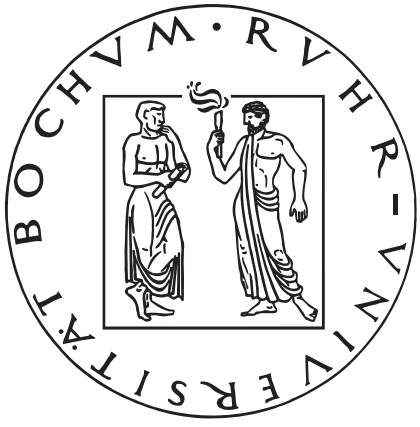
Side-Channel based Reverse Engineering for Microcontrollers

Martin Goldack

January 9, 2008

Diplomarbeit
Ruhr-University Bochum



Chair for Communication Security
Prof. Dr.-Ing. Christof Paar
Co-Advised by Thomas Eisenbarth

Statement

I hereby declare that the work presented in this thesis is my own work and that to the best of my knowledge it is original, except where indicated by references to other authors.

Hiermit versichere ich, dass ich meine Diplomarbeit eigenständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Ort, Datum

Unterschrift

Abstract

Since side-channel analysis was introduced in the mid-1990s, it has permanently been enhanced and become a reliable method for cryptanalysts to break physical implementations of cryptographic algorithms. Recently, these methods have become of interest to be used for reverse engineering program code running on microcontrollers (e.g., [QS02], [No03]), which are often used in security critical environments such as the financial sector.

Until now, statistical methods using a huge number of side-channel observations are often used for this purpose. However, in some scenarios such an approach is not feasible, for example, when the target device is only available for a short period of time.

Hence, in this work it is examined in how far the analysis of single observations of a side-channel can be utilized in order to gather information of a program. For that, a commercially available microcontroller is used as an exemplary target platform. Furthermore, templates as introduced by Chari et. al in [CRR02], specially suited to get the most information out of single traces, are applied as a side-channel analysis method.

As a result, we present a power model for the PIC which provides a basis for a wire range of applications, as for example an improved DPA on this device. Moreover, in conjunction with templates we show that reverse engineering of secret parts of a cryptographic algorithm or program path detection of known code is feasible. In this thesis, the latter one is analyzed in more detail and formulated as an algorithm which can, for example, be used for debugging in the field, version checking, or it can be helpful for gaining a basic design level understanding of a program.

Contents

List of Figures

List of Tables

Nomenclature

$Cov(x)$	covariance of X
d	known data
$E(X)$	expected value of X
h	estimator for the power consumption of a intermedate value
$HD(v_1, v_2)$	Hamming distance between v_1 and v_2
$HW(v)$	Hamming weight of v
k	key
P_{bus}	The factor of the power model weighing the influences of the data bus
$P_{const,Qx}$	The constant fraction of the power consumption concerning Qx
P_{data}	The data dependent fraction of the power consumption
$P_{ext,Qx}$	The additional fraction of the power consumption at Qx caused by unknown influences
P_{fetch}	The fraction of the power consumption caused by the fetching process
P_{inst}	The fraction of the power consumption caused by the opcode of the current instruction
P_{opcode}	The factor of the power model weighing the influences of the current or subsequent opcode
$P_{prog.bus}$	The factor of the power model weighing the influences of the instruction bus
P_{Qx}	The estimated power consumption at Qx
P_{sw}	The switching noise fraction of the power consumption

Qx ..... clock cycle x of the instruction cycle

SNR ..... Signal-to-Noise Ratio

v ..... an intermediate value

V_{dd} ..... power supply voltage

$v_{opcode+1}$ ..... The opcode of the subsequent instruction

v_{opcode} ..... The opcode of the current instruction

$v_{operand}$ ..... The operand provided by an instruction

v_{result} ..... The result calculated by the ALU

$Var(x)$ ..... Variance of X

ALU ..... Arithmetic Logical Unit

CISC ..... Complex Instruction Set Computer

CMOS ..... Complementary Metal Oxide Semiconductor

DES ..... Data Encryption Standard

DPA ..... Differential Power Analysis

EEPROM ..... Electrically Erasable Programmable Read Only Memory

EUSART ..... Enhanced Universal Synchronous/Asynchronous Receiver and Transmitter

FSR ..... File Select Register

GND ..... ground

IC ..... Integrated Circuit

INV ..... inversion ($\overline{A}$)

IR ..... Instruction Register

ISP ..... In-System Programming

LSB ..... Least Significant Bit

MSB ..... Most Significant Bit

NAND ..... inversion of logical AND ($\overline{AB}$)

NMOS ..... N-channel Metal Oxide Semiconductor

NOR ..... inversion of logical OR ($\overline{A+B}$)

NVM ..... Non-Volatile Memory

PC ..... Program Counter

PMOS ..... P-channel Metal Oxide Semiconductor

RAM ..... Random Access Memory

RISC ..... Reduced Instruction Set Computer

ROM ..... Read Only Memory

RSA ..... Rivest Shamir Adleman

SPA ..... Simple Power Analysis

UART ..... Universal Asynchronous Receiver/Transmitter

W ..... working register

1 Introduction

Up to the mid-1990s, cryptographic devices were considered to be black boxes that, given some input (plaintext), produced an output (ciphertext) by means of a secret key stored in the device. Thus, attacks were based on known plaintexts, ciphertexts or plaintext/ciphertext pairs. No further information appeared to be available.

Today, it is known that this is not true. In physical implementations, there are always additional outputs that leak information, the so-called side-channel information, that can be passively observed and then exploited by cryptanalysts to break a cryptographic system, as long as an attacker has physical access to the computing device. This is referred to as side-channel analysis or side-channel attack. Execution time, power consumption or electromagnetic radiation are typical examples of the side-channels used.

The first attack of this kind was presented to the scientific world by Paul C. Kocher in [Ko96]. In this paper a method is described which uses timing information in order to reveal the secret key of an RSA implementation. Since this groundbreaking publication, many new analysis techniques have been presented utilizing visual analysis, statistics, or destined mathematical models. Representatives for these techniques are for example Simple Power Analysis, Differential Power Analysis, or Template Attacks [KJJ98], [KJJ99], [CRR02].

Above all, microcontrollers, often in the form of smart cards, became of special interest for applying these analysis methods, since they are often used as a platform for cryptographic primitives in security critical environments like, for example, bank cards, pay-tv, health cards or machine readable travel documents (MRTD).

Recently, these methods have additionally become of interest to be used for reverse engineering purposes (e.g. [QS02], [No03], [Cl04], and [Ve06]). In this scheme it is not the intention to reveal a secret key of a particular device but to retrieve secret program code, i.e. the intellectual property of a programmer, in order to duplicate the program similar to the reverse engineering of hardware, or to gain a basic design level understanding of an implemented program making a porting to another platform possible [CC90]. Furthermore, the term reverse engineering is frequently used to describe the disclosure of secret algorithms.

In this thesis, we want to examine what can be achieved in terms of reverse engineering by means of merely analyzing single observations of a side-channel, which

has been sparsely discussed so far. Yet, statistical methods utilizing a huge amount of observations are applied more frequently for reverse engineering. However, in some applications, as for example the recognition of a program path that was taken during a certain program execution, only single observation may be taken into account. Thus, following this approach is reasonable and worth the effort to be examined.

Hence, for the analysis, a commercially available mircocontroller, namely the PIC16F687 manufactured by Microchip, is used as a working example. Since only single observations are to be analyzed, templates as introduced by Chari et al. in [CRR02], specially suited for this purpose, will be utilized. In addition, the side-channel used in this thesis is the power consumption of the device, since this is a well known leakage source which can be easily observed.

The remaining chapters of this thesis are organized as follows:

The subsequent chapter covers the basics needed for the understanding of this work. The causes for side-channel leakage are presented as well as the major side-channel analysis techniques developed during the last years, i.e. Simple Power Analysis, Template Attacks, and Differentiel Power Analysis based on difference of means and correlation coefficients. Further, a brief overview of microcontrollers and the already performed work on side-channel based reverse engineering will be given.

In Chapter 3, the target platform is examined from a theoretical point of view. This comprises the design, i.e. components and their interaction, and further instruction cycle and instruction set.

Assumptions concerning the power consumption properties of the device can be drawn up and analyzed in detail by means of Simple Power Analysis in Chapter 4. As a result a power model for the PIC16F687 is presented.

Due to this knowledge, several template creation techniques for the sake of instruction recognition can be concluded and tested in order to discover a suitable approach with respect to the underlying hardware in Chapter 5. Based on the performed tests, some applications for templates in terms of reverse engineering are presented.

Finally, in chapter 6, an algorithm is presented, which can be used to detect program paths from single power consumption traces if the program code is known. Furthermore, its functionality is proven before this thesis is concluded and an outlook is given in the last two chapters.

2 Background

Chapter 2 lays the basis for this thesis. At first, the causes for side-channel leakage in the power consumption of CMOS devices are discussed. Afterwards, common power analysis techniques will be presented in order to get a basic overview of these techniques and further to select promising methods for reverse engineering. We will exclusively focus on the power consumption because it is the most frequently used side-channel and the one we will exploit in this work. However, the techniques discussed in Section 2.1.3 can be applied to other side-channels. Subsequently, some basic design principles for microcontrollers are presented. Although side-channel analysis was invented to break cryptographic systems, some work has already been done to exploit side-channel analysis for reverse engineering purposes on microcontrollers. Section 2.3 summarizes this work.

2.1 Power Consumption

Digital circuits like microcontrollers are constructed of logic cells which carry out either fundamental boolean functions (e.g. NAND, NOR, INV) or storage functions (e.g. Latch, Flip-Flop). The techniques used for implementing logic cells is called logic style. The major one for building digital integrated circuits is the Complementary Metal-Oxide-Semiconductor logic due to its low-power consumption, large noise margins and ease in design [KL03]. Actually, almost all digital circuits are created as CMOS device. It should be obvious that the power consumption highly depends on the logic style used. Because of this, the following section will explain CMOS technology to a level needed for understanding power analysis methods. Additionally, a brief overview of power consumption measurement will be given.

2.1.1 CMOS Technology

CMOS logic cells consist of NMOS and PMOS transistors, which are arranged in a complementary structure. PMOS transistors are used to create a so-called Pull-Up network, which is connected to V_{dd} (logical '1'), whereas the NMOS transistors build up a Pull-down network, connected to GND ('logical 0'). Both networks share cell in- and output. The number of transistors needed to carry out a certain function

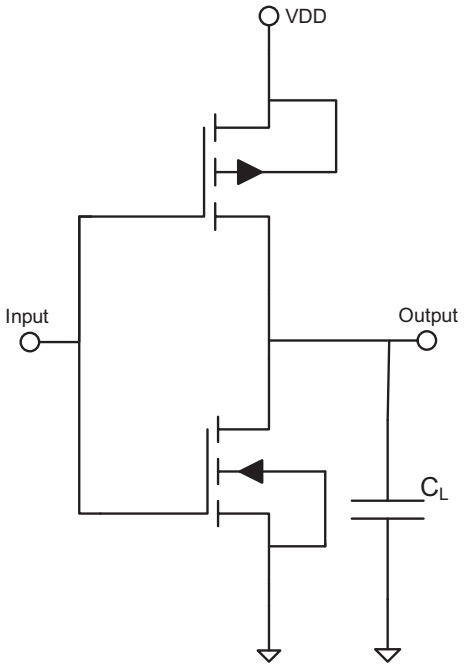


Figure 2.1: CMOS inverter

increases with its complexity. However, the number of transistors in each network is always the same due to the complementary structure.

To discuss the power consumption characteristics, we will now focus on the most simple logic cell, the inverter, which is illustrated in Figure 2.1. As can be seen, in this case, Pull-up and Pull-down network solely consist of one transistor each. When there is no switching activity, i.e. the input is constantly logical '0' or '1', then there is no direct connection between V_{dd} and GND. Either the PMOS transistor is conductive and the NMOS transistor insulating or vice versa. Nonetheless, a small leakage current I_{leak} flows depending on the physical parameters of the transistors. This causes a power consumption, which is referred to as the static power consumption P_s. It can be calculated as

$$P_s = I_{leak} \cdot V_{dd} \tag{2.1}$$

When the input causes a cell to change its output, then the load capacitance, which comprises the internal capacitances connected to the output as well as the external capacitances, is either charged or discharged by a charging current. For example, in the case of the inverter, C_L is discharged over the NMOS transistor to GND on an input transition from '0' to '1' and charged by V_{dd} over the PMOS transistor on a transition from '1' to '0'. Thus, we get an additional power consumption, the so-called dynamic power consumption P_d. It can be computed as

$$P_d = C_L V_{dd}^2 f_p \tag{2.2}$$

in which f_p is the switching frequency of the cell. Since C_L and V_{dd} are given, we can conclude that a cell build in CMOS technology consumes the more power the higher the switching frequency f_p, i.e. the more often C_L has to be reloaded in a given time. Note that this frequency is determined by the input or more precisely by the input sequence of the cell. Hence, P_d is data dependent.

The total power consumption P_{total} is then given by

$$P_{total} = P_s + P_d \tag{2.3}$$

in which the dynamic power consumption is usually the dominant component. However, in modern process technologies with structure sizes smaller than $100nm$, the leakage current is increased significantly, thus increasing the static power consumption as well. Hence, the gap between both components becomes smaller.

More information about CMOS design can for example be found in [KL03] and [TK98].

2.1.2 Measure Requirements

In order to measure the power consumption of an IC, a shunt resistor has to be inserted in-between the GND contact of the chip and GND of the power supply as shown in Fig. 2.2. Alternetively, it can be implemented in-between V_{dd} and V_S.

The voltage drop at this resistor can then be measured by means of a digital oscilloscope. Due to Ohm's law, the measured voltage is proportional to the power consumption. The following equation shows how the power for sample k of a measured trace can be calculated from the resistor voltage $V_R(k)$, resistor R and supply voltage V_S:

$$P(k) = V_S \cdot I(k) = \frac{V_S \cdot V_R(k)}{R} \tag{2.4}$$

Yet, the power consumption is usually not calculated for power analysis. Instead the voltage values are used directly. This is possible because V_s and R are constant values so that in consequence the P_k is equal to the measured voltage $V_R(k)$ except for a constant c.

$$P(k) = c \cdot V_R(k) \tag{2.5}$$

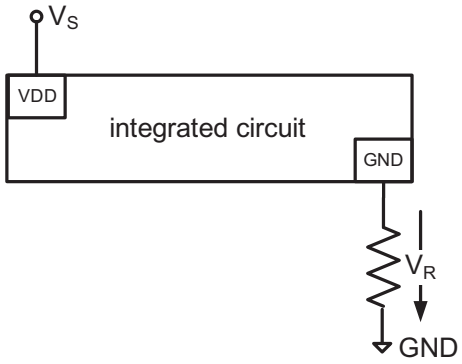


Figure 2.2: Measuring the power consumption

2.1.3 Noise

When the power consumption of a deterministic device is measured with constant inputs during a constant operation, power consumption traces should equal each other. This is because the device performs exactly the same transitions with every execution therefore resulting in a deterministic power trace. However, there is always a certain amount of electronic noise, i.e. random deviations from the intrinsic signal, present in each trace that can not be avoided completely. As a result the measured power trace P_{meas} can be described as the sum of the power consumption of the device P_{sig} and noise $P_{el.noise}$.

$$P_{meas} = P_{sig} + P_{el.noise} \tag{2.6}$$

There are various causes for noise, including power supply, USB/RS232 interfaces that are connected to the device, radiated emissions, temperature, or quantization noise caused by analog/digital conversion of the oscilloscope [MOP07].

The so-called Signal-to-Noise ratio is commonly used in signal theory to express the ratio between signal, the component that contains information, and noise, the component that contains no information at all:

$$SNR = \frac{P_{sig}}{P_{el.noise}} \tag{2.7}$$

Thus, the higher the Signal-to-Noise ratio, the lower the fraction of unwanted noise and the easier it is to extract information out of the recorded power traces. Hence, the aim is to design a measurement setup that suppresses noise as much as possible.

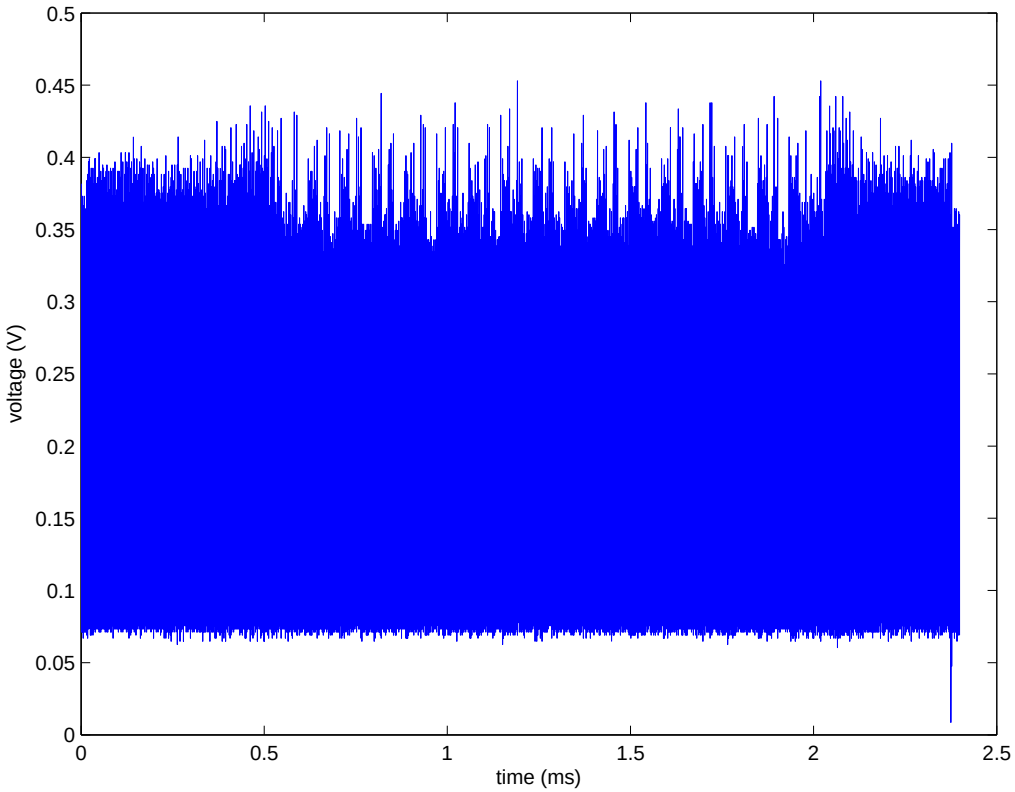


Figure 2.3: Single power trace of a DES implementation on a ATMega163 smart card, DES rounds are clearly visible

2.2 Power Analysis

Now that the power consumption of CMOS devices has been discussed we can go over to the methods which use power consumption characteristics to break a physical implementation of a cryptographic algorithm.

2.2.1 Simple Power Analysis

The Simple Power Analysis was first introduced in the groundbreaking paper of Kocher et al. in [KJJ99]. Basically, SPA is a technique that directly interprets power consumption traces that are measured while a cryptographic algorithm is performed. As shown in Section 2.1, the power consumption highly depends on the switching behavior of the cells. Hence, the instructions that are executed on the device and especially the intermediate values that are produced during the encryption process determine the shape of the power consumption trace. Large features, like encryption rounds, can therefore be identified by inspecting plots of power consumption traces.

Fig. 2.3 exemplarily shows the power consumption trace of a DES[1] encryption that was implemented on a ATMega163 smart card. As can been seen, the sixteen DES rounds can be clearly identified.

Additionally, it is possible to identify even small differences by inspecting a visual representation of just a small part of the trace. An attackers intention is to find such differences. More precisely, he searches for key dependent varieties in order to reveal the secret key bit by bit. Messerges et al. for example showed that the complexity of a brute-force attack on DES can be reduced from 2^{56} to approximately 2^{38} by observing a certain peak heights during the PC1 permutation [MDS99]. Further attacks are based on conditional branches, which, for instance, occur in register rotations or permutation tables [KJJ99]. Some attacks on other cryptographic algorithms like IDEA or RC5 can be found in [KSWH98]. Further, in [JQ01] an attack on elliptic curve cryptography is provided.

As one can imagine, detailed knowledge about the implementation of the algorithm is usually required to mount an attack. However, if only single or just a small number of power traces can be recorded, e.g. when the device is only available for a short period of time, SPA is quite useful in practice.

On the other hand, SPA is a great tool for reverse engineering, since it can be used to analyze the implications on the power consumption for different code sequences or values.

2.2.2 Differential Power Analysis

Differential Power Analysis, also published in [KJJ99], is the most popular power analysis method. Compared to SPA, it has some important advantages. First, this type of analysis does not require any implementation details. In fact, it is sufficient to know the implemented algorithm. Due to Kerckhoffs' principle[2] this can generally be taken for granted. Second, DPA can even be used in extremely noisy environments. On the other hand, the need for a large number of power traces has to be mentioned. This is because traces are not analyzed visually but by means of statistics.

The basic idea is the following. At first, the attacker chooses an intermediate result v (1 bit) of the algorithm which can be calculated from some known data d and a part of the key k (unknown).

$$v = f(d, k), v \in \{0, 1\} \tag{2.8}$$

The function $f(d, k)$ is called selection or partitioning function. Subsequently, a large number of power traces is recorded of encryptions performed with random

[1] Data Encryption Standard, a detailed description of the algorithm can be found in [MOV01]
[2] Kerckhoffs' principle: the security of a system should solely rely on the key [S99]

plaintexts, which are additionally stored together with the traces. Now, the attacker can calculate the results of $f(d, k)$ for each stored trace and each key hypothesis. For each key hypothesis, the measurements are divided into two parts based on the results of the selection function, either 0 or 1. The mean values of the traces for both partitions are then calculated and the difference is built. If the key hypothesis was correct, the resulting difference trace shows significant peaks exactly at positions where the chosen intermediate result was processed. The correct key corresponds to the difference trace with the highest peaks.

The reasons for this are quite simple. The power consumption for processing the intermediate result $v = 0$ and $v = 1$ varies in parts in which v is processed. However, due to noise, this rather small difference is not visible in the difference of just two traces. As a result, a large number has to be recorded in order to later reduce noise by computing mean traces. The selection function helps to determine whether a trace belongs to $v = 0$ or $v = 1$ for a certain key guess and the difference of the mean traces of both partitions indicates the correlation to this key guess. For a better understanding, we first assume that the key hypothesis was correct. Thus, $f(d, k)$ always yields the correct result so that in the end the set of traces is partitioned one-hundred percent correctly. In this case, the noise reduction for the mean traces is maximized so that dissimilarities according to the value of v are equally maximized. If the key hypothesis is not correct, both partitions contain a certain amount of incorrect traces. As a result, the difference trace either shows an almost flat line if the key hypothesis was completely uncorrelated to the traces or some rather small peaks in cases in which the key guess is somehow correlated to the correct subkey.

As a concrete example, we will again consider a software implementation of the DES algorithm. As a first step, the intermediate value v has to be specified. We choose v as the first output bit of the first S-Box that is calculated in round one. The selection function is then given by

$$v = S_1((E(IP(d)_{32-63})_{0-5} \oplus k_{0-5}))_1 \tag{2.9}$$

where d is the plaintext and k_{0-5} represent 6 bits of an unknown subkey. Note, that this is only one of many possible selection functions. We then record traces of encryptions with random plaintexts. Subsequently, for each of the $2^6 = 64$ key hypotheses the traces are partitioned and averaged in order to create a difference trace. All sixty-four differences can then be compared to find the correct key. Fig. 2.4 exemplarily shows two differences of a DPA performed with five hundred measurements. In contrast to the upper trace, in which no peaks are visible, the bottom curve clearly shows two peaks indicating the correct key. At this point of the attack, only six subkey bits are known. However, the attack can be repeated with other selection functions that match the other S-Boxes in order to reveal the remaining

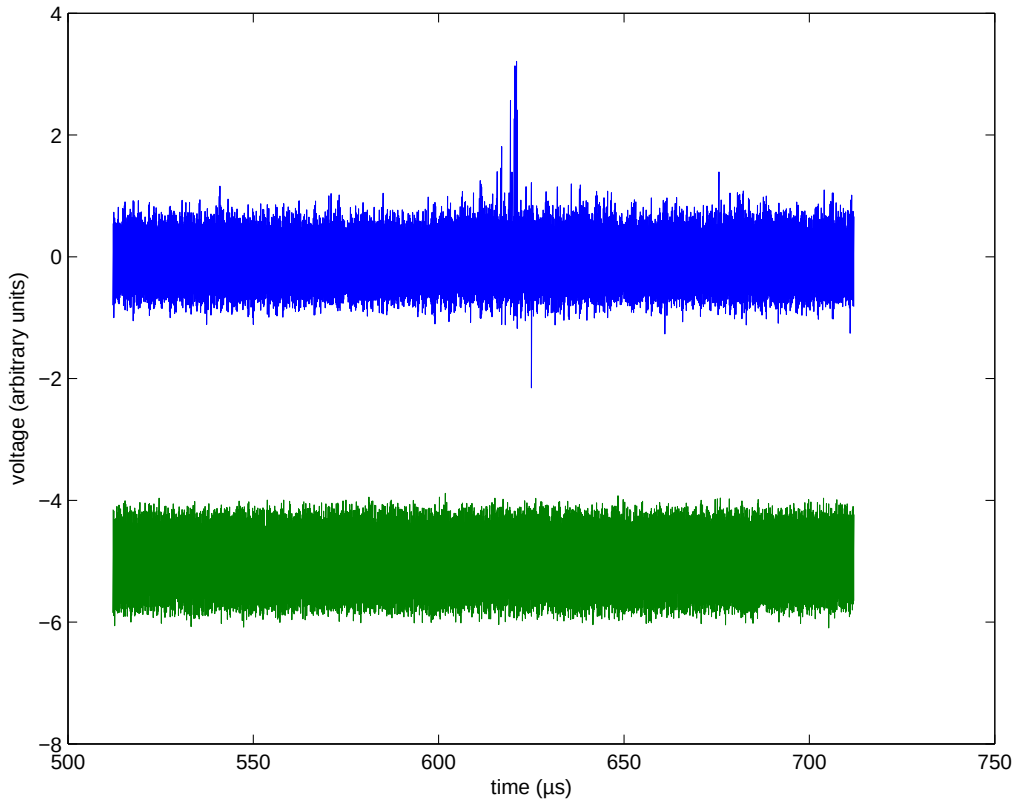


Figure 2.4: Two difference traces of a DPA on a DES implementation (ATMega163 smart card), wrong (top) and correct key hypothesis (bottom)

subkey bits for round one. If this is done, the key scheduling can be recomputed to achieve the secret 56 key bits.

Kocher additionally noted that the combination of more than one intermediate value within the selection function can yield better results, especially if countermeasures are active to prevent basic DPA attacks (see Section 2.2.4). This type of power analysis is referred to as higher-order DPA.

2.2.3 Correlation Coefficient and Power Models

A way to improve the DPA by Kocher is to use correlation coefficients instead of difference of means to estimate the correct key. This method was first published by Brier et al. in [BCO04] and runs as follows. At first, intermediate values are calculated as with the DPA by Kocher. Subsequently, these values are mapped to hypothetical power consumptions based on a destined power model. The correct key can then be found by correlating the recorded power consumption traces to the

hypothetical power consumptions. To understand this attack in detail, we first have to discuss power models.

As already mentioned, an attacker tries to simulate the power consumption of intermediate values. However, his knowledge about the hardware is generally limited. As a result only simplistic power models can be used for this purpose. One basic model is the *Hamming-Weight model*. In this scheme it is assumed that the power consumption is proportional to the number of bits set in a chosen intermediate value and can thus be estimated by this number.

Definition 2.2.1 (Hamming-Weight Model) *In the Hamming-Weight model it is assumed that the power consumption of a device is directly or inversely proportional to the Hamming-Weight of intermediate value v. Hence, the power consumption can be estimated with*

$$h = HW(v) \tag{2.10}$$

Another model is the *Hamming-Distance model* which takes into account the number of altering bits. Applying this model can become more complicated in practice because in addition to the current value the previous one needs to be known. However, this model is generally more accurate for many devices.

Definition 2.2.2 (Hamming-Distance Model) *In the Hamming-Distance model it is assumed that the power consumption of a device is directly or inversely proportional to the Hamming-Distance of two intermediate values v_1 and v_2. Hence, the power consumption can be estimated as*

$$h = HD(v_1, v_2) = HW(v_1 \oplus v_2) \tag{2.11}$$

On the basis of one of these models, the hypothetical power consumption for each intermediate value can be calculated. In a concrete attack in which D traces were recorded and K keys are possible, we can write the hypothetical power consumptions for each value to a $D \times K$ matrix $\mathbf{H}$ in which each column corresponds to a certain key hypothesis and each row to a certain plaintext.

By now, two conclusions can be drawn. First, both models are appropriate for intermediate values that are not binary. This is one of the main advantages a DPA based on correlation coefficient has over the difference of means approach in which only binary models, i.e. $v \in \{0, 1\}$ are feasible. Second, the values that occur in both models are inaccurate concerning the values of the real power consumption. To understand why this is not a problem, we will now have a look at the correlation coefficient.

The correlation coefficient is a statistical method to measure linear relationships between two random variables X and Y and is defined as follows:

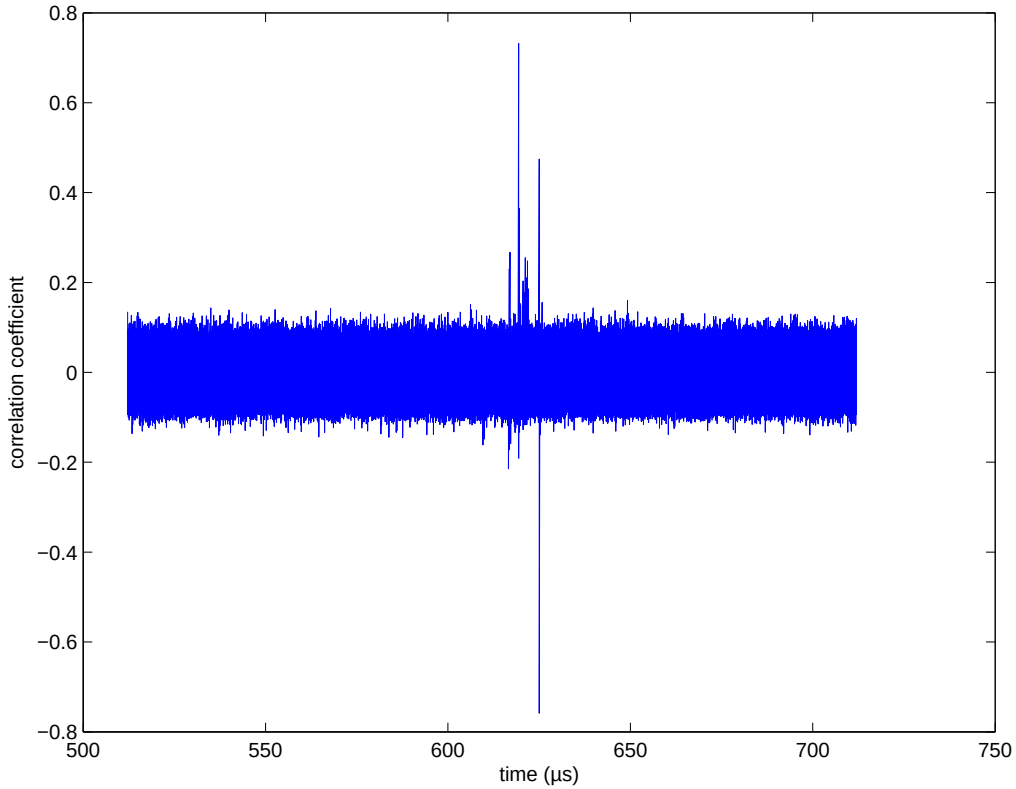


Figure 2.5: Correlation coefficients for a DPA on DES with the correct key hypothesis, the peaks toward 0.8 and −0.8 indicate a high correlation

$$\rho(X,Y) = \frac{Cov(X,Y)}{\sqrt{Var(X) \cdot Var(Y)}} \tag{2.12}$$

In this equation, the so-called covariance $Cov(X,Y)$, given in (2.13), is a measure for statistical dispersion concerning X and Y. In the case that both variables are independent it holds that the expected value $E(XY)$ is equal to $E(X) \cdot E(Y)$ and thus $Cov(X,Y) = 0$. If only one variable is considered we can rewrite (2.13) as (2.14). This is referred to as the variance of X and represents a measure for the squared average deviation from the mean.

$$Cov(X,Y) = E((X - E(X)) \cdot E(Y - E(Y))) = E(X \cdot Y) - E(X) \cdot E(Y) \tag{2.13}$$

$$Var(X) = E((X - E(X))^2) \tag{2.14}$$

Since the expected values of X and Y are usually not known, they have to be estimated by the arithmetic mean. The estimators $\sigma^2_{x,y}$ for the covariance and σ^2_x for the variance are then given by

$$\sigma^2_{x,y} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}) \tag{2.15}$$

$$\sigma^2_x = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 \tag{2.16}$$

in which n is the number of samples and $\overline{x}$, $\overline{y}$ are the mean values. With (2.15) we can estimate the correlation coefficient as

$$r = \frac{\sum_{i=1}^{n} x_i\, y_i - n\,\overline{x}\,\overline{y}}{\sqrt{(\sum_{i=1}^{n} x_i^2 - n\,\overline{x}^2)(\sum_{i=1}^{n} y_i^2 - n\,\overline{y}^2)}} \tag{2.17}$$

in which r is dimenionless and normalized ($-1 \leq r \leq 1$) [Pa01]. A value of 1 indicates that the two sets are perfectly correlated and a value of -1 that both are perfectly anti-correlated. In case of $r = 0$ exists no linear relationship.

Hence, it follows that the values for the hypothetical power consumption do not have to be equal to the real power consumption. If there is a proportionality, then in consequence there is a linear relationship and thus r will be significantly high in these cases.

Assuming that we have recorded a set of D traces with T samples each, we can write the traces to a $D \times T$ matrix $\mathbf{T}$ and correlate each column h_i of $\mathbf{H}$ which contains the estimated power consumption values for a certain key to each column t_j which contains all trace samples for a certain known value d with:

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i}\, t_{d,j}) - n\,\overline{h_i}\,\overline{t_j}}{\sqrt{(\sum_{d=1}^{D} h_{d,i}^2 - n\,\overline{h_i}^2)(\sum_{d=1}^{D} t_{d,j}^2 - n\,\overline{t_j}^2)}} \tag{2.18}$$

If the key hypothesis i was correct, the correlation coefficient $r_{i,j}$ will then show significant peaks at positions j where the intermediate value is processed.

For an exemplary DPA on DES with the Hamming-Weight model as the model of choice, we can for example estimate the power consumption as

$$h_{i,j} = HW(S_1((E(IP(d)_{32-63})_{0-5} \oplus k_{0-5}))) \tag{2.19}$$

for each possible key k and plaintext d to correlate these values to our power consumption traces. Fig. 2.5 shows the results for a correlation DPA on DES on an ATMega163 smart card with 200 measurements and a correct key hypothesis. As can be seen, the trace shows significant peaks indicating a high correlation of about 0.75. By comparing this figure to Fig. 2.3 we can see that the correlation occurs at the time of the first DES round, i.e. exactly where the intermediate value is processed.

Please note, that a DPA by means of correlation coefficients is not a higher-order DPA. As with the difference of means approach, just one intermediate value is exploited, although the number of bits of this value was increased from one to four bits.

2.2.4 Template Attacks

Template attacks were introduced by Chari et al. in [CRR02] as a new type of attack that, similar to SPA, aims to extract information out of single traces. Because this form of side-channel analysis will be intensively used in this thesis, it will be discussed in higher detail.

Basically, the attack is performed in two phases: a *template building phase* and a *template matching phase*. In the *template building phase*, the attacker builds up power consumption models, i.e. templates, for certain instructions or instruction sequences. To be able to do this, it is assumed that the attacker has access to an experimental device that he can program to his choosing and that is identical to the device under attack. In the *template matching phase*, templates built in the previous phase are applied to traces of the device under attack in order to distinguish which template matches best. With this knowledge, information about the key can then be derived.

Template building phase The first question that occurs is how the power consumption can be modeled. As we have seen in Section 2.1.2, the measured power consumption is the sum of the intrinsic signal P_{sig} and electronical noise $P_{el.noise}$. As a result, both parts have to be taken into account in order to get an adequate model. For this purpose, Chari et al. proposed a probability distribution called the Multivariate-Gaussian model, which is an extension of the (univariate) gaussian / normal distribution to a higher dimension.

If a random variable X is normal-distributed, this means that X is symetrically distributed around a maximum and that the higher the deviation of a sample x from this maximum, the lower its probability. This kind of distribution is completely defined by its expected value $E(x)$ and its standard deviation $\sqrt{Var(X)}$. As discussed in Section 2.2.2, a good estimation for $E(X)$ is the arithmetic mean $\overline{x}$. Furthermore,

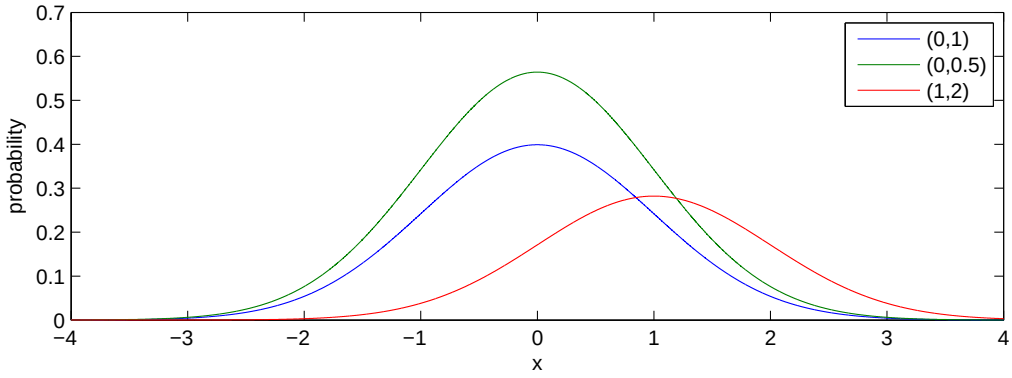


Figure 2.6: Three plots of normal distirbutions with different values $(\overline{x}, \sigma_x)$

the standard deviation can be estimated by the square root of σ_x^2 which has been defined in (2.16). The probability density function for a normal distributed random value is then given by

$$f(x) = \frac{1}{\sqrt{2\pi\sigma_x}} \exp\left(-\frac{1}{2}\left(\frac{x-\overline{x}}{\sigma_x}\right)^2\right) \tag{2.20}$$

in which $f(x)$ represents the probability to observe value x. As an example, some plots of normal-distributed random values with different values $(\overline{x}, \sigma_x)$ are illustrated in Fig. 2.6.

Measured variables are often normal-distributed. This usually also holds for the noise component of power consumption measurments which is distributed around P_{sig}. Hence, we can estimate P_{sig} as $\overline{x}$ and the noise for each point of a trace with the standard deviation σ_x. However, the power model would be more accurate if the relationship between different points was taken into account. For this purpose, the model can be extended to the Multivariate-Gaussian model.

In this scheme, the distribution of n points is described by a mean vector $\mathbf{m}$ and a covariance matrix $\mathbf{C}$. The probability density funcion is given as

$$y(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n \cdot det(\mathbf{C})}} \cdot \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{m})^T \cdot \mathbf{C}^{-1} \cdot (\mathbf{x}-\mathbf{m})\right) \tag{2.21}$$

in which the resulting $y(\mathbf{x})$ is a scalar representing the probability to observe a vector $\mathbf{x}$. The elements $c_{i,j}$ of $\mathbf{C}$ represent the covariance between point i and j and are thus a measure of their correlation. The covariance matrix has some important properties. First, since $COV(X,Y) = COV(Y,X)$ it follows that $\mathbf{C}$ is a symmetric matrix. Second, the main diagonal contains the variances of the points. This is because $Cov(X,X) = Var(X)$. Finally, the determinant is a positive value. This

guarantees that the square root in (2.21) can be calculated. The mean vector $\mathbf{m}$ is a vector of n elements and contains the arithmetic mean for each point.

With the Mulivariate-Gaussian distribution we now have an adequate model for the power consumption. The mean vector $\mathbf{m}$ represents the static component of the power consumption, i.e. P_{sig} and the dynamic component $P_{el.noise}$ is modeled by the covariance matrix $\mathbf{C}$. As a result, we can define templates as follows:

Definition 2.2.3 (Template) *A template $h = (\boldsymbol{m}, \boldsymbol{C})$ build from a set of m traces $\boldsymbol{T}$ with n points*

$$\boldsymbol{T} = \begin{pmatrix} t_{1,1} & t_{1,2} & \dots & t_{1,n} \\ t_{2,1} & t_{2,2} & \dots & t_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ t_{m,1} & t_{m,2} & \dots & t_{m,n} \end{pmatrix} \tag{2.22}$$

consists of a mean vector

$$\boldsymbol{m} = (\overline{t_1}, \overline{t_2}, \dots, \overline{t_n}) \tag{2.23}$$

and a covariance matrix

$$\boldsymbol{C} = \begin{pmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,n} \\ c_{2,1} & c_{2,2} & \dots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \dots & c_{n,n} \end{pmatrix} \tag{2.24}$$

in which $c_{i,j} = \frac{1}{m-1} \sum_{k=1}^{m} (t_{k,i} - \overline{t_i})(t_{k,j} - \overline{t_j})$.

With Definition 2.2.4, we are now able to build multiple templates h_i covering different cases of power consumption characteristics. For instance, to identify which Hamming weight belongs to an intermediate value, we could create several templates, one for each Hamming weight.

Template matching phase In the template matching phase the goal is to find out which template is the most likely for a power trace taken by use of the device under attack. This can be achieved by calculating the result of the probability density function for each template as shown in the following equation.

$$p(\mathbf{x}, h_i) = \frac{1}{\sqrt{(2\pi)^n \cdot det(\mathbf{C}_i)}} \cdot \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{m}_i)^T \cdot \mathbf{C}_i^{-1} \cdot (\mathbf{x} - \mathbf{m}_i)\right) \tag{2.25}$$

The resulting values $p(\mathbf{x}, h_i)$ indicate how well a template fits to an observed vector $\mathbf{x}$. If all templates are equiprobable, then the best choice is to choose the template

with the highest probability, i.e. the highest value. This approach is called the maximum-likelihood decision rule [Pa01].

By now, we have discussed the theory that is needed to apply a template attack. In practice however, problems can occur caused by the covariance matrix. First, the size of the covariance matrix grows quadratically with the number of points. As a result, it is adequate to limit the number of points to only those which are likely to contain information. For this purpose, Chari et. al proposed to calculate pairwise differences of averaged traces for each observation. Consequently, points showing a high difference are the ones containing information and should thus be used for the template building phase. The second problem is that the covariance matrix can be badly conditioned so that the inverse of $\mathbf{C}$, which has to be calculated in (2.25), tends to be close to singular. Consequently, the values calculated in the exponent tend to be very small, which causes additional numerical problems. Basically, there are two ways to counter the mentioned problems.

The first way is to avoid the exponentiation completely. To do this, the logarithm can be applied to (2.25):

$$\ln\left(p(\mathbf{x}, h_i)\right) = -\frac{1}{2}(\ln((2\pi)^n \cdot det(\mathbf{C}_i)) + (\mathbf{x} - \mathbf{m}_i)^T \cdot \mathbf{C}_i^{-1} \cdot (\mathbf{x} - \mathbf{m}_i)) \tag{2.26}$$

In this case, the template h_i that leads to the smallest absolute value has the highest probability.

In a second approach, we can set the covariance matrix to the identity matrix. A template built this way is called *reduced template* [MOP07]. Clearly, this solves the problem of inverting $\mathbf{C}$ but as a drawback infomation about the noise component is lost. For the use of reduced templates, (2.25) and (2.26) can be written as (2.27) and (2.28) respectively.

$$p(\mathbf{x}, h_i) = \frac{1}{\sqrt{(2\pi)^n}} \cdot \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{m}_i)^T(\mathbf{x} - \mathbf{m}_i)\right) \tag{2.27}$$

$$\ln\left(p(\mathbf{x}, h_i)\right) = -\frac{1}{2}(\ln((2\pi)^n) + (\mathbf{x} - \mathbf{m}_i)^T(\mathbf{x} - \mathbf{m}_i)) \tag{2.28}$$

Although template attacks were introduced as way to improve SPA, it can be even used for DPA attacks. This is quite obvious because templates provide a more accurate way of modeling the power consumption than with the methods discussed in Section 2.2.2 in which the power consumption was modeled by simplistic power models. More information about this topic can be found in [ARR03].

Recapitulatory, templates seem to be a promising candidate for reverse engineering because this analysis method aims to get the most information out of single

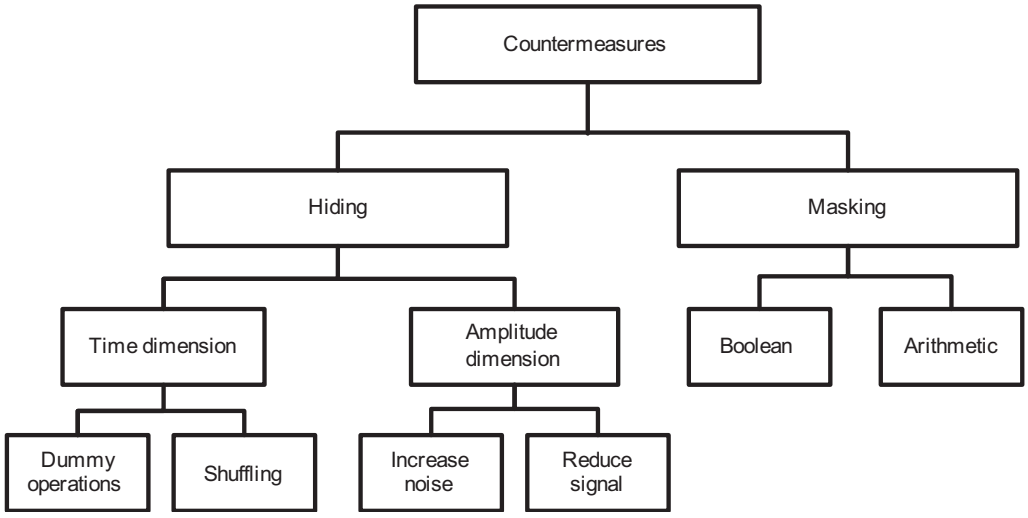


Figure 2.7: Overview of countermeasures to prevent side-channel analysis

traces. Furthermore, it is a statistical method providing a probability value for an observed vector. Hence, traces can be automatically analyzed and do not have to be analyzed visually.

2.2.5 Countermeasures

The described power analysis attacks have in common that they exploit key dependent differences which occur in the power consumption traces. Consequently, the goal of countermeasures is to prevent these key related dependencies. In this section we will briefly discuss various countermeasures that can be taken into consideration.

As shown in Fig. 2.7 countermeasures are divided into hiding and masking schemes. In hiding schemes, the designer tries to hide the power consumption in a way that the measured power consumption is independent of the intermediate values and independent of the performed operations. This means that both dimension, time and amplitude have to be secured.

In the time domain, one way to avoid operation dependent execution times is to insert random dummy operations. Thereby, an attacker can not determine the position of the actual algorithm that is performed. A drawback of this technique is that the throughput is reduced. Another method to hide the time domain is shuffling. In this approach the sequence of executed primitives of the algorithm is randomly changed in every iteration so that it becomes more complex to find the chronologic position of the attacked intermediate value.

For the amplitude dimension, mainly two methods are feasible. The first one is to purposely increase the noise of the device to drastically reduce the SNR. This again significantly increases the number of measurements needed to mount an attack. One way to realize an increase of noise it to implement specific noise generators. The second method of amplitude hiding is the opposite of the first. Instead of increasing noise, the signal is reduced by adjusting the power consumption of the device to a constant level. This can be achieved by filtering or by using a dedicated logic style whose cells which ideally consume a constant power.

An alternative to hiding is masking. This technique does not hide intermediate values but it randomizes these by applying a mask. In this scenario an attacker faces the problem that he can not sort or correlate the traces to the intermediate value of his choice because this value has additionally become a function of a mask m ($v = f(d, k, m)$), which is unknown and changes in every iteration of the algorithm. There are two types of masking: boolean and arithmetic masking. In boolean masking the intermediate value is simply XORed with the mask, whereas in arithmetic masking arithmetic operations like modular addition or multiplication are applied.

Hiding and masking can be implemented in combination to increase the security. However, both techniques are not completely secure. Actually, hiding only hides key related dependencies and does not eliminate these completely. Thus, with a sufficient high amount of traces this technique can be attacked. Masking schemes can equally be attacked by means of higher-order DPA.

More information about this topic can be, for example, found in [MOP07] or [CJRR99].

2.3 Microcontrollers

To get an overview of microcontrollers, this section provides a brief overview about its components. Furthermore, some general design principles are discussed. Specific information of the device that will be analyzed in this work is covered in Chapter 3.

As shown in Fig. 2.8, a microcontroller consists of CPU, Memory and peripheral components which are connected via a bus system.

The CPU represents the main part. Its function is to fetch instructions from memory, interpret and execute them. Basically there are two architectures to implement this as illustrated in Fig. 2.9. The first one is the von-Neumann architecture. In this design, instructions and data are fetched from the same memory over the same bus of width x. Usually several bus accesses are needed to fetch an instruction in this scenario. Further, data may have to be fetched subsequently. As a result, the bus can become extremely congested. To avoid this, the second approach, called Havard

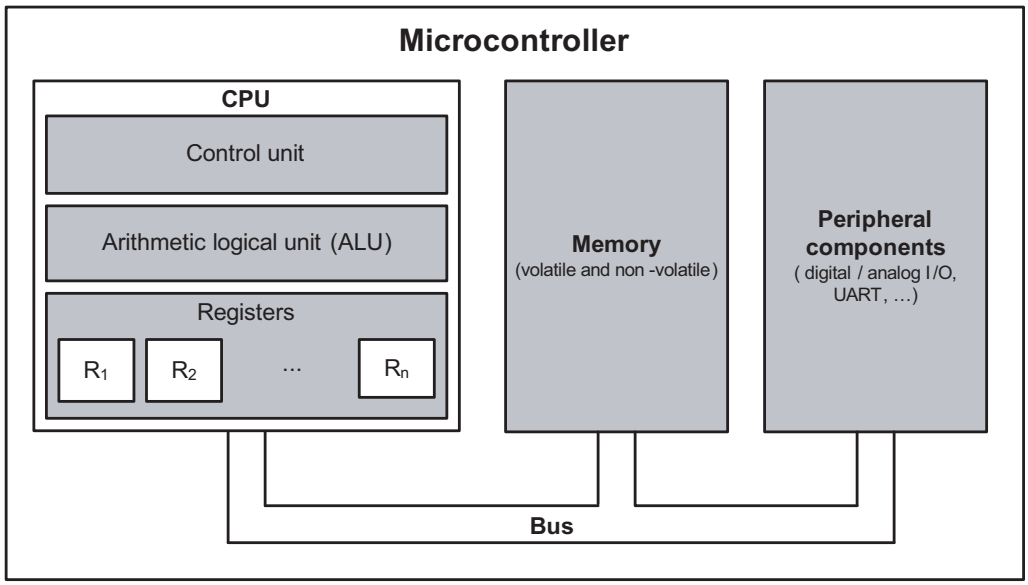


Figure 2.8: Block diagram of a Microcontroller

architecture, separates data and program memory. Hence, both can be accessed independently over separate busses which may have different widths. The instruction bus width is usually designed to match the bit size of an instruction whereas the data bus width is usually smaller, e.g. a processor word of eight bits. Consequently, instructions can be fetched within one clock cycle and data loaded in parallel, which speeds up the CPU.

Another aspect of CPU design is how the instruction set is constructed. In a Reduced Instruction Set Computer (RISC), instructions are designed in a way that allows quick issuing and execution to maximize the number of processed instructions in a certain time. By contrast, a Complex Instruction Set Computer (CISC) implements instructions which can execute several low-level operations, e.g. loading two operands from various sources, combining them, and storing them back to memory. The execution time for such an instruction is significantly higher than for a RISC instruction. Which scheme is faster depends on how fast a RISC instruction can be executed compared to a CISC instruction. For instance, if it takes 4 RISC instructions to perform the same operation as one CISC instruction but the RISC instruction can be executed more than four times faster, RISC wins. This is usually the case so that state-of-the-art microcontrollers are often built in RISC architecture or as an enhanced hybrid design of RISC and CISC.

Irrespective of the architecture, a CPU consists of a control unit, several registers and an arithmetic logical unit (ALU). The control unit is responsible for fetching an instruction and determining its type, which is referred to as decoding. Registers can store a certain amount of bits depending on their type. One important register is

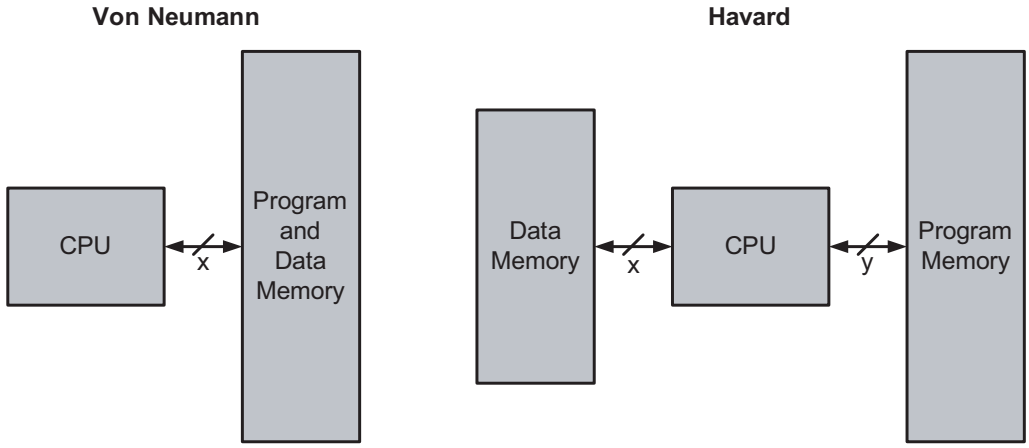


Figure 2.9: Processor architectures - von-Neumann (left) and Havard (right)

the so-called program counter (PC). It points to the next instruction to be executed. Another one is the instruction register (IR) which stores the current instruction. The ALU is the part of the CPU which performs arithmetic and logical operations like addition or shifting. Usually, it has two input registers representing the operands and an output register to store the result. From this register the result can then be transferred to memory or to peripheral components via a bus. The sequence of fetching, decoding, executing and storing one instruction is called instruction cycle and may take more than one clock cycle.

The memory of a microcontroller occurs in various forms. Generally, volatile and non-volatile memory is distinguished. Volatile memory is used to store intermediate values that are calculated during program execution. It is often implemented as static RAM. In contrast to this, Non-Volatile memory (NVM) is used to store data that is supposed to maintain in memory even when the device is switched off. This contains the program data itself but can also contain additional data which according to the previous paragraphs is either stored in the same NVM or in a separate one. In the past, EEPROM was the major memory architecture to store the program. Today, EEPROM is often replaced by Flash ROM for economical reasons.

To communicate to the outside world, the microcontroller uses its peripheral components. Among others, these can be digital ports to activate other devices or to receive commands from them, analog ports to connect sensors, or a Universal Asynchronous Receiver/Transmitter (UART) as serial interface. Another peripheral component is the programming circuit. In early microcontrollers one had to apply a high programming voltage to be able to store the program to memory. Nowadays, the device can often generate its own programming voltage from its native power supply voltage. Furthermore, a microcontroller can usually be programmed in the field. This is referred to as In-System Programming (ISP). Besides these components, a

microcontroller may contain special purpose hardware like USB or LCD driver.

More information about CPU design can for example be found in [Ta90].

2.4 Related Work

The intention of side-channel based reverse engineering for microcontrollers is to retrieve information about the program that is executed on the device. This is not limited to revealing the program stored on it. Further, it is the intention to gain a sufficient design-level understanding of the program or parts of it. This means that the reverse engineer tries to discover how the program performs its task.

Some work on this topic has already been performed. In [Ve06], Vermoen shows how to acquire information about bytecodes executed on a Java smart card. The method used in his work is based on averaging traces of certain bytecodes in order to correlate them to an averaged trace of an unknown sequence of bytecodes. Further, in [QS02] a method is presented that recognizes executed instructions in single traces by means of self-organizing maps, which is a special form of neural network. In [No03] Novak presented a method to recover the values of one out of two substitution tables of the secret A3/A8 algorithm, which is used in GSM communication. However, Novak's attack has two drawbacks. First, the attacker has to know the values of the other substitution table and, second, he has to know the secret key. This approach was improved by Clavier in [Cl04] to an attack retrieving the values of both permutation tables and the key without any prior knowledge.

The approach followed in this work is different, since it is the intention to retrieve information of a program running on a microcontroller by means of single measurements. Under this premise, averaging like in Vermoens approach is not practicable. Furthermore, the use of self-organizing maps seems to be inadequate since the possibilities to re-adjust this approach in case of insufficient results is highly limited. Thus, the instruments used in this work are templates as introduced in [CRR02] and discussed in Section 2.2.3. This side-channel analysis method seems to be notably suited for analyzing single measurements, since it takes the noise component into account and provides for probability for an observation.

3 Device Examination

In the last chapter it was discovered that templates are a promising analysis technique for reverse engineering purposes. However, in order to create adequate templates for our subject or more precisely for the instructions that are executed on it, a basic understanding of the device is needed.

Therefore, we will first have to examine the components of the device, namely the PIC16F687[1] manufactured by Microchip. This is covered in the first section. Afterwards, an overview of the instruction cycle is given, i.e. how instructions are timely fetched and executed, before the instruction set itself will be presented in the last section.

3.1 Components and Interaction

The PIC, whose internals are illustrated in Fig. 3.1, is an **8-bit RISC Microcontroller** designed in **Havard architecture**. According to this, data and program memory are separated, i.e. driven by different busses. The program memory can be accessed via a **14-bit program bus** and is built as a **Flash-ROM** which can store up to 2048 instructions. In contrast to this, the **128 bytes SRAM** and **256 bytes EEPROM** data memories are connected to an **8-bit data bus** which is further connected to the peripheral components (ports, A/D converter, EUSART, etc.). To discuss the interaction of the main components of the device, we will now roughly follow the path of an instruction being processed.

The instruction cycle starts at the instruction register (IR), which contains the instruction to be executed. This register is connected to an Instruction Control and Decode unit, an SRAM address multiplexer, and an ALU multiplexer.

The **Control and Decode unit** controls the instruction cycle, i.e. it determines which operands are sent to the ALU, which file register is addressed, which component stores the result of an operation and so forth. Recapitulatory, all instruction dependent events are controlled by this component.

To be able to modify or read file-registers of the SRAM by use of an instruction, seven bits of IR are connected to an **address multiplexer** as a direct address.

[1] in the following simply referred to as PIC

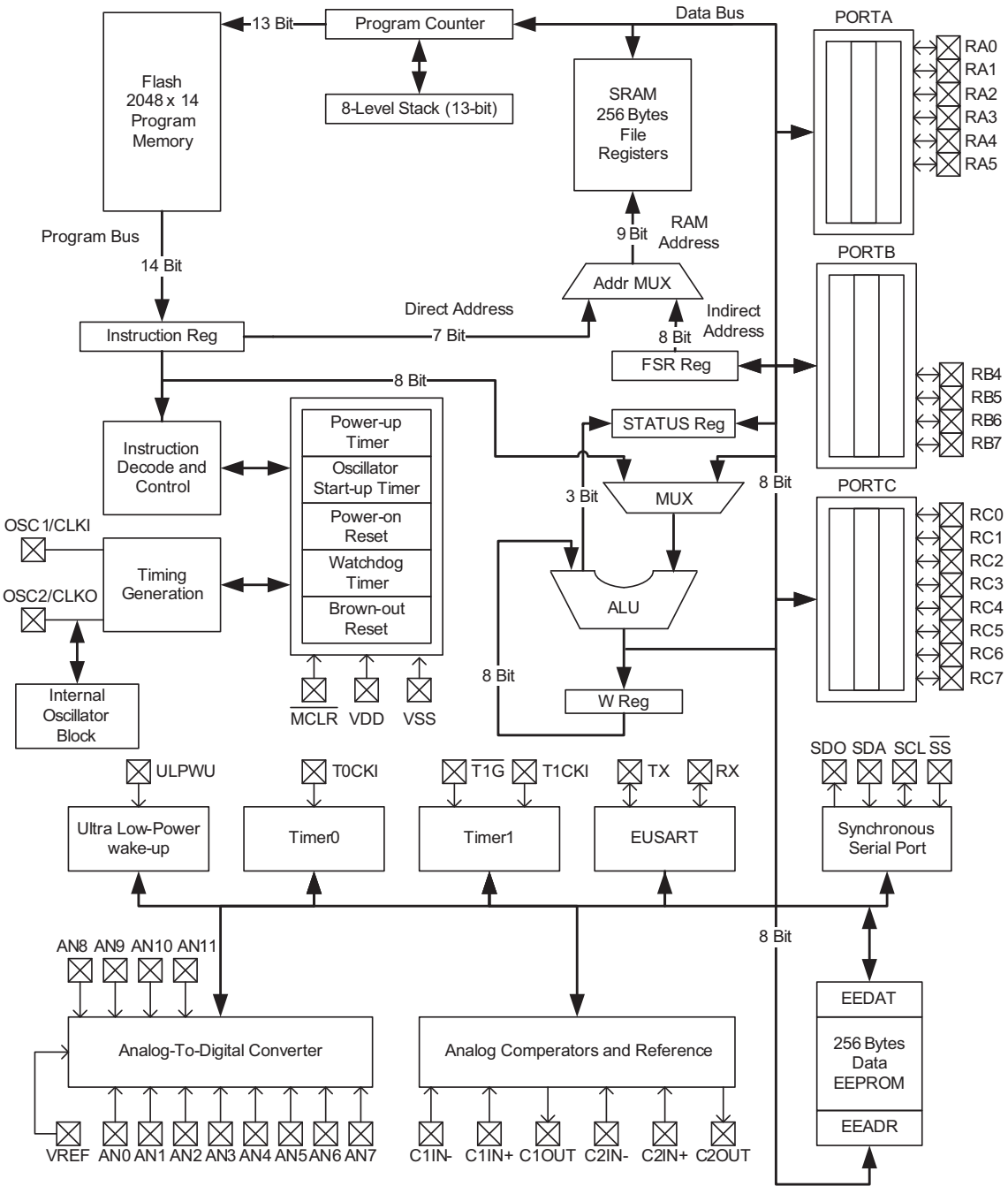


Figure 3.1: Block diagram of the PIC16F687 microcontroller [MC07]

Another way to access the SRAM is to use an indirect address stored in FSR. The register content to which the applied address points to is sent to the data bus later on. Thus, all components connected to it have access to this data.

Moreover, an instruction can contain constants to be processed by the **ALU**. For this reason, eight bits of IR are connected to an **ALU multiplexer** whose second input is the data bus. Which one is sent to the ALU is controlled by the Decode and Control unit. The second operand of the ALU is provided by a **working register** (W). Consequently, if two operands stored in memory or in other components connected to the data bus, are to be processed by an instruction, one of these has to be loaded into W first. After the ALU has combined these operands, the result is sent to the data bus. Additionally, the **status register** may be affected by this operation, e.g. when a carry bit occurs or when the result of an operation is zero.

After the ALU has performed an operation and the result is stored to some component connected to the data bus, the next instruction has to be fetched from the program memory. The address of this instruction is determined by the **program counter** (PC). Usually it is incremented every instruction cycle to point to the next instruction. However, if branches are taken, the PC is modified by the data bus in order to point to the instruction to be branched to. Further, the currently addressed memory location can be saved on an **8-level stack** to be able to return to this instruction later on.

All the discussed actions which are executed by the PIC are controlled by a clock. This can be an internal clock generated by an internal oscillator or an external clock, e.g. a crystal. Depending on the type of the clock, the working range of the device differs. With an external clock the entire clock range (up to 20 MHz) can be utilized, whereas the internal clock is limited to a range from 32 kHz to 8 MHz.

3.2 Instruction Cycle

By now, we have discussed the various parts of the microcontroller and their interaction. However, until now we do not know exactly how instructions are timely processed. This will be discussed in the following.

In the PIC architecture, one instruction cycle consists of four clock cycles, referred to as $Q1$ to $Q4$. In general, each instruction is executed within one instruction cycle. However, some instructions will modify the PC as part of an unconditional or conditional branch. In this case it takes two instruction cycles to perform the operation with the second instruction cycle executed as a NOP (**N**o **OP**eration).

This behavior is caused by a two-stage pipeline which means that, while one instruction is decoded and executed, the next one is fetched simultaneously to be prepared for the next instruction cycle. Consequently, if an instruction modifies

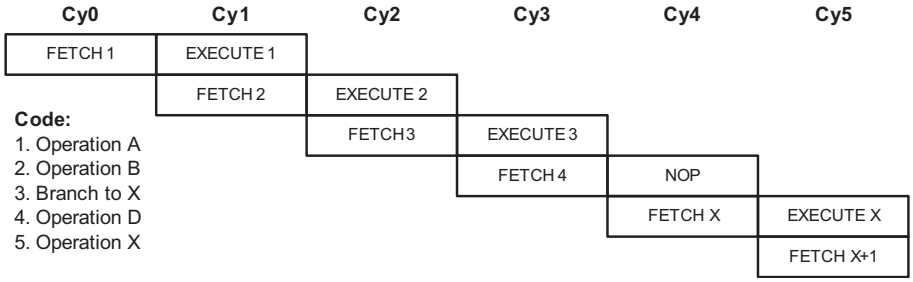


Figure 3.2: Instruction pipeline flow [MC97]

the PC, then the wrong instruction will be fetched during this cycle. Hence, an additional instruction cycle is needed in order to fetch the correct instruction.

An example for the operation mode of the pipeline is illustrated in Fig. 3.2 and shows how a code sequence consisting of five instructions is processed. During *Cy1*, the second instruction is fetched while the first instruction is executed. The same holds for the next instruction cycle. In *Cy3* a branch to instruction five is executed. Unfortunately, instruction four was fetched in this cycle. Thus the next cycle will be executed as NOP to be able to set PC to the address of instruction five.

At this point it should be clear that the pipeline can become problematic for instruction identification because prospectively we will not be able to build templates without having inherent influences of the next instruction.

The architectural documentation [MC97] does not provide detailed information about the operations that are executed during $Q1$ to $Q4$ when fetching the next instruction. The only information we have is that the fetching starts in $Q1$ by incrementing PC and ends in $Q4$ with latching the fetched instruction into IR. Thus, it will be the task of the practical tests in Section 4 to determine which clock cycle is significantly influenced by the fetching process. By now, we can assume that the Hamming weight of the instruction or the Hamming distance of the fetched and executed instruction may influence $Q4$.

Concerning the decoding and execution of an instruction, information is more detailed: The instruction is latched into the instruction register in $Q1$, operands are read in $Q2$, execution then starts in $Q3$, and the result is stored back in $Q4$.

3.3 Instruction Set

The instruction set contains 35 instructions which can be devided into three categories: byte-oriented, bit-oriented, and literal/control operations. According to the program bus width, each instruction is defined by a 14-bit opcode which consists of

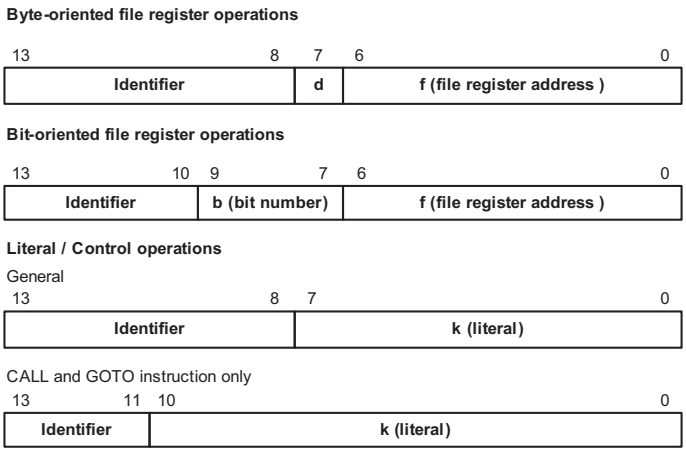


Figure 3.3: General format for instructions [MC07]

an identifier specifying the operation and one or more operands. The instruction formats for each operation type are presented in Fig. 3.3.

For byte-oriented instructions, f is a 7-bit file register address ($00h - 7Fh$). According to Section 3.1, these bits thus represent the direct address that is sent to the address multiplexer. Bit d is a destination designator which specifies where the result of the operation is to be stored. If d is zero, the result is stored in W, if it is one the result is stored in file register f.

For bit-oriented instructions, b is a 3-bit value indicating which bit of file-register f is used to perform the operation. A value of seven corresponds to the most significant bit (MSB) of f and a value of zero to the least significant bit (LSB).

For literal/control instructions, operand k, in general, is an 8-bit literal. It is sent to the ALU in order to be processed with the contents of W. In case of a Call or Goto instruction, i.e. an unconditional branch, it is an 11-bit constant value used to modify PC. Since the data bus can not handle all bits directly, only eight bits are handled by the ALU and sent via data bus. The remaining bits are handled by the Decode and Control unit.

An overview of the instruction set is provided by Fig 3.4 and organized as follows. The first column shows the mnemonic and needed operands for each instruction. A brief description is then given in the next column. The third column indicates how many instruction cycles are needed to perform an instruction. As can be seen, some instructions can take one or two cycles to perform. These are conditional branch instructions, which, according to Section 3.2, will only need two cycles if the branch is taken. The last column shows the bit representation for each instruction.

Besides this, the table is roughly divided into three categories, one for each operation type. In the first category we see that the byte-oriented instructions mainly

Mnemonic, Operands		Description	Cycles	14-Bit Opcode
BYTE-ORIENTED FILE REGISTER OPERATIONS				
ADDWF	f, d	Add W to f	1	00 0111 dfff ffff
ANDWF	f, d	AND W to f	1	00 0101 dfff ffff
CLRF	f	Clear f	1	00 0001 1fff ffff
CLRW	-	Clear W	1	00 0001 0xxx xxxx
COMF	f, d	Complement f	1	00 1001 dfff ffff
DECF	f, d	Decrement f	1	00 0011 dfff ffff
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00 1011 dfff ffff
INCF	f, d	Increment f	1	00 1010 dfff ffff
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00 1111 dfff ffff
IORWF	f, d	Inclusive OR W to f	1	00 0100 dfff ffff
MOVF	f, d	Move f	1	00 1000 dfff ffff
MOVWF	f	Move W to f	1	00 0000 1fff ffff
NOP	-	No Operation	1	00 0000 0xx0 0000
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff
BIT-ORIENTED FILE REGISTER OPERATIONS				
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff
BSF	f, b	Bit Set f	1	01 01bb bfff ffff
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff
LITERAL AND CONTROL OPERATIONS				
ADDLW	k	Add literal to W	1	11 111x kkkk kkkk
ANDLW	k	AND literal to W	1	11 1001 kkkk kkkk
CALL	k	Call Subroutine	2	10 0kkk kkkk kkkk
CLRWDT	-	Clear Watchdog Timer	1	00 0000 0110 0100
GOTO	k	Go to address	2	10 1kkk kkkk kkkk
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk
RETFIE	-	Return from interrupt	2	00 0000 0000 1001
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk
RETURN	-	Return from Subroutine	2	00 0000 0000 1000
SLEEP	-	Go into Standby mode	1	00 0000 0110 0011
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk

Figure 3.4: The instruction set of the PIC16F687 [MC07]

consist of arithmetic and logical operations, e.g. addition and rotation, which work on file-registers. In the area of bit-oriented instructions principally only two types of operations occur. One for (conditionally) setting a certain bit in a register and one for (conditionally) clearing it. The last category offers literal and control operations. Literal operations perform arithmetic and logical operations exactly the same way as the byte-oriented instructions but with constant values instead of file-registers. Control operations are mainly used for unconditional branching and for controlling special features of the CPU, e.g. setting it to sleep mode.

4 Power Consumption Properties

In the upcoming sections, a detailed overview of the power consumption properties of the PIC is provided. Thereby, each section will only cover one property at a time even though more than one influencing factor may be visible as a side effect. The procedure is as follows. At first expectations are discussed. Then, these are verified by means of visual analysis, i.e. Simple Power Analysis, of power consumption traces which were measured while the device executed case specific code sequences. As a result, a power model for the PIC is presented.

However, beforehand it has to be discussed how power traces of the PIC are basically analyzed. Therefore, in Fig. 4.1, an exemplary power consumption trace (bottom/blue) with corresponding trigger (center/red) and clock signal (top/green) is illustrated. In this figure, the abscissa is the time axis and shows at which point in time, relative to the origin, a certain sample was measured. The ordinate indicates the measured value and is provided as a voltage value, which is proportional to the power consumption. However, in case of Fig. 4.1 it is given in arbitrary units, since clock and trigger signal are in a five volts range, whereas the power consumption trace is in the millie volts range.

As can be seen, the power trace shows significant peaks with every rising edge of the clock. This is quite obvious because a rising edge forces the controller to perform the next transitions in an instruction cycle resulting in a significantly increased dynamic power consumption. The trigger signal is used to activate the power trace acquisition and is set by the last instruction before a test code sequence starts. More precisely, it is generated by setting one output pin of the PIC to high. From the power consumption perspective, this consumes additional energy so that the trigger signal is equally visible in the trace as an extremely high peak. Hence, with this unique peak, we have a good indication for the point in time where the first instruction of the test code starts, namely directly after this peak. As we know from Section 3.2, it takes four clock cycles ($Q1$ to $Q4$) to perform one instruction cycle. Thus, the first four clock peaks correspond to the first instruction cycle, the following four peaks to the second instruction cycle and so forth. From this it follows that in the illustrated trace the power consumption of three instruction cycles ($Cy1$ to $Cy3$) is visible.

In summary, all information needed for interpreting a power consumption trace is inherently given, so that trigger and clock signal are omitted in all following power

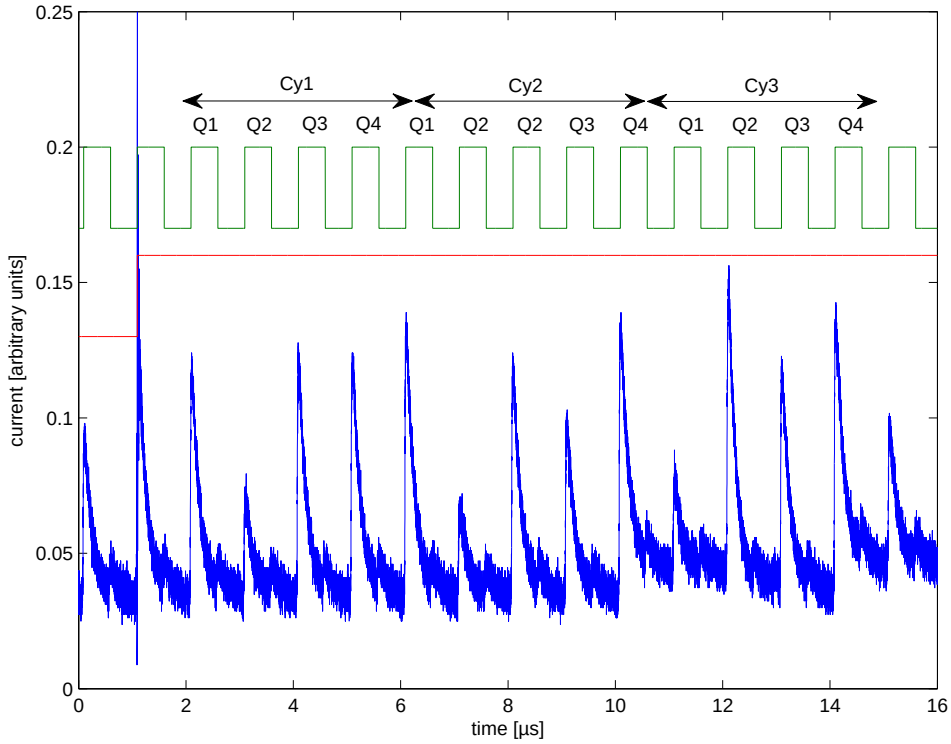


Figure 4.1: How to interpret power consumption traces of the PIC16F687 – clock signal (top/green), trigger signal (center/red) and power consumption (bottom/blue)

consumption figures.

4.1 Clock

First, the influences of the clock will be analyzed. As we have seen in the exemplary figure of the previous section, peaks occur at rising edges of the clock. If the clock rate is increased, a higher power consumption can be expected (see equation (2.2)) due to a higher switching frequency of the transistors. Additionally, effects due to capacitances may be visible, mainly because the distance between peaks decreases with higher clock rates.

To test this, one and the same code sequence consisting of five instructions was executed for clock frequencies of 32kHz, 1MHz, and 4MHz respectively. The results are shown in Fig. 4.2.

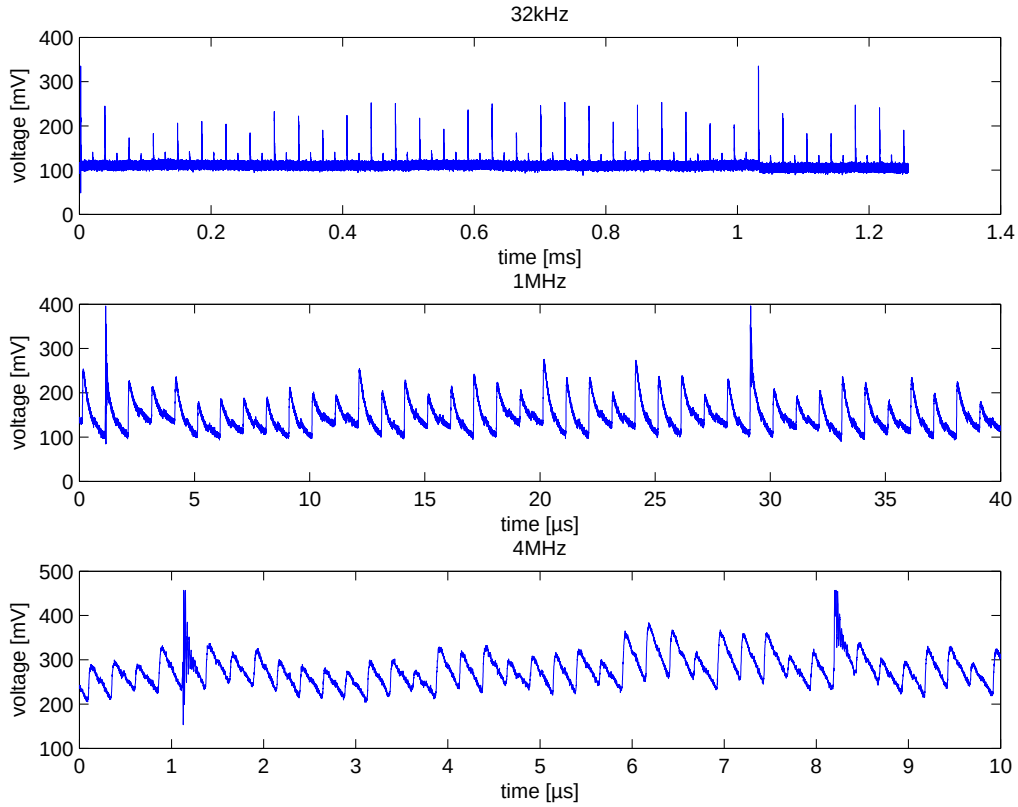


Figure 4.2: Clock rate influences – same code sequence at 32kHz, 1MHz, and 4Mhz

As can be seen in this figure, at a clock speed of only 32kHz, the power consumption trace looks extremely regular. Peaks only occur at rising and falling edges of the clock, whereas even the peaks of falling clock edges have a constant height. Thus, only the peaks at rising edges show differences and are therefore the only indication for the kind of instructions executed during that instruction cycle. The measured voltages are in-between 100 and 320 mV.

At a clock rate of 1MHz the trace becomes more dynamic. At rising clock edges we see that the power consumption rises rapidly but falls back slowly[1]. As already mentioned, this is most likely caused by reloading capacitances of the device. As a result, when the capacitances are not completely reloaded before the next clock edge rises, the trace gets a steppy shape. Additionally, the peaks which occur at falling clock edges can no longer be as clearly identified as in the case of 32kHz. They are only indicated by a small bias on top of the falling curve. The measured voltage is in-between 100 and 400 mV and thus slightly higher than at a clock rate of 32kHz.

[1] a similar behavior can even be seen in the 32kHz trace. However, due to the high time scale, this can not be identified in the figure

If the clock rate is further increased, as shown by the example of 4MHz, the time for reloading the capacitances becomes even shorter. As expected, the time is reduced to a fourth of the 1MHz test run. Thus, the power consumption trace gets an even more dynamic shape. Actually, the power consumption of one instruction cycle propagates into the next one, which was not the case at a clock rate of 1MHz. For instance, the trace of 4MHz has a lower constant component at the beginning than at the end, whereas the 1MHz trace always falls back to the same level at the end of an instruction cycle. The measured voltages in this case are significantly higher than in the case of 32kHz and 1MHz.

At this point, the decision has to be made with which clock rate to proceed. Obviously, the use of 4 MHz is not appropriate since the recognition of single instruction cycle will be hindered. Thus, 1 MHz or 32 kHz are applicable. Since 1 MHz is closer to a real case clock rate and further speeds up the measuring process, solely this clock rate will be utilized in this work. Yet it is merely questionable whether a clock rate of 32 kHz yields better results.

4.2 Working Register

The working register functions as one of two operands of the ALU. Thus, if an instructions works with this operand and consequently with the ALU, this will influence the power consumption. However, even if an instruction does not need the ALU, the content of W may be sent to the ALU and from there to the data bus anyhow. For instance, a NOP may be executed as adding zero to W. Hence, we can assume that the contents of W may influence the power consumption in any case.

To check this, the test described in Test Description 4.2.1 was performed. It covers two test cases. In the first case, W is set to `0x00h` before the trigger signal is set. Then, a sequence of five NOP instructions is executed and measured before the trigger signal is set back. The second case works exactly the same way but with setting W to `0xffh`. As a result, in both cases identical code sequences are executed. Since W is not modified by NOP instructions, the only difference concerning both cases is the actual content of W, which is stored before the trigger signal is set and thus before measuring starts. The following reasons justify the use of `0x00h` and `0xffh` as values of W.

Test Description 4.2.1 (Working register) *To see how the working register affects the power consumption, the following test was performed:*

1. *set W to* `0x00h` *and* `0xffh` *respectively*
2. *set trigger signal*
3. *execute and measure a sequence of five NOP instructions*

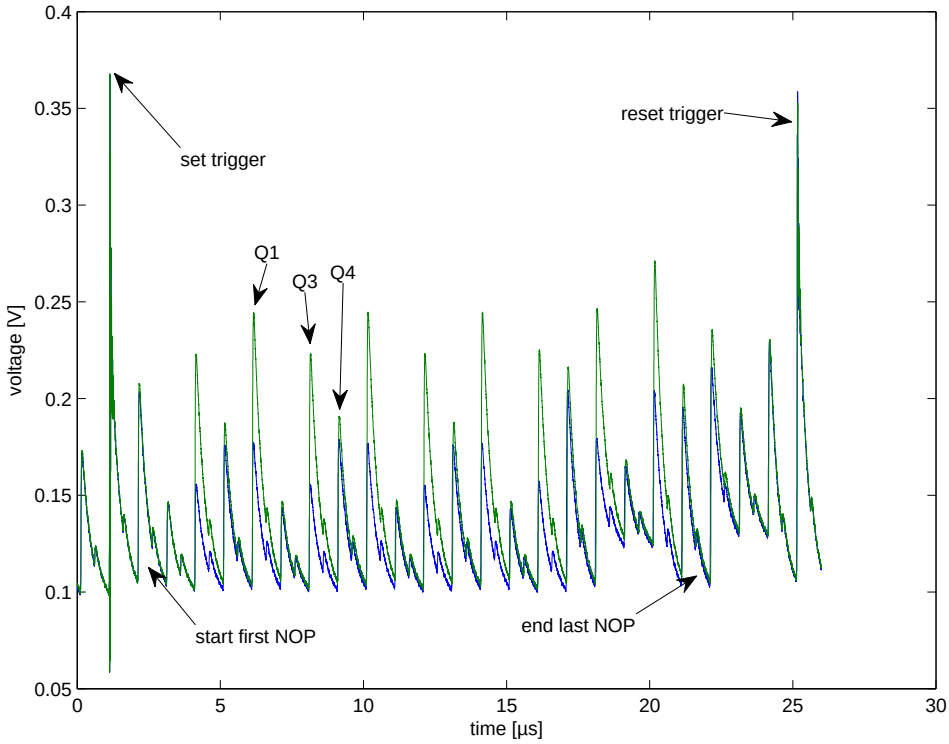


Figure 4.3: Power consumption influences of the working register – power traces of two code sequences of five NOP instructions with different working register contents

4. *reset trigger signal*
5. *repeat one hundred and fifty times*
6. *calculate the average trace for each test case*

By setting W to `0x00h` and `0xffh` respectively, we have two completely different Hamming weights: in one case it is zero and in the other case it is eight. According to the discussed Hamming-Weight power model in Section 2.2.2 and under the assumption that this model is adequate, the result will show significant deviations in the power consumption traces since the power consumption in this model is assumed to be proportional to the Hamming weight of a value.

The results of the test are shown in Fig. 4.3. The trace for the value `0x00h` is plotted in blue, whereas the trace for value `0xffh` is plotted in green. Both are averaged over one hundred and fifty measurements. To review how power traces are analyzed, the most important points are marked by arrows. At first, we have two arrows indicating the set and reset instruction for the trigger signal, visible as

comparatively high peaks at the start and end of the traces. Hence, these points mark start and end of the test sequence. The first peak following the set-trigger peak corresponds to $Q1$ of the first NOP instruction[2]. Thus, this peak together with the following nineteen peaks comprise the five NOP instructions.

As can be seen, $Q1$, $Q3$, and $Q4$ of the second instruction, indicated by arrows, show offsets which can only be caused by different contents of W. More precisely, the $Q1$,$Q3$, and, less distinctive, the $Q4$ peak for a value of `0x00h` are significantly smaller than for a value of `0xffh`. Same holds for the third and fourth NOP. Since it is known from the device examination that the instruction is executed in the third and stored in the fourth clock cycle, $Q3$ may be caused by sending the contents of the ALU to the data bus and from there back to W during $Q4$. About $Q1$ at this state, no assumption can be drawn up. However, the Hamming-Weight model seems to describe the discussed peaks quite accurately which was further verified by additional tests performed with different Hamming weights.

The contents of the working register highly influences $Q1$, $Q3$ and less distinctively $Q4$. $Q2$ remains unaffected.

In contrast to this, the last NOP is different compared to its predecessors. The peaks of $Q2$ to $Q4$ in this case provide an offset of approximately $25\,mV$. However, the same features are visible as with the previous NOP instructions. Consequently, this deviation is most-likely caused by a fetching process, i.e. fetching the BCF instruction resetting the trigger. Yet, this is only an assumption and has to be verified by adequate tests later on.

Furthermore, the first NOP is also different to the second to fourth NOP, since $Q1$ does not provide the same offset. In consequence of this, the Hamming-Weight model does not describe the power consumption of $Q1$ accurately in this case. We have to keep this in mind when analyzing other aspects of the power consumption in order to reveal causes for this difference.

Recapitulatory, even though the working register may not be needed by an instruction, its contents can have significant influences on the power consumption of $Q1$ and $Q3$ and less distinctive on $Q4$. However, $Q2$ remains unaffected.

4.3 Fetching Process

In the last section we have discovered that the power consumption of an instruction may be affected by the two-stage pipeline, i.e. the power consumption may depend

[2] Actually, this was a working hypothesis which turned out to be correct due to the fact that all subsequent observations can be explained by assigning the highest peak to $Q4$ of the (re)set trigger instruction.

on the instruction that is fetched while another one is executed. This will be analyzed in more detail throughout this section.

By now, it is known from Section 3.2 that the fetching process starts by incrementing PC in $Q1$ and ends at $Q4$. Hence, theoretically the power consumption of an entire instruction cycle may be influenced.

Test Description 4.3.1 (Fetching Process) *To see how the fetching process affects the power consumption, the following test was performed:*

1. *set W to* `0x55h` *to have fixed value*
2. *set trigger signal*
3. *execute and measure one of the following sequences*

   ```
   NOP NOP [NOP / SUBWF 0x5Dh / SUBLW 0x55h] NOP NOP
   ```

4. *reset trigger signal*
5. *repeat one hundred and fifty times for each possible code sequence*
6. *calculate the average trace for each test case*

To analyze this, the test described in Test Description 4.3.1 was performed. At first, W is set to `0x55h` simply to have a clearly defined value throughout the test sequence. Then, the trigger signal is set to activate the trace acquisition before the test sequence consisting of five instructions starts. In this sequence, all instructions are fixed except for the third one which is one of the following : NOP, MOVWF `0x5Dh`, or MOVLW `0x55h`. The MOVWF instruction is a file-register operation which moves the contents of W to the file-register its operand points to, in this case `0x5Dh`. In contrast to this, MOVLW is a literal operation moving a literal (`0x55h`) to W. Note that both instructions do not modify the contents of W so that after instruction execution the entire data path is in the same state as before. As a result, the successive instructions should not be affected.

Further, the opcodes of MOVWF `0x5Dh` and MOVLW `0x55h` have the same Hamming weight of six, in contrast to the NOP which has a Hamming weight of zero (see Fig. 3.4). Thus, by comparing the NOP trace to the others, differences concerning the Hamming weight can be identified as far as they exist. By comparing MOVWF to MOVLW we may then be able to identify differences concerning the type of instruction, i.e. file-register or literal operation.

The results of the tests are illustrated in Fig. 4.4. The upper plot shows the entire averaged traces for each scenario. The trace for the NOP instruction is plotted in blue, MOVWF is plotted in green, and MOVLW in red. As expected, all traces have an identical shape except for the second and third instruction cycle in which the test instruction is fetched and executed respectively.

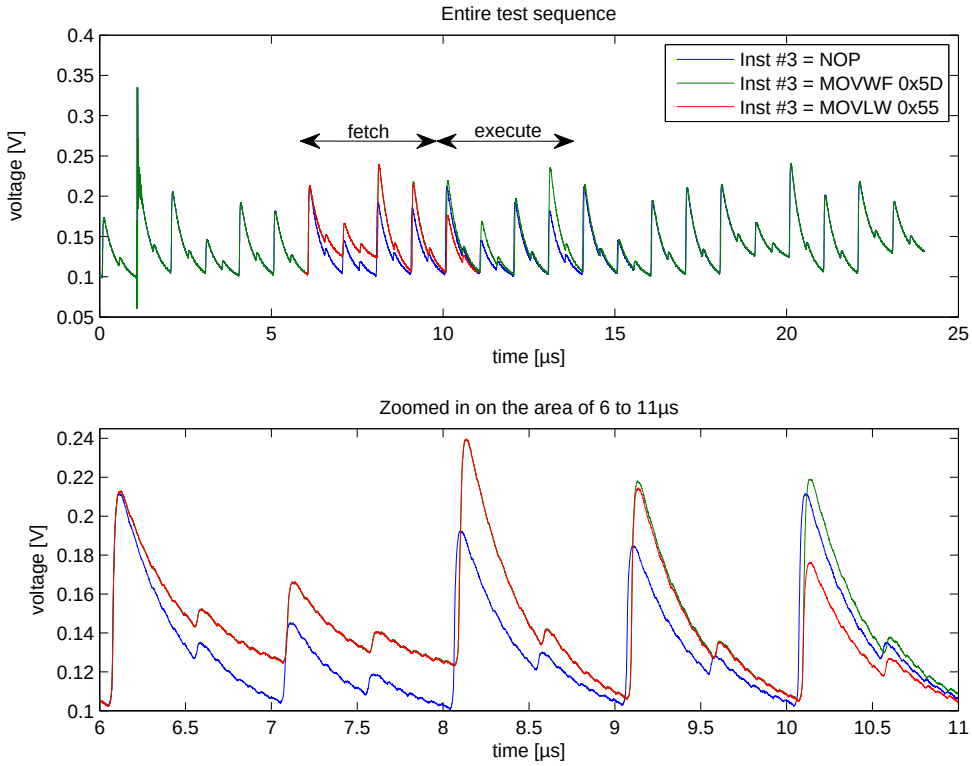


Figure 4.4: Power consumption influences of the two-stage pipeline – the upper plot shows three traces for three different test cases, the lower plot shows a zoom of the upper trace in the area of 6 to 11 μs

The fetching process caused by the two-stage pipeline increases the overall power consumption of $Q2$ to $Q4$ in a direct proportional relationship to the Hamming weight of the opcode. $Q1$ remains unaffected.

The lower plot shows a zoom-in on the area of 6 to 11 μs which contains $Q1$ to $Q4$ of the instruction cycle when the test instruction is fetched plus $Q1$ of the instruction cycle when it is executed. As can be seen, $Q1$ of fetching the test instruction is equal for all three cases. In contrast to this, $Q2$ to $Q4$ are obviously affected by the Hamming-Weight of the opcode because they are equal for MOVWF and MOVLW whose opcode has the same Hamming weight but not for NOP which has a Hamming weight of zero. Since, $Q2$ to $Q4$ are significantly higher for MOVLW and MOVWF, this indicates a direct proportional relationship to the Hamming weight of the opcode which could be verified by additional tests. Further, when comparing the MOVLW trace to MOVWF we see that $Q4$ of MOVWF has s slightly higher peak. As a result, this may be an indicator for the kind of operation that is fetched.

At this point, we have extensively discussed the influences on the power consumption caused by the fetching process. However, the lower plot of Fig. 4.4 shows an unexpected feature from what is known so far. Namely, $Q1$ of the instruction cycle in which the test instruction is executed differs for every test case. This is unexpected because in Section 4.2.1 we have identified $Q1$ to be influenced by the contents of W which, in our case, has a constant value of `0x55h`. Hence, the power consumption should be equal as well. Since this is not the case, it follows that the power consumption has to be additionally influenced by the opcode, which is the only difference concerning the three test cases. Obviously, there is no proportional or anti-proportional relationship to the Hamming weight of the opcode because in this case MOVWF and MOVLW, whose Hamming weight is six, would be equal at $Q1$. Therefore, the observation may be caused by the Hamming distance of W and an operand given by the opcode. In the case of MOVLW, the value of the operand is `0x55h`. Hence, we get a Hamming distance of zero compared to the contents of W. This explains why MOVLW shows the smallest peak but it does not explain why MOVWF shows the highest. In this case, the operand given by the opcode is `0x5Dh`. Hence, the Hamming distance is one and thus close to the case of MOVLW. As a result, the high peak for MOVWF may be caused by the Hamming distance of W and the contents of the file-register the instruction points to. Since this register was not manipulated before, it is set to `0x00h` which results in a Hamming distance of four and is thus an explanation for the high peak. However, this has to be proved by additional tests in the following section.

4.4 Data Bus

In the last section we made an unexpected observation concerning the power consumption of $Q1$ and put this down to the Hamming distance of the contents of W and the second operand of the ALU, which is either provided as part of the opcode in case of literal instructions or as the contents of a file-register stored in the SRAM. If this is true, both values have to interact somewhere in the data path. As can be seen in the block diagram of Fig. 3.1, both SRAM and IR are connected to the data bus, IR via a multiplexer. Hence, if our assumption about the power consumption of $Q1$ is correct, it is most-likely justified by a changing value from W to the operand on the data bus.

Test Description 4.4.1 (Data Bus – Literal) *To see how the multiplexer connected upstream the ALU affects the power consumption concerning literal operations, the following test was performed:*

1. *set W to* `0x55h` *to have fixed value*
2. *set trigger signal*

3. *execute and measure one of the following sequences*

   `NOP [ADDLW 0x55h / ADDLW  0xAAh / ADDLW  0x33h] NOP`

4. *reset trigger signal*
5. *repeat one hundred and fifty times for each possible code sequence*
6. *calculate the average trace for each test case*

To test our hypothesis, two tests are performed as described in Test Descriptions 4.4.1 and 4.4.2 respectively. The former uses the literal instruction ADDLW to add a constant value to W, whereas the latter one uses the equivalent file-register operation ADDWF. By this means, we are able to decide whether our assumption is correct and further whether it holds for both operation types.

Before the test sequence of the first test starts, W is set to a value of `0x55h` as in the previous test. Then three test cases are executed and measured one hundred and fifty times. The test cases differ in the operand which is provided by the opcode of the ADDLW instruction and is either set to `0x55h`, `0xAAh` or `0x33h`. These values are chosen in a way that three different Hamming distances occur, as shown in (4.1) to (4.3).

$$HD(\texttt{0x55h}, \texttt{0x55h}) = HW(\texttt{0x55h} \oplus \texttt{0x55h}) = 0 \tag{4.1}$$

$$HD(\texttt{0x55h}, \texttt{0x33h}) = HW(\texttt{0x55h} \oplus \texttt{0x33h}) = 4 \tag{4.2}$$

$$HD(\texttt{0x55h}, \texttt{0xAAh}) = HW(\texttt{0x55h} \oplus \texttt{0xAAh}) = 8 \tag{4.3}$$

For the second test ADDLW is replaced by the file-register instruction ADDWF, which performs the same operation but by means of a file-register. Since the summand added to W is not provided by the opcode in this case, the respective file register has to be set to the test value in advance. However, as with the previous test we get three different Hamming distances and can thus decide if the Hamming distance model describes the power consumption adequately.

Test Description 4.4.2 (Data Bus – File-register) *To see how the multiplexer connected upstream the ALU affects the power consumption concerning file-register operations, the following test was performed:*

1. *set file-register to* `[0x55h / 0xAAh / 0x33h]`
2. *set W to* `0x55h`
3. *set trigger signal*
4. *execute and measure the following sequences*

   `NOP ADDWF 0x40h,1 NOP`

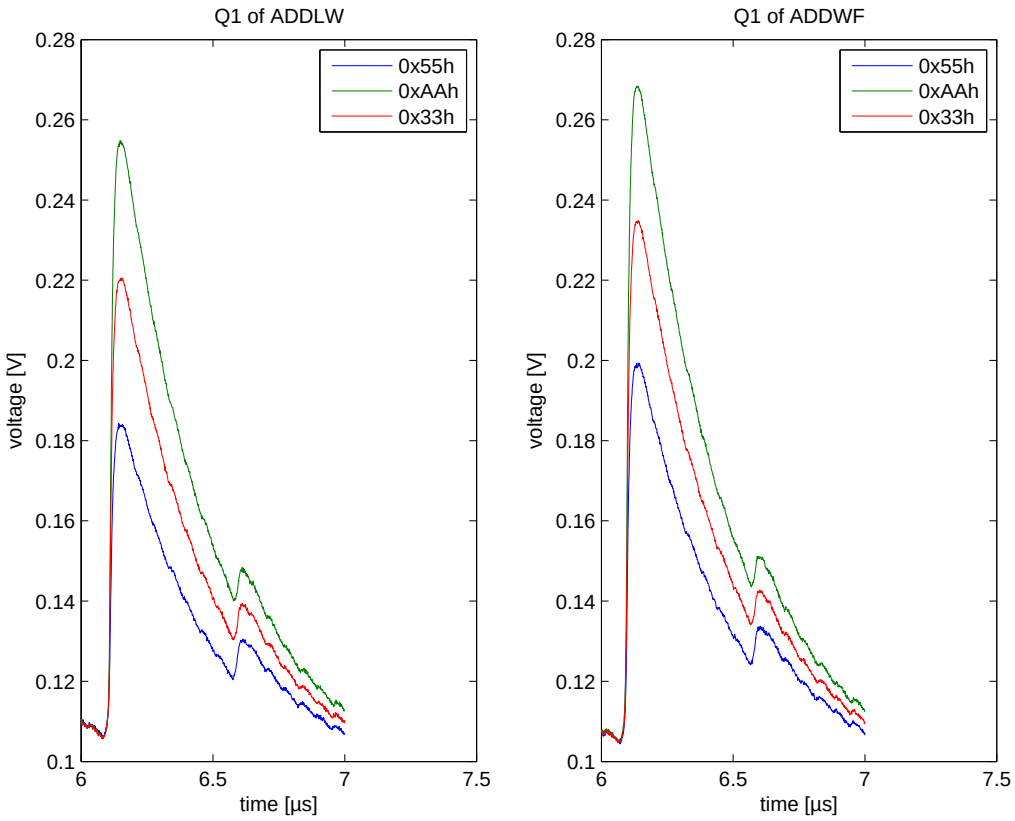


Figure 4.5: Effects caused by the data bus – three average traces for ADDLW (left) and MOVWF (right)

5. *reset trigger signal*
6. *repeat one hundred and fifty times for each value of file-register* `0x40`
7. *calculate the average trace for each test case*

Fig. 4.5 illustrated two power consumption plots of $Q1$ each showing the averaged traces for the three different test cases. The left plot corresponds to the execution of ADDLW, whereas the right plot corresponds to ADDWF. As can be seen, the plot for ADDLW shows a direct proportional relationship to the Hamming distance of W and the three literals, i.e. `0xAAh` with a Hamming distance of eight shows the highest peak and `0x55h` the smallest. Same holds for ADDWF and the different contents of the file-register. Hence, our assumption about the influences on $Q1$ was correct. Additionally, when comparing both plots, we see that the power consumption, in average, is higher for the file-register operations than for the literal operations. This may be due to the fact that SRAM is activated during the execution of a file-register operation.

$Q1$ is determined by the Hamming distance of the operand sent to the data bus and the value which is replaced by it.

Since the assumption about the Hamming distance model at $Q1$ was correct we can further state that this behavior is in conjunction with changing values on the data bus, as explained at the beginning of this section. This means that whenever data on the data bus is modified, the power consumption will be proportional to the number of bits that have to be changed. Consequently, we can assume that when the ALU executes the actual instruction, e.g. performs an addition, this will result in an equivalent behavior, since the result is sent to the bus. This will be analyzed in the next section.

4.5 Arithmetic Logical Unit

As is known from the theoretical examination of the device, instructions are processed by the ALU in $Q3$. Hence, due to the results of the last section we can assume that the Hamming distance is an adequate model to describe this peak. More precisely, we know from the last section that during $Q1$, the last result of the ALU is replaced by the operand needed to perform the current instruction. Hence, it is quite logical that in $Q3$ this operand is again replaced by the result calculated by the ALU. As a result, $Q3$ will show a direct proportional relationship to the Hamming distance of the operand used to perform the instruction and the result calculated by the ALU.

To test this, we can use the traces of the tests performed in the previous section. Descriptions can be found in Test Descriptions 4.4.1 and 4.4.2. In these tests, either `0x55h`, `0xAAh` or `0x33h` was added to W, which was initially set to `0x55h`. Hence, the results calculated by the ALU have the following Hamming distances compared to the operand on the data bus:

$$HD(\texttt{0x55h}, \texttt{0x55h} + \texttt{0xAAh}) = HD(\texttt{0xAAh}, \texttt{0xFFh}) = 4 \tag{4.4}$$

$$HD(\texttt{0x33h}, \texttt{0x55h} + \texttt{0x33h}) = HD(\texttt{0x88h}, \texttt{0xFFh}) = 6 \tag{4.5}$$

$$HD(\texttt{0x55h}, \texttt{0x55h} + \texttt{0x55h}) = HD(\texttt{0x55h}, \texttt{0xAAh}) = 8 \tag{4.6}$$

$Q3$ is determined by the Hamming distance of the result calculated by the ALU and the literal or file-register content the instruction was performed with.

As can be seen, we get three different Hamming distances, which should be identifiable by inspecting the power consumption trace of $Q3$ for both tests. This is

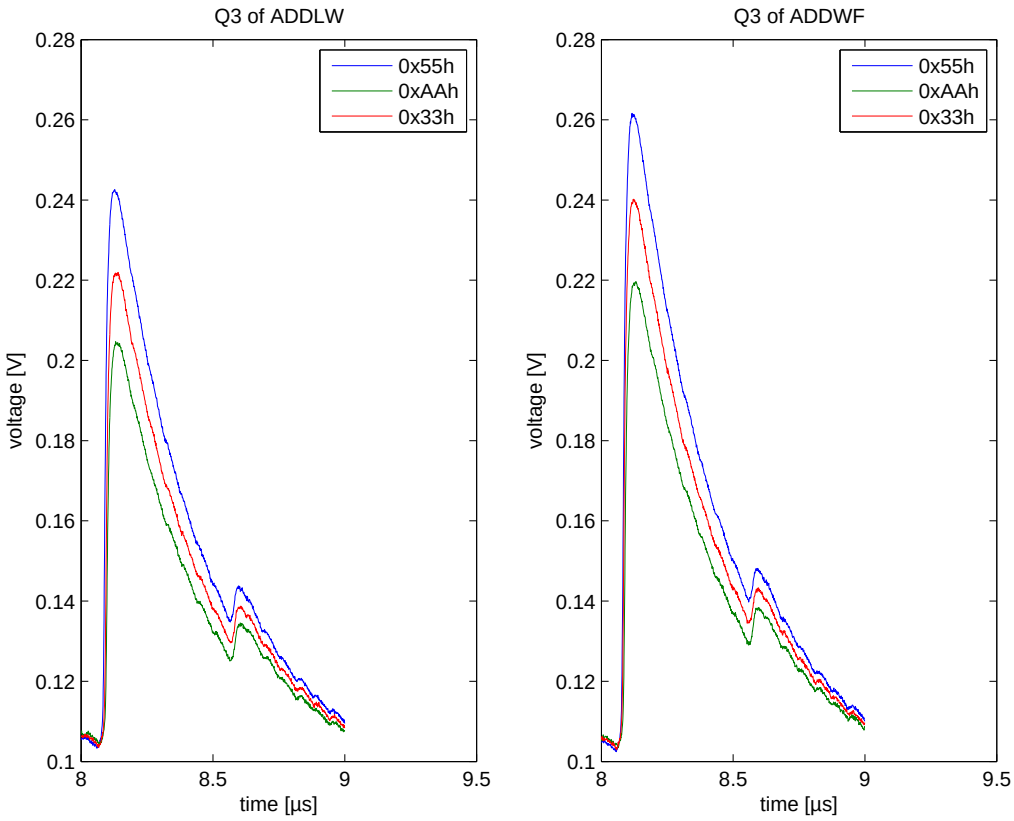


Figure 4.6: Effects caused by the result of the ALU – three average traces for ADDLW (left) and MOVWF (right)

shown in Fig. 4.6. The left plot shows the three average traces for MOVLW and the right plot for MOFWF. As expected, the value of `0x55h` shows the highest power consumption in both plots, since the Hamming distance in this case is maximized. According to this, `0xAAh` with a Hamming distance of four shows the smallest peak. Hence, this indicates the correctness of our assumption. Again, this was verified by additional tests.

4.6 Opcode

In the last two sections we have discovered how literals or file-register addresses affect the power consumption of $Q1$ and $Q3$ due to the data bus. However, until now we do not know if the Hamming weight of the opcode in general influences the power consumption.

To check this, the procedure given in Test Description 4.6.1 was performed, which

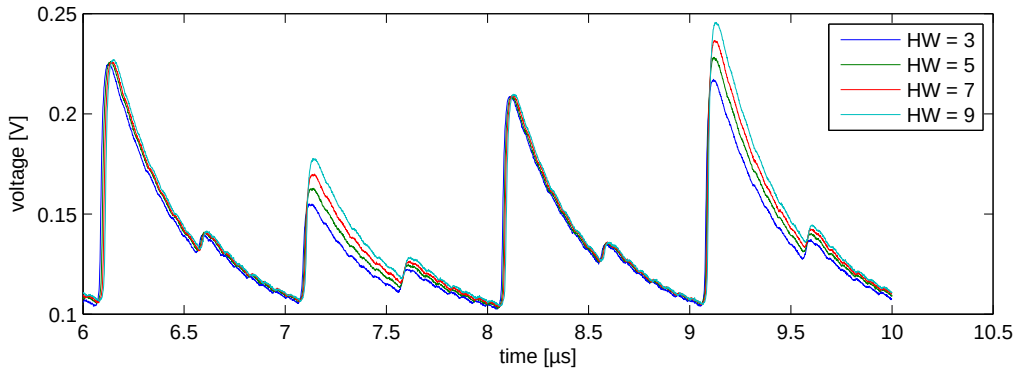


Figure 4.7: Effects caused by Hamming weight of the opcode – averaged traces for four different Hamming weights

works with the MOVWF instruction. It comprises four test cases for four different file-register addresses. At first, the respective file-register is cleared and W is set to `0x55h` before the trigger signal is set. Then the test sequence is executed and measured for one hundred and fifty times. The MOVWF instruction has an important property: the Hamming weight of the opcode can be increased without changing any other conditions possibly influencing the power consumption. For instance, the operands and thus the result calculated by the ALU is equal for each test case as long as the file-register is cleared before the test execution.

Test Description 4.6.1 (Opcode) *To see how the Hamming weight of the opcode affects the power consumption the following test was performed:*

1. *clear file-register* `[0x40h / 0x54h / 0x5Dh / 0x7Fh]`
2. *set W to* `0x55h`
3. *set trigger signal*
4. *execute and measure the following sequences*

   ```
   NOP MOVWF [0x40h / 0x54h / 0x5Dh / 0x7Fh],1 NOP
   ```

5. *reset trigger signal*
6. *repeat one hundred and fifty times for each file-register address* `0x40`
7. *calculate the average trace for each test case*

The results of the test are illustrated in Fig. 4.7. It shows the average traces for the four different Hamming weights of the opcode, which are in range of three to nine. As can be seen, $Q1$ and $Q3$ are constant in each test case due to a constant value of W and the respective file-register. However, $Q2$ and $Q4$ are affected by the

Hamming weight of the opcode in a direct proportional relationship. Hence, these peaks can be described by means of the Hamming-Weight model.

> $Q2$ and $Q4$ are directly affected by the Hamming weight of the opcode in a direct proportional relationship.

Consequently, $Q2$ and $Q4$ do not only indicate the Hamming weight of the opcode, but additionally the operands which are stored in it, i.e. literals, bit addresses, or file-register addresses. Thus, these peaks alone can not be used to clearly identify an instruction, since, as we have seen by the example of MOVWF, the Hamming weight of one and the same instruction can significantly vary in the Hamming weight.

4.7 Instruction Register

The results of Section 4.3 and 4.6 implicate the question whether the data on the program bus which is connected to IR can be identified in the power consumption.

For this, a new test is performed as given in Test Description 4.7.1. At first, W is set to `0x55h` by executing a MOVLW instruction. Subsequently, another MOVLW is executed with the operand either being `0x55h`, `0x5Ah`, or `0xAA`. Since we know from the last section that the Hamming weight of the opcode affects the shape of the power consumption trace, these operands all have the same number of bits set. However, the Hamming distance between the opcode of executing this instruction and the previous one differs. More precisely it is either zero, four or eight. As a result, we will be able to determine if the Hamming distance between changing values on the program bus or in IR respectively can be identified. If this assumption is true, then it should occur at $Q4$ since this is the time when the new instruction is latched into IR.

Test Description 4.7.1 (Instruction Register) *To see how the program bus connected to the instruction register affects the power consumption, the following test was performed:*

1. *set trigger signal*
2. *execute and measure the following sequences*

   ```
   NOP MOVLW 0x55h MOVLW [0x55h, 0x5Ah, 0xAA] NOP NOP
   ```

3. *reset trigger signal*
4. *repeat one hundred and fifty times for each operand of ADDLW*
5. *calculate the average trace for each test case*

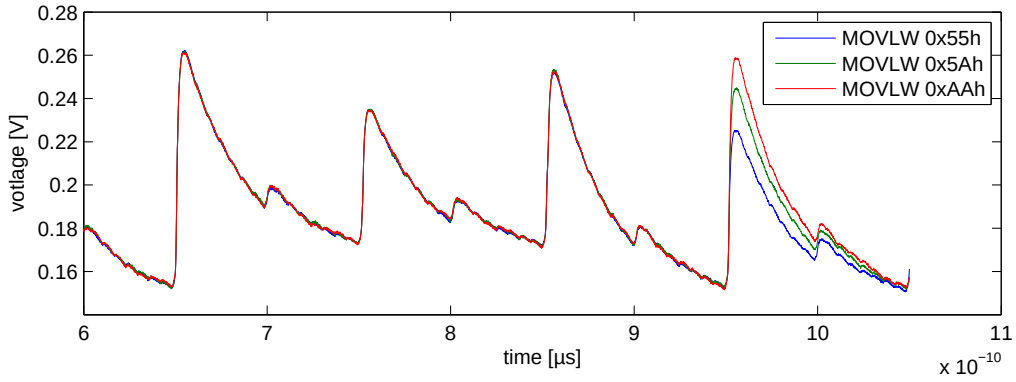


Figure 4.8: Effects caused by Hamming distance of the current and previous opcode – averaged traces for three different Hamming distances

$Q4$ shows a direct proportional relationship to the Hamming distance of the executed and fetched opcode.

Fig. 4.8 shows the power consumption of the instruction cycle in which the first MOVLW instruction is executed. As can be seen, $Q1$ to $Q3$ are undistinguishable. Nonetheless, $Q4$ shows differences proportional to the Hamming distance of the current instruction being executed and the subsequent one, which is the second MOVLW instruction differing in the used literal. Hence, our assumption has been proved.

4.8 Noise

The last influence that is analyzed is the noise component of the measured power consumption. For this, the power consumption of a fixed code sequence with fixed operands was measured one thousand times. The distribution for one fixed point in time can then be illustrated by a histogram, which is often used to present the distribution of data values. The results are shown in Fig. 4.9. At the axis of abscissae the measured voltage is plotted. The ordinate shows the frequency of the measured voltage. Thus, the sum of all occurrences is equal to the number of measurements, i.e. one thousand. As can be seen from the histogram, the noise component is close to normal distributed. Solely the highest peak seems to be unusual for the normal-distribution indicated by the remaining peaks. However, with only a set of one thousands samples such divergences can occur.

As a result, since the noise component of single points is normal-distributed, the Multivariate-Gaussian distribution is an appropriate model for the noise component of the PIC.

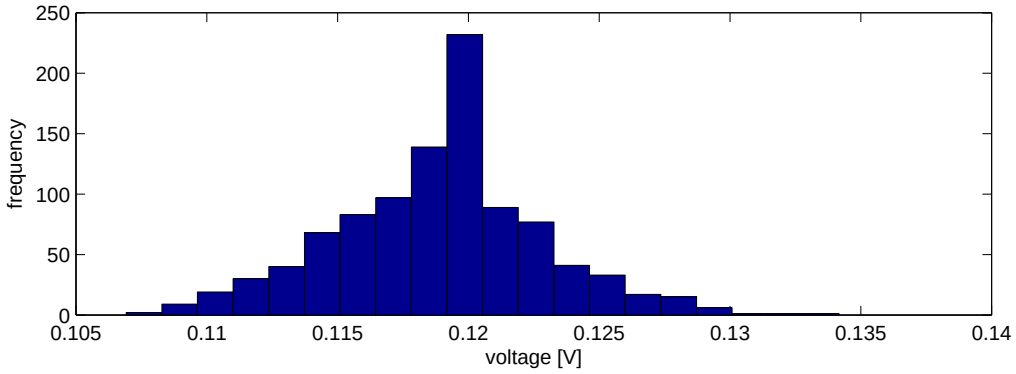


Figure 4.9: Histogram showing the noise distribution for a fixed point in time

4.9 Other Influences

Until now, we have discussed the major influences on the power consumption. Nevertheless, the device contains additional units like, e.g., I/O ports, A/D converter or the EUSART, which as a matter of course affect the power consumption. We have seen by the example of the I/O port, which is activated when setting the trigger that these effects can be significant. However, throughout this work, none of these components will be activated during the execution of a test sequence, so that the power consumption of the peripheral components can be assumed as constant so that is should not have significant negative effects.

Furthermore, the power consumption dependencies of the register contents which are modified by an instruction read during $Q2$ and written during $Q4$, have been tested. However, the effects were comparatively insignificant so that these can generally be ignored. However, as we have seen in the first test, the NOP instruction seems to be an exception since in this case $Q4$ showed deviations which can not be explained by one of the discussed influences.

4.10 Summary and Conclusion

In the last sections we found out much about the power consumption properties of the device by simply visually analyzing power consumption traces. The results are summarized in Fig. 4.10.

At first, we discovered that the clock rate significantly influences the shape of power consumption traces, mainly caused by effects due to capacitances. Hence, the clock rate was set to 1MHz for the following tests. Further, the fetching process

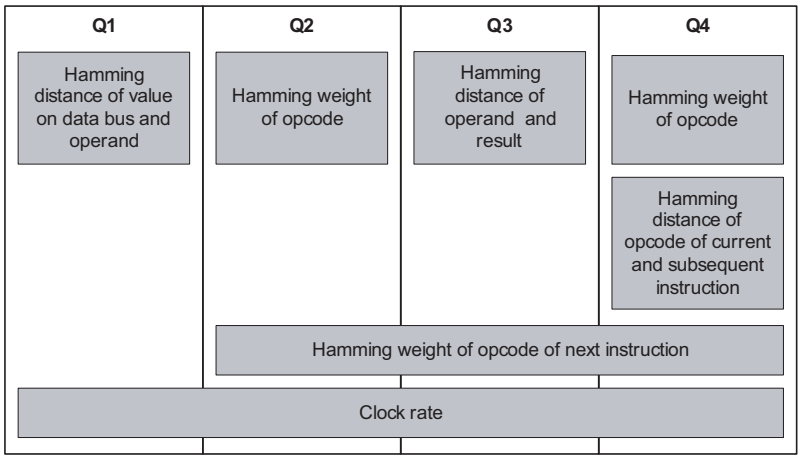


Figure 4.10: Summary of identified power consumption influences

affects the power consumption of $Q2$ to $Q4$ proportionally to the Hamming weight of the fetched instruction.

In $Q1$ the fetched instruction is already stored in IR so that in case of a literal instruction, the operand is sent to the ALU via the ALU multiplexer. In case of a file-register operation, it is sent from the SRAM to the data bus which is connected to the ALU using the same multiplexer. When these events proceed, the value that was previously on the data bus is replaced by the operand of the instruction which results in a proportional power consumption to the Hamming distance of both values.

In the time of $Q2$ the operand on the data bus is read by the ALU. A proportional relationship to the Hamming weight of the opcode could be identified as the only effect on the power consumption. This does not take into account the mentioned effects caused by the fetching process of the next instruction. Hence, it is further proportional to the literal or file-register address stored in the opcode.

In $Q3$ the ALU processes the instruction. The calculated result is sent to the data bus. Since the actual value on this bus is the operand loaded in $Q1$, we see a direct proportional relationship to the Hamming distance between these values.

Finally, in $Q4$ the result on the data bus is stored to memory or another peripheral component connected to the data bus. As we have seen by the example of setting the trigger signal, sending data to an output port causes a significant increase in the power consumption. However, storing data to the SRAM or to W has shown negligible effects. Furthermore, the next instruction is latched into IR in $Q4$ causing the power consumption to be directly proportional to the Hamming distance of the executed and latched instruction.

But nonetheless, the influences discussed so far constitute only the major effects

since we can not act on the assumption that the power consumption properties are described completely. For instance, certain bits of an operand may be weighted higher than others. Yet, this has not been analyzed. Moreover, we mainly focused on file-register and literal operations which work on two operands. Other instructions like, e.g., CLRW for clearing W, was not analyzed. However, the discussed instructions theoretically show the highest activity, since operands are loaded from memory, processed and stored back. Consequently, we can assume that other instructions, which for example only use one operand, show significantly less activity concerning the power consumption. For instance, a MOVLW instruction shows no deviations on $Q3$ concerning the used literal since this value is already on the data bus in $Q1$. Moreover, literal and file-register operations that work on two operands constitute the majority of nearly fifty percent of the entire instruction set and examinations of freely available program code showed that these are the most frequently used operations together with MOVLW and MOVWF. Hence, to get an overview of the power consumption properties, it was reasonable to concentrate on those effects.

As a consequence of the revealed influences, we can derive a qualitative power model for the various peaks of the instruction cycle as described in Definition 4.10.1.

Definition 4.10.1 (Power Model for the PIC) *The power consumption of $Q1$ to $Q4$ of an instruction cycle can be described by the following equations:*

$$P_{Q1} = P_{const,Q1} + P_{bus} \cdot HD(v_{bus}, v_{operand}) + P_{ext,Q1} + P_{el.noise} \tag{4.7a}$$

$$P_{Q2} = P_{const,Q2} + P_{opcode} \cdot (HW(v_{opcode}) + HW(v_{opcode+1}) + P_{ext,Q2} + P_{el.noise} \tag{4.7b}$$

$$P_{Q3} = P_{const,Q3} + P_{bus} \cdot HD(v_{operand}, v_{result}) + P_{opcode} \cdot HW(v_{opcode+1}) + P_{ext,Q3} + P_{el.noise} \tag{4.7c}$$

$$P_{Q4} = P_{const,Q4} + P_{opcode} \cdot (HW(v_{opcode}) + HW(v_{opcode+1}) + P_{prog.bus} \cdot HD(v_{opcode}, v_{opcode+1}) + P_{ext,Q4} + P_{el.noise} \tag{4.7d}$$

with $P_{bus} > P_{prog.bus} > P_{opcode}$ *and* $P_{el.noise} \sim N(0, \sigma)$.

As can be seen, each equation contains a constant power consumption $P_{const,Qx}$, which is caused by leakage currents or those transistors switching unaffected by the performed instruction or used operands. Furthermore, each clock cycle contains an extra power consumption $P_{ext,Qx}$, which corresponds to the part of the power consumption caused by effects that have not been revealed throughout this chapter. Hence, when estimating the power consumption with the provided equations, this part can not be taken into account. The electronic noise component $P_{el.noise}$, also a component of every equation, is, as shown in Section 4.8, a normal-distributed random variable which can be described by a mean value of zero and a variance σ depending on the measurement setup. The power consumption of the first clock

cycle P_{Q1} in particular further depends on the Hamming distance between the value on the data bus v_{bus} and the operand provided by an instruction $v_{operand}$. Hence, the power consumption can be estimated by multiplying a constant for the data bus – referred to as P_{bus} – to this Hamming distance. Similarly, the influences of the opcode concerning the current instruction (v_{opcode}) and the subsequent instruction ($v_{opcode+1}$) can be computed by means of a factor named P_{opcode}. Furthermore, the power consumption of the instruction bus affecting the power consumption at $Q4$ is represented by $P_{inst.bus}$. Finally, v_{result} occurring in $Q3$ corresponds to the value calculated by the ALU.

According to the performed tests, approximated values for the various components of the power consumption can additionally be provided as illustrated in Tab. 4.1. As can be seen, the values for $P_{const,Q1}$ to $P_{const,Q4}$ range from 144.5 mV up to 180 mV. Furthermore, P_{bus} increases the power consumption at $Q1$ and $Q3$ by about 10 mV per Hamming distance, which is significantly higher than for $P_{inst.bus}$. This was predictable as the data bus connects several peripheral components resulting in comparatively long wires and thus in a high load capacitance. At last, P_{opcode} can be estimated by 4.2 mV per Hamming weight and the tests showed that this value is suitable for weighting the influences of v_{opcode} as well as for $v_{opcode+1}$.

Component	Value (in mV)
$P_{const,Q1}$	176.5
$P_{const,Q2}$	144.5
$P_{const,Q3}$	163.5
$P_{const,Q4}$	180.0
P_{bus}	10.0
P_{opcode}	3.6
$P_{inst.bus}$	4.2

Table 4.1: Approximated values for the provided power model

Note, that the provided power model is more suitable than a simplistic Hamming-weight or Hamming-distance model. Hence, this power model may be used for DPA attacks by means of correlation coefficients on the PIC and should yield significant better results. Yet, this has not been proved.

To conclude, when executing one and the same instruction, the power consumption of that instruction is highly influenced by operands and opcode. Further, the next instruction influences the shape of the power consumption trace due to the fetching process as well as the previous instruction due to its effect on the current value on the data bus. Hence, it will most-likely become difficult to determine the executed

instruction by means of side-channel analysis, especially if only single traces can be taken into consideration.

5 Template Creation

In the last chapter we have discussed the power consumption properties of the PIC in detail. Now, this knowledge will be used in order to build adequate templates for identifying instructions because this would clearly enable to retrieve unknown program code. Furthermore it would provide the means for program path detection.

For this, in Section 5.1 and 5.2 respectively, the two main criteria for building templates will be discussed, i.e. how to select proper points and further how to partition templates. This means how many effects of the power consumption are comprised by a template. Then, the general procedure for testing the quality of the templates is discussed in Section 5.3 before several tests are performed and the results are presented.

5.1 Selecting Points

A template consists of a mean vector $\mathbf{m}$ and a covariance matrix $\mathbf{C}$. As discussed in Section 2.2.3, $\mathbf{C}$ grows quadratically in the number of points so that this number has to be limited in order to minimize numerical problems and computational complexity. Hence, the selection of points is an important and critical part of the template creation process.

As an example, if we measure an instruction with a sample rate of $1GS/s$ and a clock rate of 1MHz, one instruction is represented by 4000 samples as shown in (5.1).

$$n = \frac{1GS}{s} \cdot \frac{1}{1MHz} \cdot 4 = 4000\, S/Instruction\, Cycle \tag{5.1}$$

Hence, the covariance matrix would have sixteen million entries.

However, when limiting the number of points, we loose information which consequently reduces the accuracy of a template. Hence, it is ones intention to use only those points which most likely contain the most information and omit the rest.

One way to achieve this is to create average traces for different scenarios to be identified by means of templates later on. For instance, two different instructions or one and the same instruction with different operands. By calculating pairwise differences of these traces, points that contain information are then indicated by

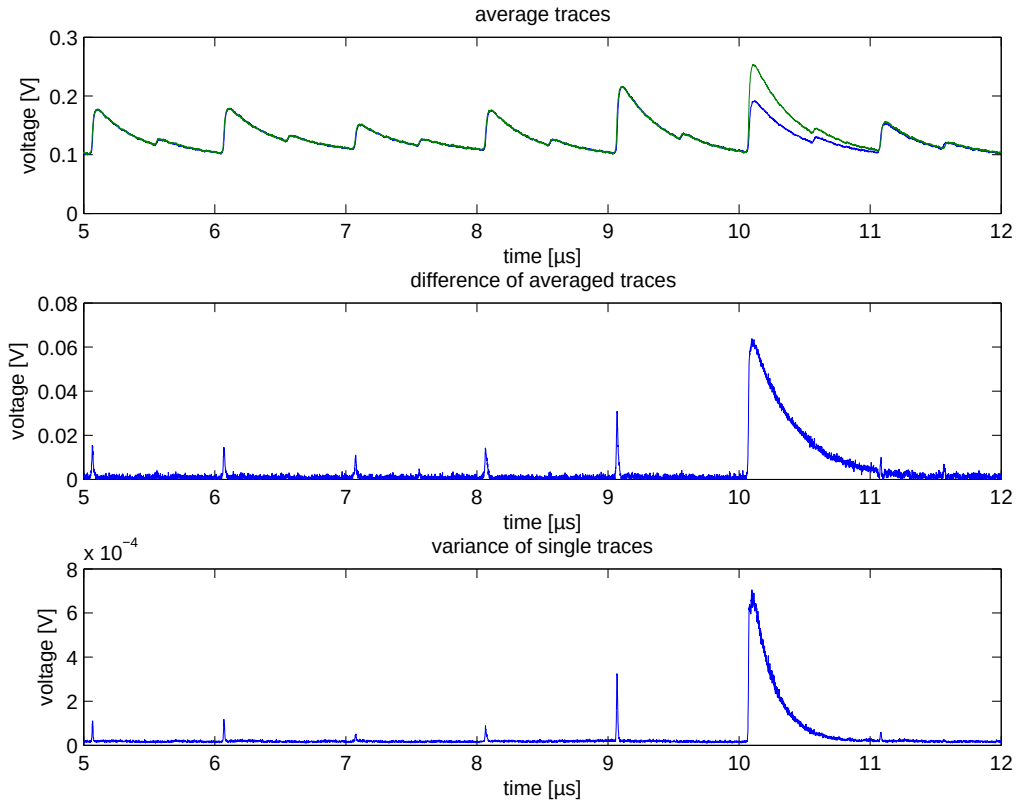


Figure 5.1: Two approaches for identifying points of information – the upper plot shows average traces for two scenarios, the center plot shows the difference of both, and the lower plot shows the variance taken from eighty measurements

peaks. In this case, points of information are points which help to identify the patterns a template was built for. Hence, building the difference is a good approach.

Another approach is to execute and measure code sequences of several scenarios in order to identify interesting points by checking the variances of each point. By this means, points that vary significantly more than due to noise can be identified similarly to the difference approach.

When following both methods for two instructions with randomly chosen inputs for every measurement we get results as presented in Fig. 5.1. The upper plot shows the average traces for the two scenarios. The corresponding difference is illustrated in the center plot. As can be seen, a significant difference occurs at $10\,\mu s$. Thus, points at this position are useful to determine which instruction was executed. Additionally, the lower plot shows the variance for both instructions and was calculated from eighty measurements (forty for each instruction). Obviously, this plot is qualitatively equal to the difference trace but has the advantage that

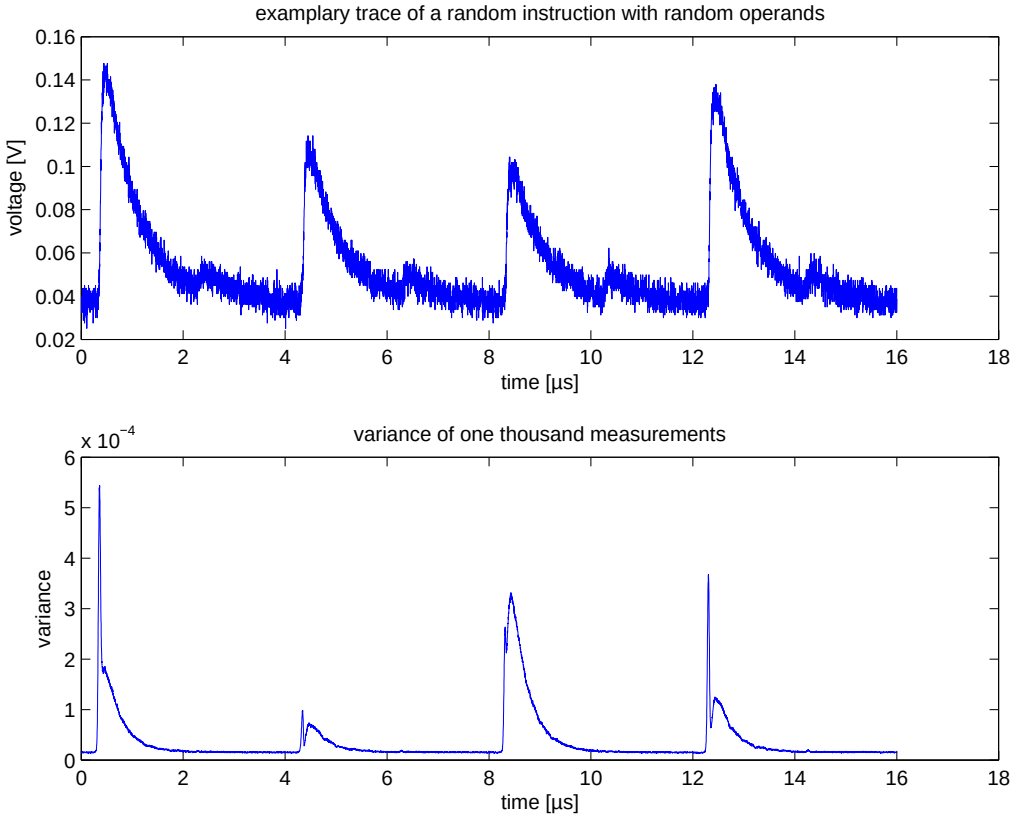


Figure 5.2: Identifying points of information by means of randomly executed instructions – the upper plot exemplary shows a measurement of a random instruction, the lower plot shows the variance of one thousand measurements

traces do not have to be sorted and averaged in order to create this plot.

Hence, to find out which points, in case of the PIC, generally leak information about the instruction, we execute random code sequences with random operands and calculate the variance of each point. Consequently, the points at which high variances occur constitute relevant points. Fig. 5.2 shows the results for one thousand measurements. The upper plot exemplarily shows one trace of a randomly executed instruction and the lower plot the variance of one thousand measurements. As can be seen, variances significantly higher than zero solely occur at rising clock edges. Consequently, only these points contain information. Moreover, the variances generally reproduce the shape of a real measurement in parts of rising clock edges but additionally show a bias at the position at which the peaks usually reach its maximum. Note, that the illustrated variances are in line with the provided power model, i.e. $Q2$ shows the smallest variance since this clock cycle is not affected by P_{bus} or $P_{prog.bus}$ which were identified to be the main influences.

Therefore, the most important points of a trace are most likely the maxima of $Q1$ to $Q4$. Another indication for this is the way the curve bottoms out. It seems to be defined by the height of the peaks as can be seen in Fig. 4.6. In any case, this may be a promising approach. In other approaches, the maxima plus a certain amount of subsequent points or down-sampled traces can be used. The tests presented in Section 5.4 have to prove which approach works best.

5.2 Partitioning

As we know from Section 2.1.2, the measured power consumption consists of the intrinsic power consumption P_{sig} and noise. However, in terms of instruction recognition, P_{sig} contains parasitic components which complicate the recognition process as we know from the last chapter. For instance, the power consumption of an instruction depends on the processed data. Hence, this component can be considered as noise. In general, the part of the power consumption which does not help to retrieve the wanted information is referred to as switching noise P_{sw} [MOP07]. Consequently, P_{sig} can be divided as follows:

$$P_{sig} = P_{op} + P_{sw} \tag{5.2}$$

In this equation P_{op} is the operation dependent power consumption we like to exploit and P_{sw} the component which summarizes all parasitic parts. In detail, these are the ones discovered in the previous chapter. At first, we have a data dependent component P_{data} caused by the operands which are processed. Then, there is a component P_{fetch} which is in conjunction with the subsequent instruction due to the fetching process. Further, the power consumption showed a dependency concerning the Hamming weight of the current instruction P_{inst}. Therefore, in the case of the PIC, P_{sw} can be written as

$$P_{sw} = P_{data} + P_{fetch} + P_{inst} \tag{5.3}$$

When creating templates, the regular approach is to simulate the electronic noise component of the power consumption by the covariance matrix. The question is whether the switching noise or its components can additionally be modeled by the covariance matrix together with the electronic noise. This can not be taken for granted, since templates model noise as a Multivariate-Gaussian distribution and the switching noise is not necessarily in line with this model. As a result, the distribution of the components of P_{sw} has to be estimated in order to answer the question.

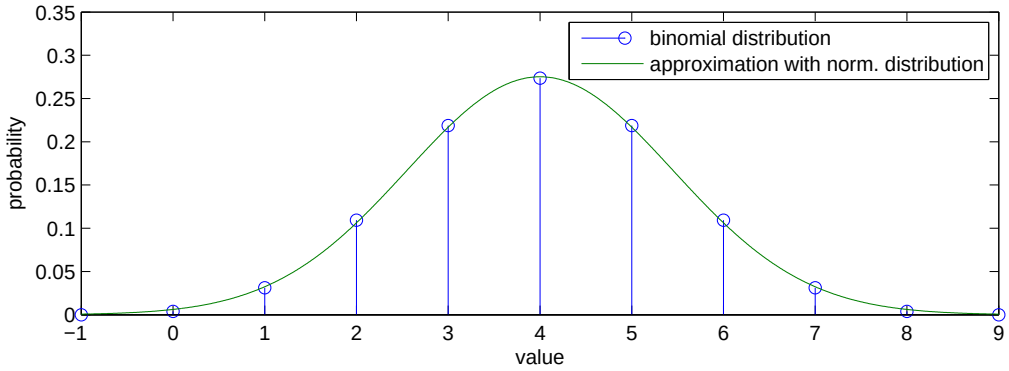


Figure 5.3: Approximation of a binomial distribution – binomial distribution (blue) and approximated normal distribution (green)

Due to the results of Chapter 4, we know that P_{fetch} and P_{inst} can be described by the Hamming-Weight model. To estimate the distribution for this model, we will consider the MOVLW instruction whose opcode is constant except for an 8-bit literal. We can assume that this literal is uniformly distributed in general, i.e. the probability to observe a literal of a certain value is 1 to 256. Consequently, when considering the Hamming weight of all 256 values, we get a binomial distribution. This is due to the fact that there is only $\binom{8}{0} = 1$ possible value for a Hamming weight of zero, $\binom{8}{1} = 8$ for a Hamming weight of one and so forth, as illustrated in (5.4).

$$256 = \binom{8}{0} + \binom{8}{1} \ldots \binom{8}{7} + \binom{8}{8} = 1 + 8 + 28 + 56 + 70 + 56 + 28 + 8 + 1 \quad (5.4)$$

Further, the Hamming weight is symmetrically distributed around a maximum and can therefore be approximated by a normal distribution as shown in Fig. 5.3. Hence, P_{fetch} and P_{inst} can be considered as normal distributed.

The data dependent power consumption component P_{data} occurs at $Q1$ and $Q3$ and is proportional to the Hamming distance of two values. If we again assume that these values are uniformly distributed, the Hamming distance like the Hamming weight can be considered as normal-distributed. This is because, given a bit of a value a, there is a probability of $1/2$ that the corresponding bit of a value b matches the bit of a. Hence, given an 8-bit value, the most frequent Hamming distance has a value of four, simply because the probability to match fifty percent of the bits is the highest.

Consequently, we can assume that P_{sw} and all its components can be modeled by a Multivariate-Gaussian distribution.

As a first naive approach, templates can therefore be created to model both, the switching and the electronic noise. In this case, we get one template for a certain instruction. The problem in this approach is that P_{sw} has to be modeled accurately in order to not become unfeasible. This means, that the test scenario has to guarantee that the executed instruction with all its influences is executed in a way that the distributions are represented correctly. For example, to model P_{inst} of a certain instruction every possible opcode has to occur in the correct amount, i.e. a Hamming weight in the mid-range has to occur more often than others.

In other approaches, successively fewer components of P_{sw} are described by the covariance matrix. As a consequence, one instruction is represented by several templates, one for each model specific case of the component that is not modeled in the template. For instance, if the noise model is supposed to include all components of P_{sw} except for P_{fetch}, this results in fourteen templates, one for each possible Hamming weight of a subsequent instruction.

In the extreme case that solely $P_{el.noise}$ is modeled by the covariance matrix one template is created for each combination of P_{data}, P_{fetch}, and P_{inst}. For example, one template for an instruction with a Hamming weight of x, a subsequent instruction with a Hamming weight of y and a Hamming distance between the operand and the result of z. Hence, when building templates this way, the number of measurements needed to get a sufficient amount of traces for each case increases significantly. Moreover, it becomes difficult to keep track of all parameters during the measuring phase. Above all, P_{data} is difficult to determine in this case, since it depends on the value on the data bus and the result calculated by the ALU, thus differing for every instruction.

Therefore, in this thesis, the following approaches will be followed. At first, we try to model P_{sw} completely by the covariance matrix. Second, we will partition the measurements made in the first approach by the values concerning P_{inst} and P_{fetch}, i.e. their Hamming weight. P_{data} will be modeled as noise in order to avoid the mentioned problems and thus simplify the template creation.

5.3 Test Procedure

In the last sections, the main criteria for template creation were discussed and theoretically promising approaches were presented. In the following section these approaches will be tested. But beforehand, the basic test procedure is described in this section.

A test, as can be seen in Test Description 5.3.1, starts with measuring power consumption traces for a certain instruction with respect to the approach that is followed in the test, i.e. which points are selected to create a template and further

which part of P_{sig} is modeled as noise. Depending on the test case, this results in one or more templates for one instruction. This procedure is repeated for a set of four instructions, namely NOP, CLRW, XORLW and ANDLW. In this way, we have two instructions that do not work by means of operands provided in the opcode. More precisely, NOP naturally contains the lowest amount of switching noise. Solely the fetching process and the contents of W influence the power consumption. CLRW deletes the contents of W and is therefore more data dependent. With ANDLW and XORLW, we have two instructions processing two operands. Hence, the switching noise is significantly higher. Further, the instruction identifier of XORLW and ANDLW have the same Hamming weight so that we are able to determine whether this affects the success of instruction identification or not. Note, that in the following tests no file-register instructions are included. However, these work similarly to the literal instruction so that the results of ANDLW and XORLW can be applied to file-register operations.

To check, whether the templates are practical to recognize instructions, five sequences containing fifty instructions each are randomly selected consisting of only those instructions for which templates have been created. The operands that occur in the sequences are also chosen randomly. Subsequently, these sequences are executed and measured for one hundred times in a random order to exclude time dependent interferences. This means, when the amount of noise, for some reason, is significantly higher for a certain time during a measuring session, this will affect all test sequences uniformly. Which sequence was executed at a given time was stored together with the measurement file so that the executed instructions for a certain measurement are known later on.

Finally, the templates are applied to the instructions of the one hundred test sequences one by one in order to determine whether the winning template corresponds to the executed instruction or not. Thereby, the percentage of correctly and incorrectly recognized instructions can be calculated indicating the quality of the constructed templates.

Test Description 5.3.1 (General Test Procedure for Templates) *To be able to evaluate the quality of templates the following test is performed:*

1. *measure traces with respect to the followed approach in order to create templates for four different instructions*
2. *measure one out of five test sequences, each containing fifty randomly selected instructions with random operands, for one hundred times*
3. *apply templates to the measured instructions of all test sequences and determine the template with the highest probability*
4. *calculate the percentage of correctly and incorrectly determined instructions*

5.4 Tests

In this section various approaches for creating templates for the PIC will be tested and compared in order to reveal the best method.

5.4.1 Single Templates

As a first approach, P_{sw} will be modeled completely by the covariance matrix. In order to create a template for a certain instruction in this way, its power consumption during execution has to measured for several times with random operands, i.e. literal and W, and the subsequent instruction to cover the influences of P_{fetch}, P_{inst} and P_{data}. For this test five thousand measurements were recorded per instruction. From these traces, the actual instruction was selected and the four maxima for $Q1$ to $Q4$ were extracted. Then, the covariance matrix $\mathbf{C}$ and the mean vector $\mathbf{m}$ were calculated forming the template for this instruction.

The created templates were then applied to the random test sequences. This means, the logarithmic probability density function given in (2.26) was calculated for each observed instruction of the test sequences in order to find the template with the highest probability, i.e. lowest value. The instruction this template was created for was then chosen as the recognized instruction. Note that the logarithmic function was used, since (2.25) showed numerical problems, i.e. inverting $\mathbf{C}$ resulted in a close to singular matrix.

Instruction	Recognized as				Sum	Percentage
	NOP	CLRW	ANDLW	XORLW		(correct)
NOP	0	11	207	1020	1238	0
CLRW	0	0	422	782	1204	0
ANDLW	0	0	520	826	1346	38.6
XORLW	0	0	210	1002	1212	82.6
				Overall	5000	30.4

Table 5.1: Results for applying templates, created by modeling P_{sw} completely, to random code sequences

The results of the test are shown in Tab. 5.1. The first column contains the instructions for which templates have been created and which were thus executed in the test sequences. The next four columns show the number of how many times an instruction was recognized as one of the others. Further, the next column presents

the sum of executed instructions and finally the last column indicates the percentage of correctly identified instructions for each instruction independently as well as accumulated for all instructions.

Obviously, the templates for the literal instruction ANDLW and XORLW cover the other templates so that the instruction recognition shows a clear tendency towards these. Actually, NOP and CLRW were not recognized correctly in a single case. Moreover, the accumulated result shows that in total only thirty percent of the instructions were correctly identified. Since the test sequences only contain four instructions, namely the ones templates were created for, the chance to guess an instruction is twenty-five percent. Hence, the results show that our templates are not significantly better than guessing an instruction and therefore unfeasible.

Thus, the test did not show any indication that modeling all components of P_{sw} in one template is practical.

5.4.2 Single Templates with Constant P_{fetch}

According to the results of the previous section, a different approach will be followed in which the fraction of components modeled by the covariance matrix is reduced. By this means we are able to evaluate whether a noise reduction, although just simulated, yields better results.

For this, the amount of switching noise is reduced by avoiding influences of P_{fetch}. This is done by fixing the subsequent instruction to NOP, i.e. a constant instruction, which results in a constant P_{fetch}. Then, again, measurements are acquired and templates are created. Furthermore, the test sequences also contain a NOP in one out of two instruction to work on the same basis. This also means that the templates are applied to every other instruction of a code sequence.

The results are illustrated in Tab. 5.2. With an overall recognition success rate of 70.4 percent, this test yields significantly better results than the previous one. In the first row of the table we see that NOP is the best recognized instruction with over 88 percent. Only a few incorrect interpretations occur for CLRW, ANDLW and less significantly for XORLW. The instruction CLRW additionally shows a high success rate of approximately 79 percent. In contrast to this, ANDLW and especially XORLW have lower success rates. Please note that XORLW and ANDLW have the same Hamming weight in the instruction identifier of the opcode. Hence, we can assume that when using literals with the same Hamming weight, we can expect that P_{inst} is equal in both cases. Consequently, these instructions most-likely show a very similar power consumption. This is actually the case leading to misinterpretations. This mainly applies for XORLW, which has only a success rate of approximately 45 percent and is in almost any case mistaken for a ANDLW. Nonetheless, both instructions can still be distinguished to a certain degree.

Instruction	Recognized as				Sum	Percentage (correct)
	NOP	CLRW	ANDLW	XORLW		
NOP	503	19	39	5	566	88.8
CLRW	50	547	56	34	687	79.2
ANDLW	0	15	408	146	569	71.7
XORLW	0	9	366	303	678	44.6
				Overall	2500	70.4

Table 5.2: Results for applying templates, created with a constant subsequent instruction, to random code sequences

When comparing the success rates, it can further be seen that the success rate decreases with an increasing amount of switching noise. As already mentioned, for a NOP instruction P_{sw} only depends on the contents of W and the instruction being fetched. Since this instruction is fixed during these tests, the amount of switching noise is minimal. For CLRW, P_{sw} is increased due to clearing the contents of W. Considering the last two instructions, P_{data} becomes a dominant factor as can be concluded from the discussed power model presented in Definition 4.10.1. Thus, the amount of switching noise is maximized.

Consequently, reducing the switching noise is a reasonable method to increase success rates significantly.

5.4.3 Partitioned Templates with Constant P_{fetch}

As was discovered in the last section, reducing the amount of switching noise modeled by the covariance matrix results in better success rates. Hence, as a next step, switching noise can further be reduced by partitioning the templates according to the contents of W, which is randomly defined before every iteration of the measuring process.

For this, the measurements of the previous test can be used, since the value of W was stored with each measurement. To be able to create templates, the set of measurements for one instruction is partitioned according to the Hamming weight of W. As we know from Section 4, sorting according to the Hamming distance may be the better approach, but the problem is the Hamming distance's dependence on the additionally used operand and the result of the previous instruction. Hence, sorting this way becomes complicated so that the Hamming weight is applied for simplicity reasons.

For each partition one template is then created. This leads to nine templates

per instruction due to nine different Hamming weights. These templates can then again be applied to the test sequences. As in the previous tests, the template with the lowest absolute value of the logarithmic probability density function is the most likely one.

Instruction	Recognized as				Sum	Percentage (correct)
	NOP	CLRW	ANDLW	XORLW		
NOP	471	12	59	24	566	83.2
CLRW	22	449	57	159	687	65.3
ANDLW	0	1	467	101	569	82.1
XORLW	0	2	176	500	678	73.7
				Overall	2500	75.5

Table 5.3: Results for applying templates, created with a constant subsequent instruction and sorted by the Hamming weight of W, to random code sequences

The results of the test are shown in Tab. 5.3. With the new approach even better results can be achieved. The overall success rate is increased by approximately five percent compared to the previous test in which templates were not partitioned. Furthermore, it can be seen that the success rates for ANDLW and XORLW are increased significantly. Particularly, the chance to identify an XORLW instruction was increased by 19 percent, from 44 to 73.1 percent, whereas the chance for ANDLW was only increased by approximately eleven percent. This is mainly due to the fact that XORLW is significantly less mistaken for an ANDLW. In contrast to this gain, the success rate of CLRW and NOP is decreased with the new approach. Above all, CLRW is more often confused with ANDLW reducing the success rate by 14 percent. The percentage of correctly identified instructions for NOP is only slightly decreased by approximately five percent.

As a result, the fractions of correctly identified instructions are rearranged when using partitioned templates. However, the overall success rate is increased so that this approach is an effective way to to achieve better recognition results.

5.4.4 Optimizing Point Selection

The next step is intented to increase the overall success rate by a better selection of points. Note, that selecting the maxima of $Q1$ to $Q4$ was only a first approach based on the observations made in Section 5.1.

Therefore, the maximum number of points has to be revealed that can be handled by the analysis program in terms of the logarithmic probability density function. For this, some tests based on the measurements, recorded for the previous tests, were performed. It turned out that the most critical part of equation (2.26) occurs within the natural logarithm

$$ln(2\pi^n \cdot det(\mathbf{C})) \tag{5.5}$$

because the value of $2\pi^n \cdot det(\mathbf{C})$ causes overflows for large n of around 180 and larger. Note, that this value is only a rough approximation since equation (5.5) also depends on the determinant of $\mathbf{C}$. Hence, even values smaller than 180 could result in overflows.

However, if acting on the assumption of 180 points, the following tests can be performed to reveal the best selection of points.

At first, an equal amount of points from $Q1$ to $Q4$ can be selected resulting in a range of 1 to 45 points per clock cycle. These can either be symmetrically or asymmetrically distributed around the maxima of the peaks. Hence, different point ranges will be tested for both symmetrical and asymmetrical approach. However, the step size for the asymmetrical approach will be shorter and in the direction of the consecutive peaks, since the way the peaks bottom out is more likely to contain information than the rising edge of the peak.

Furthermore, the entire trace of an instruction can be down-sampled from eight thousand points to one hundred and eighty points or less in order to get a template that is equally based on the power consumption of the complete instruction. More precisely, down-sampling to x points can be achieved by selecting point 1, $1 \cdot \lfloor \frac{8000}{x} \rfloor$, $2 \cdot \lfloor \frac{8000}{x} \rfloor$ and so forth. With this approach, we may be able to find out whether the primary approach of selecting points from the peaks is reasonable.

As in the previous tests, the quality of the templates will be given by the overall success rate which arises from applying the templates to the test sequences. Herein the templates are built according to Section 5.4.2, i.e. the subsequent instruction is fixed but not sorted by the Hamming weight of W.

The results are shown in Tab. 5.4. In the first column, the selection type is presented which is either symmetrical, asymmetrical or down-sampled. The next column specifies the variations of selected points for each selection type. Thereby, the first number correspond to the number of points taken from the left side of the maximum and the second number to the points taken from the right side of the maximum. In case of the down-sampled template only one number is provided as the overall number of points used. Finally, columns three to seven illustrate the percentage of correctly identified instruction for each instruction independently as well as the accumulated results.

As can be seen, the overall results are quite similar and distributed in the range of 70 to 77 percent. The first row contains the reference values for this test, since

in this case only the maxima of the peaks are selected. Obviously, increasing the number of points selected next to the maximum increases the success rate slightly. However, for values larger than eleven in the case of a symmetrical points selection, and thirty-one in case of an asymmetrical point selection, this gain is already lost.

Selection Type	Points per Peak (left/right)	Percentage (correct) NOP	CLRW	ANDLW	XORLW	Overall
symmetrical	0/0	88.8	79.2	71.7	44.6	70.4
	2/2	90.1	82.7	78.2	59.4	77.0
	5/5	89.9	81.2	77.9	60.3	76.8
	10/10	90.3	78.1	79.2	62.2	76.8
	16/16	89.2	71.4	79.3	62.5	74.8
	22/22	87.6	63.0	78.7	62.4	72.0
asymmetrical	0/5	90.1	83.0	78.2	57.2	76.5
	0/15	90.0	81.2	78.0	61.6	76.7
	0/30	88.3	75.1	78.6	61.3	75.2
	0/44	86.4	66.2	76.6	58.0	71.0
down-sampled	80	86.3	79.1	75.3	48.5	72.3
	180	85.7	71.0	73.1	62.7	73.1

Table 5.4: Results for various points selection methods

The maximum success rate is reached for a symmetrical selection of only two points next to the maximum of a peak, which results in a mean vector of twenty points per template. Similar results are shown by the asymmetrical point selection in which the best overall success rate reaches 76.7 percent.

This increase is probably caused by reducing noise at the maxima by means of surrounding points. However, when the number of points exceeds a certain level this gain is lost due to numerical problems.

When comparing the single success rates for each instruction, we see that the values for NOP and ANDLW are rather constant, only varying in a range of four percent, whereas CLRW and XORLW show deviations of up to twenty percent. Hence, CLRW and XORLW are more significantly affected by the selected points.

Interestingly, even down-sampling yields good results of 72.3 and 73.1 percent respectively. The reason for this is not obvious. One assumption is that the shape of the trace is reproduced, included the maxima and their relationships, so that even if the maximum values are not reached completely due to the fact that the absolute maxima are not sampled, it may be sufficient to identify an instruction.

However, selecting points around the maxima is more practical. Not only because the success rates are higher but furthermore because the complexity for creating and applying templates is reduced. For instance, in the best approach of Tab. 5.4, only 20 points are sufficient to create a template. Hence, the covariance matrix is reduced by a factor of 16 compared to the case of down-sampling to 80 points and by a factor of 81 when down-sampling to 180 points.

To test whether the approach of the previous section is equally affected by a different point selection, the most promising method of selecting the maxima plus/minus 2 points was applied for creating templates which are additionally sorted by the Hamming weight of W. In this case, the success rate was increased from 75.7 to 82.5 percent, which is approximately the same gain as for the other approach.

In conclusion, the point selection approach which was initially used turned out to be a good choice. However, it can be improved by adding some points around the maxima. Above all, selecting five point per clock cycle was the most successful technique.

5.4.5 Reduced Templates

Up to this point we discovered that extracting a small number of points from the peaks is a good approach for template creation for the PIC. Additionally, partitioning templates yields better results. In this section, we want to evaluate the contribution of the covariance matrix has in terms of instruction recognition.

For this, we will repeat the tests of the last section but by means of reduced templates. The logarithmic probability density function is then calculated without the use of the covariance matrix. In other words, $\mathbf{C}$ is set to the identity matrix, which results in equation (2.28).

The results are summarized in Tab. 5.5. As can be seen, the overall success rates are significantly decreased for all selection types. The highest value is reached for the symmetrical point selection with 45 points. However, the other values are very close to this. Solely the success rates for the down-sampled cases is even more decreased to only 58 and 59 percent respectively.

Furthermore, we see that the success rates for NOP and CLRW are significantly higher than for ANDLW and CLRW. Obviously this tendency is in line with the amount of switching noise of an instruction. As already mentioned, the NOP instruction only depends on the contents of W. Hence, this instruction is better recognized as the ANDLW instruction, which also depends on the operands, the calculated results and so forth. This is not unexpected since the covariance matrix models this noise. Therefore, when omitting this part in the probability density function as in this case, instructions which highly depend on the switching noise are less often recognized correctly.

Selection Type	Points per Peak (left/right)	Percentage (correct) NOP	CLRW	ANDLW	XORLW	Overall
symmetrical	0/0	94.5	81.0	56.0	22.4	62.5
	2/2	95.0	85.0	55.2	23.2	63.7
	5/5	95.6	85.6	55.5	23.6	64.2
	10/10	95.4	86.5	55.9	23.6	64.4
	16/16	95.4	86.5	55.9	23.6	64.5
	22/22	95.8	85.9	56.4	23.3	64.7
asymmetrical	0/5	95.2	86.3	55.5	22.8	64.1
	0/15	95.4	86.2	55.5	23.5	64.3
	0/30	95.6	84.9	54.7	22.9	63.6
	0/44	96.1	85.3	55.0	23.3	64.0
down-sampled	80	89.4	73.4	54.0	19.9	58.0
	180	91.5	81.0	48.9	19.8	59.4

Table 5.5: Results for various points selection methods and the use of reduced templates

As a result, the covariance matrix constitutes an important part of our templates. Omitting it significantly reduces the success rate, especially for instructions which are highly influenced by the switching noise.

5.4.6 Peak Selection

In this section it will be tested whether better results can be achieved by selecting only a subset of the peaks for creating templates. The decision needs to be made which peaks to select and which to neglect. The first approach is to select only those peaks showing a comparatively small variance. An examination of the the variances of the template created as explained in the previous section, showed that the variances for $Q1$ to $Q4$ for NOP are comparatively small. In contrast to this, CLRW shows a significantly higher variance at $Q1$ than at $Q2$ to $Q4$. Furthermore, ANDLW and XORLW show high variances concerning $Q1$ and $Q3$ when the Hamming weight of W reaches a mid-range value. This is due to the fact that $Q1$ and $Q3$ are predominantly determined by the Hamming-Distance model, which differs from the Hamming-Weight model in these cases but not as significantly for small or large values, in which the Hamming weight is often equal to the Hamming distance.

Hence, the following peak selection methods will be applied. First, $Q2$ to $Q4$ will be taken into account to avoid influences of $Q1$. Then, $Q1$ and $Q3$ will be omitted

to avoid the high variances for ANDLW and XORLW. In a next step this selection is inverted to reveal whether omitting peaks with high variances is a good approach.

The discussed methods will be tested by creating templates according to the previous section, i.e. five points are selected from each peak and the templates are further partitioned by the Hamming weight of W. This was the best approach so far with a success rate of 82.5 percent. The results of the tests can therefore be compared to this value in order to measure the quality of the different approaches.

Selected Clock Cycles	Percentage (correct)				
	NOP	CLRW	ANDLW	XORLW	Overall
$Q1$ to $Q4$	90.3	82.3	86.1	73.3	82.5
$Q2$ to $Q4$	95.2	75.1	73.1	68.1	77.3
$Q2$ and $Q4$	97.5	60.7	71.2	68.3	73.5
$Q1$ and $Q3$	23.9	30.9	73.0	64.0	47.9

Table 5.6: Results for various peak selection methods

Tab. 5.6 presents the results of the performed tests. As can be seen, the success rates vary in a range from 47.9 to 82.5 percent. The highest percentage of correctly recognized instructions occurs for the primal case, i.e. selecting $Q1$ to $Q4$. Furthermore, we see that $Q2$ and $Q4$ contain more information for instruction recognition than the others since the success rate for these cycles is significantly higher than for $Q1$ and $Q3$ in which, above all, NOP and CLRW are seldom recognized correctly. Nonetheless, omitting $Q1$ and $Q3$ causes a decrease in success rate for CLRW, ANDLW and XORLW respectively.

Hence, we can conclude that omitting the points of one or more clock cycles causes a loss of information and thus results in an decreased overall success rate.

As a consequence, adding peaks of the previous instruction may be a promising approach since additional information about the executed instruction, i.e. its fetching process, would then be usable. However, tests in this respect resulted in success rates similar to the first approach of modeling P_{sw} completely. This can be explained by means of the established power model from which follows that P_{Q1} to P_{Q4} are mainly influences by the execution of an instruction. Hence, the fraction of P_{fetch} is comparatively small.

5.4.7 Partitioned Templates

In the last tests the subsequent instruction was held constant in order to avoid influences of P_{fetch}. Nevertheless, methods were discovered to improve the overall

success rate. These will now be applied and extended to the general case in which the subsequent instruction is not fixed.

For this, the templates will be partitioned by the Hamming weight of W as in the previous sections but further by the Hamming weight of the subsequent operation and the Hamming weight of the opcode of the executed instruction. By doing this, we get more accurate templates, which has turned out to be a probable approach throughout the last sections. Further, points of all clock cycles, i.e. $Q1$ to $Q4$, will be extracted by selecting five points per cycle. Hence, one template consists of a mean vector of twenty points and a covariance matrix of four hundred points.

Due to the partitioning process, the number of templates for one instruction increases and with it the number of measurements which have to be recorded in order to get a sufficient amount for each partition. For instance, by partitioning the templates as described we will get

$$num_{measurements} = 9 \cdot 9 \cdot 15 \cdot 50 = 60750 \tag{5.6}$$

measurements to create all 1215 templates for a literal operation if each template is created from only fifty measurements. This is because there are nine different Hamming weights for W, nine different Hamming weights the opcode can have due to the literal, and fifteen different Hamming weights of the subsequent instruction. Note, that thus the number of measurements is not fix. A NOP for example has a constant opcode so that in this case only 6750 measurements are required to create all templates.

To test the templates, these were again applied to random test sequences using random operands, i.e. the logarithmic probability function was calculated for each template. The template resulting in the lowest value corresponded to the recognized instruction. Unfortunately, the results showed that almost all instructions were mistaken for XORLW instructions. Hence, the success rate was approximately 25 percent, similar to the approach discussed in Section 5.4.1.

To exclude that this is caused by the extended partitioning approach, the test was repeated with only sorting templates according to the Hamming weight of W and by the Hamming weight of the subsequent instruction which has shown high success rates for a fixed subsequent instruction. However, this also leads to the same results. Obviously, the set of templates for XORLW covers the other instructions which was not the case for a fixed subsequent instruction.

This can be explained by means of the power model presented in Definition 4.10.1. If we omit the common components $P_{const,Qx}$ and $P_{el.noise}$, which are independent of the performed instruction, we see that the power consumption highly depends on several Hamming distances and further on the Hamming weight of the current and subsequent instruction. Concerning the Hamming weight we can expect that

these values are not appropriate indicators for an instruction since the Hamming weight can vary significantly for instructions that use literal of file-register addresses. Furthermore, P_{Q2} and P_{Q4} lead to the conclusion that fetching an XORLW and executing a NOP for example causes the same power consumption when neglecting the effects of IR in $Q4$. Furthermore, $Q1$ will show high variances since v_{bus} is predefined randomly for every measurement and was unaccounted for regarding template partitioning to keep the number of measurements practicable. In addition, $Q3$ highly depends on the v_{result} calculated by the ALU which may be the reason why ANDLW is mistaken for an XORLW. This is a result of the assumption that the variance at $Q3$ is higher due to the fact that the probability to flip a bit during an XOR operation is $1/2$ in contrast to $1/4$ in case of ANDLW.

In consequence, the instruction recognition for the general case highly depends on the power consumption influences not explicitly revealed, i.e. $P_{ext.Qx}$. Obviously, this component does not contain sufficient information so that the chance for one of the 1215 templates of XORLW or ANDLW being mistaken for one of the 135 templates for NOP and CLRW is remarkably high.

This leads to the conclusion that a general instruction recognition by means of templates is not feasible for the device under attack, at least, if only single measurements are taken into account.

5.5 Summary and Conclusion

In this section various methods for creating templates were covered.

As a first step, the template creation process was once more examined theoretically but with respect to the device used. This resulted in two approaches. In the first one, the selection of points was considered. Due to the high variances at the points at which $Q1$ to $Q4$ reach their maxima, we determined these points to be the points to start with for further tests. Furthermore, in the second approach, the partitioning of templates was discussed and resulted in the assumption that the switching noise or part of it may be modeled by means of the covariance matrix.

Thereupon, practical tests were performed. The original idea of selecting points of the maxima of the peaks turned out to be an adequate approach. However, by selecting four additional points from each peak the success rates could be increased.

In contrast, the idea to model the switching noise by means of the covariance matrix turned out to be inappropriate. Feasible results could only be accomplished when the subsequent instruction was fixed to get a constant P_{inst}. Further improvements were achieved by partitioning templates according to the Hamming weight of W. In contrast to this, omitting the covariance matrix caused a loss of information and manifested in decreased success rates. According to this, it is reasonable

to partition templates in order to reduce the amount of switching noise which is modeled by the covariance matrix. This approach was tested for the general case in which the subsequent instruction was not fixed. Unfortunately, this approach did not result in feasible success rates.

In conclusion, the performed tests did not indicate that instruction recognition for the PIC is practicable by means of templates and under the premise that only single measurements are used and no a priori information is available. Hence, instruction recognition of unknown code does not seem to be feasible in this respect.

However, if a priori information is available, the success rate for recognizing instructions can be improved, as was shown by the tests performed by means of a fixed subsequent instruction. Consequently, we may be able to detect program paths of known code by applying suitable templates to the expected positions of single measurement traces.

Furthermore, the results calculated by an instruction may be identified when partitioning templates according to all possible values or more precisely to the Hamming distance of $v_{operand}$ and v_{result} as can be derived from the power model. This can, for instance, become useful when the implementation of a cryptographic algorithm is known except for some secret S-Boxes often implemented as Look-Up-Tables. In this case, templates created for the instruction performing this task can be applied to the parts of the power consumption at which the look-up is performed in order to reveal the Hamming weight of an entry as an intermediate step for revealing the entire table. Another way to achieve this would be to use the introduced power model in order to correlate power traces to hypothetical power consumptions for each possible Look-Up-Table.

In addition, if more than one measurement can be recorded for analysis, averaging several traces may be useful to reduce the high data dependency in order to enhance the instruction recognition. However, this may only be appropriate for highly regular code in which no conditional branches are taken so that one and the same kind of instruction is used for averaging at all times. Nonetheless, this is not an unlikely scenario since cryptographic algorithms are often implemented in such a way to avoid time dependencies.

As a result, although instruction recognition without a priori information is not practicable several applications may nevertheless be feasible. Due to time constraints, solely one of these – namely the path detection of known program code – will be analyzed in more detail in the next chapter.

6 Path Detection

In the last section we created templates for the sake of instruction recognition. Although the tests indicated that this does not work properly without additional knowledge, templates may nevertheless be used to recognize the path a program took at a time by utilizing a priori information, i.e. assuming the program code or parts of it to be known.

To examine this, the first section discusses the basic idea about how path recognition can be applied for the PIC microcontroller. Further, working conditions and application areas are presented. The subsequent section deals with the algorithm that can be used for program path recognition. The functionality of the presented algorithm is then proved by practical tests and the results are presented.

6.1 Basic Idea

As we have seen in the last section, when creating templates for the general case of instruction identification, recognizing instruction becomes difficult. This is mainly caused by the fact that no information is available on the executed instruction other than the power consumption trace. Consequently, all templates have to be applied to this trace in order to find the one with the highest probability which resulted in mistakes. However, we can assume that the values calculated by means of the logarithmic probability density function are in average smaller in cases when a certain template is applied for the right instruction with the right conditions, i.e. Hamming weight of the subsequent instruction etc., than in cases when an inappropriate template is applied.

Hence, when using additional information and chained probabilities, the recognition process can most-likely be improved. As information source we will exploit the program code itself in this chapter to perform path detection. In this scheme it is the goal to identify which path the program took at a given time, i.e. which conditional branches were taken and which not. By doing this, we are able to get a better design level understanding of the program which may be useful in the case that the program code was somehow revealed but it is now known how the program works in detail. Another application area for side-channel based path recognition is debugging in the field which can become useful, when some parameters like for

example analog inputs are not known. Or it can even be used to reveal the program version currently running on the device.

To perform path recognition, the program code, as already mentioned, needs to be known. With this source of information, possible paths can then be calculated from a certain starting point. Moreover, the opcode for each instruction is known since the literal and file-register addressed are included in the code. Solely the content of the registers are generally not known since they depend on the inputs of the performed algorithm which can not necessarily assumed to be known. With this information we are able to apply the appropriate templates for each possible path to a given power consumption trace. The calculated values can then be summed up for each path. When using the logarithmic probability density function, the overall smallest value corresponds to the path with the highest probability. To understand how this works, an example will now be presented.

Listing 6.1: Example A for path detection

```
1 CODE1 : NOP, ADDLW 0x55 , NOP, ADDLW 0x55 , NOP
2 CODE2 : ADDLW 0x55 , NOP, ADDLW 0x55 , NOP, ADDLW 0x55
```

The assembler code given in Listing 6.1 contains two code sequences, namely CODE1 and CODE2, each containing five instructions. Both alternate NOP and ADDLW instructions but in a complementary order. When one of these code sequences is executed and measured, the probability for CODE1 is calculated by applying the template for NOP to the power consumption of the first instruction cycle, the template of ADDLW to the power consumption of the second instruction cycle and so forth. When using partitioned templates, only those templates matching the basic conditions are applied. For instance, only the template for the NOP instruction created under the premise that the subsequent sequence has a Hamming weight of seven would be used because seven is the Hamming weight of ADDLW `0x55h`. The results of the logarithmic probability density function are then summed up. By doing this we get the chained probability to observe this sequence simply because it is the sum of the single probabilities. This process is then repeated for CODE2. As a consequence, the code sequence with the smallest value has the highest probability. Again, if the appropriate template in average yields better results than an inappropriate templates a code sequence of sufficient length will be recognized correctly.

An important factor for the success of this method is that correct templates have to be applied to the correct instruction cycle of the power trace. This is not as trivial as one might think. As can be seen in Listing 6.2, in this case, CODE1 and CODE2 both consist of the same code sequence. Nonetheless, these sequences can be distinguished because the respective part of the power consumption differs for both sequences. The explanation for this is as follows:

Listing 6.2: Example B for path detection

```
1 BTFSC 0x40h , 1
2 CALL CODE1
3 CALL CODE2
4
5  CODE1 : NOP, ADDLW 0x55 , NOP, ADDLW 0x55 , NOP, RETURN
6  CODE2 : NOP, ADDLW 0x55 , NOP, ADDLW 0x55 , NOP, RETURN
```

The BTFSC is a conditional branch instruction which reads the first bit of file-register `0x40h` and then executes CODE1 if this bit is set and CODE2 otherwise. If the condition is true, then BTFSC takes two instruction cycles because in this case the next instruction is skipped which means that it is executed as a NOP as explained in Section 3.2. If the condition is not true, the next instruction is executed as given. Hence, if the conditional branch implies the execution of CODE1, this sequence starts with the fifth instruction cycle and with the fourth otherwise. Therefore, the respective parts for the executed instruction have different positions in the power consumption trace, which has to be kept in mind when applying the templates.

In conclusion, it is thus crucial to determine the correct part of the power consumption for a certain instruction.

6.2 Algorithm

Due to the results of the previous section, an algorithm can be defined in order to detect program paths from a side-channel. Note, that this algorithm does not depend on the underlying hardware so that it can be applied to other microcontrollers as well.

The preliminaries to run the algorithm are as follows. At first, a set of templates has to be created for several instructions. For a better success rate these templates should be partitioned according to the main influences on the power consumption which have to be analyzed in advance. As in the case of the PIC, these where for example the Hamming weight of the subsequent instruction and the Hamming weight of the instruction itself. Operand-dependent influences do not have to be taken into account since these dependencies can not be concluded from the assembler code. When templates for N instructions are created with M templates for each instruction, then these can be stored to a $N \times M$ matrix $\mathbf{T}$ to function as an input of the algorithm.

Secondly, the program code or at least the part for which the path detection is supposed to be performed has to be known in order to calculate possible paths. This does not have to be necessarily the assembler code. Even the binary representation

of this code can be used. However, in this case the binary code has to be translated back to the assembler code as an intermediate step.

If these two preliminaries are given, the algorithm described as in Algorithm 6.2 can be performed.

Algorithm 6.2.1 (Path Detection) *Given a set of templates $\boldsymbol{T}$ for n instructions and m basic conditions, a recorded trace $\boldsymbol{R} = (r_1, ..., r_k)$ of k instruction cycles, and an assembler code $\boldsymbol{A} = (a_1, ..., a_l)$ of l instructions, the most-likely path which was taken at the time of recording $\boldsymbol{R}$, can be detected as follows:*

1. *Calculate all hypothetical paths up to a length of k instruction cycles from $\boldsymbol{A}$.*
2. *Select only those instructions from the hypothetical paths for which templates have been created and find the best matching template $m_{i,j}$ of $\boldsymbol{T}$ for each instruction j and path i, with respect to the program code.*
3. *Determine the corresponding instruction cycle $r_{i,j}$ for each instruction cycle j of each path i. If an instruction takes more than one instruction cycle, select the first one and handle the second instruction cycle as a NOP.*
4. *For each of x hypothetical paths this leads to a $2 \times d$ matrix $\boldsymbol{H}_i$, $i = 1, \ldots, x$, which contains the best matching templates for the respective path together with the expected positions in $\boldsymbol{R}$:*

$$\boldsymbol{H}_i = \begin{pmatrix} r_{i,1} & r_{i,2} & \ldots & r_{i,d} \\ m_{i,1} & m_{i,2} & \ldots & m_{i,d} \end{pmatrix} \tag{6.1}$$

5. *For each hypothetical path $\boldsymbol{H}_i$ calculate its overall logarithmic probability:*

$$p(H_i) = \frac{1}{n} \sum_{n=1}^{d} |\ln(p(r_{i,n}, m_{i,n}))| \tag{6.2}$$

6. *calculate the winning path w with*

$$w = \{H_i \,|\, p(H_i) = \min(p(H_i))\} \tag{6.3}$$

In the first step, all hypothetical paths up to a length of k instruction cycles are calculated. Obviously it makes no sense to generate longer paths since the recorded trace only contains the power consumption of k instruction cycles. Thus, additional instruction cycles can not be evaluated by means of $\mathbf{R}$.

Then, in the second step, all irrelevant instructions are omitted, whereby irrelevant means irrelevant in terms of path detection. Moreover, the best matching template is determined for each of the non-omitted instruction. These templates will later be used to calculate the probability for a certain hypothetical path.

From this it follows that not all templates have to be known in order to apply the algorithm, i.e. we can start with a few templates and adjust the model step by step by adding more templates. As a result, the algorithm is quite flexible in this respect.

Moreover, it is useful to predefine the best matching templates on basis of the program code **A** for complexity reasons. By doing this, templates are determined only once and not later on in a loop for each hypothetical path.

In step 3, for each instruction of each hypothetical path the corresponding part of the recorded trace is determined. This means that it is calculated at which instruction cycle one instruction should occur for a given path. In case of a two cycle instruction, the first instruction cycle is selected. The second one can be handled as a NOP since this is the instruction that is executed during the second instruction cycle.

This information together with the templates of step two is written to a matrix H_i for each hypothetical path i. Thereby, H_i contains the positions $r_{i,j}$ of in the power consumption trace together with the most accurate template $m_{i,j}$ for each instruction j of this. Thus, all information needed to calculate the overall probability for each hypothetical path is given and can be computed in the next step.

The overall probability to observe path H_i is calculated in step 5 by computing the arithmetic mean over all single-probabilities as illustrated in equation (6.2). In this equation $|\ln(p(r_{i,n}, m_{i,n}))|$ represents the logarithmic probability density function for the Multivariate-Gaussian model as defined in (2.26). The mean is computed instead of the sum because the length d may differ for the various paths since the number of relevant instructions included in a path depends on the code that is executed by it. Another approach is to adjust the hypothetical paths to an even amount of instructions. Actually, this can become complicated when paths equal in most of the instruction and corresponding positions so that the arithmetic mean is the better approach.

Please note that in case that templates are partitioned according to the used operands or calculated results, all templates of this kind have to be selected as best matching templates since the operand may not be identifiable by the program code. At least if no information about the input of the program is known. Hence, in this case all templates have to be applied to the logarithmic probability density function in order to reveal the best template which is then used to compute the overall probability by means of equation (6.2).

As a result of the algorithm, the most likely path is the one with the smallest value of $p(H_i)$. Remember, that the logarithmic density function is used, so that the smallest absolute value indicates the highest probability. Therefore, this also holds for the arithmetic mean.

6.2.1 Complexity

Clearly, the complexity highly depends on the number of hypothetical paths since the number of iterations for all steps of the presented algorithm increase with this number.

As we know, different paths only occur with conditional branches which take one clock cycle to perform if the condition is true and two otherwise. For the worst case, in which the code solely consists of conditional branch instructions, the maximum number of paths h_{max} for a sequence of k instructions is then given by the following approximation:

$$h_{max}(k) \approx 2^{k-1} \tag{6.4}$$

As a result, the complexity of the algorithm increases quadratically with the number of conditional branches. However, since the program code is known the complexity can be estimated before the algorithm is applied so that the considered code can be limited in order to reduce the complexity. Furthermore, the algorithm can be improved as discussed in Section 6.2.2 to counter the mentioned problems. Please note that paths may recombine after having spread apart for some time, for example to perform different subroutines. Thus, the question may occur why the algorithm does not take this recombination into account to handle these paths independently to reduce the complexity. Unfortunately this is not feasible. Even though paths may recombine, these will be performed at different points in time so that they have to be handled independently in order to calculate the probabilities.

Furthermore, the complexity depends on the number of templates applied to the trace. If the set of templates is large, comparatively few instructions will be omitted in step two and as a consequence, more computations have to be calculated in step five in which the singe probabilities are computed. Consequently, the storage and CPU requirements are significantly increased. Nonetheless, when the template set is kept small to only the most important instruction, the complexity is significantly reduced. On the other hand, this may lead to a loss of accuracy since the overall probability is calculated by means of fewer templates. Nonetheless, the algorithm offers a sufficient amount of flexibility to find a good trade-off between performance and accuracy.

6.2.2 Improvements

For the case that the recorded power consumption traces contain a huge amount of instruction cycles, i.e. k is large, the algorithm may be improved in the following way. Instead of calculating all hypothetical paths up to k instructions, a threshold th is defined and paths are only calculated up to this threshold. Then the algorithm is performed. At the end, the winning path w is selected as the survivor path and

the algorithm repeated from the state where w ended. By doing this, the complexity is reduced, since not all paths have to be managed simultaneously and the number of hypothetical paths is reduced with the threshold. For instance, if the power consumption trace contains 100 instruction cycles, th can be chosen as 25. Thus, instead of calculating the paths for a length of k cycles, the algorithm is repeated four times for a length of th cycles. Acting on the assumption of a worst-case scenario only 4 times 2^{24} hypothetical paths have to be managed. In contrast to 2^{99} this yields a reduced complexity of 2^{73}.

6.3 Test

In this section the functionality of the algorithm is proved by a test which further functions as a real case example.

The Matlab code that implements the algorithm for the PIC can be found on the enclosed DVD as well as all other scripts used for this thesis. This includes measurement scripts as well as template creation scripts, assembler code and so forth. An overview about the contents of the DVD can be found in the appendix.

The code used in this test is given by Listing 6.3. Note, that this code does not perform any reasonable function. It is just a code created to test the algorithm. The problem with real code is that it gets significantly more difficult to determine whether the output of the algorithm is correct or not than in the case of dedicated test code.

For instance, branches depend on bits that may be modified during a program execution. Hence, to know if a branch was taken or not, the entire program has to be simulated with all its calculations up to this points. Additionally, real code often contains CALL and RETURN instructions. Keeping track of the branch addresses can become complicated and is not supported by the analysis program yet.

Listing 6.3: Testcode to prove the functionality of the algorithm

```
1  START
2      MOVF W_BUF, 0
3      MOVWF FOP_HW1
4      BTFSC FOP_HW1, 0
5      GOTO ALPHA
6      CLRW
7      NOP
8      CLRW
9      CLRW
10     NOP
11 ALPHA
```

```
12      MOVF W_BUF, 0
13      MOVWF FOP_HW5
14      MOVWF FOP_HW4
15      MOVLW LOP_HW6
16      CLRW
17      NOP
18      CLRW
19      NOP
20      CLRW
21      BTFSC FOP_HW5, 1
22      GOTO BRAVO
23      CLRW
24      CLRW
25      NOP
26      CLRW
27  BRAVO
28      MOVF W_BUF, 0
29      MOVLW LOP_HW8
30      XORLW LOP_HW2
31      ANDLW LOP_HW5
32      CLRW
33      CLRW
34      CLRW
35      MOVLW LOP_HW6
```

As a first step the hypothetical paths for this code have to be calculated up to the number of instruction cycles the recorded trace contains, which in this case is equal to thirty instruction cycles. This results in four paths, as illustrated in Fig. 6.1, since two conditional branches occur. In the first path the jump to ALPHA and BRAVO is taken, in the second path the jump to ALPHA is taken but not the one to BRAVO and so forth. Note, that the length for the paths differ since less instructions are executed by the program when parts of the program are skipped due to branches.

Now the second step can be performed in which all instructions without a template are discarded. In this test we assume that only templates for the CLRW instruction have been created yet. Hence, all other instructions have to be omitted. Further, the best matching templates have to be revealed for each instruction. For the CLRW operation templates can be created for each Hamming weight of the subsequent instruction. Consequently, this Hamming weight has to be calculated in order to associate the templates to the instructions. For instance, a CLRW that occurs before a MOVLW LOP_HW6 would be associated to the template created for CLRW with

the Hamming weight of eight of the subsequent instruction.

Then, in the next step, the corresponding positions of the recorded traces are associated to the CLRW instructions as shown in Tab. 6.1. According to this, the first CLRW of the first path should occur at the tenth instruction cycle, the second at the twelfth instruction cycle and so on.

Path	Positions
1	(10, 12, 14, 22, 23, 24)
2	(10, 12 14, 17, 18, 20, 25, 26, 27)
3	(5, 7, 8, 14, 16, 18, 26, 27, 28)
4	(5, 7, 8, 14, 16, 18, 21, 22, 24, 29, 30, 31)

Table 6.1: Association of the instructions to the various paths

These positions together with the matching templates are written to the matrix H_i for each hypothetical path.

Now, the calculation of the overall probability for each hypothetical path can be calculated to determine the most-likely path for a certain recorded trace. For this test thirty measurements were recorded which randomly executed either the first, second or third path. The fourth path was never executed to see whether measurements would be recognized as this sequence anyhow, due to the control code enframing the test code.

First, the second path was executed and measured. In this case the overall logarithmic probabilities that were calculated are given as shown in Tab. 6.2. As can be seen, the smallest value occurs for the second path and is significantly smaller than the values for the other paths. Thus, in this case the path was recognized correctly.

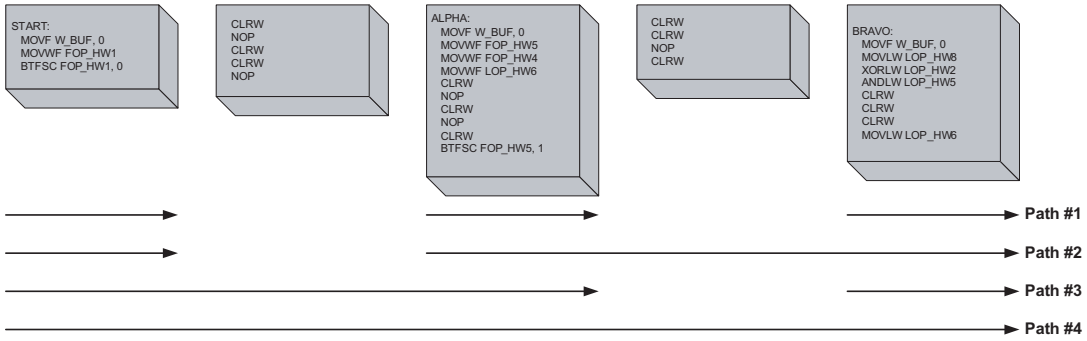


Figure 6.1: Hypothetical paths for the test code

	Hypothetical Path			
i =	1	2	3	4
$p(H_i)$	207.9	89.3	149.2	179.0

Table 6.2: Overall logarithmic probabilities for all hypothetical program paths, calculated for the first recorded trace

The results for all thirty recorded measurements is given by Tab. 6.3. As can be seen the path detection worked one hundred percent correctly.

Measured Trace	1	2	3	4	5	6	7	8	9	10
Executed Path	2	3	3	1	1	3	3	1	2	1
Detected Path	2	3	3	1	1	3	3	1	2	1
Measured Trace	11	12	13	14	15	16	17	18	19	20
Executed Path	3	2	2	1	3	1	2	3	1	3
Detected Path	3	2	2	1	3	1	2	3	1	3
Measured Trace	21	22	23	24	25	26	27	28	29	30
Executed Path	2	2	1	1	3	1	2	3	3	2
Detected Path	2	2	1	1	3	1	2	3	3	2

Table 6.3: Executed and detected program paths for thirty traces

This test was repeated with NOP templates and a combination of NOP and CLRW templates. Furthermore different codes were used. In any case the algorithm yield good results. However, when adding templates for ANDLW and XORLW the recognition success rate was decreased. However, this may be due to inaccurate templates. Note, that these instructions are significantly more influenced by switching noise than NOP and CLRW. However, in terms of path recognition NOP may be an even more important instruction than all other instructions because every conditional test which is not true results in the execution of a NOP instruction.

In conclusion, the provided algorithm has been proved to be feasible. Hence, our assumption that a priori information can be utilized to improve the recognition process by means of templates and single measurements has been confirmed. Consequently, additional use cases like the reverse engineering of secret parts of an algorithm, as discussed in Section 5.5, are additionally feasible.

7 Summary and Conclusion

In this thesis it was shown that Simple Power Analysis is a great tool for the characterization of power consumption properties, concerning the instruction processing of a device. By means of a theoretical examination based on the data sheet, one is able to draw up assumptions of hypothetical power consumption influences and can verify them individually by visually analyzing traces measured while executing dedicated test codes. Further, we have seen that this is an iterative process. When the results are not in line with the assumption, new assumptions have to be drawn up and, again, verified later on. Additionally, even if no data sheet is available for a theoretical observation, we can expect that experience in characterizing other devices can help to understand how an unknown device operates.

As a result of this examination process, a power model was defined which can be used to explain and estimate the power consumption of the device. With this model, it may be for example possible to improve DPA attacks on the PIC significantly, since the power consumption should show a higher correlation compared to the Hamming-weight and Hamming-distance model.

Furthermore, it was shown that the side-channel based instruction recognition on the basis of templates and single measurements highly depends on the presumed knowledge of the observer. If no a priori information is available, instruction recognition is difficult due to the various influences on the power consumption. These are predominantly effects caused by fetching the next instruction and the data dependencies. Moreover, it was shown that modeling switching noise by use of a covariance matrix is not practicable. Instead, it is useful to create several templates for one instruction each covering different influences. At this point, the intensive characterization or more precisely the defined power model becomes useful in order to partition these influences. Nonetheless, the performed tests have shown no sufficient evidence for the possibility of effective and adequate instruction recognition by means of templates and single measurements for the PIC16F687 when no a priori information is available. However, the tests indicated that the use of additional information can improve the recognition process.

Therefore, other possible applications, assuming a higher a priori knowledge about the executed code were introduced and shown to be feasible. As an example, path recognition was elaborated. In this approach, the basic idea is to determine the positions in time when instructions should occur for every hypothetical path in

a given trace in order to apply proper templates to these positions. If applying proper templates to proper positions yields to better probabilities in average, the most likely path has the best overall probability. This idea was formulated in an algorithm whose functionality was then proved for the PIC.

8 Future Work

During the course of writing this thesis, various aspects of side-channel based reverse engineering have been examined and discussed. Yet, this may only be a starting point for even more extensive research into certain details.

At first, the algorithm needs to be tested with real program code and not with dedicated test code in order to prove the practicability. As already mentioned, the problem is to reconstruct the path a program took to have a comparative value to the output of the algorithm. In the test code this path was determined by two bits which were previously set by sending a dedicated command to the microcontroller. Hence, this command could be stored together with the measurement file. In the general case, a program path is extremely data dependent so that one has to simulate the code with all its states in order to get this comparative value. Hence, such a simulator can be programmed in the future.

Secondly, the algorithm can be further improved for the path detection of those power consumption traces which contain a huge amount of conditional branches. One improvement has already been presented by adding a threshold, choosing a survivor path, and repeating the algorithm until the number of instruction cycles has been reached. Further improvements might be possible.

Third, since the code has to be familiar in order to apply the algorithm, it may be a good idea to combine it with an existing disassembler. By this means, the program can start from a HEX code representation of a program. This may also improve the code interpretation, e.g., jump-mark interpretation and so forth.

Furthermore, in this thesis only the power consumption side-channel was exploited. Hence, it may be useful to try other side-channels. The most promising one in this context might be the electromagnetic emanation due to the fact that single parts of the microcontroller can thereby be measured independently which may reduce the huge amount of switching noise that was faced. Another reason for utilizing this side-channel is that no mechanical contact to the device is needed. Therefore, the application of the provided algorithm for version checking or debugging would be simplified since no mechanical impact is needed. Placing an adequate probe close to the device would be sufficient.

Additionally, the provided power model can be further improved. For example, single bits of a bus may be weighted differently which was not taken into account so

far but is possible due to different wire lengths causing different load capacitances and thus different power consumption properties. Moreover, the power model can be used to mount a DPA attack in order to show if the formulated model yields better results than a DPA attack based on the simplistic Hamming-weight and Hamming-distance model.

Some other applications have been additionally mentioned in this work that might work on the basis of detailed a priori information like for example the reverse engineering of secret parts of an algorithm. These can be performed and verified in the future.

At last, template creation can be applied to other microcontrollers. As we have seen, the PIC showed a lot of dependencies in the power consumption. Above all, the fetching process and the huge data dependency rendered a direct instruction recognition from the side-channel to be unlikely. However, other microcontrollers may be better suited for this purpose. The original 8051, for example, executes one instruction within 12 clock cycles. Hence, three times the clock cycles can be analyzed and thus there may be more instruction related information to create adequate templates. Yet, today's 8051 are built differently to the original one. Actually, most of them have an improved design enabling them to execute one instruction within one clock cycle by introducing pipelines which was shown to be disadvantageous for our purpose. However, there are still some microcontrollers compatible to the 8051 that work on twelve and six clock cycles. The Atmel AT89C51ED2, for example, may be a promising candidate for future tests.

A Test Setup

A.1 Block Diagram

The block diagram of the test setup used to perform the power consumption measurements for this work is illustrated in Fig. A.1.

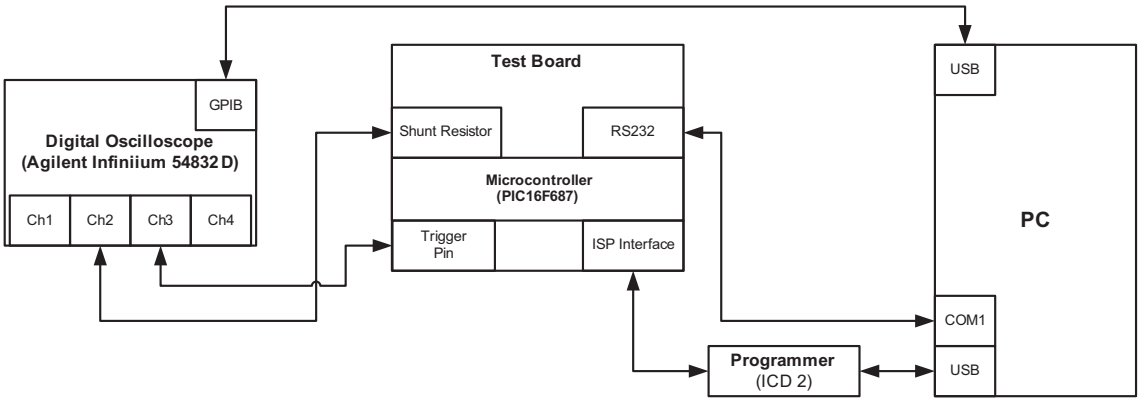


Figure A.1: Block diagram of the test setup

As can be seen, the test setup comprises a digital oscilloscope, a test board, a programmer, and a PC.

The oscilloscope, namely the Agilent Infinium 54832D, measures the power consumption. Therefore, one channel is connected to the resistor located on the test board. Further, channel 3 is connected to the trigger pin of the PIC which is used to indicate the start of the data acquisition during a test.

The test board contains the microcontroller and all connectors needed. Besides the mentioned trigger pin, these are a RS232 connector enabling the microcontroller to communicate with the PC and an ISP interface to be able to re-program the device via the connected programmer, the ICD 2.

The PC is the main control unit of the test setup. It controls the oscilloscope via a GPIB interface, which is additionally used to transfer power consumption traces from the oscilloscope to the hard drive of the PC. Furthermore, it controls the PIC by sending dedicated commands via its serial port.

A.2 Test Board

The schematic of the test board is given by Fig. A.2.

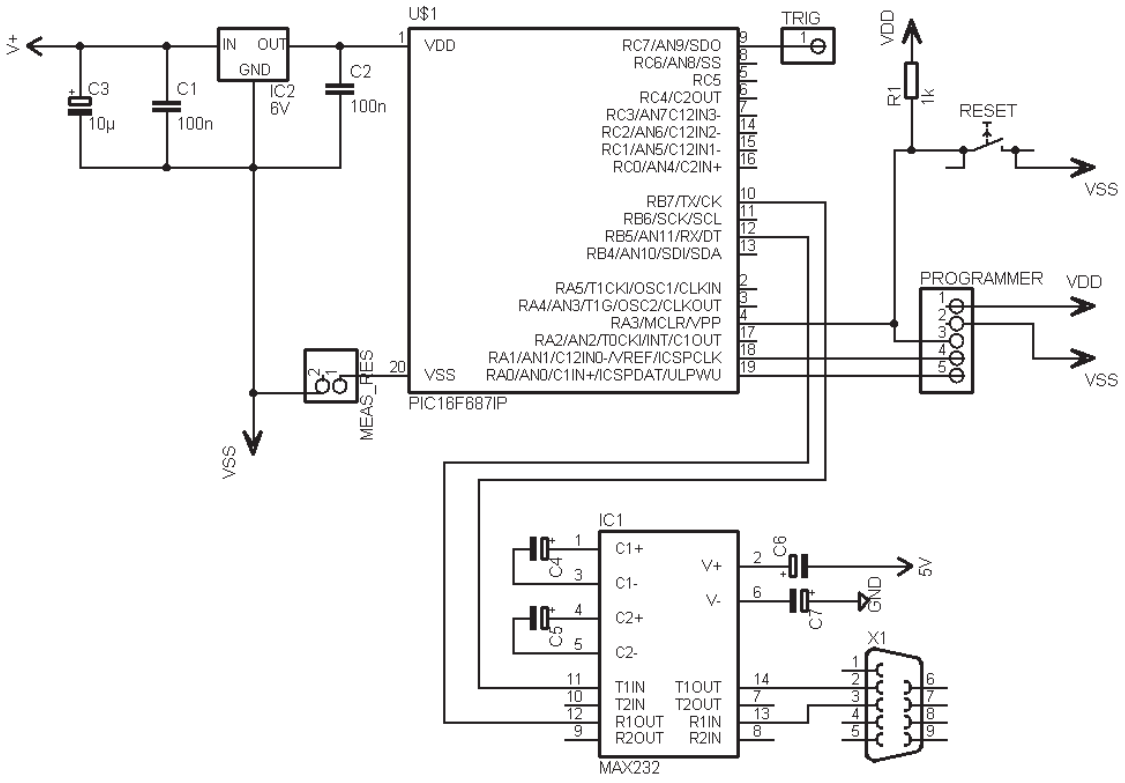


Figure A.2: Schematic of the used test board

As can be seen, a voltage regulator is used to set V_{dd} to a voltage of 6V, which, according to the data sheet, is the maximal voltage for the PIC. The power supply voltage is additionally stabilized by several capacitances.

To communicate with the PC, a serial port cable can be connected to a female DSUB port. The MAX232 connected to this port is used to convert the 12V voltage of the serial port to a 5V voltage applicable for the PIC. Again, this component is stabilized by additional capacitances.

The programmer can be connected to the PIC via five pins. In detail these are: V_{dd}, GND, reset, programming clock, and programming data. In addition, a reset can be manually initiated by a push-button.

Furthermore, one pin head is provided for triggering and two pin heads for inserting a shunt resistor in-between the GND contact of PIC and board.

B DVD Contents

The enclosed contains the assembler and Matlab code used for the test performed throughout this work. Furthermore the most important measurement files are included. A quick overview of the DVD directory structure can be found in Fig. B.1.

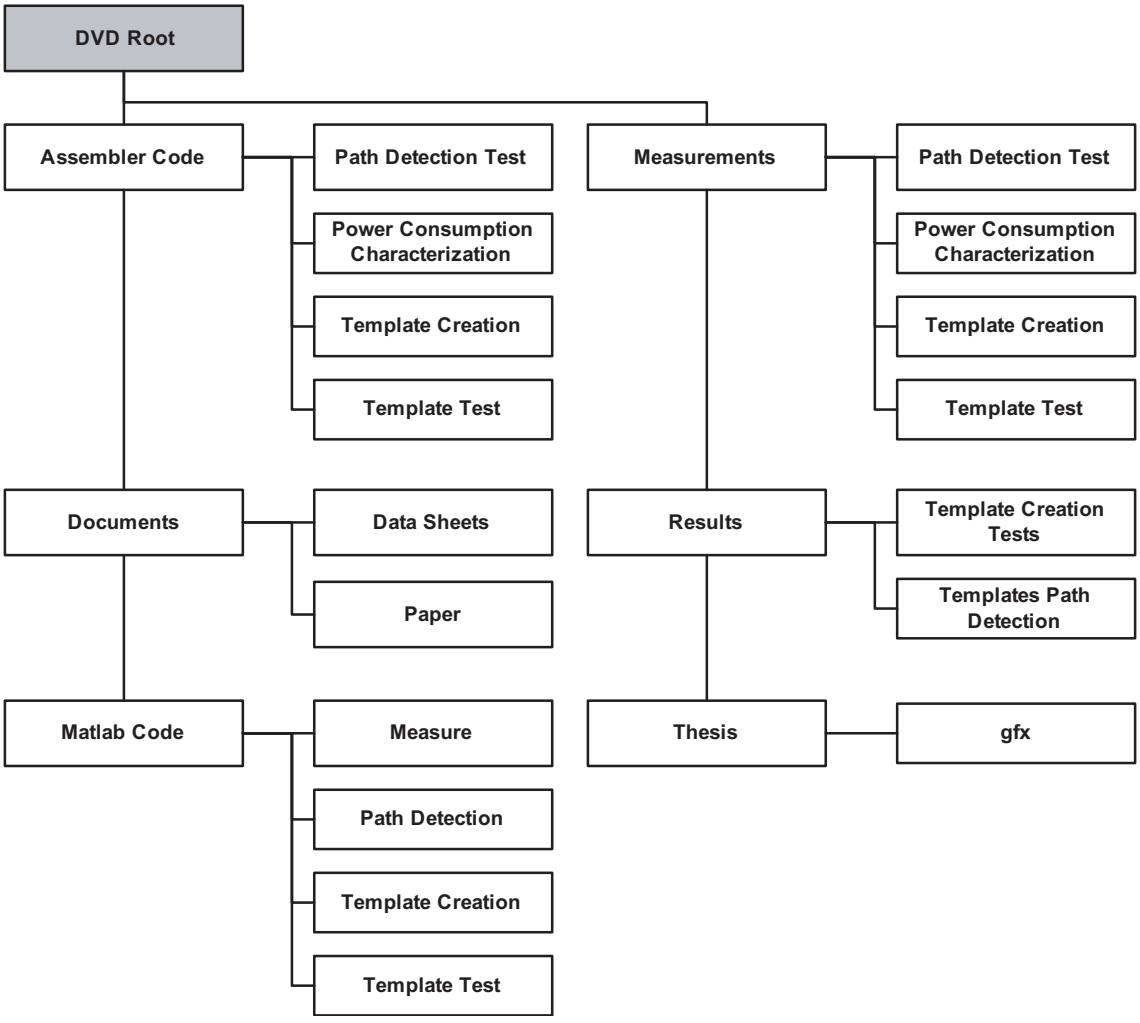


Figure B.1: DVD Directory Structure

The *Assembler Code* directory contains the assembler code which was executed on the PIC for creating and testing templates. It is divided in four subdirectories. In *Path Detection Test* the assembler code used for testing the algorithm as described in Chapter 6 is stored. *Power Consumption Characterization* contains the code with which the various tests for analyzing the power consumption properties of the PIC as described in Chapter 4. Furthermore *Template Creation* includes the code for the template creation tests of Chapter 5 and *Template Test* the corresponding code for testing the quality of those templates.

The *Measurement* directory is organized the same way as *Assembler Code* but contains the most important measurement files. Due to space limitations not all measurements can be included.

Similar to the last directories, the *Matlab Code* directory stores the Matlab scripts for creating and testing templates. Additionally, the script for measuring the power consumption are provided in *Measure*. The code which implements the path detection algorithm for the PIC is stored in *Path Detection*.

In the *Result* directory, the results of the template creation tests, stored in *Template Creation Tests*, and the templates applied in the algorithm test of Chapter 6, contained in *Templates Path Detection*, are comprised.

The data sheets of the oscilloscope and the PIC as well as most of the paper referred to in the bibiography are stored in two subdirectories in *Documents*.

Finally, the Latex code this elaboration is contained in *Thesis*. Furthermore, all figures used are stored in the subdirectory *gfx*.

C Bibliography

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun - Differential Power Analysis, http://www.cryptography.com/resources/whitepapers/DPA.pdf

[KJJ98] Paul Kocher - Introduction to Differential Power Analysis and Related Attacks, http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf

[MOV01] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone - Handbook of Applied Cryptography, http://www.cacr.math.uwaterloo.ca/hac/

[MDS99] Thomas S. Messerges, Ezzy A. Dabbish, Robert H. Sloan - Investigations of Power Analysis Attacks on Smartcards, http://www.sagecertification.org/publications/library/proceedings/smartcard99/full_papers/messerges/messerges.pdf

[TK98] Yuan Taur, Tak H. King - Fundamentals of Modern VLSI Devices, Cambridge University Press

[Bi06] Christopher M. Bishop - Pattern Recognition and Machine Learning, Springer Science+Business Media

[KSWH98] John Kelsey, Bruce Schneier, David Wagner, Chris Hall - Side Channel Cryptanalysis of Product Ciphers, http://citeseer.ist.psu.edu/kelsey98side.html

[JQ01] Marc Joye, Jean-Jacques Quisquater - Hessian Elliptic Curves and Side-Channel Attacks, http://citeseer.ist.psu.edu/637897.html

[MOP07] Stefan Mangard, Elisabeth Oswald, Thomas Popp: Power Analysis Attacks - Revealing the Secret of Smart Cards, Springer Science+Business Media

[S99] Simon Singh - Geheime Botschaften, dtv

[BCO04] Eric Brier, Christophe Clavier, Francis Olivier - Correlation Power Analysis with a Leakage Model, Proceedings of CHES 2004

[Pa01] Lothar Papula - Mathematik für Ingeneure und Naturwissenschaftler (Band 3), Vieweg Verlag

[CRR02] Suresh Chari, Josyula R. Rao, Pankaj Rohatgi - Template Attacks, Proceedings of CHES 2002

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, Pankaj Rohatgi - Towards Sound Approaches to Counteract Power-Analysis Attacks, Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology , 1999

[Ta90] Andrew S. Tanenbaum - Structured Computer Organization, Prentice Hall

[CC90] Elliot J. Chikofsky, James H. Cross II - Reverse Engineering and Design Recovery: A Taxonomy, http://labs.cs.utt.ro/labs/acs/html/resources/ReengineeringTaxonomy.pdf

[ARR03] Dakshi Agrawal, Josyula R. Rao, Pankaj Rhatgi - Multi-channel Attacks, Proceedings of CHES 2003

[Ve06] Dennis Vermoen: Reverse engineering of Java Card applets using power analysis http://ce.et.tudelft.nl/publicationfiles/1162_634_thesis_Dennis.pdf

[No03] Roman Novak - Side-Channel Based Reverse Engineering of Secret Algorithms

[Cl04] Christophe Clavier - Side Channel Analysis for Reverse Engineering (SCARE), http://citeseer.ist.psu.edu/cache/papers/cs/30519/http:zSzzSzeprint.iacr.orgzSz2004zSz049.pdf/clavier04side.pdf

[QS02] Jean-Jacques Quisquater, David Samyde - Automatic Code Recognition for smart cards using a Kohonen neural network, Proceedings of the 5th Smart Card Research and Advanced Application Conference

[MC07] PIC16F631/677/685/687/689/690 Data Sheet, Microchip Technology Inc., 2007, http://ww1.microchip.com/downloads/en/DeviceDoc/41262D.pdf

[MC97] PICmicro Mid-Range MCU Family - Reference Manual, Microchip Technology Inc., 1997, http://ww1.microchip.com/downloads/en/devicedoc/33023a.pdf

[KL03] Sung-Mo Kang, Yusuf Leblebici - CMOS Digital Integrated Circuits – Analysis and Design, McGraw-Hill Higher Education

[Ko96] Paul C. Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ,`http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf`

[FP99] Paul N. Fahn, Peter K. Pearson - IPA: A New Class of Power Attacks, Proceedings of CHES 1999