NTNU
Norwegian University of
Science and Technology

Instruction Set for Bit-Banging Operations

Increasing flexibility for low power communication

Henrik Olav Solvang

Master of Science in Electronics
Submission date: January 2016
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Jan Frode Lønnum, Nordic Semiconductor ASA

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

PROBLEM DESCRIPTION

Title: Instruction Set for Bit-Banging Operations
Increasing flexibility for low power communication

Focus: Computer architecture, Processor design, Serial Communication.

Bit-banging techniques are being widely used by embedded programmers to emulate dedicated hardware in microcontrollers. Today a wide range of serial engines such as SPI, UART, UASRT, TWI, USB, I2S, MIPI etc. are required in a broad range ASIC targeted for high volume production. It is costly in terms of development time and chip area to make dedicated circuitry for these diverse communication protocols.

In order to increase the efficiency of communication protocols based on bit-banging methodology, this master thesis will propose a new core logic unit with an efficient instruction set for performing bit-banging operations. It is important that the architectural study on the instruction set is targeting efficient operations, high flexibility and the stringent timing requirements found in many communication peripherals. A more general use of the instruction set beyond serial engines should be considered. If time permits an example engine in C/SystemVerilog should be developed.

Responsible professor:	Kjetil Svarstad, IET
Supervisor:	Jan Frode Lønnum, Nordic Semiconductor

ABSTRACT

A microcontroller can only offer a limited amount of communication interfaces. When designing an ASIC targeted for high volume production, flexibility must often give way to increased energy efficiency. The limitation in the number and types of communication interfaces may force embedded designers to use inefficient bit-banging techniques to communicate with various modules. Can a co-processor optimized for bit-banging provide this flexibility with an acceptable loss in power efficiency, compared to dedicated hardware modules?

A cycle-accurate instruction set simulator has been developed to support the design of instruction sets and optimization of the bit-banging programs. It is also used to determine the run/sleep ratio for these programs on the individual instruction sets. Complexity based power estimations, that make use of power and complexity measures from an ARM Cortex M0 implementation, are used to estimate the dynamic power consumption of the various instruction sets.

A new set of instructions, called the SOL-instructions, was developed to optimize the output and input of serial data. Together with the addition of the *REPEAT*-instruction, a 36% reduction in active time was achieved compared to a simple instruction set. Compared to dedicated modules the difference in dynamic power consumption vary with transmission frequency. The power consumption range from 6.7%(UART, 9.6kbps) to 529%(SPI, 1000kbps) of the dedicated hardware's power consumption.

Flexibility is added at the cost of reduced power efficiency for high speed transmissions. The bit-banging processor is perhaps best suited as an addition to existing modules. It can not completely replace dedicated modules for common protocols, but show very promising results as an alternative to bit-banging in the host processor.

SAMMENDRAG (NORWEGIAN)

En mikrokontroller kan bare tilby et begrenset antall kommunikasjonsgrensesnitt. Når man designer en ASIC som skal produseres i høyt volum, må ofte fleksibilitet ofres for økt energieffektivitet. Begrensningen i antall kommunikasjonsgrensesnitt kan tvinge utviklere av innebygde systemer til å bruke inneffektive bit-banging-teknikker for å kommunisere med ulike moduler. Kan en coprosessor som er optimalisert for bit-banging tilby denne fleksibiliteten med et akseptabelt tap i energieffektivitet, sammenlignet med spesialiserte hardwaremoduler?

En syklus-nøyaktig instruksjonssett-simulator har blitt utviklet for å støtte designet av instruksjonssett og optimaliseringen av bit-banging-programmene. Den blir også brukt til å bestemme aktiv/inaktiv-forholdet for disse programmene for de individuelle instruksjonssettene.

Et nytt sett med instruksjoner, kalt SOL-instruksjoner, har blitt utviklet for å optimalisere utmating og innmating[1] av seriell data. Sammen med REPEAT-instruksjonen ble den aktive kjøretiden redusert med 36% i forhold til et simpelt instruksjonssett. Sammenlignet med spesialiserte hardware-moduler så varierer forskjellen i dynamisk effektforbruk med overføringsfrekvensen. Effektforbruket varierer fra 6,7%(UART, 9.6kbps) til 529%(SPI, 1000kbps) av de spesialiserte hardware-modulenes effektforbruk.

Fleksibiliteten er økt på bekostning av redusert energieffektivitet for høyhastighets-kommunikasjon. Bit-banging-prosessoren er kanskje best egnet som et tillegg til eksisterende moduler. Den kan ikke erstatte spesialiserte hardwaremoduler for de mest brukte protokollene, men er et veldig lovende alternativ til bit-banging i vertsprosessoren.

1 Output/Input

PREFACE

This thesis completes a Master of Science degree in Electronics, Design of Digital Systems. The assignment was given by Nordic Semiconductor in August 2015, and the thesis was delivered in January 2016.

While my studies have focused on the design of digital circuits, this thesis focuses on an architectural approach to a problem. With no accurate measurements of the chip power nor complexity available, and with no time to perform such measurements, a practical approach to power estimations was adopted. It was challenging to perform, and not nearly as satisfying as having more trustable results, but at an architectural level it is simply not practical to implement all solutions. It has been very educational to work at this level of abstraction, and I believe the work has given me valuable insight into the early stages of chip design. Insight that I will benefit from as an engineer.

I would like to thank my Professor Kjetil Svarstad for his advice during the design and writing processes. Jan Frode Lønnum, who has been my supervisor at Nordic Semiconductor, for advice and proofreading of the thesis. And finally I would like to thank Vemund Bakken for designing the problem, for very helpful advice in the starting phases of my work, and for input on the thesis.

Henrik Olav Sollesnes Solvang
NTNU, IET
Trondheim, January 24, 2016

CONTENTS

Abstract . . . iii
Sammendrag (Norwegian) . . . v
Preface . . . vii
1 INTRODUCTION 1
1.1 Motivation . . . 1
1.2 Contributions . . . 2
1.3 Limitations . . . 2
1.4 Method . . . 3
1.5 Structure of the thesis . . . 3
2 BACKGROUND AND PREVIOUS WORK 5
2.1 Bit-banging . . . 5
2.2 Protocols . . . 5
2.2.1 Serial Peripheral Interface – SPI . . . 5
2.2.2 Inter-Integrated Circuit – I2C . . . 8
2.2.3 Two Wire Interface – TWI . . . 11
2.2.4 Universal Asynchronous Receiver/Transmitter – UART . 11
2.3 Instruction set architecture . . . 13
2.4 Classifying instruction set architectures . . . 14
2.4.1 RISC - Reduced Instruction Set Computer . . . 14
2.4.2 CISC - Complex Instruction Set Computer . . . 14
2.4.3 Number of operands . . . 15
2.4.4 Operand Type . . . 16
2.5 Assembly . . . 17
2.6 Low power methods . . . 17
2.6.1 Dynamic power consumption . . . 17
2.6.2 Static power consumption . . . 19
2.6.3 Time to idle . . . 19
2.7 Pads . . . 19
2.8 Power estimation methods . . . 20
2.9 Previous work . . . 21
3 PROCESSOR ARCHITECTURE OUTLINE 23
4 INSTRUCTION SIMULATOR 27
4.1 Structure . . . 27
4.2 Register file . . . 28
4.3 I/O-map . . . 29
4.4 Waveform generation . . . 29
4.5 Report generator . . . 30
4.6 Testbench . . . 30
4.7 I-functions . . . 31

4.8 A-programs 33
4.9 Benchmark 36

5 INSTRUCTION SET ARCHITECTURES 37
5.1 Minimal Instruction Set 37
5.1.1 Protocols 38
5.1.2 Results 43
5.2 Three operands 43
5.2.1 Results 44
5.3 Repeat 44
5.3.1 Protocols 46
5.3.2 Results 47
5.4 SOL instructions 48
5.4.1 Results 50
5.5 SOL and Repeat 50
5.6 Conditional Execution 52
5.6.1 Protocols 54
5.6.2 Results 55
5.7 Parallel execution 56
5.7.1 Protocols 58
5.7.2 Results 61
5.8 Comparison 61
5.9 Max speeds 62

6 POWER 63
6.1 Power estimation method 63
6.2 The Cortex M0 as a baseline 64
6.3 Power estimate per functionality 65
6.3.1 Unused functions 65
6.3.2 Repeat 69
6.3.3 SOL 70
6.3.4 Conditionals 70
6.3.5 Parallel Execution 71
6.4 Dedicated modules 71
6.5 Estimation results 72
6.5.1 Equivalent power 73

7 COMBINED RESULTS 75
7.1 Inter-Integrated Circuit – I2C 75
7.2 Universal Asynchronous Receiver/Transmitter – UART 76
7.3 Serial Peripheral Interface – SPI 77

8 DISCUSSION 79
8.1 Power estimation 79
8.2 Slave implementations 80
8.3 The BBP as a general processor 81
8.4 Full coprocessor 81
8.5 Frequency reduction 82

8.6 Voltage scaling 82
9 CONCLUSION 83
9.1 Recommendations for Further Work 83
Bibliography 85

Appendix 89
A CELL LIST 91

LIST OF FIGURES

Figure 2.1 Simple SPI transaction 6
Figure 2.2 SPI configurations with sampling on the leading edge. . 7
Figure 2.3 SPI configurations with sampling on the trailing edge. . 7
Figure 2.4 Typical transfer of two bytes on I2C 9
Figure 2.5 Slave *clock-stretches* to catch up when master writes two bytes of data. 9
Figure 2.6 Glitch in start of UART TX. 12
Figure 2.7 Switching power in a CMOS inverter. 18
Figure 2.8 Nordic semiconductor input/output pad. 20
Figure 2.9 Power consumption of UART implemented on a CPU compared to a dedicated hardware module[1] 22
Figure 3.1 Bit-banging processor architecture illustration. 23
Figure 4.1 File structure and control flow of the instruction set simulator. 28
Figure 4.2 VCD-file 29
Figure 4.3 Flow graph of the algorithm shown in algorithm 4.12b. . 35
Figure 5.1 State chart for UART 42
Figure 5.2 Simulation results for minimal ISA. 43
Figure 5.3 Simulation results for three operand ISA. 44
Figure 5.4 Illustration of hardware implementation of the REPEAT-instruction. 45
Figure 5.5 Simulation results for REPEAT ISA. 48
Figure 5.6 SOL TX instructions(SOL/SOM) in hardware 50
Figure 5.7 SOL RX instructions(SIL/SIM) in hardware 51
Figure 5.8 Simulation results for SOL ISA. 51
Figure 5.9 Simulation results for SOL and REPEAT ISA. 51
Figure 5.10 Illustration of conditional execution implementation. . . 54
Figure 5.11 Simulation results for conditional execution ISA. 55
Figure 5.12 Illustration of the UART program in the parallel execution instruction set. Operations shown as horizontal arrows does not incur a pipeline flush. 57
Figure 5.13 Illustration of parallel execution implementation. 57
Figure 5.14 Simulation results for parallel execution ISA. 60
Figure 5.15 Running ratio for all instruction sets divided by protocol. 61
Figure 5.16 Running and total time of benchmark on all instruction sets, normalized to SolRepeat. 62
Figure 6.1 Simple register file model. 66

Figure 7.1 Power estimates for the I2C protocol divided by implementation. 75
Figure 7.2 Power estimates for the UART protocol divided by implementation. 76
Figure 7.3 Power estimates for the SPI protocol divided by implementation. 77

LIST OF TABLES

Table 2.1 Typical structure of SDA when transmitting two bytes. . 9
Table 2.2 UART frame 12
Table 2.3 Structure of an instruction with corresponding mnemonic assembly code. 14
Table 5.1 Minimal Instruction set 37
Table 5.2 Condition register and condition checks. 53
Table 5.3 Maximum bitrates for SolRepeat ISA. 62
Table 6.1 Cortex M0 area and dynamic power consumption. 64
Table 6.2 Area and power estimates of register reduction. 67
Table 6.3 Area and power estimates of addition/subtraction functionality. 67
Table 6.4 Area and power estimates of datapath width reduction. . 68
Table 6.5 Area and power estimates of multiplier. 68
Table 6.6 Area and power estimates of Repeat functionality. 69
Table 6.7 Area and power estimates of SOL functionality. 70
Table 6.8 Area and power estimates of Conditional functionality. . 71
Table 6.9 Area and power estimates of Parallel Execution. 71
Table 6.10 Power estimates of dedicated hardware modules. 72
Table 6.11 Area and current estimates of unused Cortex M0 functionality, and reduced Cortex M0 core. 72
Table 6.12 Area and current estimates of each instruction set architecture. 73

LIST OF ALGORITHMS

2.1 SPI master bit-banging 7
2.2 I2C master start bit and address transfer. 10
2.3 I2C master write data byte. 10
2.4 I2C bit-banging write single bit transaction. 11
2.5 UART bit-banging single bit. 12
2.6 UART bit-banging. 13
2.7 Assembly pseudo-language example. 17

4.8 I-functions used in the simulator. 32
4.9 jmp-function 32
4.10 cpseJmp-function (Compare Skip if Equal) 33
4.11 Program that outputs data serially, MSB first. 34
4.12 Program with if-else type structure. 35

5.13 Start-up A-program for the SPI-protocol 38
5.14 SPI byte loop. Full duplex communication. 40
5.15 I2C Write 40
5.16 I2C Write Acknowledge 41
5.17 UART top function 42
5.18 Receive algorithm 42
5.19 Repeat added to SPI 46
5.20 Nop padding to allow UART to use repeat. 47
5.21 Outputting single bit from a register without destroying data in register or changing other outputs. 49
5.22 Program with if-else type structure. 53
5.23 Parallel Execution Function (shortened for readability) 58
5.24 I2C parallel execution A-program 59
5.25 UART RX parallel execution A-program 60

ACRONYMS

BBP Bit Banging Processor

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver/Transmitter

I2C Inter-Integrated Circuit

I2S Inter-Integrated Sound

TWI Two Wire Interface

ASIC Application-Specific Integrated Circuit

RISC Reduced Instruction Set Computer

CISC Complex Instruction Set Computer

I/O Input/Output

ISA Instruction Set Architecture

PC Program Counter

RTL Register Transfer Level

CPU Central Processing Unit

LSB Least Significant Bit

MSB Most Significant Bit

AHB Advanced High-performance Bus

VCD Value Change Dump

ALU Arithmetic Logic Unit

ASPR Application Program Status Register

Instruction Acronyms: see table 5.1 and chapter 5.

1 INTRODUCTION

Reducing power consumption in ASIC designs has been an important topic for research and development for many years. The innovations in low power has created, and continues to create, new markets as well as constantly improving existing product lines. Optimizing for battery life and performance is as important as ever, but in the process of doing so the complexity constantly increases. The market for low-power microcontrollers is changing. The bulk of the market is large companies with time and resources to design complex systems. However, small start-ups achieving rapid innovation through the use of off-the-shelf modules is an increasingly important group of users. To capture these markets, ease of use is an important factor, and this includes the ease of which a microcontroller can communicate with existing modules.

1.1 MOTIVATION

Communication with sensors, external memory, motors and other external on- or off-chip modules is an intrinsic part of any microcontroller. There are several different communication protocols, where *Inter Integrated Circuit*(I2C), *Serial Peripheral Interface*(SPI) and *Universal Asynchronous Receiver/Transmitter*(UART) are among the most common. In serial communication, data is sent one bit at a time. If the microprocessor is used to transmit and receive these bits, known as bit-banging, it can have a severe impact on performance and power consumption. Traditionally, this is solved by adding peripheral modules that communicates on the bus with minimal interaction from the microprocessor. It is necessary to add one module for each open communication port, so the number of available ports is limited. In the Nordic Semiconductor chip *nRF52* there are in total one *Inter-Integrated Sound*(I2S), one UART, two *Two Wire Interface*(TWI, I2C-compatible) and three SPI modules. Some of these modules share resources so only five modules can be active simultaneously, where the two SPI-s and two TWI-s share resources. Although this selection of communication protocols may suffice in most cases, they still represent a limitation when the microcontroller is used in a larger system. Can an application specific processor designed to bit-bang protocols be a more flexible alternative, and what impact does it have on power? In this thesis the architecture of such a processor will be investigated. It is not expected that the power consumption will equal the consumption of dedicated hardware, but it is hoped that it can be well within an order of magnitude.

1.2 CONTRIBUTIONS

A cycle-accurate instruction set simulator has been created in C. The simulator emulates timing and creates waveform-files which can be viewed in a waveform viewer. It is simple to add and remove features of the simulated instruction sets, and seven instruction set variations has been implemented. The simulator can handle conditional parallel execution, where one of three fetched instructions is performed. Finally, the simulator creates report files that can be used to compare different instruction set implementations.

A concept architecture for a bit-banging processor has been created, and power consumption for this processor has been estimated. The ARM Cortex M0 is used as a baseline for power estimates. Based on analysis, the size of some Cortex M0 features has been calculated to allow for more accurate power estimations. The hardware-cost of implementing some instructions has been calculated for the same reason. The simulation and hardware estimation results have been combined to compare the different bit-banging processor implementations to each other and to dedicated hardware.

Using the simulator and power estimates, an instruction set architecture with promising results in terms of power efficiency has been created. A processor based on the instruction set architecture can be added to a microcontroller to offer a flexible communication module at an acceptable reduction in power efficiency, compared to dedicated hardware. Suggestions to further improve the power consumption of the bit-banging processor is presented.

1.3 LIMITATIONS

Time is the number one limiting factor when writing a thesis, and consequently it is important to limit the scope of the thesis to ensure that all tasks can be performed with the necessary depth. The main focus of this thesis is to develop an instruction set which is optimized for serial communication. It was decided that an RTL-implementation of the processor would not be feasible. Since there has been no RTL-implementation, the power estimates are based on an existing processor, the ARM Cortex M0. Although the Cortex M0 is a low-power processor, it is a larger and more powerful processor than the bit-banging processor. Through analysis and existing estimates, the size of Cortex M0 submodules that are not included in the bit-banging processor has been calculated and removed, but the accuracy of the power estimations are not expected to be high. Furthermore, complexity-based power estimation is not a very accurate method, and this further reduces the reliability of the power estimations. There has not been time to consider all aspects of the protocol implementations. Specifically, clock-stretching in I2C and clock synchronization in UART, are not implemented.

1.4 METHOD

The instruction set simulator has been designed using an iterative design methodology with ad-hoc testing. Using testbenches in the instruction set simulator to verify the results, the instruction sets have been developed using an analysis based methodology to explore the major candidates for improving the power efficiency. The power estimation is performed using an analysis based gate equivalent model widely used in industry for fast, architectural, power estimations.

1.5 STRUCTURE OF THE THESIS

First the communication protocols studied in this thesis will be presented in detail. An introduction to concepts of different processor architectures, and a short introduction to power estimation, is given in chapter 2. The conceptual bit-banging processor is discussed in chapter 3 to give the reader a basis for reading the subsequent chapters. To give the reader an opportunity to verify the instruction set simulator, it is explained in detail in chapter 4. The iterative development process for the instruction sets is highlighted in chapter 5. The instruction sets are explained, results are presented, analysed and the candidates for improvement is shown and implemented. In chapter 6 it is explained how the power estimation is performed, and the assumptions necessary to simplify the power estimation. The results for the different instruction sets are presented and discussed. The most promising solution is compared to the power estimates of the dedicated hardware modules in chapter 7. In chapter 8 the implications of the results, and the reliability of the power estimations is discussed. Finally conclusions and suggestions for further work is presented in chapter 9.

2 BACKGROUND AND PREVIOUS WORK

This chapter briefly describes relevant background theory for this thesis. Some assumptions and previous work is also presented. It is expected that the reader is familiar with basic processor construction techniques, and the assembly language. For further reading into this topic it is referred to *Computer Organization and Design* by Patterson and Hennessy[2].

2.1 BIT-BANGING

Bit-banging is the colloquial name for using a software program on a microprocessor to toggle general purpose input/output pins to adhere to some communication protocol. It is often used in embedded programming if the microcontroller lacks dedicated modules to communicate on a given protocol. Depending on the energy consumption of the processor, bit-banging is usually power inefficient because the general processor, as compared to dedicated hardware, is not optimized for the process. Timing in the protocols is also important and the bit-banging must be given high priority in the processor to ensure that timing is met. It may even be necessary to perform no other operations while the processor is transferring data, and assuming that the communication frequency is much lower than the processor frequency, the processor will spend a large amount of time simply waiting until the next bit can be transferred.

2.2 PROTOCOLS

In this section three communication protocols are presented: SPI, I2C and UART. These protocols have been used as reference protocols when designing the instruction sets in this thesis. They are presented in detail with pseudocode for the bit-banging procedures.

2.2.1 SERIAL PERIPHERAL INTERFACE – SPI

There is no specific standard for the SPI-protocol, which is evident in the considerable amount of protocol options. SPI is simple to implement, and can sustain high bitrates (>10mbit/s [3]). Every SPI slave needs a unique slave select signal. This means that the designer either has to use a *General-Purpose Input/Output*(GPIO)-pin for each slave or add a decoder to distribute the chip select signals. SPI does not allow for more than one master on the bus. Although 8-bit words are common in SPI modules, and word lengths that are multiples of 8 are even more so, SPI is not limited to these word lengths. SPI is commonly full-duplex, using four wires to allow simultaneous sending and receiving to and from slave, but can be reduced to a half-duplex three wire interface using a single wire for both sending and receiving. Extra status wires may be added to indicate data-ready or other flow control related tasks. Some SPI implementations send LSB first, others have active low chip selects. In the description below a full-duplex, four wire SPI is described.

The four-wire interface consist of:

◦ SCK	-	Serial Clock
◦ MOSI	-	Master Out Slave In
◦ MISO	-	Master In Slave Out
◦ CS[n..0]	-	Chip Select

The bus master drives the SCK, MOSI and CS signals while MISO is driven by the bus slave. As shown in figure 2.1 communication is initiated when the master asserts the slave's chip select signal. The first data-bit follows on the first (either rising or falling) edge of SCK and the predefined number of bits follow on each edge. When the transmission is completed the chip select is de-asserted. The pseudocode of the bit-banging procedure is shown in algorithm 2.1.

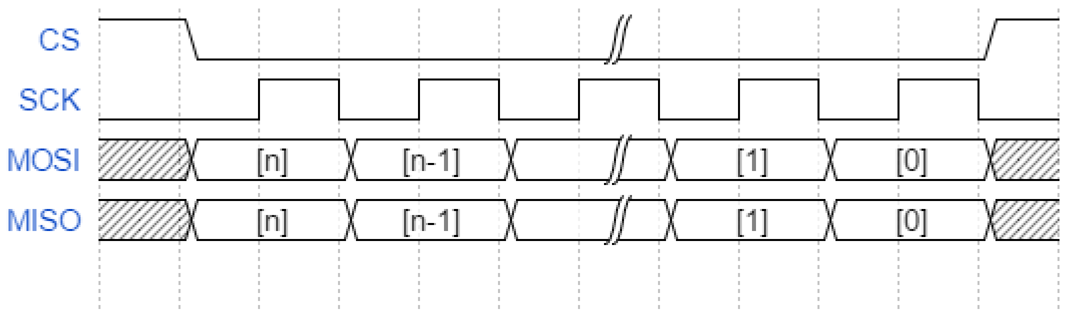

Figure 2.1: Simple SPI transaction

CONFIGURATIONS

SPI is not strictly defined and different slaves and masters may expect slightly differing protocols. In figure 2.1 the data (MOSI/MISO) is changed on the falling edge of SCK and is sampled on the leading edge. It is common for SPI to be configurable in terms of clock polarity, leading/trailing edge sampling and data order. Figure 2.2 shows the different configurations for MOSI, MISO, SCK and CS when the data is sampled on the leading edge, while figure 2.3 shows the same for trailing edge sampling. The sampling edge is in both cases indicated by an arrow on SCK. Changing the sampling edge is equal to a change in the phase relationship between SCK and the data. As can be seen in these

Algorithm 2.1 SPI master bit-banging

Precondition: Full-duplex SPI with sampling on the leading edge.

```
function spiTransfer(sendData)
    assert CS
    for word length do
        shift next sendData-bit on to MOSI
        wait half a SCK period
        assert SCK
        read MISO to shift register
        wait half a SCK period
        de-assert SCK
    end for
    de-assert CS
    return read data
end function
```

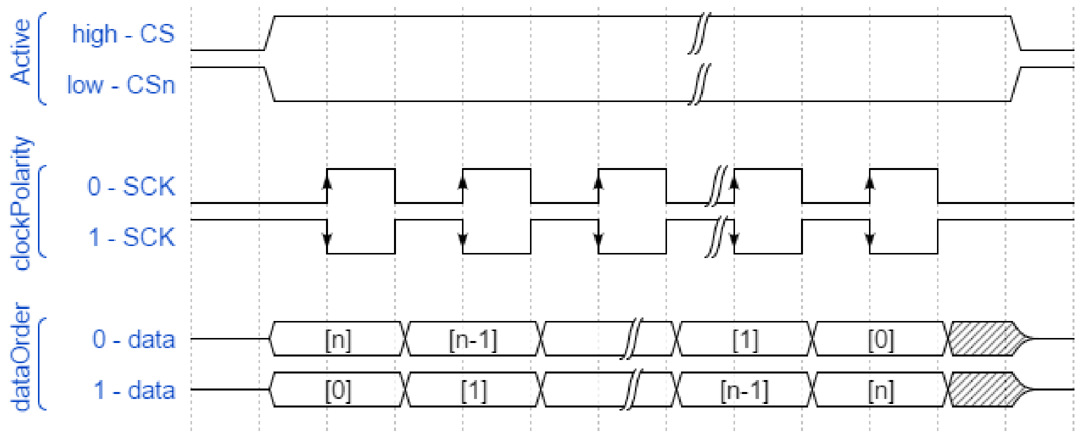

Figure 2.2: SPI configurations with sampling on the leading edge.

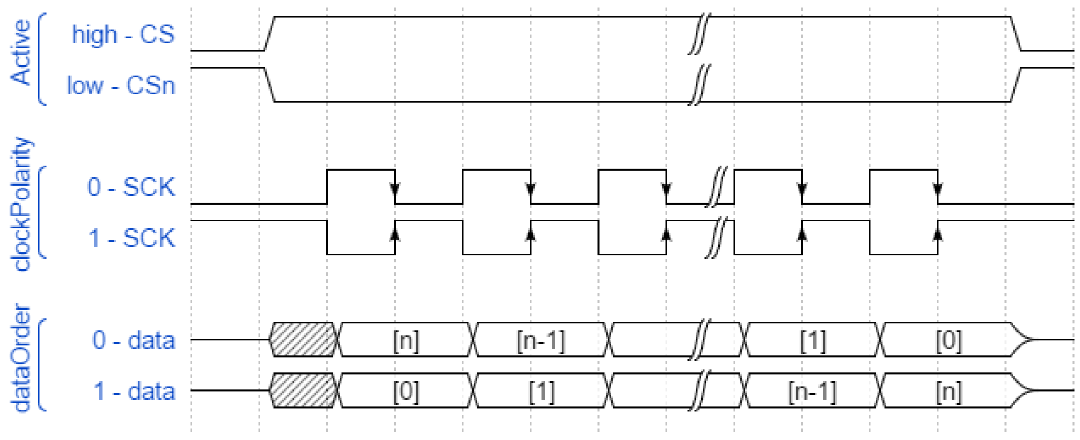

Figure 2.3: SPI configurations with sampling on the trailing edge.

figures, polarity and phase controls *when* the data is changed in relation to SCK. Notice that data changes when SCK is high both when polarity is inverted with normal phase and when the polarity is not inverted with sampling on the trailing edge. In addition to these configurations some systems may use separate phase and/or polarity for MOSI and MISO.

2.2.2 INTER-INTEGRATED CIRCUIT – I2C

The Inter-Integrated Circuit(I2C) is a two wire synchronous protocol. It is a multi-master/multi-slave protocol which, unlike SPI, is well defined and allows for all masters and slaves to share two wires. The increase in members on the bus comes at the cost of reduced speed. 100 and 400kbit/s are normal speeds, although 3.4Mbit/s is supported in some devices[4]. I2C is an active high system where the buses rely on pull up resistors to pull the bus to a logical one. In other words the slaves and masters on the bus will connect the bus to ground to indicate a logic 0 and output high impedance to indicate a logical 1. The two-wire interface consist of:

- SCL - Serial Clock
- SDA - Serial Data

As shown in table 2.1, communication is performed with two 9-bit *frames*: the address frame and the data frame. All communication is first initiated by a master by pulling down SDA from high to low while SCL is high, this can be seen in the first change of SDA in figure 2.4. This is different from all other I2C communications because all regular changes on SDA is defined to happen only when SCL is low. The start bit alerts all other members on the bus that a new transaction is coming. The initiating master will then issue an address, 7 or 10 bits long, which all slaves receive and evaluate. The master then indicates whether it wants to write or read with a logical 0 or logical 1 respectively. If the address matches a slave, and it is ready to be written or read, the slave will acknowledge by pulling SDA low. If there is no acknowledge, something is wrong and the master has to decide how to proceed. The data phase starts and either the master(write) or the slave(read) will toggle SDA when SCL is low and hold until the next falling edge on SCL. The receiver will acknowledge the byte as the 9-th bit in the data frame. These 9-bit data frames will continue to be sent until the master issues a stop bit by changing SDA from low to high while SCL is high.

Sometimes a master's data rate will exceed the data rate of the slave. That means that the slave will not be ready to receive or send data when the master expects it to. The slave can then *stretch* the clock by pulling down SCL until it is ready to transmit again, as shown in figure 2.5. While the slave keeps SCL low the master is not allowed to change SDA until SCL is released by the slave

Table 2.1: Typical structure of SDA when transmitting two bytes.

Start	Address	R/W	ACK	DATA	ACK	DATA	ACK	STOP
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	8 bits	1 bit	1 bit
	Address frame			Data frame		Data frame		

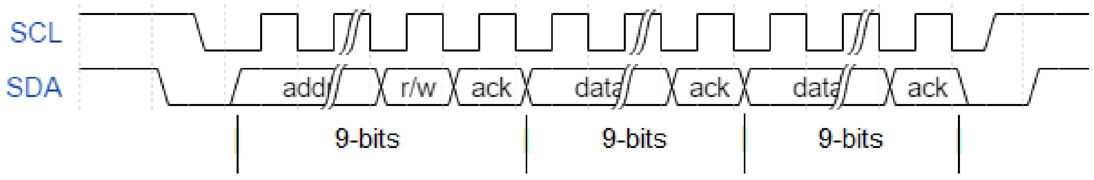

Figure 2.4: Typical transfer of two bytes on I2C

plus half a SCL period. Clock stretching is usually performed between the data byte and the acknowledge bit.

Arbitration between masters on the I2C-bus is performed by the masters themselves. Whenever a master outputs data on SDA it checks whether or not SDA was set correctly. If it was not, it assumes that some other master won arbitration on the bus. There may of course be some other, erroneous, reason for SDA not being set correctly but in that case the transaction has become worthless either way. Because SDA is either released (and pulled high by the pull-up resistor) or pulled low, the master with the most zeroes in its message will always win arbitration. In the worst case scenario two masters may try to read from the same slave(or write the exact same data) and will not detect the other master on the bus because they will transmit exactly the same data. If the slave is a sensor or similar this is not a problem because both masters was supposed to receive the same data, but for some application where the order or number of accesses is important, this may cause errors. Including such slaves on a multi-master bus must be done with utmost care. A pseudocode for the I2C master address phase can be seen in algorithm 2.2. Algorithms 2.3 and 2.4 show the write procedure. In addition to the three algorithms shown, a read procedure as well as a stop procedure is needed.

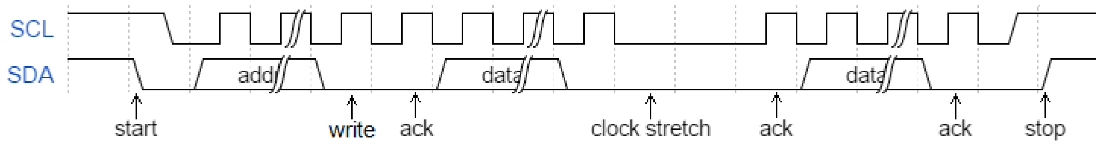

Figure 2.5: Slave *clock-stretches* to catch up when master writes two bytes of data.

Algorithm 2.2 I2C master start bit and address transfer.

Precondition: Function i2cStartAddr must be followed an i2cWriteByte or i2cReadByte function if not aborted. io[SDA] is in/out pin for the SDA-signal

```
function i2cStartAddr(address, rwBit)
    while !SCL do
        wait a short period                    ▷ Waiting for SCL to be released.
    end while
    pull down SDA                                               ▷ Start bit.
    wait half a period
    for bits in address do
        arbit = i2cSendBit(bit)
        if arbit then
            return 1                       ▷ Abort because arbitration was lost.
        end if
    end for
    i2cSendBit(rwBit)                                      ▷ 1: Read, 0:Write
    nack = io[SDA]
    return nack                          ▷ Returns 1 if no slave acknowledges.
end function
```

Algorithm 2.3 I2C master write data byte.

Precondition: Function i2cWriteByte must be followed by a stop bit or a i2cWriteByte function.

```
function i2cWriteByte(wByte)
    for bit in wByte do
        arbit = i2cSendBit(bit)
        if arbit then
            return 1                       ▷ Abort because arbitration was lost.
        end if
    end for
    nack = i2cReadBit()
    return nack                         ▷ Return acknowledge on data transfer
end function
```

Algorithm 2.4 I2C bit-banging write single bit transaction.

Precondition: io[SDA] is in/out pin for SDA-signal.

```
function i2cSendBit(bit)
    while !SCL do
        wait a short period          ▷ Waiting for slave to release clock stretching.
    end while
    wait half a period
    pull down SCL
    io[SDA] = bit
    wait half a period
    release SCL
    readSDA = io[SDA]
    if readSDA != bit then
        return 1                     ▷ Master lost arbitration.
    end if
    return 0
end function
```

2.2.3 TWO WIRE INTERFACE – TWI

The Two Wire Interface is essentially the same interface as I2C. The name was first taken in use by Atmel because, while the I2C protocol was not patentable, the name I2C was trademarked. Thus, TWI will be able to connect to an I2C interface[5].

2.2.4 UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER – UART

As the name states, UART is an asynchronous protocol. This means that there is no clock transmitted together with the data signal, and the protocol is depending the transmitter and receiver having synchronous internal clocks. UART is usually configured as a full duplex system with two wires, one for each direction of communication.

Unlike I2C, UART has no clear perception of a master/slave relationship. Both sides of a communication can initiate a transfer. The asynchronous nature of UART means that the speed at which transfers are to be performed has to be decided prior to transmission. Both communicators also has to agree upon the length of the transfer (usually 5, 8 or 9 bits), whether a parity bit is going to be used and how many stop bits that should be sent at the end of transmission. A UART module usually has two pins, receive data(RXD) and transmit data(TXD).

When the wires are not transmitting the wires are held in an idle state, indicated by logical high for most systems. A UART frame is shown in table 2.2. A

Table 2.2: UART frame

Bit	1	2	3	4	5	6	7	8	9	10	11
	Start	Data								Stop	

module starts a data transfer on TXD by pulling the wire to a logical low for at least half a clock cycle. All shorter fluctuations on the wire will be considered noise and ignored, this can be seen in figure 2.6. The receiver will synchronize its clock to the falling edge of this start bit if it is not ignored. After this, data is sent on every positive edge of the transmitters internal clock, with the least significant bit first. Some UART modules may synchronize its internal clock on every received data edge. A parity bit, for rudimentary error detection, may be sent after the data. Finally, one or more stop bit(s)(logic high bit(s)) is sent. Unlike I2C, the stop bit does not need to be a transition from logic low to high, but is merely sent to ensure that the line ends in idle mode. Pseudocode for UART is shown in algorithm 2.6 and 2.5.

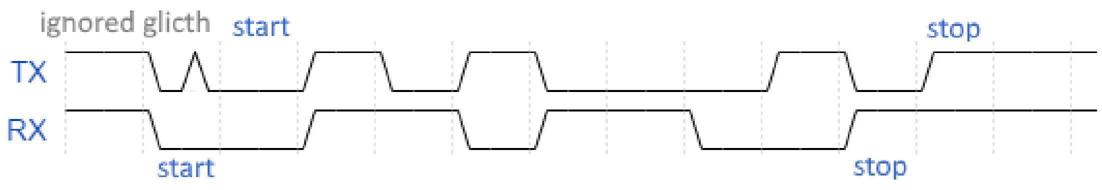

Figure 2.6: Glitch in start of UART TX.

Notice that unlike I2C and SPI the UART algorithm is not created for sending one byte at a time, but is instead controlled by *TX Active* and *RX Active*. This is because there is no master/slave relationship between the two communicators, thus the algorithm has to always check if the communication is initiated when it is idling. Counters, which is not shown in the pseudocode, should keep track of the number of bits sent and received and should clear the *active*-flags when a byte is completed.

Algorithm 2.5 UART bit-banging single bit.

```
function UARTBITTRANS(bit)
    wait posedge baud rate generator
    io[TX] = bit
    wait negedge baud rate generator
    rx = io[RX]
end function
```

Algorithm 2.6 UART bit-banging.

```
function UART
    if TX Active then
        uartBitTrans(LSB of byte)
        shift byte
    else
        if TX Start then
            uartBitTrans(0)                                   ▷ start bit
        else
            uartBitTrans(1)
        end if
    end if
    if RX Active then
        add read bit to byte
    else
        check for start condition
    end if
end function
```

2.3 INSTRUCTION SET ARCHITECTURE

The instruction set architecture(ISA) is defined by Patterson and Hennessy as a key interface between hardware and low-level software[2]. As a layer of abstraction it allows software to run on vastly different hardware implementations. As an example, both *AMD Opteron* and *Intel Core i7 (Nehalem)* processors implement the x86 instruction set, but their implementations differ greatly in both pipeline and cache[6].

The instruction set of a processor is set of simple operations that the processor can perform. It usually includes loading, storing and moving data; doing logical and arithmetical operations(*AND*, *OR*, *ADD*, etc.); control-related operations like jumping and comparing data; and other basic operations that can be combined to perform more complex operations. On the low-level software side the instruction set is represented as a set of mnemonics, as shown in table 2.3. A software developer can use these mnemonics to write an assembly code describing the series of operations he/she wishes to perform. The assembly code is then *assembled* into a set of machine readable bit-strings. On the hardware side, the instruction set defines which operations the processor must be able to perform, and how the processor should interpret the bit-strings. It does not specify how the operations should be performed, and this allows the implementation of hardware to be independent of the software.

Table 2.3: Structure of an instruction with corresponding mnemonic assembly code.

0	0	1	0	0	0	1	0	0	0	0	1	1	0	0	0
Operation Code					Register			Immediate							
Mnemonic:					ori			R2		24					

2.4 CLASSIFYING INSTRUCTION SET ARCHITECTURES

There are several ways to classify different instruction sets. This section will explain some of the different groupings of instruction sets and describe their respective strengths and weaknesses.

2.4.1 RISC - REDUCED INSTRUCTION SET COMPUTER

The *Reduced Instruction Set Computer*(RISC) is constructed around the idea that fast execution of a series of simple instructions, is preferable to slower execution of more complex instructions. It seeks to simplify the decoding stage in the computer by using fixed length instruction words, with fixed positions for various elements like the operation code, registers and immediate. RISC architectures always employ a structure where all arithmetic operations are performed between registers(register to register operations), and separate LOAD/STORE instructions are used to access memory. RISC architectures usually implement a homogeneous register set(in contrast to sets of specialized registers) both to simplify hardware and to simplify the compiler design.

A study shows that 96% of instructions in a general use-case are simple instructions [6]. So while the implementation of more complex instructions may increase the performance of very specific operations, a RISC architecture seeks to perform simple operations so effectively that the performance gap can be closed in the sum of operations. The most well-known RISC architecture today is perhaps the *Advanced RISC Machine*(ARM) architecture. The ARM architecture dominates the portable market today, ranging from 32-bit microcontrollers to the high end tablet and mobile CPU-s.

2.4.2 CISC - COMPLEX INSTRUCTION SET COMPUTER

The term *Complex Instruction Set Computer(CISC)* was coined to differentiate architectures from the RISC type instruction set architectures. A CISC architecture will typically use varying length instruction words, and logic/arithmetic instructions that can load data from memory. Thus, a CISC can perform logic/arithmetic operations without temporarily storing the data in internal registers. CISC architectures can also include complex instructions that seeks to improve performance for high-level functionality such as array accesses, loop control and procedure call operations. The x86 instruction set architecture is the most prevalent version of CISC instruction sets. Due to its popularity and age, the software support for x86 is good and widespread. Most personal computers and servers today use the x86 CISC architecture[7]. However, with iOS, Android and Windows 10 running on ARM instruction sets, the gap is closing.

A study, by E. Blem et al., comparing RISC and CISC type instruction set architectures for high performance CPU-s, conclude that the type of instruction set architecture has no impact on performance or energy consumption for the tested high performance architectures. The hardware implementation, the microarchitecture, is the deciding factor. For low performance and power CPU-s, they note that the overhead of CISC instruction sets makes them "*untenable*" for the sub 50 milliwatt CPU-s[7]. For the applications in this thesis, memory to memory operations are not generally needed. Furthermore, a simple and power effective hardware-design will be important, and CISC architectures are not chosen as a basis for the design in this thesis.

2.4.3 NUMBER OF OPERANDS

Instruction set architectures can use different numbers of operands in their arithmetic and logic operations.

Zero operand architectures are also called stack architectures. With no way of addressing registers, all operations are stored on top of the same stack. PUSH/PULL operations are used to load and remove data from the stack while logic and arithmetic operations are usually performed between the two top elements on the stack.

Logic/arithmetic operations in one operand architectures, or accumulator architectures, have an implicit temporary register storage location, the accumulator. The logic/arithmetic operations are performed between the operand location, usually a memory location, and the value in the accumulator. The result is then stored in the accumulator. The widely used MSC-51(8051) instruction set is a one operand architecture.

Two operands is the minimum amount of operands needed to be able to use a general register file. Two operands allows the program to perform a logic/arithmetic operation on two registers or memory locations and store the result to one of the locations. Because the instruction words needs to be kept short, two operands are very common in 16- and 8-bit RISC architectures. Two is also the prevalent number of operands in CISC architectures.

Three operands is the most commonly used number of operands in modern high performance RISC architectures. With three operands it is possible to perform an operation between two registers and store it in a third register. It increases the length of the instruction words compared to a two operand architecture, but it allows for more flexibility in the re-use of data.

Four operands allow for complex instructions such as fused multiply-add (FMA), where two registers are multiplied, a third register is added to the result and the final result is stored in a fourth register. Four operands is highly specific and is today just used in a few instructions in the AMD x86 architecture-family—Bulldozer. It is reportedly dropped in AMD's future architectures[8].

While zero and one operand architectures can be effective in some applications, they lack effectiveness in the general case due to the strict ordering of operations that the limited storage implies. A two or three operand architecture is the most likely choice for the applications in this thesis.

2.4.4 OPERAND TYPE

How many memory locations a logic/arithmetic operation can access also differentiates one instruction set from another. The number of memory operands can range from zero to four, but almost all computers can be put into one of three groups.

In register-register(load-store) architectures only registers can be accessed in its logic/arithmetic operations. It uses a separate load instruction to access memory and then temporarily stores the data in registers, performs operations on these registers and finally saves the result to memory using a store instruction. This allows for more compact, fixed length, instructions. It also ensures that all instructions take a predictable number of clocks to execute. This structure is one of the defining factors for RISC architectures.

Register-memory architectures allows for one of the operands to be a memory location. These architectures uses two operands where either the destination or source *can* be a memory location, but not both. The x86 architecture uses a register-memory architecture, and most CISC-type architectures employ this type of operand addressing. Register-memory architectures allow for more compact assembly code because data can be accessed without separate load/store-instructions, but the number of registers that can be addressed can be limited by the need for memory locations to be encoded in instructions. The number of clocks it takes to execute an instruction can also vary depending on whether only registers are used or if memory is accessed.

Memory-memory architectures allows all operands to be memory locations. It doesn't need to use registers for temporary storing, and can therefore produce the most compact assembly code. The instruction words can vary a lot in size depending on which type of operands is used, and there can be large variations in the execution time for each instruction. Furthermore, memory will quickly become a bottleneck due to its significantly lower speed compared to registers. Memory-memory architectures are generally not used in modern computers.

As mentioned earlier, for the applications considered in this thesis memory accesses are not needed because the intention is to let the CPU run in the same way as any dedicated hardware peripheral. A register-register architecture is the natural choice for the CPU in this thesis.

2.5 ASSEMBLY

Assembly languages come in several different forms and dialects. They can have architecture specific operand names and the operands can be ordered differently. Throughout this thesis the assembly language used is a pseudo-language created for readability only, an example can be seen in algorithm 2.7. In arithmetic/logic mnemonics the first operand is the target, or storage, operand. Registers are denoted as RX, with X being a number identifying the register. Immediates are simply designated as hexadecimals and some instructions also use Booleans.

Algorithm 2.7 Assembly pseudo-language example.

```
start:
    andi R0 0xF0
```

2.6 LOW POWER METHODS

The main goal of the instruction set architectures described in this thesis is low power. Some well-known low power concepts and methods will be presented as they will be used in the planned architectures.

2.6.1 DYNAMIC POWER CONSUMPTION

Simply put, the dynamic power consumption is the power consumed in gates when a value is changing state, either $0 \rightarrow 1$ or $1 \rightarrow 0$. When a value changes state, the load capacitance of the gate, mostly parasitical or wire capacitances, must be charged/discharged. The bulk of the power is consumed when the output is switched, and the output load capacitance must be charged or discharged as shown in figure 2.7[9]. This is called the switching power and the average switching power can be calculated as

$$P_{switch} = \alpha C_L \cdot V_{dd}^2 \cdot f_{clock} \qquad (1)$$

Where P_{swicth} is the dynamic power consumption, C_L is the load capacitance, V_{dd} is the supply voltage, f_{clock} is the clock frequency and α is the activity factor which is usually less than 0.2[10]. Both charging internal capacitances and the crowbar current[1] are other factors that affect the dynamic power consumption, but switching power is the dominating factor and dynamic power consumption is often simplified to equation 1[9].

1 The current resulting from a short circuit between the rails. This short circuit occurs when both PMOS and NMOS are active in the transition to a new, complementary, state.

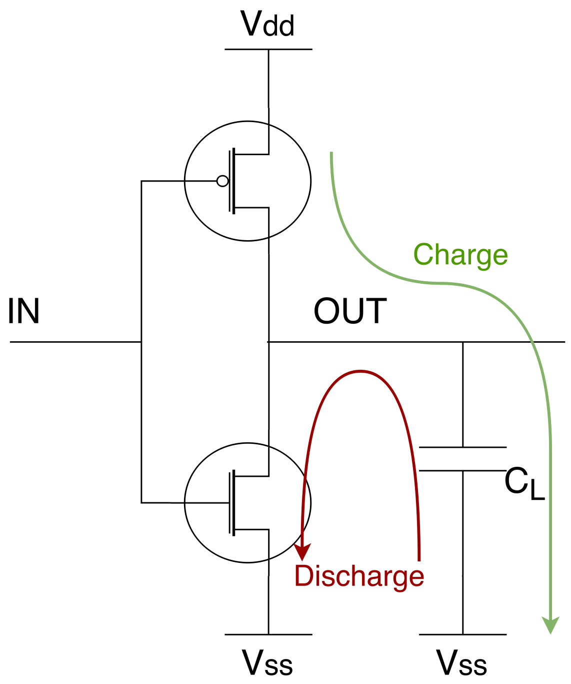

Figure 2.7: Switching power in a CMOS inverter.

VOLTAGE AND FREQUENCY

The dynamic power is quadratically proportional to the voltage in a given system. Thus, the most effective way to reduce dynamic power would be to reduce the voltage. However, the propagation delay through a gate is proportional to the voltage and reducing the voltage will lead to lower performance for the circuit. While a lower frequency will lead to a reduction in the instantaneous dynamic power, the total energy will not be reduced, considering that the application will run for a longer time. The goal becomes to find the lowest possible voltage at which the circuit can function while still meeting performance demands. In this thesis the voltage and frequency will be assumed to be set by other factors and unchangeable.

CLOCK GATING

Dynamic power consumption is dependent on the activity in the circuit, and in a sequential circuit the clock is the signal enabling all changes. While some modules are required to run continuously, some parts of a design may be waiting for some event to occur while doing nothing. Stalling the clock for these modules is called clock gating[9]. The concept is as simple as it is effective: if a module is not doing anything functional, remove the clock and no activity will be performed. Clock gating can be achieved by adding an AND-gate in the clock tree, using the clock as one input and adding an enable signal as the other. When the clock is gated, data will stop flowing through the sequential circuit, but all states in flip-flops will be retained. When the clock is once again enabled the circuit may pick up its activity just where it was stopped. Even though the clock is stopped, other inputs to a submodule may continue to toggle(e.g. bus connections). So in addition to gating clocks, one can gate the submodule's connection to the buses to further reduce the dynamic power consumption when the module is idle.

2.6.2 STATIC POWER CONSUMPTION

Static power consumption can be generalized to mean the power consumed by leakage in the transistors. It includes sub-threshold leakage, gate leakage, gate induced drain leakage and reverse bias junction leakage. Many methods for reducing static power consumption are performed at transistor level and is not within the scope of this thesis, but one method for reducing static power consumption will be described.

POWER GATING

Power gating is the main method for reducing static power consumption at the architectural level[9]. Similar to clock gating, power gating is the act of removing power from unused modules in the design. Power can be gated by distributing VDD to the whole circuit while implementing a power-gating network for VSS—where VSS can be pulled to a virtual rail close to VDD—or vice versa. Power gating is a complex method and includes isolation of inputs and outputs, possibilities for data retention and the implementation of a switching fabric. Generally the wake-up time from a power gated state is much larger than the wake-up time from a clock gated state, and there is also a cost related to the discharging and charging of capacitances in the circuit when the power is turned off and on[11]. This limits the usage of power gating to cases where long periods of sleep time is expected. Properly designed, power gating gives the developer the opportunity to dynamically add and remove modules from a system.

2.6.3 TIME TO IDLE

Time to idle will throughout this thesis be used as a measure of the performance of instruction sets. This is based on the assumption that the processor is put in sleep mode whenever it does not have instructions to perform. It is also assumed that the frequency and voltage of the processor is set and unchangeable. Thus the dynamic energy consumption of the processor is solely decided by the number of gates and time to idle.

2.7 PADS

The inputs and outputs used at Nordic Semiconductor uses bi-directional pads to connect to the outside world. As shown in figure 2.8 pads have four inputs;

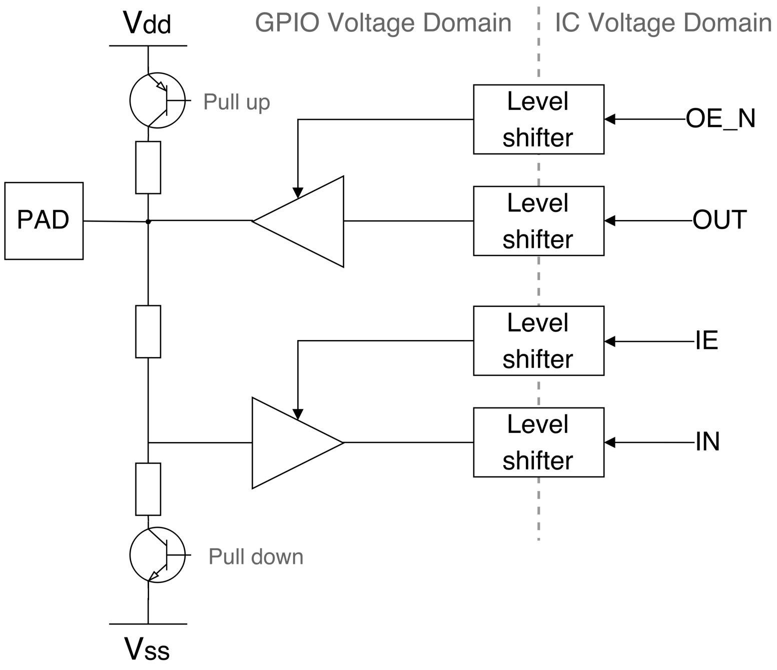

Figure 2.8: Nordic semiconductor input/output pad.

input enable, output enable, input and output. Input enable(IE) allows the integrated circuitry to read the data on the pad-bus. When input enable is not asserted the input(IN) signal is tied low. Output enable(OE_N) controls a tri-state buffer and if output enable is de-asserted the output from the pad is high impedance. If output enable is asserted, the pad-bus is connected to the output(OUT) signal through the buffer.

2.8 POWER ESTIMATION METHODS

Accurate power estimation can be performed on a gate-level design using statistical, stochastic or simulated input to model the expected activity on the chip. These techniques are to a large degree incorporated into commercial tools. Gate-level estimation requires synthesis of all modules in a design, and is thus impractical for use in architectural power estimations.

High-level estimation techniques are designed to perform faster power estimations with less input information, at the cost of lower accuracy. Some methods require the designer to know the approximate power of every macro-module in the design[12]. The macro-models can be produced from: known, of-the-shelf, modules; using power factor approximation [13] or by creating activity based models[14]. All of these techniques require the designer to have a gate level implementation of key modules in the design, or at least to have power figures from the same.

The simplest, and perhaps least accurate, model is a complexity based analytical model. In one method presented, one tries to relate the power consumption to the number of gates(e.g. NAND2 equivalents) in the design[15]. It directly uses equation 1 and assumes that the typical energy consumption for a gate equivalent, and the modules average load capacitance and activity, can be used to calculate the power consumption for the module as:

$$P_{mod} = N(P_{typ} + C_L * V^2) * \alpha * f \qquad (2)$$

Where N is the number of gate equivalents, P_{typ} is the typical power consumption of a gate equivalent, C_L is the average load capacitance, α is the average activity factor and f is the running frequency. Landman[13] and Raghunathan[12] argues that the method is poor at estimating the power for specialized structures like memory, I/O and clock networks. Leading to the conclusion that to make this model is accurate, it is important to account for regular modules like memory, as they will present activity factors different from other logic. Some methods for this is presented in [16].

2.9 PREVIOUS WORK

Extensive literature searches with the search parameters protocol, processor, ASIP, CPU, bit-banging, bit, bang, software peripheral, software, SPI, I2C and UART in many different combinations, has been performed. However, there has not been found any previous work that try to create a processor optimized for bit-banging. Processor designs for wireless protocols[17] and Ethernet protocols[18] were found, and although processor design in general is related work, the protocols implemented are too dissimilar to the protocols considered in this thesis.

A single article concerning the effectiveness of software-implemented UART was found[1]. Lioupis et al. investigates the power consumption of implementing UART as software as compared to a hardware implementation. It is assumed that the processor is active with other tasks 80% of the total running time, and that there is no penalty for interrupting the CPU. They do not specify where their numbers for CPU and UART power consumption are from, and they conclude that for baud-rates lower than 65Kbits the 30MHz processor is in fact more power efficient than the dedicated hardware. An excerpt from their results can be seen in figure 2.9. The CPU's power cost increases linearly with the baud-rate, while the dedicated hardware is considered to have constant power consumption for all baud-rates. Thus, as the baud-rate increases the processor becomes a less efficient solution.

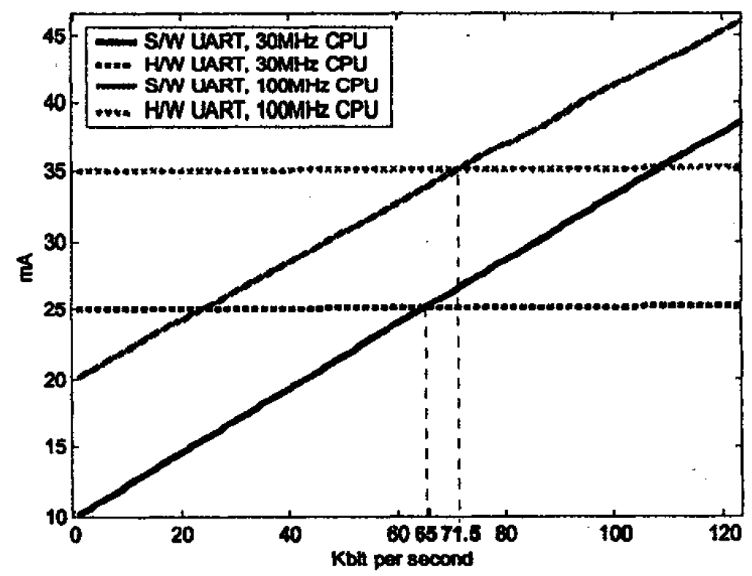

Figure 2.9: Power consumption of UART implemented on a CPU compared to a dedicated hardware module[1]

3 PROCESSOR ARCHITECTURE OUTLINE

While an instruction set architecture can be largely independent of the hardware implementation it is necessary to define the functional structure of the processor in order to design the instruction set. This chapter will present the processor architecture that will be used as a basis for instruction set development. The bit-banging processor will throughout the thesis be referred to as the *BBP* while the host will be referred to as the host processor.

It is expected that the BBP will not perform data-heavy operations. Indeed, its main task is to receive data from some other source and output that data on a bus. For the serial protocols considered in this thesis this data is usually sent one byte at a time. Consequently, it is assumed that an 8-bit processor will perform well for these tasks. Even though most serial protocols uses 2–4 I/O-pins the BBP is given the ability to connect to eight separate I/O-pins. This is to promote flexibility, and enables the BBP to control up to 5 slaves in an SPI configuration.

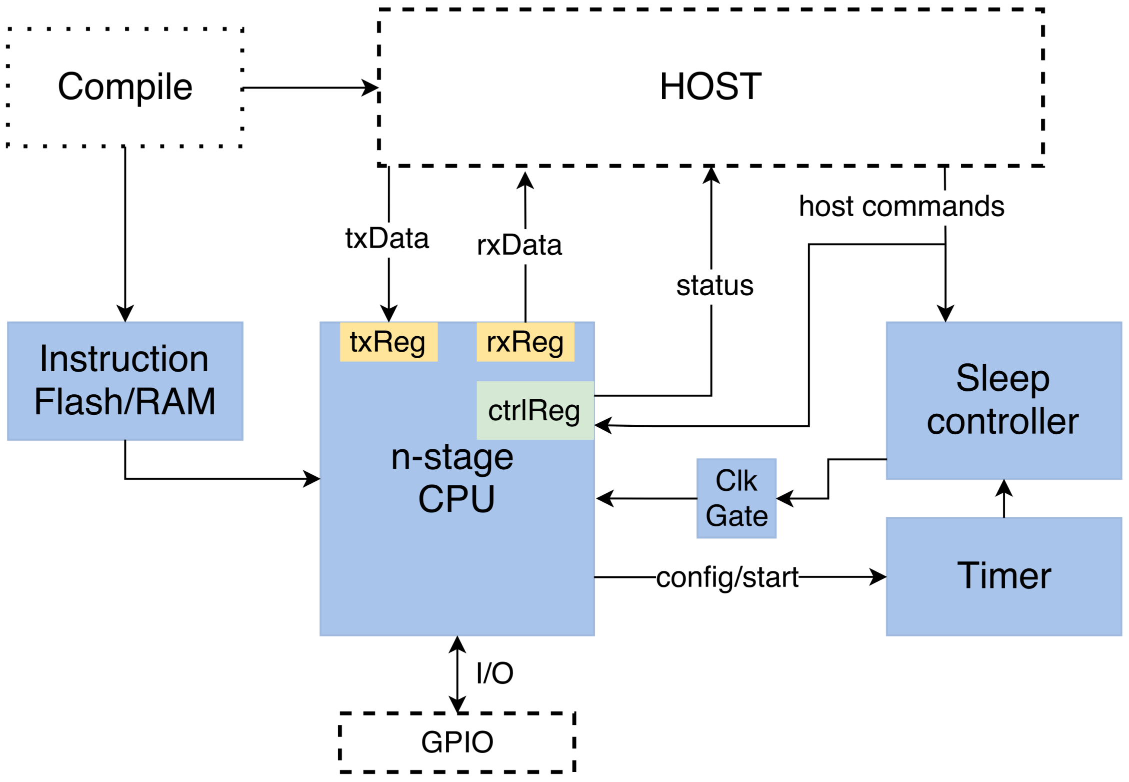

Figure 3.1: Bit-banging processor architecture illustration.

The BBP is a co-processor designed to have an interface to its environments similar to that of a peripheral module. Low power and efficient I/O-accessing are the predominant focuses of this architecture. In figure 3.1 an illustration of the architecture is shown. The BBP must have some interface to the host processor. Transmit data, received data and control signals, are necessary interfaces. The transmit/receive data will be put in special host-accessible registers, TXREG and RXREG. The host can write and read these registers respectively, and they should also be connected to a *Direct Memory Access*(DMA) system to relieve the host processor from the task of reading and writing data to the peripheral. Between the DMA and the host control there is no need for the BBP to access data memory, and both instructions and hardware used for data memory access can be excluded. Thus, implementing the BBP does not introduce a new type of master on the memory bus of the system, and integration into existing solutions is simplified. The host processor must be able to control transfers on the BBP and a control register (similar to the control registers in peripherals) should be included. This register will include all host processor related communications like start new transfer, transfer finished, data ready, error and stall. The registers should be accessible even when the BBP is in sleep mode.

The communication protocols are dependent on accurate timing, this is particularly true for the asynchronous UART protocol. Timing could be resolved by using NOP-s to wait between I/O-updates, but this would leave the BBP always active and is a very inefficient way of waiting for the next event. A counter based timer has been added to the architecture to allow the BBP to sleep while waiting for the next time action is required. The timer can be configured by the host and the BBP, and will act as a baud-rate generator for the BBP. It will be started at the beginning of operation and count continuously asserting a wake-up event signal whenever a specific count is reached, and restart the count. The counter should be dynamically configurable to allow for clock synchronization which is especially important for UART.

The BBP should have a low wake-up time from sleep mode. This can be implemented using an external clock gating module[19]. The clock will be gated whenever the BBP makes a call to the wait for event(WFE, see section 5.1) instruction. The events that will wake the BBP include a start transfer command from the host processor and when the external timer overflows. It would be possible to add some edge checker that could wake the BBP given some start sequence on the I/O, but this has not been considered when implementing the instruction sets. When the BBP calls the WFE-instruction the BBP's clock will be stopped and all activity will be ceased, but the state of the BBP will be retained. Thus, when the clock is started again the BBP will immediately continue with the next instruction. A single cycle wake-up is preferred, but may not be necessary for correct functionality. Additionally the module as a whole should be power-gateable, where only the instruction memory is retained. It is assumed throughout the rest of the thesis that the BBP is powered down whenever it is not used.

It is assumed that the BBP will not be dynamically switched from one protocol to another. This is a reasonable assumption because the modules connected to the I/O-pins are not likely to be dynamically changed. Thus, the program(the protocol) that the BBP will run can be put into a local instruction memory at compile time. Dynamical switching between protocols can easily be added by extending the size of the instruction memory and allowing the host to control the starting address of the BBP.

Slave protocols for SPI and I2C has not been implemented in this thesis, but UART-RX has a slave-like behaviour and some thought has been put into how one could improve upon these protocols. Generally, a slave behaviour implies a lot of waiting for a master to establish contact. Using the BBP to wake up at every interval to check for a transfer is not a very economical solution. For SPI this essentially means waiting for the chip select to be asserted and can be easily added to the clock gating circuitry. In I2C the slave has to check the address every time a master initiates a transfer on the bus, but can ignore the data that follows the address if it is meant for another slave device. This is more complex and requires circuitry that can detect the *start of transmission* signals, and a separate detection module would probably be necessary. For UART the start bit is just a transition from high to low, and can be implemented with the same circuitry as the edge detector for the SPI signal. These features has not been considered when comparing instruction sets in the following chapters.

4 INSTRUCTION SIMULATOR

To compare and analyse different instruction sets, a CPU simulator was implemented in C/C++. The simulator is a cycle-accurate instruction set simulator that includes internal registers and IO-pins. Programs can be run by writing assembly-like code in the form of C-functions. The simulator produces cycle-accurate waveform-files for debugging and report-files for further analysis of the performance. A C++ testbench ensures that the assembly-like programs adheres to the protocol specified for each program. Each of the protocols implemented has a single test set, and these test sets is used for all instruction set implementations.

4.1 STRUCTURE

The simulator is constructed around a set of functions representing the instruction set, for clarity I will call these functions *I-functions*, example I-functions can be seen in algorithm 4.8. A program is designed by calling these I-functions sequentially, simulating the way an actual microprocessor executes instruction words. These assembly-like programs will be called A-programs, and an example A-program is shown in algorithm 4.11. Each I-function implements the functionality expected in the corresponding instruction, including reading and writing registers, I/O-pins and other programmer changeable parts of the processor. For this purpose, arrays (with separate sets of read and write functions) is implemented for both registers and IO-pins. A typical I-function simulating a logic instruction will read operands from the register file, perform an operation on the operand, and store the data back into the register file. Just like you would expect an instruction to be performed in a microprocessor. Additionally, reporting functions is used in the I-functions, register file and I/O-pins to dump new values to a waveform file and report statistical information to a report file.

The structure of the system can be seen in figure 4.1. The files inside the yellow box constitute the framework of the simulator. It includes a register file, regFile.c; an I/O-pin structure, ioMap.c; a reporting structure, report.c and a waveform generation structure, vcd.c. The framework is the same for each instruction set architecture implementation. When new functionality is needed in a new instruction set, new instruction set specific functions can be added to the framework without affecting the other implementations.

The files in the blue box in figure 4.1 are instruction set specific files. Each implementation of an instruction set architecture has a unique *instr.c* file, and it contains the I-functions that describe the instruction set architecture. Assembly-

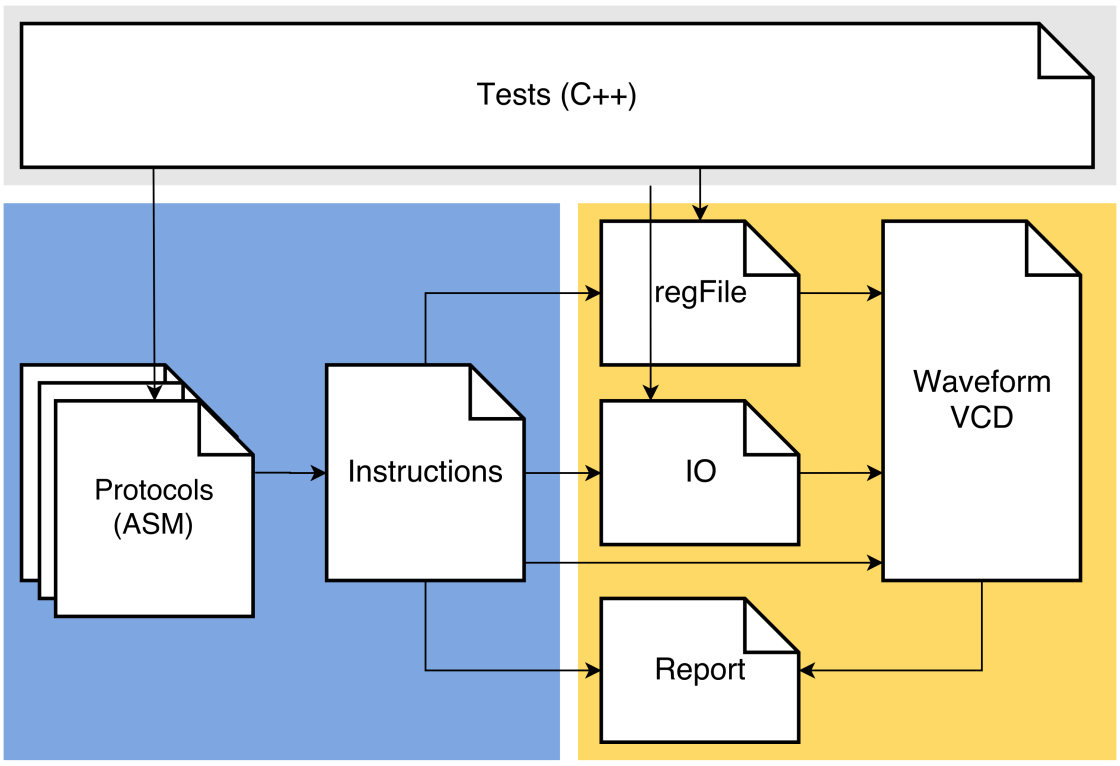

Figure 4.1: File structure and control flow of the instruction set simulator.

like implementations of the protocols studied in this thesis are included for each instruction set, these A-programs implement the communication protocols SPI, UART and I2C.

Finally the grey box includes the testbench. The testbench uses the CppUTest framework [20][21]. There is one set of tests for each protocol and the tests are used to ensure that all A-programs comply with the expected protocols. The tests, which will be described in further detail in section 4.6, initializes registers to some known value, loads data into the input array and then runs the A-program.

4.2 REGISTER FILE

The register file contains an array of int-s that represent the general register file, and include functions to read/write a single element in the array and to reset the array. The two special registers, TXREG and RXREG are also included in this array and can be handled like regular registers but special functions to allow for the SOL-functionality(see section 5.4), has also been implemented. In addition to updating the array, the register write functions also call functions related to the updating of the waveform.

4.3 I/O-MAP

Because of the pad structure described in section 2.7, the array in ioMap has four elements; one for each pad control signal. For testing it is important to keep a record of what the output has been at any time. To store this data the array is two dimensional and is created to be long enough to store all changes to each pad control signal separately. Whenever a write operation is performed on the IO, the program will increment a counter and at the end of the program, when checks are performed, will retain all outputs. This also allows the input array to be loaded before the program is executed. When read-operations are performed, the function will read the next data in the array.

4.4 WAVEFORM GENERATION

The waveform generator creates a *Value Change Dump*(VCD) file that can be read by most waveform viewers. The structure of a VCD-file is set up like shown in figure 4.2. In few words, a VCD-file contains a list of value and time updates. In the highlighted segment in figure 4.2 value A is changed to 0 at time 3400 and then back to 1 ten nanoseconds later. Notice that time updates are not described as increments in time, but as updates of the absolute time. The specification of value change dump files can be read in [22, p. 325].

The waveform generator creates and initializes the VCD-file. The initialization include giving the file a date-stamp, declaring and naming the signals and setting all signals to be initially unknown, X. The values of the variables is updated in two separate functions: one for the register file and one for the I/O-pins. These functions takes the corresponding register or I/O-pin and the new value as arguments, and writes a new line to the VCD-file with the corresponding value update. To improve debugging, a single time-step is also added after every value update. This allows the user to explore the sequence the values has been changed in.

```
$dumpvars
#0
xwireA
bxxxxxxxx reg0
$end

#3300
b00000000 reg0
#3400
0wireA
#3410
1wireA
#3500
b10101010 reg0
```

Figure 4.2: VCD-file

The waveform generator operates with three different time counters: delta-time, clock-time and timer-time. The delta-time is incremented by one every time a variable is updated. The clock-time is used to emulate the processors clock cycles and timer-time emulates the external timer's period.

To simulate processor cycles, a wait cycle function is created. This function increments the clock-time with the clock period and updates the active time in the report. It is checked that delta-time is not larger than the new clock-time. If it is, too many variables has been changed within a clock cycle and this signifies an error. Delta-time is set equal to the clock-time, and the time in the waveform-file is updated. If the clock frequency simulated is increased significantly, one may have to remove the delta-cycle functionality or change the time-precision in the waveform.

Similarly the wait timer function simulates the timer's cycles. Timer-time is updated by a preset timer period and it is checked that neither delta-time nor clock-time is larger than the new timer-time. In the protocols implemented in this simulator the timer is in charge of controlling the frequency on the communication channel. Thus, the active time(the clock-time) should never be larger than the timer-time. If it is, too many instructions are performed between each time the communication channel expects updates and the processor is not able to adhere to the timing constraints of the communication channel. Finally, both delta-time and clock-time is set equal to the new timer-time. The waveform is updated with the new time and the sleep-time in the report is increased by the difference between timer-time and clock-time.

4.5 REPORT GENERATOR

To compare instruction sets, statistical information about each implementation is needed. Specifically, how many times an instruction has been performed, how many pipe flushes has been performed and how long the processor has been active/sleeping are important parameters. The report generator monitors these data and write them to a file for further analysis. It implements an instruction reporting function, a pipe flush reporting function and active/sleep reporting functions. Every I-function will make a call to the instruction reporting function, and jump-functions also make a call to the pipeline flush function. The functions simulating cycle increments and sleep in the waveform generator will call the active and sleep function respectively. The report generator creates a file and stores all the information as comma separated values(CSV) that can be analysed in Excel or the data analysis tool of your choice.

4.6 TESTBENCH

Hardware engineers are constantly writing tests to check that their modules comply with some expected operation. A common way to perform tests is to connect to the interface of the module and use the testbench to simulate the environment that the module will be put in. This requires the testbench to adhere to timing and for the testbench to be able to run in parallel with the unit under test. These concepts are not entirely applicable for tests in C. The timing in the simulator is only a construct that is emulated when creating the VCD-file and is not really a part of the operation. Furthermore, running a testbench in parallel with the simulation is difficult to achieve. C/C++ tests are usually tests that runs a function with some given inputs and then checks that the outputs are as expected. CppUTest is a unit testing framework made for testing C/C++ code. It contains functions and macros that enable the running of groups of tests with simple constructs that are quick to write. Checking macros that will report errors to terminal, and generally simplifies the testing, are also included.

In hardware testing we control the inputs of the module and expect some reaction from the module, it is not so for C-testbenches. A test will first initialize the structures in the simulator to some specific values(e.g. the TXREG and the input array), it will then run the A-program for a set number of iterations. When the A-program finishes it can check that the structures are set to the correct values. This include checking that the RXREG is the correct value and that the outputs has been the correct values at the correct times. Checking the RXREG is a simple comparison between the value stored in register file array and an immediate, but checking that the output sequence has been correct is more complex.

In a HDL-testbench the values on the bus will usually be checked periodically on the positive edge of some clock and compared to the expected value in that instant. In the C-testbench we must check that the output-array contains the correct sequence of data, this requires intimate knowledge of how the output-array is updated. For the synchronous protocols, SPI and I2C, we can in fact emulate the positive edge checking nature of HDL-tests. A function which will start at some offset, and then for every positive edge of the array representing the clock wire, will check the bit representing the data-bit. For the asynchronous UART protocol there is no clock output to synchronize the checks to. Luckily, unlike the synchronous protocols that will update several different outputs(clock, data and select), the UART A-programs only access the output when new data is written. Thus, the output array contains only the data information and one can simply iterate through the output array to collect the values and compare them to the expected stream.

4.7 I-FUNCTIONS

Each instruction set has a unique set of I-functions that mirrors the instructions in the set. Each I-function should replicate the behaviour of its corresponding instruction and should update both the report and the waveform whenever called. In algorithm 4.8 the implementations of two instructions are shown; AND and OUT. The AND-function(algorithm 4.8(a)) reads the registers, performs a bitwise AND on the data and stores the result to the storage operand. Additionally, it calls the report instruction function from the report file to increment the number of AND-operations performed, and calls the wait cycle function from the waveform generator to indicate that time should be increased by one cycle. The register write functions also calls a function within the waveform generator to update register value in the waveform.

The OUT-function(algorithm 4.8(b)) reads the operand register and stores the value in the indicated IO-pin; either output, output enable or input enable. The OUT-function also calls the report instruction and wait cycle functions(as every I-function does). The IO-arrays are updated with the IO-write function, which also updates the waveforms.

Algorithm 4.8 I-functions used in the simulator.

(a)AND

```
void and(unsigned int regd,
         unsigned int rega){
  reportInstr(I_AND);
    vcdWaitCycle();
  unsigned int vald, vala;
  vald = regRead(regd);
  vala = regRead(rega);
  vald = vald & vala;
  regWrite(regd, vald);}
```

(b)OUT

```
void out(io_t padCtrl,
         unsigned int reg){
  reportInstr(I_OUT);
  vcdWaitCycle();
  unsigned int val;
  val = regRead(reg);
  ioWrite(padCtrl, val);}
```

The functions controlling the flow of the A-program(e.g. jump and compare) are slightly different. In regular assembly code looping and conditionals are solved by jumping to the appropriate label. In C-code looping will be solved using the *FOR* statement, but the goal of this simulator is to create an assembly like structure that can later be converted into true assembly code. To mimic this type of assembly behaviour in the C-simulator the A-program was divided into sections where each section is a function. The JMP-function will call the appropriate section. In long loops this can lead to a deep call-stack, where the section recursively calls itself N number of times. In the very late stages of the thesis work the C-statement *goto* was found. It does not only mimic assembly-code but executes in exactly the same way, and deep call stacks are avoided[1]. Due to the late discovery the goto-statement is not used in this simulator. The JMP function is shown in algorithm 4.9 and takes a function pointer as an argument. The JMP function performs its reporting and waveform updating and then calls the function which was passed to it.

Algorithm 4.9 jmp-function

```
int jmp(void (*section)()){
    reportPipeFlush();          //Report pipeline flush
    reportInstr(I_JMP);         //Report instruction
    vcdWaitCycle();             //Register CPU cycle
    section();}                 //Section call
```

Instead of conditional branching compare-skip functions are implemented, and this is further described in section 5.1. The compare skip if (not)equal functions are divided into two sets of functions; one set for logic functions(*cpseLogic*) and one set for jump functions(*cpseJmp*). This is because the logic functions

1 While the *goto* is a preferable way of solving the jump function, it can complicate the cpseJmp-type functions. You need to pass the address of the label into the jmp and (particularly) the cpseJmp functions, and getting the address of a label is not supported in regular C. However, the GCC compiler supports this and '&&' should be used, instead of the regular '&', to get the label address.

take a number of integers as their arguments, while the jump functions takes a function pointer. The *cpseJmp* function is shown in algorithm 4.10 and takes a register, an immediate, a jump function and a section function as arguments. If the condition is met it will call the JMP function which in turn will call the label function. If the condition is not met it will call a NOP. The *cpseLogic*-function has a similar structure but will take integers as arguments instead of the function pointer(**section*) and call a logic function with these integers as arguments.

Algorithm 4.10 cpseJmp-function (Compare Skip if Equal)

```
int cpseJmp(unsigned int rega, unsigned int imm,
            void (*jmpFunc)(), void (*section)()){
  reportInstr(I_CPSE);         //Report instruction
  unsigned int val;
  val = regRead(rega);         //Read register
  vcdWaitCycle();              //Register CPU cycle
  if (val != imm)              //If register value not equal
  {
      jmpFunc(section);        //Jump instruction call
      return 1;                //Return 1 to indicate that jump was taken
  }
  else                         //If equal, skip
      nop();                   //Perform nop
  return 0;}                   //Return 0 to indicate that jump was not taken
```

4.8 A-PROGRAMS

The A-programs use the I-functions to create an assembly-like calling structure. An assembly program and an A-program that outputs an eight bit value using a loop, is shown in algorithm 4.11(a) and 4.11(b), respectively. In the A-program one will first notice that labels in the assembly program are functions instead. The output8Bits function will run its initializing operations and then continue to the loop function. The loop function will run through its operations and finally recursively call the loop function until the condition is met.

As seen in algorithm 4.11, the instructions between each label in assembly is divided into *sections* and constructed using functions in the simulator. If the user wishes to jump out of a *section* if some condition is met, this is enabled using *if* in the C-simulator. The cpseJmp function shown in algorithm 4.10 returns 0 if the branch is not taken and returns 1 if the branch is taken. Thus a jump out of a section can be performed by using an *if* to check whether *cpseJmp* returned 1 and then exit the section by calling return. This is shown in a concept program in algorithm 4.12. The flow of this program can be seen in figure 4.3, and will be described in detail.

In the case when R0 is equal to 1, the assembly program(algorithm 4.12(a)) will skip the JMP function and continue to run the instructions on the label *eq* and *finish*. The simulator(algorithm 4.12(b)) will also call the *cpseJmp* func-

Algorithm 4.11 Program that outputs data serially, MSB first.

(a)Assembly	*(b)Simulator*

```
output8Bits:                void output8Bits(void){
   ldi R0 0xAA                 ldi(R0, 0xAA);       //Init data
   ldi R3 0x80                 ldi(R3, 0x80);       //Init counter
   timst 1000                  timst(1000);         //Init timer
                               loop();
                            }
loop:                       void loop(void){
   mov R1 R0                   mov(R1, R0);         //Copy data
   andi R1 0x80                andi(R1, 0x80);      //Mask MSB
   wfe                         wfe();               //Wait for interrupt
   out OUT R1                  out(OUT, R1);        //Put the bit on the OUT-bus
   lsl R0 1                    lsl(R0, 1);          //Shift data left
   lsr R3 1                    lsr(R3, 1);          //Shift counter right
   cpse R3 0x00                cpseJmp(R3, 0x00,    //Jumps if R3 != 0
   jmp loop                            jmp, loop);} //Else, performs a nop.
```

tion which will perform a NOP and return zero, it will then continue to the code labelled *eq* and then call the *finish* section to perform the final part of the instructions.

When the branch is taken (that is, when R0 is not equal to 1), the assembly code will perform the jump, do the operations following the *notEq* label and finally jump back to the *finish* label. The simulator will call *cpseJmp* function which calls the *jmp* function which in turn calls the *notEq* function. The instructions in *notEq* will be performed and finally another call to the JMP function to call the *finish* function. When the finish function has run all its instructions it will return, and *notEq* will return and finally the *cpseJmp* function will return 1, and the *ifElse* function will also finish without performing the instructions following the *eq* label. This may seem complex, but all the user has to think about is to add the if-return part of the code to ensure that unwanted code is not run.

Algorithm 4.12 Program with if-else type structure.

(a)Assembly

```
ifElse:
   //Do something
   cpse R0 0x01 //if not equal
   jmp notEq
eq:
   //Do something


finish:
   //Do something

 notEq:
   //Do something
   jmp finish
```

(b)Simulator

```
 void ifElse(void){
    //Do something
    if( cpseJmp(R0, 0x01, jmp, notEq) )
        return;
//eq:
    //Do something
    finish();
 }
 void finish(void){
    //Do something
 }
 void notEq(void){
    //Do something
    jmp(finish);}
```

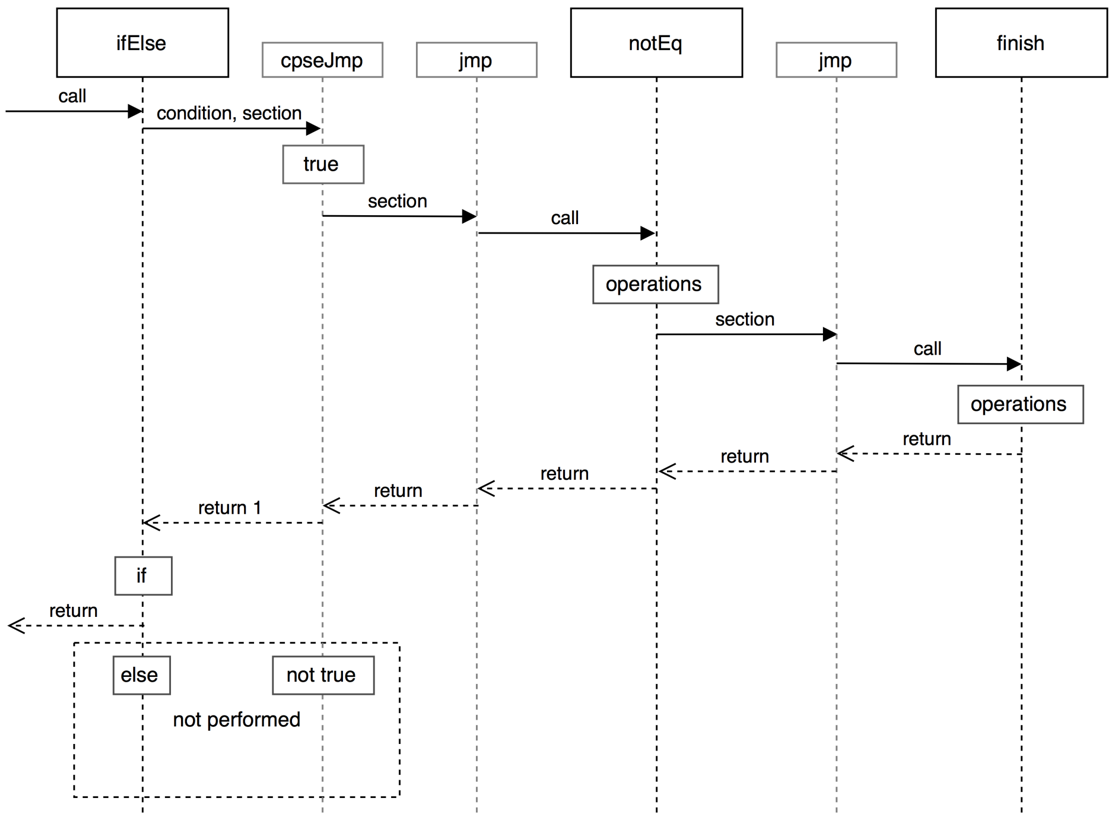

Figure 4.3: Flow graph of the algorithm shown in algorithm 4.12b.

4.9 BENCHMARK

The power estimation of the processor will be based on the power consumption in active mode and the average ratio of time spent in active/sleep mode respectively. To get good measures of the average ratio it is important that the protocols are run in probable use-cases. For the two synchronous master protocols, SPI and I2C, it is assumed that whenever the processor is not performing a transfer it is in sleep mode waiting for an interrupt from the host. Therefore the use-case reduces to considerations of how long each transfer will be. Before data can be sent the protocols demand that the transfer should be initiated in some particular way. The initialization cost of sending a single byte will then be higher per byte than when transferring several bytes. A significant amount of effort has been put into trying to find statistical information about the use-cases of the protocols. In the end the transfer length is application specific and the power consumption will therefore also be application specific. In the benchmarks used for I2C and SPI the single byte transfers and several byte transfers are therefore spread somewhat evenly; performing a single nine byte transfer, a three byte transfer, a two byte transfer and three single byte transfers. One could argue that more transfers should be performed, but as long as you keep the ratio between the lengths of transactions equal, increasing the number of transfers would not change the result. The asynchronous UART protocol is a bit different. Because a RX-transfer can occur at any time, the UART has to wake up and check for start bit at every transaction time. It is reasonable to assume that much of the running time in UART may involve idling with no communication on the bus. This is also represented in the benchmarks. In total two TX single byte transfers, one single byte RX transfer, one transfer where the bus is idle 60% of the running time and one test where running for the equivalent of seven bytes while transferring four RX bytes and five TX bytes. For the comparison of instruction sets it is also important that the protocols are weighted equally, the benchmarks has therefore been tuned to run for an approximately equal simulation time[2].

2 The time simulated by the simulator. Not to be confused with wall-clock time, the actual real world time the simulation takes to run.

5 INSTRUCTION SET ARCHITECTURES

In this chapter the development of the instruction sets will be presented in the order they were developed. To show the chronological process of my work, each section but the first, will first discuss the analytical reasoning for the new instruction set based on the results of a previous instruction set. Then, the instruction set is presented, and finally simulation results for each set is given in each section.

5.1 MINIMAL INSTRUCTION SET

Table 5.1: Minimal Instruction set

Mnemonic	Description
AND	Logical and
ANDI	Logical and immediate
OR	Logical or
ORI	Logical or immediate
XOR	Logical xor
XORI	Logical xor immediate
LSR	Logical shift right
LSL	Logical shift left
MOV	Copy register to register
LDI	Load immediate
CPSE	Compare skip if equal
CPSNE	Compare skip if not equal
CBSE	Compare bit skip if equal
CBSNE	Compare bit skip if not equal
JMP	Unconditional jump
IN	Read IO
OUT	Write IO
OUTI	Write IO immediate
TIMST	Set up and start timer
WFE	Wait for event

As a baseline for development a minimal set of instructions needed to run the protocols was implemented. The name *minimal* should not be confused with minimal instruction sets in general. The protocol implementation was analysed and some simplifications were identified. The processor needs to stringently comply with timing, it must be effective in reading and setting I/O-pins and it must be able to loop through a byte of data. Some instructions found in a typical instruction set are not needed. In the protocols implemented there is no need for arithmetic calculations. However, arithmetic operations are not only needed for performing calculations, but is also used for decrementing count registers in loop control. In the applications this instruction set is designed for, the largest loop has a length of eight. With 8-bit registers, counting eight iterations can be solved by setting a bit in the most significant position of the register and right shift the register until empty. We can then from the algorithms in section 5.1.1 see that there is no need for arithmetic calculations in this instruction set.

The minimal instruction set is a small, two-operand, RISC-based instruction set and can be seen in table 5.1. A two operand RISC instruction set was chosen for the reasons discussed in section 2.4. Most of the instructions should be known to the reader, but I will explain some in detail. You will notice that there are no conditional branch instructions in the list. Instead, *ComPare Skip if Equal/Not Equal(CPSXX)* and *Compare Bit Skip if Equal/Not Equal(CBSXX)* are added. These instructions will skip the next instruction if their condition is true. When a JMP instruction follows a skip-instruction you achieve the same functionality as you would with a compare and then branch-conditional combination, with the added benefit of being able to perform skips on all types of instructions. WFE and TIMST are instructions related to the event timer controlling the active period of the processor. The *Timer Start(TIMST)* instruction is used to configure and start the timer. Instead of using the BBP to configure the timer one could let the host take care of this operation. But, anticipating continuous frequency adjustments to synchronize to external modules, the configuration is left to the BBP. The *Wait For Event(WFE)* instruction is borrowed from the ARM instruction set. WFE sets the processor in sleep-mode by indicating that the clock should be gated, and lets supporting modules know that it should be woken on some event. In the planned architecture the available events are overflow in the timer and a wake-up from the host processor.

5.1.1 PROTOCOLS

An overview of how the protocols are implemented in the simulator will be given. The assembly code may not be completely straight forward at times and please use the pseudocodes presented in section 2.2 for reference.

All the protocols have a set-up phase where the outputs and inputs are enabled and the timer is configured and started. The set-up for SPI is shown in algorithm 5.13, and is very similar for the other protocols. The *outi*-functions set OE_N (output enable active low) to zero for all outputs needed for the protocol, and *IE* (input enable) to one for all inputs used. The *timst*-function configures the timer to send an interrupt whenever it counts to *SPI_CK_TIME* and starts the timer.

Algorithm 5.13 Start-up A-program for the SPI-protocol

```
void spiMInit(void){
    outi (OE_N, 0xFF ^ (1<<MOSI | 1<<SCKO | 1<<CSn));   //Enable outputs
    outi (IE, 1<<MISO);                                  //Enable input
    timst(SPI_CK_TIME);}
```

SPI MASTER

The SPI protocol is the simplest to design. As described in section 2.2, SPI has many possible configurations. Only one configuration is implemented, namely a leading edge, non-inverted clock, MSB-first configuration. The assembly program does not need to be configurable as the program itself can be changed if one wishes to use a different configuration. A transfer on the SPI master will always be initiated by the host processor. Thus, it can be started by an event from the host, run for the designated number of bytes and then go back into sleep.

A start-section where *CSn* is asserted and registers are reset, as well as a stop section where *CSn* is de-asserted, exists. The main part of the SPI-protocol can be seen in algorithm 5.14. Lines 4 to 6 copies, masks and outputs the MSB in TXREG. The other outputs, CSn and SCK, are both low at this point and will not be changed by line 6. In line 7 the data in TXREG is then shifted. Transmit-data is changed while SCK is low, and data is received while SCK is positive. The program goes into sleep until the next edge by calling WFE. The clock is gated until the timer overflows and enables the clock; the program then promptly cycles SCK high in lines 10 and 11. Because the protocol is a MSB first protocol the received data must be stored in the LSB-position of RXREG, and then shifted left. The left shift must take place before reading the data to avoid overflow after the last bit has been stored, and is performed in line 13 before the inputs are read in line 14. MISO may not be in the LSB-position of the eight bit registers and because the only active input is MISO, the data can safely be shifted right to store MISO at the LSB-position. The loop counter is shifted right, and the BBP waits until the next edge. Finally SCK is cycled low again, and on line 22 the program either loops back to the start(by calling itself recursively) or continues the execution(by returning).

I2C MASTER

Unlike SPI each I2C transfer consist of two separate phases: the address phase and the data phase. The data phase can in turn be either a write or a read. The program for I2C is consequently slightly more complex and a lot longer than the SPI implementation. The program is divided into start-up, address, write, read and stop. Instead of pulling the output up or down, I2C either pulls the output low or disconnects from the output by setting high impedance on the output pin. Thus in the start-up *OUT* is set to zero, while *OE_N*(see section 2.7) is logic high for all pins. When the program wishes to pull the bus low it sets *OE_N* low rather than controlling *OUT*.

In algorithm 5.15 the write loop is shown. Similar to the SPI algorithm, this loop cycles the clock and outputs the MSB of TXREG. Unlike SPI, the data should not be output on the negative edge of the clock signal, but rather while the clock is low. So SDA is changed in line 12, four instructions after SCL was pulled low to create a delay.

Algorithm 5.14 SPI byte loop. Full duplex communication.

```
//At start of loop: RXREG = 0x00 and loop counter R2 = 0x80.
void spiMRepeat(void){
  //Set MOSI at negedge
    mov(R0, TXREG);              //Copy data
    andi(R0, 0x00 | 1<<MOSI);    //Mask MSB
    out(OUT, R0);                //Output Bit. CSn and SCK low
    lsl (TXREG, 1);              //Shift left
    wfe();                       //Wait next edge
  //Cycle SCKO high
    ori (R0, 1<<SCKO);           //SCKO, !CSn
    out (OUT, R0);
  //Read MISO
    lsl (RXREG, 1);              //Shift left
    in(R1);
    lsr(R1, MISO);               //Shift data to LSB.
    or(RXREG, R1);               //Assuming RXREG = 0x00 at start.
    lsr (R2, 1);                 //Shift counter right
    wfe();                       //Wait next edge
  //Loop
    cpseJmp(R2, 0x00, jmp, spiMRepeat);}
```

Algorithm 5.15 I2C Write

```
//At start of loop: RXREG = 0x00 and loop counter R2 = 0x80.
void i2cMSendBit(void){
  //Cycle SCL
    wfe();                       //Wait until negedge
    andi(R0, 0xFF ^ 1<<SCL);     //Pull SCL low while holding SDA equal.
    out(OE_N, R0);               //Set OE_N
  //Set SDA
    mov(R0, TXREG);              //Copy data
    andi(R0, 0xFF ^ 1<<SCL);     //SDA = TX[7], SCL = 0
    lsl(TXREG, 1);               //Shift data
    lsr(R2, 1);                  //Shift counter
    out(OE_N, R0);               //Placed last to create delay
  //Cycle SCL
    wfe();                       //Wait until posedge
    ori(R0, 1<<SCL);             //Release SCL
    out(OE_N, R0);
  //Loop
    cpseJmp(R2, 0x00, jmp, i2cMSendBit);}
```

The ninth bit in both address and transmit phases is the slave acknowledge. The acknowledge program is shown in algorithm 5.16. If data has been properly received the slave will assume control over SDA and pull it low. The master must therefore release SDA, cycle SCL and then read SDA. If SDA is not equal to zero it should abort the transmit and initiate some error procedure. The first part of algorithm 5.16, lines 3–11, performs this operation. Notice that data is output two times, this is because transitions on SDA should not happen on the edge of SCL, but while SCL is low. The final part either stops the transfer if the current byte is the final byte or initiates another transfer. To allow the testbench to control how many transfers will be performed, a simulator workaround has been added. An iterator keeps track of the number of bytes transmitted and a dummy *compare bit skip if equal* is inserted to emulate the operation of the actual assembly command.

Algorithm 5.16 I2C Write Acknowledge

```
void i2cMTxAck(void){
 //Cycle ACK
   wfe();                              //Wait for negedge
   andi(R0, 0xFF ^ 1<<SCL);            //Pull SCL low
   out(OE_N, R0);
   outi(OE_N, 0xFF ^ (0<<SDA | 1<<SCL));   //Release SDA
   wfe();                              //Wait for posedge
   outi(OE_N, 0xFF);                   //Release SCL
   in(R1);                             //Receive ack
   if(cbseJmp(R1, SDA, 0, jmp, i2cMNotAck))//If not ack
       return;

   /***Simulator Workaround to allow testbench to run N iterations***/
   i--;                                //Decrement iterator
   cbse(CTRL_REG, I2C_STOP, 0);        //Dummy for checking
   if(i == 0){                         //If all transfers completed
       jmp(i2cMStop);                  //Stop signal
       return;
   }else{
       jmp(i2cMSendByte);              //Send new byte
       return;}}
```

UART

The UART protocol does not have a master-slave relationship. Both sides of a communication can initiate a transmit and must always be able to receive. This results in the UART having four different states as shown in figure 5.1.

The transfers themselves are quite simple. When woken from sleep by the timer: transmit or receive data and iterate the corresponding counter. The brunt of the operations lie in the switching between states. While the UART protocol does not have a clock signal, it has an internal baud rate generator. Generally the generator will create a pulse or a clock that the internal UART system uses to perform transfers at the correct time. For the processor the baud rate pulse is the wake-up signal from the timer, and it will transmit on one wake-up and receive on the next. A UART implementation typically synchronizes its internal clock to the other communicator's clock at the start bit, or every bit, of a reception. This has not been implemented in the simulator.

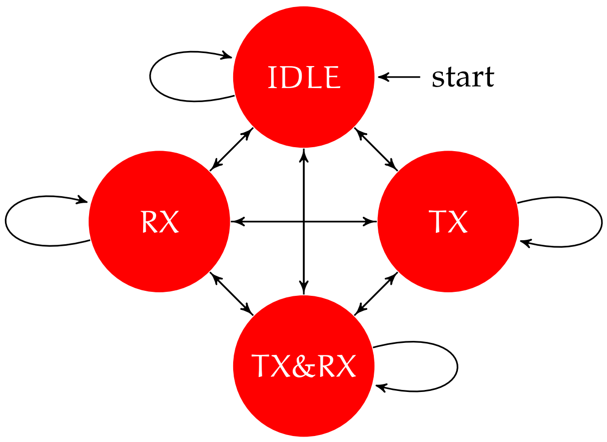

Figure 5.1: State chart for UART

As shown in algorithm 5.17 the processor will wake up, perform TX and go to sleep. It will then wake up again to perform RX and return to sleep. Algorithm 5.17 does not look like typical assembly code, but the functions uartTX and uartRX contain assembly like I-functions and are used as references to increase the readability of the code. The *for loop* enables the testbench to control how many iterations UART will run for, in an actual implementation UART is a never ending loop. In algorithm 5.18 the uartRx function is shown. RX can either be idling, checking for a start signal, or be active. In line 3 the program checks an internal status bit to determine if RX is running. If it is, a jump to the subroutine for RX-active is performed. Lines 5 and 7 reads the IO and checks if a start bit is present. Both RxRun and RxStart has `jmpDummy("uart")` as their final instruction.

Algorithm 5.17 UART top function

```
void uartFull(int cycles){
    int i;                          //Simulator workaround
    for (i = 0; i < cycles; i++){
        wfe();                      //Wait until pulse
        uartTx();                   //Perform TX
        wfe();                      //Wait until pulse
        uartRx();}}                 //Perform RX
```

Algorithm 5.18 Receive algorithm

```
  void uartRx(void){
      if(cbseJmp(CTRL_REG, RXRUN, 0,  //If running
3             jmp, uartRxRun))          //jump to RxRun
              return;
      in(R3);
      if(cbseJmp(R3, RXD, 1,          //If start signal
              jmp, uartRxStart))        //jump to RxStart
8             return;
      jmpDummy("uart");}              //Else jump to top
```

5.1.2 RESULTS

In figure 5.2 the simulation results for the minimal instruction set is shown. Unsurprisingly WFE is the most used instruction. The number of WFE-instructions are directly linked to the time the protocols have been running for. Thus, the number of WFE-instructions is generally unchangeable as it is the instruction that lets the processor wait for the next edge of the baud rate clock. XOR and XORI remain unused but are considered important instructions in a processor, and are kept to promote flexibility. Although unused, the AND function should not be removed simply because the ANDI function uses the same hardware to operate.

The considerable amount of NOP-instructions are due to the compare skip-functions. Whenever an instruction is *skipped* it is in fact a NOP being performed. Thus, it is a functional skip but the cycle is still consumed. Further analysis of the results will be performed in the sections below.

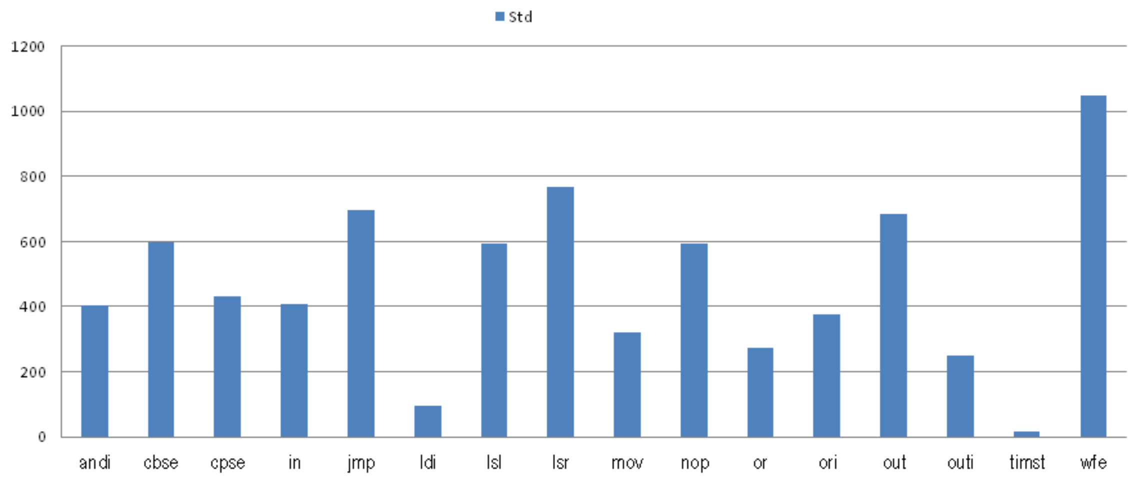

Figure 5.2: Simulation results for minimal ISA.

5.2 THREE OPERANDS

The minimal instruction set has a two operand implementation. As mentioned in section 2.4 three operand instruction sets will improve the flexibility in re-use of data. Thus, it will improve performance by avoiding some load-instructions by temporarily storing a value that is going to be reused. When analysing the A-programs one can see that the protocol applications are not data heavy. Generally the data passing through the processor will either be the values to be output or the counters used in loop control. Common for both of these is that they do not contain any constant information that can be read from some location. The values are either known constants set by immediates, such as the bus clock, or not reusable, such as the counters. These factors led to the assumption that a two operand instruction set would perform better for the processor in this thesis. Aside from using three operands, the instruction set architecture equal to the minimal instruction set.

5.2.1 RESULTS

When comparing the number of instructions performed, shown in figure 5.3, it can be observed that using three operands instead of two reduces the number of instructions. Specifically, the MOV-instructions used to copy TX-data into a register are removed(see algorithm 5.14 line 4. This improvement comes at the cost of longer instruction words. Longer instruction words requires higher complexity in the instruction decoder stage of the processor. More importantly, longer instruction words increase the memory requirements for the instruction memory. The cost is assumed to cancel out the fewer number of instructions in terms of power efficiency. There are more effective ways of reducing the number of MOV-instructions as will be seen in section 5.4, the power of the three operand instruction set will therefore not be estimated in the next chapter.

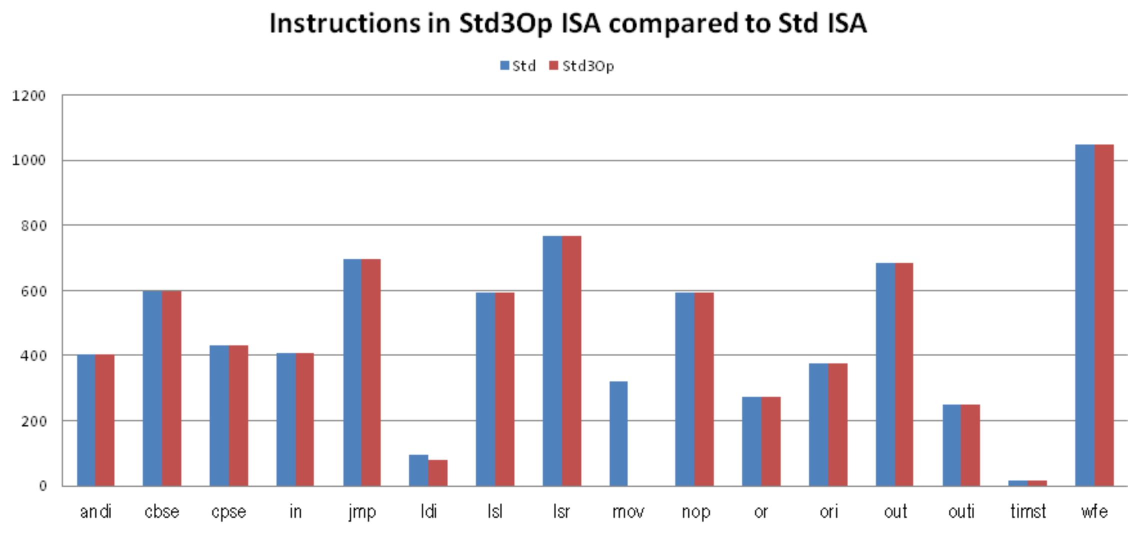

Figure 5.3: Simulation results for three operand ISA.

5.3 REPEAT

REPEAT	Repeat

The simulation results and A-programs of the minimal instruction set was analysed to find potential for improvement. It is not surprising to find that loop overhead is a large factor in the A-programs, this overhead includes compare-skip instructions, jump, load immediate and right shift operations. As can be seen in figure 5.2 these instructions take up a large part of the executed instructions. All the protocols are expected to output a series of bits and each bit will be output in the same way each time; looping is a natural consequence of the behaviour. One could solve this by unrolling the loops, but an important factor when designing for low power is to limit memory, and other solutions should be considered.

The REPEAT-instruction[23, p. 495] provides a possible solution. The REPEAT-instruction takes two arguments, the number of instructions in the loop, N, and the number of iterations of the loop, M. The REPEAT-instruction initializes and starts a control-sequence in the program counter, and the next N instructions are performed M times. A typical loop overhead in the minimal instruction set include: one load immediate, eight right shift, eight compare-skip, and eight jumps per byte. For each loop performed the number of cycles can be reduced by 24 by adding the REPEAT-instruction. Additionally, a normal jump instruction will also result in a pipeline flush. Depending on the hardware implementation the flush penalty will vary, but for the three stage pipeline planned in this thesis a flush penalty of two cycles has been assumed. Seven jumps will be performed during one of these loops, and this adds an additional 14 cycles saved per loop.

The hardware needed to add repeat-functionality is not inconsequential. A loop-controller must be added to the program counter path. It includes two decrement counters, two equality checkers, 5 multiplexers and an OR-gate [23, p. 496]. A possible hardware implementation is shown in figure 5.4 and includes, for clarity, a part of a program counter. When N-counter is equal to zero *repeatFlag* is set. This causes the M-counter to be decremented while the program counter is set from a stored initial value, the start of the loop.

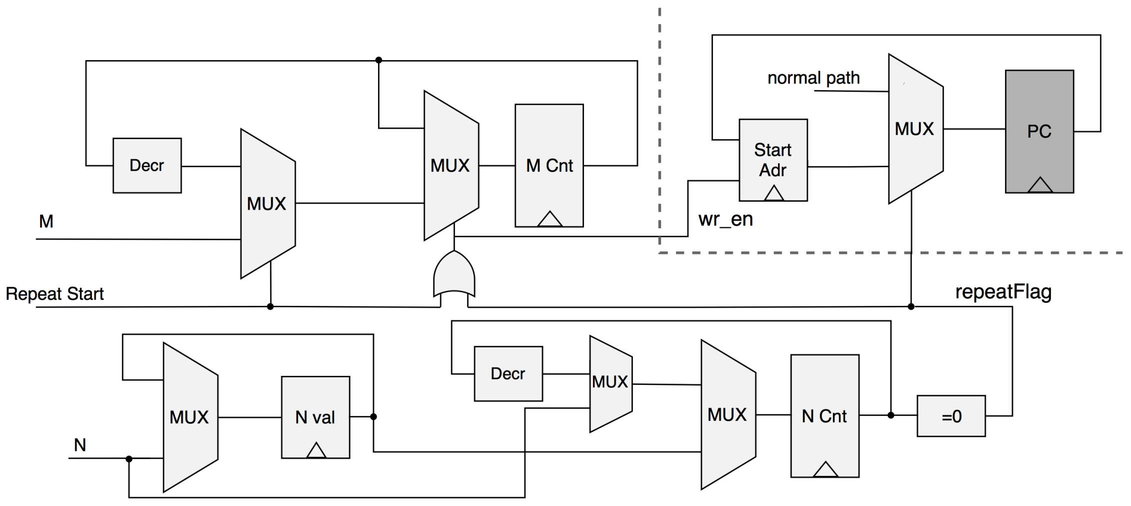

Figure 5.4: Illustration of hardware implementation of the REPEAT-instruction.

5.3.1 PROTOCOLS

SPI MASTER

The REPEAT-instruction takes iterations and code length as arguments and repeats the subsequent instructions a number of times. It is added in the simulator as shown in algorithm 5.19. Where the section *spiMRepeat* is a version of algorithm 5.14 without the loop-control instructions. SPI benefits greatly from the REPEAT-instruction. The final line of algorithm 5.14 can be removed, as well as the shift operation for the counter. The REPEAT-instruction has to be added before the loop is started, but the counter initialization is removed, resulting in no change in number of instructions. Three instructions and a pipeline flush for each iteration of the loop is removed.

Algorithm 5.19 Repeat added to SPI

```
void spiMTransferByte(void){
  andi(CTRL_REG, 0xFF ^ 1<<SPI_BYTE_READY);   //De-asserted ready signal
  ldi(RX_REG, 0x00);                          //Reset RX_REG
  //ldi(R2, 0x80);                            //Removed, set counter
  //spiMRepeat();                             //Removed, perform loop
  repeat(8, spiMRepeat);                      //Repeat bit-transfer 8 times
  ori(CTRL_REG, 1<<SPI_BYTE_READY);}          //Assert ready signal
```

I2C MASTER

The loop control instructions can be removed from algorithm 5.15 as well, and similarly for the transmit address and receive data programs. For each bit sent it also reduces the number of instructions by three and avoids the flush penalty.

UART

In the UART program a RX transfer may begin in the middle of a TX transfer, and this makes it hard to use the REPEAT-instruction. UART is an asynchronous, full duplex, protocol where both sides can initiate a transfer at any time. Thus, a RX-transfer may start or stop in the middle of a TX-transfer and this causes problems for the REPEAT-instruction. Performing jumps to subroutines within a repeat-loop will change the program counter. If the subroutines are not of an equal length the loop will perform instructions that are not supposed to be performed or skip instructions. If RX is not needed, a UART TX protocol will be able to benefit from the REPEAT-instruction and performance will be significantly better.

In spite of this, UART was implemented to include the REPEAT-instruction. In algorithm 5.20 it is shown how the RX program was padded with NOP-s to make all subroutines of an equal length. *RxRunRep* is five instructions long, and the pipeline flush adds another two cycles. *RxStartRep* only contains two instructions, but before it is started three instructions are performed. Thus, it must be padded with two NOP-s to be of equal length. Finally, assuming that the repeat counter does not pause when there is a flush, six NOP-s must be added for the RX-idle case. This makes the three cases of an equal length in number of instructions, and *UartTxRun* can be used in a repeat statement. This technique is very vulnerable to changes in the instructions and a compiler will usually struggle with trying to incorporate jumps within a repeat loop. Consequently, the technique used here is not well suited for user changeable code.

Algorithm 5.20 Nop padding to allow UART to use repeat.

```
void uartTxRun(void){
    wfe();
    mov(R0, TXREG);
    lsl(R0, TXD);       //Data is LSB first, shift data into correct position
    out(OUT, R0);
    lsr (TXREG, 1);
    wfe();
    //Run
    if(cbseJmp(CTRL_REG, RXRUN,0,
        jmp, uartRxRunRep))         //Length: 7 + 2(flush)
        return;
    //Start
    in(R3);                         //start: 1
    if( cbseJmp(R3, RXD, 1,         //start: 2
        jmp, uartRxStartRep))       //start: 3 + 2(flush)
        return;
    //RX idle
    nop();                          //Idle: 4 (flush)
    nop();                          //Idle: 5 (flush)
    nop();                          //Idle: 6
    nop();                          //Idle: 7
    nop();                          //Idle: 8
    nop();}                         //Idle: 9

void uartRxStartRep(void){
    ldi(R1, 0x80);                  //start: 6
    ori(CTRL_REG, 1<<RXRUN);        //start: 7
    nop();                          //start: 8
    nop();}                         //Start: 9
```

5.3.2 RESULTS

The simulation results shown in figure 5.5 show that the number of instructions has been reduced significantly. Although REPEAT was added to reduce the number of compare-skip and jump instructions, there are still a high number of them. This is mainly due to the checks in the UART protocol. The high amount of NOP-instructions also stem from these checks. Nevertheless, the

UART protocol has benefitted from REPEAT as can be seen in figure 5.15 at the end of this chapter. For the benchmarks the REPEAT-instruction reduces the running time by 23%. However, the REPEAT-instruction adds a significant amount of hardware, and if the energy efficiency is indeed improved remains to be seen.

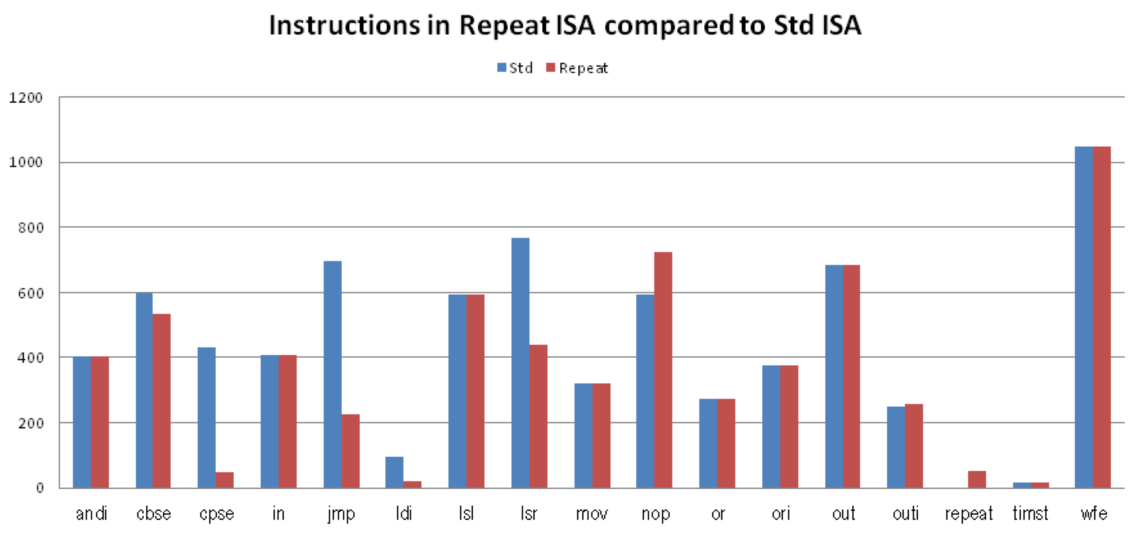

Figure 5.5: Simulation results for REPEAT ISA.

5.4 SOL INSTRUCTIONS

SOL	Serial Output Least Significant bit
SOM	Serial Output Most Significant bit
SIL	Serial Input Least Significant bit
SIM	Serial Input Most Significant bit

Further analysis reveals that reading and writing single bits on the bus is a slightly cumbersome operation. A general algorithm for outputting the most significant bit from a register is shown in algorithm 5.21, the least significant bit can be output by changing the immediates. Line 4 and 5 are necessary because the R1 register will contain information for all output pins and the other bits cannot be overloaded.

The general case can be optimised for each protocol implementation. If it can be assumed that the output wire, TXWIRE, will always be the wire in *the most significant* position(that is, position seven) line three can be removed. Furthermore, if the other outputs are known(for example SCK and CSn in SPI, or SCL in I2C) line four and five can be replaced by `andi R2 (0xFF ^ (1<<SCL | 1<<CSn))` for SPI, and `ori R2 (1<<SCL)` for I2C. When the output is

known one can also assume that all bits that are not set are don't care and we can remove line two, reducing the number of instructions by three in total. A similar analysis can be performed for reading bit-values. So with an optimised code, three instructions is used for outputting a bit, and three instructions are used for reading a bit. It can be argued that reading and writing bit-values is the main task of the implemented programs. It is therefore assumed that creating a specialised instruction to perform this set of operations effectively, would improve performance.

Algorithm 5.21 Outputting single bit from a register without destroying data in register or changing other outputs.

Precondition: R0 stores all bits that are in queue to be sent with MSB as the next bit. R1 has stored the current outputs on all wires. TXWIRE is the wire where the data should be put.

```
1  mov  R2 R0                  //Copy the data
2  andi R2 0x80                //Mask TX-bit
3  lsr  R2 (7-TXWIRE)          //Shift TX-bit to correct position
4  andi R1 (0xFF ^ 1<<TXWIRE) //Clear the current TX-bit in R1
5  or   R1 R2                  //Combine TX-bit with current output
6  out  R1                     //Send the data to out-pin
```

The SOL instructions are single cycle instructions that can perform this functionality. SOL/SOM take the LSB/MSB from a specialized register, put the bit into a general register and output the result. SIL/SIM reads all inputs into a general register and puts a single bit into the LSB/MSB of a specialized register. These instructions reduce the number of instructions needed for each write/read by two. In an 8 cycle loop where both reading and writing is performed this combines into a total of 32 instructions per byte transferred. Making the same assumptions as for the implementation in the previous paragraph, namely that the TX-bit always will be output on the same wire and that RX will either be on the same wire(Half-Duplex) or one other specific wire(Full-Duplex), the hardware size of such an instruction should be minor. The specialized registers mentioned will, in the case of the proposed processor, already be registers that are different from the general registers, namely the host-controllable RX- and TX-registers. A proposed, untested, implementation of the SOL instructions is shown in figure 5.7 and 5.6 for the IN- and OUT-functions respectively.

From figure 5.6 it can be seen that the SOL/SOM instructions introduce two new multiplexers in the processor; one to choose between MSB and LSB and one to insert the bit into the *most significant* wire in the data bus. This instruction can be extended to include bit insertion into any wire in the data bus by adding a decoder and multiplexers on every bit-line, but the *most significant* wire assumption holds well across all implementations in this thesis. When the

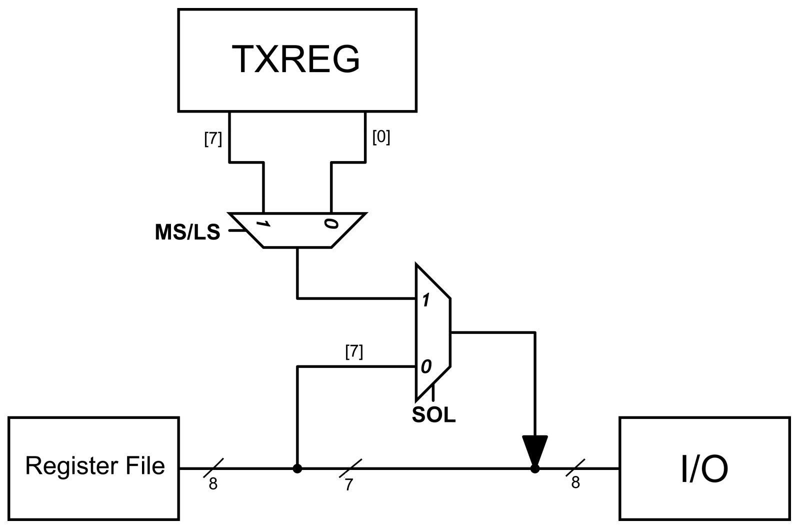

Figure 5.6: SOL TX instructions(SOL/SOM) in hardware

bit has been inserted into the data-bus it will follow the pre-existing datapaths of the processor and be written back to the register file.

The SIL/SIM instruction hardware is shown in figure 5.7 and introduce no new multiplexers in the existing datapath. It is essentially an IN-instruction with added functionality outside the general datapath. Three multiplexers are introduced, one to insert the RX-bit into the *most significant* wire of the RXREG-bus, one for inserting the RX-bit into the *least significant* wire of the same bus and one to switch between the *most significant* wire and the *second most significant* wire. The last multiplexer is necessary because some protocols are full-duplex and use two separate wires for TX and RX, while others are half-duplex and share a single wire for both TX and RX. With the assumption that TX is always set to the *most significant* wire, the SIL/SIM instruction must be able to chose between two different positions. Again, the SIL/SIM instructions can be extended to include all bit positions, this time by adding a larger multiplexer to span the whole bus of the I/O.

5.4.1 RESULTS

The simulation results in figure 5.8 show that there is a considerable reduction in number of cycles after adding the SOL-instructions to the minimal instruction set. For the benchmarks the improvement is 12% and considering the sparse increase in hardware these instructions represent, it can be assumed that the SOL-instructions improve energy efficiency.

5.5 SOL AND REPEAT

The SOL- and REPEAT-instructions were combined to create an energy frugal instruction set. The two most conspicuous candidates for optimization has been targeted, and the total improvement in number of cycles is 36%.

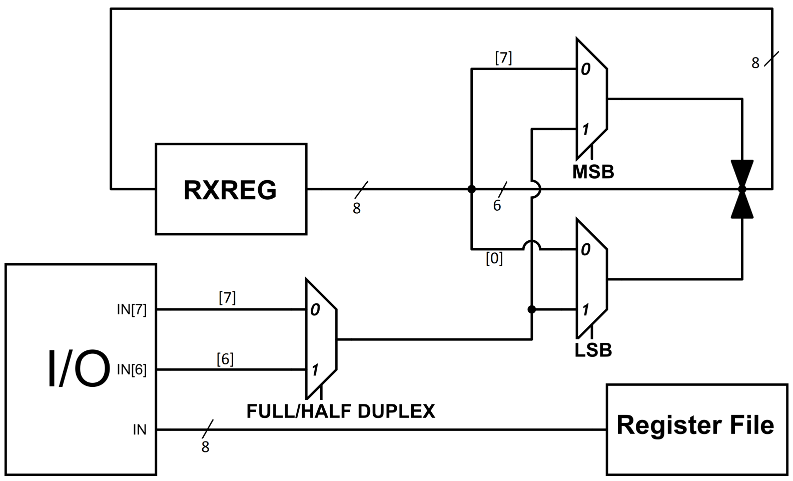

Figure 5.7: SOL RX instructions(SIL/SIM) in hardware

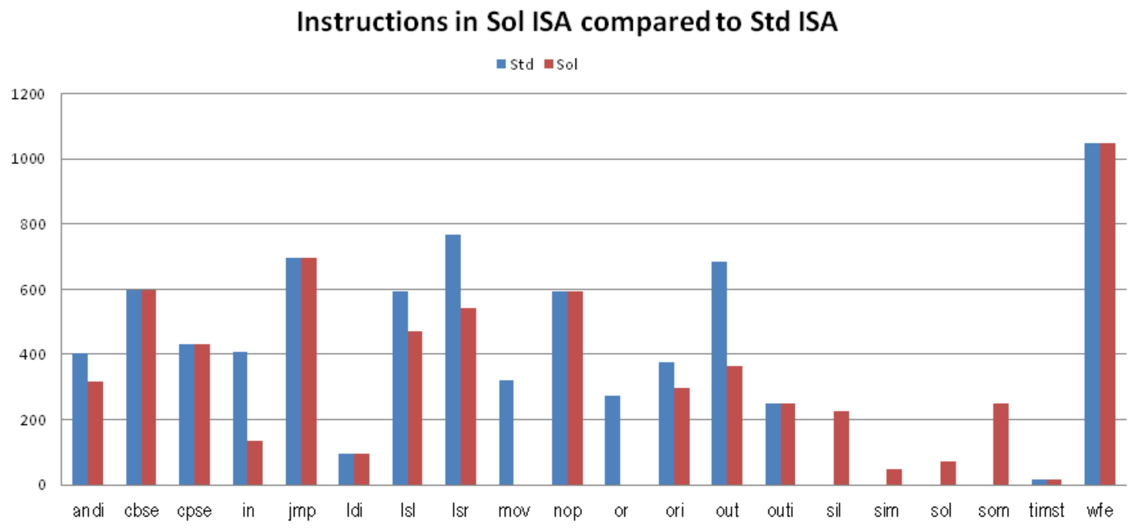

Figure 5.8: Simulation results for SOL ISA.

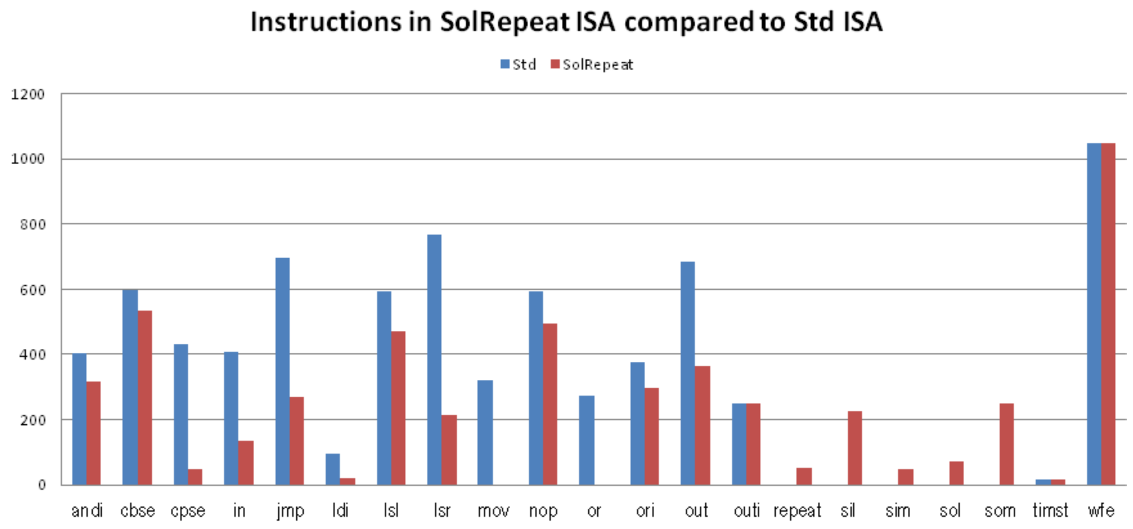

Figure 5.9: Simulation results for SOL and REPEAT ISA.

CLR	Clear conditional bit
SET	Set conditional bit
CPI	Compare Immediate, set conditional bit
CMP	Compare register, set conditional bit

The SPI and I2C master protocols easily lend themselves to be optimized because the processor, as master, can control when and for how long, information should be sent. The UART-protocol, on the other hand, is harder to optimize due to its *master-less* nature. As can be seen in figure 5.9, compare and jump operations still represent a large part of the total instructions, and most of these are in the UART program. These compare-jump operations are not due to loop overhead, but due to the several different states the UART program can be in. The states are showed in figure 5.1. The UART protocol may switch from one state to any other state at any time, and this makes UART more ineffective as a bit-banging protocol than SPI and I2C. Remember that every jump performed incurs a pipeline flush penalty, and the different states results in frequent jumps. In an attempt to mediate these effects, conditional execution was added to the SOL and REPEAT instruction set architecture.

Conditional execution is a property added to every instruction in the instruction set. It allows each instruction to be executed only if a condition register is in a certain state. In the ARM-instruction set this condition register is the *Application Program Status Register*(ASPR) [24], and is the same register as conditional branching performs its checks against(i.e. equal, carry, zero and negative flags). In the case of the UART program, the branching is usually performed due to some equality check, but the equality check is not any natural part of the program flow. Furthermore, when using a condition register that is set by several ALU operations—as is the case with ASPR—it is hard to store the state of the program over several instructions. For this reason a register containing four conditional bits that can be set by special instructions is created. The bits can be set or cleared by the instructions shown in the start of this section[1]. All instruction words must be extended to include four bits for conditional execution, the different states that can be checked are shown in table 5.2.

In algorithm 5.22 two programs are shown: one which uses traditional branching to execute a set of instructions if some condition holds true(5.22(a)), and the same program implemented with conditional execution(5.22(b)). In the case where the branch is taken, the traditional code will execute the compare and jump instructions, inducing a pipeline flush. It will then execute the subroutine and return to the execution with another jump instruction inducing another pipeline flush. If the subroutine is sufficiently short(like in this example), the

1 The mnemonics are already in use in other instruction sets, and should probably be renamed to avoid confusion, but these are the names used in the instruction set simulator.

Table 5.2: Condition register and condition checks.

Condition Register T=[T3, T2, T1, T0]

Mnemonic	Condition
Tn	T[n] = 1
!Tn	T[n] = 0
Tn&Tm	T[n] = 1 && T[m] = 1
T	T = 0xF

Algorithm 5.22 Program with if-else type structure.

(a)Normal

```
void skipIf(void){
    cpsneJmp(R1, 0x00, jmp, subroutine);
    and(R2, 0x80);
    out(OUT, R2);
    next();}

void subroutine(void){
    in(R2);
    and(R2, 0x80);
    next();}

void next(void){ /* Operations */ }
```

(b)Conditional

```
void skipIf(void){
    cpi(T1, R1, 0x00);
    inc(R2, T1);
    andc(R2, 0x80, T1);
    andc(R2, 0x80, !T1);
    outc(OUT, R2, !T1);
    next();}

void next(void){ /* Operations */ }
```

execution time of the subroutine will be a lot higher due to the branch penalty. The conditional code, however, will execute the series of instructions without inducing any pipeline flushes. However, when the branch is not taken algorithm 5.22(a) will execute the compare instruction, skip the jump instruction, and will continue with the program. The conditional code will still load all the instructions in the subroutine, but skip all of them, and will execute the program using the exact same amount of cycles as when performing the subroutine. If we assume that the branch is taken in 50% of the cases, conditional execution is only preferable if the subroutine is so short that

$$1 + 2 \times \text{pipeline flush} < \text{subroutine length}. \qquad (3)$$

The calculation is further complicated by the fact that the branch may be taken in any percentage of the cases. As example from the UART protocol we know that TX-stop will be performed once for every eight transmissions. Thus, the subroutine must be an eighth of the length of the jump penalties, resulting in a length of less than one instruction.

The hardware implementation of conditional execution can be constructed by forcing a NOP-instruction to be performed whenever a *condition true* signal is low. The *condition true* signal will be set by a decoder circuit which, based on the state in the *condition register* and the *condition bits* in the instruction word, outputs a single true/false bit. A simple circuit with this functionality is shown in figure 5.10. The NOP functionality needed for conditional execution can probably be merged with the *compare skip if equal* functionality of this instruction set.

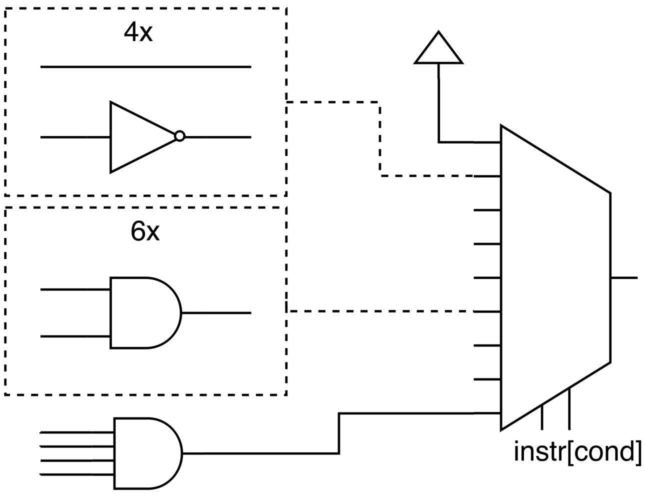

Figure 5.10: Illustration of conditional execution implementation.

5.6.1 PROTOCOLS

A set of conditional instructions was added to the instruction set. In the new set each I-function has an extra argument, the conditional, and will execute a NOP if the conditional is not true. A 4-bit register that contains the conditional flags was added, and a set of functions(set, clr, compare set if equal CPIE, compare set if not equal CPINE) was added to control the flags in the register. A typical use-case for the conditional execution is shown in algorithm 5.22(b).

SPI MASTER

The SPI implementation was not changed at all because no openings for improvement was present. The SPI program shown in algorithm 5.14 has no compare and jump combinations, and thus it cannot benefit from this type of conditional execution.

I2C MASTER

In I2C the address phase is followed by either the receive phase or the transmit phase, and has a possibility for using conditional execution. But, if one assumes an equal distribution of branch taken/not taken, conditional execution is only effective if the conditional code incurs a lower penalty if not executed than the compare-jump combination. Conditional execution was implemented, but in this case both the receive and transmit programs are quite long and the gain from removing the jump-penalty is completely absorbed by the amount of NOP-s produced by the, not executed, conditional I-functions. The conditional I2C-implementation is not included in the final results.

UART

Conditional execution was mainly added to improve the UART program. Several different implementations has been tried, without Repeat, with Repeat, with a large amount of the I-functions running conditionally and with very few running conditionally. The TX-phase and RX-phase are not being performed at the same time, and there is no gain in trying to make these conditional in relation to each other. The start and stop operations are targets for using conditional execution. However, start and stop are only executed once for every byte sent. With a large amount of the time being spent either idling or sending data-bits, the NOP-s incurred by the non-executed conditional instructions outweigh the gain of reducing jumps. In the end none of the implementations improved the cycle performance of the UART-protocol.

5.6.2 RESULTS

The simulation results of the conditional execution instruction set show that no improvement was achieved; the number of cycles is in fact increased. For SPI and I2C no opportunities for improvement was found and the implementation are the same as for the SOL and REPEAT instruction set, so the detrimental effects are solely in the UART program. In figure 5.11 one can see that the number of jumps have in fact increased in the attempt to utilize conditional execution. This may be due to my inability to optimize the UART program, and is perhaps a case of over-use of the conditional execution functionality. But many solutions have been tried, and this is the best result that was achieved without excluding conditional execution altogether.

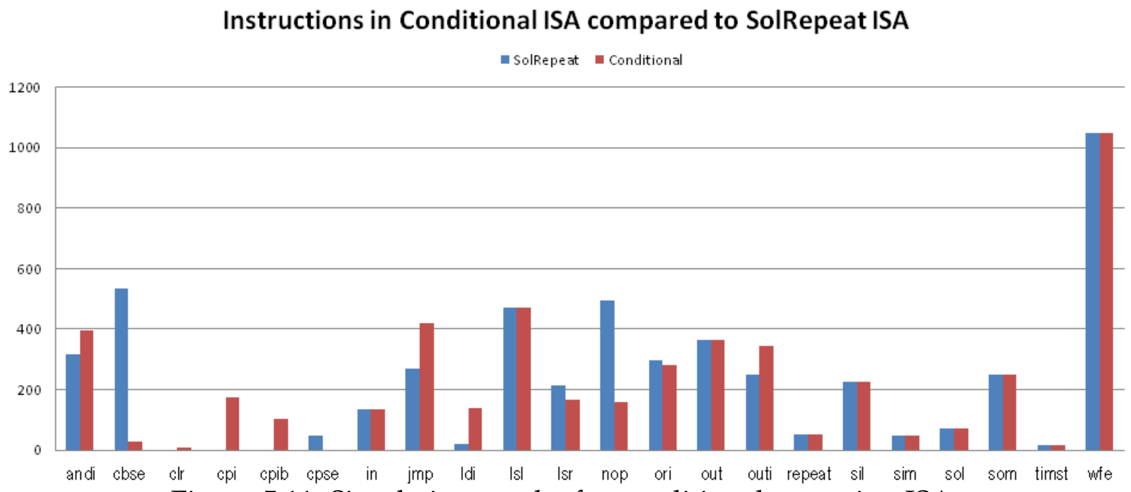

Figure 5.11: Simulation results for conditional execution ISA.

5.7 PARALLEL EXECUTION

The UART-program is always in one of four states and a significant part of the overhead of the UART-program is spent jumping between these states. In a final attempt to reduce the overhead the conditional execution instruction set was extended into a parallel execution instruction set. In the simulator the instruction set was created to support four instructions, but only three parallel instructions are necessary.

This instruction set has the ability to load a finite number of instructions in parallel, but only choose one instruction that will be executed. An illustration of such a program flow is shown in figure 5.12 and this allows switching between the states of UART to be performed by changing the conditional registers. One important thing to note is that all operations that are performed in parallel must have orthogonal conditions. That is, the intention of the program must be to always execute only one of the instructions because no form of scheduling or stalling is intended in this instruction set. In figure 5.12 the four different states of the UART is shown as rounded boxes and the lines extending vertically from the boxes can be considered as a timeline. The horizontal arrows from one timeline to another indicates a change in states by changing the conditional register. There is no pipeline flush incurred when changing state. In the different timelines different operations are performed, but the assembly code must be carefully designed so that whenever the state is changed the parallel operations are running in the same phase of the program. That is, that a state change does not result in any unwanted operations.

Figure 5.13 shows a concept draft of the hardware implementation for three parallel instructions. There is no need for actual parallel execution but three instructions must be fetched and none or one of them should be executed. This can be achieved by a condition checker (not unlike the one in figure 5.10) controlling a 4-to-1 multiplexer where one of the instructions is a NOP. Although this implementation is significantly smaller than having true parallel execution, it also introduces some limitations.

Because the conditional is checked in the fetch stage, some circuit for forwarding the conditional checks performed in the execution stage must be added. This was discovered too late in the project to implement in the power estimations. But as will be seen from the results of this section and section 6.3.5, it does not change the final conclusion.

This implementation implies the use of an instruction word that is 3 $\times$ *instruction word length* long. This results in a large bus width to the instruction memory or an instruction-fetch module that is three times faster than the rest of the processor. If the parallel execution functionality is always on, the size of programs that are not able to utilize the parallelism will be three times larger than necessary. So the processor should be configurable to turn off parallel execution if it is not needed.

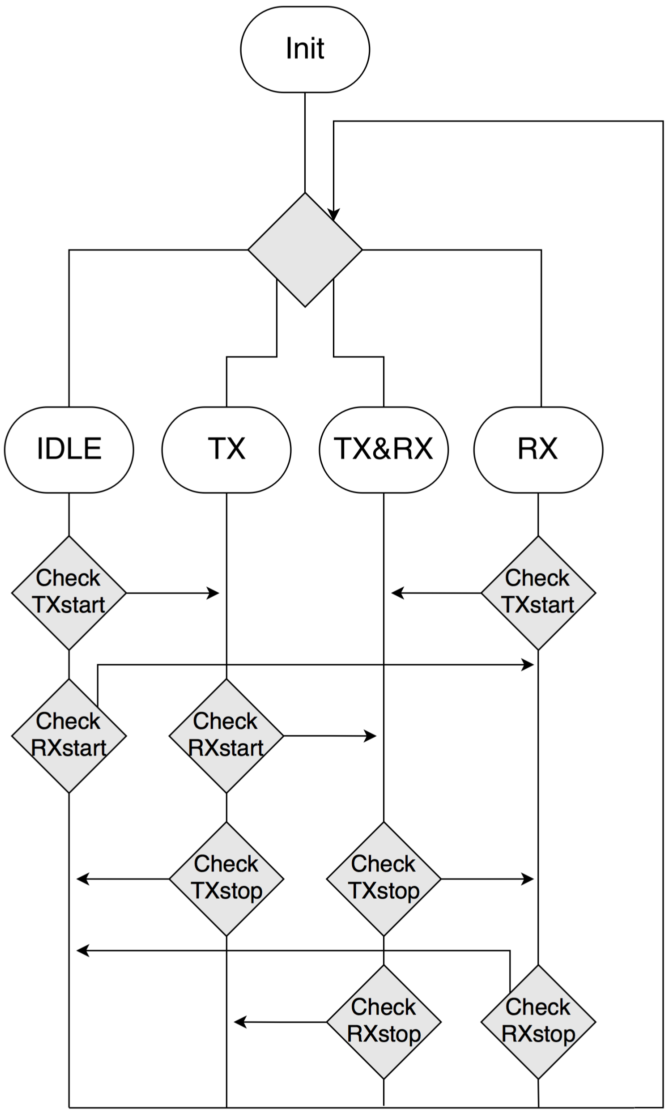

Figure 5.12: Illustration of the UART program in the parallel execution instruction set. Operations shown as horizontal arrows does not incur a pipeline flush.

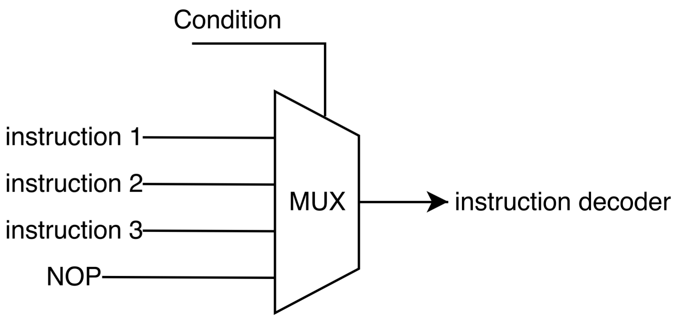

Figure 5.13: Illustration of parallel execution implementation.

5.7.1 PROTOCOLS

Parallel execution has been implemented in the simulator as a function that can take up to four different function pointers, the arguments to these functions and a Boolean as the condition for execution. The function is shown in algorithm 5.23. It prioritizes the functions from first to last and executes the first function that has a true conditional, if no conditions are true it executes a NOP. In the other instruction sets the functions take a varying amount of arguments, e.g. OUT takes only two arguments while AND takes three. To allow *parallelEx* to call all instructions, some I-functions have been extended from two to three arguments by including dummy-inputs.

Algorithm 5.23 Parallel Execution Function (shortened for readability)

```
int parallelEx(
void (*ex0)(unsigned int,unsigned int,unsigned int),
    unsigned int ex0var0, unsigned int ex0var1, unsigned int ex0var2,
    unsigned int ex0Cond
,void (*ex1)(unsigned int, ...
,void (*ex2)(unsigned int, ...
,void (*ex3)(unsigned int, ...
reportParallel();
if(ex0Cond)
  ex0(ex0var0, ex0var1, ex0var2);
else if(ex1Cond)
  ex1(ex1var0, ex1var1, ex1var2);
else if(ex2Cond)
  ex2(ex2var0, ex2var1, ex2var2);
else if(ex3Cond)
  ex3(ex3var0, ex3var1, ex3var2);
else
  nop();
return 0;}
```

SPI MASTER

For the same reasons as for the conditional execution instruction set, SPI has not been changed at all.

I2C MASTER

The I2C algorithm has three different types of phases it can run. Namely the address transmit, data transmit and data receive. All three phases are 9-bits long, and an A-program that run these phases in parallel has been created. An excerpt of the I2C parallel execution code is shown in algorithm 5.24. The three threads are loaded in parallel, but only one is executed. Note that the address thread and the send thread are very similar, while the receive thread needs to be padded with NOP-s to execute as many cycles as the other two threads.

Algorithm 5.24 I2C parallel execution A-program

```
void i2cParallelBit(void){
   // ex0 send Address Bit,   T0
   // ex1 send Bit,           T1&!T0
   // ex2 receive Bit,        !T1&!T0
   /***** Excerpt *****/
   parallelEx(
      somPar, OE_N, SDA, R2, T0,          //Output address bit
      somPar, OE_N, SDA, R2, T1&&!T0,     //Output data Bit
      outiPar, OE_N, 0xFF, NAN, !T1&&!T0, //Toggle SCL
      nopPar,0,0,0,0);
  parallelEx(
      wfePar, NAN, NAN, NAN, T0,          //Wait
      wfePar, NAN, NAN, NAN, T1&&!T0,     //Wait
      lslPar, RX_REG, 1, NAN, !T1&&!T0,   //Left shift RXREG
      nopPar,0,0,0,0);
  parallelEx(
      lslPar, TX_REG, 1, NAN, T0,         //Left shift TXREG
      lslPar, TX_REG, 1, NAN, T1&&!T0,    //Left shift TXREG
      nopPar,0,0,0,0,                     //NOP
      nopPar,0,0,0,0);
   /***** Excerpt *****/  }
```

Although it is entirely possible to run I2C in this way, I2C does not regularly switch between these phases and the gain from parallel execution is limited. A few added NOP-s due to the unevenness of the transmit- and receive-programs, and the added control instructions to change the conditional makes this A-program less effective than the SolRepeat A-program. Granted, I2C always switches from address to either transmit or receive, and it is conceivable that running address without parallel execution before continuing to a parallel data phase could improve the performance. But the address phase actually runs next to the transmit phase without extra cost, and the larger problem is to run transmit in parallel with receive.

UART

UART has the greatest opportunity to benefit from parallel execution. In algorithm 5.25 the A-program runs *check start RX* and *running RX* in parallel. In the program it can be seen that there is no need for any jumps to switch between the two states, and in fact the A-program as a whole(including TX) contains only the final jump back to the top (line 26 in algorithm 5.25). This significantly reduces the number of pipeline flushes. However, whenever the RX-protocol is in the idle state(i.e. neither running nor starting) there are two NOP-s performed each time this program is executed. With the assumption that a significant amount of time will be spent in an idle-state this incurs a significant penalty to the performance. The same is the case for the TX-program and this effectively negates the gain from running without jumps.

Algorithm 5.25 UART RX parallel execution A-program

```
void uartRx(void){
  //ex0 = check start,  T0 == starting
  //ex1 = running,      T1 == running
    wfe();                           //Read middle
    parallelEx(
        inPar,R1, NAN, NAN, !T1,     //Read data
        lsrPar, RX_REG, 1, NAN, T1,  //Shift RX_REG
        nopPar,0,0,0,0,
        nopPar,0,0,0,0);
    parallelEx(
        cpiePar, rT0, R1, 0<<RXD, !T1,  //If RXD was high
        simPar , RXD, R0, NAN, T1,   //Read RXD
        nopPar,0,0,0,0,
        nopPar,0,0,0,0);
    parallelEx(
        ldiPar, R3, 0x80, NAN, T0,   //Initiate counter
        lsrPar, R3, 1, NAN, T1,      //Shift Counter
        nopPar,0,0,0,0,              //NOP if not starting or running
        nopPar,0,0,0,0);
    parallelEx(
        setPar, rT1, NAN, NAN, T0,   //Set running high
        cpinePar,rT1, R3, 0x00, T1,  //If counter=0, stop
        nopPar,0,0,0,0,              //NOP if not starting or running
        nopPar,0,0,0,0);
    clrc(rT0, T0);                   //Clear start
    jmpDummy("uartRun");}            //Jump to top
```

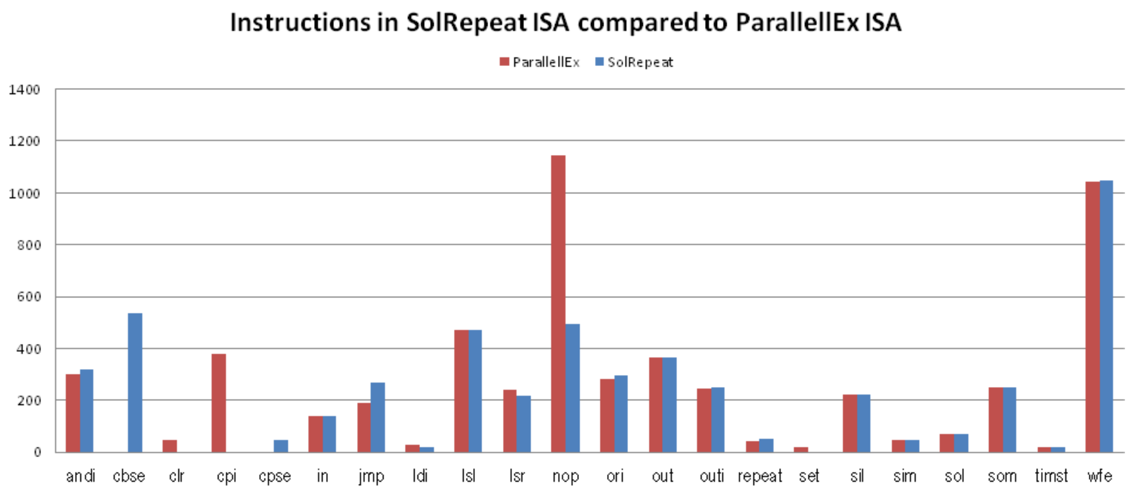

Figure 5.14: Simulation results for parallel execution ISA.

5.7.2 RESULTS

Again, the simulation results(figure 5.14) show that the new A-program implementations did not achieve any improvement compared to the *SolRepeat* instruction set. Although the number of jumps—and consequently pipeline flushes—for UART is decreased, the increased number of compare instructions and NOP-s more than cancel out the improvement. The NOP-s stem from the fact that the instruction sequences that we wish to run in parallel are not necessarily of equal length and introduces NOP-s on the shorter sequence.

5.8 COMPARISON

From figure 5.16 it can be seen that REPEAT very effectively reduced the number of pipeline flushes. The pipeline flushes was further reduced by both conditional and parallel execution, but the total running time was increased for both of these instruction sets compared to SolRepeat.

When comparing the results in figure 5.16 and 5.15 it can clearly be seen that the most promising instruction set architecture is the *SolRepeat* ISA. There is a 36% reduction in running time for *SolRepeat* as compared to the minimal instruction set(*Std*). A change in bitrate only affects the sleep time of the program, and 36% reduction is valid for all bitrates that does not violate the time constraints described in section 4.4. The REPEAT-instruction will add considerable hardware to the processor, so if the improvement will indeed be that large when power consumption is considered remains to be seen.

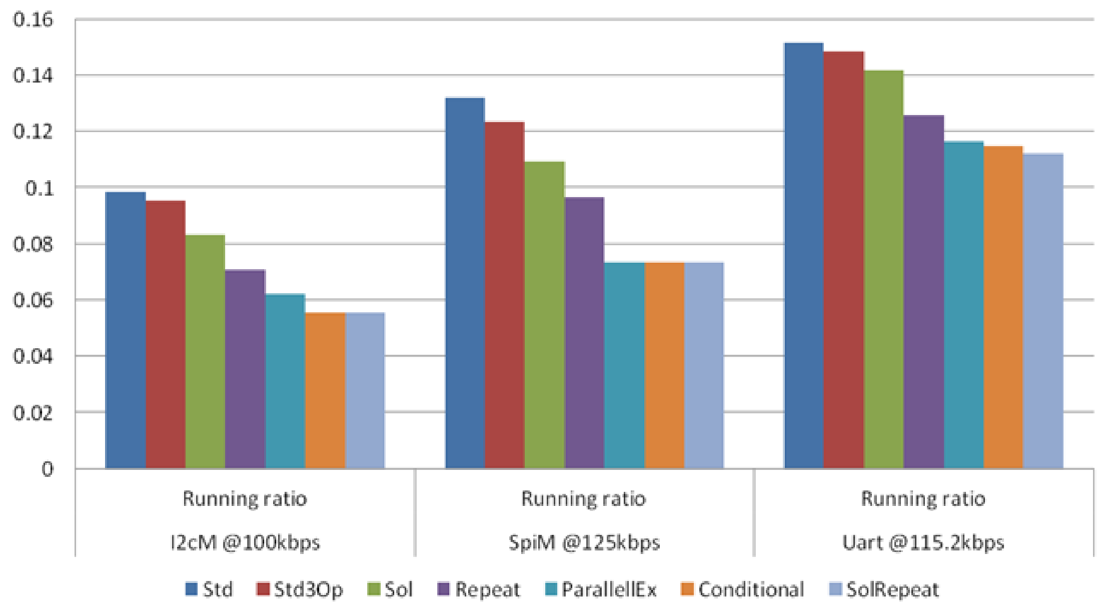

Figure 5.15: Running ratio for all instruction sets divided by protocol.

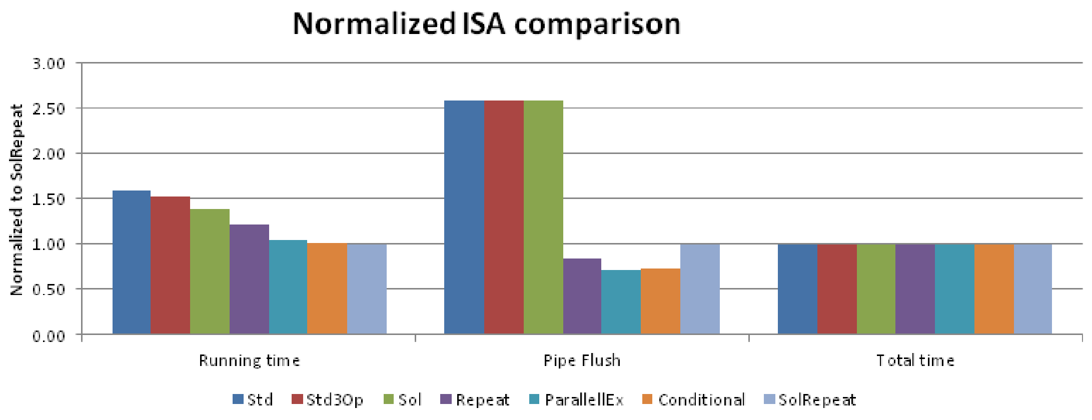

Figure 5.16: Running and total time of benchmark on all instruction sets, normalized to SolRepeat.

5.9 MAX SPEEDS

The maximum bit-rates for *SolRepeat* is shown in table 5.3, these values have been found by simulation. The bit-rate maximum speed is limited by the processor frequency and the longest sequence of instructions between two wait for event(WFE) instructions.

Table 5.3: Maximum bitrates for SolRepeat ISA.

	SPI	I2C	UART
Bitrate[kbps]	1152.1	806.5	448.0

6 POWER

In this chapter the dynamic power consumption of each instruction set from chapter 5 will be calculated. To do this a power estimation method based on an existing processor will be presented. The area-estimates are

6.1 POWER ESTIMATION METHOD

In section 2.8 an analytical, complexity based, power estimation method was presented. It proposes that there is a relationship between the complexity (i.e. the number of gate equivalents) and power. The equation for this method is repeated in equation 4 for convenience.

$$P_{mod} = N(P_{typ} + C_L * V^2) * \alpha * f \tag{4}$$

It assumes that an average load capacitance and activity factor can be estimated. In this chapter a method based on these principles is used. The power for a similarly structured component, namely a processor, has been measured. Based on the assumption that the power consumption of a component is proportional to its number of gate equivalents the power per module can be calculated as:

$$P_{mod} = N * P_{gate} * f \tag{5}$$

Where

$$P_{gate} = (P_{typ} + C_L * V^2) * \alpha \tag{6}$$

and P_{gate} can be estimated as:

$$P_{gate} = \frac{P_{measured}}{N_{gates}} \tag{7}$$

This estimation method is common practice in the industry[25]. However, this method of approximation does not necessarily yield an accurate result of the power consumption for all modules, but for modules with similar activity profiles (e.g. two processors) the results should provide a reasonable fidelity[1]. Assuming that the fidelity is high, correct architectural decisions can be made on the basis of these estimates.

According to measurements performed at Nordic Semiconductor the Cortex M0 consumes 567µA when running at 16MHz, and the leakage current is 0.2µA.

1 Fidelity describes to which degree the estimator is able to predict if one solution is better/worse than the other solutions. This is in contrast to accuracy which is a measure of how close the estimator is to the actual value. Fidelity can be high even with low accuracy.

The number of gate equivalents in the Cortex M0 is estimated based on area-results after synthesis. It is calculated as how many NAND2-gates one can fit on the area that the design requires. The parameters for the Cortex M0 implementation is shown in table 6.1.

Note that the power consumption is directly proportional to the current by a factor of 1.2(the voltage), and throughout this chapter the power consumption will be discussed although most estimated values will be given as currents.

Table 6.1: Cortex M0 area and dynamic power consumption.

Area	NAND2	Activity	Active Current	Voltage
228357μm^2	22881	21%	567μA	1.2V

6.2 THE CORTEX M0 AS A BASELINE

The Cortex M0 has been chosen as a baseline for power estimations. This is primarily because power measurements of the processor was readily available, but the Cortex M0 is not a bad candidate for a baseline in any case. The Cortex M0 is a low power, 32-bit, processor created for the embedded market. It uses the ARMv6-M Thumb instruction set which is more extensive in some respects than the instruction sets planned in this thesis.

The data width is perhaps the most obvious difference between the cortex M0 and the planned processor. The 32-bit Cortex M0 datapath is four times larger than in the 8-bit planned processor. Furthermore, the Cortex M0 has 13 general purpose register compared to the planned four general purpose registers in the BBP. It also implements memory operations, addition/subtraction and single cycle multiplication; these instructions has not been considered necessary in the BBP. The Cortex M0 does not allow for single cycle I/O access, but its close sibling Cortex M0+ has this functionality. Unfortunately, there were no measurement data available for this processor.

In general the Cortex M0 can be viewed as a superset of the minimal instruction set. It is therefore expected that the estimates performed based on the Cortex M0 will overestimate the number of gates needed for BBP by a large amount. As a result, the impact of adding new functionality may be improperly weighted. Adding 1000 gates to the Cortex M0 will constitute about a 5% increase in hardware, while 1000 gates in the BBP may be a significantly larger relative increase. If the addition of an instruction reduces the time to idle by 10% while increasing the hardware size of the Cortex M0 with 5%, it will add up to a total decrease in energy for the application This may not be the case for the BPP, where the area increase will be a larger relative cost.

In the following sections some efforts will be made to reduce the area gap between the expected BBP hardware and the Cortex M0. One could argue that it would be simpler to estimate the size of the BBP and use the power per gate estimate from the Cortex M0 to calculate the power consumption. However, it was considered likely that when estimating the BBP-size one could easily overlook important factors of the processor and underestimate the size of the BBP. Consequently it was expected that the method chosen would represent a more accurate power estimation. Throughout this chapter all estimations assumes that it is better to err on the side of caution, and that it is better to underestimate the performance of the BBP than the opposite.

6.3 POWER ESTIMATE PER FUNCTIONALITY

To be able to compare the instruction sets it is necessary to be able to approximate the power usage of each functionality so that they can be removed and added for the different instruction sets. A power estimate of the unused functionality in the Cortex M0 will also be approximated in order to improve the accuracy and fidelity of the estimates. In this section the size requirements for different instructions/functions will be analysed, and a possible hardware implementation will be described and constructed. The total size requirement will be calculated from the size of each added cell, and finally the total area will be reduced to NAND2 equivalents for comparison. The area numbers for the Cortex M0, as well as the following sections, is based on a subset of the TSMC 0.18μm Process at 1.2V. The list of cells in the subset can be found in appendix A, and the cell descriptions in [26].

6.3.1 UNUSED FUNCTIONS

As previously mentioned the processor described in chapter 3 is a lot smaller than the Cortex M0 as it does not need to be as general. To be able to compare the processor to the dedicated hardware modules, the power consumption of some key functions will be calculated and subtracted from the total power.

DEBUG

The Cortex M0 has a debug functionality that occupies 3825 gate equivalents. The debug-functionality is clock-gated and will not consume any dynamic power when not enabled. During the measurements performed at Nordic Semiconductor the debug functionality has not been enabled and the gates can be removed from the total number of gates prior to determining the power per gate.

REGISTERS

The Cortex M0 has thirteen general purpose registers and three special registers. Namely the stack pointer, the link register and the program counter[27]. The protocol implementations in chapter 5 indicate that only four general purpose registers, two special TX/RX registers and the program counter are necessary

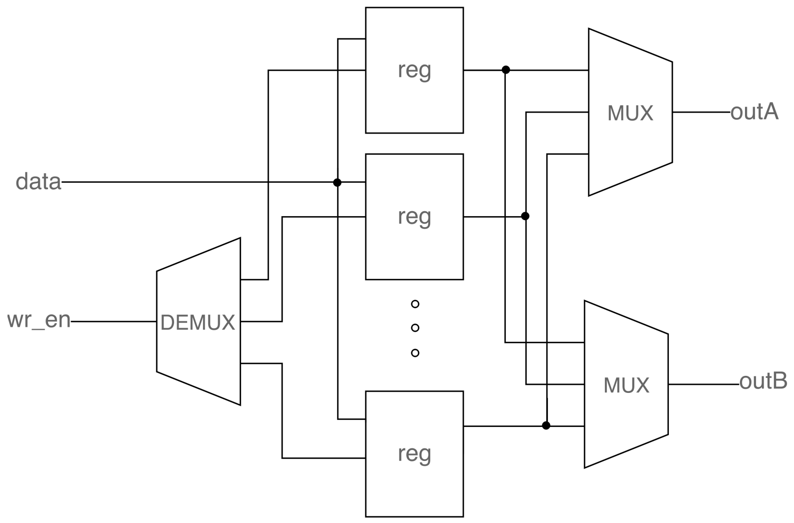

Figure 6.1: Simple register file model.

in the BBP. The stack pointer is unnecessary because it is used when passing more than four arguments to a subroutine call. The BBP does not have enough general purpose registers to pass more than four arguments. Creating subroutines that can be called from any part in the code and is expected to return to the same location(i.e. branch linked operations) is inefficient for short subroutines, and is not used in the protocol implementations. Thus, the link register is rendered unnecessary. These two assumptions reduces the BBP's ability to operate as a general processor.

The general purpose register file in the Cortex M0 represent 28.5% of the total area for the chip according to the synthesis results from tests at Nordic Semiconductor. Additionally, there are several multiplexers distributed on the chip to control the write inputs of the register file. The register file also has separate clocks and write enable signals for the program counter and stack pointer, but the logic of these will not be considered. A simple register file model, as shown in figure 6.1, will be used to calculate the reduction in area. For simplicity the program counter register will be left out of the subsequent discussions.

While the program counter is updated every cycle, only one of the registers is updated within a cycle. Thus, it can be assumed that the activity factor in the registers will be a fifteenth of the average activity, an activity factor of 1.40%. One 32-bit register can be constructed using 32 SDFFR-cells, giving a total of 480 SDFFR-cells. Six 8-bit registers can be created using 48 SDFFR-cells. Additionally the register file uses two 32-bit 15-to-1 multiplexers to feed the value of two registers to the ALU, and one 1-bit 1-to-15 demultiplexer to distribute the write enable signal to the fifteen registers. The size of both the 32×15 register file and the 8×6 register file and the total reduction in current is shown in table 6.2. Note that the number of registers should perhaps be a power of two to fully utilize the instruction word, but considering that the program counter has been left out of these discussions this is not taken into account as the program counter can be expected to be larger than an 8-bit register. A smaller register file implies a shorter data-word which will reduce power in the fetch and decode stages. The reduced power in the fetch stage will not be calculated and an overestimation of the BBP-size is expected.

Table 6.2: Area and power estimates of register reduction.

		Cell	Area[μm^2]	NAND2	Activity	Current[nA]	№ of
32-bit	Register	SDFF	2980.45	298.67	1.40 %	592.70	15
	Mux	MX2	851.56	85.33	21 %	2540.13	28
1-to-16	Demux	Gate	345.95	34.67	21 %	1031.93	1
8-bit	Register	SDFF	745.11	74.67	1.40 %	148.17	6
	Mux	MX2	212.89	21.33	21 %	635.03	10
1-to-8	Demux	Gate	153.01	15.33	21 %	456.43	1
Total reduction			**62143.80**	**6227.33**		**73350.32**	

ADDER–SUBTRACTOR

The Cortex M0 adder is not needed for the implemented protocols. Although the particulars of the on-chip implementation is not known, one can assume that the adder is constructed using 2-bit full-adders in a ripple-carry configuration. This is one of the smallest possible implementations of an adder[28], a total of 32 full-adders. For subtraction the same adder is used with one operand inverted *carry in* on the *least significant* full-adder set high, 32 XOR-gates are needed. The Cortex M0 does not use the same XOR-gates for operand negation and the EOR-instruction. There are two full adders available in the cell-library, normal and high speed, the smallest possible implementation is assumed and the ADDFX2-cell is chosen. The size and power consumption for the adder–subtractor is shown in table 6.3.

Table 6.3: Area and power estimates of addition/subtraction functionality.

	Cell	Area[μm^2]	NAND2	Current[nA]	№ of
Full-adder	ADDFX	69.85	7.00	208.37	32
XOR	XOR2	26.61	2.67	79.38	32
	TOTAL	**3086.90**	**309.33**	**9207.98**	

DATAPATH WIDTH

The datapath width in the Cortex M0 is four times larger than in the BBP. Estimating the size of the datapath can be complicated because the datapath is not a single module, but is instead incorporated into almost every module in the design. The gates in the datapath will mainly consist of the *Arithmetic Logic Unit*(ALU), the *Shift and permute unit*(SPU) and multiplexers to pass the data through the processor, in addition to the already considered register file[2]. The ALU size is 1018 gate equivalents and the SPU size is 644 gate equivalents according to synthesis results at Nordic Semiconductor. These modules are not simple to analyse and with the rest of the datapath distributed across the processor some rough assumptions are made to simplify the calculations. Although the control-path of the modules is assumed to comprise a large part of the module, the SPU's size is divided by four. It is assumed that not altering the size of the ALU and not estimating the distributed data path will more than make up for the underestimation of the SPU's size. A total of 483 gate equivalents is removed from the design to allow for the narrower datapath of the BBP, it is shown in table 6.4. For comparison a 4-to-1 32-bit multiplexer is calculated to 426 gate equivalents, so this is considered a very conservative estimate.

Table 6.4: Area and power estimates of datapath width reduction.

	NAND2	Current[nA]
Datapath	483	14378

MULTIPLICATION

The Cortex M0 implements single cycle multiplication. Results from synthesis tests performed at Nordic Semiconductor show that the multiplier uses a total of 2587 gate equivalents. The multiplier is not a part of the BBP and the estimated power and gates used by the multiplier can be removed. However, the multiplier has an enable signal that will tie all inputs to logic low if the multiplier is not enabled. The switching activity factor on the internal gates will be zero when the multiplier is not enabled. The power measurements used for the Cortex M0 in this thesis is based on a program calculating prime numbers. The prime number calculation algorithm uses the multiplier often and the switching activity in the multiplier when operating is usually high depending on the input. The multipliers power consumption is, for simplicity, assumed to add its full gate equivalent power to the measured power. The NAND2 equivalents and current consumption is shown in table 6.5.

Table 6.5: Area and power estimates of multiplier.

	NAND2	Current[nA]
Multiplier	2587	77008

DATA MEMORY

Memory is accessed using the AHB-interface, this bus is used for both instruction and data memory as well as peripheral control and all other communication. Because the BBP is designed as any other peripheral module it will not have an AHB-master interface, but will instead have a dedicated instruction memory and *Direct Memory Access*-module. It is assumed that the logic required for the AHB-interface can act as substitute for these modules.

6.3.2 REPEAT

Adding the repeat instruction involves adding a loop controller that should be connected to the program counter logic. A possible hardware implementation for the repeat instruction is shown in figure 5.4. For the protocols implemented in this thesis performing eight loops of eight instructions is sufficient. But some communication protocols may require larger loops with more instructions, for example an I2S protocol sending music with 16-bit precision[29]. With this in mind a 16X16 repeat instruction is suggested here. The two decrement adders can be implemented using TSMC standard cell full-adders. A 16 bit decrementer uses 16 full-adders in a ripple-carry configuration with the with the B input tied to logic high, thus all A-values are added to two's complement negative one(0xFFFF). The area, number of NAND2 equivalents and estimated power can be seen in table 6.6. A zero-checker is simply a 16-input OR-gate. Using the available TSMC library this can be constructed using six 3-input and three 2-input OR-gates.

Table 6.6: Area and power estimates of Repeat functionality.

	Cell	Area[μm^2]	NAND2	Current[nA]	№ of
Decrement	ADDF2	1087.73	109.00	3244.62	2
Zero-check	OR3/2	146.36	14.67	436.59	2
4-bit reg	SDFFR	372.56	37.33	1111.31	3
31-bit reg	SDFFR	2887.32	289.33	8612.64	1
2-to1 MUX	MUX2	26.61	2.67	79.38	112
Or	OR2	13.31	1.33	39.69	1
	TOTAL	**9466.93**	**948.67**	**28239.14**	

Although the decrement circuit uses two's complement addition to decrement, it is not necessary to store the values with five bits. The fifth bit that is needed to represent 16 positive values in two's complement form is implicitly removed(i.e. not connected) at the end of decrement calculations. The four registers can be implemented using D-flipflops. The flip-flops available in the Nordic Semiconductor subset of TSMC standard cells are scan-flip-flops for scan based testing. Four 4-bit registers, 16 SDFFR cells in total. Additionally, the initial instruction memory pointer must be stored so that it is possible to return to this address. The required size of the register depend on the address-space of the instruction memory. The address space in the Cortex M0 is 31-bits, thus another 31 SDFFR-cells must be added in addition to a 31-bit wide 2-to-1 multiplexer. Furthermore, five 16-bits wide multiplexers are used in the loop controller. In total 112 2-to-1 multiplexers. The total number of gates needed to implement the REPEAT-instruction is 955 gates, a significant addition.

6.3.3 SOL

As described in section 5.4 the SOL-instructions adds five multiplexers in total. The total added area and power is shown in table 6.7.

Table 6.7: Area and power estimates of SOL functionality.

	Cell	Area[μm^2]	NAND2	Current[nA]	№ of
2-to-1 mux	MX2	26.61	2.67	79.38	5
	TOTAL	**133.06**	**13.33**	**396.90**	

6.3.4 CONDITIONALS

As described in section 5.6 conditional execution can be implemented as an enable signal for all signals that enable changes in the states of registers, conditional flags and I/O. The logic proposed to compute conditional execution contains four inverters, six AND-gates, a four input AND and the 16-to-1 multiplexer, see figure 5.10.

In the subset of the TSMC library 2-to-1 multiplexer standard cells are available. The 16-to-1 multiplexer can be constructed by using 8+4+2+1=15 2-to-1 multiplexers, or 16 4-input AND-gates, 16 inverters and five 4-input OR-gates [30]. As shown in table 6.8, the multiplexer approach yields an area of 40 NAND2 equivalents, while the gate approach gives a total of 51 NAND2 equivalents. Using the multiplexer approach, conditional execution adds 45.33 NAND2 equivalent gates in total.

Considering the total size of the M0 this is not a very large amount, but perhaps the most important impact conditional execution has on the system is the bits it occupies in the instruction word. 4-bits of the instruction word is used for conditional coding and assuming that the current instruction words are irredundant this adds four bits to the instruction memory and the fetch logic in theory. Because the simulation results of the conditional instruction set show no improvement the increased power consumption of a larger instruction word is not calculated due to the complexity of the task.

Table 6.8: Area and power estimates of Conditional functionality.

	Cell	Area[μm^2]	NAND2	Current[nA]	№ of	Included
16-to-1 mux	MX2	399.17	40	1190.69	1	✓
	Gates	508.94	51	1518.13	1	×
And2 gate	AND2	13.31	1.33	39.69	11	✓
Inverter	INV	6.65	0.67	19.84	4	✓
And4 gate	AND4	19.9584	2	59.53437	1	✓
	TOTAL	**525.57**	**52.67**	**1567.74**		

6.3.5 PARALLEL EXECUTION

Adding parallel execution includes adding the conditional calculation logic described in the section above and a 16-bit wide 4-to-1 multiplexer, the estimated values are shown in table6.9. Additionally the width or speed of the instruction memory bus must be tripled, and this will constitute the bulk of the added logic. Calculating the power usage of this circuit is complex, and because the parallel execution shows no improvement in number of cycles executed, the cost of implementing is not calculated and the power consumption of this instruction set is expected to be underestimated.

Table 6.9: Area and power estimates of Parallel Execution.

	Cell	Area[μm^2]	NAND2	Current[nA]	№ of
Conditional		525.57	52.67	1567.74	1
4-to-1 mux	MX	79.83	8	238.14	16
	TOTAL	**1802.91**	**180.67**	**5377.94**	

6.4 DEDICATED MODULES

Although some power-measurements of the dedicated modules used in Nordic Semiconductors microcontrollers exists, the same power estimation method will be used to estimate the power consumption for the dedicated modules. Because the power estimates are not necessarily accurate, comparing measured values for the hardware implementations to the estimates may give misleading conclusions. Furthermore, the measurements included IO-switching power, baud-rate generator and DMA, and the power measurements were significantly higher than the estimates. The estimates for the dedicated modules can be seen in table 6.10. Most of the logic in the dedicated hardware modules run at a 16Mhz clock [31], while a small part runs at the bus speed. In the estimations it is assumed that the lower effect of the slower running part of the modules make no significant impact on the total power. The modules are therefore assumed to consume a constant amount of power for all bit-rates.

Table 6.10: Power estimates of dedicated hardware modules.

Protocol	NAND2	Current[μA]
SPI	1335	39.75
I2C	2981	88.75
UART	1697	50.53

6.5 ESTIMATION RESULTS

First, the gate count of the Debug functionality is removed from the total number of gates in the Cortex M0 shown in table 6.1, the current per gate is then calculated as shown in equation 7. We have,

$$I_{gate} = \frac{567.26\mu A}{22881 - 3825} = 0.029767\mu A \tag{8}$$

Table 6.11 show the total number of gate equivalents and current for each functionality which is removed from the Cortex M0, and the final current of the reduced Cortex M0 which will be used for calculating the power of the minimal instruction set.

Table 6.11: Area and current estimates of unused Cortex M0 functionality, and reduced Cortex M0 core.

	Removed		Total	
	NAND2	Current[μA]	NAND2	Current[μA]
CortexM0			22881.45	567.26
Debug	3825.00	0.00		
Adder	309.33	9.21		
Register	6227.33	73.35		
Multiplier	2587.00	77.01		
Data Path	483.00	14.38		
Reduced M0			**9449.79**	**393.31**

In table 6.12 the dynamic current of every instruction set is shown. For simplicity it is assumed that the static power consumption is the same for every instruction set, the changes in static power consumption between the sets is minimal, and will not impact the estimates significantly.

Table 6.12: Area and current estimates of each instruction set architecture.

	NAND2	Current[μA]
Std	9449.79	393.31
Sol	9463.12	393.71
Repeat	10398.45	421.55
SolRepeat	10411.79	421.95
Conditional	10464.45	423.52
ParallelEx	10592.12	427.33

6.5.1 EQUIVALENT POWER

The dedicated hardware modules have a constant power consumption throughout operation. The processor, on the other hand, switches between active and sleep mode. To compare the two an equivalent constant power is calculated for the processor. The equivalent power is calculated as:

$$P_{eq} = P_{active} \times \lambda + P_{sleep} \times (1 - \lambda) \qquad (9)$$

where P_{active} is the power consumption when the processor is active, P_{sleep} is the power consumption when the processor is clock gated, the power is calculated as the current multiplied with the chip voltage, 1.2V. λ is the ratio of running time compared to the total time reported by the simulator. λ is calculated as:

$$\lambda = \frac{t_{active}}{t_{total}} \qquad (10)$$

7

COMBINED RESULTS

In this chapter the results from chapter 5 and 6 are combined to produce the equivalent power described in equation 10. The protocols are simulated at several bit-rates, and the active/sleep ratio is combined with the dynamic and static power to produce an equivalent power that can be compared to the dedicated hardware. The static power consumption is assumed to be 0.24μW.

7.1 INTER-INTEGRATED CIRCUIT – I2C

As can be seen in figure 7.1, the I2C the BBP performs surprisingly well. The *SolRepeat* instruction set is better than the hardware implementation for both I2C running frequencies; 100kbps and 400kbps. 500 and 800 kbps, are not typical I2C bitrates, but are simulated to show that it is possible to run I2C at these bitrates on the BBP. The good performance is probably because the I2C A-program does not implement the full functionality of I2C, both multi-master support and clock-stretching is not implemented. The dedicated hardware module(TWI) does not support multi-master mode [32], but it does support clock stretching. Assuming that clock stretching will only be performed on the byte-level, which is a reasonable assumption, every byte which is not stretched will take three more instructions to execute. This is a considerable addition, but does not entirely account for the good performance of the *SolRepeat* implementation. It can be assumed that the BBP will be a decent I2C replacement.

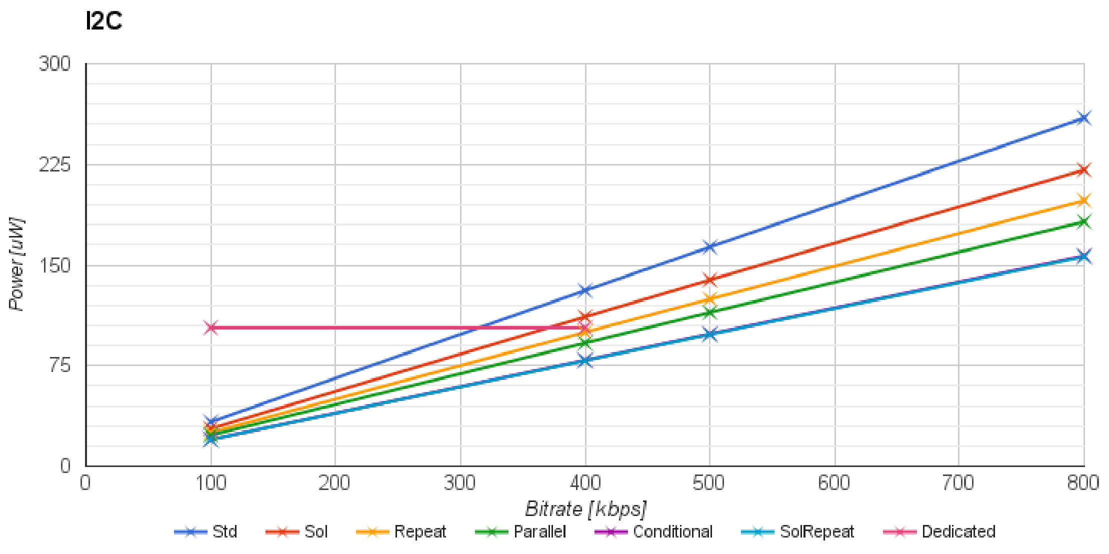

Figure 7.1: Power estimates for the I2C protocol divided by implementation.

7.2 UNIVERSAL ASYNCHRONOUS RECEIVER/ TRANSMITTER – UART

In figure 7.2 the estimation results for some of the possible baud-rates for UART are shown. UART is designed to run at 16 different frequencies, and a subset has been simulated. The baud rates simulated are: 9.6, 19.2, 76.8, 115.2, 250 and 403 kbps. 403 kbps is not a common UART baud-rate but it is the highest baud rate for which all instruction sets can run; Repeat is the limiting instruction set. For low bit-rates the BBP performs better than the hardware implementation for all ISA. However, the power consumption of the BBP increases linearly with the bit-rate and quickly becomes less power-effective. The *SolRepeat* instruction set has the lowest power consumption and performs better than the hardware implementation when the bit-rate is lower than approximately 140kbps. For bit-rates higher than this, all instruction sets performs worse than the hardware implementation. Compared to the running times the power estimations produce no surprising results. Note that, as discussed in section 6.3.5, the power estimate for the parallel execution instruction set is overly optimistic and will probably have higher power consumption than what is shown.

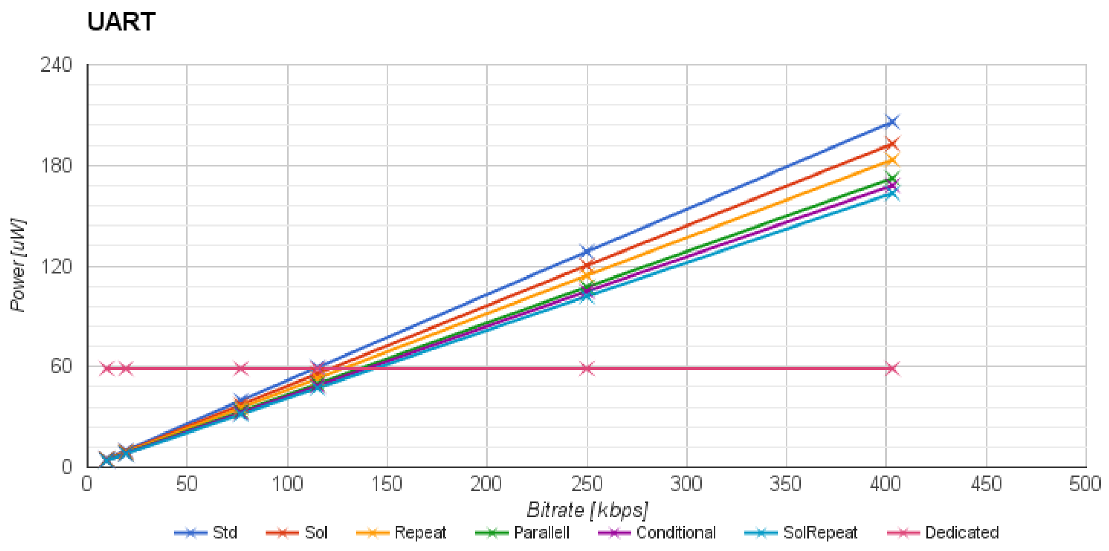

Figure 7.2: Power estimates for the UART protocol divided by implementation.

7.3 SERIAL PERIPHERAL INTERFACE – SPI

As one can see from figure 7.3 the BBP performs worse than the hardware implementation for almost all bit-rates. The bit-rates simulated are 125, 250, 500, 750 and 1000 kbps, all except 750kbps are bit-rates that the dedicated hardware module is designed to run at[32]. *Std* and *Sol* was not able to run at 1000kbps. Again, the *SolRepeat* instruction set is the best performing instruction set. It is slightly than the hardware implementation when running at 125kbps, but the difference is so small that it is beyond the expected accuracy of the estimations and one cannot conclude that the BBP is indeed better. Again the results after estimating the different power consumptions of the different instruction sets does not change the conclusions one could have drawn from the running time simulations alone. *Conditional*, *Parallel Execution* and *SolRepeat* performs almost equally well. This is because the SPI A-programs could not be improved by adding conditional execution, they therefore all run the same A-program. Parallel Execution probably performs worse than shown in this graph for the same reasons as mentioned in the previous section.

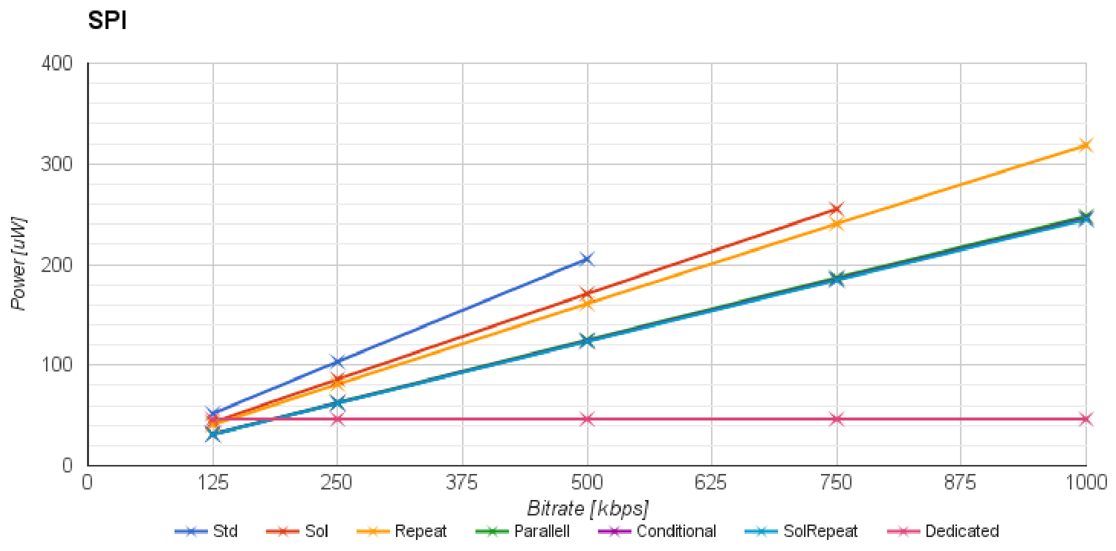

Figure 7.3: Power estimates for the SPI protocol divided by implementation.

8 DISCUSSION

For low bitrates the BBP show very promising results. The power consumption is on the level of the dedicated hardware modules and the added flexibility provides an overall improvement. As the bitrates increase, however, the processor spend more time in the active state relative to the sleep state for each bit it is sending. That is, the absolute power consumption of sending a bit does not increase(in fact it is slightly reduced due to less time spent in sleep mode), but the interval between bits is reduced. The dedicated modules does not have an equivalent to the sleep mode and will consume roughly the same amount of energy at any bitrate. Consequently, the performance of the BBP compared to the dedicated modules is significantly worse for higher bitrates.

Although the power consumption is not directly comparable to the dedicated modules, it was never expected to outperform the existing solution. The existing solution with dedicated modules require considerable engineering work to implement into a system. Firmware must be created for each module, and each module also takes it share of area on the chip. Adding a single BBP will add SPI, UART, I2C and, although yet untested, several more protocols to the chip. This simplifies implementation both at the hardware and the firmware level, and may represent a shorter time to market. Furthermore, production flaws may be easier to correct in a processor with a software workaround in the event that something is designed or produced incorrectly.

It also presents the opportunity to add uncommon or user designed protocols, which will usually not be present on a high-production-volume chip. Today these protocols might be bit-banged on the less energy efficient main processor, and the BBP can offer a significant reduction in power consumption. The BBP represent an opportunity to add a single module, next to the existing ones, that can solve any needs a user might have for an additional or specialized protocol. For example, the current Nordic Semiconductor chips are unable to run in a multi-master I2C configuration. But the BBP can be reprogrammed to allow for multi-master arbitration, and can energy-efficiently solve the problem. This flexibility is the strength of the BBP.

8.1 POWER ESTIMATION

The power estimations performed are very likely to be inaccurate. The inaccuracy of the complexity based estimation model can be very high, and because the area estimates does not take into account all the differences between the Cortex M0 and the BBP, it is expected that the accuracy can be even lower. When estimating the size of the BBP by removing gates from the Cortex M0 there are many structures that, for simplicity, has not been taken into consideration. The control network of the processor has not been reduced to mirror the reduction in functionality. This is because it would be hard to estimate due to the distributed nature of such a network. The same argument is used when not considering the implication of the reduction in the clock tree. The average power per gate has been estimated across the whole design, but the activity and load capacitance will differ for different chip-structures. This has only been taken into account when reducing the size of the register file, and should have been taken into account if the clock network was considered.

In order to create a trustable estimate, the size estimations try to be as close to, but not better than, the worst case scenario as possible. Steps have been taken to avoid underestimating the power consumption of the BBP. Specifically the Cortex M0 is a much larger and comprehensive processor than the BBP, and even though the Cortex M0 has been reduced in size to increase the accuracy and fidelity, the size reduction estimates has been as conservative as possible. I feel confident that the comparisons between the dedicated hardware and the bit-banging implementations favour the dedicated hardware, and that the BBP performs better than the results show. Although, and this can not be stated clearly enough, there is no direct proof to support this.

8.2 SLAVE IMPLEMENTATIONS

The slave counterparts of the SPI and I2C protocols have not been implemented in this thesis, and this is certainly an object for further work. The BBP performs well because it can quickly go into sleep mode whenever no transfers are required. As we've seen in the UART protocol, the BBP is not as well suited for slave implementations as it is for pure master protocols. This is due to the fact that the BBP must wake to check if a transfer has been initiated by some other source. The synchronous slave protocols are also slightly different because they need to run on an external clock, this complicates the implementation. For SPI a slave is selected using the chip select, CSn. In order to detect that a transfer is beginning the BBP needs to checks this signal, but at what interval should the BBP wake up to check this? Since the master controls the bus-clock, it is not necessarily known for the slave. There is generally not possible to know when a transfer might begin, and the BBP might have to run in busy-wait loop, constantly checking the chip select signal. It is therefore necessary to implement some detecting module that can wake the BBP upon assertion of the chip select. Furthermore the BBP has to output/read new data on every edge(positive or negative depending on the SPI implementation) of the serial clock(SCK) which is supplied by the bus master. Thus, the timer cannot be used to synchronize the wake-ups with the edges. Again, some module that can detect edges and wake the processor should be implemented.

For the I2C it is arguably even more complex. An I2C transfer is initiated by a master by pulling the data line(SDA) low while the clock line(SCL) is high. Again it is necessary to edge detect this transition, and a module can detect these transitions should be implemented to wake the processor. In an I2C transaction an address is sent, and if the address does not match the slave's address the slave should do nothing and wait for the next start of transfer. The processor can go back to sleep and wait for the edge-detection module to wake it.

The UART implementation discussed in this thesis can also greatly benefit from an edge-detection module that can wake the BBP when a RX-transfer is initiated.

The edge detection module will not be of an entirely simple design. It must be configurable to both detect positive and negative edges and it must be able to differentiate between pure edge detection and edge detection while some other condition is set on a different wire in the design. Detection hardware like this is already implemented in the dedicated hardware modules, but the need to implement a different edge-detection unit for each slave protocol reduces the flexibility of the BBP, so some configurable detection module or interrupt controller could be created.

8.3 THE BBP AS A GENERAL PROCESSOR

The bit-banging processor has been designed with optimization for bit-banging as the main goal. The structure proposed in chapter 3 outlines a processor which is designed to function as a peripheral unit without a store/load unit, and this significantly decreases its use as a general processor. Furthermore the BBP does not have any arithmetic instructions and has very few general purpose registers, further decreasing its usefulness as a general processor. However, although the power estimations has targeted this structure they are not without usefulness in describing the BBP as a general processor. The load/store unit is never removed in the estimations and it is entirely possible to estimate the power consumption of a slightly larger, more general processor. If we add an 8-bit adder/subtractor unit, and return the number of general purpose registers to 16, the total current of a general *SolRepeat* becomes 423µA. This does now not include added instruction memory, as it was assumed to replace the AHB, so the total current will probably be higher. Although, it can be assumed that the AHB can be used to access instruction memory and that a more general BBP can use existing on-chip RAM as instruction memory. Thus the BBP can be implemented as a general processor at the cost of, a very approximate, 7.6% increase in power.

8.4 FULL COPROCESSOR

The results show that the BBP cannot entirely replace dedicated modules from a low power perspective. However, some microcontrollers(e.g. the NXP LPC4300-series) employ a big-little architecture where both a high performance processor and a low power processor is implemented on a single chip. The coprocessor can be used to assist the high performance processor during computation intensive periods, or it can be used as the only running processor in periods of low activity. The methods investigated in this thesis can be implemented into such a co-processor to offer reasonably low-power serial communication as well as being a co-processor that can perform more general tasks.

8.5 FREQUENCY REDUCTION

The relatively poor performance at high frequencies can lead to the conclusion that one should avoid using the BBP when high transmission rates are necessary. If high transmission rates are not necessary, then the BBP core frequency can be reduced. While reducing core frequency reduces the dynamic power, it does not reduce the dynamic energy consumption as throughput decreases to match the instantaneous power reduction. But a lower frequency provides a different improvement. Typically, cell libraries contain two types of cells: high threshold voltage cells and low threshold voltage cells. Low threshold voltage cells are faster, but has a significantly higher static leakage than the slower high threshold voltage cells[33]. If the frequency for the BBP can be lowered, it is possible that the BBP can be designed with high threshold cells. This would reduce the static power while making the BBP unable to run at higher frequencies.

8.6 VOLTAGE SCALING

In this thesis it has been assumed that the BBP will run on a 16MHz clock and that voltage is non-changeable. This assumption leads to the conclusion that short time to idle is the best way to reduce dynamic power consumption, but there are alternatives. As can be seen in equation 1 the dynamic power consumption is quadratically proportional to the voltage. If the voltage can be reduced, both the static and dynamic power consumption will also be reduced. A limiting factor for reduction of the supply voltage is the maximum frequency. When the voltage is reduced, the speed of each gate is also reduced and the maximum frequency is lowered[33]. A possible, although complex, improvement would be to implement a voltage scaling technique where voltage can be reduced if the full speed of the processor is not needed. The time to idle would increase, but the voltage and frequency would decrease and lead to a total decrease in energy consumption. It can be difficult to implement voltage scaling due to the variable timing through the processor when the voltage is variable. Timing is also crucial for the communication protocols and it can become very difficult to ensure proper bus-timing if the processor has variable frequencies, but nevertheless it is mentioned as a possible strategy.

9 CONCLUSION

In order to investigate the possibility of implementing a bit-banging processor an instruction set simulator has been created in C. The simulator is cycle accurate, can perform waits of arbitrary length and produces a waveform diagram that can be used for debugging and analysis. A general minimal instruction set has been designed and UART, SPI and I2C protocol implementations have been created. New instructions have been added to improve the performance of the protocol implementations. The SOL-instructions have been created to optimize output/input of serial data, and we find that the Repeat-instruction and the SOL-instructions combined give the best results for *running time* of the protocols. In the instruction set *SolRepeat* running time has been improved by 36% compared to the minimal instruction set. Complexity based power estimates have been performed and while the improvement is somewhat reduced by the increased power consumption of the added hardware, the improvement of *SolRepeat* is still 32%. To avoid overestimating the performance of the BBP, the estimates have been conservative and it is expected that they favor the dedicated hardware implementation. When comparing the bit-banging processor approach to the estimated power consumption of the dedicated hardware modules, we find that the bit-banging processor is not a power-effective solution for high speed SPI or UART. However, I2C, and low-speed UART and SPI show promising results and performs better than the dedicated hardware modules according to our estimates. The BBP was not implemented to replace the dedicated hardware, but to reduce the power consumption of protocols bit-banged in the host processor today. The BBP adds flexibility to low-power communication, and the results are promising.

9.1 RECOMMENDATIONS FOR FURTHER WORK

- Implement slave protocols. An edge detection module that can wake the processor is recommended.
- Implement I2S, USB and other serial protocols.
- Implement parallel protocols, extending the BBP as needed.
- Create the bit-banging processor RTL to perform more accurate power estimates.
- Consider reducing or scaling the frequency to allow for low leakage cells to be used.
- Analyse the need for a protocol processor, and consider implementing a general co-processor with SOL and Repeat functionality.

BIBLIOGRAPHY

[1] D. Lioupis, A. Papagiannis, and D. Psihogiou. A systematic approach to software peripherals for embedded systems. In *Ninth International Symposium on Hardware/Software Codesign. CODES 2001 (IEEE Cat. No.01TH8571)*, pages 140–145. ACM, 2001.

[2] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, fifth edition, 2012.

[3] Byte Paradigm. Using SPI for embedded system debug, 2015.

[4] NXP Semiconductors. I2C-bus specification and user manual, 2014.

[5] telos Systementwicklung. TWI Bus. URL `http://www.i2c-bus.org/twi-bus/`.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, fifth edition, 2011.

[7] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, feb 2013.

[8] G. Gopalasubramanian. [PATCH] add znver1 processor., 2015. URL `https://sourceware.org/ml/binutils/2015-03/msg00078.html`.

[9] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi. *Low power methodology manual: For system-on-chip design*. Springer Publishing Company, Incorporated, 2007.

[10] K. Moiseev, A. Kolodny, and S. Wimer. Timing-aware power-optimal ordering of signals. *ACM Transactions on Design Automation of Electronic Systems*, 13:1–17, September 2008.

[11] K. Agarwal, K. Nowka, H. Deogun, and D. Sylvester. Power Gating with Multiple Sleep Modes. In *7th International Symposium on Quality Electronic Design (ISQED'06)*, pages 633–637. IEEE, 2006.

[12] A. Raghunathan, N. K. Jha, and S. Dey. *High-level power analysis and optimization*. Springer Science & Business Media, 2012.

[13] P. Landman. High-level power estimation. In *Proceedings of 1996 International Symposium on Low Power Electronics and Design*, pages 29–35. IEEE, 1996.

[14] T. Sato, Y. Ootaguro, M. Nagamatsu, and H. Tago. Evaluation of architecture-level power estimation for CMOS RISC processors. In *1995 IEEE Symposium on Low Power Electronics. Digest of Technical Papers*, pages 44–45. IEEE, 1995.

[15] K. D. Muller-Glaser, K. Kirsch, and K. Neusinger. Estimating essential design characteristics to support project planning for ASIC design management. In *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, pages 148–151. IEEE Comput. Soc. Press, 1991.

[16] C. Svensson. Power consumption estimation in CMOS VLSI chips. *IEEE Journal of Solid-State Circuits*, 29(6):663–670, jun 1994.

[17] M. Sheets, F. Burghardt, T. Karalar, J. Ammer, Y. Chee, and J. Rabaey. A Power-Managed Protocol Processor for Wireless Sensor Networks. In *2006 Symposium on VLSI Circuits, 2006. Digest of Technical Papers.*, pages 212–213. IEEE, 2006.

[18] T. Henriksson, U. Nordqvist, and D. Liu. Embedded protocol processor for fast and efficient packet reception. In *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 414–419. IEEE Comput. Soc, 2002.

[19] ARM. Application Note 216, Implementing sleep control for low-power Cortex-M1, 2008.

[20] M. Feathers. CppUTest manual. URL http://cpputest.github.io/manual.html.

[21] J. W. Grenning and J. Carter. *Test-driven development for embedded C*. Pragmatic Bookshelf, 2011.

[22] IEEE Computer Society. *IEEE Standard for Verilog Hardware Description Language – Std 1364-2005*. 2006.

[23] D. Liu. *Embedded DSP Processor Design: Application Specific Instruction Set Processors*. Morgan Kaufmann, 2008.

[24] ARM. Instruction Set Summary - Cortex-M0 Technical Reference Manual, 2009.

[25] Discussions with Jan Egil Øye, 24.11.2015.

[26] Artisan Components. *TSMC 0.18µm Process 1.8-Volt SAGE-XTM Standard Cell Library Databook*. 2003.

[27] ARM. Core registers - Cortex-M0 Devices Generic User Guide, 2009.

[28] A. Th. Schwarzbacher, J. P. Silvennoinen, J.T. Timoney, and Nui Maynooth. Benchmarking CMOS Adder Structures, 2002.

[29] Philips Semiconductors. I2S-bus specification, 1986.

[30] K. Chapman. Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources, 2014.

[31] Discussions with Vegard Endresen, 03.12.2015.

[32] Nordic Semiconductor. nRF52832 Objective Product Specification v0.6.3, 2015.

[33] J. A. Butts and G. S. Sohi. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture - MICRO 33*, pages 191–201, New York, New York, USA, dec 2000. ACM Press.

APPENDIX

A

CELL LIST

Table A.1: Subset of TSMC standard cells[26] used in this thesis.

Cell type	Area[μm^2]	Cell type	Area[μm^2]
ADDFHX1_1V2	76.5072	MXI2X1_1V2	23.2848
ADDFX1_1V2	69.8544	NAND2BX1_1V2	13.3056
ADDHX1_1V2	39.9168	NAND2X1_1V2	9.9792
AND2X1_1V2	13.3056	NAND3BX1_1V2	16.632
AND3X1_1V2	16.632	NAND3X1_1V2	13.3056
AND4X1_1V2	19.9584	NAND4BBX1_1V2	26.6112
AOI211X1_1V2	16.632	NAND4BX1_1V2	23.2848
AOI21X1_1V2	13.3056	NAND4X1_1V2	16.632
AOI22X1_1V2	16.632	NOR2BX1_1V2	13.3056
AOI2BB1X1_1V2	16.632	NOR2X1_1V2	9.9792
AOI2BB2X1_1V2	23.2848	NOR3BX1_1V2	19.9584
BUFX12_1V2	33.264	NOR3X1_1V2	13.3056
BUFX16_1V2	43.2432	OAI211X1_1V2	19.9584
BUFX1_1V2	13.3056	OAI21X1_1V2	16.632
CLKBUFX12_1V2	53.2224	OAI221X1_1V2	23.2848
CLKBUFX16_1V2	63.2016	OAI22X1_1V2	19.9584
CLKBUFX1_1V2	13.3056	OAI2BB1X1_1V2	16.632
CLKINVX12_1V2	63.2016	OAI31X1_1V2	19.9584
CLKINVX16_1V2	83.16	OR2X1_1V2	13.3056
CLKINVX1_1V2	9.9792	OR3X1_1V2	19.9584
DLY1X1_1V2	19.9584	SDFFRX1_1V2	93.1392
DLY2X1_1V2	19.9584	SDFFSX1_1V2	106.4448
DLY3X1_1V2	23.2848	TLATNX1_1V2	36.5904
DLY4X1_1V2	23.2848	XNOR2X1_1V2	26.6112
INVX12_1V2	43.2432	XOR2X1_1V2	26.6112
INVX16_1V2	56.5488	TLATNRX1_1V2	43.2432
INVX1_1V2	6.6528	TLATNSX1_1V2	53.2224
MX2X1_1V2	26.6112		