FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Power side channel information leakage of a microcontroller
Student:	Marina Shchavleva
Supervisor:	Ing. Jiří Buček
Study Programme:	Informatics
Study Branch:	Computer engineering
Department:	Department of Digital Design
Validity:	Until the end of summer semester 2018/19

Instructions

Study the methods of power analysis attacks on microcontrollers. Design and implement an experiment with the aim to analyze the dependencies of power consumption on individual instructions of an AVR microcontroller. Dependency on instruction type, operand value, and instruction address will be analyzed. Create an experimental measurement setup, execute necessary measurements, and analyze the results.

References

Will be provided by the supervisor.

doc. Ing. Hana Kubátová, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 2, 2018

Acknowledgements

I would like to thank my supervisor Ing. Jiří Buček for introduction to power analysis, for his help and valuable advices. Also I want to thank my partner, Ladislav, for his support and, of course, my parents, who sent me to study abroad in the first place, for their endless love and support throughout my study here and their patience.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 15, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Shchavleva, Marina. *Power side channel information leakage of a microcontroller.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Spotřeba zařízení může prozradit mnoho informací ohledně jeho vnitřní struktury a toku dat který toto zařízení zpracovává. Jednoduchá a Diferenciální odběrové analýzy jsou v odborné literatuře široce probrané techniky pro útoky postranními kanály. Tato práce uvádí čtenáře do problematiky analýzy spotřeby a dává krátký přehled metod, které jsou k tomu používány.

Hlavním cílem této bakalářské práce je analýza spotřeby mikrořadiče při vykonávání různých operací z jeho instrukční sady, konkrétně se jedná o mikrořadič ATMega163, vestavěný do smartkarty. Jsou probrány důležité aspekty toho, jak mikrořadič funguje a jak zpracovává instrukce: instrukční cyklus, adresy v paměti programu, hodnoty operandů a tok dat. Kromě toho práce popisuje jak typ instrukce ovlivňuje spotřebu, jinak řečeno, jaké procesy se odehrávají uvnitř mikrořadiče při zpracování dat a kontrole toku programu.

Klíčová slova útoky postranními kanály, analýza spotřeby, SPA, DPA, spotřeba mikrořadičů, ATMega163, instrukční sada

Abstract

Through power consumption of a device a lot of information about it internal structure and data it processes can be leaked. Simple and Differential power analysis are well described techniques for such side channel attacks. This work gives brief introduction to the idea of power side channel analysis and methods it uses.

The main objective of this Bachelor's thesis is power side channel analysis of a microcontroller's instruction set specifically ATMega163 which is embedded in a smartcard. Important aspects of microcontroller's operation and it's instructions are discussed: instruction execution cycle, address in a Program Memory, operand values and data flow. Also how instruction type affects power consumption, in other words, what does microcontroller internally do to process data or manage program flow.

Keywords side channel attacks, power analysis, SPA, DPA, microcontroller's power consumption, ATMega163, instruction set

Contents

List of Figures

Introduction

Often, when designing cryptographic algorithm, developer is concerned mainly with it's mathematical properties: if not mathematically unbreakable, algorithm needs to be practically unbreakable. This is the first Kerckhoffs's principle[6], set by Auguste Kerckhoffs among others in nineteenth century that is still relevant. On the flip side, algorithm needs implementation, and programs do not exist in vacuum.

Secure implementation is as important as secure algorithm. Without well thought implementation security of a whole system might be compromised. Device might produce some side-effects, like different time of computation given different data, noise produced by hardware device itself, or power consumption that is dependent on processed data. Side-channel attacks based on such properties of a device may pose threat to security of a system. Power side channel analysis attacks target power consumption of a device.

Main objective of this work is to analyze power consumption of a microcontroller, how it's internal architecture can affect power traces during execution of various instructions. To perform this analysis the preparation stages should be taken, such as picking up a microcontroller to test, creating measurement setup and design a number of tests.

Work structure

Chapter 1 introduces reader to the idea behind power analysis attacks and describe main methods of it's implementation.

In chapter 2 I describe test prerequisites for practical part of the work to be done: what measurement setup I had, which tools I used to acquire and process power traces, what problems I encountered and how I dealt with them.

Chapter 3 present the practical output of my work. Here I describe internal architecture of ATMega163 — microcontroller used for analysis — and how it affects power consumption. I continue with describing instruction set and

types of instructions. I analyze power consumption of individual instructions with dependency on the type of instruction, operands and data they process.

Chapter 1

Power side channel analysis attacks

This chapter will provide introduction to power side channel analysis attacks, and what methods of power analysis exist. I will briefly mention historical development and significant concepts in this topic and how it influenced my work.

1.1 Power analysis attacks

First to introduce cryptographic community to power analysis attacks were Paul Kocher, Joshua Jaffe and Benjamin Jun, in their report "Differential power analysis" in the year 1998[7]. Despite the name, they've brought up two methods of power analysis: Simple Power Analysis and Differential Power Analysis. Simple power analysis targets implementation details, such as conditional pieces of code, which are or are not executed in dependency on data that are being processed and structures such as loops or any repetitive pieces of computation. Differential power analysis exploits statistical properties of data and respective power consumption trace.

1.1.1 Simple Power Analysis

To successfully implement Simple power analysis attack only a few power traces are sufficient, even one power trace may expose all information that attacker needs. Power consumption is analyzed along time axis, because important part of it is algorithm itself. Great example of Simple power analysis attack is provided by Marc Joye[5]. In this example Simple Power Analysis attack was performed against basic square-and-multiply implementation, see algorithm 1.

The problematic part here is that at the moment of checking conditional statement, when condition is not met (bit i of d is 0) execution of this iteration

Algorithm 1 Basic square-and-multiply algorithm.

$k \leftarrow bitsize(d)$
$y \leftarrow x$
for $i = k - 2$ **downto** 0 **do**
  $y \leftarrow y^2 (\mathrm{mod} n)$
  if bit i of d is 1 **then** $y \leftarrow y * x (\mathrm{mod} n)$
  end if
end for

of a loop is shorter in comparison to iterations where condition is met. Which is afterwards can be seen at the power trace of a device, that used straightforward implementation of this algorithm, as seen on figure 1.1.

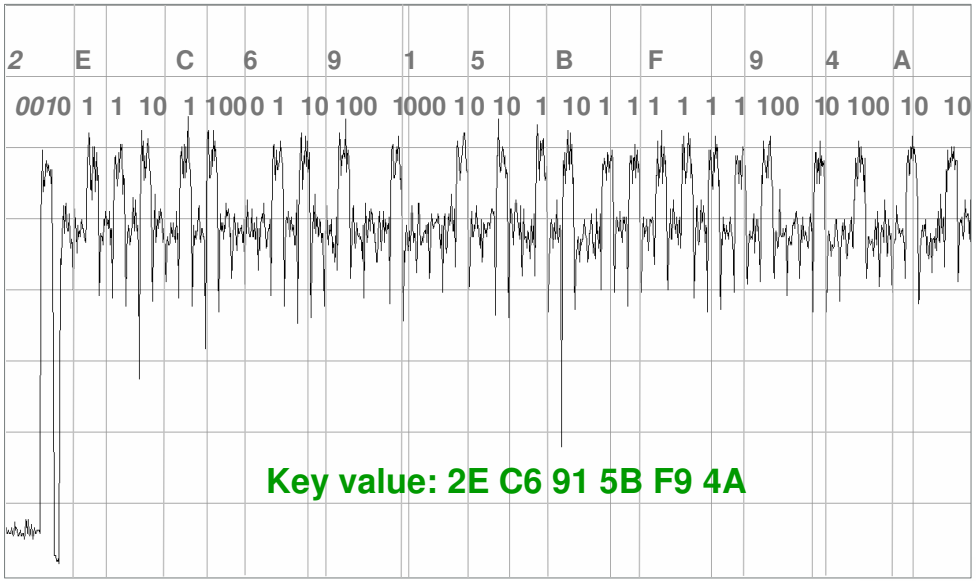


Figure 1.1: SPA trace of basic square-and-multiply algorithm [5].

Another typical example is observation of repeated structures, for example, rounds in block ciphers. Although just from a power consumption of a block or groups of blocks, Simple Power Analysis is used as supplement to Differential power analysis to "remove irrelevant regions" [8].

Rita Mayer-Sommer in [9] pointed out, that SPA can be applied to individual instructions, not only to conditional branching instructions. She objected the statement from [7], that SPA is easily prevented "by avoiding conditional branching and jumps" and presented results of power consumption measurement of `MOV` instruction in PIC16C84 chip. Those measurements show that by a mere SPA attacker can figure out Hamming weight of data processed, which creates a threat no one before anticipated.

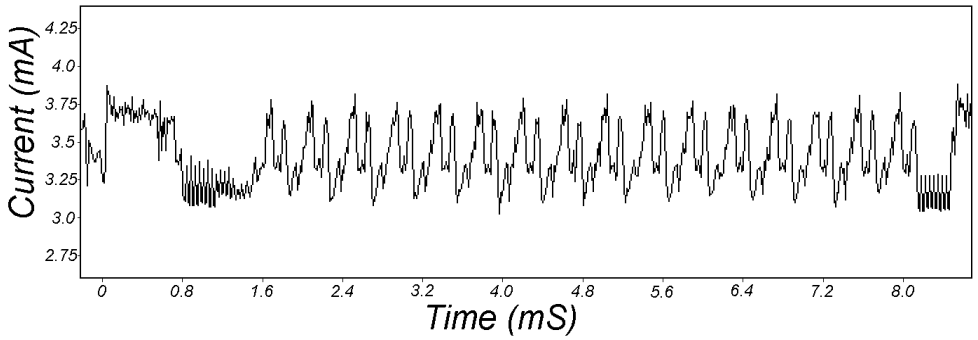


Figure 1.2: SPA trace showing an entire DES operation [7].

1.1.2 Differential power Analysis

In contrast to Simple Power Analysis, Differential Power Analysis requires a large amount of power traces to perform it, therefore it is usually necessary to physically possess one attacked device[12]. On the upside, attacker doesn't need to know implementation details about cryptographic device, all that he needs is "being merely informed about general code structure"[9].

Generally, to perform Differential power analysis attack few steps need to be taken according to Mangard[12]:

1. **Choose intermediate value of executed algorithm:** it needs to be a result of a function $f(d, k)$, where d is a non-constant data value (either plain text or cipher text) and k is constant key;

2. **Measure power consumption:** attacker needs to obtain power traces of ciphering/deciphering data, known to the attacker. It is important for resulting power traces matrix to be aligned, i.e. at every point in time (column in matrix) needs to be performed same operation;

3. **Calculate hypothesis:** attacker has to have some model (more on models later in text), by which he can create hypothesis about intermediate value in dependency on key;

4. **Mapping hypothesis to power traces and compare them.**

There are two basic models for calculating hypothesis, based on Hamming Weight[1] and on Hamming Distance between two intermediate values[2].

Technique, that is widely used to calculate liner correlation between power traces and hypothesis is sample Pearson correlation coefficient[3]. Formula for it reads as follows

$$r = \frac{n\sum x_i y_i - (\sum x_i \sum y_i)}{\sqrt{[n\sum x_i^2 - (\sum x_i)^2][n\sum y_i^2 - (\sum y_i)^2]}}$$

where n is the sample size, x_i and y_i are samples at i position in the set.

1.1.3 High-Order Differential Power Analysis

Number of countermeasures to DPA attacks exist, such as masking, insertion of random operations, that do not affect the computation, shuffling and so on, discussed at great detail by Mangard[12]. But for every countermeasure there is better yet attack scheme.

Higher-order (second or more) DPA attacks deal with not one, but with a number of points in measurement. This attack requires even more power traces for it to success, but is able to deal with various standard countermeasures against first-order DPA attack[4].

1.2 Summary

Thematically, my work is close to that of Rita Mayer-Sommer I mentioned above while describing Simple Power Analysis. Before discovering her work, I came up with idea of testing instructions against known values in order to see, how instructions behave with regard of their power consumption.

As she stated in [9], SPA attack can be performed against individual instructions. With that in mind, I would like to say that objective of this work is to perform Differential Power Analysis, so future works could rely on it to perform Simple Power Analysis. Difference between her work and mine is that here I provide more comprehensive exploration of almost every instruction, with more detailed overview of timings[3].

[1] Hamming weight of a value is number of ones in it's binary representation.

[2] Hamming distance between two values is Hamming weigh of a result of operation "exclusive or" between the two.

[3] Not to mention that we analyze different microcontrollers

Chapter 2

Test design and measurement setup

In this chapter i would like to present my measurement setup, tools and software I used to obtain and process power traces. Note about the order of sections: at the first glance it may seem illogical to start with measurement setup and continue with test design; after all, tests had to be designed before any measurement could take place. I wanted to swap those chapters in order to make transition from design to implementation more smooth.

So I would like to start with measurement setup and explain necessary steps that I took to get power traces that are prepared for analysis, then proceed with logic behind test design, and at the end explain my analysis software choice.

2.1 Measurement setup

Microcontroller under test in this work is ATMega163, embedded in a smartcard. This allowed me to use tools, that were available for smartcard communcation and power consumption analysis.

2.1.1 Obtaining power traces: SC Power Measurement

To obtain power traces I used Agilent oscilloscope MS06104A that's communicating with a computer through SC Power Measurement program from course MI-BHW.16, created by Ing. Jiří Buček and Ing. Vyleta Petr[10]. This program performs arbitrary amount of power measurements through oscilloscope. It communicates with a smartcard through a reader. SC Power Measurement creates four output files:

- **traces.bin**, binary file that contains measurement itself,
- **traceLength.txt**, text file with the number of samples per trace,

- **plaintext.txt**, text file with randomly generated plain data of a size 16 bytes, number of "plain texts" generated is respective with number of power traces collected in **traces.bin**,

- **ciphertext.txt**, text file with respective to the **plaintext.txt** encrypted texts.

I made slight changes to suit my needs. Firstly, I was not interested in the input or output texts, since it is not my objective with this work, so I am not generating those. Secondly, due to amount of different measurements I had to perform to various instructions, it was necessary to create subfolders where I could store my measurements so I could tell to which test those power traces belong. Before prompt about amount of measurements needed, in my version program asks to enter a name of a test. After this it creates folder with the current date (unless this folder already exists) and inside this folder created a subfolder with with a test name. Lastly, I the only thing this program sends to a smart card is header of a APDU with command that starts the test. No data from this program are processed in my tests, so there is no need to send anything else.

Another piece of hardware I used is measurement adapter for smartcards created by Ing. Jiří Buček.

2.1.2 Post-processing

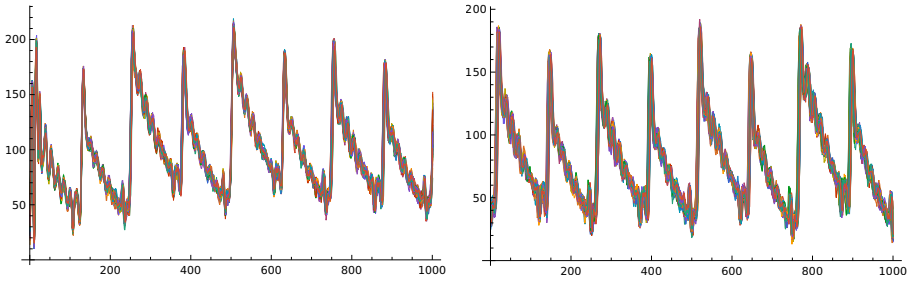
2.1.2.1 Timing problem

Sample rate of oscilloscope was set to 1GSa (1 sample every nanosecond), and with 4MHz frequency of microcontroller's internal clock it gives 250 samples per clock.

First problem that I had to deal with is electronic noise, which is solved easily by collecting a reasonably large amount of traces and calculating mean.

And right there is the actual obstacle: clock period has some deviation, not very large, but still significant enough to interfere. This results in a situation, when one clock cycle might be actually by a mere nanosecond longer or shorter than documented. At the point where trigger signal was sent it might not be such a problem, but some milliseconds after this event error accumulates. Figure 2.1 depicts how this situation looks like in reality. Notice how much less precise clock match is on the figure 2.1b than that of 2.1a.

When calculating mean of those raw traces, result is blurred due to the fact that clocks were not properly aligned, which makes analysis results less precise. Furthermore, power traces are not only misaligned between each other, but clocks inside are as well, and alignment of pieces of traces within one trace is even worse.



(a) Right after trigger signal is sent.

(b) After 3ms from sending trigger signal.

Figure 2.1: One hundred of raw power traces of 1 μs.

2.1.2.2 Solution

So two problems arise: noise and alignment. The way to solve those problems is to find at which point power consumption grows the most in every clock cycle (due to change in a clock signal from 0 to 1) and align those points. Afterwards it is possible to calculate mean value of power traces without loosing any important pieces of power consumption.

I designed a simple utility for that purpose. This program loads single power trace, aligns it and accumulates in array, that represents sum of all aligned power traces, those steps repeated until end of traces file is reached (i.e. all power traces are read).

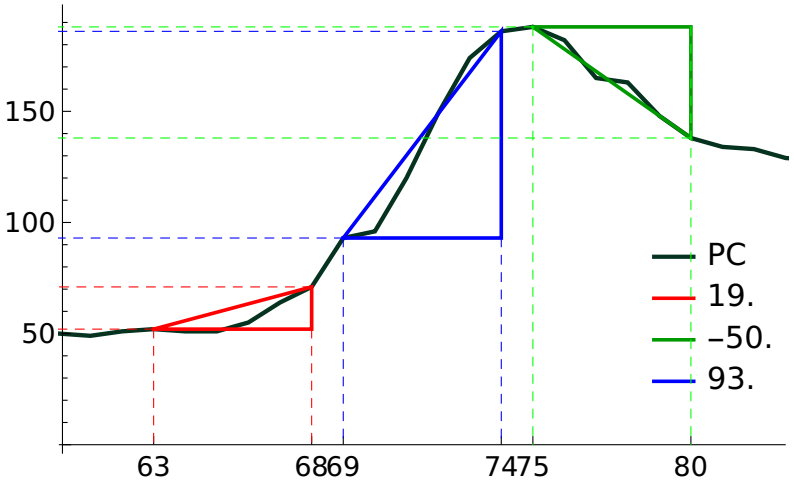


Figure 2.2: Analysis of a three sample pairs in a window for $n = 5$ with position to analysis 63, 69, 75. Legend shows difference value by which the starting point is chosen.

Here I would explain aligning a little further. Aligning process starts after identifying beginning of a relevant part (i.e. where trigger signal is sent). As an initial step first set of samples in a size corresponding to one clock cycle is copied to accumulator array. From this point aligning uses concept of "sliding window". Program selects set of samples, that contain the change of a clock signal from zero to one; in other words, position, where execution of instruction starts. To find exact place of start, set of samples is searched for the edge with largest angle. Program takes first sample in the set and nth sample after it, where n is fixed value for every iteration for the sake of consistency. For the needs of this program the "angle" in question can be sketched as difference between values of second and first samples. Then programs picks second sample and nth sample after it, and defines that difference for them, and continues until it reaches the end of a sample set. After every such sample pair in set is calculated, program picks the one with the largest positive difference and sets position of it's first sample as the beginning of a clock.

Figure 2.2 tries to explain this concept. Here pair of samples with positions $69, 74$ clearly has biggest positive difference between values, meaning that between those two points difference in power consumption is the biggest, which makes position 69 a candidate for a "clock starter". Code 1 present implementation of this algorithm, which I used to post-process[4] power traces.

2.2 Test design

2.2.1 Design properties

When instruction is executed, a lot of variables can contribute to it's power consumption, such as it's opcode, position in memory, operand values, data it operates on, opcode of a next instruction and power consumption of previously executed instruction, and also hardware-specific variables such as electronic noise. According to [12] we can compose those variables into four groups. With that said, we can roughly estimate that power consumption of any specific point in time can be described with a following formula

$$P_{total} = P_{data} + P_{op} + P_{el.noise} + P_{const}.$$

Means that total power consumption at any given point in time has data dependent component, operation dependent component, and also electronic noise and constant component. From the cryptanalytic point of view, most important components are P_{op} and P_{data}, as well as $P_{el.noise}$. P_{const} doesn't provide any exploitable information, so it's insignificant. On the other hand, $P_{el.noise}$ doesn't provide any information either, but it's presence can obscure power consumption and make power analysis way harder.

[4] Or pre-process, depending on point of view

Listing 1 C++ implementation of aligning algorithm.

```
#define CLOCK 250
void MeanCalculator::ClockAllignment ()
{
  for ( unsigned i = 0 ; i < CLOCK ; i++)
    mean[i] += trace [start + i];

        unsigned clock = CLOCK;
        unsigned first_40p_clock = (CLOCK * 4 ) / 10;
        unsigned clock_skip      = (CLOCK * 9 ) / 10;
        unsigned clock_skew      = CLOCK / 50;
  unsigned max_pos = 0;
  double   max = 0;

  for ( unsigned i = start + clock_skip          ;
        i < trace_length && clock < mean_length  ;
        i = max_pos + clock_skip                 )
  {
    for ( unsigned j = i; j < i + first_40p_clock ; j++)
    {
      double dif = (trace[j+clock_skew] - trace[j]);
      if ( dif > max )
      {
        max = dif;
        max_pos = j;
       }
    }
    for ( unsigned j = 0                              ;
         j < CLOCK && max_pos + j < trace_length  ;
         j++                                      )
      mean [clock + j] += trace[max_pos + j];
    max = 0;
    clock += CLOCK;
  }
}
```

But for the sake of my tests, I keep up to more detailed way of breaking up power consumption components. For example, when considering P_{data} component of a power consumption, we can compose it from a list of variables. For example, for `LD` instruction, which loads memory entry value from specified address and to specified register, P_{data} can be broken down to $P_{reg.value}$ (contents of a register), $P_{mem.value}$ (value that is loaded) and $P_{src.addr.}$ (source address). So it is good idea to either isolate those components (for example, loading from different addresses same value), or to keep track of everything that can affect power consumption.

My tests use both approaches, because in some cases using only isolated data analysis means omitting important components such as Hamming distance between destination and source data, and on the flip side, manipulate with a large amount of possible dependencies can be convoluted and make power analysis much harder.

2.2.2 Test generation and communication with measurement setup

To apply those tests I needed to have a way to communicate with computer in order to obtain power traces. I used program [11], modified by Filip Štěpánek. Practically this modification is rather a downgrade, since I cut off everything, that was not necessary for the purpose of my work.

Besides actual `.hex` file, that gets programmed to a microcontroller, this tool outputs very important debug data, that are useful in further analysis and test design, such as on what exact addresses instruction will be located, what opcode it has and so on.

Other than than I wrote a number of bash scripts that helped me managing test generation and logging of important files.

2.3 Software

As I mentioned in the introduction to this chapter, the software I chose is Wolfram Mathematica. I am familiar with this software from past courses BI-CAO and BI-PMA so it was a logical choice for me. Besides, Wolfram Mathematica provides wide range of analytic possibilities, not only implemented in a language itself but with a use of custom functions and libraries that user can program himself. It also has great graphic drawing functions. In my analysis I used is `Manipulate` that allows user to change the variable in specified function and see how this change affects it on the go. It was helpful in visual analysis of power traces. Mathematica can also export graphics into various formats and I used this functionality to provide graphs for this work.

I implemented a small library to shortcut frequently used sets of functions, such as reading traces from a file and creating graphics.

Chapter 3

Power consumption analysis of a microcontroller ATMega163

This chapter presents the main output of this work. Here reader will find power consumption analysis of a microcontroller ATMega163. Following sections will describe dependencies on a different factors and variables in executed instruction, at which point of execution those dependencies occur and the thought process that has led me to those conclusions.

Before anything is necessary to outline the internal architecture and execution cycle of microcontroller ATMega163. After clarifying those points we can proceed to the analysis.

First things that need to be resolved are values that are not necessarily dependent on instruction type: address of a currently executed instruction and it's operand value. Then we will explore power consumption of every instruction type thoroughly through different instructions.

3.1 Architecture and instruction execution of ATMega163

Not everything about ATMega163 microcontroller is available in it's datasheet, such as the actual topography of a chip. Despite that, it contains a lot of useful data on instruction cycle, from which user can conclude what to look for in power traces analysis. This section largely depends on information provided by datasheet, so assume that everything descriptive about ATMega163 comes from it [1], unless I explicitly cite something else, or state my own observations.

ATMega163 has a Harvard architecture, meaning that it has separated program memory and data memory, which implies that it can access both simultaneously. This property enables instruction level parallelism in a form of two-stage pipelining with "fetch" and "execute" stages.

Figure 3.1 (page 16 of datasheet) illustrates simply how it works. At first clock microcontroller fetches the instruction and at second microcontroller executes it. Obviously, the "fetch" loads instruction from program memory, meaning that it holds a memory position at Program Counter. This observation is important: it suggests that at the instruction execution stage power consumption will be somehow dependent on address and opcode (and operands values that come with it) of instruction that is currently fetching. I explore it more in subsection 3.4.2, where I analyze address dependency. In subsection 3.4.4 I will examine it even further with analysis of branch and jump instructions.

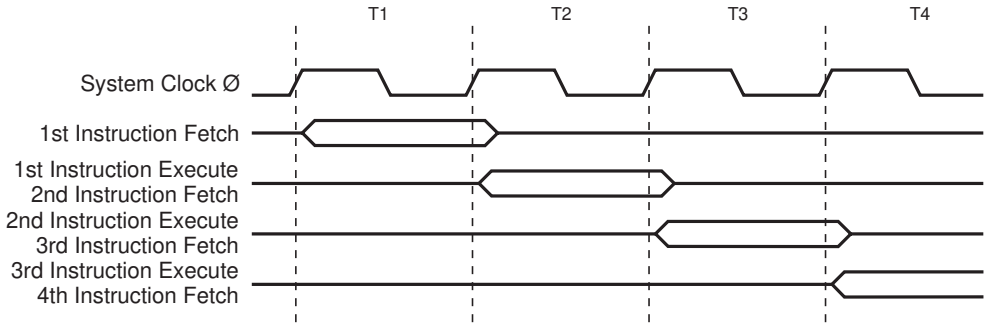


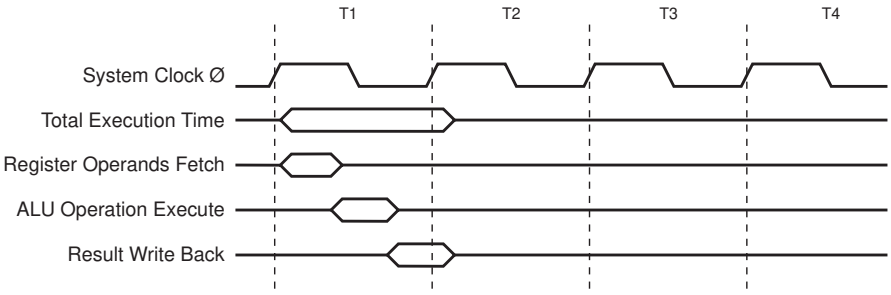
Figure 3.1: The Parallel Instruction Fetches and Instruction Executions.

Next lets examine timings of actual instruction execution. Basically, excluding manipulations with PC and program memory I looked at higher in the text, vast majority of instructions involve either arithmetic logic unit (ALU for short) or data memory (from this point I will refer to it as SRAM as in datasheet).

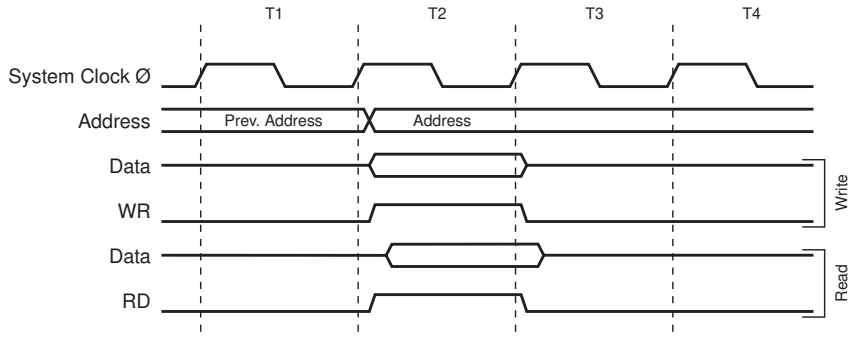
Quite expectedly for ALU operations, as seen at figure 3.2a (page 17 of datasheet), execution goes step-by-step: collect data to process, compute something with those data and save the result. Generally I would expect some similar dependency occurring in power traces, more on that in subsection 3.4.1. Majority of ALU operations takes just one clock cycle, except for those involving two registers, which are interpreted as word and which are processed in two clock cycles. This makes me suppose that in two-clock instructions I will see something similar to two corresponding one-clock ALU operations in a row.

Everything that accesses SRAM takes two clocks. No further information to figure 3.2b (page 17 of datasheet) is given, and I can only suppose that at the first clock of memory access instruction execution SRAM controller decodes address to requested memory entry. This theory will be tested at subsection 3.4.3.

ATMega163 has 8K $\times$ 16 Program Memory, 1K $\times$ 8 Data Memory and 32 general purpose registers and 64 I/O registers. SRAM is organized in a way,



(a) Single Cycle ALU Operation.



(b) On-chip Data SRAM Access Cycles.

Figure 3.2: Operations during execution stage of ATMega163.

that program can access general purpose registers and I/O registers same as ordinary memory entries. It means, that maximum address that is available is `$45f`. What is notable, that instructions that directly or indirectly address Program Memory of SRAM, have more than needed amount of bits to address it. For example, instruction `LDS` (Load Direct) loads byte from 16 bit position, which is more than needed to address total of `$45f` memory positions. While testing instructions that access memory locations, I found out that loading data from address that is bigger than `$45f` throws away higher nibble of high byte entirely and cuts two most significant bits from lower nibble. This observation will be useful at testing instructions, that manipulate with addresses, for example Data Transfer Instructions (see subsection 3.4.3).

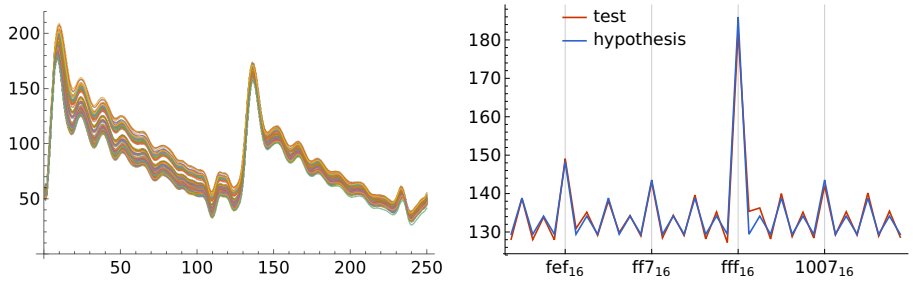
3.2 Instruction address dependency

To test power consumption of instructions I needed to test instruction dependency first. Since Program Counter change is always there, it might be a significant complication in analysis, if instruction is placed in a wrong place. There is not always an option to place tested instructions at the exact same address: while the problem of testing data dependency in registers is eas-

ily solved by "increment–branch" pair, testing operand change would require absurd amount of measurements.

To test instruction address dependency, I used `NOP` instruction, which doesn't operate on any data, and has all-zeros opcode. I found out, that power consumption is heavily dependent on address of instruction that is currently in execution and next address. It can be explained by the fact, that ATMega163 has two-stage pipeline, and at the moment of execution of current instruction next one is in fetch stage. Apparently, increment and logically following change in a number of bits creates larger power consumption.

In result, power consumption at 14 ns of instruction execution is dependent on a change between current and next instruction. Correlation between power consumption at this point and Hamming distance of those two addresses is 0.988.



(a) Overview of every clock.

(b) Closer look at address dependency on addresses from `$feb` to `$100e` at 14 ns.

Figure 3.3: `NOP` executed at 6000 different addresses, starting from `$2b5`.

On figure 3.3b it is clear that at address `$fff` power consumption is the largest. And indeed, Hamming distance between `$fff` and next address, `$1000`, is 12, while, for example, Hamming distance between `$ff8` and `$ff9` is only 1.

On the same figure there is noticeable difference between hypothesis and real power consumption. At address `$1000` it is significantly higher, than expected. This happening is due to residual consumption after a large peak.

I had to choose how to place instructions in such a way, that it has highest possible density (but without interfering with helper instructions) and to minimize address change impact.

Table 3.1: Lower three bits of address with it's power consumption dependency

Address	000	001	010	011	100	101	110	111
Next address	001	010	011	100	101	110	111	X000
Hamming distance	1	2	1	3	1	2	1	>3

Table 3.1 shows how Hamming distance changes in dependency on last three bits. With what was mentioned before, I needed to choose some address, that doesn't have large Hamming distance with next instruction (so `$7` and `$f` are not an option), and on any address that goes after those (so `$0` and `$8` are also excluded). I chose `$2` and `$a`: Hamming distance with next address is minimal, and address before it has second minimal Hamming distance with next address, address after has third minimal Hamming distance.

3.3 Operand value dependency

There are two types of operand values.

1. Immediate values
    - value to process (`LDI`, `ORI` and `ANDI`),
    - address in SRAM (`LDS`, `STS`, displacements),
    - address (direct or relative) in Program Memory (jumps, branches and calls),
    - address of ports (`IN`, `OUT`),
    - positions in register (`BSET` and `BCLR`).
2. Register numbers
    - source and destination registers in most of instructions,
    - registers for indirect addressing memory (Program or SRAM) (`LD`, `ST`).

Usually in instructions there is wider range of immediate values available then register numbers. Even with relatively big number of general purpose registers AVR microcontrollers dispose (in ATMega163 it's 32), a lot of instructions narrow user's options down to 16 or even 6 registers.

Logically, it is expectable to see dependency on immediate values, since those are the ones that get finally processed. And as opposed to immediate values, little to no dependency on register number is expected.

3.3.0.0.1 Register number dependency In reality register number can noticeably affect data dependency in computations. I examined `ADD` instruction on values from 0 to 31 in four tests. It is combination of two sets of options. First set is where does data change occur: in source or destination register. Second set is at which position occurs change in register number, again, either source or register.

What I found out that data in destination register expose themselves through power consumption less than data in source register. On top of that,

what data are there in destination register is much harder to define: not only it alters power consumption, but generally makes it more noisy, see figure 3.20. Correlation between Hamming weight of data and power consumption doesn't get higher than 0.608 and at a different point in time through every register number. All in all, there is some dependency, but it is not very clear one.

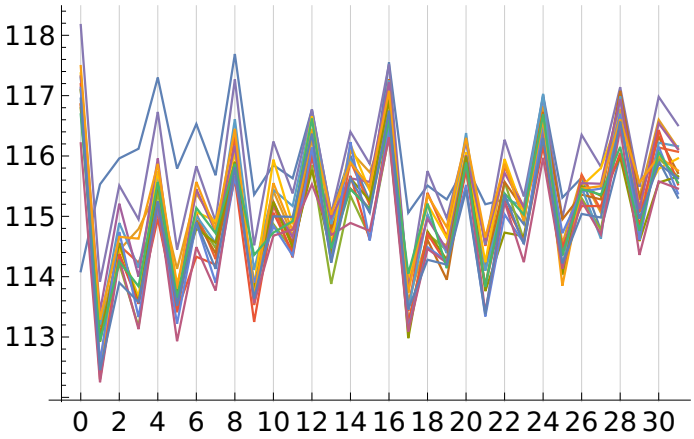
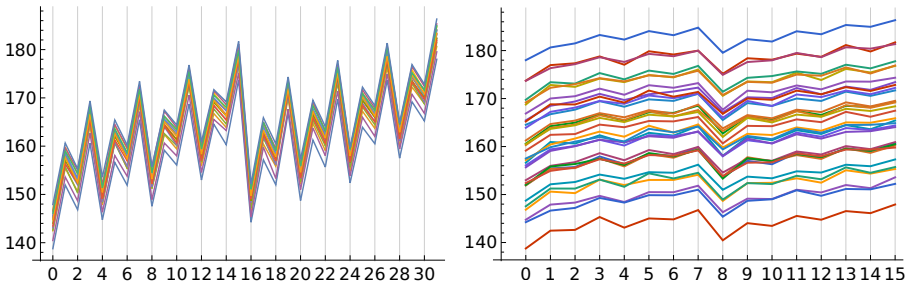


Figure 3.4: Power traces at 18 ns for constant number of destination register and different data stored in it.

On the other hand, when data in destination register remain constant, data dependency is clear with correlation up to 0.990, as shown on 3.5a. More on data dependency reader will find in further sections that provide instruction analysis. Another noticeable thing is that register number also leaks in a form of Hamming weight, adding up to total consumption, see 3.5b.



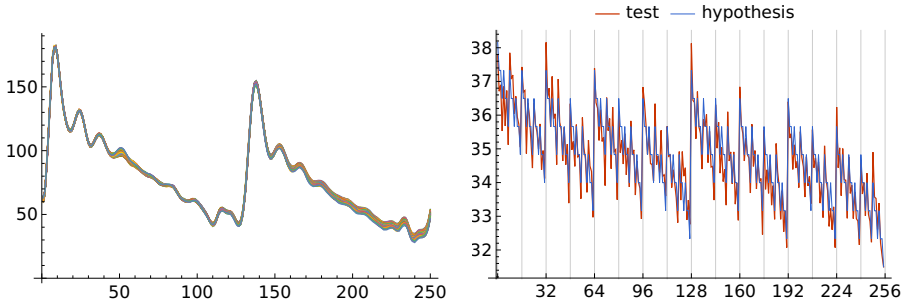
(a) Data dependency of power consumption through all the register numbers.

(b) Register number dependency of power consumption through data.

Figure 3.5: Power traces of `ADD`, where destination register is constant in data and it's number and changing number of source register and their data.

3.3.0.0.2 Immediate value dependency More interesting then data dependency during execution stage of an instruction is opcode change. I described in subsection 3.4.2 how much PC change affects power consumption. Here I am interested in examining how opcode can affect fetch of an instruction.

To test it I picked two instructions that use 8 bit immediate value: `LDI` and `ORI`. Dependency was found, which starts after the falling edge of a clock (as seen on 3.6a) and is defined by Hamming weight of negated value. Figure shows this dependency with hypothesis. Notice how opcode's Hamming Weight dependency occurs at the very end of instruction fetch.



(a) One clock before execution of `ORI`.

(b) Dependency on immediate value at 243 ns of `ORI` fetch.

Figure 3.6: Fetch of `ORI` with different immediate values.

In conclusion I have to say that even though some dependency on opcode is present, it is not as significant as data dependency, that will be examined later in the work.

3.4 Instruction type dependency: instruction set analysis

3.4.1 Arithmetic and logic instructions

Tests of arithmetic and logic instructions for any intelligible results need to operate with data. Every such instruction manipulates with general purpose registers.

As was said before, we would expect some dependency on data stored in registers at the beginning of execution, then something that would resemble computation happening in between and at the end storing the data to destination register.

Some instructions are not described, because they are redundant and are practically aliases to existing instructions.

3.4.1.1 Arithmetic instructions

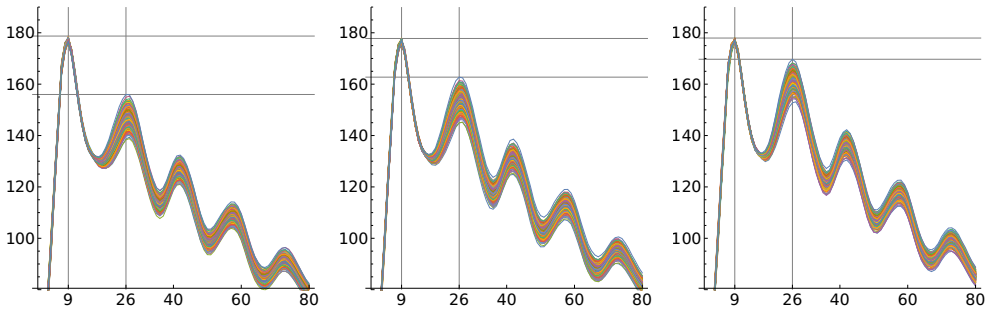
3.4.1.1.1 General positioning of dependencies examined on `ADD` and `ADC`

Table 3.2: Detailed description of `ADD` and `ADC` instructions.

Description	Add two Registers	Add with Carry two Registers
Mnemonics	`ADD`	`ADC`
Operands	`Rd, Rr`	`Rd, Rr`
Operation	`Rd ← Rd + Rr`	`Rd ← Rd + Rr + C`
Flags	Z,C,N,V,H	Z,C,N,V,H
Opcode	`0000 11rd dddd rrrr`	`0001 11rd dddd rrrr`
Clocks	1	1

`ADD` starts it's register operands fetch roughly at 18 ns after start of a clock and is mostly visible around 24 ns. Testing with constant data in destination register and ranging data from 0 to 255 in source register, picking a power trace measurement at 24 ns shows dependency very similar to Hamming weight of a ranging value, that currently is loaded to ALU. Correlation between Hamming weight of this value and power consumption of processing those values is roughly 0.983 (mean between correlations when destination register is set to `$00`, `$0f` and `$ff`). Same holds for situation when we switch destination and source.

Figure 3.7 shows, that with different data stored in source register and three different values stored in destination register, data consumption changes significantly. It can be seen at the smaller wave peak at 27 ns that with different values that peak gets higher: with destination register set to `$00` the largest peak is 155.87 in contrast to destination register set o `$ff`, where it reaches 168.63. Notice, how main (first) large peak at around 178 ns is not changing no matter what is stored in both registers.



(a) `r16 = $00`. (b) `r16 = $0f`. (c) `r16 = $ff`.

Figure 3.7: Power traces of `ADD r16, r17`, where `r17` is ranging from 0 to 255 and with different values stored in `r16`

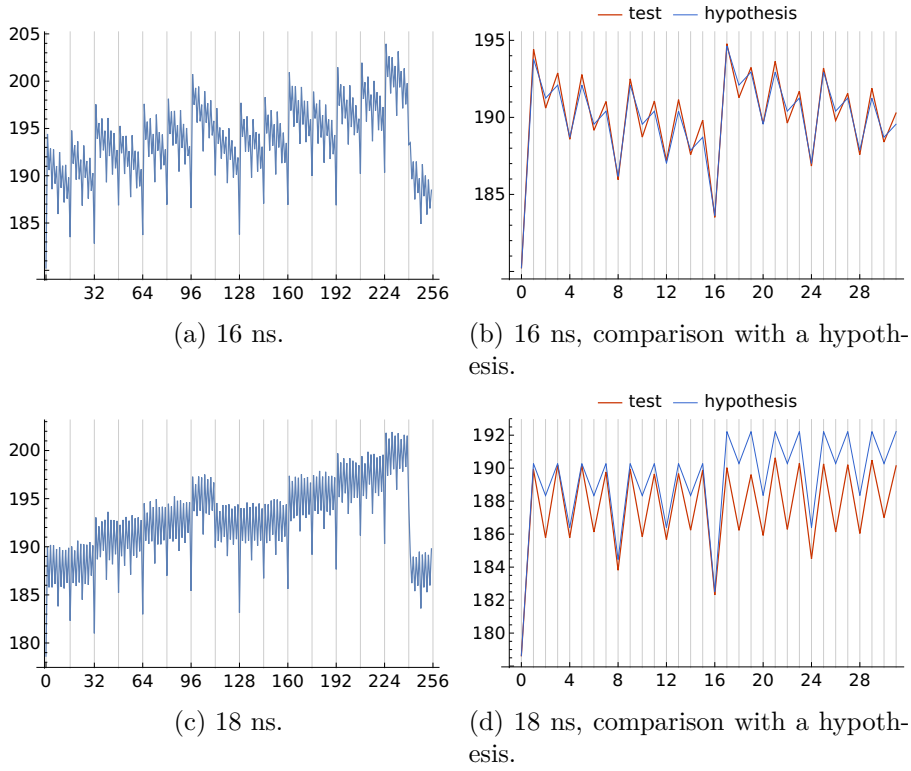
ALU operation execute and result write back occur around falling edge of a clock. Therefore power consumption there is more steady with residual peaks of consumption. This effect is more pronounced after processing large values, yet still too small to distinguish it from electronic noise.

What is remarkable is that the result is leaking the clock after the `ADD` execution, which I found a rather common thing practically in every instruction. The progression of changes in this "after-clock" is much more interesting than in the main clock. It starts off with a dependency Hamming weight of data and "exclusive or" of data and the result at 11th ns, reaches it's peak at 16 ns with ratio 1 (data) to 3 (data "XOR" result) with correlation 0.987. After this it moves towards 1 to 1 ratio at 18 ns and proceeds with more dependency on result.

Figure 3.8 shows an example with destination register initially set to `$0f`. Figures on the left present power measurement at described points in time through all values in source register. Figures on the right show a comparison between those power traces and hypothesis. Reader can notice, how neatly hypothesis and result are correlating.

`ADC` acts exactly as one would expect: same as `ADD`. No matter if carry flag is set or not: correlation between Hamming weight of a processed value and power consumption at 22 ns is around 0.98. If carry flag is cleared then the rest of execution goes pretty much the same way. If carry flag is set, then consumption seems to be dependent on Hamming weight of current data and difference between data and incremented value with ratio 3 to 2 with correlation 0.888. The clock after `ADC` show expected dependencies to what I've seen in `ADD` instruction.

Another test I've done with `ADD` is flag test. I've set flags to a different values and performed same `ADD` operation with the same values. My idea was that if there is five bits that instruction can change, then it would be somehow noticeable. Unfortunately, this returned no significant differences in power consumption. In my opinion the reason for that is that bits that operation



(a) 16 ns.

(b) 16 ns, comparison with a hypothesis.

(c) 18 ns.

(d) 18 ns, comparison with a hypothesis.

Figure 3.8: Power traces of clock after `ADD r16, r17`, where `r16` contains `$0f` and `r17` is ranging from 0 to 255 at different points of execution.

sets arrive to `SREG` in slightly different time, and change in just one bit doesn't create much of a power consumption. Or may be that dependency is so small that it is insignificant in the scope of power analysis attacks anyway.

3.4.1.1.2 `SUB` and `SBC`

Table 3.3: Detailed description of `SUB` and `SBC` instructions.

Description	Subtract two Registers	Subtract with Carry two Reg-s
Mnemonics	`SUB`	`SBC`
Operands	`Rd, Rr`	`Rd, Rr`
Operation	`Rd` ← `Rd - Rr`	`Rd` ← `Rd - Rr - C`
Flags	Z,C,N,V,H	Z,C,N,V,H
Opcode	`0001 10rd dddd rrrr`	`0000 10rd dddd rrrr`
Clocks	1	1

"Subtraction" is a short way to say "Addition of a negative value". And this is exactly what I saw in my tests.

At 23 ns of instruction execution power traces show dependency on Hamming weight of data currently processed. At the next clock at 16 ns starts to show dependency on result. My hypothesis that it depends on a data before processing, negated source data, difference between result and data in destination register and on a result itself with ratio 2/1/5/1 respectively shows correlation 0.984. Figure 3.9 shows power consumption with this hypothesis.

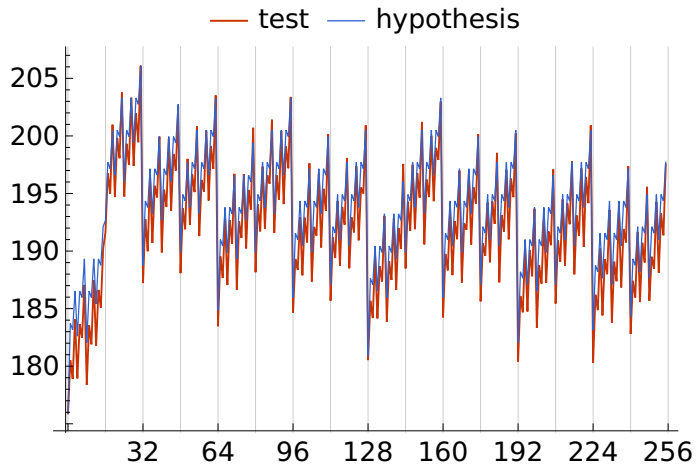
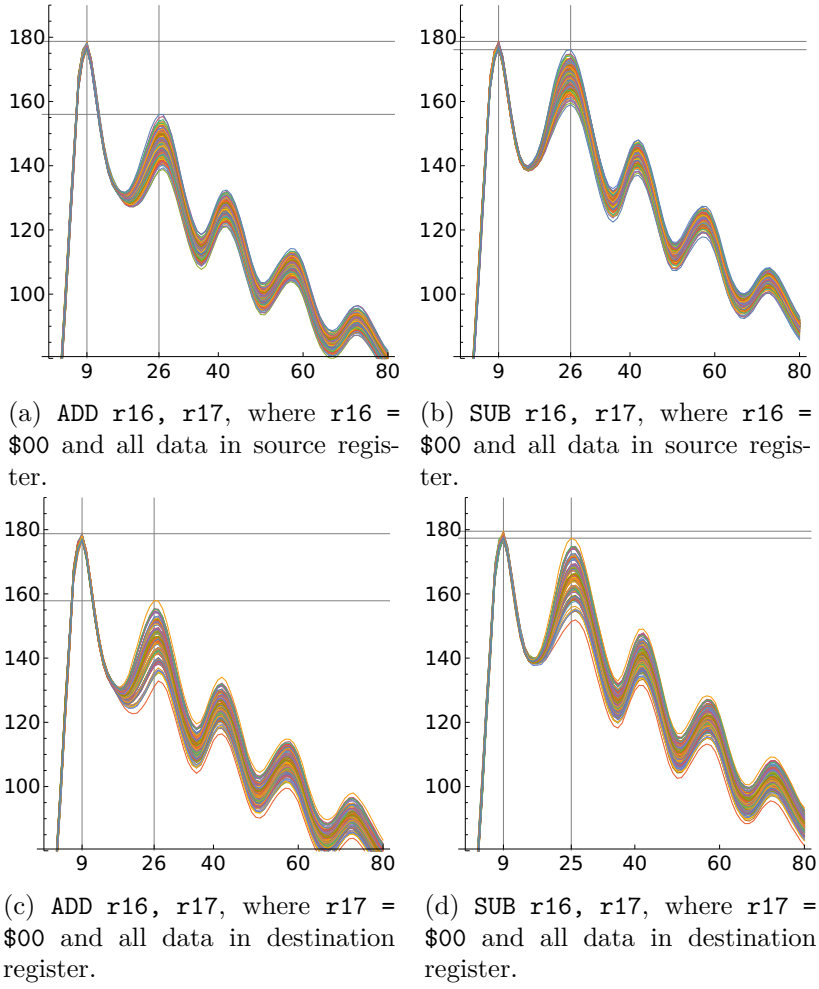


Figure 3.9: Power consumption in the clock after `SUB` execution at 16 ns, with hypothesis.

Speaking in general, `SUB` produces more power consumption than `ADD` does due to negation operation that needs to be precomputed. In figure 3.10 we can notice, that at 26 ns power consumption of `SUB` is noticeably larger that of `ADD`. The second peak of power consumption at maximum when at one of the registers zero is stored, for `SUB` reaches 179.51, while for `ADD` doesn't reach even 160.

Instruction `SBC` is no exception from the rule I talked about examining `ADC`. Instructions, that just add carry to the computation leak data and result of computation pretty much at the exact places as their regular counterparts.



(a) `ADD r16, r17`, where `r16 = $00` and all data in source register.

(b) `SUB r16, r17`, where `r16 = $00` and all data in source register.

(c) `ADD r16, r17`, where `r17 = $00` and all data in destination register.

(d) `SUB r16, r17`, where `r17 = $00` and all data in destination register.

Figure 3.10: Comparison between power consumption of `ADD r16, r17` and `SUB` with different data being processed.

3.4.1.1.3 `SUBI` and `SBCI`

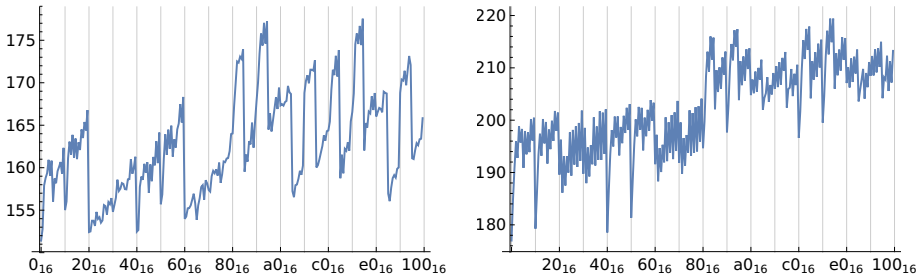
Table 3.4: Detailed description of `SUBI` and `SBCI` instructions.

Description	Sub. Cons. from Reg.	Sub. with Carry Const. from Reg.
Mnemonics	`SUBI`	`SBCI`
Operands	`Rd, K`	`Rd, K`
Operation	`Rd ← Rd - K`	`Rd ← Rd - K - C`
Flags	Z,C,N,V,H	Z,C,N,V,H
Opcode	`0101 KKKK dddd KKKK`	`0100 KKKK dddd KKKK`
Clocks	1	1

For analyzing instructions, that operate over both register and immediate value, I did two tests. First test changes data stored in register and performs instruction with the same immediate value. Second test keeps data in destination register constant and changes the immediate value. For the first test of `SUBI` I chose data to start at value `$00` and always subtract one. That way I can test both data in register, result and control the testing loop. In the second test I stored value `$ff` in destination register, and after every subtraction restored it.

At first I will describe dependency on value stored in register. At around 9 ns `SUBI` starts to fetch data, clear enough dependency on Hamming weight of data occurs at the 24 ns (correlation with hypothesis 0.755). The clock after execution show dependency on a result and Hamming distance between result and data stored previously.

What is more interesting is dependency on immediate data. I expected something among the lines of classic `SUB`, but I was mistaken. Apparently, at the operand fetch ALU does load immediate value. And in my opinion, power consumption here consists of more parts than `SUB`. Execution clock of `SUBI` with constant value stored in destination register in both tests and different immediate value shows pattern, that is quite common for instructions that operate over immediate values, see figure 3.11a.



(a) Execution clock. (b) After execution clock.

Figure 3.11: Power consumption of `SUBI` at 21 ns with constant value in destination register and different immediate values.

But the clock after execution of `SUBI` starts with something that resembles Hamming Weight of immediate value. I can not be sure about that, since I was not able to guess closer than 0.5 correlation between data and my hypothesis. Moreover, at 21 ns there is another (but similar) power consumption throughout all immediate values, see figure 3.11b.

"Carry" version `SBCI` doesn't help either. Both instructions clearly have structure to their power consumption in time throughout the immediate values, and for both I wasn't able to find that dependency rule.

Power consumptions with and without carry flag set and `SUBI` look similar.

3.4.1.1.4 ADIW and SBIW

Table 3.5: Detailed description of ADIW and SBIW instructions.

Description	Add Immediate to Word	Subtract Immediate from Word
Mnemonics	ADIW	SBIW
Operands	Rd, K	Rd, K
Operation	Rdh:Rdl ← Rdh:Rdl + K	Rdh:Rdl ← Rdh:Rdl - K
Flags	Z,C,N,V,S	Z,C,N,V,S
Opcode	1001 0110 KKdd KKKK	1001 0111 KKdd KKKK
Clocks	2	2

First I examined instruction ADIW. What stroke me as unusual is that unlike other instructions, that deal with immediate values, ADIW actually act as expected. First execution clock fetches immediate data. What is notable that apparently the higher nibble of a lower byte is more significant in the scope of power consumption, because last three quarters of a tested data (ones that have some bit of immediate set in a higher nibble) show significantly larger power consumption. At the second clock of execution dependency on the value stored in lower register shows, more specifically, on difference between data stored in destination register and immediate value.

Dependency on data in registers is located as predicted, dependency on hamming weight of lower byte is located at the first clock of execution, dependency on Hamming weight of a higher byte and Hamming distance between result of a first operation and data that were stored in lower register prior to addition. One clock after execution, logically, shows dependency on hamming weight between result of addition of carry flag to high byte and what was stored before.

Analysis of SBIW showed similar dependencies.

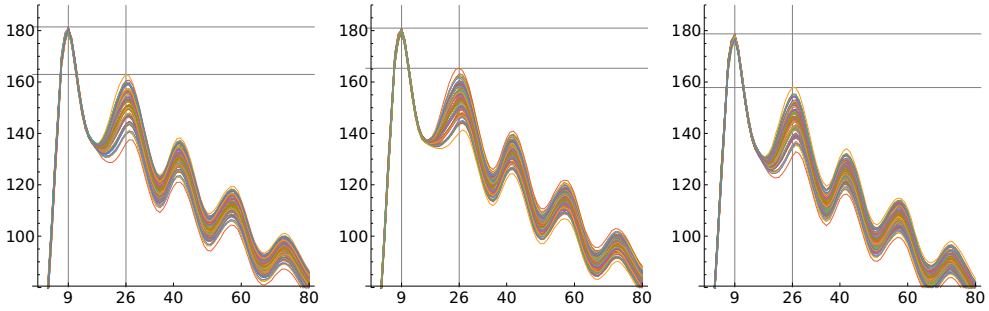
3.4.1.1.5 INC and DEC

Table 3.6: Detailed description of INC and DEC instructions.

Description	Increment	Decrement
Mnemonics	INC	DEC
Operands	Rd	Rd
Operation	Rd ← Rd + 1	Rd ← Rd - 1
Flags	Z,N,V	Z,N,V
Opcode	1001 010d dddd 0011	1001 010d dddd 1010
Clocks	1	1

Both instructions at execution leak Hamming weight of data stored in register. Again, starting at 18 ns and continuing through rest of execution with most detectable dependency at 24 ns.

Clock after execution both instructions start off with some Hamming distance between data previously stored in register and result at 10 ns, then at around 23 ns power consumption is more dependent on a Hamming weight of result itself, but after that it continues with dependency on processed data. Correlation between my assumptions and real power consumption is around 0.75 to 0.85. This holds for both `INC` and `DEC`.



(a) `INC r16`. (b) `DEC r16`. (c) `ADD r16, r17` with `r16` set to different values and `r17` constant.

Figure 3.12: Comparison between `ADD` and `INC`

When compared with `ADD`, those instructions practically do not differ from it, as shown at figure 3.12.

3.4.1.1.6 `MUL` and derivatives ATMega163, together with lots of Microchip's microcontrollers, has hardware multiplier, that process data just in two clocks.

Both versions of `MUL` fetch data from register at the first clock, as seen by dependency on Hamming weight of contents of registers, when one of the register is set to zero. What is interesting, that at the second clock of execution, dependency on Hamming weight of a multiplicand register (`RD`) is much more significant, then that of multiplier (`Rr`). Which is even stated in Atmel's Application note on hardware multipliers [2].

Multiplication itself starts right it the first clock. Immediately it can be seen, that calculation is started, when at 17th ns power consumption starts to be dependent on a lower byte of a result. And at 25 ns dependency on a Hamming distance between result and contents of a `R0` starts to be prevalent and keeps until the end of a first clock.

Table 3.7: Detailed description of `MUL` and `FMUL`, `MULS` and `FMULS`, `MULSU` and `FMULSU` instructions.

Description	Multiply Unsigned	Fractional Multiply Unsigned
Mnemonics	`MUL`	`FMUL`
Operands	`Rd, Rr`	`Rd, Rr`
Operation	`R1:0` ← `Rd` × `Rr`	`R1:0` ← `Rd` × `Rr`
Flags	Z,C	Z,C
Opcode	`1001 11rd dddd rrrr`	`0000 0011 0ddd 1rrr`
Clocks	2	2
Description	Multiply Signed	Fractional Multiply Signed
Mnemonics	`MULS`	`FMULS`
Operands	`Rd, Rr`	`Rd, Rr`
Operation	`R1:0` ← `Rd` × `Rr`	`R1:0` ← `Rd` × `Rr`
Flags	Z,C	Z,C
Opcode	`0000 0010 dddd rrrr`	`0000 0011 1ddd 0rrr`
Clocks	2	2
Description	Multiply Sig. and Unsig.	Fractional Multiply Sig. and Unsig.
Mnemonics	`MULSU`	`FMULSU`
Operands	`Rd, Rr`	`Rd, Rr`
Operation	`R1:0` ← `Rd` × `Rr`	`R1:0` ← `Rd` × `Rr`
Flags	Z,C	Z,C
Opcode	`0000 0011 0ddd 0rrr`	`0000 0011 1ddd 1rrr`
Clocks	2	2

Next clock shows a familiar picture, more precisely Hamming distance between a value and it's two's complement, which is the case of the multiplicand and the lower result byte. I am not sure about what exactly causes this behavior.

Instructions that operate with signed values act in the similar fashion.

3.4.1.2 Logic instructions

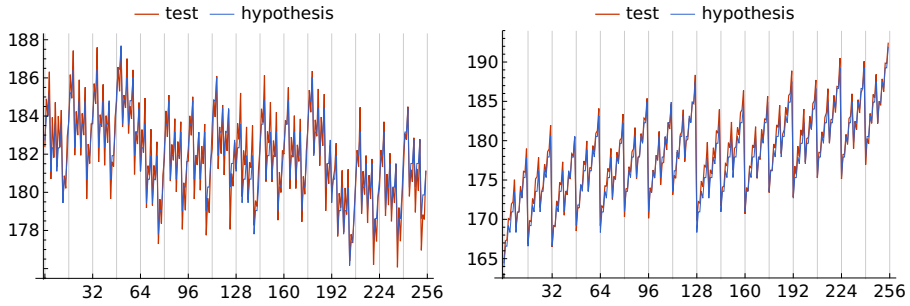
3.4.1.2.1 `AND` and `ANDI`

Table 3.8: Detailed description of `AND` and `ANDI` instructions.

Description	Logical AND Registers	Logical AND Register and Constant
Mnemonics	`AND`	`ANDI`
Operands	`Rd, Rr`	`Rd, K`
Operation	`Rd` ← `Rd` ∧ `Rr`	`Rd` ← `Rd` ∧ `K`
Flags	Z,N,V	Z,N,V
Opcode	`0010 00rd dddd rrrr`	`0111 KKKK dddd KKKK`
Clocks	1	1

First I want to examine "regular" `AND`. As with all ALU instructions, data fetch exposes itself through Hamming weight of data stored in registers in it's execution clock.

The next clock starts with dependency on Hamming distance between final result and data processed with correlation up to 0.93. Figure 3.13 shows comparison between my hypotheses and actual power consumption at two points: 16 ns and 24 ns. Hypothesis for power consumption is sum of processed data Hamming weight, result of `AND` operation and Hamming distance between those. At 16 ns with ratio 3/1/8 respectively it has correlation 0.927. For 24 ns with ratio 3/3/1 correlation is 0.992.



(a) Power consumption at 16 ns (b) Power consumption at 24 ns

Figure 3.13: Power consumption of clock after `AND r16, K`, with `r16 = $cc` and all possible `K` at different points of execution with respective hypotheses.

Logic instructions such as `AND`, `OR`, `EOR` are especially good in examining dependency on a result. With those instructions it is easy to see with a naked eye what is going on at particular points of execution.

Lets proceed to examining `ANDI` instruction. Instructions with immediate values act in a way that is not always plain as other data dependencies. Power consumption at execution of a `ANDI` instruction seem to be dependent on

immediate value it processes, which is expectable. But it seems like it is broken down into pieces of length 16. Figure 3.14 shows what I mean by that.

It looks like Hamming weight of values from 0 to 15 repeated 16 times without much significant difference. This pattern continues through entire execution clock. My theory is, that ALU fetches immediate value nibble by nibble. Which is a valid theory, given that immediate value is stored by nibbles in opcode: higher nibble is stored in higher byte of `ANDI`'s opcode and lower nibble in lower byte, see table 3.8.

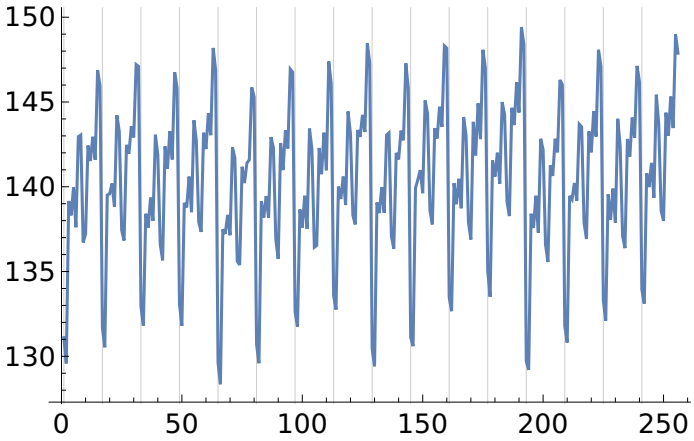
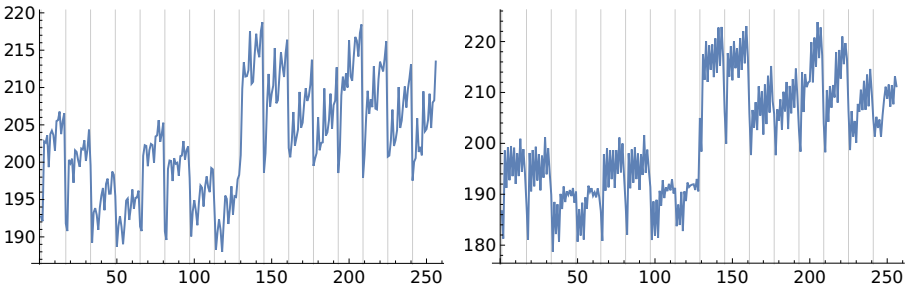


Figure 3.14: Power consumption of `ANDI` with different immediate values and constant value stored at destination register at the clock of execution at 16 ns.

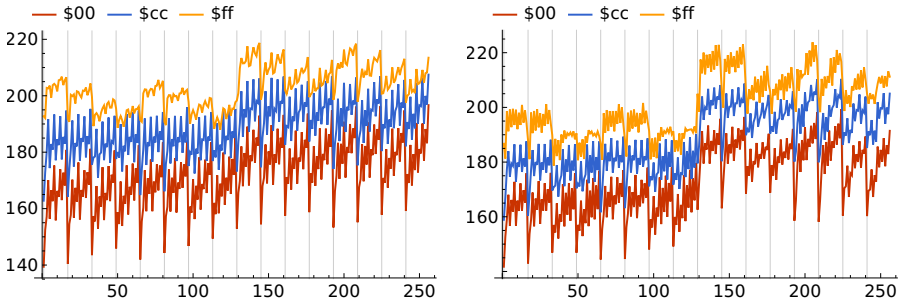
Review of a clock after execution of `ANDI` actually supports this theory. At 10th ns power consumptions resembles what I've seen at execution clock, but at 15th ns it starts to change into something, again, divided into blocks of size 16, but now with significantly different "base line" — inside those blocks it still resembles what I've seen at execution clock, but when taking each block as a whole, they differ in their power consumption quite noticeably, see 3.15a. This change goes even further and at 21 ns it resembles a bit different structure.

Despite my effort, I was not able to make anything specific out of this. It does differ if data in destination register are changed (see figure 3.16), yet I can't come up with any rule. Yes, there is some general increase in consumption when a larger constant is stored in destination register. There is regularity in the way ATMega163 consumes power at executing those instructions, so I decided to leave this information in a hope that it is still useful.



(a) Clock after execution, at 16 ns. (b) Clock after execution, at 21 ns.

Figure 3.15: Power consumption of `ANDI` with different immediate values and constant value stored at destination register at and the clock after.



(a) Execution clock at 16 ns. (b) Clock after execution, at 16 ns.

Figure 3.16: Power consumption of a clock after execution of `ANDI` with different immediate values and with different constants stored at destination register.

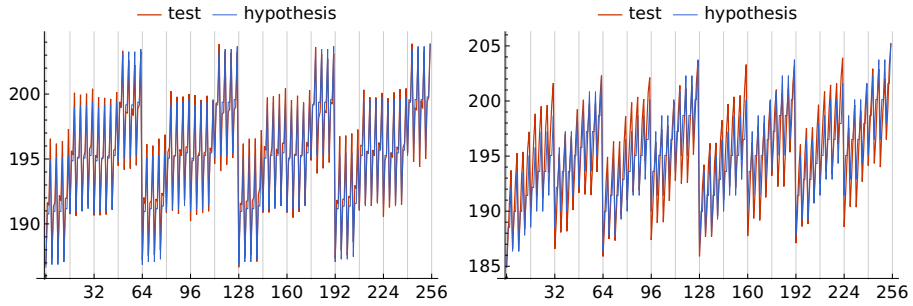
3.4.1.2.2 `OR` and `ORI`

Table 3.9: Detailed description of `OR` and `ORI` instructions.

Description	Logical OR Registers	Logical OR Register and Constant
Mnemonics	`OR`	`ORI`
Operands	`Rd, Rr`	`Rd, K`
Operation	`Rd` ← `Rd` ∨ `Rr`	`Rd` ← `Rd` ∨ `K`
Flags	Z,N,V	Z,N,V
Opcode	`0010 10rd dddd rrrr`	`0110 KKKK dddd KKKK`
Clocks	1	1

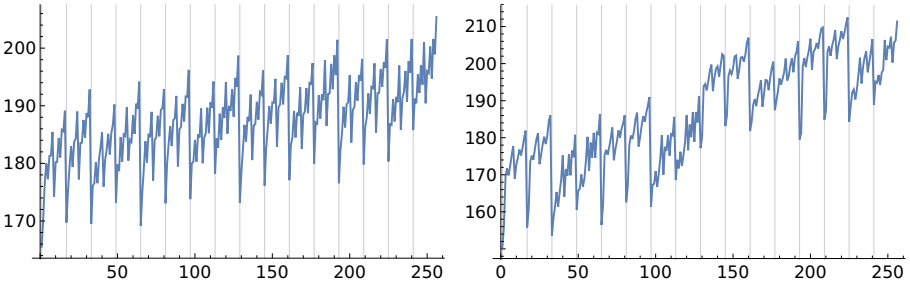
Instruction `OR` acts in a same predictable way as `AND` does. Dependency on Hamming weight of data stored in registers happens throughout execution. Dependency on a result occurs in the clock after at the beginning.

What is much more interesting to study is power consumption of a `ORI`. It does notably differ from that of `ANDI`. Now at the clock of execution there could



(a) Power consumption at 16 ns (b) Power consumption at 20 ns

Figure 3.17: Power consumption of clock after `OR r16, K`, with `r16 = $cc` and all possible `K` at different points of execution with respective hypotheses.



(a) Clock after execution, at 13 ns. (b) Clock after execution, at 21 ns.

Figure 3.18: Power consumption of `ORII` with different immediate values and constant value stored at destination register at and the clock after.

be something, that actually does resemble progression of Hamming weight of immediate value.

Yet the clock after execution shows similar progression of power consumption as `ANDI`. At the 13 ns there is dependency on immediate value, and at 21 ns there is structure that resemble that of `ANDI` (see figure 3.18).

3.4.1.2.3 `EOR`

Table 3.10: Detailed description of `EOR` instruction.

Description	Exclusive OR Registers
Mnemonics	`EOR`
Operands	`Rd, Rr`
Operation	`Rd` $\leftarrow$ `Rd` $\oplus$ `Rr`
Flags	Z,N,V
Opcode	`0010 01rd dddd rrrr`
Clocks	1

Exclusive OR (in ATMega163 instruction set `EOR`, otherwise known as `XOR`) is the key operation in the whole cryptography. It's power consumption at execution doesn't differ from other logical operation instructions such as `AND` and `OR`. At execution clock microcontroller exposes Hamming weight of data stored in destination register.

Second clock exposes dependency on Hamming weight of data at the beginning, after 15 ns it looks like it is now more dependent on Hamming weight of a fraction of a data stored in source register.

3.4.1.2.4 `COM` and `NEG`

Table 3.11: Detailed description of `COM` and `NEG` instructions.

Description	One's Complement	Two's Complement
Mnemonics	`COM`	`NEG`
Operands	`Rd`	`Rd`
Operation	`Rd` ← `$FF - Rd`	`Rd` ← `$00 - Rd`
Flags	Z,C,N,V	Z,C,N,V,H
Opcode	`1001 010d dddd 0000`	`1001 010d dddd 0001`
Clocks	1	1

`COM` and `NEG` instructions both perform very similar operation, with only difference that `NEG` practically adds one to the result. What seems to be rad when one sees those instructions in the datasheet is their description. Not that those descriptions are incorrect — they are perfectly fine with regard of the result — but rather a strange way to describe them. Why not write for `COM`, for example, that this instruction performs bit negation? Well, analysis of power consumption of those instructions seems to explain it.

As it is common, first clock of execution of both instruction depends solely on Hamming weight that is is destination register. One clock after execution shows more differences, and with `NEG` dependencies are exactly those that are expected. At the beginning at 8 ns power consumption is still dependent on a Hamming weight of data previously stored, and at the Hamming weight of the result with ratio 1/4 respectively and with correlation 0.931. Over time dependency on the result grows and at 24 ns power consumption is dependent on those values with ratio 1/1 with correlation 0.957.

But `COM` acts in another way. First of all, clock after execution is not dependent on data, but rather on the lower five bits. To that adds dependency on the Hamming distance between bit negated and result.

3.4.2 Bit and bit-test instructions

Bit manipulations are also ALU instructions. Even more, shifts to the left are all aliases for addition register to itself, "logical shift left" `LSL Rd` stands for `ADD Rd, Rd` and "rotate left through carry" `ROL Rd` stands for `ADC Rd, Rd`. For information on those instruction reader can see subsection 3.4.1, where I described how those instruction behave in general.

For instructions, that manipulate with SREG, `BSET` and `BCLR`, exist eight aliases for each, so those were not mentioned.

Another instructions that I won't touch here are `NOP` (stands for "no operation", was used in analyzing instruction address dependency, see), and `SBI`, `CBI` (both manipulation with I/O registers), `SLEEP` and `WDR` for a reason, that I can not test instruction itself because effect it has is much more significant in consumption.

3.4.2.0.1 Shifts to right

Table 3.12: Detailed description of instructions, representing different shifts to the right.

Description	Arithmetic	Logical
Mnemonics	`ASR`	`LSR`
Operands	`Rd`	`Rd`
Operation	`Rd(n)` ← `Rd(n+1)`, `n=0..6`	`Rd(n)` ← `Rd(n+1)`, `Rd(7)` ← `0`
Flags	Z,C,N,V	Z,C,N,V
Opcode	`1001 010d dddd 0101`	`1001 010d dddd 0110`
Clocks	1	1
Description	Rotate through Carry	
Mnemonics	`ROR`	
Operands	`Rd`	
Operation	`Rd(7)` ← `C`, `Rd(n)` ← `Rd(n+1)`, `C` ← `Rd(0)`	
Flags	Z,C,N,V	
Opcode	`1001 010d dddd 0111`	
Clocks	1	

All shifts start with fetching the operand, therefore every such operation exposes dependency on data in register. In the clock of `ASR`, `LSR` and `ROR` dependency on data outgrows any dependency on result.

Again, clock after actual execution there is dependency on final result. It is combination of data that was processed, result and Hamming distance between result and data. I found correlation around 0.6–0.7 between consumption at 16 ns and my hypothesis (original data, difference between data and result

and result with ratio 4/1/1). Yet still it looks like to me, that I might be missing something in my hypothesis (that is why correlation is relatively low in comparison to other tests).

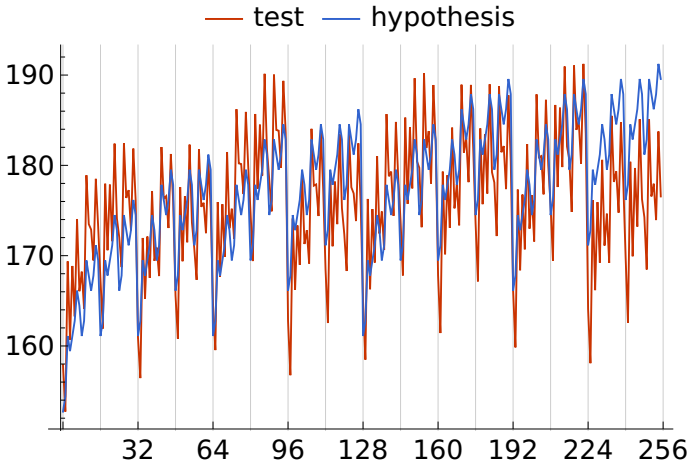


Figure 3.19: Clock after execution of `ASR` in comparison with my hypothesis, that power consumption is dependent on original data and a result.

Figure 3.19 shows that my hypothesis and actual power consumption at 16 ns in clock after execution are similar, but not so neatly aligning as I expected.

3.4.2.0.2 `SWAP`

Table 3.13: Detailed description of `SWAP` instruction.

Description	Swap Nibbles
Mnemonics	`SWAP`
Operands	`Rd`
Operation	`Rd(3..0)` ← `Rd(7..4)`, `Rd(7..4)` ← `Rd(3..0)`
Flags	-
Opcode	`1001 010d dddd 0010`
Clocks	1

This is ALU instruction, so, again, basic rules, that I described earlier in this work, apply. Data fetch starts at 18 ns and reaches it's peak at 25 ns. Dependency on result is found in the clock after. First at 9 ns there is still data dependency, then at 11 ns dependency on Hamming distance between data processed and result starts to emerge. Dependency on Hamming distance is at max at 17 ns and then data dependency adds a little.

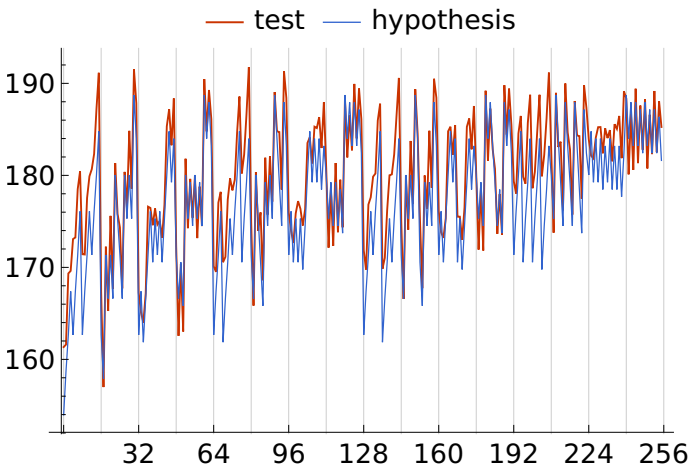


Figure 3.20: Dependency on a Hamming Distance between result and data processed at the clock after `SWAP` execution

3.4.2.0.3 `BSET` and `BCLR`

Table 3.14: Detailed description of `BSET` and `BCLR` instructions.

Description	Flag Set	Flag Clear
Mnemonics	`BSET`	`BCLR`
Operands	`s`	`s`
Operation	`SREG` ← `1`	`SREG` ← `0`
Flags	SREG(s)	SREG(s)
Opcode	`1001 0100 0sss 1000`	`1001 0100 1sss 1000`
Clocks	1	1

In a larger scale those instructions are insignificant. Not that their effect is useless, but in a scope of power analysis attacks event though it can give some useful information, but the problem here is that analyzing such a instruction consumes more effort than it gives a result.

Analysis shows that those instructions leak Hamming weight of a SREG and a result, and power consumption of a `BSET` differs more than `BCLR` when manipulating with bit at ranging positions of a SREG.

3.4.2.0.4 `BST` and `BLD`

Table 3.15: Detailed description of `BST` and `BLD` instructions.

Description	Bit Store from Register to T	Bit load from T to Register
Mnemonics	`BST`	`BLD`
Operands	`Rr, b`	`Rd, b`
Operation	`T` ← `Rr(b)`	`Rd(b)` ← `T`
Flags	T	-
Opcode	`1111 101d dddd 0bbb`	`1111 100d dddd 0bbb`
Clocks	1	1

I tested `BST` and `BLD` with all 256 values of data stored in register and with different position of bit to manipulate with. Whilst we can not expect significant differences at execution `BST` (as I mentioned at subsection 3.4.1, I was not able to find any dependency on a flag change), `BLD` might affect result that gets stored in register.

Both of those instructions leak Hamming Weight of a data in register they manipulate with in the execution clock. Like other ALU instructions, some data dependency occurs in the clock after `BST` and `BLD`, however the clock after `BLD` instruction presents dependency, that appears to be dependency on data before processing, not after.

3.4.3 Data transfer instructions

Data transfer instructions do not perform any operations, that alter data, only transfer them from one place to another. ATMega163 can transfer data from register to register, from register to memory location and vice versa, but it is not able to transfer data from one memory location to another without register manipulation in between.

The only instructions that were not described are I/O space access instructions.

3.4.3.1 Manipulations only with registers

3.4.3.1.1 `MOV` and `MOVW`

Table 3.16: Detailed description of `MOV` and `MOVW` instructions.

Description	Move Between Registers	Copy Register Word
Mnemonics	`MOV`	`MOVW`
Operands	`Rd, Rr`	`Rd, Rr`
Operation	`Rd` ← `Rr`	`Rd+1:Rd` ← `Rr+1:Rr`
Flags	-	-
Opcode	`0010 11rd dddd rrrr`	`0000 0001 dddd rrrr`
Clocks	1	1

Ability to copy data from register to register is one of the basic requirements for microcontroller to operate. `MOV` provides such operation.

Obviously, this instruction leaks data from both registers. Execution of this instruction starts with data fetch and it fetches values from both registers, despite the fact that it actually needs value of only one of them. In my opinion, ATMega163 performs this operation through ALU. Fetches destination register and then overwrites it's value by value of source register.

This instruction is easy in analysis. At execution clock fetch starts at 17 ns, and is clear at 24 ns with correlation between Hamming weight of data and power consumption at this point of execution around 0.991.

Clock after execution of `MOV` depends mostly on change between data originally stored in destination register and data that was in source register (as contents of destination register change). Figure 3.21 shows how power consumption at clock after execution exposes the result. My hypothesis was that power consumption is dependent on data (in a form of it's Hamming weight) and difference between old and new data (their Hamming distance). I found that at 16 ns of clock after execution power consumption does depend on those values with ratio 1/2 respectively. Logically power consumption is dependent more on data that were previously in destination register.

Moving to `MOVW` instruction, we notice something interesting about it: even though this instruction moves two registers it still performs in just one clock cycle.

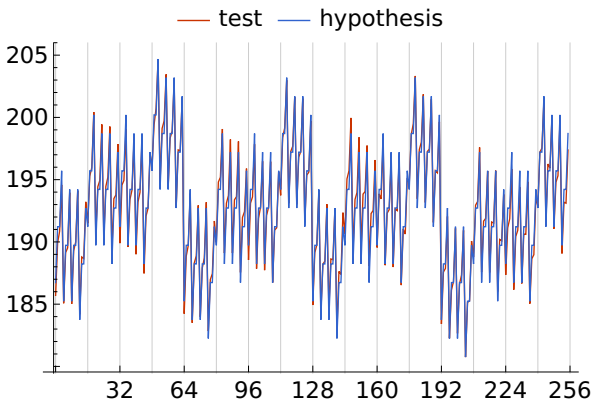


Figure 3.21: Power consumption of `MOV` with different values stored in source register and constant value stored at destination register at the clock after execution at 16 ns.

I examined this instruction with two tests: changing value in a high byte of word and in a low byte. I found out that power consumption at execution clock is not dependent on value stored in high byte of source register, which is in contrary with what `MOV` instruction looks like. Though it is still dependent on Hamming weight of data in low register.

Clock after shows dependency on difference between data previously stored in register word and new data (now depending on data stored in high byte too), yet for some reason much less clear and generally difference between consumptions of different values is lower than that of `MOV`.

3.4.3.1.2 `LDI`

Table 3.17: Detailed description of `LDI` instruction.

Description	Load Immediate
Mnemonics	`LDI`
Operands	`Rd, K`
Operation	`Rd` ← `K`
Flags	-
Opcode	`1110 KKKK dddd KKKK`
Clocks	1

`LDI` is a short way to load some constant into register. It is logical to use program memory to store constants, but it is not always necessary (`LPM` instruction can load data from program memory, more about it later) As for every instruction that uses immediate value, it is harder to defy particular dependency.

I tested this instruction with immediate value set to zero and different contents of destination register and with constant data stored at register and different immediate values.

Power consumption most definitely depends on Hamming weight of data stored in register, again, apparently this sort of "move" instruction operates through ALU. And certainly it depends on immediate value, or more precisely, on opcode.

One clock after execution power consumption depends on Hamming distance between data that were previously stored in destination register and new data. Also it is dependent on immediate value, but by which rule immediate values affect power consumption I was not able to defy, see 3.22.

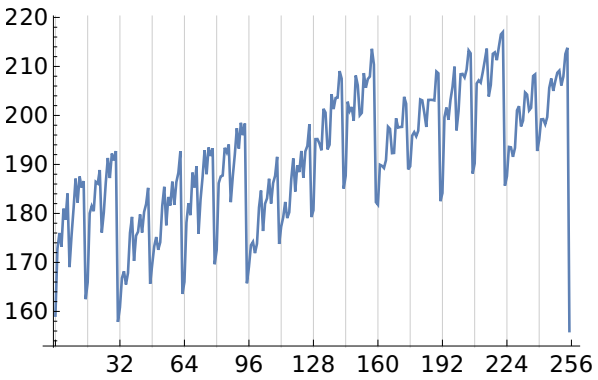


Figure 3.22: Power consumption of `LDI` with different immediate values and zero stored in destination register at the clock after execution at 24 ns.

3.4.3.2 Loads from SRAM

3.4.3.2.1 `LD` and `LDS` I would like to start with `LDS`. This instruction contains in opcode direct data memory address. First clock of execution seems like to be fetching address (as I wrote about at section 3.1), but difference between power consumption is actually not really clear, that there is little no chance to guess out what address is being processed. Second clock of execution is far, far more richer on dependency. At 13 ns of second clock of execution there is dependency on Hamming weight of address. Figure 3.23 show it, correlation between hypothesis is 0.94.

Table 3.18: Detailed description of `LD` and `LDS` instructions.

Description	Load Indirect	Load Direct from SRAM
Mnemonics	`LD`	`LDS`

Operands	`Rd, X`
	`Rd, Y`
	`Rd, Z`
		`Rd, k`

Operation	`Rd ← (X)`
	`Rd ← (Y)`
	`Rd ← (Z)`
		`Rd ← (k)`

Flags	-	-

Opcode	`X: 1001 000d dddd 1100`
	`Y: 1000 000d dddd 1000`
	`Z: 0000 000d dddd 1000`
		`1001 000d dddd 0000`
		`kkkk kkkk kkkk kkkk`

Clocks	2	2

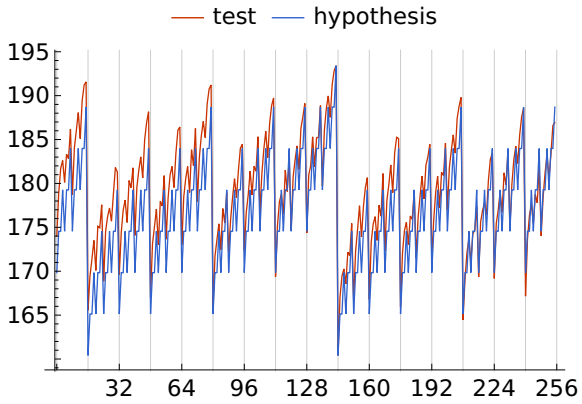


Figure 3.23: Power consumption of second execution clock of `LDS` with different addresses as operand value, loading 256 data entries from addresses starting with `$0070` at 13 ns.

One clock after execution of `LDS` there is residual consumption, that changes form to something, that I was not able to recognize (see 3.24).

Another thing I tested is how microcontroller will deal with bits in direct address in opcode, that can not be used (higher five to six bits, depending on if program tries to access addresses between range `$400` and `$45f`). I mentioned it at section 3.1.

What I found out is that in the first clock of `LDS` execution, there is difference in power consumption with changing values of six higher bits of direct address. And all the clocks after have practically indistinguishable power consumptions. This means, that at the first clock microcontroller tries to set exact value stored in opcode, and at the second clock the width of

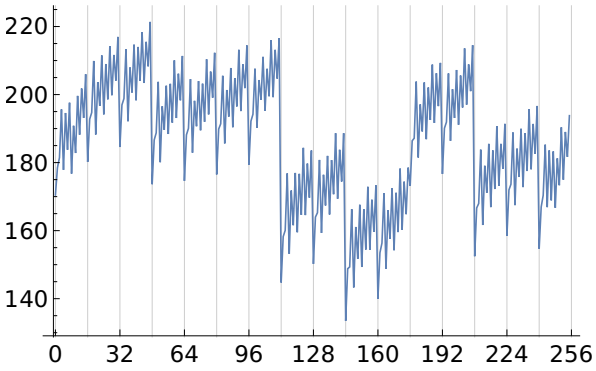


Figure 3.24: Power consumption of clock after execution of `LDS` with different addresses as operand value, loading 256 data entries from addresses starting with `$0070` at 22 ns.

address is cut due to actual processing of address. All of what I described can be seen at figure 3.25.

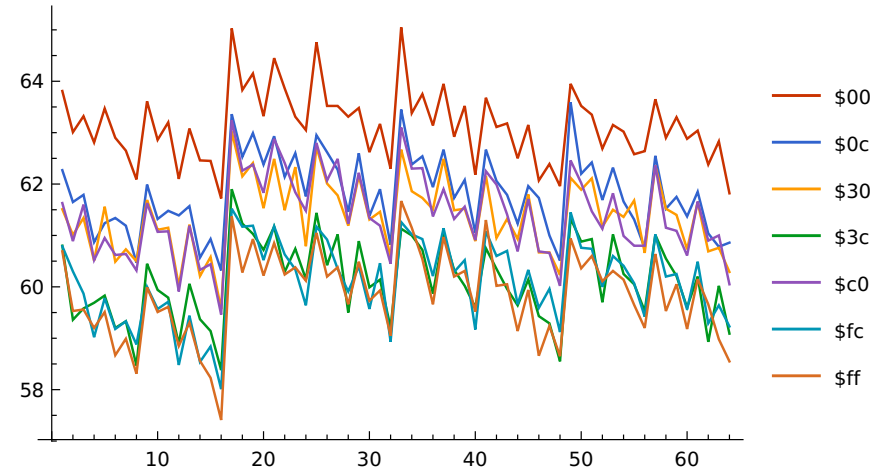
Proceeding to `LD` instruction, regarding address it acts almost the same, differences are:

- There is no "unknown" part in a equation: power consumption of this instruction is dependent plainly on Hamming weight of address processed.
- At every clock of execution this dependency shows up, not only in the second and the clock after execution for a short period of time, correlation between Hamming weight of address processed and power consumption goes up to 0.968.
- From the previous point it follows, that microcontroller at the first clock fetches address from register word.

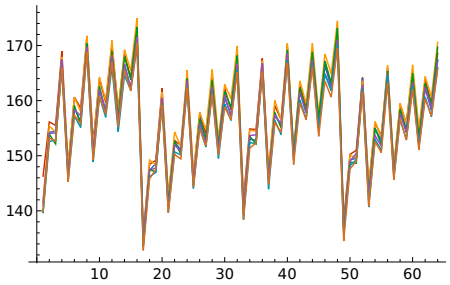
Summing up, address-procession-wise `LD` acts more predictable then `LDS`.

Since operations `LDS` and `LD` are practically (except for the addressing way) perform same operation, I decided to test data dependency only in power consumption of `LD`. To test it, I store test value at the specific constant address, load it to register, that is set to some constant value, increment test value and repeat from "store" point.

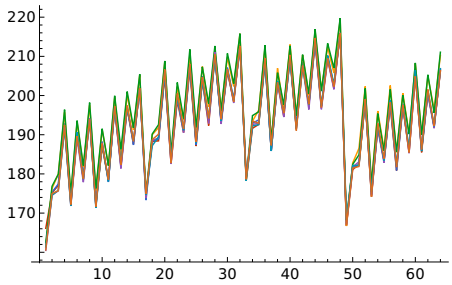
Logically, as I wrote in section 3.1, first clock of execution deals only with address, so while dealing with constant address no change is present. Since I process data in a row from 0 to 255, I can notice specific graphs, so what I saw at the second clock of execution is dependency on Hamming distance between subsequent data, see 3.29a. And only the next clock (clock after execution) is dependent on Hamming distance between data that were stored in register and that are loaded from SRAM, see 3.29b.



(a) First execution clock, 190 ns.



(b) Second execution clock, 21 ns.



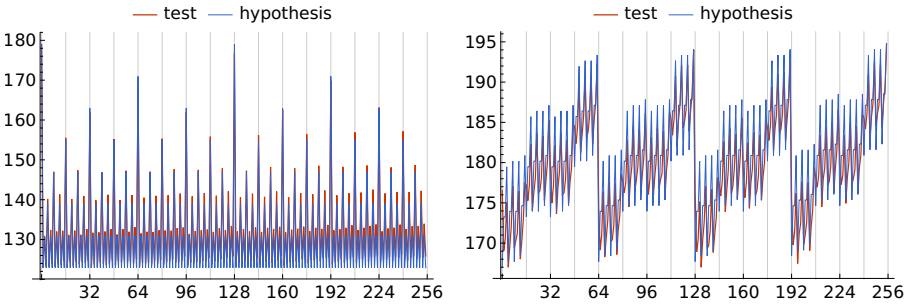
(c) Clock after execution, 21 ns.

Figure 3.25: Power consumption of `LDS` with addresses `$70` to `$b0`, with first six bits prepended with different values at different .

I think that dependency on Hamming distance between the data is due to change in what was accessed in SRAM before.

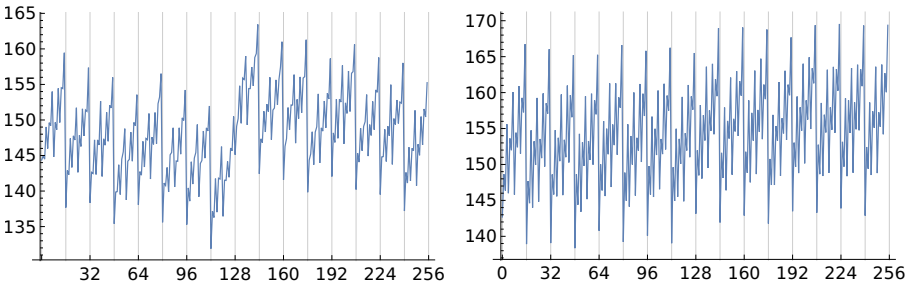
Next topic for discussion is dependency on a value stored in destination register. Test is designed like this: some value is stored at specific constant address, and for every `LD` execution data in destination register are different.

Again, at the first execution clock nothing happens because address is again kept constant. At the second execution clock dependency on Hamming weight of data stored in destination register shows roughly at 21 ns (with correlation to hypothesis around 0.965). And after that power consumption progression looks like Hamming weight of it's lower nibble. Similar process is occurring at the clock after execution, see both clock at figure 3.27.



(a) Second execution clock, 153 ns. (b) Clock after execution, 19 ns.

Figure 3.26: Power consumption of `LD` with constant address, different data are loaded into register that is set to `$cc`.



(a) Second execution clock, 28 ns. (b) Clock after execution, 28 ns.

Figure 3.27: Power consumption of `LD` with constant address, at loaded memory entry `$00` is stored, different data are stored in register.

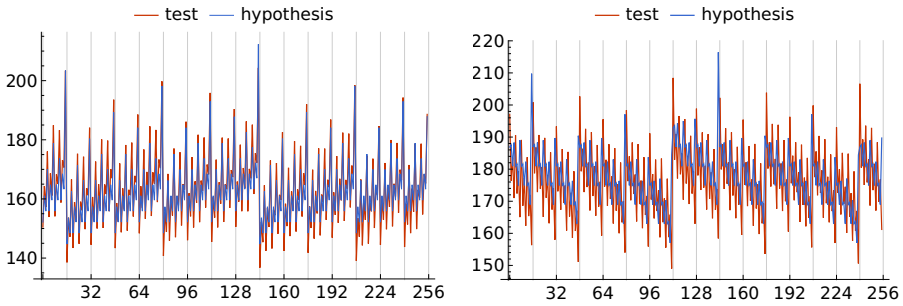
3.4.3.2.2 `LD` with Post-Increment and Pre-Decrement

Table 3.19: Detailed description of `LD` with Post-Increment and Pre-Decrement instructions.

Description	Load Indirect and Post-Inc.	Load Indirect and Pre-Dec.
Mnemonics	`LD`	`LD`
Operands	`Rd, X+`	`Rd, -X`
	`Rd, Y+`	`Rd, -Y`
	`Rd, Z+`	`Rd, -Z`
Operation	`Rd ← (X++)`	`Rd ← (--X)`
	`Rd ← (Y++)`	`Rd ← (--Y)`
	`Rd ← (Z++)`	`Rd ← (--Z)`
Flags	-	-
Opcode	`X: 1001 000d dddd 1101`	`X: 1001 000d dddd 1110`
	`Y: 1001 000d dddd 1001`	`Y: 1001 000d dddd 1010`
	`Z: 1001 000d dddd 0001`	`Z: 1001 000d dddd 0010`
Clocks	2	2

These instructions by themselves alter contents of address register. This functionality is especially great when processing arrays of data.

At the first clock of execution microcontroller fetches data from address register. At the second clock of execution it increments/decrements the address. Dependency on data stored in address register before increment starts at 10 ns and roughly around 14 ns starts to emerge dependency on Hamming distance between original address and incremented/decremented version. With ratio 1/1 (Hamming weight of previous address and Hamming distance between this and new address) correlation between hypothesis and power consumption at 17 ns is roughly 0.93.



(a) `LD r16, Y+`. (b) `LD r16, -Y`.

Figure 3.28: Power consumption at the second clock of `LD` with increment/decrement, both destination register and memory entry at tested addresses are set to `$00`, with respective hypothesis, at 17 ns.

Clock after execution power consumption is dependent on Hamming weight of a new address and Hamming distance between old and new address.

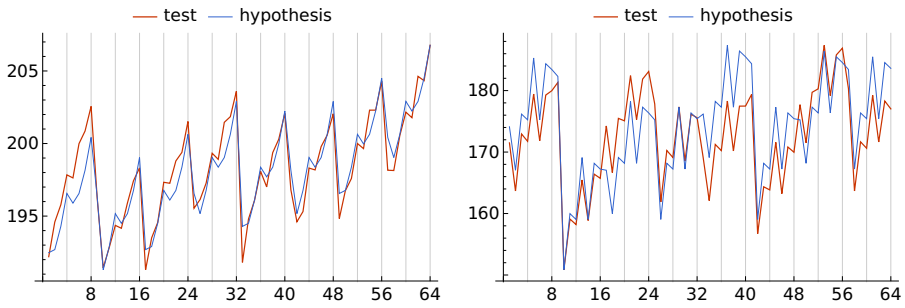
3.4.3.2.3 `LDD`

Table 3.20: Detailed description of `LD` with displacement.

Description	Load Indirect with Displacement
Mnemonics	`LDD`
Operands	`Rd, Y+q`
	`Rd, Z+q`
Operation	`Rd ← (Y+q)`
	`Rd ← (Z+q)`
Flags	-
Opcode	`Y: 10q0 qq0d dddd 1qqq`
	`Z: 10q0 qq0d dddd 0qqq`
Clocks	2

First clock of `LDD` seem to be loading a displacement value from opcode. Second clock depends on the displacement: on Hamming weight of a base

address, Hamming weight of a displacement and Hamming distance between the two. But as it is always a deal with immediate values, not as clear as if it was a dependency on a data in register.



(a) 10 ns (b) 15 ns.

Figure 3.29: Power consumption at the second clock of `LDD`, both destination register and memory entry at tested addresses are set to `$00`, with respective hypothesis, address is set to `$77`.

Clock after execution shows dependency on Hamming weight of a displaced address.

3.4.3.3 Stores to SRAM

3.4.3.3.1 `ST`

Table 3.21: Detailed description of `ST` and `STS` instruction.

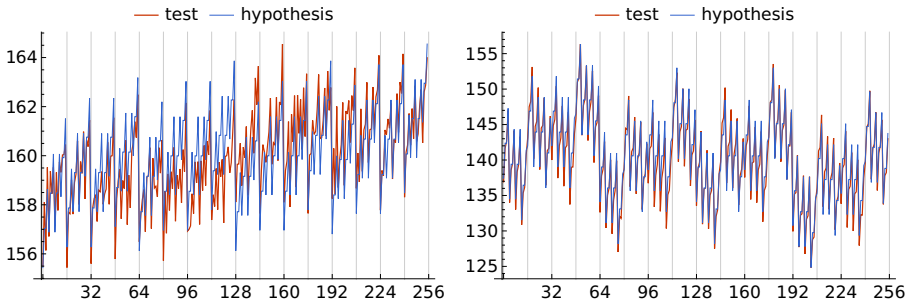
Description	Store Indirect	Store Direct from SRAM
Mnemonics	`ST`	`STS`
Operands	`X, Rr`	`k, Rr`
	`Y, Rr`	
	`Z, Rr`	
Operation	`(X)` ← `Rr`	`(k)` ← `Rr`
	`(Y)` ← `Rr`	
	`(Z)` ← `Rr`	
Flags	-	-
Opcode	`X: 1001 001r rrrr 1100`	`1001 001d dddd 0000`
	`Y: 1000 001r rrrr 1000`	`kkkk kkkk kkkk kkkk`
	`Z: 1000 001r rrrr 0011`	
Clocks	2	2

In a way complement operation to “loads” is “store”.

Analyzing just address dependency of a `STS` instruction doesn’t give any new information to what I’ve described earlier in analysis of `LDS` instruction.

Logically address processing is practically same operation for both instructions.

Much more noteworthy are data dependencies, since those instructions are practically inverse. Described in paragraph 3.4.3.2.1, power consumption of operation "load" in the second clock of execution depends on a Hamming distance between value that is now being loaded from memory location and value that had been previously processed by SRAM controller.



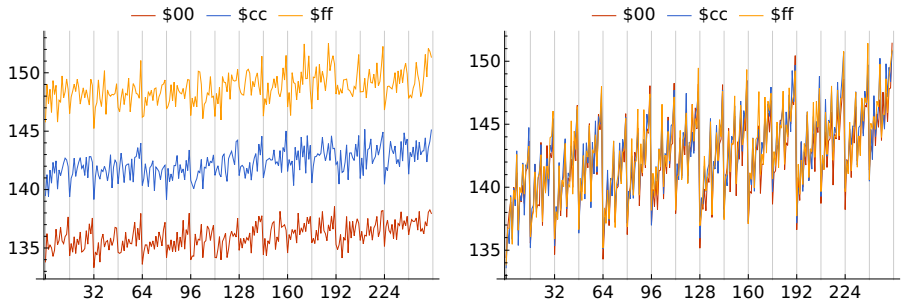
(a) 24 ns. (b) 47 ns.

Figure 3.30: Power consumption of `ST` with constant address, different data are stored from register, with memory entry set to `$cc`.

Power consumption of `ST` is radically different from that. At 24 ns of second execution clock for a brief period of 5 ns there shows a dependency on Hamming weight of data stored in source register. But not exactly on the whole value, at 31 ns there is significant change, that exposes that power consumption at this point is dependent rather on a lower nibble of data stored in source register, as if microcontroller processes those data by halves. After 32 ns dependency on the Hamming distance between data stored previously at the memory entry and data in source register starts to prevail an at 47 ns power consumption is fully dependent on that value. Figure 3.30 shows power consumption and comparison with hypothesis.

Clock after execution exposes a residual power consumption after second clock of execution, but at much lesser scale.

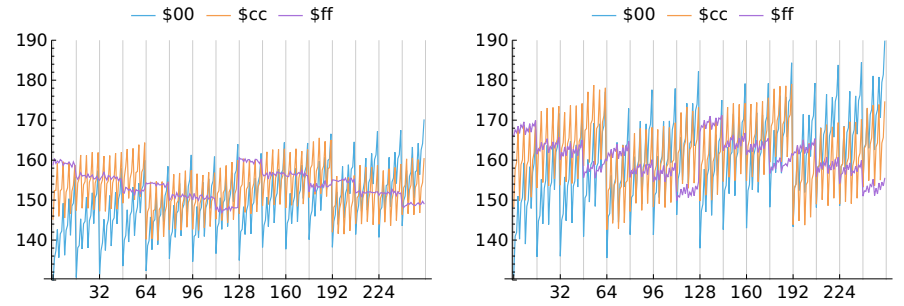
In comparison to dependency on a data stored in source register, dependency on data stored in memory is much less easy to spot. Generally, dependency on data in memory occurs later than dependency on data in register, as seen on figure 3.31. Notice how at 21 ns on figure 3.31a there is significant difference in overall consumption with different constants stored in register, and how on figure there is practically no difference with different data stored in memory.



(a) Value in register is set to constant value, data in memory change.

(b) Value in register change, but memory entry value remains the same.

Figure 3.31: Power consumption of `ST` tested at 21 ns.



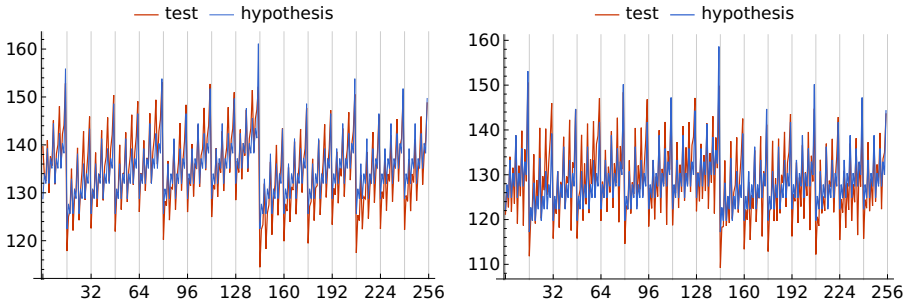
(a) Value in register is set to constant value (see legend), data in memory change, at 35 ns.

(b) Value in register change (see legend), but memory entry value remains the same, at 39 ns.

Figure 3.32: Power consumption of `ST` in dependency on data in source register and those stored in memory.

Otherwise power consumption depends on the same values: Hamming weight of data in register and in memory, Hamming distance between those values (exposing that there is change occurring) Hamming weight of a lower nibble of both register value and memory value. Another difference is that whilst exhibiting almost the same behavior, value in register holds more weight than data in memory. As seen on figure 3.32, changes in value stored in register provoke more difference between power consumption.

Clock after execution expose only residual power consumption. When compared to a behavior of `LD` instruction, where clock after exposes data dependency very clearly, differences in consumption of `ST` are insignificant. This can be probably connected to the fact that power consumption of transitions in SRAM is less significant then transitions in registers.



(a) LD with increment. (b) ST with increment.

Figure 3.33: Power consumption of ST and LD at 50 ns with increment and respective hypotheses.

3.4.3.3.2 ST with Post-Increment and Pre-Decrement

Table 3.22: Detailed description of ST with Post-Increment and Pre-Decrement instructions.

Description	Store Indirect and Post-Inc.	Store Indirect and Pre-Dec.
Mnemonics	ST	ST
Operands	X+, Rr	-X, Rr
	Y+, Rr	-Y, Rr
	Z+, Rr	-Z, Rr
Operation	(X++) ← Rr	(--X) ← Rr
	(Y++) ← Rr	(--Y) ← Rr
	(Z++) ← Rr	(--Z) ← Rr
Flags	-	-
Opcode	X: 1001 001r rrrr 1101	X: 1001 001r rrrr 1110
	Y: 1001 001r rrrr 1001	Y: 1001 001r rrrr 1010
	Z: 1001 001r rrrr 0001	Z: 1001 001r rrrr 0010
Clocks	2	2

Power consumption of ST with increment/decrement differs from that of same functionality LD. This instruction exposes more dependency on Hamming distance between address and result of it's increment or decrement.

Figure 3.33 compares power consumption of LD and ST with post-increment. As a reference point I used 50 ns of second clock of execution. My hypothesis about power consumption of LD is that at this point in time it is dependent on Hamming weight of a currently processed address and Hamming distance between current and new address with ratio close to 1/1. Hypothesis for power consumption of ST is close to 1/2 respectively. Reader can notice that despite seemingly different hypothesis, those difference are very subtle.

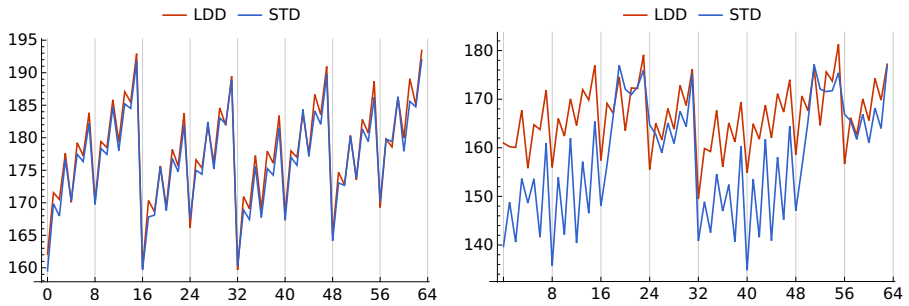
3.4.3.3.3 STD

Table 3.23: Detailed description of ST with displacement.

Description	Store Indirect with Displacement
Mnemonics	STD
Operands	Rd, Y+q
	Rd, Z+q
Operation	Rd ← (Y+q)
	Rd ← (Z+q)
Flags	-
Opcode	Y: 10q0 qq1d dddd 1qqq
	Z: 10q0 qq1d dddd 0qqq
Clocks	2

Execution of STD is similar to LDD at the first clock of execution. Those instructions are also similar in a way they confuse power consumption of other clock and the clock after execution.

Until the 13 ns in the second clock of execution they behave similarly expectable: power consumption of both of them is dependent on Hamming weight of displacement value and Hamming distance between address before and after displacement. From this point to 21 ns dependency on Hamming distance grows. And after around 25 ns power consumption of LDD and STD starts to differ. Figure 3.34 illustrates this phenomena.



(a) 13 ns. (b) 28 ns.

Figure 3.34: Comparison between power consumption of a second clock of instructions LDD and STD.

Here I think my hypothesis about dependency on opcode makes sense: opcode of LDD and opcode of STD differ in one bit (when same word register used). Problem with it is that I was not able to found exact way in which power consumption depends on it.

3.4.3.4 Stack manipulations

Table 3.24: Detailed description of `PUSH` and `POP` instructions.

Description	Push Register on Stack	Pop Register from Stack
Mnemonics	`PUSH`	`POP`
Operands	`Rr`	`Rd`
Operation	`Rr` ← `STACK`	`SP` ← `SP + 1`
	`SP` ← `SP - 1`	`Rd` ← `STACK`
Flags	-	-
Opcode	`1001 001d dddd 1111`	`1001 000d dddd 1111`
Clocks	2	2

From a practical point of view stack instructions are same as `LD` and `ST` with increment and decrement, with a few differences:

- `LD` and `ST` can do both increment and decrement, respective `POP` and `PUSH` are left with only one,
- `LD` and `ST` perform post-increment and pre-decrement, while `PUSH` does post-decrement and `POP` pre-increment: this is because stack grows down,
- stack instructions can load/store indirectly using only one pre-set register, basic memory instructions can pick from three general purpose registers.

With that said it's only natural to expect similar behavior to basic memory instructions. I've set stack pointer to the end of reserved address space I used in previous memory tests.

First I examined how are those instructions dependent on the value of a stack pointer (memory location). In both instructions dependency on the Hamming weight of value of a stack pointer starts to show at 9 ns.

First clock of execution is a little bit dependent on address, but not clearly. Second clock of execution differs in consumption much more and actually exposes action performed with stack pointer. `PUSH` instruction (*post-decrement*) during the whole clock is dependent on the current value of a stack pointer, whilst `POP` (*pre-increment*) is dependent on it only at the start (around 9 ns), and after this point starts to convert to the dependency on Hamming weight of the incremented stack-pointer and Hamming distance between old and new SP value.

On clock after execution shows dependency on previous clock.

Second test was dependency on data. Expectedly, first clock of `PUSH` execution was highly dependent on Hamming weight of data stored in register with correlation 0.995, while first clock of `POP` did not show any dependency.

Second clock of `PUSH` starts again with dependency on Hamming distance between data stored at the address of stack pointer and data stored in source register. This dependency reaches peak of dependency on this difference between values at 12 ns with ratio 1/1 and then it regains dependency on data in register at 20 ns with ratio 1/5 with dominant component of Hamming weight of value in register.

Second clock of `POP` now exposes dependency on Hamming distance between data stored in destination register and data that arrived from stack.

Clock after execution shows clear dependency on Hamming weight of data now stored in register no matter what was there before.

These two instructions behave in a very direct and clear fashion and therefore are great targets of DPA attacks.

3.4.3.5 Manipulations with Program Memory

Table 3.25: Detailed description of `LPM` instruction.

Description	Load Program Memory
Mnemonics	`LPM`
Operands	None
	`Rd`
	`Rd, Z+`
Operation	`R0` ← `(Z)`
	`Rd` ← `(Z)`
	`Rd` ← `(Z++)`
Flags	-
Opcode	`1001 0101 1100 1000`
	`1001 000d dddd 0100`
	`1001 000d dddd 0101`
Clocks	3

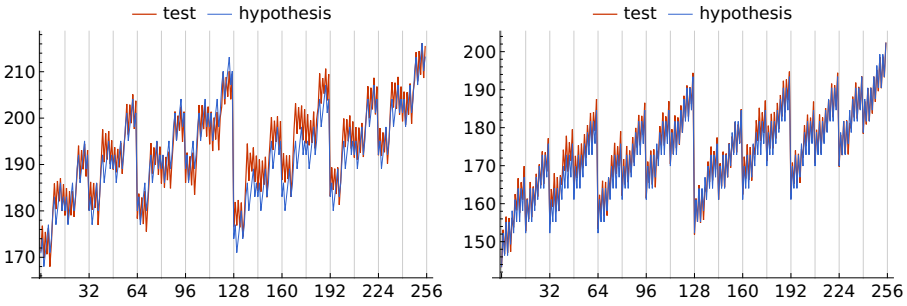
`LPM` is different from regular memory instructions: it doesn't manipulate with SRAM where data are stored, but with flash memory, where is stored the firmware. I tested every variant of this instruction.

First clock of execution actually depends on a current value of Program Counter. Since `LPM` loads data to register, it points to byte, whilst PC addresses point to words. That means, that in order to load lower byte of a word, microcontroller needs to shift left by one target address. To load higher byte of a word, it shifts the address and adds one. This means that ATMega163 has Little-Endian order of storing data, but it is a digression.

As a means to explain this behavior, my hypothesis is that flash memory controller kept some residual value of the memory location it last accessed during the fetch of instruction. What I found is that power consumption depends on Hamming weight of target address and Hamming distance between

target address and PC value (which is at the time of execution of `LPM` is address of an instruction after it) times 2 (i.e. shifted to the left).

At the beginning of execution clock one at 9 ns microcontroller accesses flash memory at the target address, so power consumption here depends largely on Hamming distance between target address and shifted value of PC. At the 13 ns reaches balance between those values and after that dependency on Hamming weight of a target address grows and at 28 ns dependency on target address is twice as significant as dependency on difference between PC and target. For the last point correlation between power consumption and hypothesis is 0.992. Figure 3.35 shows power consumption at the last two points together with respective hypotheses. Second clock shows behavior similar to that of first clock at 13 ns throughout whole clock.



(a) 13 ns. (b) 28 ns.

Figure 3.35: Power consumption of first clock of `LPM` execution from address `$582`, accessing 256 different addresses.

All the versions of `LPM` exhibit the same behavior at all of the clocks of execution. Even `LPM Rd, Z+`, which is unexpected: I was not able to find at which point happens to be increment of address register. Power consumption is dependent on Hamming weight of data loaded at the third clock of execution.

3.4.4 Branch instructions

Branch instructions are program flow control instructions. Without those, program can not have if-else statements, loops, conditional processing and so on and so forth. ATMega163 has plenty of them.

Instructions, that I won't describe in this section are instructions, that in manipulate with input and output of microcontroller. And again, as with bit manipulation instructions, there is a lot of mnemonics for conditional branches, which are in fact only aliases, so it is not necessary to talk about those.

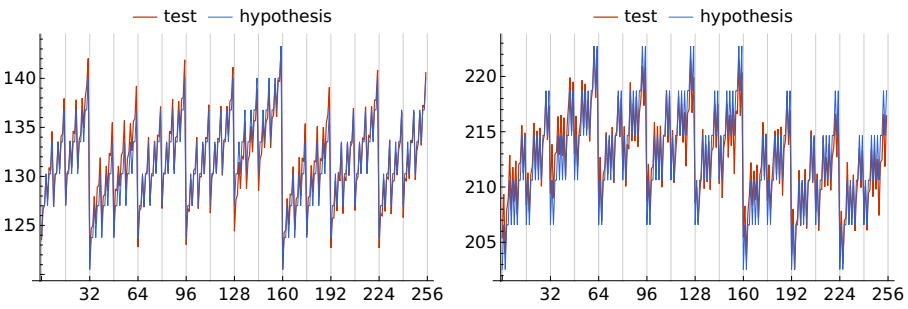
3.4.4.1 Jumps

3.4.4.1.1 `IJMP`

Table 3.26: Detailed description of `IJMP` instruction.

Description	Indirect Jump to (Z)
Mnemonics	`IJMP`
Operands	-
Operation	`PC` ← `Z`
Flags	-
Opcode	`1001 0100 0000 1001`
Clocks	2

First clock of `IJMP` depends on a value in `Z` register, in other words, on a target address. Correlation between Hamming weight of target address and power consumption of the first clock at 33 ns is 0.97. Second clock of `IJMP` shows dependency on target address and difference between it and PC with ratio around 1/1. Figure 3.36 presents both clocks with described hypotheses.



(a) First clock, 33 ns. (b) Second clock, 9 ns.

Figure 3.36: Power consumption of `IJMP` with respective hypothesis.

`IJMP` is instruction that is perfect for such analysis: it doesn't have any immediate values nor changing operands.

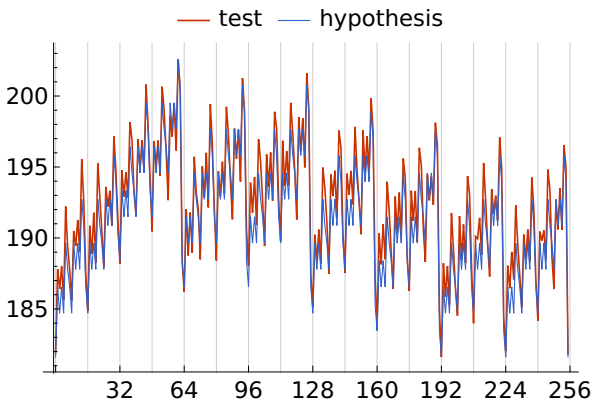


Figure 3.37: Power consumption of `RJMP`, third clock of execution at 10 ns.

3.4.4.1.2 `JMP`

Table 3.27: Detailed description of `JMP` instruction.

Description	Direct Jump
Mnemonics	`JMP`
Operands	`k`
Operation	`PC` ← `k`
Flags	-
Opcode	`1001 010k kkkk 110k` `kkkk kkkk kkkk kkkk`
Clocks	3

This instruction uses immediate value but in my tests used same target address. What can be said about this instruction, is that it has one more clock than other jumps. This is due to twice as large opcode than other such instruction have and it is a common thing for instruction, that has double-word opcode, to have one extra clock. As I mentioned, in my test opcode value was constant, and the first two clocks show same power consumption throughout every instance. Which leads me to a conclusion, that those clocks fetch address.

Third clock of execution is dependent on Hamming distance between current PC value and Hamming weight of a current address (at 10 ns correlation for ratio 4/1 respectively is 0.972). This point in execution can bee seen at figure 3.37 with respective hypothesis.

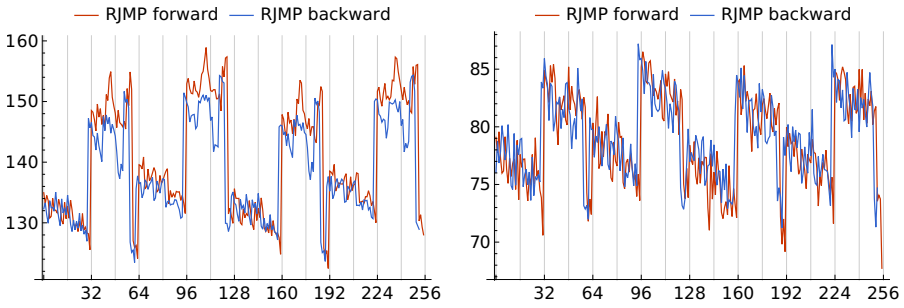
3.4.4.1.3 RJMP

Table 3.28: Detailed description of RJMP instruction.

Description	Relative Jump
Mnemonics	RJMP
Operands	k
Operation	PC ← PC + k + 1
Flags	-
Opcode	1100 kkkk kkkk kkkk
Clocks	2

RJMP is one of those instructions, where analysis lead me to nowhere. Due to the fact that relatively large proportion of this instruction is immediate value, and the fact that I was not able to understand how exactly immediate values affect power consumption, this instruction remains large mystery for me.

My test for this instruction included jumping to the same address from a number of places. I used IJMP to proceed through all RJMP's. Jumps were forward and backward, so I can see how different sign of immediate value affects power consumption. The obstacle I encountered analyzing this instruction is that while jumping from same addresses forward or backward with almost the same immediate value power consumption was almost same, which suggests opcode (or immediate) value dependency. The closest I could get to the actual power consumption dependency was hypothesis about somehow combining Hamming distance between RJMP address and target address and Hamming weight of a opcode, which resembles vaguely (and with a bit of imagination) power consumption of second execution clock at 1 ns, which is not a reliable source for analysis.



(a) 20 ns. (b) 130 ns.

Figure 3.38: Power consumption of RJMP, first clock of execution, both executed from same addresses, with different relative address value.

RJMP was performed from addresses starting with $30a every eighth address, and jumped to the address $b05 while jumping forward and to $2fd

jumping backward. With that Hamming weight of a opcode of jumps at each address remained same. Figure 3.38 compares power consumption of a first clock of `RJMP` execution. Note how similar those consumptions are, and yet there are differences: which is logical, since opcode value is not exactly the same. And test of `JMP` only proves the fact, that power consumption of a first execution clock is dependent on opcode and may be current/next instruction address.

3.4.4.2 Calls

3.4.4.2.1 `RCALL`

Table 3.29: Detailed description of `CALL`, `ICALL` and `RCALL` instructions.

Description	Long Call to a Subroutine	
Mnemonics	`RJMP`	
Operands	`k`	
Operation	`PC ← PC + k + 1`	
	`STACK ← PC + 1`	
Flags	-	
Opcode	`1001 010k kkkk 111k`	
	`kkkk kkkk kkkk kkkk`	
Clocks	4	
Description	Relative Call	Indirect Call
Mnemonics	`RJMP`	`IJMP`
Operands	`k`	-
Operation	`PC ← PC + k + 1`	`PC ← Z`
	`STACK ← PC + 1`	`STACK ← PC + 1`
Flags	-	-
Opcode	`1101 kkkk kkkk kkkk`	`1001 0101 0000 1001`
Clocks	5	5

Calls are practically just jumps with extra "push" operation. That is why this operation has one more clock in comparison to respective `JMP` instructions. At the first clock all `CALL` instructions push the PC value. Change of SP can be observed, together with value of a PC. At this point those instructions are quite similar to instructions that operate with stack.

Otherwise power consumption of those instructions is similar to that of their counterparts `JMP`.

3.4.4.3 Branches

3.4.4.3.1 `BRBS` and `BRBC`

Table 3.30: Detailed description of `BRBS` and `BRBC` instructions.

Description	Branch if Status Flag Set	Branch if Status Flag Cleared
Mnemonics	`BRBS`	`BRBC`
Operands	`s, k`	`s, k`
Operation	if `SREG(s) = 1` then `PC` ← `PC + 2 or 3`	if `SREG(s) = 0` then `PC` ← `PC + 2 or 3`
Flags	-	-
Opcode	`1111 00kk kkkk ksss`	`1111 01kk kkkk ksss`
Clocks	1/2	1/2

Branch instructions are able to conditionally jump relatively to 64 address forward and backward. First clock of execution fetches the immediate value stored in opcode, and checks the flag. Second clock, if condition matched, performs PC change to new relative value.

As it is the case with many instructions, that use immediate values, it is much harder to find exact dependency. Further more, testing of `BRBS` and `BRBC` is complicated due to small amount of values than can be tested in reliable fashion.

3.4.4.4 Comparisons and skips

3.4.4.4.1 Comparisons: `CP`, `CPC`, `CPI`

Table 3.31: Detailed description of instructions, representing different comparisons.

Description	Compare	C. with Carry
Mnemonics	`CP`	`CPC`
Operands	`Rd, Rr`	`Rd, Rr`
Operation	`Rd - Rr`	`Rd - Rr - C`
Flags	Z,N,V,C,H	Z,N,V,C,H
Opcode	`0001 01rd dddd rrrr`	`0000 01rd dddd rrrr`
Clocks	1	1
Description	C. Register with Immediate	
Mnemonics	`CPI`	
Operands	`Rd, Rr`	
Operation	`Rd - K`	
Flags	Z,N,V,C,H	
Opcode	`0011 KKKK dddd KKKK`	
Clocks	1	

Those comparisons are disguised SUB, SBC, SUBI instructions, the difference is that they do not write result to the destination register, but do change flags. With that said, both dependency on data in registers and a distance between are present in these instructions as well, but dependencies on a result and Hamming distance between result and data in register are much less significant.

Compare instructions can not really count as instructions that work with data (such as arithmetic and logic instructions or bit manipulation instruction), but rather control instructions, because they don't alter data in any way. They are supply for other instructions, most of the time they are applied for branch control (if statements, skips, loops, etc.). That is why instructions CP, CPC and CPI are in a group of "Branch instructions" and not anywhere else.

3.4.4.4.2 Skips: SBRS and SBRC

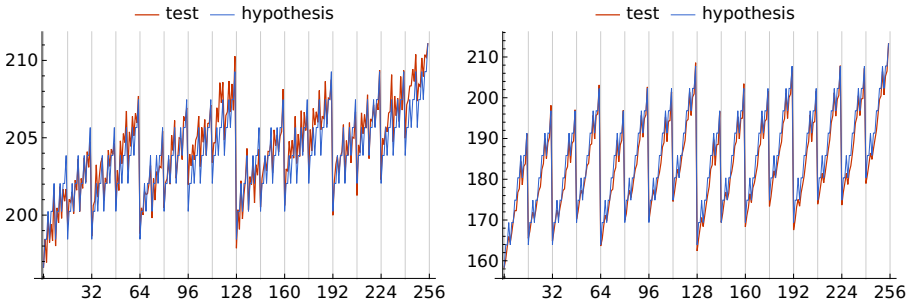
Table 3.32: Detailed description of SBRS and SBRC instructions.

Description	Skip if Bit in Register Set	Skip if Bit in Register Cleared
Mnemonics	SBRS	SBRC
Operands	Rr, b	Rr, b
Operation	if Rr(b) = 1	if Rr(b) = 0
	then PC ← PC + 2 or 3	then PC ← PC + 2 or 3
Flags	-	-
Opcode	1111 111r rrrr 0bbb	1111 110r rrrr 0bbb
Clocks	1/2/3	1/2/3

Power consumption of skip instructions depends data stored in register instruction manipulates with. First clock of execution depends solely on Hamming weight of data in source register. Second clock: the clock after execution or the clock where PC change happens, exposes Hamming weight of data in register as well. What should be noted is that if skip does occur, then will occur change in PC as well, so it will increase power consumption.

Before 13 ns power consumption keep dependency on Hamming weight of data, and after this point Hamming weigh of change in PC shows. At 23 ns power consumption is dependent on Hamming weigh of data and Hamming distance between PC (which is at the moment of SBRS or SBRC execution is set to the next address) and address after next instruction with ratio 1/2. Correlation to my hypothesis is 0.991. Figure 3.39 shows how second clock after execution start processes data at different points. Note that this figure presents both situations: if next instruction is skipped or not.

Skip Instructions can alter the length of program execution. For example, if one of skip instruction has to skip ADD or other instruction that is executed in one clock, the one who analyzes the power trace may not notice any change, especially if this instruction operates over same data. But if next instruction



(a) 11 ns. (b) 23 ns.

Figure 3.39: Power consumption of `SBRS r16, 4`, second clock of execution, different data are set into register, so skip does or does not occur in dependency on bit 4.

is executed in two or more clock cycles, and this part of the program is located in loop, so there are portions of power trace that repeat themselves, skip will change the length of one iteration.

3.4.4.4.3 `CPSE`

Table 3.33: Detailed description of `CPSE` instruction.

Description	Compare, Skip if Equal
Mnemonics	`CPSE`
Operands	`Rd, Rr`
Operation	if `Rd = Rr`
	then `PC` ← `PC + 2 or 3`
Flags	-
Opcode	`0001 00rd dddd rrrr`
Clocks	1/2/3

This instruction is a hybrid of two previously discussed groups of instructions: compare and skip. Difference between other skip instructions is that `CPSE` as a skip control bit uses zero flag from SREG. Otherwise this instruction act as expected: as `CP` in the first clock of execution and as `SBRS`, as zero flag is set if result of subtraction is zero (i.e. values in registers are equal), except this instruction doesn't expose value of status register.s

3.5 Analysis summary

In this section I would like to sum up the analysis by compare power consumption of different instructions and discuss general results of analysis.

3.5.1 Instruction comparison

As I analyzed instruction in groups largely by themselves, it is important to compare different instructions of different types. Generally instructions do not look the same in power trace. It depends on operation they perform.

Figure 3.40 shows how `NOP` might look like in power traces, when executed from different addresses. First thing that should be noted is position where PC change starts. It starts relatively early, since most of the instructions start to operate some time after the first large consumption peak. Another notable thing is that after peak power consumption goes down rapidly, making second small peak (that occurs due to imperfections of power supply) insignificant when compared to other instructions.

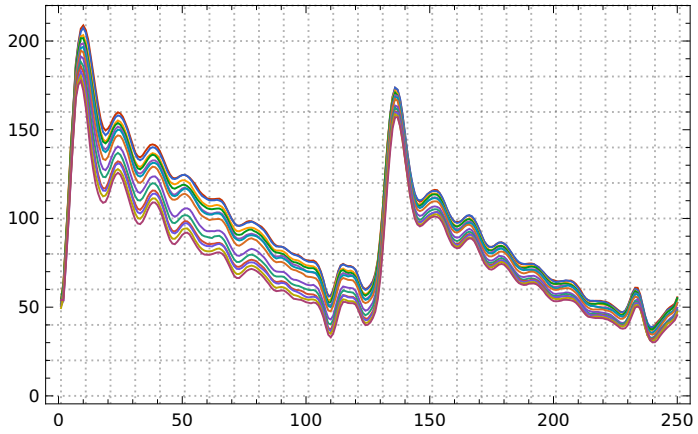
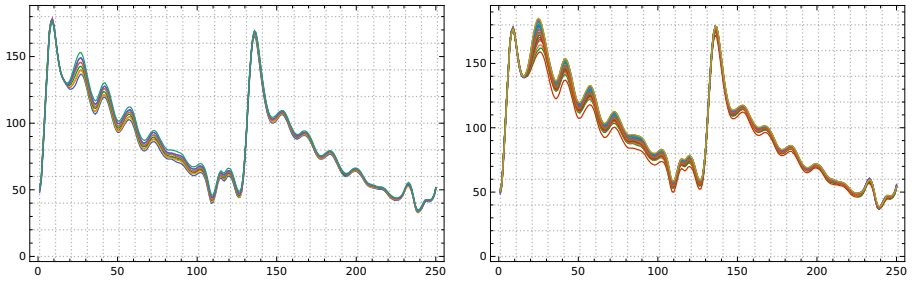


Figure 3.40: `NOP` with different Hamming distance between current and next address.

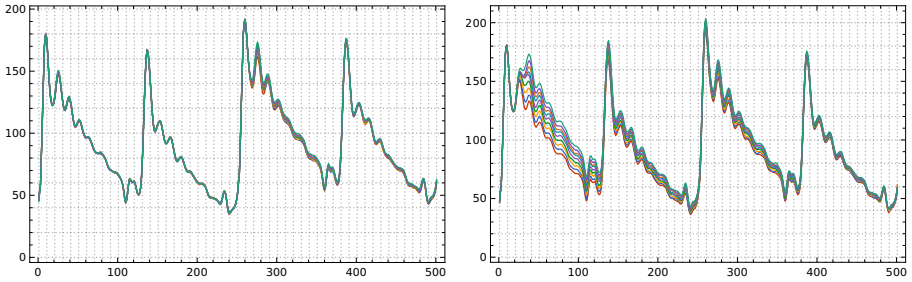
Another comparison I actually brought up before: `ADD` and `SUB`. Basically, all arithmetic and logic instructions look very much alike. The only significant difference is when data needs to be pre-processed, or generally require more transitions. Classic example could be `SUB`, which needs to prepare negative value of data in source register, and this operation generates more power consumption due to amount of transitions that need to be performed in order to change value to negative. Notice from the figure 3.41 how in comparison to `NOP` second smaller peak of those instructions is higher. Also we can observe that beginning of data fetch of those instructions occurs later than change in PC.



(a) Instruction `ADD` (b) Instruction `SUB`

Figure 3.41: Power consumption of two arithmetic instructions.

Data transfer instructions, especially when they manipulate with data memory, consume a lot of power. Inside the group of data transfer instructions, when considering memory access, there are two types of instructions: store and load. They differ in timings of events. As described in section 3.1, memory can not be accessed in a first clock of instruction. So `LD`-like instructions in the first clock are mostly dependent on address they are accessing, when `ST`-like instructions may also be dependent on data that is fetched from register. Those differences can be observed at figure 3.42.



(a) Instruction `LD` (b) Instruction `ST`

Figure 3.42: Power consumption of two data transfer instructions.

Branch instructions can reveal it's position, especially when used as a loop control. Actually, the very presence of some periodical action can be used as a mean of differential power analysis in the scope of only one power trace. On figure 3.43 it can be seen what significant change to power consumption instruction execution address brings.

3.5.2 Results

Power consumption depends on what and how microcontroller does. Greatest contribution to power consumption create data and addresses, and only their values, but differences between them. Hamming weight and Hamming distance can be used as very precise models of power consumption. With help

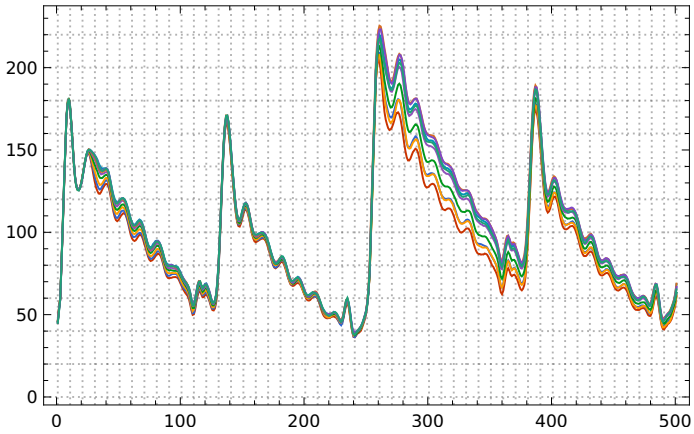


Figure 3.43: `IJMP` with different Hamming distance between current and next address.

of differential power analysis attacker can find out with great certainty what instruction does and what data it processes. When provided with a small number of power traces, it is still possible with some probability to reveal intermediate state of microcontroller.

Conclusion

The task of this work was to analyze power consumption of a microcontroller. Analysis revealed that ATMega163 can be really vulnerable: power consumption depends on data it processes, on operation it does, on addresses in Program Memory and in Data Memory. Some instructions even had the difference in consumption in dependency on data so significant, that it is possible to tell what particular program does with just a few power traces.

Not every test returned in a successful power consumption model. For some of instructions, despite the fact that they clearly had some dependency on what they operate with, I was not able to find acceptable hypothesis for their power consumption.

In the future, may be with some new experience and knowledge, I will be able to fill the gaps in this work, or it may be that someone else with a another point of view will find what I've missed. Another idea is to combine existing simulator solutions with a program, that will be capable of generating estimations of power consumption based on a source code and data.

I believe this work can be helpful for people, who would like to analyze some other microcontroller themselves, or for those, who design cryptographic systems to better secure their creations.

Working on this thesis I learned a lot about power analysis attacks and how they work, about AVR assembler and machine oriented languages in general, gained better understanding of microcontroller's architecture.

Bibliography

[1] *8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash.* 1142E–AVR. [5] Atmel Corporation. Feb. 2003. URL: http://ww1.microchip.com/downloads/en/DeviceDoc/doc1142.pdf.

[2] *AVR201: Using the AVR Hardware Multiplier, Application note.* http://ww1.microchip.com/downloads/en/AppNotes/Atmel-1631-Using-the-AVR-Hardware-Multiplier_ApplicationNote_AVR201.pdf. Accessed: 2018-04-25. Atmel Corporation.

[3] Paul Bottinelli and Joppe W. Bos. "Computational aspects of correlation power analysis". In: *Journal of Cryptographic Engineering* 7.3 (Sept. 2017), pp. 167–181. ISSN: 2190-8516. DOI: 10.1007/s13389-016-0122-9. URL: https://doi.org/10.1007/s13389-016-0122-9.

[4] Benedikt Gierlichs et al. "Revisiting Higher-Order DPA Attacks:" in: *Topics in Cryptology - CT-RSA 2010.* Ed. by Josef Pieprzyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 221–234. ISBN: 978-3-642-11925-5.

[5] Marc Joye and Francis Olivier. "Side-Channel Analysis". In: *Encyclopedia of Cryptography and Security.* Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 1198–1204. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_516. URL: https://doi.org/10.1007/978-1-4419-5906-5_516.

[6] Auguste Kerckhoffs. "La cryptographie militaire". In: *Journal des sciences militaires.* Ed. by Michael Wiener. Vol. IX. 1883, pp. 5–83, 161–191.

[5] Since Microchip took over Atmel in 2016, the original links to datasheets are broken, and atmel.com is redirected to microchip.com.

[7] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. ISBN: 978-3-540-48405-9.

[8] Paul Kocher et al. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering* 1.1 (Apr. 2011), pp. 5–27. ISSN: 2190-8516. DOI: 10.1007/s13389-011-0006-y. URL: https://doi.org/10.1007/s13389-011-0006-y.

[9] Rita Mayer-Sommer. "Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards". In: *Cryptographic Hardware and Embedded Systems — CHES 2000*. Ed. by Çetin K. Koç and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 78–92. ISBN: 978-3-540-44499-2.

[10] *MI-BHW.16, course material*. https://edux.fit.cvut.cz/courses/MI-BHW.16/. Accessed: 2018-03-10.

[11] *SOSSE - Simple Operating System for Smartcard Education*. http://www.mbsks.franken.de/sosse/. Accessed: 2018-03-10.

[12] Thomas Popp Stefan Mangard Elisabeth Oswald. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. 1st ed. 2007. ISBN: 9780387381626.

Appendix A

Acronyms

DPA Differential Power Analysis

SPA Simple Power Analysis

APDU Application Protocol Data Unit

PC Program Counter

SP Stack Pointer

ALU Arithmetic Logic Unit

SRAM Static Random Access Memory

SREG Status Register

HW Hamming Weight

HD Hamming Distance

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `analysis` .... the directory of preprocessing and analysis source codes
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format