NTNU

Det skapende universitet

Improving the energy efficiency of a microcontroller instruction fetch using tight loop cache

Maja Popovic

Embedded Computing Systems

Innlevert: juli 2015

Hovedveileder: Donn Morrison, IET

Medveileder: Marius Granaes, Silicon Labs

Norges teknisk-naturvitenskapelige universitet

Institutt for elektronikk og telekommunikasjon

NTNU – Trondheim
Norwegian University of
Science and Technology

Improving the energy efficiency of a microcontroller instruction fetch using tight loop cache

Maja Popovic

Electronics System Design and Innovation

Date of submission: July 2015

Supervisors: Donn Morrison
Marius Grannaes

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Recently there has been a lot of effort in making the Internet of Things (IoT) a reality. A central component of this vision is to make low power edge sensor nodes (i.e., nodes with few connections that are not used to route data) in a mesh network. Such systems are often composed of a low power microcontroller coupled with a low power radio operating at low speeds with long duty cycles. A lot of research has been conducted with regards to reducing the power consumption of these systems.

A significant portion of the energy is used for fetching instructions from flash. In some low-power microcontrollers a small cache is used to exploit temporal locality in the instruction stream. Energy is saved, because SRAM used in caches require less energy than flash.

Cache is a very old and well known technique to exploit such differentials in speed/energy. This master thesis will build on the tight loop cache approach, use software simulations to evaluate if it can be used in an application to save energy, proceed to its hardware design and implementation and compare results.

Master thesis was done in collaboration between the CARD group and Silicon Labs, where Marius Grannæs was a co-supervisor.

Abstract

Energy efficiency in microcontrollers has played an important role in modern digital systems for years. With the increased need for longer battery life and increased complexity of functionalities offered, it becomes crucial to lower energy consumption as much as possible. Studies show that largest amount of energy in embedded systems gets consumed by the memory hierarchy system. Therefore there has been a lot of research pressure in the area of caching techniques with the attempt to reduce energy requirements and thus make battery life longer. Some of these techniques were studied and analyzed in author's Semester Project whereas one of them, called Tight Loop Cache, was chosen to be implemented and evaluated as the most promising when power optimization is concerned. TLC is different from conventional caching techniques because it does not include tagging of cache lines nor valid bits which makes it more attractive and easy to incorporate into a working system.

The technique was implemented both in software (Python) and hardware (Verilog) and later evaluated by counting parameters close to meanings of cache hits and misses. These parameters were then used to calculate power consumption of the system without making use of the technique and with incorporating the technique.

Software simulations showed that using TLC of size only 64B brings benefits of power savings from 10% up to 80% for some benchmarks (taking into account only the memory system consumption). Relying on these encouraging results, the technique was implemented in hardware, synthesized on Xilinx Zynq-7000 and evaluated using power reports generated by VIVADO Design Suite. Hardware implementation was built around ARM Cortex-M0 and included the design of main instruction memory, TLC Controller and tight loop cache itself.

Post implementation power reports showed that the use of TLC of 64B can bring around 25% of power savings into a system working on 10 MHz and synthesized with FPGA fiber. It is believed that, if implemented as an ASIC with completely configurable and controllable synthesis and place and route tools, the design would bring even more power savings than in the case of using a Zynq-7000.

.

Preface

This thesis was submitted to the Norwegian University of Science and Technology for fulfilling final requirements for a master's degree. The work has been conducted at the Department of Computer and Information Science, NTNU, Trondheim with Donn Morrison as a supervisor in a collaboration with company Silicon Labs from Oslo with Marius Grannaes as a co-supervisor.

Acknowledgements

I would like to thank both my supervisors, Donn Morrison and Marius Grannaes, for guiding me, giving support and new ideas even while being on holidays! Weekly meetings with Donn helped me a lot to get organized and get the feeling of time management importance, whereas Marius's vast knowledge of software compilation and EFM32 in general, discarded any doubt that crossed my mind.

Special thanks to Donn and hardware supply management at NTNU for ordering all the hardware support needed for this thesis to be successfully fulfilled.

Contents

Problem Description ... iii
Abstract ... iv
Preface ... v
Contents ... vi
List of Figures ... viii
List of Tables ... x
List of Abbreviations ... xi
1 Introduction ... 1
1.1 Energy Efficiency and Microcontrollers ... 1
1.2 Thesis Overview and Main Contributions ... 2
2 Background and Related Work ... 4
2.1 Related Work ... 4
2.1.1 Filter cache ... 4
2.1.2 Predictive Filter Cache ... 4
2.1.3 Loop Cache ... 4
2.1.4 Tagless Hit Instruction Cache ... 5
2.1.5 History Based Tag Comparison Cache ... 5
2.1.6 Horizontal Cache Partitioning with Gray encoding ... 5
2.2 Tight Loop Cache ... 6
2.3 Software System Background ... 9
2.4 Hardware System Background ... 11
2.4.1 Cortex-M0 ... 12
2.4.2 AHB bus ... 23
2.4.3 Simulating the Cortex-M0 core ... 28
2.4.4 Running the core with Instruction Memory on FPGA [18] ... 29
2.4.5 System Design Flow and Power Optimization ... 38
3 Tight Loop Cache Implementation ... 43

3.1 TLC in software 43
3.2 TLC in hardware 45
3.2.1 Adding Loop Cache 45
4 Testing and Measurements 59
4.1 Software Implementation 59
4.1.1 Methodology 59
4.1.2 Results 59
4.2 Hardware Implementation 60
4.2.1 VIVADO measurements 60
4.2.2 Real time measurements 67
5 Conclusion and Future Work 75
5.1 Conclusion 75
5.2 Future Work 76
References 78
Appendix A 80
Appendix B 86

List of Figures

Figure 2.1 - sbb instruction format 6
Figure 2.2 – Loop cache organization and access [10] 7
Figure 2.3 - Loop cache controller state machine 8
Figure 2.4 - Design of loop counter [10] 9
Figure 2.5 – Address bus content in time 11
Figure 2.6 - ARM Cortex-M0 block diagram [12] 14
Figure 2.7- Cortex-M0 memory map [13] 15
Figure 2.8- Program Memory content [13] 16
Figure 2.9- Instruction byte order in memory [14] 17
Figure 2.10- Simulation of the loop execution 18
Figure 2.11- Cortex-M0 pipeline stages[15] 21
Figure 2.12- Cortex-M0 schematics[16] 21
Figure 2.13 - Cortex-M0 access types[16] 23
Figure 2.14 - AHB-Lite block diagram [17] 24
Figure 2.15- AHB Master interface 24
Figure 2.16- AHB Slave interface 26
Figure 2.17 - AHB basic transfer: read and write [17] 27
Figure 2.18 – ARM Design Start testbench deliverables: block diagram 28
Figure 2.19 – Core simulation results 29
Figure 2.20 – Block diagram of the system built for implementation on FPGA board 30
Figure 2.21 - Clock generation on zc702 Evaluation Kit 30
Figure 2.22 – 7Series primitive elements: buffer and D flipflop 31
Figure 2.23 – SynqReset module block diagram 32
Figure 2.24 – Constant2Pulse module block diagram 32
Figure 2.25 - Counter2Constant module block diagram 33
Figure 2.26 – Reset pulse generation: simulation 33
Figure 2.27 – Block RAM principle of operation 34
Figure 2.28 – Detector module interface 35
Figure 2.29 – Generation of Detector signal: simulation 35
Figure 2.30 – Source code used in [18] 37
Figure 2.31 – Translation from .c file to .coe file 38

Figure 2.32 – IC Design general flow 38
Figure 2.33 – Digital Design Flow 40
Figure 3.1 – Difference between data transaction and program jump with instruction addresses as inputs 44
Figure 3.2 – Insertion of a pseudo state for data transaction detection 45
Figure 3.3 – Interface between the system and the user 46
Figure 3.4 – Generation of global_cache_enable signal 46
Figure 3.5 – Introducing Loop cache into the core system 47
Figure 3.6 – Interface between Controller, Instruction memory and Loop cache 48
Figure 3.7 – Two different Loop Cache Controller implementations and their interfaces 48
Figure 3.8 – Timing comparison between two different controller implementations .. 50
Figure 3.9 – Controller implementation, its state switching and control signal toggling 50
Figure 3.10 – Controller State Machine with output control signals 53
Figure 3.11 – Filling of the Loop Cache 55
Figure 3.12 – Two different Controller-Cache implementations and interfaces 56
Figure 4.1- Complete power report, VIVADO layout 63
Figure 4.2- Power report of the system with no memory hierarchy 64
Figure 4.3 – Design timing repor 64
Figure 4.4 - zc702 Evaluation Kit board layout [20] 68
Figure 4.5 - System user interface and output status (LEDs, switch and push buttons) 69
Figure 4.6 - Power domains on the ZC702[26] 70
Figure 4.7 – Format of a .csv file with real time measurements saved by TI FDPD ... 70
Figure 4.8 – Real time voltage measurements for LEDs periodically ON and OFF (Matlab) 72
Figure 4.9 – Real time power consumption for LEDs periodically ON and OFF 72
Figure 4.10 – Real time current measurements for LEDs periodicallt ON and OFF ... 73

List of Tables

Table 2.1- Cortex-M0 and Cortex-M3 specification comparison [12] 13
Table 2.2 – Branching instructions supported by ARMv6-M [14] 18
Table 2.3- Loop Instructions 19
Table 2.4- AHB Master output signals of interest 25
Table 2.5 – C code variables and corresponding core registers 37
Table 3.1 – State switching mechanism with logical switching conditions and their implementations 51
Table 3.2 – Controller output signals 54
Table 3.3 – Instruction Memory Enable signal generation 54
Table 3.4 – Controller output signals (first version) 56
Table 3.5 – Utility reports of three different implementations of the Controller 57
Table 4.1 – Simulation results for cache size 64B 59
Table 4.2- Source code modification impact on program loop size 62
Table 4.3 – Dynamic power reports after synthesis and after implementation using different toggling information (32KB Instruction Memory, 64B cache, loop size 8) 62
Table 4.4 – Power report generation details for different measurement configurations 62
Table 4.5 – Different testing configurations 65
Table 4.6 – Power and statistic reports for different system configurations 65
Table 4.7 – Total and Vcc3V3 power consumption for 4 LEDs ON and OFF 71
Table 4.8 – Total power consumption (loop size: 8 instructions) 73

List of Abbreviations

AHB	*Advanced High Performance Bus*
ALU	*Arithmetic Logic Unit*
AMBA	*Advanced Microcontroller Bus Architecture*
AP SoC	*All Programmable System-on-Chip*
APB	*Advanced Peripheral Bus*
APSR	*Application Process Status Register*
CLB	*Complex Logic Block*
COF	*Change of Flow*
CPU	*Central processing Unit*
DCM	*Digital Clock Manager*
EDA	*Electronic Design Automation*
EFM32	*Energy Friendly Microcontroller 32b*
FPGA	*Field Programmable Gate Array*
HADDR	*AHB address signal*
HDL	*Hardware Description Language*
HKMG	*High K Metal Gate*
ISA	*Instruction Set Architecture*
LUT	*Look Up Table*
LVDS	*Low Voltage Differential Signalling*
MMCM	*Mixed Mode Clock Management*
MSP	*Main Stack Pointer*
NMI	*Non - Maskable Interrupt*
NVIC	*Nested Vectored Interrupt Controller*
PLL	*Phase Locked Loop*
PMB	*Power Management Bus*
PSP	*Process Stack Pointer*
RISC	*Reduced Instruction Set Computer*
SAIF	*Switching Activity Interchange Format*
SBB	*Short Backward Branch*
TH-IC	*Tagless Hit - Instruction Cache*
TI FDPD	*Texas Instruments Fusion Digital Power Designer*

TLC *Tight Loop Cache*

TNS *Total Negative Slack*

WNS *Worse Negative Slack*

XDC *Xilinx Design Constraints*

1 Introduction

1.1 Energy Efficiency and Microcontrollers

There has always been conflict between low cost, high performance and low power consumption specifications in modern digital systems. MCUs are by far the best candidates to build systems targeting these applications mainly because they are self-contained: CPU, on chip SRAM, non-volatile flash memory and other peripherals are all integrated in one chip which consumes far less power than if all these components were used separately. Basic principle behind energy consumption minimization is to put MCU to sleep for as much time as possible, wake it up to perform a certain task as fast as possible and then put it back to sleep. There is no gain if a system consumes less power if it takes it much more time to complete the task (energy consumption is what actually matters at the end).
Low power consumption not only brings energy savings, but it also improves system reliability as a whole by reducing heat dissipation. This way components have longer life expectancy because their temperature does not change rapidly, they operate on a stable temperature and therefore there is no need for large cooling systems.

In most digital systems, memory system consumes great part of the overall power consumption and this is why recently a lot of effort has been given to memory hierarchy design in a sense it consumes as little power as possible. Instruction memory access is one of the crucial points where these design modifications can be considered. The reason lays down in the fact that in a typical RISC ISA there are usually four times more instruction than data memory accesses [1]. Moreover, data is most commonly stored in SRAM whereas programs are stored in flash memory whose access infers much more energy consumption. These are all the reasons why it is believed that reducing instruction fetch energy consumption in systems like this would bring a great deal of overall energy consumption reduction.

Embedded application programs usually consist of small number of loops executed many times. It comes natural the thought it would be very efficient to read those instructions from a small buffer (small cache) and thus reduce energy consumption. Most common approaches that involve caching hierarchies put this buffer between CPU and main memory which usually infers time penalties whenever there is a cache miss. Other schemes involve

accessing main memory in the same cycle if there is a miss but with the penalty of longer cycle time.

This project explores and implements, both in hardware and software, a caching technique that is believed to overcome both problems (longer cycle time and multiple cycle access) and offer great power savings. It involves using a tight loop cache and a loop cache controller as it will be explained in Chapter 2.2. This technique differs a lot from the original caching techniques (no tagging, no valid bits) and this is the reason why in this report not too much attention was given to basics of caching mechanism. The second reason lays down in the fact that basics of cache as well as other conventional and non-conventional techniques were covered in depth in the author's Winter Semester project and more details can be found there [2].

Since the thesis was a collaboration between NTNU and Silicon Labs, EFM32, MCU from Silicon Labs, was used as a starting point to choose the main processing core for the system to be built around. Nevertheless, TLC is processor independent and can be integrated into almost any modern system. Another reason why EFM32 was chosen was its energy friendliness and popularity amongst designers:
"In a market crowded with MCUs from larger vendors, Silicon Labs' EFM32 Gecko family merits a close look"[3].

1.2 Thesis Overview and Main Contributions

Chapter 2 First part of the Chapter describes some of the caching techniques explored in [2], both conventional and non-conventional ones. The purpose is to show how TLC compares with other techniques putting emphasis on both its simplicity and efficiency.

Second part explains the TLC technique the way it was originally proposed in [10].

Next section of the Chapter explains briefly what was done in the Semester Project and what the results and problems were.

The last section goes into detailed description of the ARM Cortex-M0 architecture, instruction set and its communication interface with other system components through AHB interface. It also gives overview of the design modules obtained from ARM as well as other hardware and

software sources found on the internet that were used to build a complete environment that could be tested and evaluated.

Chapter 3 Presents details of TLC implementation in software and hardware. Software implementation includes modifications made to the simulator used in Semester Project with a new state machine employed.
Hardware implementation describes interfaces of building modules as well as details of their operation either in principle or by original simulation figures.

Chapter 4 Presents experimental setup and results gained from performing different tests of both SW and HW implementations. Evaluation of HW implementation contains power reports obtained both form simulation (VIVADO) and real time measurements.

Chapter 5 Thesis ends with the summary of the delivered work and discussion about possible improvements that could be made to the system.

Through the work on the thesis, author made several achievements that can be regarded as a significant effort:

1. Detection of data transactions in the input benchmarks used in the software simulation
2. Software simulator [2] upgrade with the addition of pseudo states which enabled Controller ignore data transactions and thus bring more power savings
3. Hardware implementation of TLC in Verilog and its integration into a system that uses ARM Cortex-M0 Design Start operating on 10 MHz with 32KB Instruction Memory
4. Exploration, description and application of synthesis and implementation power optimization design flow offered by VIVADO tools

2 Background and Related Work

First part of this chapter briefly mentions some of the caching techniques that were explored in author's semester project [2] before tight loop cache was chosen to be implemented and tested. Chapter content then continues to explain in details the tight loop cache technique and the environment that it was implemented on. It is supposed that the reader has knowledge of basics of cache but if that is not the case, more information before reading this Chapter could be obtained from [1] and [2].

2.1 Related Work

2.1.1 Filter cache

One of the first techniques that was proposed in [4] in order to bring power savings was involving insertion of a small, so called Filter Cache, which was communicating directly with the CPU. Although this brought great amount of power savings, it led to a slight performance degradation.

2.1.2 Predictive Filter Cache

Tang, Gupta and Nicolau went further in [5] and tried to reduce this timing degradation introduced when there was a miss in the filter cache. Their strategy involved analysis of subsequent fetch addresses at runtime to predict whether next instruction was in the loop cache or not. If it was predicted to be a cache miss, the instruction was fetched from next cache level, bypassing filter cache (consuming more energy though in case the prediction was wrong but eliminating timing penalty in case it was true). The mechanism used to make the prediction was comparison of the tag of the current fetch and the tag of the predicted next fetch address: if they were the same it was predicted that the next fetch resided in the cache and vice versa.

2.1.3 Loop Cache

Technique proposed in [4] was improved in [6] by Bellas, Hajj, Polychronopoulis and Stamoulis by enhancing both compiler and hardware properties of the system. The compiler not only selected which parts of the code to put into the cache but also did the code structuring to avoid instruction conflicts. Additional hardware included extra control to choose whether to access Loop Cache or Instruction Memory. .

2.1.4 Tagless Hit Instruction Cache

Another quite recently proposed technique that also tried to improve timing penalty of the basic caching idea from [4] was presented by Hines, Whalley and Tyson in [7]. They introduced the so called Tagless Hit Instruction Cache (TH-IC) which was placed as L0 (Level 0) cache between the CPU and L1 cache with a difference it was accessed at the same time as L1 cache but the instruction was read from L1 cache only when it was not guaranteed that it resided in the cache. By accessing both memories at the same time it was guaranteed that there was no timing penalty in case of a cache miss. Another big advantage over other techniques is the lack of need for storing tags for each cache entry since the technique itself already excluded tag comparison (that is where the name Tagless Hit came from).

2.1.5 History Based Tag Comparison Cache

Authors in [8] aimed at lowering tag access energy by using the fact that cache hit rates are usually very high which means that data in the cache rarely gets changed and thus tag checks do not need to be performed if data in the cache had not been replaced. Only after there was a miss (data in the cache had been updated), tags needed to be compared in order to determine if the data still resided in the cache or not. The results showed that number of tag comparisons can be reduced by even 90% and this way bring great power savings.

2.1.6 Horizontal Cache Partitioning with Gray encoding

A study about comparison of different cache techniques that was conducted by Su and Despain with results presented in [9], showed that, in general, whenever cache size increased missed rate decreased and that direct mapped caches saved more power but had larger access time than fully associative ones which was expected as well. It was also shown that when using sub-banking, larger blocks (containing entire loops in best case) bring more power savings, whereas Gray coding of memory addresses reduces bit switching up to 33% and thus brought even more energy savings.

There were more techniques explored in [2] apart from the ones mentioned in this chapter and one of them is the one that was chosen to be implemented and tested:

“After thorough analysis, taking into account the information that can be obtained from BRCHSTAT signal as well as the notion of the instructions that can be executed by the core,

some of the techniques from Chapter 2 were completely discarded and some were purposed for implementation. Technique that was regarded as the simplest, the most straight forward and with the least hardware control overhead was the one about Tight Loop Cache...” [2]

Next sub chapter explains the technique in details, the way it was originally purposed in [10], although when implemented in this project, both in hardware and software, slight modifications were introduced as it was already explained in [2].

2.2 Tight Loop Cache

Tight Loop Cache, a technique proposed in [10] and chosen to be implemented in this project, consisted of a small direct map memory array and a loop cache controller. The advantage of using loop cache was double: it did not contain tag nor valid bit for each data instance. On the other hand, there was no timing penalty if there was a cache miss since the Controller had the early notion whether next fetch was going to be a hit or a miss. Based on this information, the core accessed either the loop cache or the main instruction memory.

General principle was based on the detection of the so called sbb (short backward branch) which, when encountered, indicated that a loop was executed for the second time, i.e. the moment when loop cache started to be filled. Next detection of the same sbb indicated the data was already in the loop cache and could be read from there.

Short backward branch instruction was any kind of branching instruction, both conditional and unconditional, that had the format as shown in Figure 2.1:

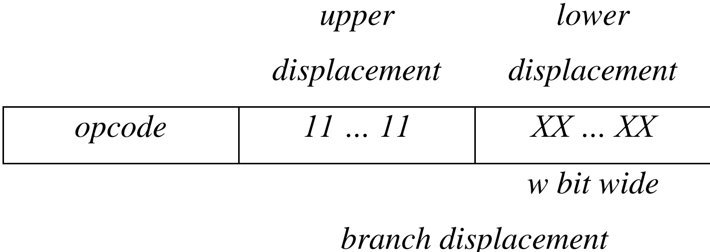


Figure 2.1 - sbb instruction format

Upper displacement containing all ones suggested that the branching was performed backwards (end of the loop is encountered) whereas the lower displacement field determined how long the backward jump was (how many instructions the loop consisted of). Lower part of

the displacement field was w-bit wide which was directly connected to the size of loop cache, i. e. cache could not contain more than 2^w instructions (in case of architecture where each memory location contained only one instruction and program counter was incremented by 1 to access next instruction).

This made sure the loop size could not be larger than the cache size. As mentioned earlier, tight loop cache was direct mapped (contained no address tags), only accessed by index field which was w bits wide. When a loop was smaller than 2^w instructions, only part of the loop cache was used and loop start did not have to be aligned to any particular address as in the case of many other techniques.

Figure 2.2 shows how loop cache was organized and accessed in a case of $n=2^w$ entries, each entry containing 2 bytes (the last bit of the instruction address was neglected).

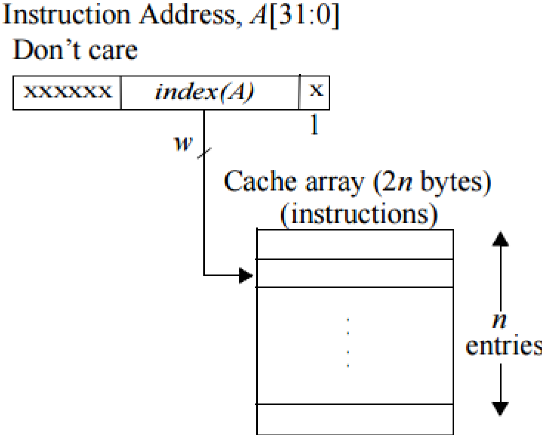


Figure 2.2 – Loop cache organization and access [10]

Loop Cache Controller was designed as a state machine with three states: IDLE, FILL and ACTIVE. Initially, the Controller was set to be in the IDLE mode all the time until it had been detected that there was an sbb in the instruction stream. If the controller determined that there was an sbb (information is received from the decoder) and that the branch was taken (information received from a branch status signal from the core), this meant there was a loop encountered and that it was going to be executed for the second time which made Controller move to FILL state. The sbb that forced the Controller enter FILL state was called triggering sbb. In the FILL state, instructions were still read from the main instruction memory, but at the same time cache was filled with the loop instruction stream. This state continued until there

was no other change of flow (cof), i.e. no other branch or jump instruction (the program execution was sequential within the loop itself).

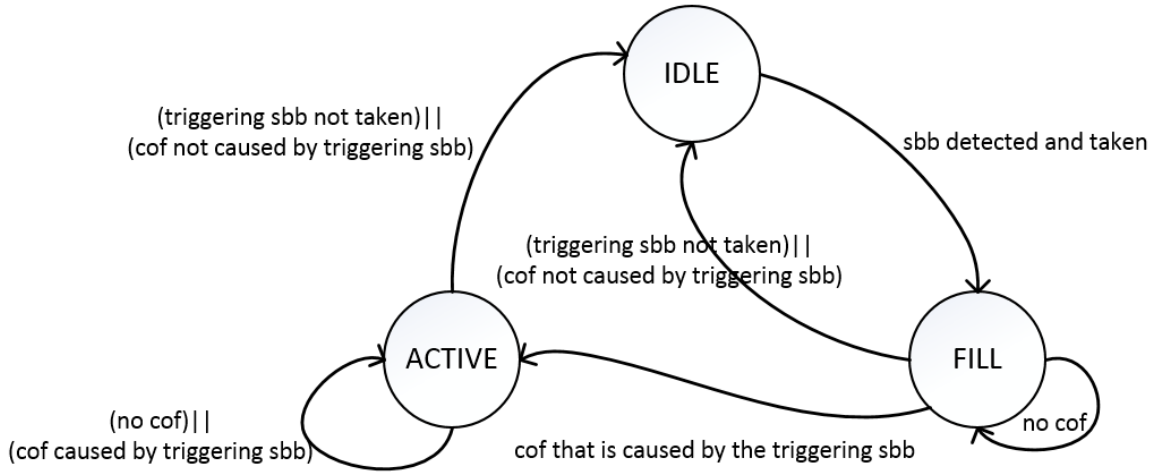


Figure 2.3 - Loop cache controller state machine

Controller went back to IDLE state in case it encountered a non-sequential stream in the loop sequence which was not caused by the triggering sbb (some other branching/jumping instruction inside the loop itself) or if it determined that triggering sbb was not taken. Finally, if the triggering sbb was taken again, the Controller entered ACTIVE state and started reading instructions from the cache. It stayed in the ACTIVE state as long as the loop within itself remained sequential and as long as the triggering sbb, when encountered, was taken (the loop was going to be executed again). In any other case, the Controller went back to IDLE state. There was no way the Controller could migrate from ACTIVE state back to FILL state which was logical considering possible scenarios.

The most important piece of information for the Controller to determine whether next instruction was a hit or a miss was to know when the triggering sbb was fetched, executed and whether the cof was caused by the triggering sbb or some other instruction. The mechanism that made this possible was implemented as the Loop Counter mechanism shown in Figure 2.4.

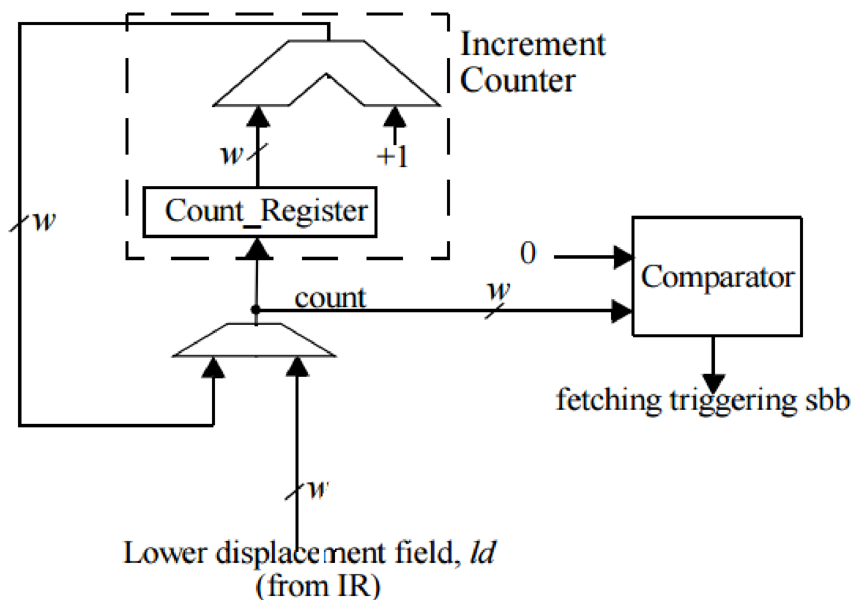


Figure 2.4 - Design of loop counter [10]

The Controller was initially in IDLE state and stayed there until sbb was detected in the decode pipeline stage when its lower displacement field got loaded into the Count Register of the Increment Counter. This displacement gave the information of how many sequential instructions needed to be executed before sbb was fetched again. After the ld field was saved and later on determined that sbb was taken as well, the Controller entered FILL state. While in this state, on each sequential instruction execution, Increment Counter was incremented by one. By the time Counter reached zero, the Controller knew that the triggering sbb was being fetched. If the sbb was taken, the Controller entered ACTIVE state and the Increment Register was again loaded with the ld field of triggering sbb. This meant that the execution of the loop started from the beginning again. Whenever the counter reached zero again, the same process was repeated. Using this mechanism, the Controller knew when a cof was caused by the triggering sbb only by examining the value in the Increment Register.

The original technique of using tight loop cache and loop controller, proposed in [10] and briefly explained here, was implemented both in software and hardware with slight modifications as it will be described thoroughly in the next chapter. Figure 3.11 shows cache being filled and read from with an example of loop execution on the core.

2.3 Software System Background

This project was a continuation of work that was conducted during Autumn student semester project during which the author explored different caching techniques and chose to

implement one of them, the so called Tight Loop Cache (simple and easily adaptable to any core environment). The principle of operation of the Loop Cache Controller was based on the principle explained in [10] and in the Chapter 2.2 but its implementation in Python was slightly different since the input data was different. The input data, as explained in [2] were benchmarks that were comprising of .csv files that contained memory addresses of consequently fetched instructions. Each next row contained next address to be fetched. A loop was detected by comparing two subsequent fetch addresses and determining that the previous address was larger than the current one. This meant that an sbb occurred and, if within the size of the loop the cache can hold, the Controller left IDLE state and entered FILL state. More details about implementation and actual Python code created as part of the Semester project can be found in [2].

The results from [2] showed that in some benchmarks the technique could bring a great deal of power savings (even up to 70%, taking into account only memory power consumption) whereas in some other benchmarks there was even loss if the cache was used. At the end of project report [2], it was noticed that there were some irregular jumps within loops themselves that caused Loop Cache Controller suddenly leave fill or active state and spend most of its time in idle state which didn't bring great deal of savings of course.

Analysing input data (program traces) more thoroughly this time and plotting them in Matlab made it possible to realize a crucial environment setup mistake that led to semester project results seem non-understandable. Figure 2.5 shows sequence of instruction fetch addresses in one of the benchmarks. As it can be seen, there was a loop consisting of 5 instructions that was over and over again (instruction addresses were gradually growing instruction by instruction and then suddenly dropping to the initial value). The Controller would never go to ACTIVE state while executing this loop and the reason is of course the non-sequential execution within the loop (large jump in address value after two loop instructions).

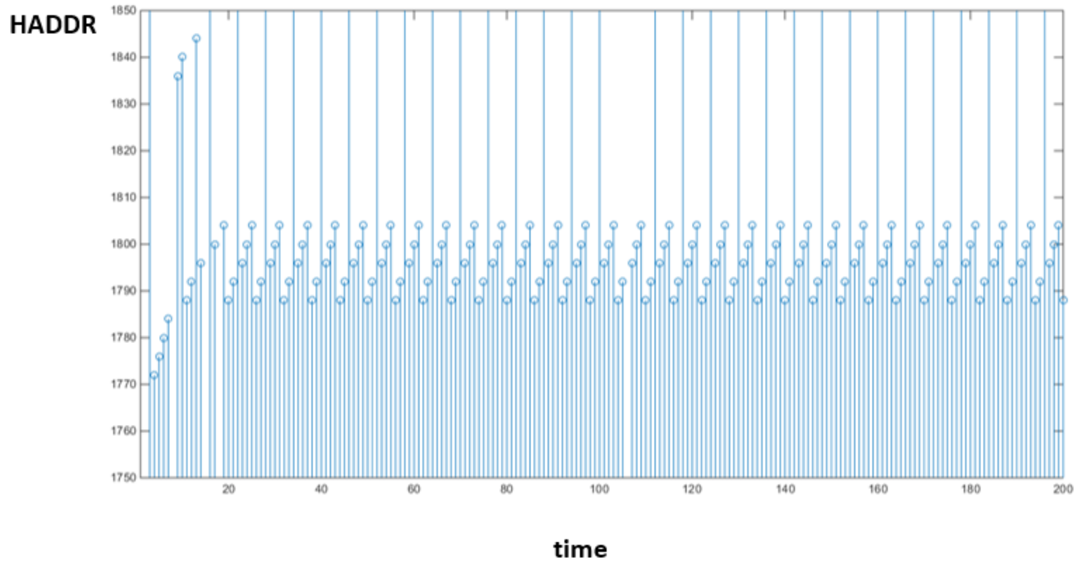


Figure 2.5 – Example of Instruction fetch addresses sequence

The only possible explanation behind these "jumps" inside loop execution lays down in the existence of data reads and writes which are addressed using the same address bus as the instruction fetches. Those data reads/writes cause the Controller toggle between IDLE and FILL states without ever entering ACTIVE which brought only more power dissipation.

Software implementation of the tight loop cache system in this project represents an upgrade of the system implemented in [2] taking into account these data fetches (sudden jumps in address values on the bus) and allowing the controller stay in the same state it was in (not going automatically to idle) even though there is a non-sequential fetch. More details about the implementation will follow in Chapter 3.1.

2.4 Hardware System Background

Work on this project was monitored and supported by the company Silicon Labs, the branch located in Oslo, Norway. The company developed two families of microcontrollers that support low energy consumption concept:

- EFM8: 8-bit microcontrollers developed around 8051 and

EFM32: 32-bit microcontrollers developed around ARM Cortex.

This project was dealing with EFM32 MCUs which are based either on ARM Cortex-M0+, ARM Cortex-M3 or ARM Cortex-M4 and are used together with low power peripherals to address any low power application (communications, alarm and security systems, control

systems, industrial sensors, medical solutions, car and traffic control systems …) According to Silicon Labs, EFM32 are the world's most friendly microcontrollers and their advantages over other similar products are certainly the use of 5 different energy modes, very fast wake up time, reduced processing time, energy efficient peripherals etc.

EFM32 are classified into 6 categories (Zero Gecko, Tiny Gecko, Gecko, Leopard Gecko, Giant Gecko and Wonder Gecko) depending on the CPU they use, amount of program memory they have and how much power they consume in different power modes. This variety of choice makes EFM32 quite attractive amongst designers:

"The EFM32 family's wide variety of MCUs, peripherals, I/O interfaces, packages, and other features gives system designers a wealth of choices. In summary, Gecko MCUs are a good choice for small embedded systems that spend most of their time sleeping and then awaken for brief bursts of activity."[3]

Next section will explain why Cortex-M0 Design Start was used as a final core in the implementation, describe its architecture organization and illustrate briefly how the program executed.

2.4.1 Cortex-M0

Initially, Cortex-M3 was chosen as the core to implement the cache system around since it was used in 4 out of 6 different categories of the EFM32 family. The biggest advantage of using Cortex M3 would be its possession of the signal called BRCHSTAT [3:0] which offers branch status information of the instruction in decode and next execute pipeline stage. As described in [11] this signal could give information if the branching instruction is conditional or unconditional, taken or not taken and in decode or execute stage of the pipeline which would be extremely useful as input information for the Loop Cache Controller to early determine whether next fetch was going to be a hit or a miss in the loop cache.

Unfortunately, it was not possible to obtain licenses, use the Cortex-M3 core itself and simulate the system behaviour in the real EFM32 environment. Instead, a Cortex-M0 core was used since its licensing with ARM was not a problem and the Verilog implementation was obtained in a fairly easy way. Main architectural differences and similarities between these cores are shown in Table 2.1. Furthermore, the obtained core was part of a so called, Design Start Package which introduced certain limitations to the usage of the core itself.

Table 2.1- Cortex-M0 and Cortex-M3 specification comparison [12]

	Cortex-M3	*Cortex-M3*
Architecture	*ARMv7-M*	*ARMv6-M*
Pipeline	*3 stage with branch prediction*	*3 stage*
Insruction set	*-Thumb (entire)*	*-Thumb (most)*
	-Thumb2 (entire)	*-Thumb2 (some)*
Interrupts	*240, plus NMI*	*32, plus NMI*
Performance Efficiency	*3.34 CoreMarks/MHz*	*2.33 CoreMarks/MHz*
Performance Efficiency	*1.25 / 1.50 / 1.89 DMIPS/MHz*	*0.87 / 1.02 / 1.27 DMIPS/MHz*

As it can be seen from Table 2.1 both cores have 3 stage pipeline and they use the same instruction set which is the necessary condition for the cache system to migrate from one core to another although with different power consumption reports of course because they have different performance efficiencies which can also be seen from Table 2.1.

2.4.1.1 Core block diagram

Cortex M0 processor is a 32b RISC processor with von Neumann architecture (program and data memory not separated, share the same bus) which uses Thumb instruction set and supports some functions from Thumb-2 (an upgrade of Thumb in a sense that it is possible for all the instructions to be executed in one CPU state). Thumb-2 instruction set includes both 16b and 32b instructions although latter is only used when none of the former cannot be used to complete the operation.

Cortex-M0 block diagram is shown in Figure 2.6. Main processing core consists of register bank (sixteen 32b registers), ALU and control logic with a three stage pipeline: fetch, decode and execute. Nested Vectored Interrupt Controller has the ability to accept NMI and up to 32 interrupt requests and decide which one to serve comparing their priority levels.

Wake Up Interrupt Controller is an optional module and is used to urge the power management unit to wake up the CPU and NVIC if they are in standby mode and there was an interrupt request.

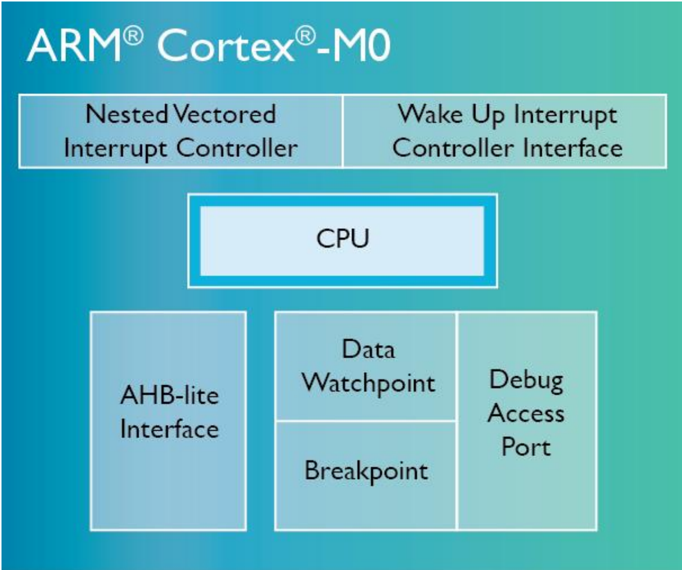


Figure 2.6 - ARM Cortex-M0 block diagram [12]

Debug Access Port is used to make system development and testing easier and faster: deals with program breakpoints, data watchpoints and debug control in general.

AHB-lite bus interface is a 32b wide on chip bus protocol which is part of the Advanced Microcontroller Bus Architecture (AMBA) specification that was developed by ARM.

2.4.1.2 Core Memory System

The Cortex-M0 has 4GB of memory address space (2^{32}) divided in regions with each region having recommended usage that enables easier software migration between devices based on this core (programming model for interrupt control and debug are the same). Figure 2.7 shows different memory regions in 4GB memory space of the Cortex-M0. The core supports memory transfers of different sizes, such as byte, half-word and word wide either little or big endian memory system.

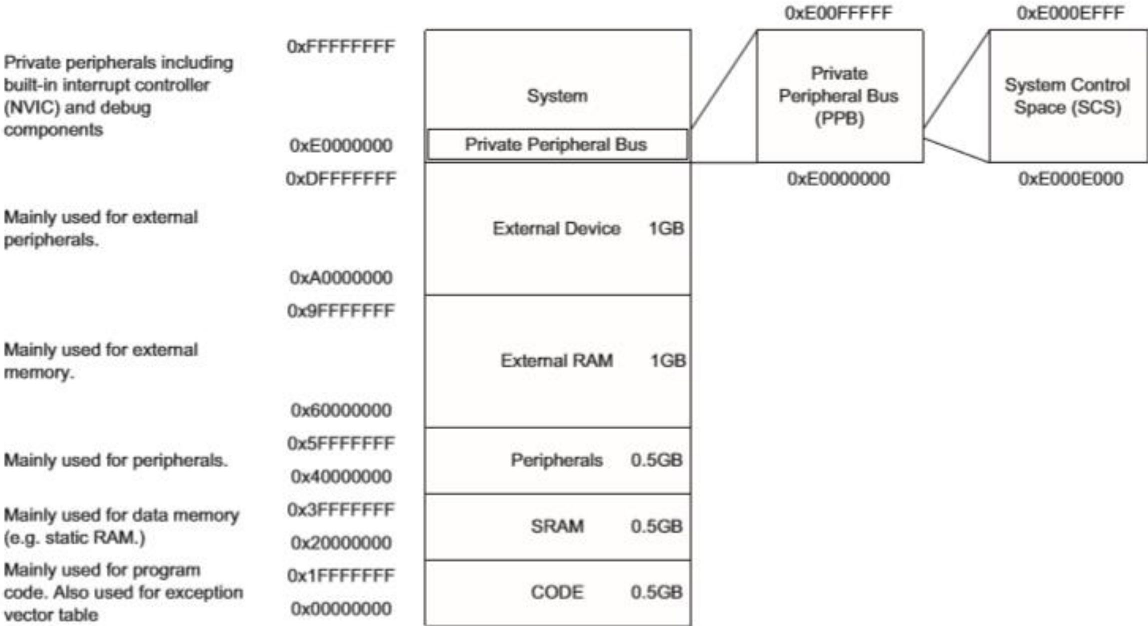


Figure 2.7- Cortex-M0 memory map [13]

Taking into account the tittle of the thesis it is natural to assume why more attention will be paid to the memory section that starts at 0x00000000 and is dedicated to the code. MCUs have on-chip flash memory that contains the binary code and some of them also have separate boot ROM that contains boot loader program which is executed before the user program (the content of this ROM usually cannot be changed).

In the case of EFM32, program image starts with a vector table (starting addresses of interrupt handlers) whose size depends on number of interrupts that are implemented (starting address of the vector table depends on the implementer of Cortex-M). Figure 2.8 shows the content of vector table with some basic interrupt vectors that have to exist in every system whereas all the vectors from address 0x00000040 (address of the vector for IRQ #0) are arbitrary and exist only if implemented in the program. As it can be seen in the Figure 2.8, first word of the vector table contains initial main stack pointer value (MSP, contains the address of

the top of the stack) and right after it comes the reset vector which contains the address of the first instruction fetch.

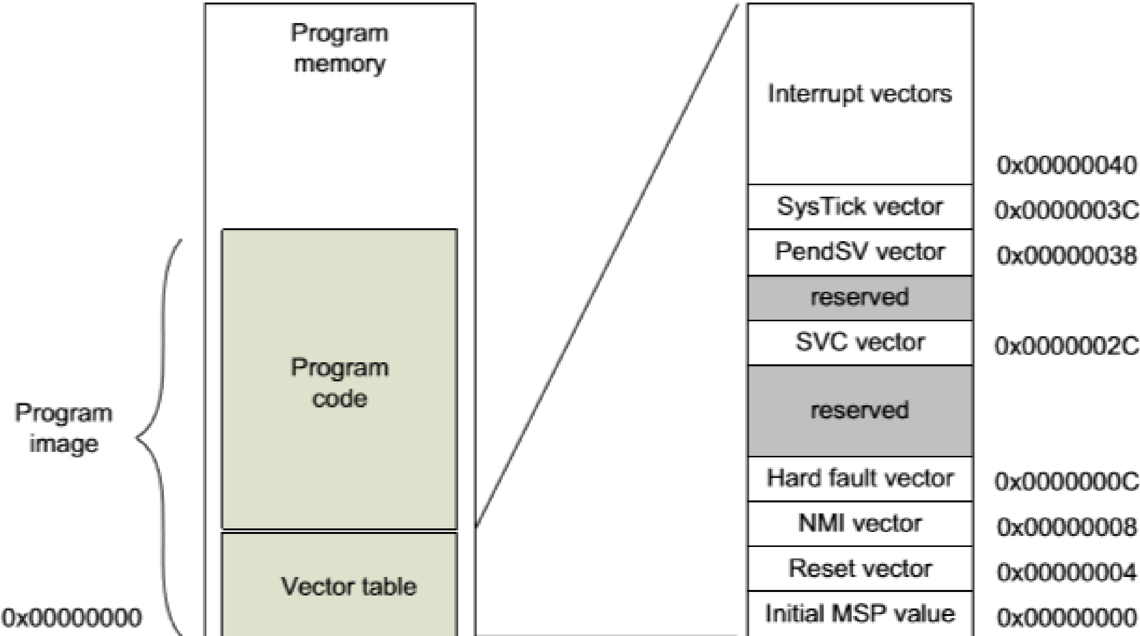


Figure 2.8- Program Memory content [13]

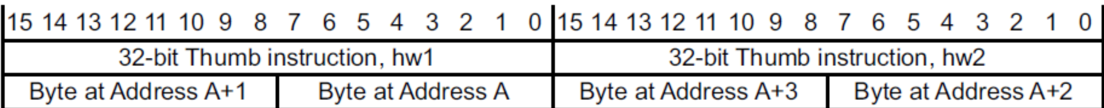
After the reset, processor first reads the first word from the memory and initializes MSP with that value. Next step is to read the second word that determines where the program starts and start fetching instructions from that address and so on in the next clock cycle. Vector table can usually be completely defined using C. This information (first two words from the vector table) are often contained in the so called startup code. The reset handler from startup code can initialize some general system features, variables and memory that are going to be used although this is usually done later in the *main()*. Startup code is usually found in development suites or in software packages from MCU vendors. After the startup code, application code is being executed and it usually contains, as stated in [13]: initialization of hardware (clock, PLL, peripherals), application processing part and interrupt service routines.

2.4.1.3 ARMv6-M instruction set

First ARM processors were using ARM 32b so called ARM Instruction Set which offered very high efficiency but occupied much more program memory space than other 8b or 16b processors. That is why with the announcement of ARM7TDMI processor, ARM

announced new 16b instruction set, called Thumb, but not completely denied use of ARM instruction set because it was not possible to implement some functionalities using only Thumb. This processor supported both ARM (32b) and Thumb (16b) instructions and used a state switching mechanism to choose which decoding scheme should be used (based on the so called T bit which was set to 1 in case it was a Thumb and to 0 in case it was an ARM instruction). Thumb instruction set creates a 30% smaller code when compared to the code with the same functionality but written using ARM instruction set. On the other hand, performance is deteriorated by 20%. This is why Thumb-2 instruction set was created: it contains some 32b instructions to perform the functionalities that could be done only by ARM set initially and all the 16b instructions from Thumb. The same functionality coded with Thumb-2 takes around 74% of the size of the code coded with ARM IS maintaining the same performance.

ARM and Thumb instructions are designed to interwork freely but ARMv6-M only supports Thumb instructions and this is why interworking instructions in ARMv6-M must only reference Thumb state execution. All instructions that were used to test hardware implementation in this thesis were 16b Thumb instructions which meant that no interworking was needed but if the reader wants to know more about it, it is described in detail in [14]. ARMv6-M, the architecture used in Cortex-M0, is mostly using 16b Thumb instructions and a minimum subset of essential 32b Thumb instructions: BL, DSB, DMB, ISB, MRS and MSR. The instruction fetches are always half-world aligned. ARMv6-M can be configured to use either little endian or big endian data interpretation, but in the case of Cortex-M0 DS, used in this project, only little endian was supported.



<table>
<tr><td colspan="2">15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td><td colspan="2">15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td></tr>
<tr><td colspan="2">32-bit Thumb instruction, hw1</td><td colspan="2">32-bit Thumb instruction, hw2</td></tr>
<tr><td>Byte at Address A+1</td><td>Byte at Address A</td><td>Byte at Address A+3</td><td>Byte at Address A+2</td></tr>
</table>

Figure A3-5 Instruction byte order in memory

Figure 2.9- Instruction byte order in memory [14]

ARMv6-M has 13 general purpose 32b registers (r0 - r12) and three special purpose 32b registers: SP (Stack Pointer, r13), LR (Link Register, r14) and PC (Program Counter, r15, loaded with reset handler when core resets) whose usage can be guessed from their names. There is also a register called APSR (Application Program Status Register) with least significant four bits used as flags: negative, zero, carry and overflow whose values are checked when conditional branching instructions are executed. In this section more attention will be

given to these instructions since they are the key instructions when a loop needs to be detected which was of crucial importance for this project.

Branching instructions supported by ARMv6-M are listed in Table 2.2. Since this project was focused on loop execution and not function or subroutine calls, only B and BX should be considered in details when implementing the algorithm described in Chapter 2.2. Chapter 3.2 explains two possible implementations of the same principle where in the first one encodings of these specific instructions would have to be used whereas in the second and actual implementation from this project, another approach was chosen where instructions encodings are not that relevant and that is why they are not mentioned in details in this section. More about instruction encodings can be found in [14].

Table 2.2 – Branching instructions supported by ARMv6-M [14]

Branch Instruction	Description	Range
B	**Branch to target address**	**+/-2KB**
BL	Call a subroutine	+/-16MB
BLX	Call a subroutine	Any
BX	**Branch to a target address**	**Any**

To demonstrate the execution of a loop, execution of a simple test program was simulated in VIVADO Design Suite - the signal waveforms are shown in Figure 2.10.

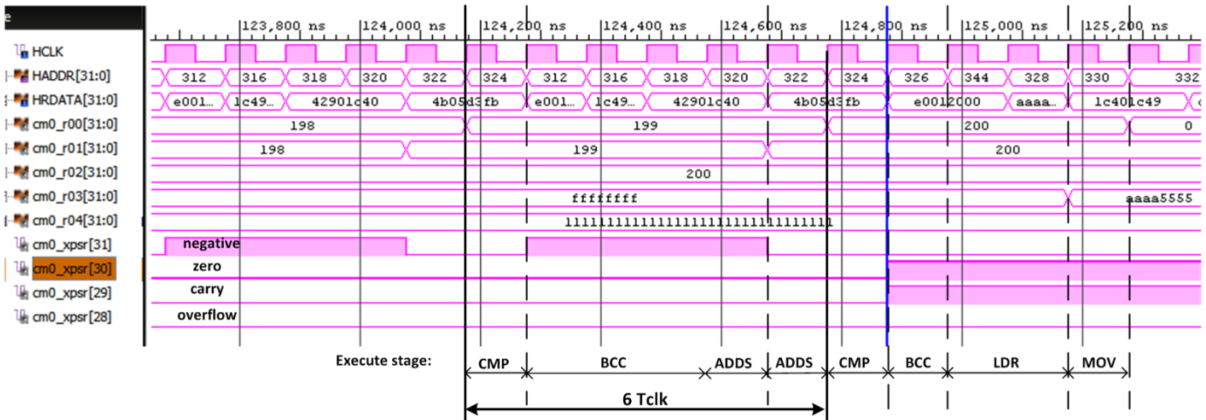


Figure 2.10- Simulation of the loop execution

Most of the time, the simulation shows execution of a loop as it can easily be noticed only by observing the values on HADDR bus: 312 – 316 – 318 – 320 – 322 – 324 and then going back to 312 and starting all over again. But the last part of the simulation shown in the Figure 2.10 shows the part where the loop breaks and the program continues to execute the rest of the code (that moment is marked by the blue line). Table 2.3 shows all the instructions that

are part of this particular loop: their address, hexadecimal encoding, corresponding assembly code, short description and the duration as reported in [12].

Table 2.3- Loop Instructions

Address	*Instruction (hex)*	*Assembly*	*Description*	*Execution Duration [CLK cycles]*
312 (6)	*E001*	*B<PC+2>*	*Unconditional branch 2 addresses forward*	*1*
	1C49	***ADDS R1,R1,#1***	*Increment the value of R1*	*1*
316 (7)	*1C40*	***ADDS R0,R0, #1***	*Increment the value of R0*	*1*
318	*4290*	***CMP R2,R0***	*Compare values in R2 and R0, set flags*	*1*

320 (0)	D3FB	**BCC<pc-12>**	Branch 10 (12) addresses backward if carry cleared	3
322	4B05	LDR R3, <pc+20>	Load value from the address pc+20 into register R3	2
324	2000	MOV R0,#0	Load zero into R0	1
	E001	B <pc+2>	Unconditional branch 2 addresses forward	1

At this point it is important to mention again that ARMv6-M architecture has a 3 stage pipeline: fetch, decode and execute and that the instruction duration depends on how many cycles it takes for it to execute (fetch and decode have fixed duration). The duration column in the Table 2.3 shows how many clock cycles the execution stage takes. As it can be seen from Figure 2.11, up to two 16b instructions are fetched in one transfer. In the next clock cycle first one of them is being decoded and in the third cycle this instruction enters the execution stage whereas the second fetched instruction is being decoded. At the same time, next two instructions can be fetched since last two left the fetch stage.

It can be seen from Figure 2.10 : in the case of a taken branch it takes 3 clock cycles to execute the branch instruction whereas only 1 in case of a non-taken branch, as stated in [12]. In the case of a taken branch some of the instructions are still fetched (those that are not in bold) but they are not executed since the architecture cleans the pipeline from them. In case of a non-

taken branch all of the instructions are executed as it can be seen in Figure 2.10 (e.g. register r3 is indeed loaded with a value read form the memory, register r0 loaded with zero etc.)

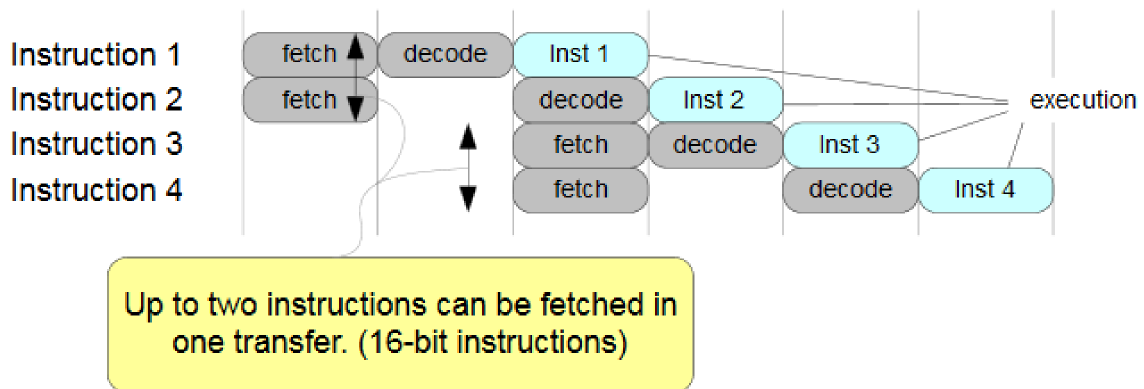


Figure 2.11- Cortex-M0 pipeline stages[15]

2.4.1.4 ARM Cortex-M0 Design Start

Design Start Implementation was delivered by ARM with two Verilog modules: top module called *CORTEXM0DS* and an obfuscated sub-module called *cortexm0ds_logic*. Top module implements ports for the AMBA 3 Lite Interface with possibility of 17 interrupt inputs (16 + NMI), three output status signals and one output event signal. The top module interface is shown in Figure 2.12 and it contains AHB interface signals, interrupt inputs, event input and event and status outputs.

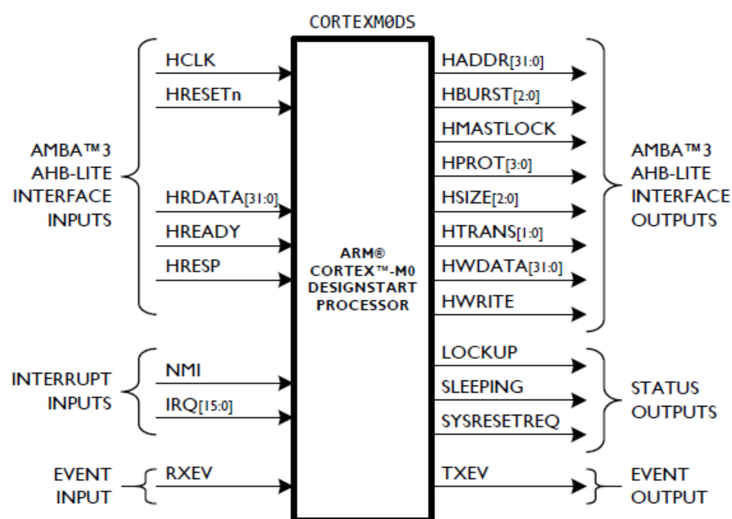


Figure 2.12- Cortex-M0 schematics[16]

All interrupt input signals are synchronous and active high, NMI has the highest priority whereas priorities of other 16 interrupt signals can be configured.

There are three output status signals and their meanings will be briefly mentioned here whereas more details can be found in [16]:

- SLEEPING: an active high signal that indicates the processor is IDLE (usually after Wait for Event - WFE or Wait for Instruction - WFI instructions) and will not execute any AHB transactions
- LOCKUP: indicates that the processor is in a non-desirable, LOCKUP state and
- SYSRESETREQ: HIGH value of this signal indicates that the software wants to perform system reset.

Event output signal TXEV is used to indicate that the processor is performing SEV instruction whereas the input RXEV signal is indicating that it should wake up from WFE instruction.

Cortex M0 Design Start implements AHB 3 Lite System Bus Interface using bus clock and reset signals as its global clock and reset signals: HCLK and HRESET. It is important to note that the Design Start implementation of the core does not offer all the possibilities offered by the full core. Here are some of the restrictions introduced that could not be influenced:

- 3b HBURST output signal value was always 000 which indicated that the processor supported only SINGLE type transfer (no BURST transfer),
- HMASTLOCK output signal is driven low all the time which indicates that the processor is not generating locked sequences.
- HPROT[1] signal is always driven low indicating that the transaction is always PRIVILEGED (program execution can access all memory resources)

Transaction	Access	Description
HTRANS[1:0] = 2'b00	IDLE	The processor does not wish to perform any transaction.
HTRANS[1:0] = 2'b10 HPROT[0] = 1'b0 HSIZE[1:0] = 2'b10 HWRITE = 1'b0	FETCH	The processor wishes to perform an instruction fetch. Instructions are fetched 32-bits at a time from memory. If required, the processor internally buffers and manages the extraction of two 16-bit instructions.
HTRANS[1:0] = 2'b10 HPROT[0] = 1'b1 HSIZE[1:0] = 2'b00	BYTE	The processor wishes to perform an 8-bit data access resulting from an LDRB, LDRSB or STRB instruction. Load instructions will drive the HWRITE signal LOW; store instructions will drive the HWRITE signal HIGH.
HTRANS[1:0] = 2'b10 HPROT[0] = 1'b1 HSIZE[1:0] = 2'b01	HALF-WORD	The processor wishes to perform a 16-bit data access resulting from an LDRH, LDRSH or STRH instruction. Load instructions will drive the HWRITE signal LOW; store instructions will drive the HWRITE signal HIGH.
HTRANS[1:0] = 2'b10 HPROT[0] = 1'b1 HSIZE[1:0] = 2'b10	WORD	The processor wishes to perform a 32-bit data access resulting from an LDR, LDM, POP, STR, STM or PUSH instruction, or as part of exception entry or return. Loads will drive the HWRITE signal LOW; stores will drive the HWRITE signal HIGH.

Figure 2.13 - Cortex-M0 access types[16]

Cortex-M0 core can generate four types of transfer depending on the signals HSIZE[1:0], HTRANS[1:0], HPROT[0] and HWRITE as shown in Figure 2.13.

The most interesting transfer type at this point is the one involving instruction fetch. As described earlier, the fetch involves fetching 32b at a time from the instruction memory with a possibility of extracting two 16b instructions from this sequence.

Although the design start does not offer possibility of hardware debug, there are some registers that are routed from internal logic so that their values could be observed while simulating. Those are 13 general purpose registers, MSP (main stack pointer), PSP (process stack pointer), link register, PC (address of instruction currently in execute), status register (XPSR), control and primask register (more about each one of them can be found in [14]).

2.4.2 AHB bus

AMBA AHB-Lite (Advanced Microcontroller Bus Architecture, Advanced High Performance Bus) is a standard on-chip intercommunication specification developed by ARM but widely used in high performance electronic designs with many different applications. It is based on a master – slave role division concept with one master (Lite suffix), although it can

be extended to include more masters, Multi-layer AHB. If there are slaves that are slower (using lower bandwidth), they are located on the APB (Advanced Peripheral Bus) part of the system

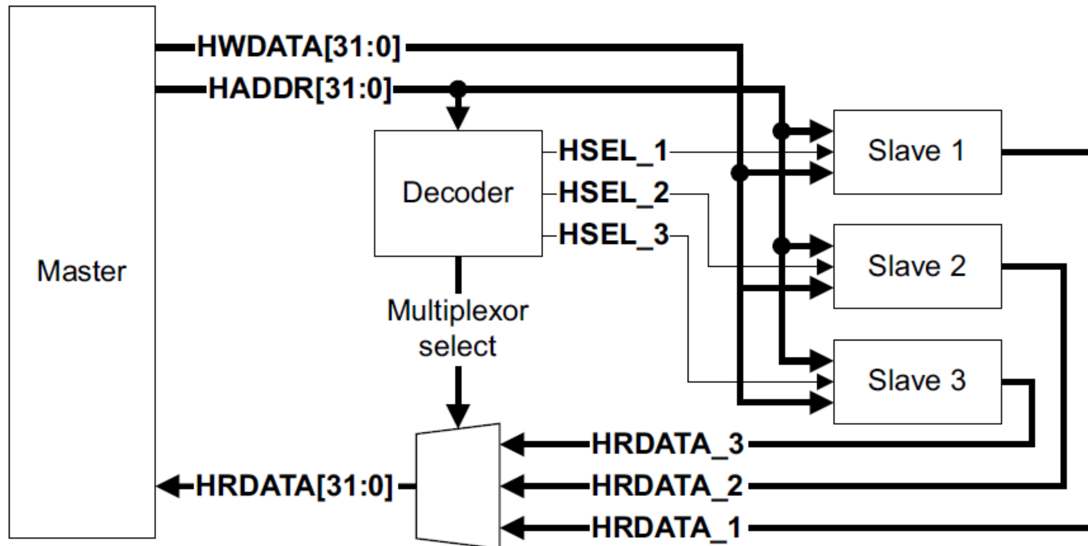


Figure 2.14 - AHB-Lite block diagram [17]

and connected to the AHB by a bus bridge.

Figure 2.14 shows AHB master with three slaves and two inevitable elements which determine which slave is going to communicate with the master: decoder and a multiplexer. Master starts both read and write processes by driving corresponding control signals and setting the address of the slave it wishes to communicate with, decoder decodes the address and generates both slave enable signals and a multiplexor selective input to pass corresponding output data from the slave into the master.

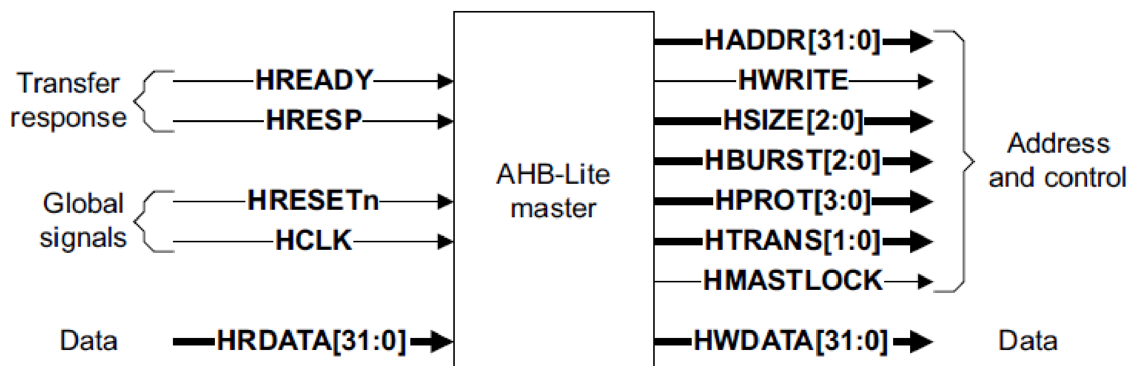


Figure 2.15- AHB Master interface

Figure 2.15 shows AHB master with its interface to the rest of the AHB system. Many of these signals already appeared and were described so in this part emphasis will be put on the situations that can happen in the case of using Cortex-M0 Design Start as a master: the transfer type can only be single, there can be no locked transfers, the transfer is always privileged and the transactions are always non sequential. Each of the output signals sets a type of control over the transfer - their role is shown in Table 2.4. Some of the transfer properties are shown in bold and those represent properties that were hard wired into the Design Start so the non-bolded transfers are not even possible to implement. More information and details about all possible transactions on the AHB bus can be found in [17].

Table 2.4- AHB Master output signals of interest

Signal	*Description*
HADDR	*Address phase interface address*
HWRITE	*Address phase read/write information*
HSIZE	*Data phase transaction data size:* *- byte* *- half-word* *- word*
HBURST	*Address phase burst information* *-**single*** *- incr* *- wrap [8 16 32]* *- incr [8 16 32]*
HPROT	*Protection information* *-cacheable,* *- bufferable,* *- **privileged** (has access to all resources)* *- **data/ opcode***
HTRANS *00 or 10*	*Address phase transfer type:* *- **idle*** *- busy* *- **non-sequential*** *- sequential*

HMASTLOCK	*Address phase locked information* - *locked* - ***unlocked***

AHB slave provides information about the status of the transaction by setting values of the two signals: HREADYOUT (when HIGH indicates a transfer has finished, but it can be driven LOW to extend the transfer) and HRESP (success or failure in the communication). Systems that were used in this project, consisted only of instruction memory and a small cache which were designed in a way they could provide new data on each new clock cycle if requested (the signal HREADY which normally would have been provided to the master by the slave, was driven HIGH all the time). Moreover, there was no possibility for slaves to generate errors so the signal HRESP was driven HIGH all the time as well.

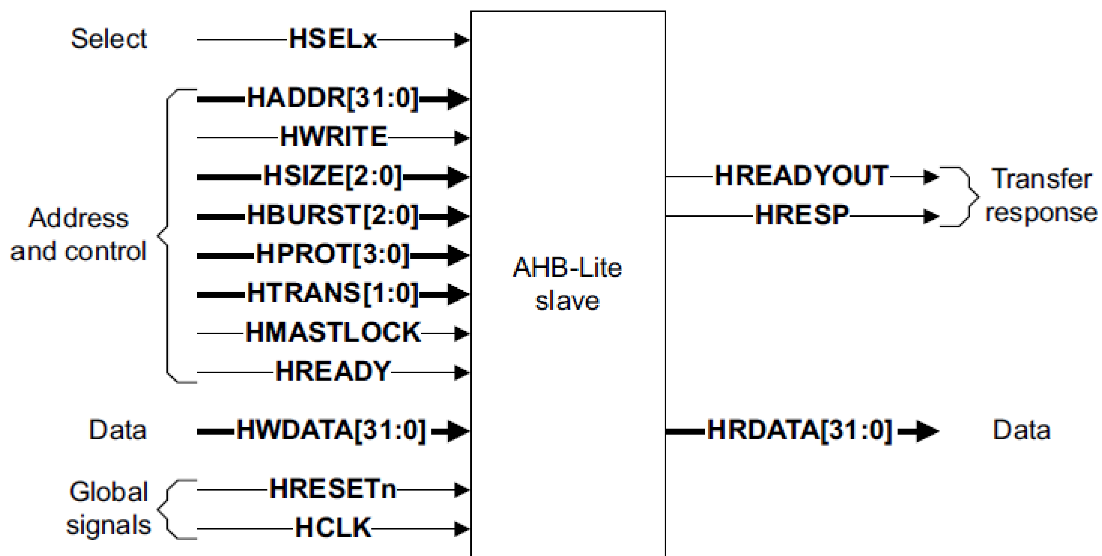
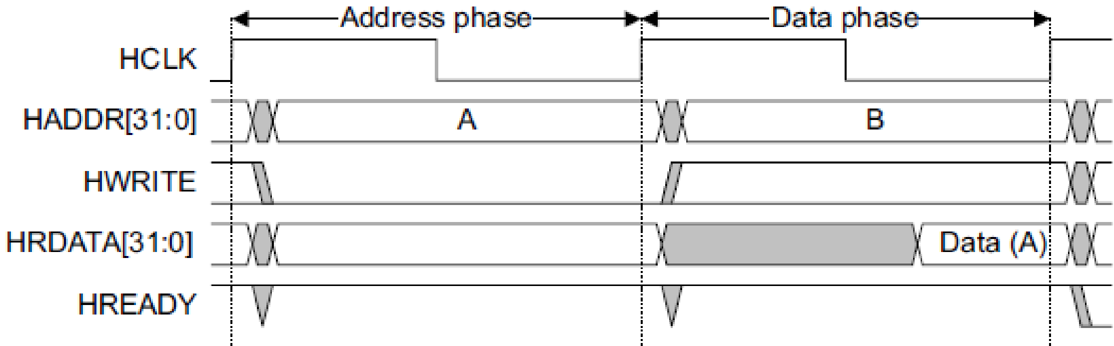


Figure 2.16- AHB Slave interface

Since Cortex-M0 DS could not perform any other but SINGLE transfer, slaves were not generating any errors and were ready to give new data on each clock cycles, the transfer that is going to be considered here is the basic unlocked single transfer with no wait states and no protection control.

AHB basic transfer consists of two phases: address and data phase and it takes three rising clock edges (two clock periods) for a transfer to complete. After the first rising edge, master sets up the right address and control signals (important signal that is not shown in these

figure is the HTRANS[1] signal which needs to be set HIGH at this point to indicate AHB transfer is about to happen). Signal HWRITE controls the direction of the transfer: if set HIGH the master broadcast data on the HWDATA bus and this data is about to be written into the slave, whereas in the case of this signal LOW the master wishes to read data from the slave and this data is going to appear on the HRDATA bus. On the next clock edge (second), the slave samples control and address signals. Finally on the third clock the processor is able to sample the correct data on the HRDATA bus or the slave is ready to sample HWDATA bus. Figure 2.17 also shows how address phase of a transfer overlaps with data phase of the previous transfer and this is the fundamental principle that makes AHB a high performance bus used by wide spectrum of applications.



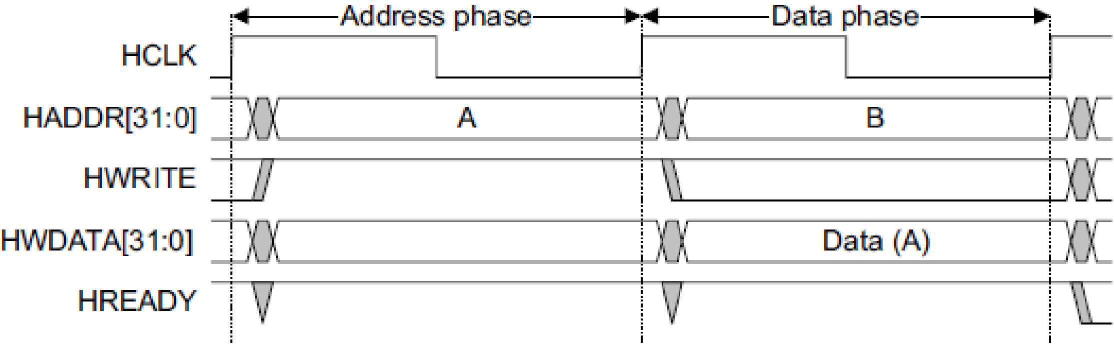


Figure 2.17 - AHB basic transfer: read and write [17]

2.4.3 Simulating the Cortex-M0 core

As it was mentioned earlier, Cortex-M0 Design Start soft core was obtained from ARM as two modules written in Verilog:

- CORTEXM0DS.v (a top designer module which shows processor interface) and
- cortexm0ds_logic.v (an obfuscated module with processor logic implemented).

Apart from the hardware implementation contained in these two files there was also a testbench provided with a top testing Verilog module which had an AHB system containing a master (the core itself) and two slaves:

- a memory model to load a binary of a program to be executed and
- an output console to show the status and results of the running program.

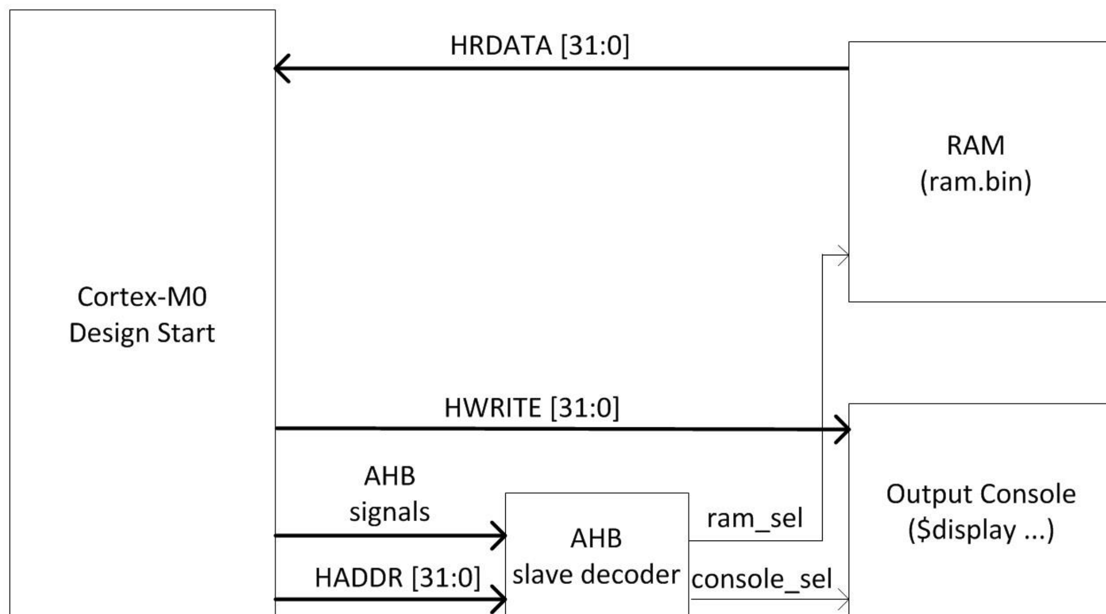


Figure 2.18 – ARM Design Start testbench deliverables: block diagram

Memory model implemented as a part of testbench was word addressable with an 18b wide address bus so the HADDR values that were selecting it were between zero and 2^18 (0x40000) whereas the address that was accessing output console was set to 0x40000000. The binary that was delivered with the testbench (ram.bin) was a binary obtained by compilation of the helloworld.c (also delivered) using ARM's Real View Complier using commands from the Makefile from the package as well. To make the processor execute this program, it was enough only to create a project in any Verilog simulator and make sure that the binary is accessible by

the core (it should be within the simulation folder of the project, otherwise there would be a mistake reported). In this thesis, verifying simulation was performed using Xilinx VIVADO. The output of the simulation were merely messages on the screen showing the status of the program execution, Figure 2.19. This simulation was performed only to prove that the core was behaving the way expected and that it was possible to load different binaries into it. In the next sub chapter it will be explained how the core was synthesized and how the code was being run on an actual FPGA board.

```
Hello World!
This message was printed from helloworld.c provided
with the ARM Cortex-M0 DesignStart processor
          436050000: Simulation stop requested by CPU

$finish called at time : 436050 ns : File "C:/Users/Dell/Desktop/core_simulation_ARM/project_1.srcs/
Memory in use : 48204 KB (peak memory: 48204 KB)     CPU usage : 73405 ms
run: Time (s): cpu = 00:00:58 ; elapsed = 00:00:50 . Memory (MB): peak = 1717.004 ; gain = 237.594
```

Figure 2.19 – Core simulation results

2.4.4 Running the core with Instruction Memory on FPGA [18]

Following ideas from [18] with slight modifications, a system for synthesis on ZC702 Evaluation Kit was developed. To develop the whole system which would be able to execute different programs, simulate processor behavior showing signal waveforms and finally verify the correctness by running the core loaded with a program on actual hardware, few system integration steps needed to be performed.

2.4.4.1 Building hardware

The system block diagram is shown in Figure 2.20and the deliverables that came within this tutorial contained these modules. All the blocks were used in the same way as in the manual form Louisianan Tech College of Engineering and Science [18] with some slight differences since in the tutorial the implementation was adapted to NEXYS2 FPGA board.

As it was stated earlier, ZC702 Evaluation Kit contains a Zynq-7000 SoC with a Xilinx Artix 7 FPGA Programmable Logic Equivalent System (part number XC7Z020-1CLG484C) that needed to be selected as a target device when creating a project in VIVADO Design Suite. Next step was to add sources to the project which will represent hardware modules. Figure 2.20

shows the overall system (top module, CM0_DSSystem.v file) with hardware modules that were part of it.

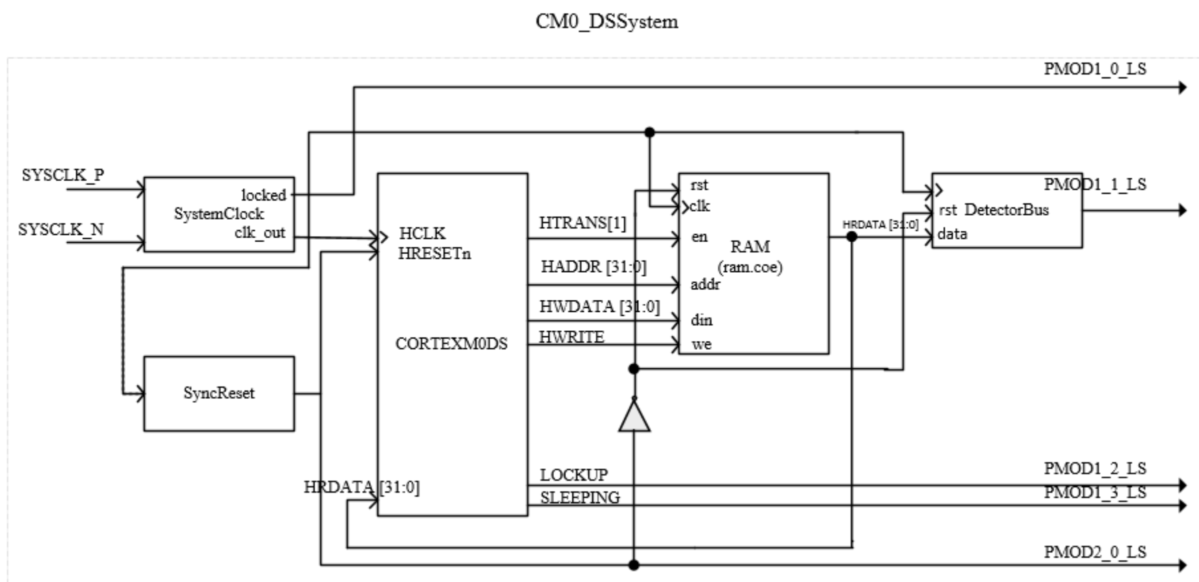


Figure 2.20 – Block diagram of the system built for implementation on FPGA board

2.4.4.1.1 Clock generation

The most important part of a synchronous system is the clock: its generation has to be stable and accurate in order for the system to behave as planned. Although the Cortex-M0 processor is said to operate up to 50MHz frequency, in the tutorial [18] system clock was set to 10 MHz and it was decided to maintain the value in this project as well.

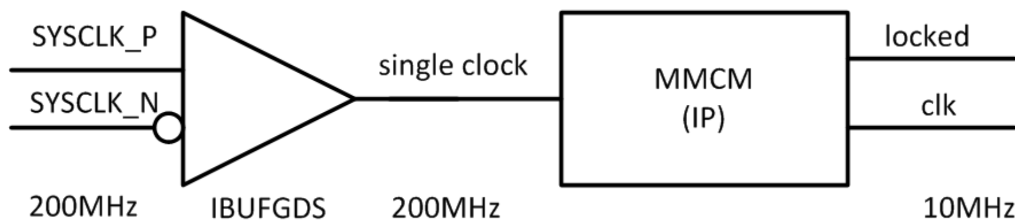


Figure 2.21 - Clock generation on zc702 Evaluation Kit

Clock module was created using Clock Wizard IP generator in VIVADO with the implementation using MMCM (Mixed Mode Clock Manager) and not DCM (Digital Clock Manager) as in [18] since Xilinx 7 Series do not offer DCM any more, only MMCM and PLLs. The EK had a 200 MHz Low Voltage Differential Signalling (LVDS) oscillator with a

differential output: a positive SYSCLK_P and a negative SYSCLK_N. Clock generator that was used to synthesize 10MHz frequency contained a single clock source as input and therefore these two signals had to be adapted to be used as a single clock signal. For these purposes, a primitive design element from 7Series FPGA, differential signal input buffer IBUFGDS was used as a connection to MMCM.

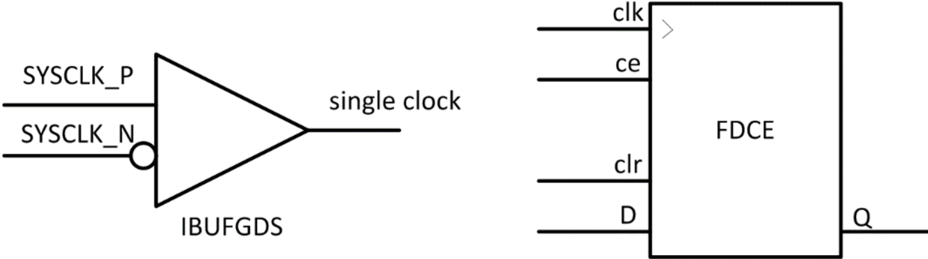


Figure 2.22 – 7Series primitive elements: buffer and D flipflop

Two input signals (SYSCLK_P and SYSCLK_N, the board oscillator differential outputs) have to have the same frequency but opposite phase. In this case output of the clock buffer was equal to the master input which is SYSCLK_P in this case. Single output of the buffer (200 MHz) is then loaded into the frequency synthesizer that gave the desired processing frequency of 10 MHz. As in the case of any frequency synthesizer, there was a locked signal at the output that was indicating if the synthesizer was locked and the output frequency was stable. This signal was also used as an output of the system to drive board LEDs and indicate if the system had a stable clock which is always the first thing to check when performing system debug. Buffer was also used to recover the signal, make rise and fall time smaller.

2.4.4.1.2 Reset generation

Second most important step was to design reset circuitry for the system, in this case provide a 4 period long low signal for each reset session since that is the reset specification for the Cortex-M0 core. There were three modules from the tutorial [18] that were used to create the clock: DelayCounter.v , Constant2Pulse.v and SyncReset.v (for the sake of simplicity, Figure 2.23shows only the top module). As it was mentioned earlier, reset pulse needed to be at least four cycles long in order to reset the core and in this implementation from [18] it was generated only once at the beginning of the system setup and completely internally, i.e. with no user controlled reset. The idea was to generate a pulse that would surely stay high for four clock

cycles and then invert it as shown in Figure 2.23. In order to make the signal four cycles long a short pulse (called trigger in figures) was propagated through four D flip flops and then outputs from all of them together with the original signal were used as inputs of a 5 input OR gate. FDCE elements in the figures are primitive design elements supported by Xilinx 7Series [19] which are nothing but D flip flops with clock enable and asynchronous active high reset.

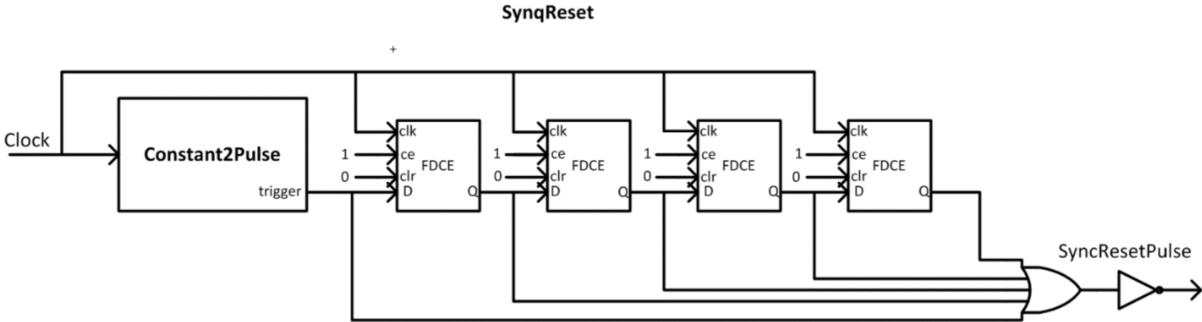


Figure 2.23 – SynqReset module block diagram

In order to generate trigger signal, a module Constant2Pulse was used. Its implementation was based on the idea that performing an XOR operation on a constant of a signal and its delayed version would produce a pulse at the output. This is shown in Figure 2.24.

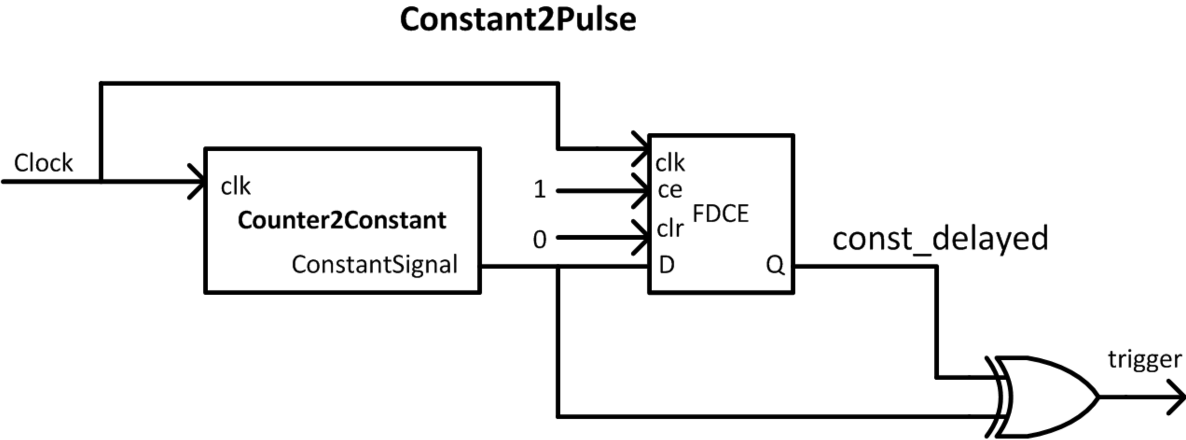


Figure 2.24 – Constant2Pulse module block diagram

Generation of the constant signal was done within the module called Counter2Constant which is shown in Figure 2.25. The basic idea was to generate a pulse using a binary counter, and once it was generated, use a positive feedback to maintain at least one of the inputs of the OR gate high which created a constant output signal. These signals are shown in Figure 2.25.

Although having a 20b output, binary counter was configured to count only up to the value F9 and then generate a high signal called *thresh* which was used as input to the OR gate, whereas the 20b output was not used later at all. Simulation of the reset signal generation is shown in Figure 2.26.

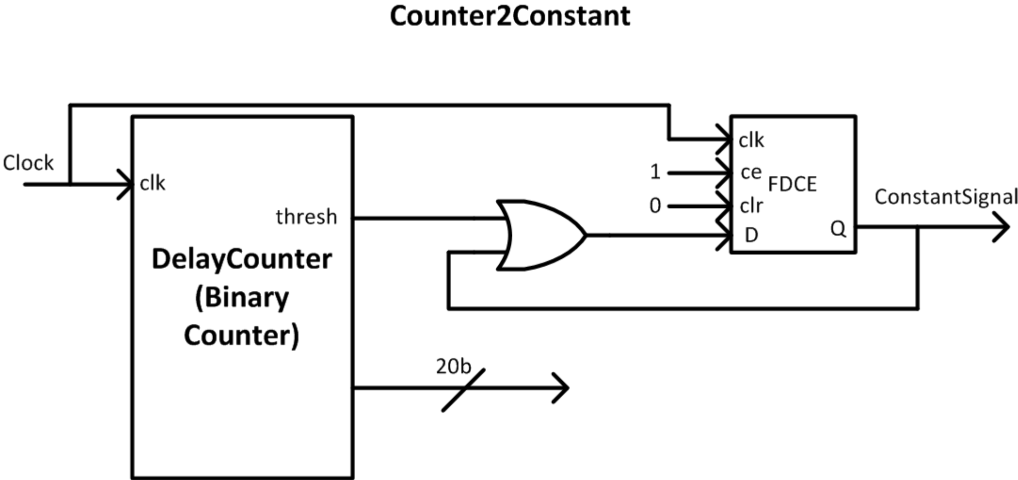


Figure 2.25 - Counter2Constant module block diagram

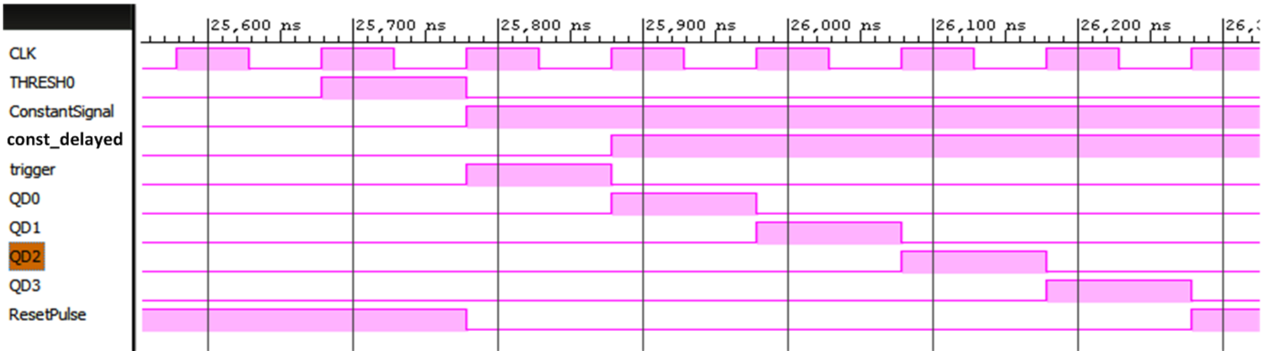


Figure 2.26 – Reset pulse generation: simulation

2.4.4.1.3 RAM memory

RAM memory was created using IP Block Generator from VIVADO that created 2KB memory which was word addressable, having 9b wide address bus (2^9 x 4B= 2KB) and an active high reset. The memory was designed to work in compliance with AHB interface, i.e. if enabled, on the next rising clock edge output data from the address that was sampled with the current clock edge, was present on the bus as shown in Figure 2.27. Enable signal of the memory

block was connected to HTRANS signal of the AHB bus which was completely understandable since this signal driven high indicated that a new transaction should be performed. This was explained in details in Chapter 2.4.2.

Memory IP block offers the possibility to load memory with initial content saved in a file with an extension .coe. Generation of this file will be explained in Chapter 2.4.4.

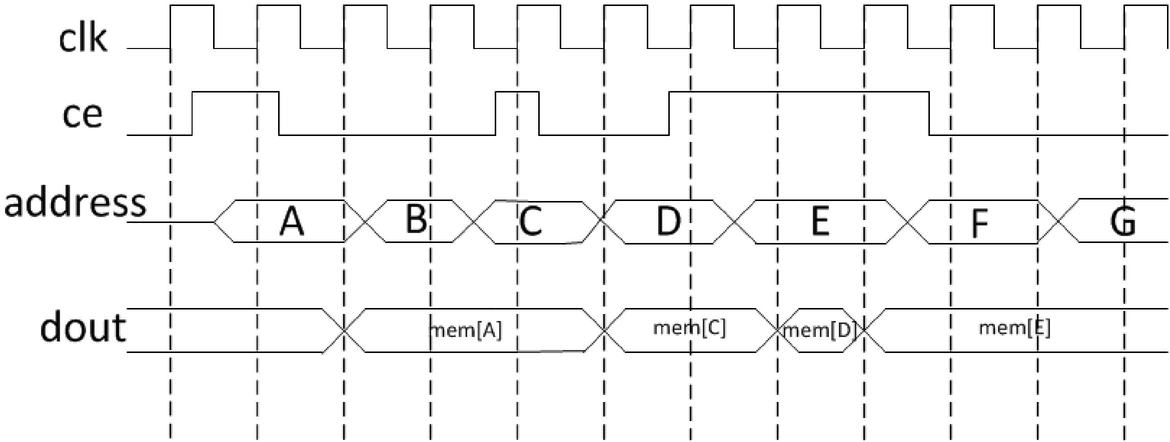


Figure 2.27 – Block RAM principle of operation

2.4.4.1.4 Detector

The last part of the system in [18] was the module called Detector that was used to indicate that the processor was executing the loaded program the way planned. It is not that easy to verify proper functioning of a processor core when it does not have peripherals built around it because there are many signals and wide buses that need to be checked. A very simple approach was used in [18]: module Detector was toggling an LED at the output when a certain value would appear on the data input bus, HRDATA. At the beginning, on a system reset, the LED was driven low. Later on, when the value 0xAAAA5555 showed up on the bus, the Detector signal would go high and remain high until value 0xF0F0F0F0 showed up. To sum up, HRDATA value of 0xAAAA5555 was triggering Detector value to go high whereas the value of 0xF0F0F0F0 was driving Detector low. This is shown in Figure 2.29.

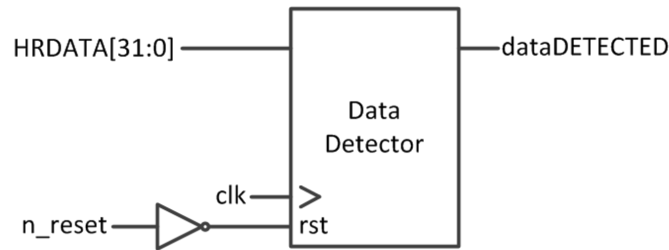


Figure 2.28 – Detector module interface

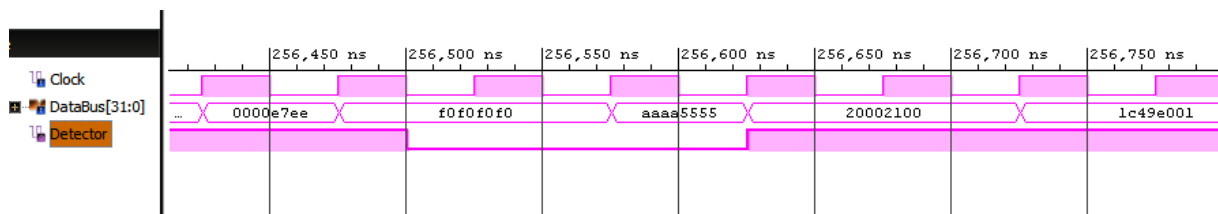


Figure 2.29 – Generation of Detector signal: simulation

The last step to configure the hardware part of the system was to add the Xilinx Design Constraint file (.xdc) to the project constraints. This file was taken over from [20] containing net names that corresponded to net names of the latest ZC702 schematics (although only some of them were used in the project).

2.4.4.2 Software

Previous section explained how hardware modules were designed, what was the purpose of each one of them and how they communicated between themselves. Another important aspect of enabling system to work is making it behave in the desired way so that processes could be performed. This is done, as in any other system, by loading a desired program, which was a .coe file in this case into its instruction memory.

µVision IDE from Keil was used to develop software project, make source file editing and program debugging with Cortex-M0 as a target core. Other project properties that needed to be set as in [21] were:

1. Target tab
- Working frequency: 10MHz
- Set ROM memory space (size 1KB, starting at zero address)
- Set RAM memory space (size 1KB, starting at address 0x400)
2. Output tab:

- Select folder which will contain object files
- Enable the option of creating .hex file when compiling

3. ASM tab:

- Choose Thumb Mode

4. Linker tab:

- Set the ROM and RAM base addresses, 0x00000000 and 0x00000400
- Set –entry 0x15 –first=vectors.o(__Vectors) as MiscControls to locate the vector table defined by vectors.c

5. Debug tab:

-make sure to tick Use Simulator for debugging purposes

The simple program used for testing the system was completely taken over from [18] and the *main.c* is shown in Figure 2.30. As it was seen in Figure 2.8, program code was starting at zero address, but the whole memory space needed to be divided in order to make space for heap and stack. This was done within the file called *vectors.c* which contained stack base address (set to 0x47f in this case) and the address of the reset handler only since no other interrupts were implemented in the system.

As it can be seen in Figure 2.8, the top of the instruction memory was set to be in the middle of the whole RAM space (0x400 is 1KB and the memory had 2KB), whereas the rest was left for other essential memory sections which were not going to be used in a case of a simple program as this one but needed to be defined.

There were four variables defined in the memory:

- period: loaded with a constant value which could have had two values depending on how the system was going to be implemented. If it was intended for the system to be simulated only in HDL simulator, in order to be able to see a Detector change in a reasonably short simulation window, the period was set to a smaller value (200) whereas in the case of the system being implemented on the board, in real time, in order for the human eye to detect a change in LED state, the period had to be set to a higher value to make the blinking slower (20000000 in this case).

- counter: a variable that was being incremented on every clock cycle

- ii: a variable that was incrementing from 0 to 200 and then resetting to zero again

-trap: a variable that was taking certain values (0xAAAA5555 and 0xF0F0F0F0) when the end of period was reached.

```c
// Define where the top of memory is.
#define TOP_OF_RAM 0x4000U
// Define heap starts...
#define HEAP_BASE 0x407fU
#define LedOn    0xaaaa5555
#define LedOff   0xf0f0f0f0

int main(void)
{
  unsigned int counter;  // dummy
  unsigned int ii;      // loop iterator
  unsigned int trap;     // memory access pattern receiver
  unsigned int period;   // time interval for memory access


  //period=20000000;  // roughly 3 seconds for a 10MHz osc in CM0_DS
  period=200;  // period for simulations

  while (1)
  {
    counter=0;
    for (ii=0;ii<period;ii++)
    {
      counter++;
    }
    trap=LedOn; // memory access pattern (turn on)
    for (ii=0;ii<period;ii++)
    {
      counter++;
    }
    trap=LedOff;   // memory access pattern (turn off)
    trap++;    // dummy
  }
}
```

Figure 2.30 – Source code used in [18]

Table 2.5 shows these variables and their equivalents within the core registers.

Table 2.5 – C code variables and corresponding core registers

C unsigned int variable	Core register
counter	R1
ii	R0
trap	R3
period	R2

Using ARM C/C++ Compiler (former ARM RealView Compiler) within uVision the C code was translated into *.axf* file which is an object file with both object code and debug information. To transform this file into a binary, *fromelf* command needed to be used whereas the translation of .bin into *.coe* file was performed by an application that was delivered together with the integrating system [18], bin2coe. This is shown in Figure 2.31.

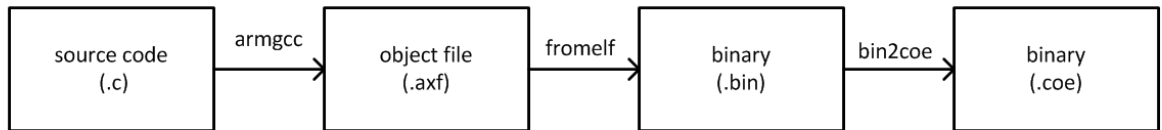


Figure 2.31 – Translation from .c file to .coe file

The resulting *.coe* file was finally loaded into the RAM. The whole system was synthesized, placed and routed, *.bit* file was generated successfully and the FPGA board programmed to blink the corresponding LED each 10s.

2.4.5 System Design Flow and Power Optimization

Typical design flow from Specification formulation to actual hardware that is handling the algorithm consists of System specification, Architectural Design, RTL design, Synthesis and Place and Route as shown in Figure 2.32.

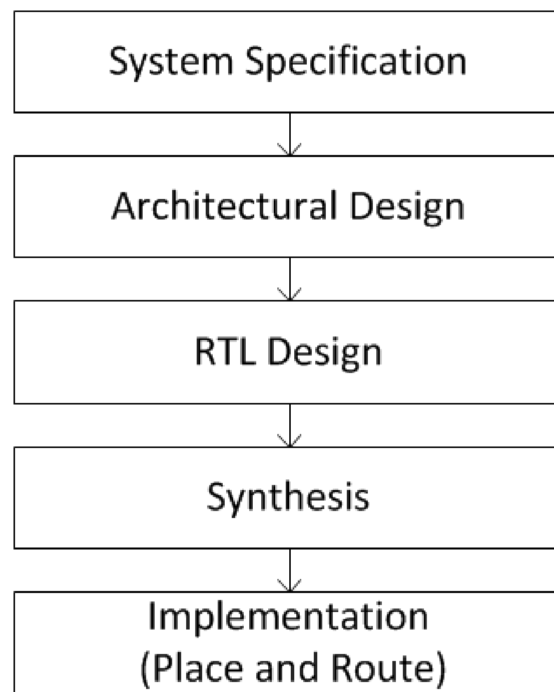


Figure 2.32 – IC Design general flow

The amount of power savings than can be achieved is getting lower as moving through the design steps. As the design is moving through different stages it gets harder to detect mistakes

and even harder to correct them. That is why proper system verification needs to be performed after each level to make sure no mistake flows further into the design
While first three steps are fully completed by the designer, last two are performed by EDA (Electronic Design Automation) tools but can still be controlled by the designer of course.

System specifications of the system were presented in Chapter 2.2. It is important to transform these abstract specifications into actual architectural design that is going to behave ain compliance with system specifications. Apart from this, other system level decisions, according to [22] include decisions about:

- cooling strategies (designing environment that is going to absorb as much power dissipated as possible by lowering ambient temperature, increasing airflow etc.)
- supply strategies (using switching regulators instead of linear ones, select regulators with high tolerances etc.)
- device selection (choose smallest possible device with largest package, smallest static consumption).

In this project, ZC702 Evaluation Kit and the Zynq device on it were selected because they offered a possibility of real time power measurements as it will be explained in the next section. Not too much attention was paid to the fact that the technology used to build the kit was 28nm and therefore dissipating a lot of static power that could not be greatly influenced.

As it will be mentioned, there were two possible architectural designs of the same TLC algorithm: using either *decode* or *address compare* controller, which would both fulfill the algorithm specifications but in two different ways. Section 3.2.1 will explain which one was chosen and why.

RTL level included HDL design of the system in order to meet system specifications. In the case of this project, these included different designs of the *sequential decoder* which will be explained in Chapter 3.2.1 and led to different utilization and power reports. Finally one of the RTL designs was chosen as the final one (before that it was checked by post-layout simulation that it still met all the system specifications).

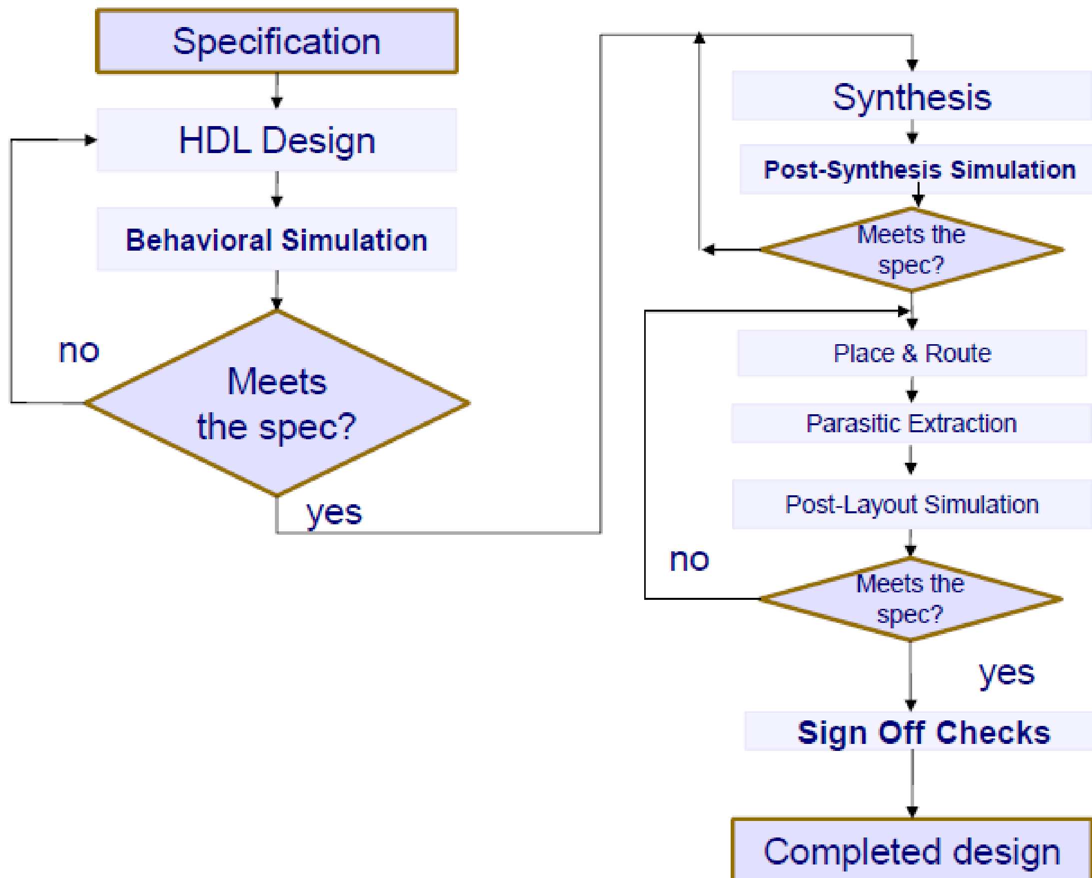


Figure 2.33 – Digital Design Flow

Figure 2.33 shows more details about the part of the flow that is supported by the EDA tools, in this case synthesis and implementation tools that are part of the VIVADO Design Suite. As this project was power optimization oriented, several different RTL designs of Controller and Loop Cache were taken all the way through Synthesis and Place and Route to determine which one of them brought most power savings. Based on the results of the Utilization Report that were shown in Table 3.5, the design using least logic elements was chosen as the final one. This, *basic* synthesis and place and route were conducted using default settings which meant there was no particular attention paid to power optimization during these processes.
Nevertheless when the final HDL design was chosen as output from the RTL design stage, VIVADO power analysis and optimization documentation [22, 23] were studied to determine how power consumption can be further influenced at these levels.

Synthesis is the process performed by a synthesis tools which translates HDL into hardware presentation and at the same optimizes the design following set of constraints. The result coming out of the synthesis process is the netlist which is a bond between HDL and final

routed hardware. Each time performed, synthesis can give different netlist and every time a small change is made, it should be verified by simulation that the netlist is still fulfilling constraints and specifications. In this project the only set of constraints that were applied were referring to constraining clock period of the 10 MHz clock.
One of the optimization techniques used in this project at the synthesis level was the flattening design which meant that the whole system could be regarded as one module although it was organized as a hierarchy. Allowing for the system to be regarded as one module makes it easier for the tool to perform optimizations although it makes it harder to simulate and analyze.

Timing parameters that are reported and have to be checked after synthesis are TNS (total negative slack) and WNS (worst negative slack). Slack time is defined as the difference between the time needed for the clock to propagate through clock path and the time needed for data to propagate through data path. System meets time constraints if the slack time is positive. WNS is the smallest slack (signed value) whereas TNS is the sum of all negative slacks. Both of these values should be positive in order for the design to meet timing constraints.

Both power and timing reports can be generated after synthesis but these values are not exact measurements since there is place and route yet to be performed. Power reports can be generated in a so called vectorless mode or using a switching activity interchange format (.saif) file. One of the main contributors to the overall dynamic power consumption are the switching activities of the nets: the more switching the greater the power consumed. Vectorless power report is done by using a default static probability and switching activity for all the nodes (12.5% in the case of VIVADO) which brings low confidence into the results whereas using a .saif file with switching activities calculated from actual simulations of the real conditions gives the report high result confidence. It is possible to provide toggling rates only for some nodes and leave the rest as default. All different measuring settings and results will be presented in Chapter 4.

Implementation or Place and Route step follows the Synthesis step and it includes actual placing of electronic components and logic elements into the FPGA cells as well as interconnecting them obeying all design rules applicable to the process of the target device. As far as VIVADO is concerned, it offers power optimization at two stages:

- default optimizations during the opt_design phase which focus on power savings in Block RAMs,
- power optimization during power_opt_design phase

All implementation and place and route processes in this project were performed with these options enabled.

3 Tight Loop Cache Implementation

3.1 TLC in software

Although the original description of the TLC technique contained implementation of the Controller as a state machine with three states, following [28] it was decided to add a COUNTER state as an intermediary state between IDLE and FILL. The Controller entered this state when a backward branch was detected but the loop instructions were still not written into the cache. The reason why this state was added was to reduce number of writes into the cache by going once again through the loop still reading from main memory but at the same time counting number of loop instructions, checking if they were sequential inside the loop itself and checking if the program loop could for sure physically completely fit inside the loop cache.

Apart from adding the COUNTER state, another modification was introduced after it was noticed there was non-sequential behaviour of the program inside the loop, Figure 2.5. As it was explained earlier, these were data reads/writes and normally the Controller should be able to distinguish between them and program jumps. Data transactions should not cause Controller leave the state it is in. Since there was no other information available apart from the addresses of the instructions from the execution flow, the way a data transaction was detected was by checking if it was “lonely” in the flow, meaning that after the data transaction program execution went back to where it was before. The way of distinguishing between data transactions and program jumps is shown in Figure 3.1.

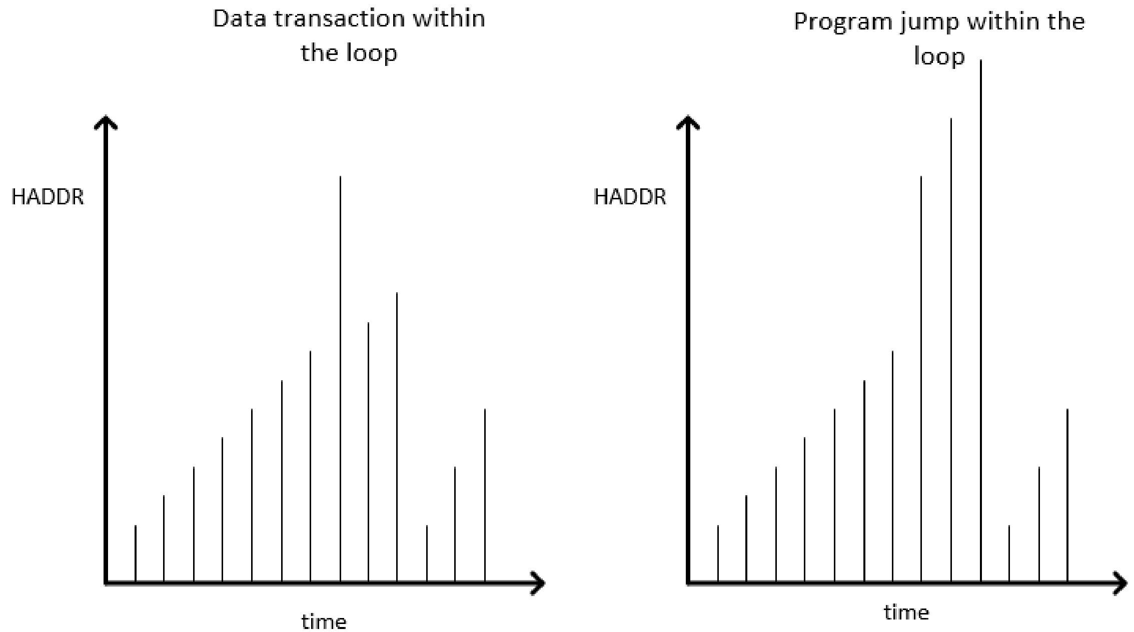


Figure 3.1 – Difference between data transaction and program jump with instruction addresses as inputs

Since the original principle of operation [10] considered only the loops sequential within themselves (those having data transactions are as well), new states had to be introduced inside the Controller to make it aware of existence of the data transaction while being in any of the states. The new implementation contained 8 states in total: IDLE, PSEUDO_IDLE, COUNTER, PSEUDO_COUNTER, FILL, PSEUDO_FILL, ACTIVE and PSEUDO_ACTIVE. Figure 3.2 shows state transitions only for the case of FILL state but all the other states have similar mechanisms of switching between states.

Important thing to notice is that it was possible now for the Controller to go into ACTIVE state directly from FILL or from the PSEUDO_FILL in case a triggering sbb was detected and taken while the Controller was in this state. To make it possible to keep the notion of the instruction flow, in this new implementation it was necessary to hold information about not only currently executed instruction but the last two instructions before it as well (variables *new, old* and *old_old*, Appendix A). This was also one of the differences between the implementation in [2] and the new one. In case it was detected that the jump which caused switching into any of the PSEUDO states was not a data transaction but an actual program jump, the Controller was going back into IDLE state.

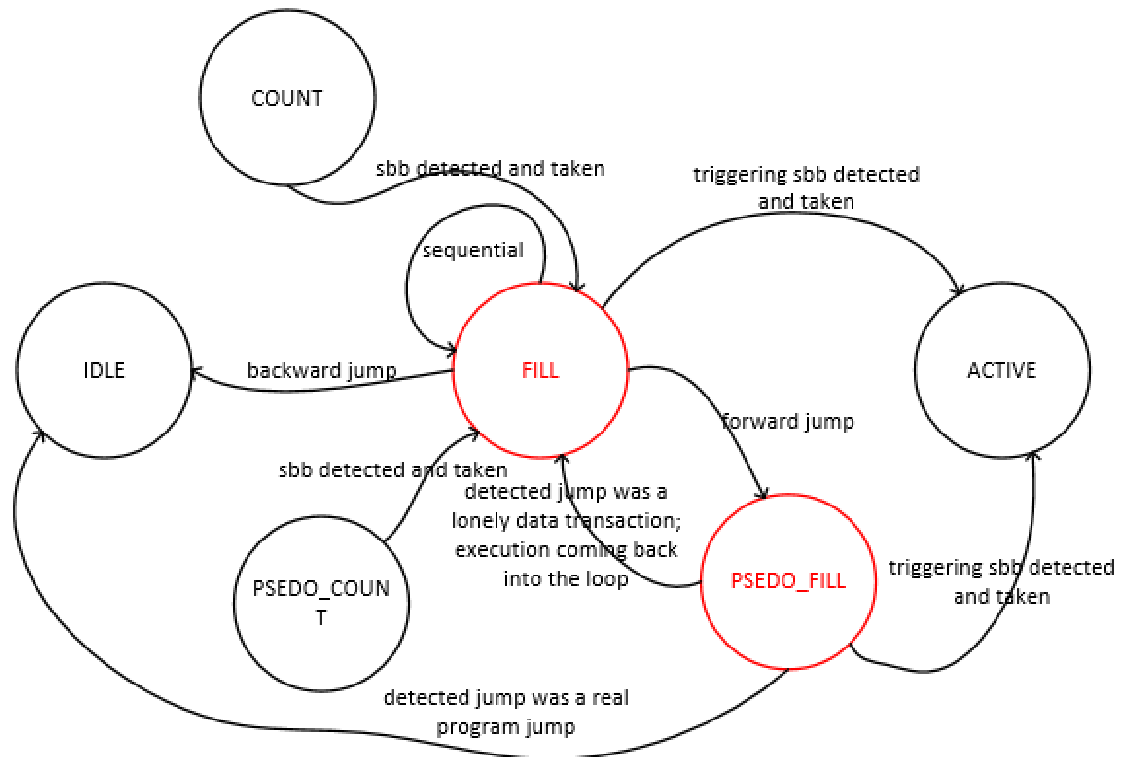


Figure 3.2 – Insertion of a pseudo state for data transaction detection

Hardware Implementation of the design included a Controller that had only three possible states since the core used in the implementation had a signal called HPROT which carried information if the transaction was involving data or code which made the solution save a lot of additional hardware. The final simulator in Python is listed in Appendix A.

3.2 TLC in hardware

This section shows how TLC system was built using ARM deliverables, modified software and hardware codes from [18] and additional hardware modules, created by the author.

3.2.1 Adding Loop Cache

After it was verified that the processor system was behaving in a correct way both in simulation and after synthesis (blinking LED on the board) it was possible to continue working on the actual topic of the thesis: the addition of small cache to facilitate the execution of program loops.

3.2.1.1 System user interface

As a difference to the system described until now, there were two ways the user could control the system:

- user controlled RESET implemented as a taster on the ZC702 board

- user controlled configuration of the system: enable or disable use of cache in general, implemented as a switch on the ZC702 board

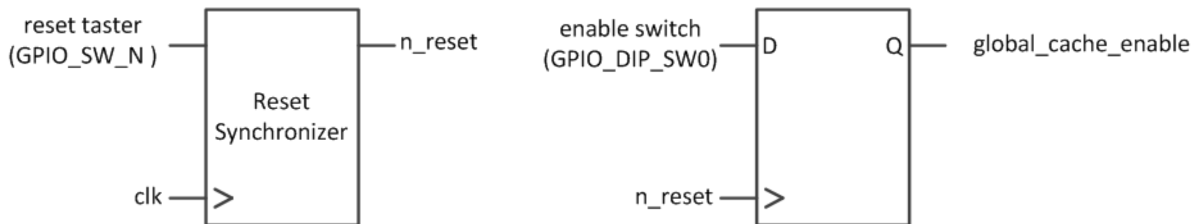


Figure 3.3 – Interface between the system and the user

These two signals and their interface to the system are shown in Figure 3.3. Reset was no longer generated internally, but controlled by the user so two modules: Counter2Constant and Constant2Pulse were not needed any more. Only SynqReset module was still used with a difference that triggering reset signal was the reset taster, user pushbutton SW5 [20] (named GPIO_SW_N in the board .xdc file), Figure 2.23. On the other hand, there was a switch on the board, SW12 [20] (GPIO_DIP_SW0 from .xdc file) that was used to control the global use of cache, i.e. whether the Loop Cache Controller was enabled to consider using cache at all or not. The value of this switch was sampled only when the system was coming out of the reset (flip flop was triggered by rising edge of the n_reset signal) and could not be changed at any other moment, Figure 3.4.

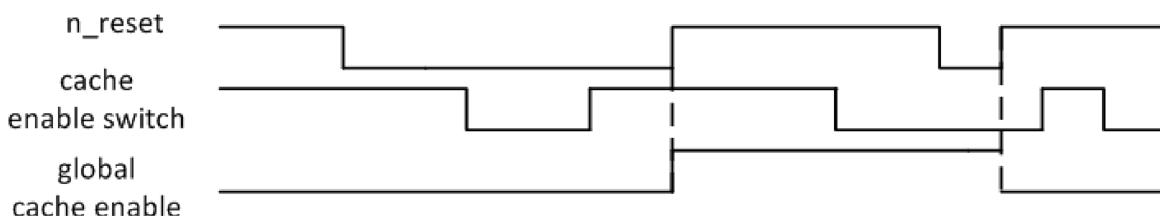


Figure 3.4 – Generation of global_cache_enable signal

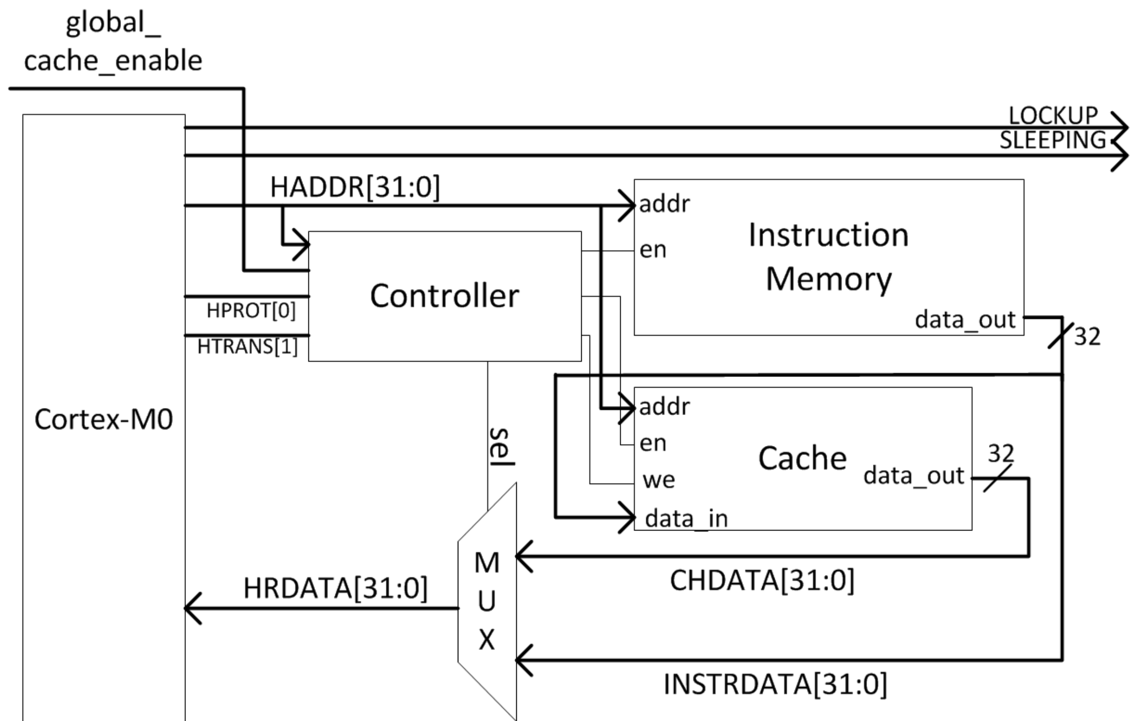


Figure 3.5 – Introducing Loop cache into the core system

The rest of the system, apart from user interface that has just been explained and the Detector module that remained the same as in the case when there was no cache, is shown in Figure 2.28. The system was implemented as an AHB bus system that contained one master: the processor core, two slaves: main instruction memory and the cache memory and a decoder (Controller) to decide which slave to access. The system was not a typical master-slave system since the nature of the communication between the Instruction Memory and the core is based on a continuous communication, the instruction memory was never written to (HWRITE was set low all the time) and therefore was ready to give data whenever the processor made a request (HREADY was high all the time). Another thing that was specific for this system was the period when the Controller was in the FILL state: both slaves were accessed at the same time: main memory was read from and that same data was written into the cache. Controller was also making decision which memory would output instructions into the core by controlling the multiplexer. So, this system had more differences than similarities to a typical AHB master slave system and in the case of need of adding more slaves to the system they would need to have their own decoder and multiplexer of course and respect the principle of continuous communication between the memory system and the core.

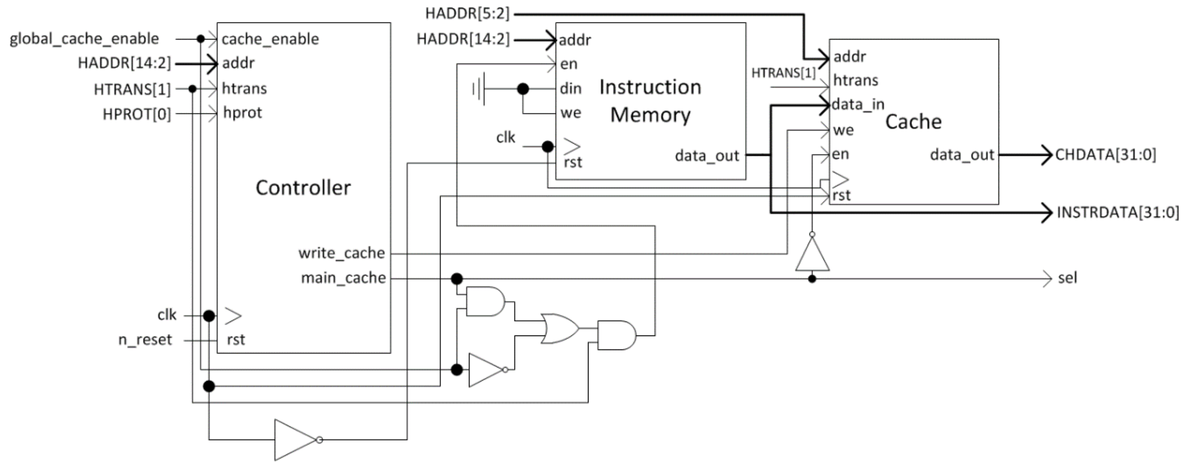


Figure 3.6 – Interface between Controller, Instruction memory and Loop cache

As it can be seen from Figure 3.5, there were three new components added to the system, the Loop Cache Controller, Loop Cache and the multiplexer and the implementation of all of them will be explained in next sections.

3.2.1.2 Loop Cache Controller

The purpose of the controller was to determine when the program execution entered a loop, fill in the cache with the loop instructions and finally from the third iteration of the loop read the instructions from the cache, of course if it was the same loop that was executed all the time. Until now, two approaches of loop detection were discussed: the one from Chapter 2.2, the original loop detection principle whereas the second approach was explained in the software implementation of the system. These two approaches are shown in Figure 3.7 where only differences between them are shown, i.e. input signals.

Figure 3.7 – Two different Loop Cache Controller implementations and their interfaces

The first principle, which is going to be called *the decode* principle, would have to have the instruction code as input and originally, as proposed in the paper, a status flag from the core

which would give the information about the branch status: if the branch was taken or not. Since Cortex-M0 had no branch status signal as it have been the case if Cortex-M3 was used, the second input would be the flags from the core (in the case of Cortex-M0 those are last four bits of the APSR register: negative, zero, cary and overflow flags). The Controller would perform decoding of the instruction: if a potential branch was decoded it would check status of the corresponding flag and determine whether to branch or not. Next step would be to calculate the branching offset and initiate counter register from Figure 2.2 with that value. The rest of the system would behave as described in Chapter 2.2.

The second principle, called *address compare* would use only address of the instruction to be fetched next, make a delayed copy of the address, compare those two and determine whether there was a backward branch or not. This principle was explained in details in Chapter 3.1 although even more information can be obtained from [2].

If Cortex-M3 was used in the project it would have made more sense to use the Decode Controller with the branch status flag as input but in the case of Cortex-M0 there was no real advantage of using this principle because the decoding logic of loop detection would have to be complicated and completely redundant since decoding is already done within the processor itself (but its results are unfortunately unavailable). Since whole software implementation in Python was done using the second principle, it was more convenient to use this approach in hardware as well. The timing differences between two implementations are shown in Figure 3.8 and it can be seen that decode approach would have the advantage of detecting a loop one clock cycle before but with a far more complicated and redundant logic whereas the address compare approach would be one cycle late. This one cycle delay cannot create great damage only if care is taken that the signal *main_cache* which controls where data should be read from was set and reset at particular rising clock edges as shown in Figure 3.8.

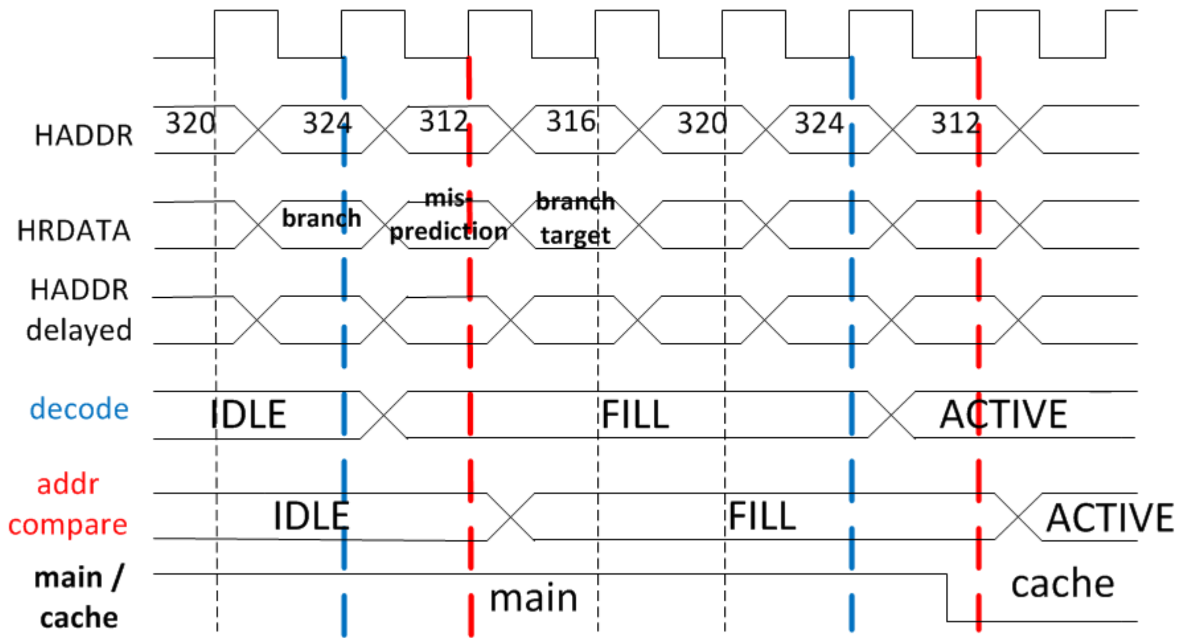


Figure 3.8 – Timing comparison between two different controller implementations

As it can be seen in Figure 3.8, in the case of *address compare* implementation, the Controller changed its states from IDLE to ACTIVE and from FILL to ACTIVE on the next rising edge after a branch target address was sampled (address 312 in Figure 3.8). The way this was really happening when simulating the system itself is shown in Figure 3.9 where it can be seen that the cache was behaving correctly and according to AHB transfer rules with an address and a data phase.

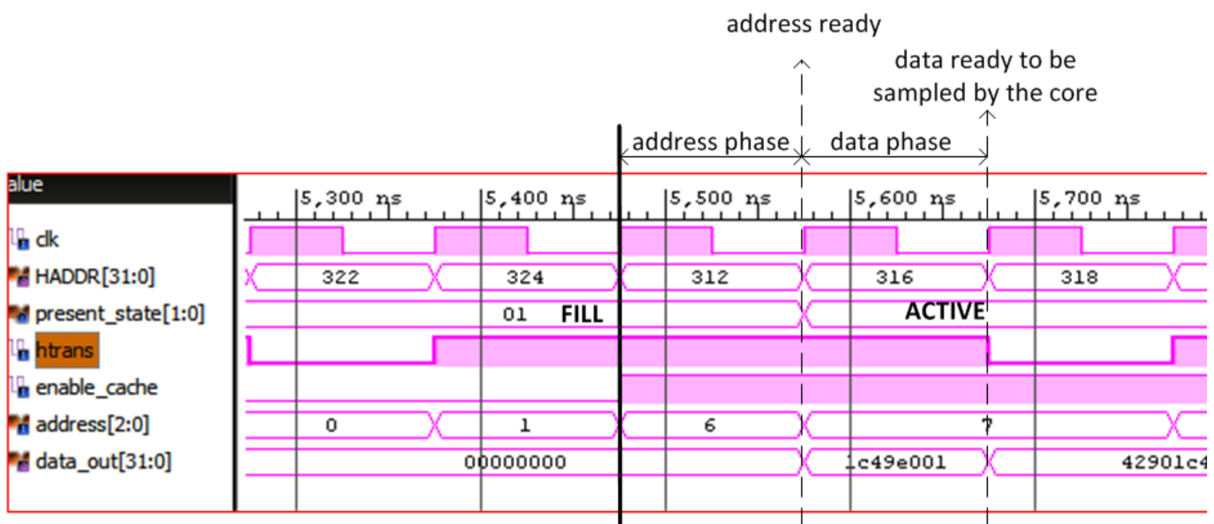


Figure 3.9 – Controller implementation, its state switching and control signal toggling

Apart from address bus as input shown in Figure 3.7, the Controller had HTRANS[1] and HPROT[0] inputs which were indicating that instruction transaction needed to be performed.

Table 3.1 shows conditions that needed to be fulfilled so that the controller switches from one state to another. The first condition column shows the logical conditions presented in Chapter 2.2 whereas the second condition column shows their equivalents and implementation in the real system using the signals that were created in this project.

Cache_size was the size of cache in words, *new_address* was nothing but *HADDR[m+1:2]* where m was the width of the main Instruction Memory address bus and *previous_address* was just a one clock cycle delayed version of the *new_address*. *Branch* was m bite wide signal that remembered the address of the instruction that caused branching in the case of the Controller transiting from IDLE state to FILL whereas *branch_target* was the address of the first instruction inside the loop (also saved when the Controller switched from IDLE to FILL).

Table 3.1 – State switching mechanism with logical switching conditions and their implementations

Present state	*Next state*	*Condition*	*Condition Implementation*
IDLE	*FILL*	*transaction is enabled and it is instruction transaction* *and* *sbb detected and taken* *and* *loop can fit into the cache*	*htrans & (!hprot)=1* *&&* *(previous address – new address > 1(* *&&* *(previous address-new address <= cache size-1)*
	IDLE	*sequential execution* *or* *cof but forwards*	*«else»*

<table>
<tr><td rowspan="3">FILL</td><td>FILL</td><td>instruction transaction
and
no change of flow
and
within limits of the loop</td><td>!hprot
&&
(new address ==
previous address
||
new address ==
previous address+1)
&&
(new address <=
branch)</td></tr>
<tr><td>ACTIVE</td><td>instruction transaction
and
triggering sbb taken again</td><td>!hprot
&&
(new address ==
branch
&&
previous address ==
branch target)</td></tr>
<tr><td>IDLE</td><td>triggering sbb not taken again
or
cof caused by some other triggering sbb</td><td>Others</td></tr>
<tr><td>ACTIVE</td><td>ACTIVE</td><td>instruction transaction
and
within loop size
and
cof caused by triggering sbb</td><td>!hprot
&&
(new address <=
branch)
&&
(new address ==
branch
&&</td></tr>
</table>

	or *no change of flow*	*previous address == branch target)* || *(new address == previous address* || *new address == previous address + 1)*
IDLE	*triggering sbb not taken* *Or* *cof not caused by triggering sbb*	*«else»*

It was very important that the signal *main_cache* toggled before that clock edge (not synchronous to the state change) so that proper memory could be used as the source. This is also illustrated in the Figure 3.10 where critical signal changes are shown in red.

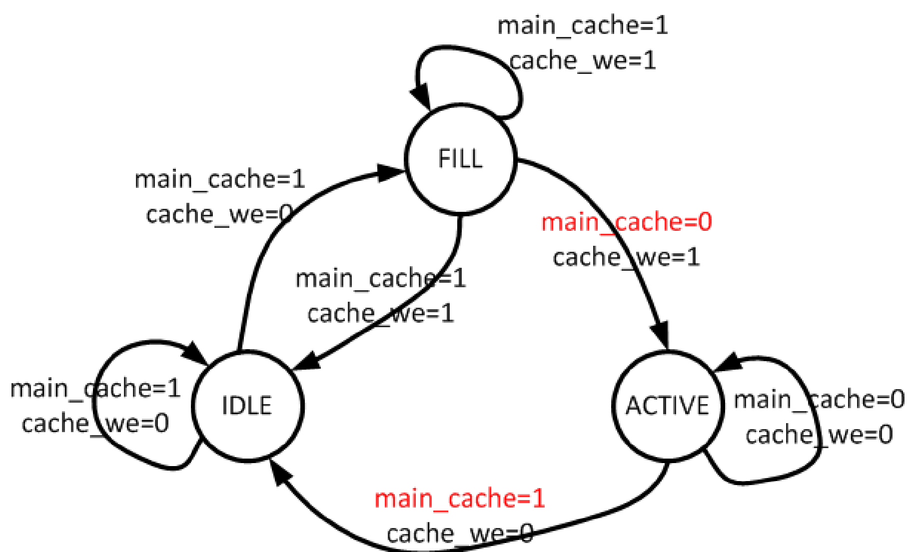


Figure 3.10 – Controller State Machine with output control signals

The signal *main_cache* and its values shown in red indicate that the signal value had to be changed as soon as a certain condition was encountered, it could not wait for the state machine to change its state. As it can be noticed, the final version of Controller had only two output control signals: cache write enable (*cache_we*) and a signal that was enabling output of either main memory or the cache, depending on the state of the controller (*main_cache*).

Signal descriptions and their values depending on the state of the Controller are shown in Table 3.2 (values shown in red are the critical ones, the ones that change before the state changes).

Table 3.2 – Controller output signals

Output signal	*Description*	*IDLE*	*FILL*	*ACTIVE*
main_cache	*Decides whether data from the address now present on the address bus should be read from cache or from instruction memory*	*1*	*1*	*0*
cache_we	*Indicates that data from the address currently on the bus should be written into the cache (on the next clock rising edge)*	*0*	*1*	*0*

The Loop Cache System interface and detailed communication between different parts of the system are shown in Figure 3.6.

Table 3.3 shows how input enable signal of the instruction memory was depended on the *global_cache_enable* signal controlled by the user and *main_cache* output signal from the Controller. Input enabling signal of the cache memory was created simply by inverting this signal.

Table 3.3 – Instruction Memory Enable signal generation

global_cache_enable *(A)*	main_cache *(from Controller)* *(B)*	instruction_memory_enable *(C)*
0	*0*	*1*
0	*1*	*1*

1	*0*	*0*
1	*1*	*1*

$$C = (\bar{A} + AB)$$

Definitive enable signal was created of course by multiplying this signal by HTRANS[1] signal of the AHB bus which was used to initiate a transaction.

3.2.1.3 Loop Cache

As it can be seen from the Figure 3.6, HTRANS[1] was used as input into the cache although it was already used to create enable signal which could be disputed as redundant. This signal was one cycle delayed within the cache itself (it is called *htrans_a* inside cache) in order to perform correct write since the address the cache was writing data in was also one cycle delayed (*address_a*). The writing process is shown in Figure 3.11. In the case simulated, address bus of the loop cache was 3b wide (8 locations each containing 4B) and those were HADDR[4:2] bits. Two least significant bits from the address bus were completely neglected in the whole system since both memories were word addressable (this is the reason why all the conditions for the address comparison in the Table compared if they differed by 1 and not by 4 what would have been the case if least significant bits of the HADDR were used as well).

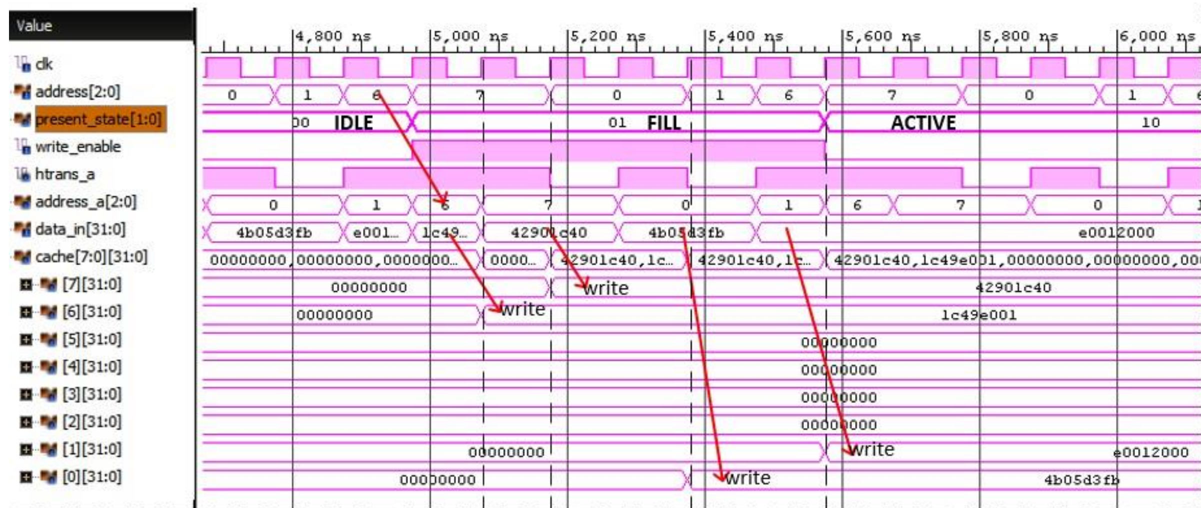


Figure 3.11 – Filling of the Loop Cache

Figure 3.11 shows how cache writing process was performed. It clearly illustrates one of the benefits of this implementation because instructions did not need to be aligned to any

starting address (first location that was written into was 6, the second one was 7, third was 0 and the last one was 1).

Address_a was only a one cycle delayed version of address signal and at first it was created within the cache module itself but later on it was noticed that there was a signal inside loop cache controller module that was already containing the previous address. If a part of that signal (lower n bits if n is the width of the address bus of the cache) was taken from there, some hardware savings could be gained.

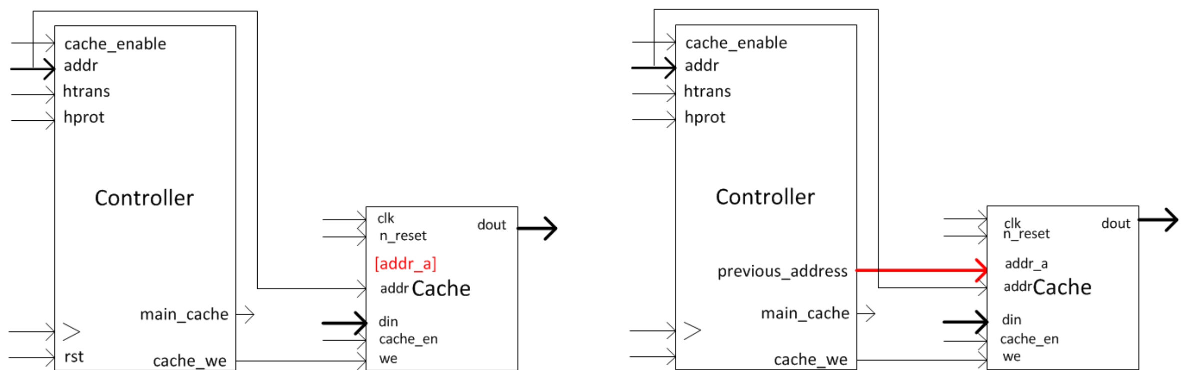


Figure 3.12 – Two different Controller-Cache implementations and interfaces

During the system development there were three different versions of the Controller that were implemented, each next version was an improved version of the previous one in a sense of resources utilization which later led to smaller power consumptions.

First version of the Controller had 2 outputs more than the version showed in Figure 3.6 - these outputs and their description are shown in Table 3.4.

Table 3.4 – Controller output signals (first version)

Output signal	*Description*	*IDLE*	*FILL*	*ACTIVE*
main_enable	*enables that the address present on the HADDR bus at that moment is read from the main instruction memory*	*1*	*1*	*0*

cache_enable	*enables that the address present on the HADDR bus at that moment is read from the cache*	*0*	*0*	*1*
cache_we	*indicates that the data from the address currently on the bus should be written into the cache (on the next rising clock edge)*	*0*	*1*	*0*
mux_sel	*decides whether processor should read data from RAM or cache*	*1*	*1*	*0*

It is quite easy to see that there was redundancy in this implementation: signals *mux_sel* and *main_enable* had the same values in each state of the Controller whereas signal *cache_enable* had their inverted values. The next logical step was to remove two redundant signals and this was how the number of outputs was cut to only two as shown and explained in previous sections. The multiplexer was controlled by the same signal that was enabling the main instruction memory whereas input enable signal of the cache was the inverted version of this signal.

Second implementation of the Controller had only two controller outputs as explained. Moreover, all the conditions to enable state transitions were implemented the way shown in Table 3.1. The system was behaving correctly but then it was noticed that there was also redundancy present in condition implementation.

Table 3.5 – Utility reports of three different implementations of the Controller

Description	*LUT*	*Slice Reg*	*Slice*	*LUT as log*	*LUT ff*	*Bonded IOB*	*BUF CTRL*
Redundancy in condition implementation and output signals	*5*	*41*	*33*	*85*	*100*	*22*	*1*
Redundancy in condition implementation	*3*	*41*	*30*	*83*	*90*	*20*	*1*

No redundancy in both number of output control signals and condition implementation	*9*	*41*	*26*	*59*	*72*	*20*	*1*

Instead of repeating same conditions couple of times, new variables (cond1, cond2, cond3…) were created to save logical information if this particular condition was fulfilled or not and later on, these signals which had already been synthesized elsewhere were reused. This lead to the third Controller implementation which brought more utilization savings as shown in Table 3.5.

Final Verilog designs of both Controller and Loop Cache are listed in Appendix B.

4 Testing and Measurements

4.1 Software Implementation

4.1.1 Methodology

Simulator input data were the same benchmarks used in [2]: *coremark, dijkstra, emlcd hex, preamp, primes, sha* and *touch.* As the first version from [2], the new simulator counted:

- number of reads from the main instruction memory (*flash_reads*),
- number of writes into the cache (*sram_writes*) and
- number of reads from the cache (*sram_reads*)

but now using a little bit more complex state machine. Based on these recorded values, simulator calculated hit rate, read/write ratio, power consumption in case cache was not used and power consumption in case it was used. The way these evaluation parameters were calculated was explained in [2]. Each benchmark was evaluated using 5 different sizes of cache: 16B, 32B, 64B, 128B and 256B.

4.1.2 Results

Table 4.1 shows power saving percentage, hit rate and read/write ratio in the case of cache size 64B. Each evaluation parameter has the value obtained in [2] using a state machine with only 4 states (called *basic* here) and next to them, on the right, the results from the new simulator (*improved*) which uses 4 additional pseudo states.

Table 4.1 – Simulation results for cache size 64B

Benchmark	*Power savings [%]*		*Hit rate [%]*		*R/W ratio [%]*	
	basic	*improved*	*Basic*	*improved*	*basic*	*improved*
Coremark	*19*	*36*	*29*	*41*	*1.02*	*2.09*
Dijkstra	*2*	*16*	*14*	*21*	*0.68*	*0.85*
Emlcd	*29*	*38*	*40*	*43*	*1.67*	*2.97*
Hex	*50*	*61*	*62*	*68*	*2.98*	*3.98*
Preamp	*42*	*62*	*54*	*68*	*2.89*	*8.07*
Primes	*83*	*79*	*97*	*85*	*18.562*	*24.19*
Sha	*73*	*81*	*86*	*89*	*8.86*	*16.46*
touch	*-1*	*43*	*8*	*48*	*0.58*	*3.48*

It can be seen that by ignoring data transaction activities on the bus (not allowing them to break cache fill or read), the cache hit rate, R/W ratio and therefore the overall power savings get to grow significantly. Recording the same system properties for different cache sizes showed that power savings get to grow as the cache size increases but only up to a certain point (with a cache of around 32B). Increasing cache size over 32B did not show to bring any advantages into the system performance mainly because the loop sizes do not get to grow larger than 16 instructions at least in the case of the benchmarks used here.

4.2 Hardware Implementation

After the system was built, it was necessary to simulate its behaviour and prove it was working correctly which was done using VIVADO simulator. The design had to work properly both after implementation and after place and route of course, which was proven by behavioural, post synthesis and post implementation simulations.

Next step was to program the board itself and verify the design was working properly in hardware as well. There was no easy and direct way of doing this on the board as it was done in simulation by checking values of processor registers, other variables and flags. Instead, the blinking diode was used as a stable proof the design was working well, as it was used in [18] as well. As in [18], a slowly blinking diode was used to verify the design was working properly, because a fast change could not be noticed by human eye at all. On the other hand, performing a VIVADO simulation of a slowly blinking diode was extremely time consuming and therefore was only used as a verification that the system was working correctly. All the power measurements and therefore all the power results shown in this report, refer to the case of a fast blinking diode because in this way the results can be compared and discussed without the need of time consuming simulations.

Next two sections show different power results for different design settings: the first shows VIVADO power reports whereas the second shows measurements performed on the board in real time.

4.2.1 VIVADO measurements

This chapter presents the results gained form VIVADO DS simulation of the system: first the Methodology used to evaluate the system is explained and later on the results are listed and commented.

4.2.1.1 Methodology

4.2.1.1.1 Low and high confidence of power reports

As it was mentioned earlier in 2.4.5 , it is possible to measure power both after synthesis and after implementation, without toggling information, with toggling information for some signals or with complete toggling information for all nodes. Table 4.3 shows power reports for the case of Instruction Memory of size 32KB and the cache size of 64B. First column shows total power (dynamic and static) consumed by the system with a default toggle rate for all nodes (12.5 %). the second column had a specified toggling rate for the *global_enable* signal (toggle rate of 0% and static probability of 1 or 0, depending on the situation examined), whereas the third column shows post synthesis power report in the case of using a .saif (switching activity interchange format) file that was generated in a post-synthesis simulation which was using a testbench with the same clock frequency that was going to be used to perform the power report. Last three columns show those results but after implementation (place and route). Next to each power number there is a level of confidence stated: low for a report with default toggle rate and high when using a switching activity file obtained from corresponding simulation.

In order to be able to get the feeling about how much power would be consumed in case of building the memory system not from FPGA fibre but with the technologies available to Silicon Labs, number of cache writes and reads as well as number of reads from main instruction memory were recorded in the simulation as well. All the numbers were recorded in the same simulation time slot of 125 us. The results are shown in Table 4.6.

4.2.1.1.2 Test programs

Testing setups were using Instruction Memory of size 32KB as in the case of Zero Gecko whereas different cache sizes (16B, 32B and 64B) and programs with different loop sizes (8 instructions, 16 instructions, 24 instructions, 32 instructions and 40 instructions) were used. This makes up total number of 15 different configurations to be synthesized and later on measured and recorded. Simulating different programs executing on the core meant that programs with different loop sizes should be loaded into the Instruction Memory. The easiest way this could be done was by modifying the program used in [18], which had 8 instructions in the loop, to obtain a greater loop size. Since the original C code had only one line in for loop part (*counter ++*), Figure 2.30, which corresponded to one assembly instruction (*inc r1*), the easiest way to enlarge the loop was to add more of these *counter++* lines in C code (as many as the difference from the desired loop size and the original loop size was). These source codes

were used to create .coe files that were later on loaded into the Instruction Memory. Table 4.2 shows how different program loop sizes were achieved. Same binaries (.coe files) were used in both VIVADO and real time measurements.

Table 4.2- Source code modification impact on program loop size

Loop size [instructions]	*Number of "counter++" lines in source file*
8	*1*
16	*9*
24	*17*
32	*25*
40	*33*

4.2.1.2 Results

Table 4.3 shows that, as expected, when having no information about the toggle rate and assuming default toggle rate for each signal, the power results get worse than in the case of knowing the exact toggle rates.

Table 4.3 – Dynamic power reports after synthesis and after implementation using different toggling information (32KB Instruction Memory, 64B cache, loop size 8)

Instruction Memory (32KB)		*1*	*2*	*3*	*4*	*5*	*6*
global_enable = *0* *(no cache)*	*dynamic power [mW]*	*104*	*103*	*100*	*117*	*115*	*114*
	report confidence	*low*	*low*	*high*	*low*	*low*	*high*
global_enable = *1* *(with 64B cache)*	*dynamic power [mW]*	*108*	*107*	*106*	*116*	*114*	*113*
	report confidence	*low*	*low*	*high*	*low*	*low*	*high*

Table 4.4 – Power report generation details for different measurement configurations

Config	*Power report generation details*

1	*post synthesis power report with default toggle rate for all nodes*
2	*post synthesis power report with exact toggle rate for* global_enable *signal*
3	*post synthesis power report with proper .saif file as input*
4	*post implementation power report with default toggle rate for all nodes*
5	*post implementation power report with exact toggle rate for* global_enable *signal*
6	*post implementation power report with proper .saif file as input*

The results also show that exact power consumption can be known only after place and route is performed and that a lot of dynamic power consumption (in this case around 12% and 7%, depending on global enable signal value) gets consumed by the clock tree and wiring itself. Complete power report (with both static and dynamic power numbers), Figure 4.1, shows that most of the power consumption (around 69%) belongs to static power consumption which is reasonable considering that Zynq-7000 AP SoCs use 28nm High-K Metal Gate (HKMG) technology. It is well known that by lowering process node technology, leakage power becomes a dominant contributor to the overall power consumption. Therefore it becomes reasonable why a non-conventional process had to be used at these gate sizes. High-K Metal Gate (HKMG) process is a process where the capacitance of the gate oxide gets increased by using a dielectric with a higher κ than the one of a SiO_2 that is normally used as a gate oxide.

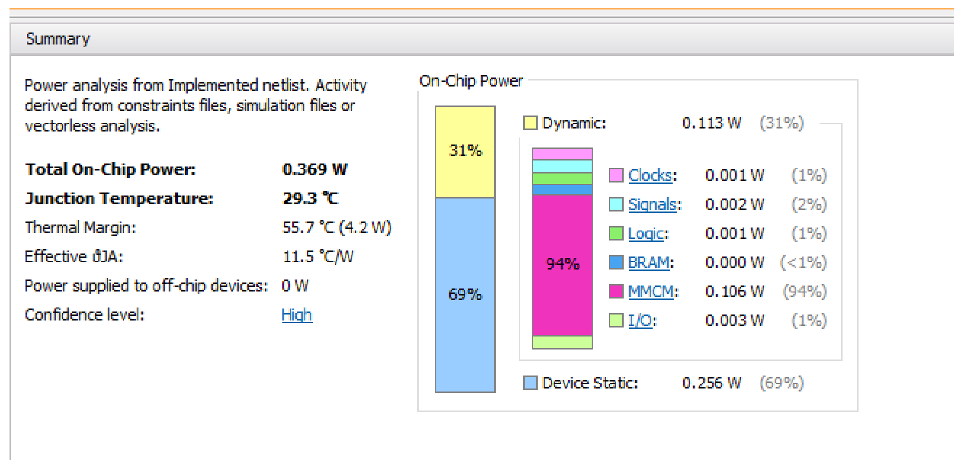


Figure 4.1- Complete power report, VIVADO layout
(32KB Instruction Memory, 64B cache, loop size 8, cache enabled)

But even with a sophisticated process like this, static power still dominates the overall power consumption and not much can be done to reduce it. That is why all the results in the following results will refer to dynamic power consumption only since the static one was fixed: 256mW.

All the power numbers reported from now on were obtained from post implementation power reports with high confidence (with usage of corresponding switching activity files).

Having a look at the results from Table 4.3, it can be seen that in the case of enabling usage of loop cache there was a dynamic power saving of 1mW compared to the case when no cache was added into the system. Considering the overall power consumption of 114mW this saving of less than 1% was not something to be too much proud of. On the other hand, these results show the consumption of the entire system (with the core itself of course) so it was necessary to separate the consumption of the core form the consumption of the memory system alone.

This was performed by synthesizing the core alone (with no Instruction Memory, no cache memory but with the rest of the system). The post implementation power report is shown in Figure 4.2.

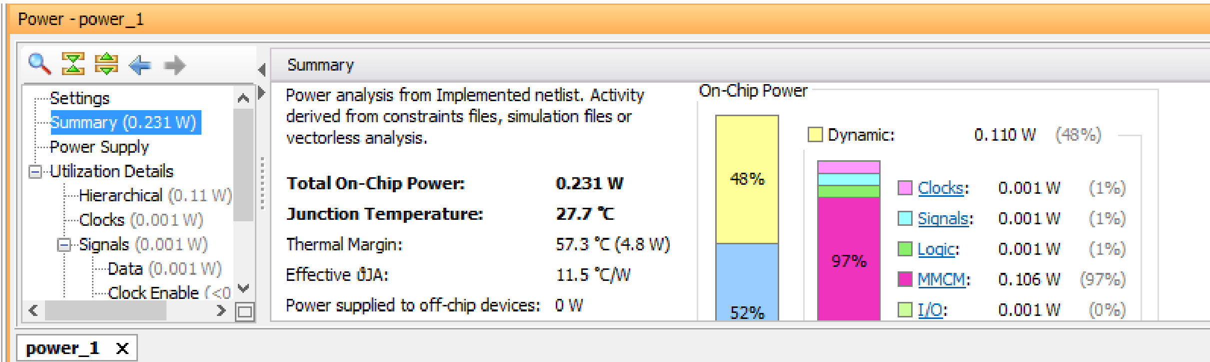


Figure 4.2- Power report of the system with no memory hierarchy

Comparing this result of dynamic power consumption of 110mW with no memory hierarchy with 114mW using only Instruction Memory and 113mW when enabling use of cache, it is easy to conclude that memory system itself consumed either 3mW or 4mW depending if the cache was enabled or not. Saving of 1mW when enabling usage of cache now becomes 25% which is a result that cannot be neglected.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	29.293 ns	Worst Hold Slack (WHS):	0.120 ns	Worst Pulse Width Slack (WPWS):	3.751 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4184	Total Number of Endpoints:	4184	Total Number of Endpoints:	1449

All user specified timing constraints are met.

Figure 4.3 – Design timing repor

Another thing worth noting at this point is the design post implementation timing report which was checked every time to make sure there were no timing violations (positive TNS and WNS) and that all constraints were met. This is clearly shown in Figure 4.3.

In order to show how different cache sizes and different loop sizes influence power reports, 15 different configurations were synthesized and simulated. Configuration settings are shown in Table 4.5 whereas the power reports are shown in Table 4.6.

Table 4.5 – Different testing configurations

Configuration	*Loop size [instructions]*	*Cache size [B]*
1	*8*	*16*
2	*8*	*32*
3	*8*	*64*
4	*16*	*16*
5	*16*	*32*
6	*16*	*64*
7	*24*	*16*
8	*24*	*32*
9	*24*	*64*
10	*32*	*16*
11	*32*	*32*
12	*32*	*64*
13	*40*	*16*
14	*40*	*32*
15	*40*	*64*

Table 4.6 – Power and statistic reports for different system configurations

	Dynamic power [mW]	*cache reads*	*cache writes*	*main reads*	*R/W ratio*	*Hit rate*
1	*113*	*790*	*8*	*20*	*98.75*	*97.53*
2	*113*	*790*	*8*	*20*	*98.75*	*97.53*
3	*113*	*790*	*8*	*20*	*98.75*	*97.53*
4	*114*	*0*	*0*	*810*	*-*	*0*
5	*113*	*782*	*16*	*28*	*48.875*	*96.54*

6	*113*	*782*	*16*	*28*	*48.875*	*96.54*
7	*114*	*0*	*0*	*810*	*-*	*0*
8	*114*	*0*	*0*	*810*	*-*	*0*
9	*113*	*774*	*24*	*36*	*32.25*	*95.55*
10	*114*	*0*	*0*	*810*	*-*	*0*
11	*114*	*0*	*0*	*810*	*-*	*0*
12	*113*	*766*	*32*	*44*	*23.9375*	*94.56*
13	*114*	*0*	*0*	*810*	*-*	*0*
14	*114*	*0*	*0*	*810*	*-*	*0*
15	*114*	*0*	*0*	*810*	*-*	*0*

As it can be seen from Table 4.6, there was not too much difference in power numbers in case of using different configurations. This can be explained by the fact that the VIVADO resolution goes as far as 1mW and the configuration changes are sometimes in order of couple of bytes which brings power changes that are more subtle than 1mW and cannot be recorded by the simulator. Furthermore, although utility reports showed that Instruction Memory was built from BRAM blocks and cache memory merely from flip flops (LUTs), there was no notion of the power ratio between a read from a BRAM and a read from a flip flop. This is why writes and reads numbers can help getting the feeling about real power consumption if the hardware was synthesized as an ASIC and not from FPGA fibre. What can be seen from power reports in Table 4.6 is that when a loop was larger than the cache size no cache writes and therefore no cache reads were performed at all and the power consumption of the system in that case was the same as in the case when cache was globally disabled, i.e. not existed at all.

Cache reads and writes, as well as hit rates and R/W ratios show very good use of the TLC technique (around 98% of power savings) but it should not be forgotten that the test programs used here were completely synthetically created and that the reads and writes were recorded for the period when only one loop was executed over and over again. Future work could include the use of test programs that would reflect more truly real embedded program execution environment.

4.2.2 Real time measurements

Apart from performing power estimations using VIVADO simulator, system was evaluated in real time as well. For those purposes the system was synthesized on a Xilinx ZC702 evaluation kit which will be briefly introduced in one of the next sections.

4.2.2.1 Methodology

This section explains which hardware the system was synthesized on, what kind of software support was needed to perform the power measurements and the rest of the evaluation system setup. Following [18] it was clear that, apart from the zc702, it was necessary to order the Texas Instruments USB-TO-GPIO Interface, download and install the TI Fusion Digital Power Designer GUI. Evaluation kit and the GUI will be briefly introduced in the next sections.

4.2.2.1.1 Evaluation Kit: zc702

For the purposes of system verification and testing Xilinx ZC702 evaluation board for XC7Z020 AP SoC was used. It provides features that can commonly be found in most embedded systems, such as DDR3 Component Memory, HDMI Codec, I2C bus, USB to UART interface, tri-mode Ethernet PHY… Before using the board it was necessary to verify its proper functioning by performing the Built In Self Test as described in [24] that came loaded into the Quad SPI Flash memory on the board (tests UART, I2C, Timer, DDR3 Memory, LEDs, Watchdog Timer, SWITHES etc.) The board layout is shown in Figure 4.4 where the elements that were used in this project are marked with blue boxes: DIP switches, tasters, LEDs, the power management system and the Zynq XC7Z020 AP SoC of course. One of the reasons why this board was chosen to be used are "the power regulators and a PMBus compliant system controller from Texas Instruments it uses to supply core and auxiliary voltages." [20] and the

possibility to easily monitor and measure those voltages through a GUI from TI called Fusion Digital Power Designer.

The whole ZC702 hardware system is built around Zynq-7000 XC7Z020-1CLG484C AP SoC which consists of an SoC integrated processing system (PS) and programmable logic (PL) on a single die. "The PS integrates two ARM Cortex-A9 MP Core processors, AMBA interconnect, internal memories, external memory interfaces and peripherals including USB,

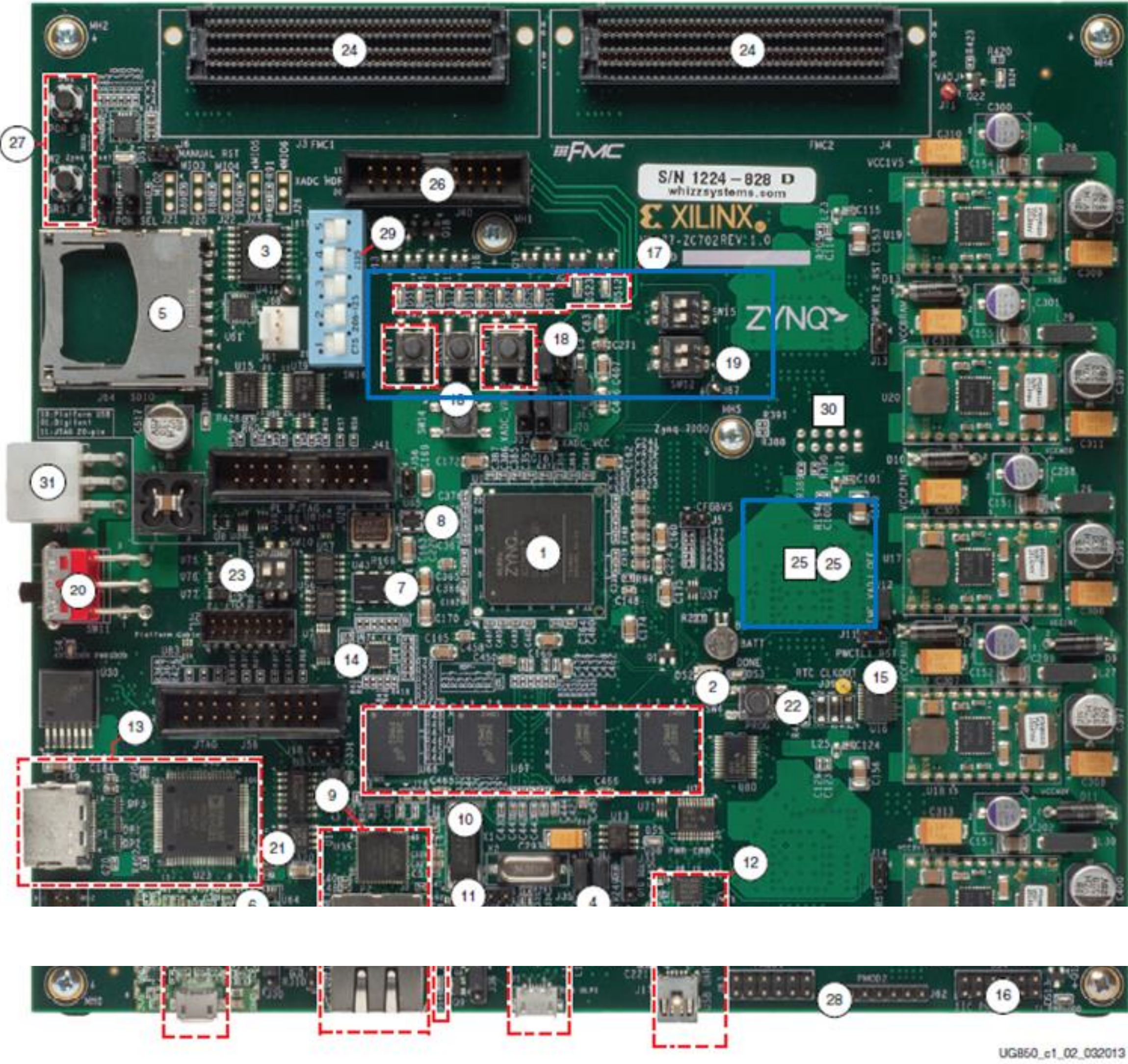


Figure 4.4 - zc702 Evaluation Kit board layout [20]

Ethernet, SPI, SD/SDIO, I2C, CAN, UART and GPIO."[20] The main advantage of Zynq-7000 family is the flexibility and scalability of an FPGA combined with performance, power and ease of use associated with ASIC and ASSP. "The integration of the PS with the PL provides better performance than the case of two chips used due to their limited I/O bandwidth, loose-coupling and power budgets."[25] All the devices from Zynq-

7000 family include the same Processing System although the Programmable Logic and I/O Resources differ from device to device. Since in this project, the PS was not used at all: all the system (including the core) was written in HDL and synthesized using only PL features of the device, such as Block RAMs for synthesizing Instruction memory, Clock Management to adapt 200MHz oscillator on the board to the system frequency of 10 MHz and of course CLBs with LUTs to synthesize the rest of the system logic. The system input interface as well as status signals that enabled communication with the user are shown in Figure 4.5 (this is the part of the system marked with a blue frame in Figure 4.4)

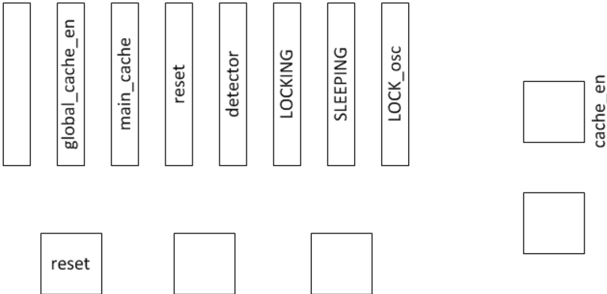


Figure 4.5 - System user interface and output status (LEDs, switch and push buttons)

Evaluation kit has several power domains as shown in Figure 4.6. The board uses power regulators and a PMBus (Power Management Bus) compliant system controller from Texas Instruments to supply core and auxiliary voltages. Apart from 12 V input supply that powers the board, there are 5 switching regulators and 1 linear regulator which generate different voltages required for different power domains. Voltage outputs of these regulators are controlled by three TI power controllers and it is possible to monitor them via a GUI called Fusion Digital Power Designer.

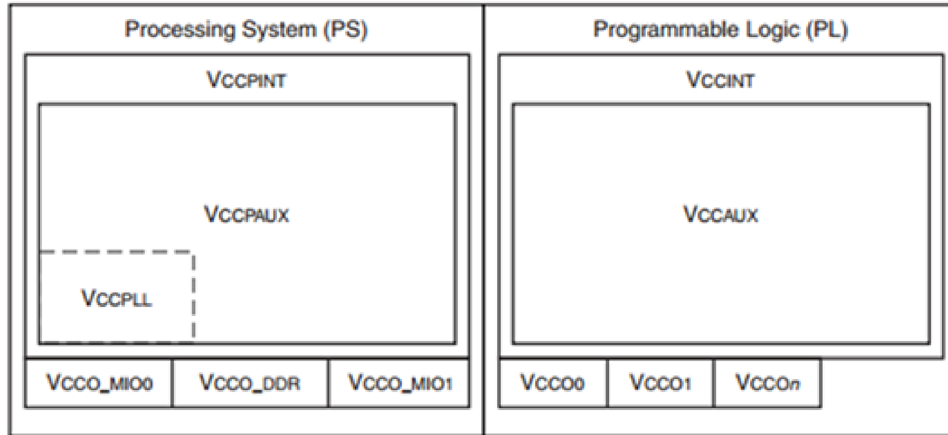


Figure 4.6 - Power domains on the ZC702[26]

4.2.2.1.2 TI Fusion Digital Power Designer (FDPD)

FDPD is a GUI specially designed to monitor 10 different rails on the board. Each rail supplies particular subset of components in a corresponding power domain. List of components each power rail supplies can be found in [26]. It is possible to monitor voltage, current, power and temperature of each node in real time: see waveforms in separate windows as well as save the measurements in a .csv file.

Saving measurements in .csv file not only does give possibility to process this data later in a desired way but it also gives opportunity to save data in a higher resolution than it can be shown in the figures drawn by the GUI itself (voltage resolution of a mV, current resolution of 100uA, power resolution of 100nW). An example of saved measurements is shown in Figure 4.7 where four different rails (their voltages, currents and power) are monitored by the same device (TI UCD9248). There are two more files with readings of the other two power management devices that monitor other 6 rails that are of importance for the supply system of this evaluation board.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Timestam	Adapter	Part_ID	Address	READ_VO	READ_VO	READ_VO	READ_VO	READ_IOU	READ_IOU	READ_IOU	READ_IOU	READ_PO	READ_PO	READ_PO	READ_POUT_4	
2	55:21.0	1	UCD9248	52	1.005	1.002	1.805	1.801	0.0313	0.1094	0.0156	0.0313	0.0471	0.0471	0.0471	0.0471	
3	55:21.4	1	UCD9248	52	1.006	1.002	1.8	1.807	0.0156	0.1406	0.0313	0	0.0314	0.0314	0.0314	0.0314	
4	55:21.9	1	UCD9248	52	1.007	1.003	1.802	1.802	0.0938	0.1094	0.0313	0.0313	0.0314	0.0314	0.0314	0.0314	
5	55:22.4	1	UCD9248	52	1.007	1.004	1.804	1.804	0.0313	0.0625	0.0313	0.0625	0.0314	0.0314	0.0314	0.0314	
6	55:22.9	1	UCD9248	52	1.004	1.002	1.798	1.805	0.0625	0.1406	0.0313	0.0938	0.0471	0.0471	0.0471	0.0471	
7	55:23.4	1	UCD9248	52	1.008	1.003	1.797	1.805	0.0313	0.1406	0.0469	0.0625	0.0471	0.0471	0.0471	0.0471	
8	55:23.9	1	UCD9248	52	1.008	1.004	1.804	1.809	0	0.125	0.0781	0.0938	0.0626	0.0626	0.0626	0.0626	
9	55:24.4	1	UCD9248	52	1.006	1.003	1.801	1.801	0.0625	0.1094	0.0938	0.0313	0.0472	0.0472	0.0472	0.0472	

Figure 4.7 – Format of a .csv file with real time measurements saved by TI FDPD

Each measurement is recorded at a particular time stamp contained in the first column. It can be noticed that a new reading is available each 500ms which is the polling interval of the GUI

itself. Voltage and current measurements show the current values at a particular time stamp whereas power values show RMS value of the power in time interval between this and the previous reading. This means that total power is calculated by summing up these particular values.

4.2.2.2 Results

As it was mentioned, the way to verify the core was running the program code on the hardware, was to observe the behaviour of the LED called detector in . In order for human eye to detect the change, the blinking period had to be long (around 10s). Similar principle was used to check whether the measurements on the board made sense: board was programmed to blink diode very slowly and measurements were recorded separately for the period of diode being on and diode being off. In case of a diode being ON, measurements should show higher power consumption since LED power consumption is quite significant: around 90mW according to [27]. To make this change even more visible the author programmed 4 LEDs to be tuned on and off. Total power readings (sum of power consumed by all the rails) was shown to be higher in the case when LEDs were off than when they were on which made no sense at all. However, the power measurements for a particular rail, VCC3V3, that is used to power up the LEDs, were smaller in case of LEDS off: 22.4436mW than in case of LEDs being on: 22.8555mW. The difference is around 400mW which is quite close to what four diodes should consume (around 360mW) but it still stays unclear why total power consumption is smaller in case of LEDs being on. These results are summed up in Table 4.7.

Table 4.7 – Total and Vcc3V3 power consumption for 4 LEDs ON and OFF

	Total power [W]	*VCC3V3 power[mW]*
4 LEDs on	*1.5584*	*680.8555*
4 LEDs off	*1.5612*	*282.4436*

Another type of measurement was conducted to try to get more information about the system: voltages and currents of the system were monitored and recorded while LEDs were first on for 10s then off for 10s and repeated that way for 4 times (7 transitions from on to off and off to on). Voltages, power and currents were then plotted in MATLAB and the results are shown in Figure 4.8 - 4.10.

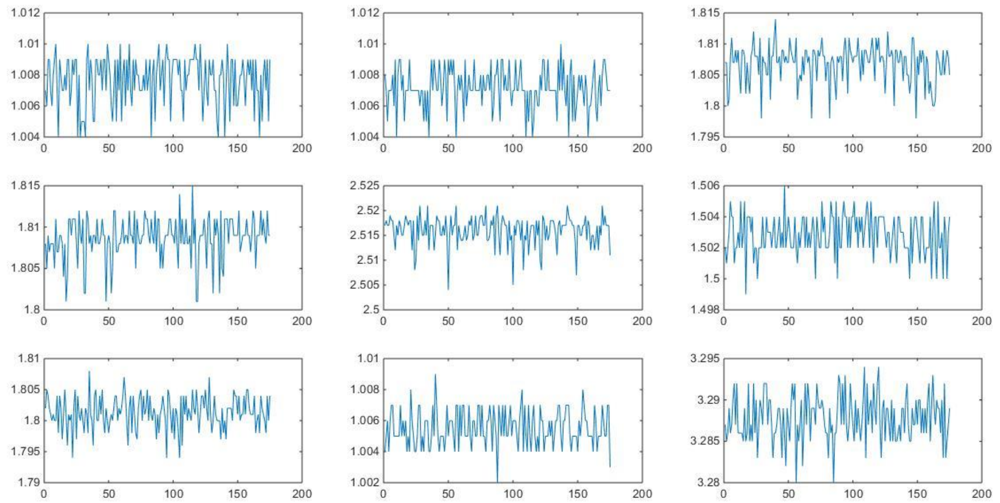


Figure 4.8 – Real time voltage measurements for LEDs periodically ON and OFF (Matlab)

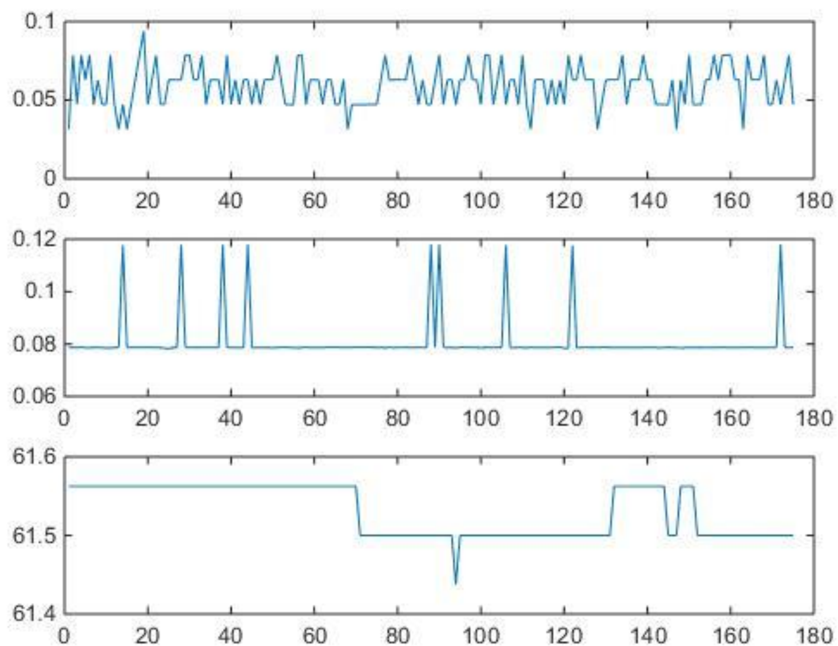


Figure 4.9 – Real time power consumption for LEDs periodically ON and OFF

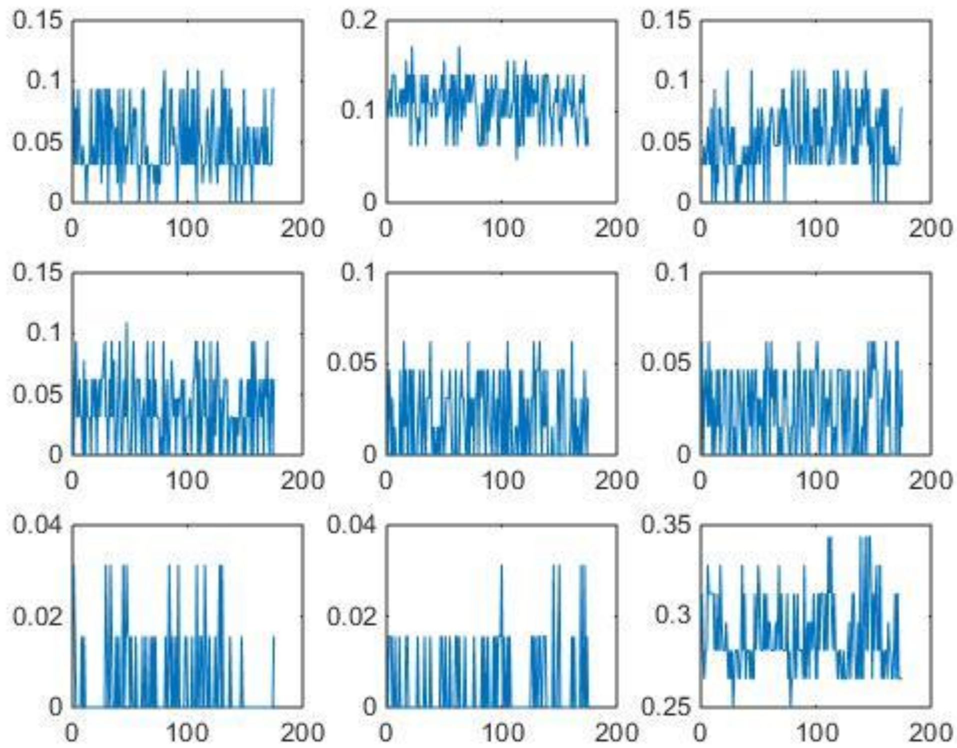


Figure 4.10 – Real time current measurements for LEDs periodicallt ON and OFF

Total number of samples in time was 160 (each state of diodes recorded in 20 samples). What was expected to be seen in the figures (only 9 rails shown for the sake of simplicity, the last one shows no more information) was that there was a current/voltage drop/rise for different states of the system since according to [27], one LED drew 30mA. As it can be seen from Figure 4.10 there were no falls/rises in values that would have helped drawing conclusions.

Nevertheless, systems with different cache sizes were configured and tested while running the original code with the loop of size 8 instructions, with enabled use of cache and disabled use of cache (controlled by the switch on the board) and the energy consumption in a particular time slot is shown in Table 4.8.

Table 4.8 – Total power consumption (loop size: 8 instructions)

	16B	*32B*	*64B*
No cache	1.3642	1.3722	1.3598
With cache	1.4037	1.4090	1.5297

These results show that when using bigger cache, energy consumption gets slightly higher in the case of running the same code which is quite clear. Power numbers are now I order of 1.5 W which is quite different than the case of VIVADO power reports (around 300mW) which is reasonable taking into account that VIVADO does not take into account the board system, but only the Zynq device consumption.

Unexpectedly, the results show that power consumption is higher in the case of including cache into memory hierarchy than not having it as part of the system at all. Although this is not very plausible taking into account the idea and goal of reducing power consumption by adding cache into the system, this result could be explained if the ratio between RAM and register read on this particular FPGA fibre was measured and maybe concluded that it was around 1:1 which would not be the case if the system was synthesized as an ASIC.
Author tried to perform these measurements in the explained experimental setup but since the GUI polling (measurement) interval was 500ms and the core was running on 10 MHz (100ns), there was no way of knowing power consumption of separate executed instructions and therefore no way of measuring RAM and register access power. Another important thing that could be checked is whether the RAM generated with IP generator and having an enable signal is actually switching its interior logic (mostly consisting of big comparators) and signals although not changing its output while disabled or disabling it means really turning of all the logic as well. Most probably latter is true but it would not be a bad idea to check this.

5 Conclusion and Future Work

5.1 Conclusion

There are many different caching techniques that are believed to improve instruction fetch energy of microcontrollers, some of them were explored and explained in [2]. One of the techniques, called Tight Loop Cache, was chosen to be evaluated as the most promising and easy to incorporate into a working system. The technique was implemented both in software (Python) and hardware (Verilog), evaluated by counting parameters close to meanings of cache hits and misses and by calculating, estimating and measuring energy consumption.

Software simulations showed that power savings with the use of loop cache can go as far as 80% (taking into account only the memory system). These measurements were performed by counting reads and writes into the cache and main instruction memory neglecting power consumption of the control logic behind the system. Such good potential savings served as initial encouragement for the system to be implemented in hardware so that the results can be checked and verified.

It was proven that the principle can be integrated within a system that uses ARM Cortex-M0 which does not offer any advanced information, such as branch status of the instructions in different pipeline stages. This leads to a conclusion that the technique could be easily integrated into any modern system. Technique was first built in software, taking into account only memory access powers (neglected the control power consumption), where simulations were performed in order to prove the feasibility of the system. Excessive simulation of the hardware implementation showed the principle can be successfully applied to any modern MCU system. Power optimization techniques of VIVADO synthesis and place and route tools were exploited to their maximum and showed that the use of the technique could bring up to 25% energy savings.

It was shown that even a small modification of RTL design of a module (Controller in this case) can lead to a completely different implementation: different utility reports and therefore different power consumptions. Smart changes in the design at this level can bring much more savings than changes in synthesis and implementation setup. Different synthesis and implementation setup options offered by VIVADO DS and guided by power optimization principles were explored deeply and their influence on final power reports was shown.

Real time measurements brought a lot of confusion and doubts that are tightly related to the initial system setup. Some of these issues were mentioned at the end of last Chapter, such as ratio of power of memory access to a bit RAM and a register bit. In the technology available at Silicon Labs, this ratio goes as far as 1:10 which would definitely bring more savings since this ratio in the FPGA fibre is believed to go as close to 1:1.

Main conclusion that can be made from all the results discussed is that the initial system hardware implementation was not set in a best possible way to achieve correct power saving numbers: FPGAs are usually used only to build prototypes and prove principles of operation. This was successfully performed: a working design that is using a small cache to store instructions from small loops was built and even brought around 25% power savings into the memory hierarchy system.

5.2 Future Work

As it was discussed in the previous section, the initial resources that were available for the system to be implemented on (FPGA fibre is not as much power optimized as possible) were not optimally chosen. It is believed that if the design gets implemented as an ASIC with completely configurable and controllable synthesis and place and route tools, it would bring more power savings than in the case of using FPGA.

Another important thing noticed when analysing benchmarks used as inputs into the software simulator was that it happened quite often that a loop was executed many times, which was followed by a quick sequential execution and then the return to the same loop execution. In the case of the TLC implemented in this project, there was no information about which loop the cache was filled with so it happened quite often that new writes were performed into the loop cache although it was already filled with the correct data. A possible modification to the Controller design would be to allow it directly enter ACTIVE state from IDLE if it was concluded that the cache was already filled with the right loop instructions. This way, unnecessary cache writes (costly FILL state) would be avoided.

Chapter 3.2.1.2 mentioned two possible implementations of the Controller but this project involved design of only one of the principles. It would be interesting to implement the second approach as well and compare the results.

Another idea that was analysed roughly but could be considered as future work was to deal with conditional branches inside the loop as well and to store both situations: branch taken

and not taken inside the cache in different places and base on the situation chose to read from different parts of the cache. This idea still sounds too costly regarding control hardware implementation but it is worth exploring as well.

Real time measurement potentials of the ZC702 were not explored thoroughly since there was not enough time and nobody at the Department ever performed these measurements before. This report gives some basic system setup to for the measurements to be done but it does not explain the results completely, That is why a good next step would be to analyse power domains on the board and see how the use of different domains influences the measurements. This would bring better understanding of the measurements reported here.

References

[1] D. A. Patterson, J. L. Hennesy, “*Computer Organization and Design”*, 5th edition, Morgan Kaufman, 2014

[2] M. Popovic, “*Techniques for lowering instruction fetch power in microcontrollers”*, NTNU, December 2014.

[3] T.R. Halfhill, “*Achieving Energy Efficiency with EFM32 Microcontroller”*, The Linley Group, 2014.

[4] Kin, Johnson, Munish Gupta, and William H. Mangione-Smith. "*The filter cache: an energy efficient memory structure*", Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture. IEEE Computer Society, 1997.

[5] Tang, Weiyu, Rajesh Gupta, and Alexandru Nicolau, "*Design of a predictive filter cache for energy savings in high performance processor architectures*", Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on. IEEE, 2001.

[6] Bellas, Nikolaos, et al. "*Energy and performance improvements in microprocessor design using a loop cache*", Computer Design, 1999.(ICCD'99) International Conference on. IEEE, 1999.

[7] S. Hines, D. Whalley and G. Tyson, *“Guaranteeing Hits to Improve the Efficiency of a Small Instruction Cache”,* Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on. IEEE, 2007.

[8] Inoue, Koji, Vasily G. Moshnyaga, and K. Murakarni. "A *history-based I-cache for low-energy multimedia applications*", Low Power Electronics and Design, 2002. ISLPED'02. Proceedings of the 2002 International Symposium on. IEEE, 2002.

[9] Su, Ching-Long, and Alvin M. Despain. "*Cache design trade-offs for power and performance optimization: a case study*", Proceedings of the 1995 international symposium on Low power design. ACM, 1995.

[10] L. H. Lee, B. Moyer, J. Arends, *“Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops”,* Low Power Electronics and Design, 1999. Precedings, 1999. International Symposium on IEEE, 1999.

[11] *“Cortex M3 – Technical Reference Manual”*, ARM, 2006.

[12] “*Cortex-M0 Technical Reference Manual”*, ARM, 2009.

[13] Joseph Yiu, “*The Definitive Guide to the ARM Cortex-M0”*, Newnes, 2011.

[14] “*ARMv6-M Architecture Reference Manual*”, ARM, 2010.

[15] www.infocenter.arm.com

[16] *"ARM Cortex-M0 Design Start"*, ARM, 2010.

[17] "*AMBA 3 AHB-Lite Protocol specification",* ARM, 2010, available at www.arm.com

[18] C. Gahagan,"*Step by step guide to implementing the Cortex-M0 using a NEXYS2 FPGA board*", Louisianan Tech College of Engineering and Science, 2013.

[19] "*Xilinx 7Series FPGA Libraries Guide for HDL Designs*", Xilinx, 2012.

[20] "*ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide"*, Xilinx, 2015.

[21] P. Martos, F. Baglivo, "*Aplication Note: Cortex-M0 Implementation in the Nexys2 FPGA Board, A Step by Step Guide*", University of Buenos Aires, 2011.

[22] "*Vivado Design Suite User Guide - Power Analysis and Optimisation",* Xilinx 2015.

[23] "*Vivado Design Suite Tutorial - Power Analysis and Optimisation",* Xilinx 2015.

[24] "*ZC702 Built-In Self Test Flash Application*", Xilinx, 2013.

[25] "*Zynq-7000 All Programmable SoC Technical Reference Manual*", Xilinx, 2015.

[26] "*Zynq-7000 All Programmable SoC Low Power Techniques Part 2 – Measuring ZC702 Power using TI Fusion Power Designer Tech Tip*", tutorial available at: http://www.wiki.xilinx.com/Zynq-7000+AP+SoC+Low+Power+Techniques+part+2+-+Measuring+ZC702+Power+using+TI+Fusion+Power+Designer+Tech+Tip

[27] "*TLMS1100, TLMO1100, TLMY1100, TLMG1100, Standard 0603 SMD LED*" Vishay Semiconductors, datasheet, 2014.

[28] Marius Grannæs, Personal communication

Appendix A

Code A.1 – Software Implementation of TLC in Python

```
"""
Created on Sun Mar 29 05:03:07 2015

@author: Dell
"""
# ---------------------------------------------------------------------------
----

import sys
import csv

def Simulate(dataSet, cache_size ):
####### states:
#IDLE:         0
#PSEUDO_IDLE:   1
#COUNT:         2
#PSEUDO_COUNT:  3
#FILL:          4
#PSEUDO_FILL:   5
#ACTIVE:        6
#PSEUDO_ACTIVE: 7

  readpwrs = {}
  writepwrs = {}
  flashpwr = 180

  readpwrs[16] = 1.875
  writepwrs[16] = 2.5
  readpwrs[32] = 3.75
  writepwrs[32] = 5
  readpwrs[64] = 7.5
  writepwrs[64] = 10
  readpwrs[128] = 15
  writepwrs[128] = 20
  readpwrs[256] = 17
  writepwrs[256] = 24
  readpwrs[384] = 19
  writepwrs[384] = 28
  readpwrs[512] = 21
  writepwrs[512] = 30
  readpwrs[640] = 23
  writepwrs[640] = 34
  readpwrs[768] = 25
  writepwrs[768] = 37
  readpwrs[896] = 27
  writepwrs[896] = 41
  readpwrs[1024] = 28
  writepwrs[1024] = 44
  readpwrs[1536] = 35
  writepwrs[1536] = 42
  readpwrs[2048] = 37
  writepwrs[2048] = 46
  readpwrs[4096] = 37
  writepwrs[4096] = 46
  sram_reads  = 0
```

```
  sram_writes = 0
  valid = 0
  hit_rate = 0
  rw_ratio = 0
  old = 0
  old_old = 0
  new = 0
  position = 0


  state = 0

  cache = []
  counter = 0
  loop_size = 0
  times_executed = 0
  firsts=[]
  sbbs=[]
  loop_sz=[]
  inst_no=[]
  times_ex=[]
  sbb = 0
  first = 0
  nextstate = 0
  flash_reads = 0


  # Open dataSet
  try:
    file = open(dataSet,'r')
  except:
    sys.stderr.write('Could not open dataSet\n')
    sys.exit()

  # Skip the first line in file (header)
  file.readline()

  # -----------------------------------------------------------------------
----
  # Each line contains one address

  for line in file:
    position=position+1
    # Split out the address, ignore time for now
    try:
      [timeString, addrHex ] = line.split(',')
    except:
      sys.stderr.write('Bad: '+line+"\n")
      continue

    try:
      a = int(addrHex, 16)
      #print a
      if (a > 0) and (a < 536870911):
        # don't save idles and dummy values as last address
        valid = valid + 1
        old_old = old
        old = new
        new = a
      else:
        # Ignore idles
```

```
            continue
        except:
          sys.stderr.write('Could not convert %s to int\n' % addrHex)
          continue

        if (valid == 1):
            old_old = a
            old = a
            new = a

        if state == 0: # idle state
            cache = []
            if (new == old+4): # idle => idle
                nextstate = 0
                flash_reads = flash_reads + 1
            else: # idle => count
                if (new < old):
                    nextstate = 2
                    sbb = old
                    first = new
                    counter = 1
                    flash_reads = flash_reads + 1
                else: # idle => pseudo_idle
                    nextstate = 1
                    flash_reads = flash_reads + 1

        if state == 1: # PSEUDO_IDLE
            if (new == old + 4) or (new == old_old + 4): # pseudo_idle => idle
                old = old_old
                nextstate = 0
                flash_reads = flash_reads + 1
            else:
                if (new < old): # pseudo_idle => count
                    nextstate = 2
                    sbb = old
                    first = new
                    counter = 1
                    flash_reads = flash_reads + 1
                else: # stay pseudo_idle
                    nextstate = 1
                    flash_reads = flash_reads + 1


        if state == 2: # COUNT
            if (new == old + 4) and (new <= sbb):
                nextstate = 2  # stay in count
                counter = counter + 1
                flash_reads=flash_reads + 1
            else:
                if (new > old + 4):
                    nextstate = 3 # count => pseudo_count
                    flash_reads=flash_reads + 1
                else:
                    if (new == first) and (old == sbb) and (counter <=
    cache_size): # count => fill
                        nextstate = 4
                        sram_writes = sram_writes + 1
                        loop_size = counter
                        counter = 1
                        cache. append (new)
                        flash_reads = flash_reads + 1
```

```
                else: # count => idle
                    nextstate = 0
                    flash_reads=flash_reads + 1
                    counter = 0

    if state == 3: # PSEUDO_COUNT
        if (new == old_old + 4): # pseudo_count => count
            old = old_old
            counter = counter + 1
            nextstate = 2
            flash_reads = flash_reads + 1
        else:
            if (new == first) and (old_old == sbb) and (counter <=
cache_size):  # pseudo_count => fill
                nextstate = 4
                sram_writes = sram_writes + 1
                old = old_old
                loop_size = counter
                cache. append (new)
                flash_reads = flash_reads + 1
            else:  # pseudo_count => idle
                nextstate = 0
                flash_reads = flash_reads + 1
                counter = 0

    if state == 4: # FILL
        if (new == old + 4) and (new <= sbb) and (counter < loop_size) :
            nextstate = 4   # fill => fill
            counter = counter + 1
            sram_writes=sram_writes + 1
            cache. append (new)
            flash_reads = flash_reads + 1
        else:
            if (new > old + 4): # fill => pseudo_fill
                nextstate = 5
                flash_reads=flash_reads + 1
            else:
                if (new == first) and (old == sbb) and (counter ==
loop_size):  # fill => active
                    nextstate = 6
                    sram_reads = sram_reads + 1
                    counter = 1
                    loop_sz.append(loop_size)
                else:
                    nextstate = 0
                    flash_reads=flash_reads + 1
                    counter = 0

    if state == 5: # PSEUDO_FILL
        if (new == old_old + 4) and (counter < loop_size): # pseudo_fill=>
fill
            old = old_old
            nextstate = 4
            counter = counter + 1
            flash_reads = flash_reads + 1
            sram_writes = sram_writes + 1
            cache. append (new)
        else:
            if (new == first) and (old_old == sbb) and (counter ==
loop_size): # pseudo_fill => active
                nextstate = 6
```

```
        sram_reads = sram_reads+1
        old = old_old
        counter = 1
        loop_sz.append(loop_size)
      else: # pseudo_fill => idle
        nextstate = 0
        flash_reads = flash_reads + 1
        counter = 0

  if state == 6: # ACTIVE
    if (new == old + 4) and (new <= sbb): # stay in active
      sram_reads=sram_reads + 1
      nextstate = 6
      counter = counter + 1
      if (new not in cache): # sanity check
         print("1. Tried to read from cache, but value not there")
         print new
         print cache
         sys.exit()
    else:
      if (new == first) and (old == sbb) and (counter == loop_size):
# stay in active but it is the begining of the loop again
        sram_reads=sram_reads + 1
        nextstate = 6
        counter = 1
        times_executed = times_executed + 1
        if (new not in cache): # sanity check
           print("1. Tried to read from cache, but value not
there")
           print new
           print cache
           sys.exit()
      else:
        if (new > old + 4):   # ACTIVE => pseudo_active
          nextstate = 7
          flash_reads=flash_reads + 1
        else:    # ACTIVE => idle
          nextstate = 0
          flash_reads=flash_reads + 1
          counter = 0
          loop_sz.append(loop_size)
          times_ex.append(times_executed)
          sbbs.append(sbb)
          firsts.append(first)
          inst_no.append(position)

  if state == 7: # PSEUDO_ACTIVE
    if (new == old_old + 4) and (new <= sbb) and (counter < loop_size
):  # pseudo_active => active
      old = old_old
      nextstate = 6
      counter = counter + 1
      sram_reads = sram_reads + 1
    else:
      if (new == first and old_old == sbb) and (counter == loop_size
):   # pseudo_active => active but begining of the loop again
        nextstate = 6
        counter = 1
        sram_reads = sram_reads + 1
        times_executed = times_executed + 1
      else:   # pseudo_active => idle
```

```
                nextstate = 0
                flash_reads=flash_reads + 1
                counter = 0
                loop_sz.append(loop_size)
                times_ex.append(times_executed)
                sbbs.append(sbb)
                firsts.append(first)
                inst_no.append(position)

    state = nextstate
  file.close()


  power_nocache=(flash_reads + sram_reads)*flashpwr
  power_withcache=flash_reads*flashpwr + sram_reads*readpwrs[cache_size*4]
+ sram_writes*writepwrs[cache_size*4]
  hit_rate=sram_reads/float(flash_reads + sram_reads)
  rw_ratio=sram_reads/float(sram_writes)
  x=power_withcache/float(power_nocache)
  power_savings = 1.0 - x
  #result='results_final_' + cache_size + dataSet
  result='results_final_' + str(cache_size) + dataSet
  with open(result, 'wb') as result_file:
    csv_writer = csv.writer(result_file)
    for row in range (1):

       csv_writer.writerow([sram_writes] + ['SRAM Writes:'])
       csv_writer.writerow([sram_reads] + ['SRAM Reads:'])
       csv_writer.writerow([flash_reads] + ['Flash reads:'])
       csv_writer.writerow([' '])
       csv_writer.writerow(['Loop size:'] + loop_sz)
       csv_writer.writerow(['Times executed:'] + times_ex)
       csv_writer.writerow(['First:'] + firsts)
       csv_writer.writerow(['Sbb:'] + sbbs)
       csv_writer.writerow(['Last instruction of the loop (its row in
dataset) read from SRAM:'] + inst_no)
       csv_writer.writerow([cache_size] + ['Cache size'] )
       csv_writer.writerow([power_nocache] + ['Total Power With No Cache']
)
       csv_writer.writerow([power_withcache] + ['Total Power With Cache'] )
       csv_writer.writerow([power_savings] + ['Power Savings'] )
       csv_writer.writerow([hit_rate] + ['Hit Rate:'] )
       csv_writer.writerow([rw_ratio] + ['RW Ratio'] )
  result_file.close()

datasets = ["coremark.csv", "primes.csv", "dijkstra.csv", "emlcd.csv",
"preamp.csv", "sha.csv", "touch.csv"]

#datasets = ["coremark.csv"]

for d in datasets:
  print d
  for size in [4,6,8,10,12,14,16,18,20,22,24,26,28,30,32]:
      Simulate(d, size)
```

Appendix B

Code B.1 – Loop Cache Controller Verilog code

```
module Loop_Cache_Controller
(
input wire clk,
input wire n_reset,
input wire global_cache_enable,
input wire htrans,
input wire hprot,
input wire [12:0] new_address,
output reg main_cache,
output reg cache_write_enable
);

parameter cache_size=16; // size of the cache in words
localparam IDLE=0, FILL=1, ACTIVE=2;

(* mark_debug = "true" *) reg [1:0] present_state, next_state;
(* mark_debug = "true" *) reg prot_trans;
(* mark_debug = "true" *) reg [12:0] branch;
(* mark_debug = "true" *) reg [12:0] branch_target;
(* mark_debug = "true" *) reg [12:0] previous_address;
(* mark_debug = "true" *) reg loop_detector;


always @(posedge clk or negedge n_reset)
begin
    if (~n_reset)
        previous_address<=13'b0000000000000;
    else
        begin
            if (global_cache_enable)
                previous_address<=new_address;
            else
                previous_address<=13'b0000000000000;
        end
end


always @(posedge clk or negedge n_reset)
begin
    if (~n_reset)
        present_state <= IDLE;
    else
        present_state <= next_state;
end

always @(posedge clk or negedge n_reset)
begin
if (~n_reset)
    begin
    branch<= 13'b0000000000000;
    branch_target<= 13'b0000000000000;
    end
else
    if (loop_detector)
```

```
        begin
        branch<=previous_address;
        branch_target<=new_address;
        end
end

always @(*)
begin
next_state=present_state;
main_cache=1'b1;
cache_write_enable=1'b0;
loop_detector=1'b0;
prot_trans=1'b0;
if (global_cache_enable)
    begin
    prot_trans = ~hprot & htrans;
    case (present_state)
        IDLE: begin
            cache_write_enable = 1'b0;
            main_cache = 1'b1;
            if  (prot_trans && ($signed(previous_address-new_address) <=
(cache_size-1)) && ($signed (previous_address-new_address) > 1))
                begin
                next_state = FILL;
                loop_detector=1'b1;
                end
            else
                next_state = IDLE;
            end

        FILL: begin
              if (~hprot)
                begin
                if ((new_address==branch_target) &&
(previous_address==branch))
                    begin
                    next_state = ACTIVE;
                    main_cache = 1'b0;
                    cache_write_enable=1'b1;
                    end
                else
                    begin
                    if (((new_address==previous_address) ||
(new_address==previous_address+1)) && (new_address<=branch))
                        begin
                        next_state = FILL;
                        cache_write_enable=1'b1;
                        main_cache = 1'b1;
                        end
                    else
                        begin
                        next_state = IDLE;
                        cache_write_enable=1'b1;
                        main_cache = 1'b1;
                        end
                    end
                end
              else
                begin
                cache_write_enable=1'b1;
                main_cache = 1'b1;
```

```
                next_state = IDLE;
                end
              end


        ACTIVE: begin
                if (~hprot)
                    begin
                    if (new_address<=branch &&
((new_address==previous_address)|| (new_address==previous_address+1)  ||
(new_address==branch_target && previous_address==branch)))
                        begin
                        next_state = ACTIVE;
                        cache_write_enable=1'b0;
                        main_cache = 1'b0;
                        end
                    else
                        begin
                        next_state = IDLE;
                        cache_write_enable=1'b0;
                        main_cache = 1'b1;
                        end
                    end
                else
                    begin
                    next_state = IDLE;
                    cache_write_enable=1'b0;
                    main_cache = 1'b1;
                    end
                end

    endcase
    end
else // default values for the case of global_enable=0
    begin
    cache_write_enable=1'b0;
    main_cache=1'b1;
    next_state = IDLE;
    prot_trans=1'b0;
    loop_detector=1'b0;
    end
end
endmodule
```

Code B.2 - Loop Cache Verilog code

```
module loop_cache #(parameter ADDRWIDTH=4)
(
input wire clk,
input  wire n_reset,
input wire htrans,
input wire enable_cache,
input wire write_enable,
input wire [31:0] data_in,
input wire [ADDRWIDTH-1:0] address,
output  reg [31:0] data_out
);
```

```
integer i;
(* mark_debug = "true" *) reg htrans_a;
(* mark_debug = "true" *) reg [ADDRWIDTH-1:0] address_a;
(* mark_debug = "true" *) reg [31:0] cache [2**ADDRWIDTH-1:0];

//ADDRESS phase
always @(posedge clk ,negedge n_reset)
begin
    if (~n_reset)
        begin
        address_a <= 4'b0000;
        data_out<= 32'h00000000;
        for (i=0; i<2**ADDRWIDTH; i=i+1)
        cache[i] <= 32'h00000000;
        end
    else
        begin
        htrans_a<=htrans;
        address_a<=address;
        if (write_enable&& htrans_a)
            cache[address_a] <= data_in;// fill

        if (enable_cache && htrans)
            data_out<=cache[address];
        end
end
endmodule
```