a System on Chip or as discrete components in a more traditional system. Such interfaces are usually implemented by dedicated hardware components, each implementing a single interface. In this thesis, the author explores the idea of combining the functionality of several such components into a single generic interface component, in order to gain an advantage over using multiple dedicated components.

The research questions mention three specific areas of potential improvement over dedicated cores. First of these is the area of power consumption, which is an important constraint both in battery powered and high performance computers. Battery powered devices consists of mobile phones and other portable computers, as well as the increasing amount of embedded computers powering the Internet of Things. The battery life of these devices depend directly on the power consumed by the device. In addition to battery powered devices, lowering power consumption has been increasingly important in high performance architectures. According to [1], energy is the key limiter of performance. This has led to increasingly parallel architectures, both traditional homogeneous CPUs as well as heterogeneous parallelism. The latter includes the use of accelerators, which are fixed function, programmable or otherwise configurable cores such as cryptography engines, hardware codecs and more dynamic accelerators like FPGAs, optimized for a specific application. Creating a generic interface controller consuming less power than a number of separate controllers will then pose an advantage both to battery powered and other systems.

Core size is the number of transistors used by the core when implemented, or in other words the area used by the cores logic on the chip. Transistor density has been increasing according to Moore's Law for decades. However, transistor scaling is reaching a limit. The threshold voltage is reaching a limit, with leakage currents increasing with decreasing transistor size. In addition, the inability to decrease voltage with increasing transistor density has driven up power consumption, which again leads to performance being limited further by energy [1]. As such, decreasing the number of transistors used is increasingly important, and decreasing the amount of transistors used for hardware interfaces will contribute to this.

The third are for improvement is pin usage. As transistor density has increased, the number of I/O pins per transistor in an integrated circuit has decreased. Reducing the number of pins reserved by hardware interface cores will free pins for other usages.

In addition to these potential improvements over using dedicated interface cores, a fully programmable generic interface might provide the advantage of increased flexibility. As the interface can be programmed and potentially reprogrammed according to application requirements, this might open the possibility of adding support for interfaces on an ad-hoc basis. This will make it possible to defer decision on which interface(s) to support and use. If the programmable interface exposes a unified and static control interface to the system, being able to change the interface implementation without large changes to system software might be possible.

1.2 Research method

In order to asses the feasibility of achieving the improvements outlined in the research questions, an implementation of both the extended I2CMST core and a fully programmable core (called GenIE, or Generic Interface Engine) will be created. They will be implemented using VHDL, and simulated using the simulator of Vivado, a VHDL environment by FPGA manufacturer Xilinx. These VHDL implementations can, in addition to functional simulation, be synthesized for implementation on a Xilinx FPGA. This will make it possible to access metrics on power, speed and area requirements for each implementation.

In addition to the implementations, a theoretical discussion of the implementations and potential alternative implementations will be done. Advantages, disadvantages and areas for further improvement and research will be discussed. This will provide a broader view of the potential solutions than the implementations alone can provide.

Based on results of the implementations along with the theoretical discussion, a conclusion will be drawn. This will attempt to answer the research questions mentioned earlier in this chapter on the basis of the results of the implementation experiment and the discussion.

1.3 Thesis structure

Chapter 2

Background theory

2.1 Protocols

Several serial protocols and interfaces are in use for communication between peripherals and other devices in a system. UART, SPI and I^{2}C are three of the most popular ones, and this section contains a presentation of each of them.

2.1.1 UART

An UART (universal asynchronous receiver-transmitter) is a simple component for serial communication. It is an asynchronous serial interface, using two wires for full-duplex serial communication, or a single wire for simplex or half-duplex communication.

An UART data line is for historical reasons held high when idle. A transfer is initiated by the line going low for one bit (this is called the start bit), then the bits are sent in order, optionally followed by a parity bit for error checking. The transfer is completed by one or more high bits called the stop bits, indicating that the transfer from the transmitter is complete.

The receiving end waits for the signal to go low, indicating a start bit. If this is maintained for at least half the bit-time, the receiver waits another bit time and starts sampling values at bit time intervals, until reaching the stop bit(s). If a stop bit is not encountered at the expected time, this indicates a framing error, typically handled as a status signal in the receiving UART.

UARTs do not share a clock, both receiving and transmitting end are clocked independently. The clock rate is typically a multiple of the bit rate, to be able to identify the middle of the start bit and sample the following bits at the correct time. Word length, bitrate (often referred to as baud rate), use of parity bit as well as the number of stop bits are all configuration parameters which need to be set to matching values in both the transmitting and the receiving UART.

An UART can be implemented using a shift register in each end, the transmitting UART shifting bits out of the register onto the line, and the receiving end sampling the signal and shifting bits into the shift register. Since an UART transmission is asynchronous, the transmission can be started immediately when data is written to the register.

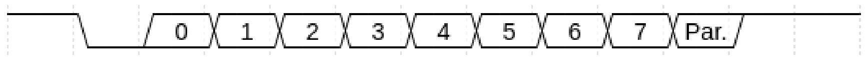

Figure 2.1: Timing diagram for the transmission of 8 bits and a parity bit using an UART with two stop bits.

An UART is commonly used to back serial ports such as RS-232, a protocol specifying voltage levels for signals used for serial data transfer between devices.

2.1.2 SPI

The SPI (Serial Peripheral Interface) does not have any readily available formal specification. It was implemented by Motorola in their 6800 series microcontrollers, and this implementation constitutes the de facto standard reference implementation of the interface. [2]

SPI is a four-wire, synchronous serial interface, operating in full duplex mode. The wires are SCLK, MOSI, MISO and $\overline{\text{SS}}$, where MOSI (master out, slave in) and MISO (master in, slave out) are being used for data transfer in each of the directions, SCLK is the interface clock, and $\overline{\text{SS}}$ is the slave select, used when there are multiple slaves connected to a master. SPI supports only a single master. To support multiple slaves, the $\overline{\text{SS}}$ is driven low only for the slave which is to transmit and receive data. The MOSI should only be driven by a slave if its $\overline{\text{SS}}$ is held low. For this reason, the MOSI is usually a tri-state driver which is held in high-impedance mode whenever the $\overline{\text{SS}}$ is high.

Data is transferred by the master issuing clock cycles on the SCLK line. Data is then transferred in full-duplex mode, with one bit being transferred on each of the MOSI and MISO lines each clock cycle. The timing of reads and writes on the data lines is governed by the so-called CPOL and CPHA settings, governing the polarity and phase characteristics of the clock used for the timing. The CPOL controls the polarity of the clock signal. CPOL=0 indicates that the clock is idle at 0, and goes to 1 for each cycle. CPOL=1 is the inverse behavior, with the clock line idle at 1 and pulses to 0. The CPHA controls where in a clock cycle the reads and writes take place. With CPHA=0, the data on the lines changes on trailing edges (falling edges with CPOL=0) and are read on leading edges (rising edges with CPOL=0). In this situation, the first bit of data must already be available on the data lines at the time of the first leading edge of SCLK, typically the bit is made availble at the time $\overline{\text{SS}}$ is pulled low. If CPHA=1, the data is read on the trailing edges, and changes on the leading edges. The last bit of data must then be kept available for at least a half period after the final trailing edge. A timing diagram of an 8-bit transfer with CPOL=0 and CPHA=0 can be seen in Figure 2.2

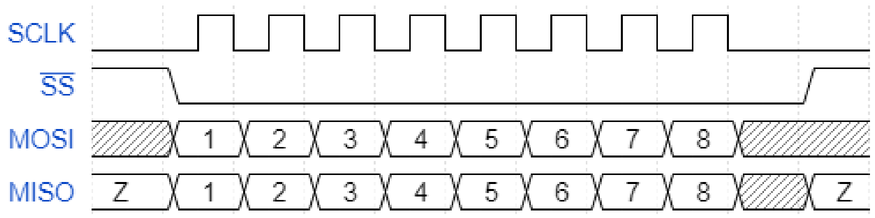

Figure 2.2: Timing diagram showing transfer of 8 bits in both directions over an SPI connection, with CPOL=0 and CPHA=0.

The data transfer in SPI is often implemented using a pair of shift registers, one in the master and one in the slave, clocked to the SCLK. On each clock cycle, one bit each is shifted from the master to the slave and vice versa via the MOSI and MISO lines. This continues until the entire contents of the shift register on each side is shifted to the other side. The entire word is then available, and the registers can then be reloaded with more data until the entire transfer is complete. The rate of the SCLK is controlled by the SPI master.

2.1.3 I^{2}C

I^{2}C (Inter-Integrated Circuit) is a serial bus developed by Philips Semiconductor [3]. Its mainly used for low speed communication between peripherals and CPUs on a single board and/or over low distances. Unlike SPI, I^{2}C supports multiple masters as well as multiple slaves, with each device functioning as either a master or a slave at any given time. It uses only two lines, SDA and SCL, and supports transfer speeds of up to 100 kbit/s in standard mode and up to 3.4 Mbit/s in faster modes. Unlike SPI and UART, I^{2}C has a significantly more complex transmission protocol, to support more complex features. Each device is able to function as either a master or a slave, either of which can transmit at a given time. In addition, features like arbitration and clock stretching also demand a more full-featured bus protocol.

Physically, I^{2}C uses two shared lines, SDA for data and SCL for clock. Both must be connected to pull-up resistors to pull the lines high whenever no device is controlling them, making the lines an open-drain design, also referred to as wired AND[4]. This means that each device drives a line low to transmit a low signal, and lets the line float and the pull-up resistor pull it high to transmit a high signal. If any device(s) drive the line low, they “win”, and the line will be at the low level. This fact is used for synchronization and arbitration in the I^{2}C protocol.

A data transfer on the I^{2}C-bus begins with a master creating a START condition, defined as a high to low transition on the SDA while the SCL is held high. Bytes are then transferred, bit by bit, by the master letting the SCL pulse high while bit values are written to SDA.

The protocol supports transferring bytes in either direction between any master and any slave. To differentiate between slaves, I^{2}C uses an addressing system, with each slave having a unique 7-bit address. The address of the slave for a given transfer, as well as an R/W bit controlling the direction of transfer (0 for writing data to the slave, 1 for reading data from the slave) is transmitted on the SDA line by the master as the first byte after the START condition.

After a byte (including the address) is transferred, the control over SDA swaps, and the receiving end transmits a single ACK or NACK bit. If the transfer was successful, the receiver drives the SDA low indicating an ACK. If for some reason the receiver could not successfully receive the byte, it does not drive the SDA low. This situation is called a NACK. Because of the open-drain design, if no device is available to receive the byte, this also results in a NACK situation.

After a transfer, the master can either generate a STOP condition by sending a low to high transition on SDA while SCL is high, or immediately send another START condition to keep control over the bus and start a new transmission of bytes using another address. A repeated START condition can also be used to change transfer direction in the middle of a transfer, by transmitting the same address with an inverted R/W bit.

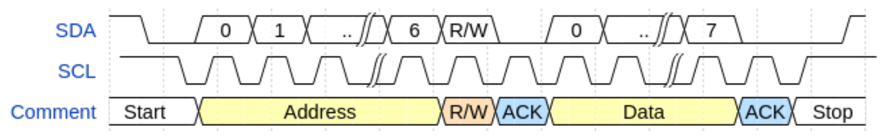

Figure 2.3: Addressing and transfer of a single byte using I^{2}C. The first ACK is sent by the slave. If R/W = 1, the data is then sent by the slave, and the second ACK by the master. If R/W = 0, the data is sent by the master, and the second ACK by the slave.

2.2 Existing solutions

The idea of a software configurable serial communications interface has been explored before. An UART is, as it is named, imagined as a Universal Receiver/Transmitter, with no regards to data format or electrical and physical characteristics of the serial transfer line. However, it is not flexible enough to be able to emulate neither SPI nor I^{2}C by itself, and does not solve the problem this thesis seeks to solve.

Texas Instruments provide microcontrollers with a software configurable universal serial communication interface (USCI)[5]. This can be toggled to function as an UART, an SPI or an I^{2}C-interface, respectively, by writing to special control registers. The data transfer registers are used the same way regardless of which configuration is specified.

2.2.1 Bit-banging

For some interfaces, particularly those with simple physical specifications and which can run at arbitrary or slow rates, the need for a dedicated hardware interface component can be replaced by *bit-banging*[6]. This is a technique where general purpose I/O pins are toggled manually by the software to simulate the behavior of a specific interface. It is limited by the CPU speed and spends CPU cycles on toggling I/O pins. In addition, it is not readily applicable where the interface requires specific electrical behavior driving the pins, such as I^2C requiring the use of high impedance and pull-up resistors.

2.3 Tri-state buffers

Normal logic signals can only be in one of two states, normally referred to as ‘0’ and ‘1’. Tri-state logic extends this by also allowing a signal to exist in a high-impedance state. This would make it seem like the signal is not connected to its output at all.

Tri-state logic is commonly used when several devices are sharing a single line, where only one of them should drive the line at any given time. By setting all outputs except the one driving the line to the high-impedance state, this can be accomplished easily.

Inference of tri-state buffers on outputs can be accomplished quite easily using VHDL by use of the special signal value ‘Z’. This indicates that the wire should be held at the high-impedance state. An example:

```vhdl
d_out <= shift_out when ss = '0' else 'Z';
```

This will drive the `d_out` with the value on `shift_out` whenever `ss` is held low, and keep `d_out` at a high impedance state otherwise.

2.4 Transport triggered architectures

Transport triggered architectures[7], or TTAs, are a superclass of VLIW instruction set architectures. They work on the principle of controlling the processor transport mechanisms directly, instead of through the operations used in more typical CISC or RISC instruction set architectures. The instruction level parallelism available using a TTA enables more direct control of transport utilization and scheduling, as well as allowing short processor cycle times and simple processor design.

Transfer between registers is the only operation available on a TTA processor. An instruction typically includes source and destination for each of the available transport lines, and as such the instruction word length depends on the number of transport lines available and the source and destination identifier size. Transfer of a value from one register to another is then a very simple operation in a TTA.

Other operations are available in a TTA processor through the use of *functional units*. A functional unit is for instance an ALU. To perform an operation on the ALU using a TTA, the necessary operands are transferd to the ALU registers, and the operation is *triggered* by writing a value to a special operation triggering register. The result will then be made available on the ALU result register.

An example of a very simple TTA with two transport lines and ALU with an ADD and SUBTRACT operation has the following instruction format and registers:

Field	Source 1	Destination 1	Source 2	Destination 2
Bits	4	4	4	4
Description	Src 1 $\rightarrow$ Dest 1, Src 2 $\rightarrow$ Dest 2			

Table 2.1: Instruction format in a very simple TTA

Register	Description
0-3	General purpose
4	ALU Operand 1
5	ALU Operand 2 ADD trigger
6	ALU Operand 2 SUBTRACT trigger
7	ALU Result

Table 2.2: Registers in a very simple TTA

Computing the sum of registers 1 and 2 and putting the result in register 0 can then be done with the following instruction sequence:

```
// Write operands and trigger ADD operation
1 -> ALU Operand 1, 2 -> ALU Operand 2 ADD trigger
// Move result to register 0, no-op on second line
ALU Result -> 0, 1 -> 1
```

2.5 OpenCores I2CMST

The OpenCores I2CMST was used as the basis for the configurable core. An overview of the I^{2}C master core from grlib can be seen in Figure 2.4. This is a lightly modified version of the OpenCores I2CMST, mainly modified to use AMBA as the system bus. [8]

The OpenCores I2CMST core[9] is composed of a shift register for actual data, and a Byte Command Controller and Bit Command Controller for dealing with the specifics of the I^{2}C protocol. The controllers are controllable through command and status registers, which along with data transmit and receive registers

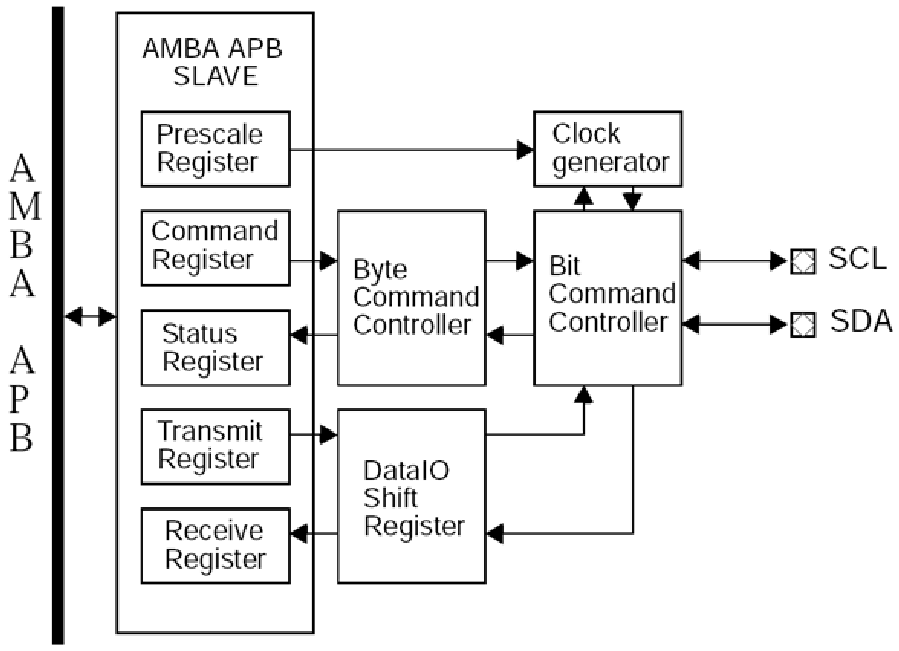

Figure 2.4: An overview of the GRLIB I^2CMST core

are accessed through a Wishbone bus interface. The core also includes a clock generator, with a prescale register for controlling the clock speed used for I^2C operations.

The interface between the bit controller and the rest of the core consists of a set of command lines for communicating with the byte controller, two 1-bit wide data lines for input and output with the DataIO shift register, and a clock input from the clock generator. In addition it provides both input, output and output enable signals for the SCL and SDA signals.

Communication between the byte command controller and the bit command controller is done through two signals, a command line from the byte command controller and the bit command controller and an ack bit from the bit command controller to the byte command controller. The available commands are:

- NOP - Do nothing
- START - Create the I^2C start condition
- STOP - Create the I^2C stop condition
- READ - Read bits from the slave
- WRITE - Write bits to the slave

The ack bit is set to 1 to signify that a bit is done transmitting, and the bit command controller is ready for the next bit of data.

In addition to the command and ack signals, two error bits are also available for the bit command controller to signify that it was not able to complete a requested command. These are the `busy` and `al` signals, indicating that the I^{2}C bus is busy or that bus arbitration was lost, respectively, in a multi-master environment.

2.6 Overview of proposed solution

The proposed solution to the problem is divided into two parts, one for each of the research questions. Both proposed solutions are in the form of one or more synthesizable hardware cores. An overview of a valid core design for answering each of the research questions is described here, along with considerations in implementation, testing and evaluation of each of the proposed designs.

2.6.1 Configurable core

To answer research question one, the task is to modify the OpenCores I2CMST core to also be able to communicate using UART or SPI. In order to answer the research question, the component should be able to accomplish the following:

1. When configured as I^{2}C, the component should behave in the same way as the original I2CMST. This includes both the actual I^{2}C communications as well as the interface against the system bus.
2. When configured as UART, the component should implement a valid and functioning UART transmitter implementation.
3. When configured as SPI, the component should behave as a valid SPI master.
4. Both when configured as UART and when configured as SPI, data can be written to and from the UART/SPI using the system bus interface.
5. (Re)configuration of the core for a specific interface should be possible through the system bus interface.

Note that the requirements omit support for behaving as an I^{2}C or SPI slave, or an UART receiver. The research question can be understood to also include the slave or receiver aspect of each of the protocols, but the author has decided to leave them out of scope for this thesis. The reasoning behind this is the fact that in the case of I^{2}C and SPI, the master provides the bus clock, the same being true for the transmitter when using UART communication. Introducing off chip clocks brings a lot of complexity into the proposed solution. In addition, the I2CMST only implements support for an I^{2}C master. Implementing an SPI master provides the equivalent capabilities in an SPI setup. In the case of UART, a transmitt-only core would, unlike the SPI and I^{2}C masters, not be able to receive data. Nonetheless, the decision was made not to support off-chip clocks because of the complexity involved.

In addition to the above requirements, in order for the proposed solution for the first research question to be of best use in answering the second research question, the following additional requirements were made:

6. The interface for transmitting and receiving data against the system bus should not depend upon which protocol the core is currently configured for. This will help create a protocol-agnostic data interface which can later be used for a programmable core.
7. Output signals for protocol communication should be available on a shared set of input/output signals for each of the protocols. In other words, there should not be specific I/O signals for each of the protocols. This has the immediate benefit of making the core more flexible as the outputs can be routed to I/O pins without regards as to which protocol(s) might be used. In addition, as a programmable core has no well-defined finite set of possible protocols, a shared set of I/O drivers able to provide all the requirements of any given protocol is a necessary feature of the protocol I/O of such a programmable component.
8. The command language of the I2CMST should be used, and if necessary modified, to provide for the needs of each of the specific protocol implementations. This has the effect of separating handling of the system bus and data I/O from the system side from the specific implementation of any protocol. The command language as well as any additional features of this interface introduced will then constitute a candidate for a general protocol-agnostic control interface, which can be used in a programmable core for answering the second research question. A more thorough description of such an interface is described in HVOR DA?

Each of the protocols has requirements for different aspects of a valid solution, outlined in the descriptions of the protocol specifications in section 2.1. The most important of these requirements are:

- The I/O drivers must be able to support tri-state logic, in order to support the shared data line of the I^2C specification.
- Clock speed requirements differ among the protocols, with I^2C having well defined rates in the specification, an UART using one of several common baud rates, and SPI operating on a frequency limited by the attached peripheral. A proposed solution must then be able to vary the clock rate used for I/O signaling.
- Both SPI and I^2C support multi-slave setups, which must be handled by a proposed solution. I^2C includes a concept of addressing, while SPI and UART do not have a concept of an address as part of the protocol. SPI, however, uses a slave select signal. The proposed solution must handle these differences while keeping the system bus interface the same across configuration.

- SPI has clock polarity and phase as implementation-specific, often configurable, details.
- I^2C requires support for multi-master setups.
- The phases of communication (with start/stop bits and conditions, actual data transfer etc.) vary between the protocols. The common command interface must span these differences.

2.6.2 Evaluating the configurable core

In evaluating a proposed solution with regards to research question one, several metrics should be considered, in order to judge whether the solution can provide an answer to the question. The question asks whether the I2CMST can be modified to support UART and SPI. Further, it mentions advantages over using separate cores in three areas, power consumption, core size and pin usage.

When deciding if a proposed solution is suitable for answering the research question, the perhaps obvious first step is to evaluate whether or not it correctly implements all the protocols. This includes any rate and timing requirements in the protocol specification, in addition to correct values and behaviour on the protocol signals.

Power consumption of a proposed solution can be compared with that of separate cores by comparing specific implementations, for instance through use of simulation tools or actual hardware measurements of implementations of both configurable and separate implementations in the same implementation fabric. In addition, a more general discussion with regards to the properties of the implementations can be done to evaluate the differences between them. When evaluating power consumption, factors like setup overhead and/or clock gating of inactive parts of the configurable core must be taken into account.

Core size can be easily measured by comparing synthesized versions of both separate core designs and the configurable design using an HDL synthesizer. When comparing the core size, the configurable core can be compared to both any single separate core, or multiple separate cores, in order to be able to discuss any trade-off effects of making a single configurable core.

Pin usage is also easily measured as a simple property of a specific hardware implementation of a solution. However, attention must be made to the fact that a common I/O stage used across multiple separate cores is a potential solution for this specific problem. This must be taken into account when evaluation a proposed solution.

2.6.3 GenIE (Programmable core)

The requirements of a programmable core suitable for answering the second research question are somewhat like the ones for the configurable core, but with some important differences. In order to have a clear definition of the requirements, they are described in total in this section, with a discussion following of differences from the configurable core requirements.

The programmable core should at the least be able to function as a drop-in replacement for the configurable core. This gives the following functional requirements:

1. Given a suitable program, the component should behave in the same way as the original I2CMST with regards to the actual I^2C communications.
2. The core should, given a suitable program, implement a valid and functioning UART transmitter implementation.
3. When programmed for SPI, the component should behave as a valid SPI master.
4. Data communications between the system and the hardware interface should be available over the system bus.
5. The program code used by the core should be able to be modified from the system side
6. The system interface for the core should be independent from the currently loaded program code. This requires a common interface implemented by every program which can run on the core.

This is mostly the same requirements as for the configurable core, reworded to suit a programmable design. The shared pin requirement from the configurable core makes little sense when talking about a fully programmable core, and is not included. System side reconfiguration of the configurable core is mirrored in the requirement of system side reprogramming of the programmable core. A common interface regardless of the currently active protocol/program is a development of the common data interface requirement of the configurable core, and the actual command design can be developed from the command design of the configurable core. However, if the programmable core is to be designed to be able to support as large a number of different possible protocols as possible, some redesign or augmentation may be required to make the interface even more general.

Again, the UART, SPI and I^2C specifications give some additional requirements:

- Any I/O drivers must be able to support tri-state logic, in order to support the shared data line of the I^2C specification.
- Clock speed requirements differ among the protocols, so this must be configurable from the system or program.
- The phases of communication (with start/stop bits and conditions, actual data transfer etc.) vary between the protocols. The common command interface must span these differences.

The latter of these can be augmented to support an even more generalized hardware interface, in order to increase the flexibility given by being able to reprogram the core.

2.6.4 Evaluating the programmable core

Evaluation of a programmable core can, much like the configurable design, be done using simulation or implementations in a hardware implementation fabric. Simulation or implementation in the same environment and/or fabric as the configurable core will make it possible to compare measurements and metrics between the two. This will make it possible to evaluate whether any advantage can be had in power consumption and core size.

With regards to the core size, the increased flexibility of a programmable core must be taken into account. The programmable core has the potential of implementing protocols other than the UART, SPI and I^2C available using the configurable core. This increased flexibility might outweigh any potential increase in core size, as further hardware cores may be made unnecessary by the programmable core. A discussion on any trade-off should be made.

Pin usage in a programmable core is, like the configurable core, simply an implementation detail of any proposed design, and can be reasoned about as such. The amount of I/O pins made available for programs will constrain the amount of possible interfaces, again a discussion should be done on trade-off between increased support for many I/O pins and increased core size, possible core complexity and power requirements.

A programmable core must be able to implement at least UART, SPI and I^2C at a minimum just as well as the configurable core. This can be proved by making programs implementing these protocols. In addition, the flexibility of the core with regards to implementing other potential protocols should be discussed.

Chapter 3

Component design

Two different component designs were created for this thesis. They are incremental steps in accomplishing the goal of having a flexible, programmable hardware interface component capable of communicating using a software-defined protocol. The first component is a configurable, but not programmable, component capable of communicating using UART, SPI or I^{2}C. The second component, GenIE, accomplishes the goal of a fully programmable core. A description of the design and implementation of each of the components follow.

3.1 Configurable core

To answer research question 1 of being able to use the same component for communicating using either UART, SPI or I^{2}C, the OpenCores I2CMST was chosen as a starting point. This was then extended to be able to communicate using UART or SPI in addition to I^{2}C. In addition, logic was added to be able to reconfigure the core for the different protocols.

In designing this component, a goal was set to keep the data interface as consistent with the I2CMST as possible. Any system use of the core should be fully agnostic of which protocol is being used, apart from the configuration setup calls.

An overview of the design can be seen in figure 3.1

3.1.1 Design

The configurable core design is an extension of the OpenCores I2CMST. A UART controller and an SPI controller component were created and added alongside the I2CMST bit controller, in the extended design referred to as the I^{2}C controller.

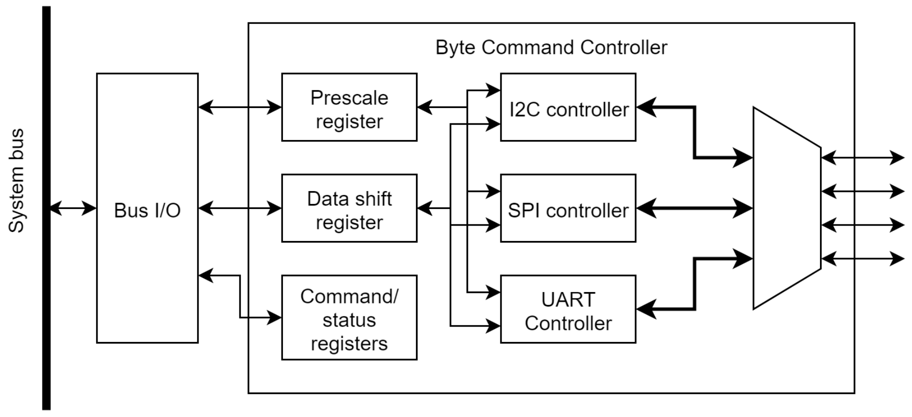

Figure 3.1: An overview of the design for the configurable core.

The I^{2}C controller, UART controller and SPI controller all use a common interface towards the Byte controller. This interface is a modified version of the interface between the Byte controller and the Bit controller, with added enable lines for each of the interface controllers. The outputs from each of the controllers is multiplexed in the output stage of the configurable core. An additional configuration register for selecting which interface to implement is added, writable from the system bus.

The UART controller and the SPI controller each implement a subset of the I2C commands sent by the byte controller. This was chosen in order to minimize time spent rewriting the byte controller.

3.1.2 Implementation

The OpenCores I2CMST contains a top level module, *i2c_master_top*. This is responsible for communication between a Wishbone bus and the rest of the I2CMST. For this thesis, the details of bus communication was seen as not important with regards to answering the research questions, so this top level module was not modified to support the configurable core, and was not used during simulation or reporting for evaluation purposes.

The I^{2}C controller, or OpenCores I2CMST bit controller, is left unmodified. All outputs from the I^{2}C controller, including the command ACK, are multiplexed with the outputs from the other controllers in the byte controller.

The UART controller is a fairly simple module. It responds to START and WRITE commands, with the START command sending the start bit and immediately ACKing the command. The WRITE command is a thin wrapper around the data output from the byte controller, relaying the data output for 8 cycles, then transmitting the stop bits. The number of stop bits is controlled by a flag signal from the byte controller, controlled via the system bus. This makes it possible to select 1 or 2 stop bits. All other commands immediately

ACK while functioning as a no-op. This makes the UART support the same command language as the I^2C controller, meaning the core can be controlled the same way without regards to which protocol is used.

The SPI controller only supports the WRITE and READ commands. As SPI is full duplex, both these commands perform the same action, namely a bi-directional transfer of 1 8-bit byte in both directions. Like the UART controller, it responds to all other commands by immediately ACKing the command. The CPOL and CPHA are not controllable. The WRITE command implementation simply enables the SDI clock for 8 cycles, while relaying the MISO and MOSI to the data input and output ports.

The I2CMST clock scaling is done by the bit command controller. The configurable core contains clock scaling logic inside both the SPI controller and UART controller in addition to the

3.2 GenIE (fully programmable core)

The fully programmable core GenIE was created to answer research question 2. It continues on from the configurable core, replacing the hardware interface controller cores with a programmable core. The goal of keeping the system data interface fully agnostic of the current software running on the component was kept, this goal being even more important with the programmable core being able to function with any protocol which can be supported within its limitations. (The limitations of the programmable core is further explored in 5.2.1)

An overview of the GenIE can be seen in figure 3.2

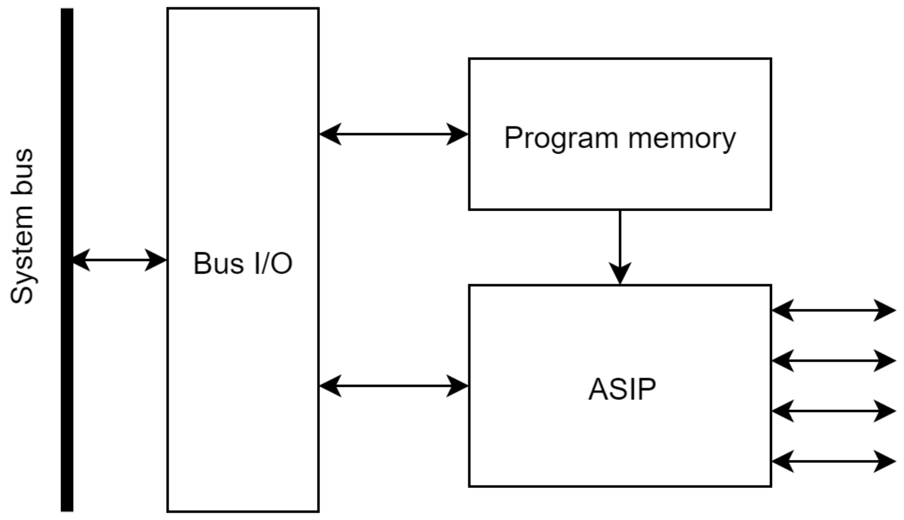

Figure 3.2: An overview of the GenIE

3.2.1 Design

As the major part of the I2CMST is concerned with actually implementing the I^{2}C protocol, and by extension a major part of the configurable core is concerned with implementing I^{2}C as well as UART and SPI, not a lot of the specific design ideas of these cores were kept for the programmable core. However, in designing the configurable core, a common interface for all three protocols became apparent. This idea of a common interface for several protocols could be reused in designing GenIE.

The governing idea of GenIE is that of a dedicated bit-banging co-processor. Instead of using the CPU for bit-banging a protocol, this dedicated component can be loaded with a bit-banging program instead, and then run independently of the CPU.

The center of the programmable core, that is the actual programmable part, is designed as an application specific instruction set prosessor, or ASIP. This was designed from the ground up, along with the instruction set, specifically for writing software with the single task of operating a set of configurable I/O pins according to a well-defined hardware protocol. It gets the instructions from a dedicated piece of memory, the specifics of which are outside the scope of this thesis. Also out of scope is the method of writing a program to this memory, although the author imagines this could be done over the same system bus used for data transfer, for ease of reprogramming the component.

The overall design of the programmable core is inspired by the design of the configurable core, with a bus I/O controller component sitting between the ASIP and the system bus. This controller component is responsible for system bus communications, and communicates with the ASIP using status lines as well as a special *command-register*. The ASIP takes on the role of the multiple protocol implementations used in the configurable core, and is responsible for controlling the I/O pin array through using configuration and data registers.

The ASIP design is a very simple processor design, using a simple Fetch-Decode-Execute loop with no pipeline. All operations operate on registers only; there is no data memory. An overview is given in figure 3.3. All I/O pins have dedicated registers. Data transfer to and from the controller component is done via two dedicated registers, one for data input and one for data output. In addition, several general purpose registers are available for computations required by the protocol program.

Among the most important instructions available on the ASIP is the MOV instruction, responsible for moving values from register to register. As the data input and output from the system bus as well as the I/O pins are controlled via registers, MOV instructions moving bits to and from the I/O pins are the main part of all protocol implementations in this thesis. As hardware communications interfaces have as a goal to transfer data between devices or components, MOV instructions will be important for any imaginable other protocol as well.

In addition to the MOV instruction, the NOP instruction is important. As

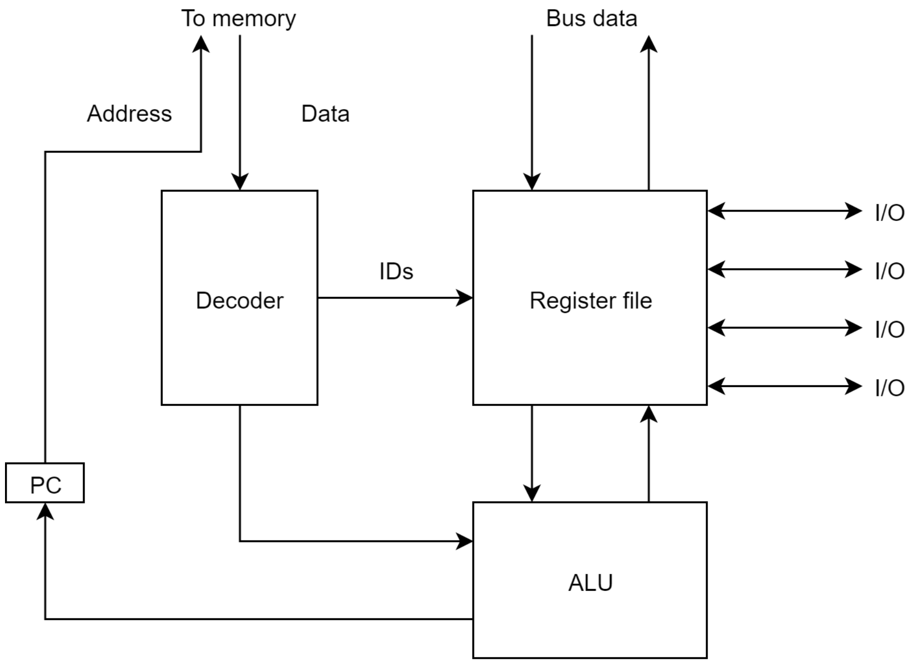

Figure 3.3: The Application Specific Instruction Set processor powering the GenIE.

the ASIP has no hardware synchronization mechanisms, all synchronization according to the protocol specification as well as other synchronization issues must be handled in software using these instructions.

The software for the ASIP is written as implementations of several predefined subroutines. These then constitute the common interface for all protocol implementations. Because of this, the design of this interface has a very large impact on which protocols are possible to implement using the component, and it is designed very generically.

The programmable core requires some static configuration data in addition to the program code for the subroutine implementations. This is provided at predefined locations in the same memory storing the program code. The configuration data required by the programmable core is:

- **Clock configuration:** The programmable core, like both I2CMST and the configurable core, contains a clock divider, which is configured using configuration data
- **Subroutine handlers:** As the software is a collection of subroutines, for each of these routines, a specific memory address is executed. This typically contains a JMP to the implementation, or a RETURN instruction if the routine is not implemented. This enables flexible layout of programs in memory, enabling program writers to divide the memory after protocol needs and complexity.

The transfer of data is done by bit by bit transfer to or from shift registers in the controller component which are read and written over the system bus. This imposes some limitations on the flexibility of the programmable core, but was done to simplify the design of the core. In section 5.2.1, these limitations along with alternative methods of data transfer are discussed.

Exit from a subroutine is done using the special RETURN instruction. This makes the ASIP idle, and signals back to the controller component that the requested command has been performed. An error condition can be indicated by setting the error flag on the RETURN instruction. This error condition is the signaled back to the controller component as well.

Instruction	**Description**
NOP	Does nothing
XOR r1, r2 $\rightarrow$ rD	Bitwise XOR of r1 and r2 stored in rD
NAND r1, r2 $\rightarrow$ rD	Bitwise XOR of r1 and r2 stored in rD
MOV r1 $\rightarrow$ rD	Moves the contents in r1 into rD
MOV r1, $\#n$ $\rightarrow$ rD	Moves the $\#n$th bit of r1 into rD
LOAD $\#number$ $\rightarrow$ rD	Loads a literal value $\#number$ into rD
NOT r1 $\rightarrow$ rD	Stores the bitwise NOT of r1 in rD
JMP $\#addr$	Unconditional jump to memory address $\#addr$
JNZ r1, $\#addr$	Conditional jump to $\#addr$ if r1 is not zero
JZ r1, $\#addr$	Conditional jump to $\#addr$ if r1 is zero
ADD r1, r2 $\rightarrow$ rD	Stores the unsigned sum of r1 and r2 in rD
SUB r1, r2 $\rightarrow$ rD	Stores the unsigned difference of r1 and r2 in rD
RETURN	Makes the ASIP idle and listen for more commands

Table 3.1: The ISA for the GenIE ASIP

The ISA is designed to contain every instruction needed to implement each of the I^{2}C, SPI and UART interfaces.

3.2.2 Implementation

As the system bus I/O is a detail beyond the scope of this thesis, only the ASIP and its components were implemented. The interface exposed by the ASIP to the system bus I/O controller consists of the following:

- Two sets of parallel data buses. These are used for the actual data being read and written by the ASIP program
- A command bus. This is inspired by the command bus from the I2CMST and the configurable core. The bus I/O controller supplies commands here to control which subroutine is to be executed by the ASIP.
- A line indicating the the last executed command subroutine has returned. This works similarly to the ack bit in I2CMST/the configurable core.

The supported commands are shown in table 3.2. The ASIP implementation itself does not assign any specific significance to any command, it is the implemented program for the ASIP which is responsible for handling the commnads according to the expected function. The auxiliary commands are provided in order for programs to expose dynamic control of part of their functionality. An example of where this can be useful is specifying an I2C address.

Command	Description
INIT	Initialization logic. Triggered on reset
SEND_BYTE	Sends the byte provided on the data input bus
READ_BYTE	Reads a byte onto the data output bus
AUX0	
AUX1	Auxiliary subroutines, typically interface specific
AUX2	
NOP	Makes the ASIP idle and wait for another command

Table 3.2: ASIP command interface specification

The ASIP decoder is quite simple, separating register IDs, opcode and any immediate value fields from the memory data value. The ALU is also quite simple. In the execute stage, it computes a value based on the decoded opcode and the register values r1 and r2, read from the decoded register IDs. It also provides a write-enable value for the rD register, as well as a return flag (set when the opcode is RETURN) and a JMP flag. It also provides a jump address bus used by the top level ASIP module if the JMP flag is set to 1.

Register	Description
P1	Pin 1 register
P2	Pin 2 register
P3	Pin 3 register
P4	Pin 4 register
DI	Bus data input register
DO	Bus data output register
R6 to R15	General purpose registers

Table 3.3: The available register on the GenIE ASIP

The register file contains logic for reading and writing values to its 8-bit registers. Table 3.3 lists the available registers. The register file has two read ports as well as one write port, the register IDs provided on register ID buses connected to the decoder outputs. In addition to handling the general purpose registers, it also provides functionality for the register mapped input and output. The DI register is a read only register mapped to the system bus data input bus. The DO register is mapped to the system bus data output bus. The registers P1 to P4 are used for reading and writing to the I/O pins, with the pin having the value of the least significant bit written to its register.

The memory address size of the ASIP is 8 bits, which limits the accessible program memory to 256 locations. Some of these are used for

3.2.3 GenIE programs

Writing programs for the GenIE consists of defining a clock divider value, and then implementing handlers for any supported command for the protocol which is to be implemented. The clock divider value controls the ASIP clock rate. Each fetch-decode-execute cycle takes three clock cycles. Exactly one instruction is executed for each ASIP cycle. This means that the divider value can be computed as System Clock Rate$/(3 \times$ ASIP Frequency$) - 1$.

A number of ASIP cycles may be spent transmitting each bit. The bitrate is then the ASIP clock rate times the number of ASIP instructions per bit. Maintaining a constant bitrate thus requires a constant number of program instructions for each bit transmitted.

Following is an example GenIE program for an UART

```
Clock divider         0 0x4242
INIT command          1 LOAD #1 -> P1
SEND command          2 JMP #007
READ command          3 RETURN
AUX0 command          4 RETURN
AUX1 command          5 RETURN
AUX2 command          6 RETURN
SEND handler          7 LOAD #0 -> P1
                      8 LOAD #0 -> R1
                      9 MOV DI #0 -> R2
                     10 XOR R2 R1 -> R1
                     11 MOV R2 -> P1
                     12 MOV DI #1 -> R2
                     13 XOR R2 R1 -> R1
                     14 MOV R2 -> P1
                     15 MOV DI #2 -> R2
                     16 XOR R2 R1 -> R1
                     17 MOV R2 -> P1
                     18 MOV DI #3 -> R2
                     19 XOR R2 R1 -> R1
                     20 MOV R2 -> P1
                     21 MOV DI #4 -> R2
                     22 XOR R2 R1 -> R1
                     23 MOV R2 -> P1
                     24 MOV DI #5 -> R2
                     25 XOR R2 R1 -> R1
                     26 MOV R2 -> P1
                     27 MOV DI #6 -> R2
                     28 XOR R2 R1 -> R1
                     29 MOV R2 -> P1
                     30 MOV DI #7 -> R2
                     31 XOR R2 R1 -> R1
                     32 MOV R2 -> P1
                     33 NOP
                     34 NOP
                     35 MOV R1 -> P1
                     36 NOP
                     37 NOP
                     38 LOAD #0 -> P1
                     39 NOP
                     40 NOP
                     41 NOP
                     42 NOP
                     43 NOP
                     44 NOP
                     45 LOAD #1 -> P1
                     46 NOP
                     47 NOP
                     48 RETURN
                     49
                     50 ...
```

Even though the UART is quite simple, this needs three cycles per bit transferred to compute the parity, and the bitrate is then equal to the scaled clock rate divided by three. Lines 32 to 48 show examples of synchronization using NOP instruction in order to maintain three instruction between pin writes. As there is no parallelism in the ASIP, multiple pins can not be written in the same cycle, so to update four output pin values, four instructions are needed.

Chapter 4

Results

These are the results obtained by simulating and synthesis and implementation of the two cores in Xilinx Vivado. Both simulation output and synthesis and implementation data is provided. In addition, similar data for comparable components is provided in order to be used when evaluating the configurable component and the GenIE.

4.1 Configurable component

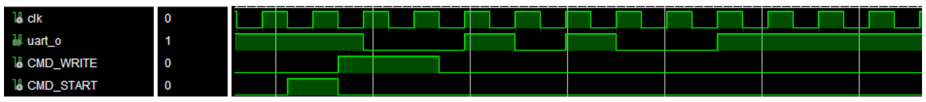

Figure 4.1: Simulation of the UART controller of the configurable component

Simulation of the UART component can be seen in figure 4.1

When synthesized and implemented for a xc7k70tfbv676-1 Kintex-7 FPGA, Vivado reports the following values:

Size	330 LUTs + 168 Flip-flops
Total power	0.087 watts

This was done using a clock constraint of 10ns clock rate, 50% duty cycle.

4.2 GenIE

When synthesized and implemented for a xc7k70tfbv676-1 Kintex-7 FPGA, Vivado reports the following values:

Size	138 LUTs + 159 Flip-flops
Total power	0.084 watts

This was done using a clock constraint of 10ns clock rate, 50% duty cycle.

4.3 OpenCores components

When synthesized in the same environment as the GenIE and the configurable component, Vivado reports the following values:

	Simple UART[10]	Simple SPI[11]	I2CMST[9]
Size	70 LUTs, 73 FF	85 LUTs, 67 FF	130 LUTs, 85 FF
Total power	0.083 watts	0.085 watts	0.085 watts

Chapter 5

Evaluation and discussion

Here the designs and results will be evaluated and discussed.

One thing to note is that the FPGA device has a static leakage power of 0.081 watts, which means the dynamic power increase from this is the power consumption caused by the implemented HDL. This is taken into account when comparing power consumption

5.1 Configurable component and research question one

The configurable component successfully implements both UART and SPI in addition to I^{2}C, with some limitations, see 5.1.1

The three OpenCores components for UART, SPI and I^{2}C use a total of 285 LUTs and 225 FFs. The configurable component successfully runs all three protocols, but uses more LUTs and a bit fewer FFs. However, both the UART and SPI components were synthesized with system bus logic, which means we would actually expect an increase in core size when using the three separate components.

With regards to improvement in power consumption, the total dynamic power usage of the three separate components is 0.01 W, while the configurable core has a dynamic power consumption of 0.006 W. This is an improvement, but as mentioned, some of the separate cores include system bus logic, which is not included in the reports for the configurable component.

Pin usage of the configurable component is 4 pins, to accommodate the four lines of the SPI protocol. The discrete components use 1, 2 and 4 signals for UART, I^{2}C and SPI, respectively. However, these signal may be multiplexed at the system level with a small amount of logic, so the reduced pin usage alone is not a large advantage for the configurable component.

5.1.1 Limitations

The implemented UART in the configurable core does not support parity bits. In addition, the SPI controller does not support configurable CPOL and CPHA. The discrete cores presented in chapter 4 has both of these features implemented. A bit larger size and power consumption would then be expected.

In addition, the SPI implementation in the configurable component does not handle the CPHA and CPOL entirely correct. CPOL controls whether to read on rising edge and write on falling edge or vice versa. However, as the communication between the Byte command controller and the SPI controller is done using a single shift register, both reading and writing must happen at the same time. This issue cannot be fixed without rewriting large parts of the Byte command controller, for which time was not prioritized during the work with this thesis.

5.2 GenIE and research question two

The programmable component, GenIE, reports less than half the LUTs used by the configurable core, and around the same number of flip-flops. It also reports a decreased power consumption. However, these numbers must be seen in conjunction with the fact that GenIE does not implement any hardware interface on its own. In order to implement an interface, it requires a program, which requires memory. As the memory is external to the GenIE, it is not included in the reported numbers from Vivado. This can then not be compared directly with the other numbers from the reports in chapter 4.

However, GenIE is quite flexible. The instruction set makes it possible to write programs resembling bit-banging, and opens up possibilites of extending the support to other interfaces beside the three used in this thesis.

5.2.1 Limitations

The implemented design has a maximum addressable memory of 256 addresses. This may be a limiting factor when implementing more complex interfaces. This could somewhat easily be mitigated by increasing the memory address size.

In addition, only one instruction may be performed at a single time. This means that if multiple pins need to change simultaneously, the interface is not possible to implement on the GenIE. This has a few possible solutions. First, one could have a separate pin-clock, with pin registers being written to the pins at constant intervals of multiple instructions. Thus the instructions could run sequentially while the pin values change simultaneously. Another potential solution is to replace the ASIP with one implementing a transport triggered architecture (see section 2.4). This way, multiple register could be written to with a single instruction.

Chapter 6

Conclusion

The fact that separate components, even when evaluated with more functionality than the configurable component, are better or as good as the configurable component, leads the author to conclude that the design presented is not an improvement over separate cores.

Whether or not the GenIE is an improvement in power or size can not be concluded based on the findings of this thesis. As the memory is an important part of the design, the act of not including it in the discussion means that no comparisons can be made. However, the increased flexibility of the GenIE is a definitive improvement in the case where support is required for an interface not originally available on the system. This makes it possible to avoid using CPU bit-banging in such cases, freeing up the CPU and making the CPU program code easier to maintan.

Bibliography

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011.

[2] F. Leens, "An introduction to i2c and spi protocols," *IEEE Instrumentation Measurement Magazine*, vol. 12, pp. 8–13, February 2009.

[3] NXP, *UM10204 I2C-bus specification and user manual.* NXP, rev6 ed., April 2014.

[4] P. J. Ashenden, *Digital Design (VHDL): An Embedded Systems Approach Using VHDL.* Morgan Kaufmann Publishers Inc, 2007.

[5] Texas Instruments, "Introduction to MSP430 communication interfaces." `http://www.ti.com/lit/ml/slap117/slap117.pdf`.

[6] J. Patrick, "Serial protocols compared." `https://www.embedded.com/design/connectivity/4023975/Serial-Protocols-Compared`.

[7] H. Corporaal, "Design of transport triggered architectures," in *Proceedings of 4th Great Lakes Symposium on VLSI*, pp. 130–135, March 1994.

[8] Gaisler, "GRLIB IP Core User's Manual." `https://www.gaisler.com/products/grlib/grip.pdf`.

[9] R. Herveille, "I 2 c-master core specification," *svn://opencores.org/ocsvn/i2c@ r76, OpenCores*, 2003.

[10] OpenCores, "Serial UART." `https://opencores.org/projects/uart`.

[11] OpenCores, "SPI core." `https://opencores.org/projects/simple_spi`.