IMPROVING THE ENERGY EFFICIENCY OF MICROCONTROLLERS

Maja Popovic, RT-RK Banjaluka

Marius Graneas, Silicon Labs Oslo

Donn Morrison, NTNU Trondheim

Branko Dokic, ETF Banjaluka

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. A technique called Tight Loop Cache, believed as the most promising when power optimization is concerned, was chosen to be implemented and evaluated. 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). This paper focuses only on the hardware implementation of the system and its results. 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.

1. INTRODUCTION

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. 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 paper briefly explains only the main principles of the work performed while the author was doing her Master Thesis at Norwegian Univesity for Science and Technology in Trondheim, Oslo in cooperation with Silicon Labs. More details about implementation and results can be found in the Master thesis itself [5].

2. TIGHT LOOP CACHE

Tight Loop Cache, a technique proposed in [2] 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 is a cache miss since the Controller had the early notion whether next fetch is 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 laid down on the detection of the so called sbb instruction (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 instruction indicated the data was already in0 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.

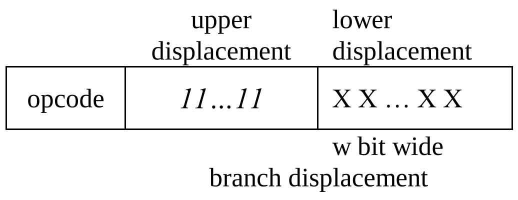


Figure 2.1 - sbb instruction format

Upper displacement containing all ones suggested that the branching was going to be backwards (end of the loop is encountered) whereas the lower displacement field determined how long the backward jump was going to be (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 an architecture where each memory location contained only one instruction and program counter was incremented by 1 to access next instruction).

This made sure that 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 2.1 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).

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 instruction in the instruction stream. If the controller determined that there was an sbb instruction (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.

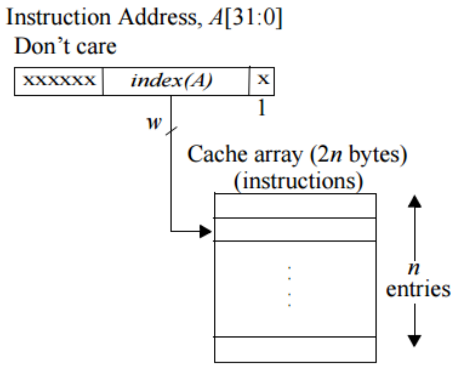
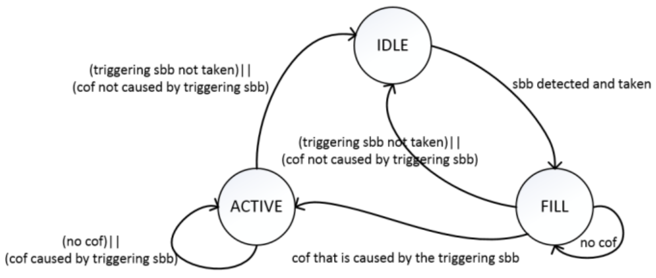


Figure 2.1 – Loop cache organization and access [2]

The sbb instruction that forced the Controller to enter FILL state was called triggering sbb instruction. 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).



IDLE
ACTIVE
FILL
(triggering sbb not taken) | |
(cof not caused by triggering sbb)
sbb detected and taken
(triggering sbb not taken) | |
(cof not caused by triggering sbb)
(no cof) | |
(cof caused by triggering sbb)
cof that is caused by the triggering sbb
no cof

Figure 2.2 – Loop cache controller state machine

Controller went back to IDLE state in case it encounterd 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 for Controller to 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 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 which is shown in the Figure 2.3.

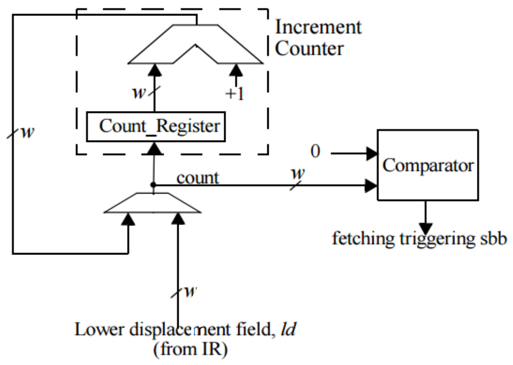


Figure 2.3 – Design of the loop counter [2]

The Controller was initially in IDLE state and stayed there until sbb was detected in the decode 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 in the execute stage, 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 loaded with the ld field of triggering sbb again. This meant that the execution of the loop started from the beginning again and when the counter reached zero again, the same process was repeated. Using this mechanism, the Controller knew when a cof is caused by triggering sbb by examining the value in the Increment Register.

The original technique of using tight loop cache and loop controller, proposed in [2] and briefly explained here, was implemented both in software and hardware with slight modifications in author's Master Thesis. This report will only present some of the results and implementationh details about the hardware solution.

3. HARDWARE IMPLEMENTATION

Project was dealing with Silicon Labs 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 …) This section shows how TLC system was built by integrating additional hardware modules around Cortex-M0 core delivered by ARM as an obfuscated verilog code. Both Loop Cache Controller, Loop Cache itself and other auxiliary modules were designed in Verilog by the author.

The system was implemented as an AHB bus system (communication bus used by ARM, more details about it can be found in [3]) 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.

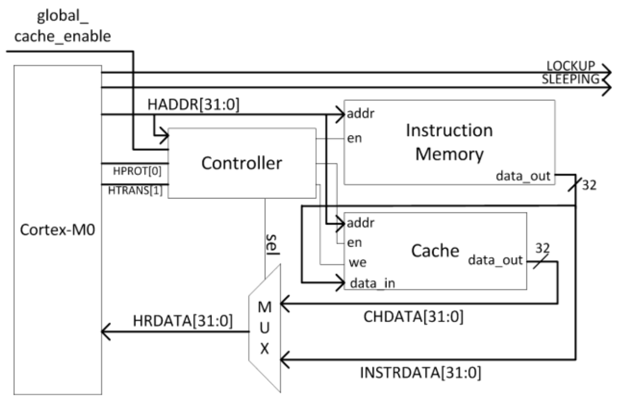


Figure 3.1– Introducing Loop cache into the core system

As it can be seen from Figure 3.1, there were three new components added to the initial core-memory system: the Loop Cache Controller, Loop Cache and the multiplexer and the implementation of all of them will be explained in next sections.

3.1. Loop Cache Controller

As it was explained in Chapter 2, 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 is the same loop that is being executed all the time. Two approaches of loop detection were discussed: the one from Chapter 2, the original loop detection principle whereas the second approach was explained in the software implemantation of the system in [5]. These two approaches are shown in Figure 3.2 where only differences between them are shown, i.e. input signals.

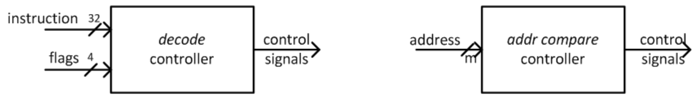


Figure 3.2 – Two different Loop Cache Controller implementations and their interfaces

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 taken. Since Cortex-M0 had no branch status signal as it would be case if Cortex-M3 was used, the second input would have been 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). More details about the ARM Cortex-M0 can be found in [3] and [4] whereas details of importantance to this project can be found in [5]. The Controller would perform decoding of the instruction, if a potential branch would have been decoded it would check status of the corresponding flag and determine whether to branch or not. The next step would be to calculate the branching offset and initiate counter register from Figure 2.3 with that value. The rest of the system would behave as described in Chapter 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 is explained in details in [5].

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. 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 unfortunatelly 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 3.3 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 singal main/cache which controls where data should be read from was set and reset at particular rising clock edges as shown in Figure 3.3.

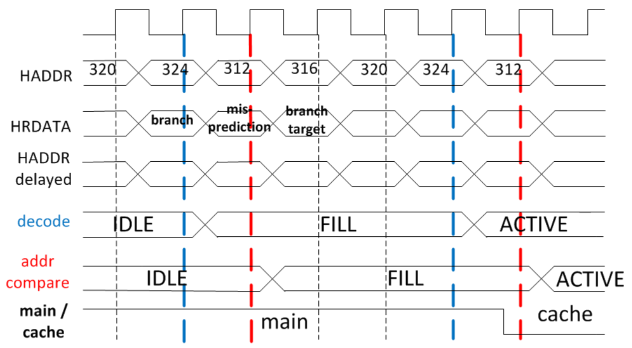


Figure 3.3 – Timing comparison between two different controller implementations

As it can be seen in Figure 3.3, for the address compare implementation, the Controller changes 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 3.3). The way this was really happening when simulating the system itself is shown in Figure 3.4 where it can be seen how the cache is behaving correctly and according to AHB transfer rules with an address and a data phase.

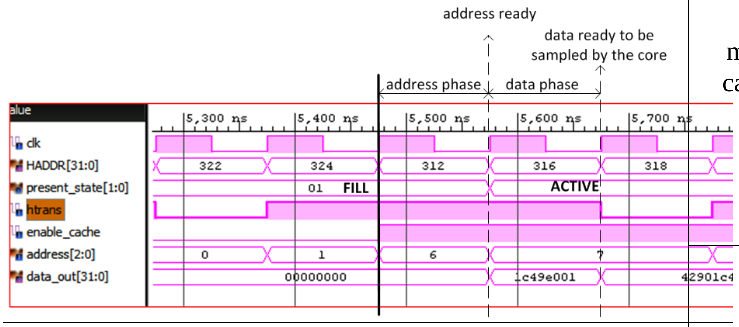


Figure 3.4 – Chosen controller implementation and its state switching and control signal toggling

Apart from address bus as input shown in Figure 3.1, the Controller had HTRANS[1] and HPROT[0] inputs which were indicating that instruction transaction needed to be performed [3]. 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 have been used as source. This is also illustrated in Figure 3.5 where critical signal changes are shown in red.

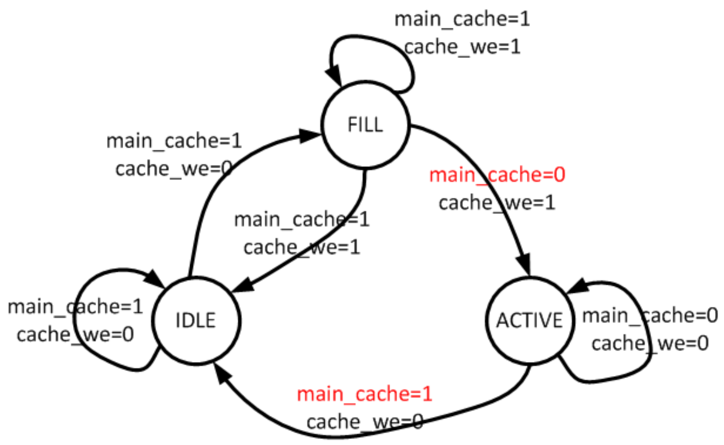


Figure 3.5 – Controller State Machine with output control signals

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 3.1 (values shown in red are the critical ones, the ones that change before the state changes).

Table 3.1 – 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 the data from the address currently on the bus should be written into the cache (on the next clock rising edge)	0	1	0

Table 3.2 shows how input enable signal of the instruction memory was depending 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 just by inverting this signal.

Table 3.2 – 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. Loop Cache

As it can be seen from Figure 3.1, HTRANS[1] was used as input into the cache as well although it was already used to create enable signal which could be maybe then disputed to be redundant. This signal is 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 is writing data in is also when cycle delayed (address_a). The writing process is shown in Figure 3.6. 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. Least significant 2 bits from the address bus were completely neglected in the whole system since both memories are word addressable.

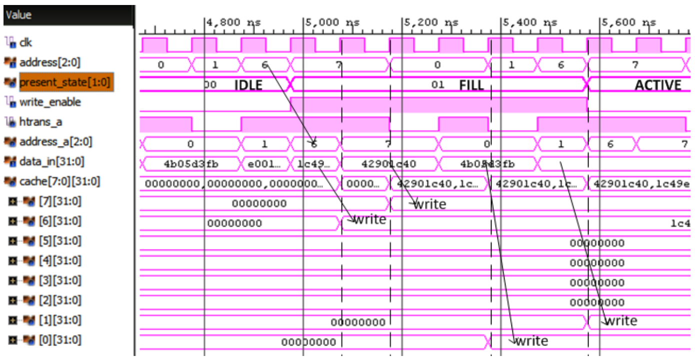


Figure 3.6- Process of filling the Loop Cache

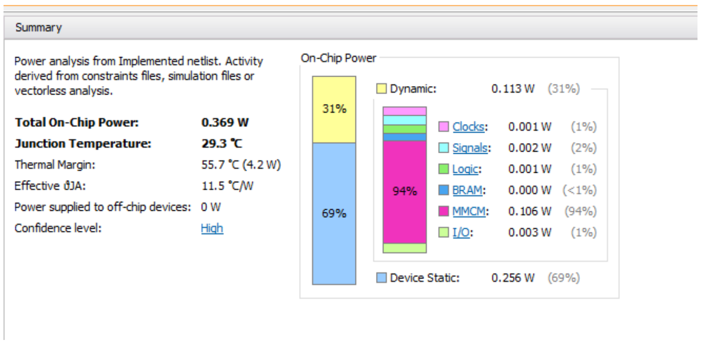
Figure 3.6 shows how cache writing process was performed and clearly illustrates one of the benefit of this implementation of the loop cache where instructions did not need to be aligned to any starting address whereas they still remained consecutive (first location to be written into was 6, the second 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 within loop cache controller module that was containing the previous address already. If a part of that signal (lower n bits if n is the width of the address buds of the cache) was taken from there already, some hardware savings could be gained.

4. TESTING SETUP AND RESULTS

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.

As it was mentioned, it was 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. 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. Not all the results will be presented here. More details can be found in [5]. Table 4.1 shows power reports for the case of Instruction Memory of size 32KB and the cache size of 64B. 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.


Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.369 W**
Junction Temperature:	**29.3 ℃**
Thermal Margin:	55.7 ℃ (4.2 W)
Effective ϑJA:	11.5 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

On-Chip Power

Dynamic:	0.113 W	(31%)
Clocks:	0.001 W	(1%)
Signals:	0.002 W	(2%)
Logic:	0.001 W	(1%)
BRAM:	0.000 W	(<1%)
MMCM:	0.106 W	(94%)
I/O:	0.003 W	(1%)
Device Static:	0.256 W	(69%)

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

Table 4.1 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 4.1 – 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
no cache	power [mW]	104	103	100	117	115	114
	confidence	low	low	high	low	low	high
with cache	power [mW]	108	107	106	116	114	113
	confidence	low	low	high	low	low	high

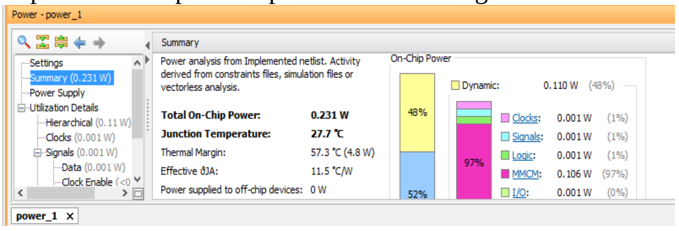
Table 4.2 – Power report generation details for different measurement configurations

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.

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.

Having a look at the results from Table 4.1, 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.


Power - power_1

Summary

- Settings
- Summary (0.231 W)
- Power Supply
- Utilization Details
  - Hierarchical (0.11 W)
  - Clocks (0.001 W)
  - Signals (0.001 W)
    - Data (0.001 W)
    - Clock Enable (<0

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.231 W**
Junction Temperature:	**27.7 ℃**
Thermal Margin:	57.3 ℃ (4.8 W)
Effective ϑJA:	11.5 ℃/W
Power supplied to off-chip devices:	0 W

On-Chip Power

Dynamic:	0.110 W	(48%)
Clocks:	0.001 W	(1%)
Signals:	0.001 W	(1%)
Logic:	0.001 W	(1%)
MMCM:	0.106 W	(97%)
I/O:	0.001 W	(0%)

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. 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.

More information about the system setup and measurements performed as well as details of the software implementation can be found in [5].

5. CONCLUSION

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. 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.

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.

6. REFERENCES

[1] D.A. Petterson, J.L. Hennessy, „Computer Organisation and Design“, 5th edition, Morgan Kaufman, 2014

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

[3] „AMBA 3 AHB Lite Protocol Specification“, ARM, 2010, available at www.arm.com

[4] „ARM Cortex-M0 Design Start“,ARM, 2010

[5] M. Popovic, „Improving the energy efficiency of a microcontroller instruction fetch using tight loop cache“, NTNU, 2015