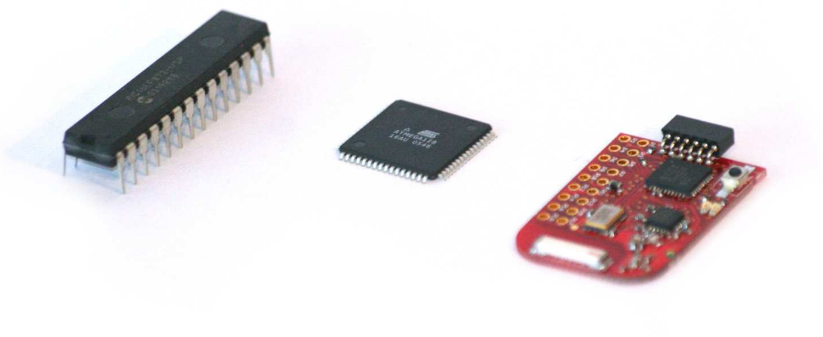
Analytical Cost Estimation for Embedded Systems

Master's Thesis



Marc de Hoop

Analytical Cost Estimation for Embedded Systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Marc de Hoop
born in Bodegraven, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`http://www.ewi.tudelft.nl`

Cover picture: A PIC16F in PDIP package, an ATmega128 in TQFP package and an MSP430 on the eZ430-RF2500T, a mini development board from Texas Instruments.

Analytical Cost Estimation for Embedded Systems

Author:	Marc de Hoop
Student id:	1170791
Email:	marcdehoop@solcon.nl

Abstract

In the today's market of microcontrollers and FPGAs, there are so much different makes and models that making the right choice for a hardware platform for an electronics design is impossible. The spectrum is so wide that an electronics designer is unable to make a good choice for the best microcontroller or FPGA for his/her design based on the information provided by the manufacturers.

To fulfill this need, this thesis presents a method of analyzing a range of hardware targets and C code. The models made of the targets and code provide an accurate prediction of the execution time of the code for each target. With this information the performance of the algorithm on a range of hardware targets can be analyzed in a minimum amount of time.

Performance is not the only measure, power is just as important. Therefore, with the performance information and the design requirements the minimum clock speed is calculated. Then using the power consumption models made in this thesis, the power consumption of each hardware platform can be calculated.

With the methods provided in this thesis, the designer can easily determine the performance and power consumption of a range of hardware platforms and make the right choice in an affordable amount of time.

Preface

This thesis describes the result of my research done on finding a method for electronics desingners to compare a range of microcontrollers and FPGAs based on performance and power consumption. I think I made something that could be very useful in certain desing situations. I would like to thank the following people for their contribution. I would like to thank Arjan van Gemund for his help, advice and corrections.

I would like to thank my family, my fiancée Annemieke and her family for their love and support during the whole peroid of researching and writing this thesis.

I also want to thank my God and Creator, for everything He gives me and of whom I think that He made things complicated for us humans so that we have something to research and think about. SDG.

Marc de Hoop
Delft, the Netherlands
October 18, 2008

Contents

List of Figures

Chapter 1

Introduction

There are numerous kinds of microcontrollers on the market, all with different specifications. When creating an electronics design involving a microcontroller, it is really difficult to determine which one is the best choice for this specific application. Some microcontrollers run on a very high clock speed but divide this clock internally. Some microcontrollers have a hardware multiplier while others don't. No microcontroller is the same. Microcontrollers also differ on their native datasize, power consumption, memory architecture, instruction set, execution speed and amount of registers. Besides microcontrollers there is also a wide range of FPGAs available that are able to fulfill the same task.

These differences may make the choice of the right hardware platform for a specific application too complex to make for a human being. So in practice this choice is usually made based on folk wisdom or the professional knowledge of the electronics designer and his/her experiences with specific microcontrollers in the past. This choice is far from objective; therefore, there is a need for a method to make this choice in an objective manner.

For example, in the TUDelft's Embedded Realtime Systems (IN4073) lab course students have to develop a controller that stabilizes an X-Ufo. This X-Ufo is a quad rotor aerial vehicle. The X-Ufo contains a microcontroller that controls the motors, reads the sensors and communicates to a ground station.

In the development of the hardware for the X-Ufo a microcontroller had to be chosen that was able to read the sensors, control the motors and communicate to the ground station. This has proven a difficult choice because all microcontrollers are different and it is hard to predict the performance.

This problem is not specific for the design of the quad-rotor hardware, it is something much more common. There are numerous cases where a engineer has to develop a control system. In most of these cases these systems the control algorithm that has to be implemented is known beforehand. The task of the engineer is to adapt the algorithm for implementation on a microcontroller and implement it. He has to choose a microcontroller that has the lowest cost in his situation. This means minimizing various factors: power consumption, performance, time-to-market, unit costs, etcetera. Finding the best minimum in all these factors is the main challenge for this thesis.

1.1 Research goal

The key point of this research is finding a method for visualizing the costs of the implementation of the algorithm on a range of platforms. This should be doable in only a small amount of time and provide sufficiently precise results to let the designer make the right choice.

This calls for an analytic method of modelling an algorithm and a range of platforms such that in a minimum amount of time, a calculation can be made that shows what the costs are of implementing the algorithm on a range of platforms. The designer can then choose the right platform, the one that minimizes the implicitly or explicitly expressed cost function.

This research we have the trade-off between making sufficiently precise cost models on one hand and the time it takes to model the hardware and the algorithm on the other hand.

1.2 Research questions

The goal of this research is to develop a method to estimate costs in terms of execution time and power consumption for a given piece of C code and for a diversity of platforms. So that based on this information the designer is able to make an overview of possible microcontrollers (or maybe something else than a microcontroller) for his specific project, rank them based on suitability and make the choice for a specific microcontroller in an objective manner. Hence, research questions investigated in this thesis are:

- How can the execution time of a particular piece of C code be determined for a specific microcontroller?
- How can the power consumption of a microcontroller be determined when it is running a particular piece of C code?
- How can an FPGA that implements the same algorithm as a microcontroller a particular piece of C code be compared to the microcontroller in terms of power and resources?

Researching these questions results in multiple models. In these models there is a trade-off between the precision of the resulting costs on one hand and the detail of the models on the other hand. In addition, the detail of the models is connected to the amount of time spent on making and using the models.

This gives rise to the following questions:

- What is the trade-off between precision of the determined costs and the time spent on making models of the hardware and the C code?
- What is the trade-off between the detail of characterization of an *algorithm* and the precision of the result?
- What is the trade-off between the detail of characterization of a *platform* and the precision of the result?

1.3 Contributions

In this thesis we have researched the possibilities of determining the execution time of a piece of C code. Two models of the platforms are made: a detailed model containing 73 distinguished operations and an abstract model containing 15 parameterized operations. Based on the operations expressed in C code and the costs of these operations given by the models it is quite good possible to calculate the costs of a piece of C code.

Together with other design requirements, like the required sample frequency or maximum execution time of the algorithm the minimum clock frequency can be determined. The minimum clock frequency can then be used to calculate the power consumption of the device using the power models derived in this thesis.

Further, we have determined the resource costs of the implementation of the equivalent of a C algorithm in an FPGA.

1.4 Outline

First existing approaches for determining the costs of a piece of C code or an algorithm will be described in Chapter 2, then in Chapter 3, the platforms used as example in this research will be described. Chapter 4 describes the detailed model which forms the basis of an abstract model as described in Chapter 5, which covers the creation of an abstract model and the derivation of the power consumption model. Chapter 6 analyses a piece of C code for time and power costs. In the end, Chapter 7 draws a conclusion and gives some ideas for further research.

Chapter 2

Existing approaches

This chapter describes existing approaches that are or could be used for finding the right hardware platform to for implementation of an algorithm. First three analytic methods are described: experience, simulation or testing on real hardware and analytical cost estimation. Then Worst-Case Execution Time (WCET)-analysis will be described, which is an example how cost estimation is used and focuses on the worst case.

2.1 Experience

The first possible approach is by using experience. An experienced electronics designer knows something about the performance of the hardware he used in previous designs. In a new design he can use this knowledge to choose a hardware platform.

When a controller has to be designed that should implement 3 discrete time PID controllers with a sampling rate of 1 kHz the designer can say: “In a previous project I used microcontroller X from brand Y and that could do 5 PID controllers at 500 Hz and there was room for more, so I suppose that 3 PID controllers at 1 kHz would be possible too”. Chances are that this turns out right and everything works. But what if it doesn’t? What if it turns out that it is not possible to reach 1 kHz? At a lower frequency the system might not be stable. He could overclock the microcontroller to get the desired results but chances are that this leads to failures in the future. He could choose a different microcontroller with the consequence that some or nearly all parts of the code have to be re-implemented and the project costs more time and money than originally planned. Advantages of this approach are that choosing a microcontroller only costs a little amount of time and the result of this choice will be a microcontroller that has been used in the past which will mean that there are no extra investments in development tools and time for familiarizing with a new microcontroller. A large disadvantage is that it is possible that the wrong choice is made and has to be corrected.

Another option is to choose a more powerful microcontroller than what the designer expected to need so there is some headroom if things turn out to be worse than expected. The advantage of this is that the probability that things go wrong is reduced. The disadvantage is that a more powerful microcontroller usually consumes more power and is more expensive.

So the conclusion is that this approach is usable if an experienced electronics designer is (and stays) available, if one is willing to take the risk that a (partial) redesign has to be done, or one has to accept that the design is suboptimal, e.g. it consumes more power than needed and the costs are higher than needed.

2.2 Simulation and testing on real hardware

To obtain the performance of an algorithm one could just implement the algorithm and see how much cycles it takes to execute the code. Development environments like those from IAR Systems[1] have a debug functionality that allows the developer to step trough the code, view the registers and see how much cycles are executed by the microcontroller.

So if the execution time of a certain algorithm needs to be determined, it is implemented and simulated and the debugger gives the amount of cycles spent. When this would be done in the diverse development environments for all targets in consideration, one could get the costs in terms of time and choose the best option. The advantage of this method is that it should be possible to get the cycle-accurate costs of a piece of code. Drawbacks are that the algorithm has to be completely implemented while the hardware is yet in development and because each platform is different, the code has to be adjusted each time to get the code to run. Simulation of interrupts and hardware connected to the microcontroller could be a difficult task and is error prone. Besides that, all development environments should be available, which means that they have to be bought and that could be expensive. IAR compilers cost, dependent on the target, between €2,500 and 4,000 for a stand-alone license of the standard edition.

Another option is to implement the algorithm on real hardware, for example using a development board. Then it is possible to determine the performance of the algorithm in the most realistic way. The advantage of this approach is that this is a very realistic situation and it is possible to not only measure the performance in the time domain but also possible to measure the electrical power consumption. The drawbacks are that when one wants to evaluate multiple platforms he has to implement the algorithm on all of these platforms, which costs much time and he needs all development environments and all hardware platforms which is very costly.

2.3 Analytical cost estimation

Our research focuses on analytical cost estimation. Analytical cost estimation means that by making an analytical model of the hardware and a model of the algorithm we want to estimate the costs of an algorithm in terms of time or power. for example, if a given algorithm contains n multiplications, m additions, etc. and it is given that a multiplication on a certain platform costs t cycles, an addition s cycles etc. then it is possible to calculate the costs of that algorithm on that platform.

[1] `http://www.iar.com`

In our research we assume sequential execution. This is justified by the low-end micro-controllers we consider. It is also possible to make a power consumption model to estimate the total power consumption.

This method does not deliver an exact figure of the costs; a model is a model and it can never be perfect, however, a model can be precise enough to get an good idea of whether it is possible or not to implement an algorithm within the given limits. An analytical model also exhibits the dependency of the costs on specific architectural parameters and gives the possibility of making a ranking of the diverse hardware platforms. The precision of this approach depends on the precision of the models and the predictability of the costs; e.g. if a multiplication sometimes costs much more time or power, that is no problem if it is predictable, because then it can be incorporated in the model, but if that's not the case the model gets inaccurate.

The advantage of this approach is that when the models of the platforms are available the only thing that has to be done is make one model of the algorithm and project that on the models of the platforms to obtain the result. This is much faster and cheaper than simulating or implementing the algorithm for all platforms.

Drawbacks are that there have to be made models of all hardware platforms, which can be a time consuming operation. On the other hand, once they are available, they can be used over and over again. Another drawback is that the results are not as precise as when simulating or implementing the algorithm, although, this method should be precise enough to make a ranking of the hardware platforms for their suitability.

An example of this kind of cost estimation can be found in [15], where using a benchmark the abstract model of a machine is derived. By modeling other benchmarks in terms of operations the performance is predicted. These benchmarks are written in Fortran and executed on machines like the CRAY-2, IBM RS/6000 530 and MIPS M/2000.

2.4 Worst-case execution time analysis

The worst-case execution time is the maximum time a certain task could take to execute on a specific hardware platform. The analysis of this time is mainly used in the field of real-time systems where it is important to know the upper bound on the time that is required to execute a specific task. This is done by analyzing the source code or disassembled binary code of a program.

WCET-analysis is a special form of cost estimation. While the before mentioned techniques can deliver best, worst and average-case figures, WCET-analysis focuses on the absolute worst-case. This is needed in demanding situations where hard timing constraints exist and when the performance has to meet a certain minimum. WCET-analysis is different from cost estimation by experience, simulation and analytical cost estimation because it does not define a method for determining the costs but it is about what is being done with the result of these cost estimations.

WCET-analysis has to take into account both the program flow at the high level and the execution time at the low level, including the effects of caches, branch prediction mechanisms, and pipelining. The complexity of nowadays hardware and software makes WCET analysis a difficult task.

Research done in this area has led to dynamic forms of WCET analysis because of the complexity of the hardware platforms. These dynamic approaches measure the execution times of basic blocks of code and then use static analysis at a higher level for computing the worst-case execution times.

Using WCET analysis for choosing a hardware platform in an electronics design is possible but it requires a full implementation of the algorithm on multiple platforms. After implementation, a WCET analysis is done on all these platforms. The result of this analysis could be used for comparing the platforms and making a choice. The advantage of this method is that the performance and the feasibility of the implementation of the algorithm are known beforehand. The drawback is that the amount of time involved because a full implementation and WCET analysis has to be done for each platform. The time involved depends on the portability of the code and the amount of time required for a WCET analysis, but is certainly higher than the time spent on other (general case focused) approaches. In addition, WCET analysis is not meant for analysis in the design phase, but more to check the timing constraints after implementation.

2.5 Summary

In this chapter we have described three approaches:

- Experience: an experienced electronics designer makes a choice based on results of previous projects. Advantages are: only a little time is spent on making the choice and the chosen controller is usually one that is used in the past eliminating the need for new tools and time spent familiarizing with a new microcontroller. Disadvantages are: there is a risk of making a wrong choice, which involves extra, unplanned, costs and there is a risk that the chosen microcontroller is not the optimal solution when looking at the financial aspect and the power consumption.

- Simulation and testing on real hardware: the algorithm is implemented for all targets in consideration and then tested for performance using a simulation or on the real hardware (for example using a development board). Advantages of this approach are: the results are precise and give a good image of the real performance. Besides that it is possible to measure the power consumption. Disadvantages are: development environments (and, if testing on real hardware, the hardware itself) have to be already available or bought which can be costly, implementation has to be done for all platforms, which is time consuming (dependent on the portability).

- Analytical cost estimation: a model of the algorithm is projected on diverse models of the diverse hardware targets in consideration. Advantages are: once a model is made it can be used over and over again, no real hardware or development environments are required and it is possible to make a model of the power consumption of the platform so this can be part of the comparison too. Disadvantages are: the results are precise enough for a ranking of the platforms but give no precise figures and models of the hardware platforms have to be made the first time.

These methods can be used for worst-case execution time analysis where the implemented algorithm is analyzed for its worst-case execution time. This is used in situations whare the minimum performance is a hard requirement and it has to be absolutely certain that this requirement is met.

In the next chapters, the research done for this thesis on analytical cost estimation will be presented. We will focus on the trade-off between time spent on estimating the costs and the simplicity of the models on one hand and the accuracy of the results on the other hand.

Chapter 3

Introduction to the platforms

In the beginning, three microcontroller series formed the basis for this research: the Microchip PIC18F, the Atmel ATmega and the Texas Instruments MSP430. Later a Xilinx Spartan-3 FPGA and an ARM7 microcontroller manufactured by NXP have been added to extend the spectrum. These devices are widely used in the industry and have a large market share. For each series one particular model was chosen, for the microcontrollers usually a top of the line model with about 128KB program memory. The research was done for this particular model but the results are applicable to all models in the series. The differences between models in these series are generally the amount of program and data memory and the amount and types of embedded peripherals. For the FPGA the model used on the Spartan 3 starterboard was chosen.

The 8-bit Atmel ATmega, the 16-bit Texas Instruments MSP430 and the 32-bit ARM7 were chosen because their instruction set and architecture is more or less comparable but their native data size differs. With these four microcontrollers, it is possible to determine the influence of the native data size on the costs in terms of time and power. The 8-bit Microchip PIC18F has a different architecture and instruction set, so evaluating this microcontroller and comparing it with the Atmel ATmega will show the influence of the architecture and instruction set.

The Xilinx Spartan-3 was chosen because it was used for the development of a controller for a helicopter or quad-rotor aerial vehicle in lab sessions of the TUDelft's IN4073 course. In the lab session it is loaded with a 32-bit softcore that could be programmed in C. In this research only the ability to implement an algorithm in hardware will be considered, so the softcore will not be covered.

In this chapter, all platforms will be described in more detail. For each series, the chosen model will be named and the specifications will be described. This includes the amount of program and data memory, the peripherals and special features (both positive and negative) will be named. The peripherals are not related to the performance, but play a role in the process of choosing the right microcontroller because the microcontroller is usually connected to some sensors or other external devices via these peripherals. The availability of a certain peripheral on a certain microcontroller can make a difference between a go or a no-go.

The type of memory architecture is described because this could for example play a role in the costs of accessing a data table (e.g. a table containing pre-calculated values of a sine function) in the program memory. Some microcontrollers have one working register while others have 16 or 32. Having many registers can save time in saving and loading data from the data memory but on the other hand, a context save can cost much more time when a lot of registers have to be saved.

For an indication of the price, the price per piece of the cheapest package variant when buying 1000 or more at Farnell or DigiKey is given.

3.1 Microchip PIC18F

From the Microchip PIC18F series the PIC18F8722 was chosen. This is an 8-bit microcontroller with 128 KB flash memory for program storage, almost 4 KB SRAM data memory and 1KB EEPROM data memory. The memory can be extended using off-chip RAM or Flash memory to a maximum of 2 MB.

Typical microcontroller I/O peripherals are available: 2 SPI/I^{2}C ports, 2 UARTS, a 10 bits 16 channel A/D-converter and up to 12 PWM channels. An 8×8 bits hardware multiplier is available too. To reduce power consumption the PIC18F8722 features nanoWatt Technology: features that can reduce power consumption. It is possible to switch to a different clock source or disable the CPU core and/or the peripherals.

According to the datasheet [11], the PIC18F8722 has a C compiler optimized architecture because of an "Optional extended instruction set designed to optimize re-entrant code". The indicative price of the PC18F8722 is €5.40 at Farnell.

3.1.1 Memory organization

The memory architecture is a Modified Harvard Architecture. There is an instruction memory (Flash) and a data memory (RAM). It is possible to read from the instruction memory and to write to the instruction memory (only in blocks of 64 bytes). The data memory is referenced as 'F' in the instruction set. All instructions are 2 bytes long and the instruction memory is word aligned.

The CPU core has an accumulator-based architecture. There is one working register called W. Most instructions use this register as operand or as destination. E.g.: the `ADDWF f,d,a` instruction adds (`ADD`) the value of the working register (`W`) to the value the location defined by `f` in the data memory (`F`) and stores the result in the working register or in the data memory based on the value of `d`.

The data memory is organized in 16 banks of 256 bytes. A Bank Select Register (BSR) that defines the selected bank. The MOVLB instruction directly loads the BSR with a fixed value. It is not possible to load a variable to the BSR. A limitation of this banking architecture is that the maximum amount of memory that can be linearly addressed is 256 bytes. For the IAR C compiler this means that the size of an array is limited to 256 bytes: 256 chars, 128 ints or 64 longs.

Peripherals can be configured and controlled via the Special Function Registers (SFRs). The SFRs are located in bank 15 as the last 160 bytes. To avoid setting the BSR for every instruction that uses an SFR, instructions with an `f` parameter also have an `a` parameter.

This `a` parameter defines if `f` is a location in the data memory or in the Access Bank. The Access Bank is formed of the first 96 bytes of bank 0 and the 160 SFRs.

To allow for 2MB of memory the address bus is 21 bits wide. The program counter (PC) can be written to. This can be used for computed GOTOs, a lookup table or a switch statement. On a call or interrupt, the value of PC+2 is pushed onto the stack. The stack is located in a separate 31×21 bits RAM that can be modified using `PUSH` and `POP` instructions that push or pop the program counter. It is also possible to modify the top of the stack via a set of registers. The length of 31 words allows for 31 calls and/or interrupts. If more calls are executed stack overflow happens: a bit is set that the stack was full and the microcontroller resets itself. For this reason implementing deep recursive algorithms on a PIC18F is not possible.

3.1.2 Instruction timing and clocking

The PIC18F can be clocked from diverse sources. This clock is internally divided by four to generate four internal clock signals; one instruction cycle is executed in four clock ticks.

Executing an instruction takes two instruction cycles, one fetch cycle and one decode and execution cycle. However, these cycles are pipelined so each instruction executes effectively in one cycle. In case of a branch, the pipeline is flushed and a NOP instruction is executed for the already fetched instruction.

PIC18F instructions are mostly single-word (16 bits). There are four double-word instructions. For these instructions the second word is executed as a NOP, thus taking effectively two cycles.

The maximum clock speed of the PIC18F8722 is 40 MHz. Combined with this clock division the PIC18F8722 can execute up to 10 MIPS.

3.2 Atmel ATmega

The Atmel ATmega128 was used in the hardware of the X-Ufo so this one was chosen to represent the ATmega series. The ATmega128 is an 8-bit microcontroller, like the PIC18F. This microcontroller has 128 KB Flash program memory, 4 KB SRAM data memory and 4 KB EEPROM data memory. It is possible to add up to 64 KB external data memory.

Available I/O peripherals include: 6 PWM channels, an SPI interface, a two-wire (I^2C) interface, two UARTS and a 10 bits 8 channel A/D converter. The cpu core has a 2-cycle 8×8 hardware multiplier. To preserve power, six power modes are available that enable to processor to stop the CPU core, the peripherals and/or the clock. The datasheet describes the ATmega128 as a ‘high-performance, low-power microcontroller with an advanced RISC architecture’ [4]. The indicative price is €7.00 at Farnell.

3.2.1 Memory organization

The memory architecture of the ATmega is a Modified Harvard Architecture. Data and instruction memory are separated. It is possible to use the instruction memory as data table and to read data from the instruction memory. It is also possible to write to the instruction memory, 128 words at a time, but this is more for reprogramming the chip and not for some kind data storage. As all instructions are 16 bits or 32 bits in length, the instruction memory is organized as 16×64K.

To be able to address the full 64K words, addresses in the ATmega are 16 bits wide. The stack pointer is implemented as two 8-bit registers in the I/O space. The stack of the ATmega is located in the data memory and grows from higher memory locations to lower memory locations. The PUSH and POP instructions push and pop registers to and from the stack.

The ATmega has a register file of 32 8-bit general-purpose registers. Six of these registers can be combined to 3 16-bit data pointer registers, X, Y and Z. It is possible to do a normal read/write, read/write with displacement, read/write with post-increment or read/write with pre-decrement.

When addressing the data memory, the first 32 locations address the register file, the next 64 locations address the standard I/O memory, the next 160 address the extended I/O memory, the next 4096 locations address the internal SRAM and the other 61184 locations address the external memory. All these parts of the memory can be read and written using standard load and store instructions. In addition to that, the first 32 I/O addresses are directly bit accessible. Bits can be set or cleared and checked in a single instruction. This saves two cycles with respect to a standard read-modify-write cycle. Most of these 32 registers configure the value and data direction of the I/O ports as these are the most useful to be bit-accessible.

3.2.2 Instruction timing and clocking

The ATmega128 can be clocked from diverse clock sources. An internal clock divider makes it possible to run at 254 Hz on a 32.768 kHz watch crystal to save as much power as possible. The maximum clock speed is 16 MHz.

The AVR instruction set, described in [5] consists of 133 instructions. Standard instructions, like `ADD`, `SUB`, `OR`, etc. execute in 1 cycle. Instructions that read and write two registers take 2 cycles. Examples are the `ADIW`: ADd Immediate to Word and `MUL`: MULtiply instructions. An exception to this is the `MOVW`: MOVe Word instruction, that moves two adjacent source registers to two destination registers in a single cycle. Load and store instructions via the X, Y or Z registers also take 2 cycles and branch instructions take 2 cycles when taken or 1 otherwise. Long jumps and short calls take 3 and long calls and interrupt or subroutine returns take 4 cycles.

Although the maximum clock speed is 16 MHz, due to the amount of multi-cycle instructions it will not be possible to reach an instruction throughput of 16 MIPS in practice.

3.3 Texas Instruments MSP430

The Texas Instruments MSP430 will be represented by the MSP430F2619. This is the most extensive model of the MSP430x2xxx family. The MSP430F2619 CPU is a 16-bit RISC CPU. It has 120 KB flash memory for program storage, 256 B flash for long-term data storage and 4 KB RAM. It's CPU architecture has deep roots in the history of computing: the instruction set is similar to that of the DEC PDP-11 (see [6] and [10]).

The MSP430F2619 has a rather extensive amount of (I/O) peripherals: an 8 channel 12-bit A/D converter, 2 12-bit D/A converters, 11 PWM channels, 2 UART/SPI interfaces and 2 SPI/I^{2}C interfaces. A three channel DMA controller makes it possible to use A/D converter without CPU intervention. A 3 cycle 16×16 multiplier capable of doing signed, unsigned, multiply accumulate, 16×8, 8×16 and 8×8 multiplications is implemented as peripheral. As peripheral, its activities do not interfere with the CPU activities.

According to the datasheet, the MSP430 family is characterized by its ultralow-power consumption and its powerful 16-bit RISC CPU [18]. The indicative price of the MSP430-F2619 is \$9.38 per piece at DigiKey, which is about €5.90.

3.3.1 Memory Organization

The memory architecture of the MSP430 is a Von Neumann architecture. The program memory, the data memory, the I/O peripherals are all mapped in the same memory space. The first 16 locations are the special function registers (interrupt enables and interrupt flags in case of the MSP430F2619), followed by the configuration registers for peripherals, the RAM and the flash memory.

There are two versions of the MSP430 CPU core: the normal MSP430 core and the MSP430X core. The first has 16 16-bit registers; the latter has 15 20-bit register and one 16-bit register (the status register). The MSP430F2619 has a 430X core to be able to address more than 64KB memory. Standard instructions only use the 16 LSB of a register. To use the full 20 bits the instruction is preceded by an extension word that provides the extra needed information.

The first 4 registers of the 16 available registers have special functions. R0 is the program counter, R1 the stack pointer, R2 is the status register and the first constant generator and R3 is the second constant generator. The program instructions and the stack are word-aligned so the values of the stack pointer and the program counter are always even. The status register contains bits like the carry bit, the negative bit, the zero bit, the general interrupt enable bit and bits to turn off oscillators or clocks.

The MSP430 user guide ([19]) describes seven different addressing modes that allow the source or destination of an instruction to be registers, based on registers, constant parameters or a combination of these. It is also possible to use a register as pointer and increment the register afterwards.

When the constant generator registers are used as source, the value returned from the register depends on the addressing mode. In this way, frequently used constants like 0, 1, 2, 4, 8 and -1 can easily be used in instructions. This saves time for loading a constant in a register.

To be able to operate on 8-bit data all applicable instructions have a bit that specifies whether the instruction is a word (16 bits) or a byte (8 bits) instruction. In assembly code, this is specified by a `.B` suffix. For example: `ADD R5, R6` adds the 16 bits value in R5 to the 16 bits value in R6 and saves the result as 16 bits value in R6. `ADD.B R5, R6` adds the 8 LSB of R5 to the 8 LSB of R6 and saves the result as an 8 bits value in R6 (the 8 MSB of

R6 become 0).

3.3.2 Instruction timing and clocking

The clock module of the MSP430F2619 can run on diverse clock sources ranging from 12kHz to 16 MHz. The clock module outputs three clock signals: one master clock for the CPU and the system and two other clock signals that are software selectable from the different clock sources for individual peripheral modules. All clocks can be divided by 1, 2, 4 or 8. In this way, the programmer is able to let a specific peripheral run on just the minimum required clock speed to save as much power as possible.

The MSP430 instruction set consists of only 27 instructions. This seems little, but the different addressing modes provide possibilities where other CPUs would need a different instruction. The time it takes to execute an instruction depends on the type of instruction and the addressing mode used. Register to register instructions (e.g.: `MOV R5, R6`) take one cycle, using more advanced addressing modes (e.g.: `MOV &FOO,4(R6)`) can take up to 6 cycles.

The maximum clock speed is 16 MHz. Because most instructions take two cycles or more the practical speed will be lower than 8 MIPS. This does not mean that this microcontroller will be slower than, for example, the ATmega128, because this microcontroller's instructions operate on 16 bits where the ATmega128's instructions operate on only 8 bits.

3.4 ARM ARM7TDMI-S — NXP LPC2144

ARM is the designer of the ARM7TDMI-S 32-bit RISC CPU core that implements the ARMv4T architecture. The designs of these cores are licensed to a number of manufacturers; NXP is one of these manufacturers. From NXP the LPC2144 was chosen for this research. The LPC2144 has 128KB Flash memory and 16KB RAM.

Around the ARM7TDMI-S core NXP added an extensive amount of peripherals: a USB 2.0 full speed device controller, 2 10-bit A/D converters with in total 14 channels, a 10-bit D/A converter, 6 PWM outputs, a Real-Time Clock, 2 UARTS, 2 I^2C interfaces and 1 synchronous serial port for SPI and other 4 wire protocols.

The fact that the designer and the producer are not the same shows up in the number of documents: there is a product datasheet from NXP that gives an overview of the LPC214x series [12], a user manual from NXP that describes all details about the LPC214x series implementation of the ARM core [16] and the ARM Architecture Reference Manual from ARM that describes the instruction set and memory architecture [2]. The indicative price of the LPC2144 is €6.48 at Farnell.

The ARM was added in a later stadium of this research so some research done on the other platforms has not been done for the ARM.

3.4.1 Memory Organization

The ARM7 has a Von Neumann architecture. The 32 bits architecture enable the CPU to access a total of 4 GB memory, much more than this microcontroller has. The parts that are used are more or less evenly spread over this 4 GB range. The first 128 K addresses are the flash memory, then there is a gap, at 1 GB (address 0x4000 0000) there is 16 KB RAM and the peripheral addresses are located at the upper 500 M addresses. From these upper 500 M addresses only 4 M is used: the upmost 2 MB is used for addressing up to 128 AHB (Advanced High-performance Bus)-peripherals and the lowest 2 MB is used for addressing up to 128 APB (ARM Peripheral Bus)-peripherals. Each peripheral has 16 KB address space to allow simple address decoding for each peripheral. In case of the LPC2144, the AHB connects the CPU core to the interrupt controller and the APB connects to all peripherals.

The ARM7 has two operating modes: ARM and THUMB: in ARM mode all instructions are 32 bits, in THUMB mode, all instructions are 16 bits. The THUMB instruction set is a subset of the ARM instruction set such that it is possible to store twice as much instructions in the same amount of memory while retaining most of the performance. In ARM mode, the instruction set counts 51 instructions; in THUMB mode this is 35. Switching is done by executing a Branch and Exchange (`BX`) instruction.

In ARM mode, 16 registers are available. R13 is reserved as Stack Pointer, R14 as Link Register (holds the return address when a branch-and-link instruction is executed) and R15 as program counter. The other registers do not have a special function. In THUMB mode only R0 to R7 are available. Some special THUMB instructions can access the stack pointer, the link register and the program counter.

One of the special features of the ARM CPU is the barrel shifter. In data processing instructions, one of the operands can be run through the shifter without extra costs. This operand can be a register or an immediate value. This value then is shifted or rotated and then used as operand. In ARM mode, most of the instructions also can be executed conditionally. The first four bits of an instruction encode the condition, for example: equal, carry set, always, etc. If the condition is met, the instruction is executed. If not, the instruction is executed as NOP.

3.4.2 Instruction timing and clock sources

The LPC2144's on-chip integrated oscillator works with crystals from 1 to 30 MHz. The internal PLL can multiply the incoming clock signal to anything between 10 and 60 MHz. This is the only available clock source for the CPU core and the peripherals. The real-time clock has its own source: a 32.768 kHz crystal and its own power supply pin so it can work independently from the main clock.

As the ARM7 CPU core is a much more complex than the PIC, ATmega or MSP. This clearly shows up in the instruction cycle timing. The ARM7TDMI-S technical reference manual describes four transaction types [3]. Instruction cycle counts are expressed in these transaction types. The time per transaction depends on the type of the transaction, the memory bus width and the memory access time. To estimate the execution time ARM has developed the ARMulator.

Due to the complexity, it is hard to give a figure about the execution speed of the LPC2144. In an application note, ARM states that the average cycles per instruction is 1.9 in case of 32 bits wide zero wait-state memory [1] which would lead to about 31 MIPS at 60 MHz (60 MHz/1.9 CPI = 31.57 MIPS). On the other hand, Philips states in a press release that the LPC21xx family executes 54 Dhrystone MIPS at 60 MHz [14]. Disscussions on the LPC2100 mailinglist of the embeddedrelated.com website indicate that this figure is not far from reality [8].

3.5 Xilinx Spartan-3 FPGA

A control algorithm could also be implemented into an FPGA. Therefore, an FPGA is also considered in this research. The XC3S400 model from the Spartan-3 series from Xilinx was chosen because in the IN4073 course this FPGA, programmed with the X32 softcore is being used.

An FPGA is not a microcontroller. A microcontroller has a CPU that executes instructions and connected to the CPU there are peripherals that implement special functions. An FPGA is much more low-level. FPGA stands for Field Programmable Gate Array: an array of logic gates that can be programmed “in the field”, or in other words: after the product left the factory. This is an advantage over an ASIC (Application Specific Integrated Circuit), which cannot be modified afterwards.

The indicative price of the XC3S400 is €18.94.

3.5.1 Internal organization

The costs of the implementation of some design in an FPGA is expressed in used LUTs, CLBs, multipliers and amount of RAM blocks. The Spartan-3 family is internally organized as a matrix of Configurable Logic Blocks (CLBs). Each CLB consists of four slices; each slice contains two Look-Up Tables (LUTs) that implement logic and two storage elements that could be used as flip-flop or latch. LUTs in a Spartan-3 have four inputs and one output. Because of their structure, they can also be used as a 16×1 RAM or as a 16-bit shift register. Figure 3.1 shows the organization of the CLBs and slices. The XC3S400 has $32 \times 28 = 896$ CLBs. That is 3584 slices, containing 7168 LUTs and flip-flops. According to Xilinx this is the equivalent of 8064 logic cells [22, page 3].

To communicate to the rest of the world there are I/O Blocks (IOBs) placed on the borders of the chip that connect the logic to the I/O pins. The distribution of clock signals is done by four Digital Clock Managers (DCMs). For on chip data storage the chip has two columns that consist of several 18 Kbit dual-port RAM blocks; 288 Kbit in total is available in the XC3S400. Each RAM block is also associated with a dedicated 18×18 multiplier, 16 in total for the XC3S400. The DCMs are positioned at the ends of the outer block RAM columns. Figure 3.2 shows the layout of all these elements.

3.5.2 Memory and clocking

The XC3S400 does not have much ram, 288 Kbit block ram and in total 56 Kbit RAM distributed over the whole chip, 16 bits per slice. This means that there is in total only 288 Kbit + 56 Kbit = 344 Kbit = 43 KByte RAM available in the XC3S400. Therefore, the Spartan-3 starter board that is used in the IN4073 lab course has an extra RAM-chip on board, pro-

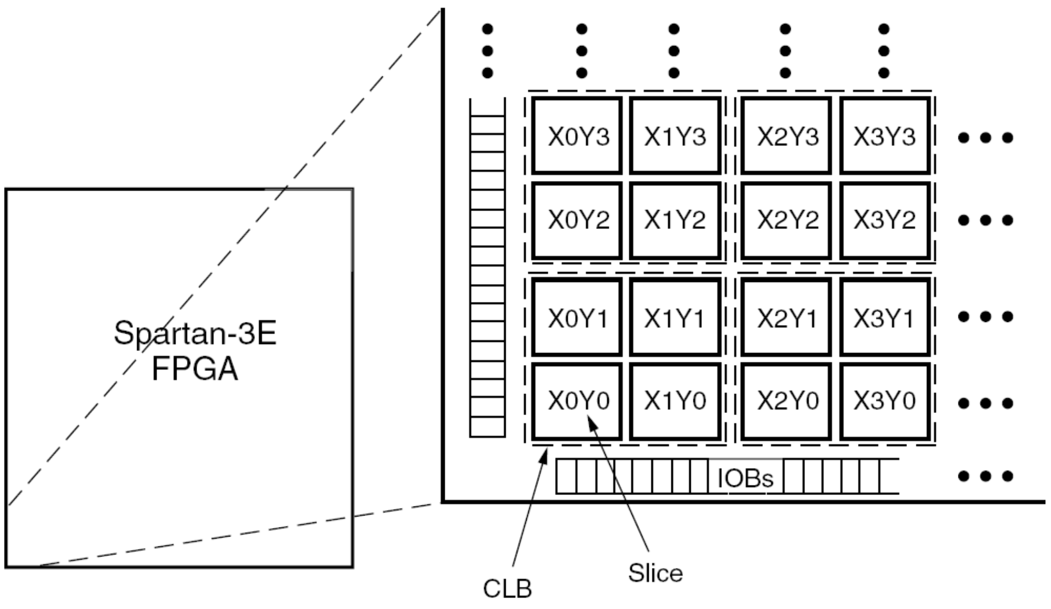


Figure 3.1: Spartan-3 layout of slices and CLBs. Source: [23].

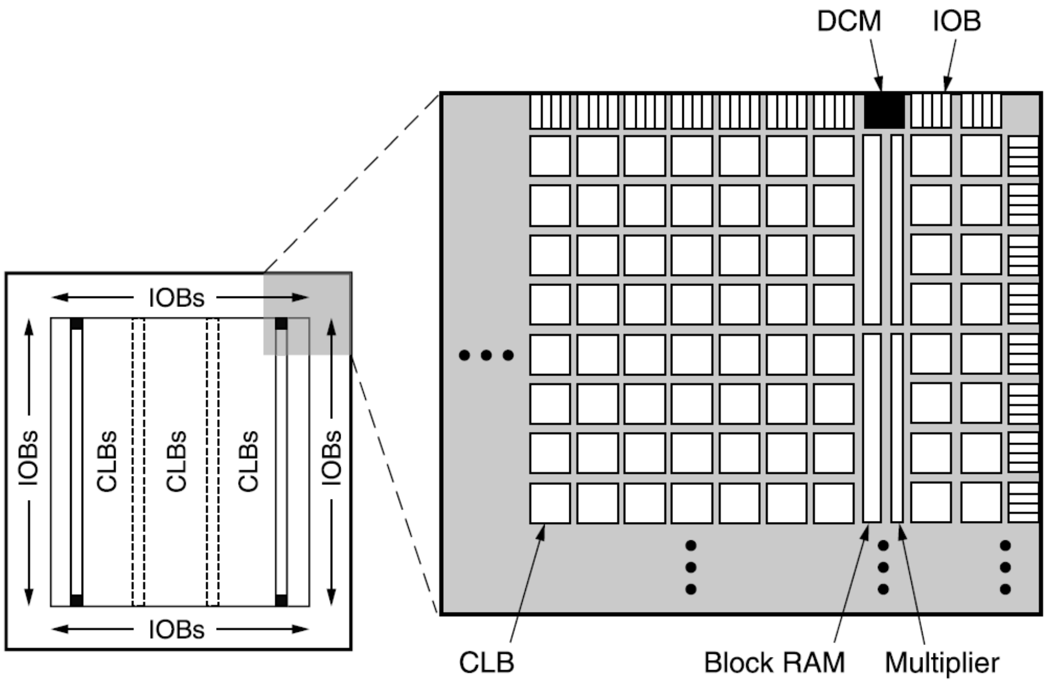


Figure 3.2: Spartan-3 layout of DCMs, block RAMs and multipliers. Source: [22].

viding 1 MB RAM. How the RAM is used in practice depends on the configuration of the FPGA.

The XC3S400 has 16 global clock inputs that can be fed by crystal oscillators. This clock signals can be fed to the four global DCMs. These DCMs can divide, multiply and phase shift the clock signal.

3.5.3 Softcores

An FPGA is programmed by writing code in a Hardware Description Language (HDL) like VHDL or Verilog. This code is being synthesized and a file is generated that contains the configuration of the LUTs and all other configurable elements on the FPGA.

It is possible to write a CPU in a HDL. Sijmen Woutersen has written the X32 softcore for the Spartan-3 [20] and Xilinx has its 32-bit MicroBlaze and 8-bit PicoBlaze cores that fit on a Spartan-3. The PicoBlaze core will be too limited for control applications because its maximum available amount of data memory is 64 bytes and its code can be at most 1K instructions. The X32 is able to run at about 3.7MIPS and the MicroBlaze can do 85 MIPS (68 Dhrystone MIPS) [21, page 2]. The X32 is freely available; the MicroBlaze Embedded Development Kit containing the MicroBlaze core costs $495 and allows the buyer to use the MicroBlaze core on a royalty-free basis.

This research does not cover softcores because softcores are not able to compete with hardcores and we want to evaluate FPGAs for their ability to build hardware at gate level. On the other hand, a design consisting of mostly HDL designed hardware for the dataprocessing and a softcore to control the hardware could be an option. This is however not considered in this research.

This research does not cover softcores because the FPGAs are considered because we want to evaluate them for the ability to create hardware at gate level and we do not want to evaluate softcores. Besides that, softcores will most probably be slower than the same core implemented as ASIC.

3.6 Referring

The chips described in this chapter can be called by various names. The LPC2144 for example can be called after its model name, LPC2144, or its series name LPC214x or its architecture, ARM7. In this thesis when the full name is used (PIC18F8722, ATmega128, MSP430F2619, LPC2144, XC3S400) that specific version of the chip is meant. When the series or architecture name is used (PIC18F, ATmega, etc.) then the more general series or architecture is meant and what is written holds for all variants of the series or architecture. In principle, there is no difference between the series or the architecture, except for the LPC214x/ARM7 because the ARM7 architecture is used by more manufacturers and series.

3.7 Summary

Five different platforms have been introduced to serve as basis for this research. There are 8, 16 and 32-bit CPU architectures with both Modified Hardvard and Von Neumann memory architectures and there is an FPGA. Table 3.1 summarizes the different specifications of the microcontrollers and the FPGA.

Chip	**PIC18F 8722**	**ATmega 128**	**MSP430 F2619**	**LPC2144**	**Spartan-3 XC3S400**
Native data size	8-bit	8-bit	16-bit	32-bit	18-bit[a]
Memory architecture	Modified Harvard	Modified Harvard	Von Neumann	Von Neumann	N/A
Flash	128 KB	128 KB	120 KB	128K B	None
RAM	4 KB	4 KB	4 KB	16 KB	1 MB
EEPROM	1 KB	4 KB	256 B	none	none
Working registers	1	32	16	16	N/A
Max addressable memory	2 MB	128 KB internal + 64 KB external	1 MB[b]	4 GB[c]	N/A
Max clock frequency	40 MHz	16 MHz	16 MHz	60 MHZ	100 MHz
MIPS	10	<16	<8	54	85
SPI/I^{2}C	2	1/1	2	1/2	[d]
UART	2	2[e]	2	2	[d]
ADC channels	16	10	8	14	none
PWM channels	12	6	11 + 2 DAC	6	[d]
Multiplier	8×8 1 cycle	8×8 2 cycle	16×16 3 cycle	32×32 multi-cycle pipelined	16 18×18
Price	€5.40	€7.00	€5.90	€6.48	€18.49

Table 3.1: Overview of the specifications of the diverse platforms

[a]Native size of the multipliers. All other operations can be of any size.
[b]Theoretical maximum of the MSP430X core. The MSP430F2619 has no external memory interface
[c]Theoretical maximum of the ARM7 core. The LPC214x series has no external memory interface.
[d]No dedicated interfaces but the designer can make as much as needed.
[e]These interfaces can also be used as SPI interface

Chapter 4

Detailed machine model

This chapter describes the detailed model of the PIC18F, ATmega and MSP430.

First, some assumptions used in this research will be stated. Then the diverse operations of the model will be described. The derivation of the model will not be covered; it is described in full detail in Appendix A. Finally a small example will be given and a summary of the chapter. The model is summarized in Table 4.1.

This chapter only describes the PIC18F, the ATmega and the MSP430 because the ARM7 was added later in this research when this part was already finished. Creating a detailed machine model of a target is a time consuming actitivity and creating another detailed model would not yield much more information. The choice was made to directly make an abstract model for the ARM7. This is described in Section 5.9.

4.1 Choices and assumptions

This research is done from the viewpoint of a designer who has to design a control system and wants to know which type of controller would be the best for his design. Within this context the following is assumed:

- The situation is a control situation: there is a stream of incoming data (for example from sensors), calculations are done with this incoming data to create output data in any form (for example a PWM signal, RS232 communication, a display device, etc…).
- There is a predefined algorithm that has to be implemented. For example: a specific kind of filter or a specific type of controller.
- Mathematics is fixed point, because floating-point calculations have to be done in software and this would cost too much time on these microcontrollers.
- If a microcontroller will be used, it will be programmed in C and compiled with a standard compiler.
- There are no ‘random’ interrupts.

- There are no cache effects.
- The execution is completely linear; there is no out-of-order execution.
- The goal is to get an estimate of the performance of the algorithm for different platforms, to be able to make a well-founded decision for a certain platform that implements the algorithm in a design.

The compilers used in this research are from IAR Systems. IAR Systems offers a complete development environment for various embedded devices. The PIC18F, AVR and MSP430 versions of the IAR compiler define a char as 8 bits, an int as 16 bits and a long as 32 bits. This definition follows ANSI C[1], but the ANSI C definition provides possibilities for different implementations, therefore the size of a datatype will always be mentioned.

4.2 Basic arithmetic

To determine the execution time of a piece of code the code is compiled for the given target and the costs in terms of cycles are counted by simulating the code and counting the number of cycles for each line. Based on the resulting costs of each line the costs of the different operations are determined. The following piece of code demonstrates this:

C code
```
h = a + b;              // 2 cycles
h = a + b + c;          // 3 cycles
h = a + b + c + d;      // 4 cycles
h = a + b + c + d + e;  // 5 cycles
```

When this would be the result of the cycle counting it is safe to decide that an assignment costs 1 cycle for this datatype and each addition also 1 cycle for this datatype. Care has to be taken that the compiler doesn't do unexpected things (like placing a variable in the data memory instead of keeping it in a register), which would lead to a wrong conclusion. Also not all situations are as straightforward as this one, the PIC18F for example has a `SUBWF` instruction to subtract `WREG` (the working register) from `F` (a value in the data memory) but no instruction to subtract `F` from `WREG`. The effects on the cycle count are demonstrated in the following example:

C code
```
h = a - b + c + d; // 5 cycles
h = a + b - c + d; // 7 cycles
h = a + b + c - d; // 7 cycles
h = a + b + c + d; // 5 cycles
```

[1][7, page 36] "The intent is that `short` and `long` should provide different lengths of integers where practical; `int` will normally be the natural size for a particular machine. `short` is often 16 bits long, and `int` either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its own hardware, subject only to the restriction that `shorts` and `int` s are at least 16 bits, longs are at least 32 bits, and `short` is no longer than `int`, which is no longer than `long`."

In line 1 and 4 everything is fine, 2 cycles for the assignment and 1 for the addition or subtraction. In line 2, the compiler lets the microcontroller calculate `a + b`, store the result in the memory, load `c` in `WREG`, subtract the result of `a + b` and then add `d`. The same goes for line 3. Cases like this are mentioned as exception.

By following this method the costs of assignments, addition, subtraction, bitwise *or*, bitwise *and* and bitwise *xor* are determined. During this process a couple special cases were found:

- Using a constant: using a constant in an operation usually costs more, because the constant has to be generated using a special instruction. The PIC18F for example has the `MOVLW` which puts a literal into the working register. Constants like 0 and 0xff are sometimes easier to generate than arbitrary numbers. This can save cycles, for example when some bytes of the constant are zero. On the PIC18F the zero-bytes are generated using the `CLRF` instruction. This instruction can directly put zero on a location in the data memory instead of first generating the constant and then moving it to a memory location. This saves one cycle for each zero-byte. The MSP430 something more special: two constant generator registers. Dependent on the addressing mode these registers can generate -1, 0, 1, 2, 4 and 8.

- Using a global variable: on multi-register architectures local variables are kept in registers. Global variables are stored in the data memory. To use these variables a load or store instruction has to be used. This causes extra costs compared to local variables.

- The statement is only an assignment: in an assignment-only instruction the compiler can sometimes save cycles by using a different instruction for the assignment.

- The statement has only two terms: in some cases cycles can be saved in the same way as with an assignment-only statement.

Using parentheses comes with some costs if the order of operation is changed. The IAR compiler does not optimize the expression to overcome these costs so they have to be counted. The costs only occur if order of operation is changed. For example when calculating $(a+b)*c$ the parentheses have to be counted while when calculating $a+(b*c)$ they don't have to be.

4.3 Multiplication and bit shifting

Multiplication is usually more expensive, surely when the datatype is larger than the architecture because in that case multiple multiplications have to be done, like when multiplying two multi-digit numbers by hand on paper. In the case of the MSP430 the multiplier is implemented as peripheral, which means that the two numbers to be multiplied have to be moved to a memory location to start the multiplication. After a few cycles the result can be retrieved from another memory location. This is a quite expensive process; multiplication of two 16 bits numbers costs 27 cycles.

There is one special case for multiplication: when the statement is a two-term multiplication-only expression cycles can be saved with respect to expressions with multiple terms.

Bit shifting is sometimes used as cheap multiplication. By shifting one position a number is multiplied or divided by 2.

A common operation when using fixed-point arithmetic is bit shifting. In fixed-point representation the amount of digits after the point is fixed to a predefined value. When for example a 16 bits datatype is used and the position of the point is defined as between the 12th and 13th digit (so there are 4 numbers after the point) the number can look as follows: 000000010100.0100. This number represents the value of $0 \times 2^{11} \ldots 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} = 20.25$.

Addition and subtraction of fixed-point numbers is exactly the same as with normal numbers. Multiplication is different: after the multiplication the number has to be shifted to the right with the amount of numbers after the point. Also to convert normal data to fixed-point data these numbers have to be shifted to the left by the amount of numbers after the point. So, bit shifting is a frequently used operation.

When shifting with a fixed amount the compiler usually tries to use a method that costs as little time as possible. This results in non-regular varying cycle times for the bit shifting operation. Therefore, the costs of bit shifting are given as range; when a statement contains a bit shift operation the resulting costs are a range too. When this does not provide sufficient detail, the real costs can be looked up in a table (see Section A.2.4 for all details).

When shifting with a variable amount the costs are a formula linearly dependent on the shift amount.

4.4 Function calls

The costs of function calls depend on return type, the number and size of the passed arguments and the number of local variables in the function.

The function call itself is listed in Table 4.1 by its return type. Then the costs of the arguments have to be added until the “Maximum amount of arguments not on the stack” is reached. Beyond that amount the arguments have to be passed via the stack instead of via the registers.

The number of local variables determined how much registers have to be saved. When the function has very little local variables, they can be stored in the not-saved-across-call registers. When the function has more also the saved-across-call registers are used. To save these registers a context-save has to be done. The context-save costs are dependent on the number of registers, but in most cases either the function is so small that no registers have to be saved or the function is so large that all registers have to be saved.

There are a two special cases: the return value of the function is a constant and the argument is a constant. In both cases a constant has to be generated, which comes with extra costs, as seen earlier.

The PIC18F only has a working register, which is not saved across the call, so no context save has to be done, which is a great advantage.

4.5 Interrupts

Interrupts are special kinds of function calls: the whole context has to be saved because the interrupted code may not notice being interrupted. Also, the interrupt is hardware-initiated and the costs of calling are implementation dependent. E.g. costs of the interrupt on the PIC18F are three times less than on the MSP430; 11 cycles versus 39 cycles plus a context save.

4.6 Boolean expressions and if statements

Boolean expressions are mostly used in if statements and as loop condition. The costs of boolean expressions are built up from some fixed costs ("Startup costs") caused by the executed jump instructions, the costs of comparing variables and the boolean operation between the results of the comparisons.

In the following situations there are extra costs or savings:

- Parentheses; when they change the order of operation.
- When comparing with a constant; for the generation of the constant.
- When the expression finally evaluates to false; because an extra jump has to be executed.
- If the expression evaluates to true and has an else part; because the else part has to be skipped.
- Extra costs for a function call, like `if (theanswer() != 42)`.
- For a short two-term expression, like `if (a==b)`.

4.7 Loops

The C programming language knows three loop constructs: while, do..while, and for. First every loop has startup costs, which count only once. The while and for loops evaluate their expression at the beginning of the loop and the do..while at the end. The loop condition is boolean expression, whose costs are determined according to the previous section. These costs are counted per evaluation. Further there are costs per loop iteration for jumping from the end of the loop back to the beginning. These costs are counted every loop iteration. In case of a for loop the loop initialization and increment statement have to be counted too, which works in the same way as normal arithmetic in Section 4.2.

4.8 A/D conversion

A/D conversion can generally be done in three ways:

1. Start the conversion and wait until it is finished.

2. Put the A/D converter in continuous sample mode and read it when a sample is needed.

3. configure the microcontroller in such a way that an interrupt is executed when the conversion is finished.

In the first case, the speed of the algorithm is limited by the configured sample frequency of the A/D converter. In the second case the costs are only the costs of reading a special function register, which is about 2 or 3 cycles. In the last case, the costs are higher: 11-48 cycles, dependent on the microcontroller.

More details about the specific differences between the microcontrollers related to A/D conversion can be found in Section A.3.3.

4.9 PWM output

PWM or Pulse Width Modulation is used to provide a semi-analog output or to control servos. The costs of this are important because PWM can be used as output for a control algorithm. The costs are low, because setting the value is only a matter of changing the value of a special function register.

4.10 Using the model

With the gathered information the costs of a code line can be determined. Consider the following code:

C code

```
1  char i;
2  char b, c;
3  b=4;
4
5  for(i=0;i<100;i++) {
6          c = c + b + i;
7  }
```

Line 1 and 2 contain declarations, these lines don't do anything and incur no costs. Line 3 is an assignment of a constant to an 8 bits char variable. For this line the costs are a summation of the operations named "Assignment", "Extra for the use of a constant" and "Extra for a short, assignment-only instruction", listed under "Char (8 bits)".

Line 5 is a for loop; the costs of the initialization statement `i=0` are counted just as line 3. The costs of the `i++` part are calculated by summing the costs of "Assignment", "Addition/subtraction" and "Extra for the use of a constant" (because `i++` means `i=i+1`). `i++` is executed 100 times, so it has to be counted 100 times.

The for loop is a loop, so for the initialization there are the loop "Startup costs", per iteration (100 times) the "Per loop"-costs. The contents of the loop, in this case only line 6, also are counted 100 times.

The costs of line 6 consist of an "Assignment" and two times an "Addition/subtraction".

This is all that is needed to calculate the costs of this piece of code. The time spent on calculating this is about a half an hour. For larger pieces of code a spreadsheet with the operations as column headers and the codelines as row headers can be used. The other cells are filled with numbers to indicate how much times a certain operations happens in a certain code line.

When the spreadsheet is completely filled in, all numbers are added up to get the total of how much times a certain operation happens in the complete code. Multiplying this with the costs of each operation gives the total time costs of the code. The complete calculation takes no more than an hour.

4.11 Summary

Diverse operations have been described. The complete list of operations and cycle counts for these operations is in Table 4.1. This table lists 73 possible operations with their costs. To determine the costs of an algorithm one has to determine the operations that take place in every line of code of that algorithm and sum up the numbers to get the total cycle count. When the code contains bit shifting operations that shift by a fixed amount the result will become a range, because the cycle count of such a bit shift operation cannot be given as a simple number or formula. Section 4.10 gives an example of using the model. The 7-line example code will cost at most one hour to analyze.

	PIC18F	**ATmega**	**MSP430**
Char (8 bits)			
Assignment	2	2	3
Addition/subtraction	1 or 3[b]	1	1
Or/and/xor	1	1	1
Extra for the use of a constant	0	0	1
Extra for the use of a global variable	2	2	2
Extra for a short, assignment-only instruction	0	0	-2
Multiplication	2	1	29
Parentheses	4	1	1
Left shift	1 to 45	1 to 3	0 to 6
Right shift	2 to 6	1 to 3	1 to 5
Variable left shift	$12+7n$	$15+6n$	$12+7n$
Variable right shift	$12+9n$	$15+6n$	$12+9n$
Pointer	14	1	0
Int (16 bits)			
Assignment	6	2	3
Addition/subtraction	4	2	1
Or/and/xor	3	2	1
Extra for the use of a constant $\leq$ 0xff	-1	-1	1
Extra for the use of a constant $\leq$ 0xffff	0	0	1
Extra for the use of a global variable	2	4	2
Multiplication	13	20	27
Short two-term expression	-2	0	-2
Multiplication only expression	-8	-1	0
Parentheses	4	1	1
Left shift	-1 to 114	2 to 48	0 to 9
Right shift	-1 to 144	2 to 48	0 to 7
Variable left shift	$15+7n$	$14+6n$	$15+7n$
Variable right shift	$15+9n$	$14+6n$	$15+9n$
Pointer	18	1	0
Long (32 bits)			
Assignment	12	4	6
Addition/subtraction	8	4	2
Or/and/xor	7	4	2
Extra for the use of a constant $\leq$ 0xff	-2	4	-1
Extra for the use of a constant $\leq$ 0xffff	-1	4	1
Extra for the use of a constant $\leq$ 0xffffff	0	4	2
Extra for the use of a constant $\leq$ 0xffffffff	0	4	2

Continued on next page

Table 4.1: Summary of all numbers for all three microcontrollers

[b] 1 cycle when the operation is the first operation or when a constant is involved, 3 otherwise.

	PIC18F	**ATmega**	**MSP430**
Extra for the use of a globale variable	2	8	4
Mutiplication	40 to 740	53	45
Short two-term expression	-2	0	0
Multiplication-only expression	0	-2	0
Parentheses	4	2	2
Left shift	9 to 149	4 to 124	0 to 40
Right shift	9 to 183	4 to 124	0 to 40
Variable left shift	$23+9n$	$16+8n$	$23+9n$
Variable right shift	$23+11n$	$16+8n$	$12+11n$
Pointer	21	1	0
Boolean expressions/if			
Startup costs	3	2	0
Per evaluated comparison ($==$, $!=$, $<$, $>$, $=>$, $=<$) char or boolean (8 bits)	1	1	3
Per evaluated comparison ($==$, $!=$, $<$, $>$, $=>$, $=<$) int (16 bits)	2	6	3
Per evaluated comparison ($==$, $!=$, $<$, $>$, $=>$, $=<$) long (32 bits)	4	12	6
Parentheses	1	0	0
Comparison with a constant	0	int 1, long 3	1/byte
Expression evaluates to false	1	1	0
If statement evaluates to true and has an else part	2	2	2
Extra costs for a functioncall	-1	-1	0
Boolean and/or	1	1	3 or 5[c]
Short two-term expression	-1	-2	0
Loops			
Startup costs	2	0	0
Per loop[d]	1	2	2
Function calls			
Returning void (“0 bits”)	5	7	7
Returning char (8 bits)	6	9[e]	9[e]
Returning int (16 bits)	7	10[e]	9[e]
Returning long (32 bits)	9	13[e]	11[e]
Extra when the return value is a constant	0	1/byte	1/word

Continued on next page

Table 4.1: Summary of all numbers for all three microcontrollers

[c] And: 3, or: 5

[d] Only the costs of the loop. Costs of the evaluation expression should be added seperately.

[e] Including the assingment of the function result.

	PIC18F	**ATmega**	**MSP430**
Context save	0	$29+4r$[f]	$15+2r$[g]
Per char (8 bits) argument	n/a	1	1
Per int (16 bits) argument	n/a	1	1
Per long (32 bits) argument	n/a	2	2
Max amount of arguments not on the stack	0	8 bytes	4 words
Costs for arguments on the stack	2/byte	2/byte	3/word
When an argument is a constant	0	1/byte	1/byte
Overig			
Interrupt	11	39	16[h]
Reading the A/D converter using the interrupt	13	48	16(+10)[i]
Reading the A/D converter without interrupt	2	2	3
Setting the value of a PWM output	8	2	3

Table 4.1: Summary of all numbers for all three microcontrollers

[f] r is the number of registers that have to be saved; $r = [1,16]$

[g] r is the number of registers that have to be saved; $r = [1,8]$

[h] Plus a context save, if applicable.

[i] The 10 cycles are the costs of the code in the interrupt routine.

Chapter 5

Abstract machine model

As seen in the previous chapter, it is possible to determine the execution time of an algorithm written in C. As we shall see in the next chapter, the precision is quite good. That method involves 73 different operations. One of the observations is that often, there is a linear relation between the size of the data type and the costs of the operation. Consequently, it should be possible to extract the data size as parameter and reduce the number of different operations. Besides that, the 73 operations contain quite some special cases that happen very infrequent and save or add one cycle to the total. Leaving these out causes a very small, negligible, error. This error is so small that it will not cause any problems.

In this chapter, we will also introduce the ARM7 microcontroller core as used in the NXP LPC2144. This microcontroller was added in a later stage of this research and we will directly make an abstract model of this core without making a detailed model first.

At the end of this chapter, we will derive a model for calculating the power consumption of microcontrollers based on the calculated time consumption and the resulting required clock frequency of the microcontrollers.

5.1 Definitions

In this chapter the following definitions are used: B is the bit size of the architecture, which is 8 for the PIC18F and the ATmega, 16 for the MSP430 and 32 for the ARM7. N is the bit size of the operation: 0 for void, 8 for char, 16 for int and 32 for long. The sizes of an int and a long depend on the compiler; the PIC18F, ATmega and MSP430 compilers take an int as 16 bits and a long as 32 bits, the ARM7 compiler takes a short as 16 bits and an int as 32 bits. Therefore the size of a datatype is always specified.

As the time cost of operations is often determined by both B and N, we define the following two parameters V and V_2. V is defined as:

$$V = \left\lceil \frac{N}{B} \right\rceil \tag{5.1}$$

and V_2 as

$$V_2 = \left\lceil \frac{N}{2B} \right\rceil \tag{5.2}$$

V indicates the ratio between the size of the architecture and the size of the operation and V_2 indicates the ratio between the size of the architecture and twice the size of the operation. Execution times are referred to as t and expressed in cycles, unless stated otherwise. The reason that t is specified in cycles and not in seconds is because the time cost in seconds depend on the clock frequency. If an operation takes 1000 cycles, a microcontroller runs at 1 MHz and executes 1 cycle per clock tick then the time costs are $\frac{1 \cdot 1000}{1 \cdot 10^6} = 0.001\,\text{s}$. When calculating the time cost in seconds one has to take the clock frequency of the target into account.

5.2 Neglected operations

From Table 4.1 the following operations have been neglected:

- Extras related to advantages for short expressions and multiply only expressions.
- Costs when an if statement evaluates to false or when it has an else part that is not executed.
- The special case of returning a constant from a function, this can be counted as normal constant.
- Functions with too much arguments, such that these have to be stacked. Function arguments in the PIC18F always have to be stacked so they will be considered as normal arguments.
- Reading the AD converter or setting a PWM value will not be listed.

These operation have minimal influence on the result. Therefore it is safe to neglect these operations for the sake of simplicity of the model and to save time on modelling the code.

Modeling the costs of a certain operation can be straightforward but there are cases where it is more difficult. This chapter describes the process of modeling the costs of the operations for the PIC18F, the ATmega and the MSP430. The models are based on the results of the detailed modeling, listed in Table 4.1. The final results can be found in Tables 5.2, 5.3 and 5.4.

5.3 Basic arithmetic

Assignment is an operation linear in V. For every multiple of the native data size, the microcontroller has to do the same operations again. When, for example, assigning the contents of a 16 bits variable to another 16 bits variable on an 8 bits microcontroller the microcontroller first moves the first 8 bits and then the second 8 bits.

Addition and subtraction are also linear in V. Although the costs are usually the same for an addition or subtraction as they are for an assignment, there is a structural difference between those two operations. In case of an assignment there is no special order in which the parts of the variable have to be transported. In case of an addition there is the carry to keep in mind. Microcontrollers have *add* and *add with carry* instructions for this. When adding two 16-bits numbers on an 8 bits microcontroller, the least significant bytes are added using the *add* instruction and the carry is stored in some status register. Then the most significant bits are added together with the carry using the *add with carry* instruction.

Bitwise *and*, *or* and *xor* are linear in V. For each B bits an `and`, `or` or `xor` instruction has to be executed. Boolean *and* and *or* are slightly different because in C false is defined as zero and true as anything but zero. The compiler has to take care of this, because just using a bitwise `and` or `or` instruction could give wrong results. On the other hand, when using the C99's bool type as defined in `stdbool.h` the compiler can better handle booleans and maintain consistency so that the `and` and `or` instructions can be used to handle boolean variables.

On some microcontrollers, using a constant involves extra costs, linear in V. This is because instructions do not have the ability to directly use an ordinary constant in something like an add instruction. These microcontrollers have an instruction to load a constant in a register and then this constant can be used in a calculation. This costs an extra instruction per B bits with respect to using a variable.

Using a global variable usually involves additional costs, linear in V because the compiler maps global variables to memory locations. When the variables need to be used, they have to be loaded from the memory into a register before they can be used. Instructions to do this have to be executed for every B bits.

The next subsections describe the platform-specific details.

5.3.1 PIC18F

Assignments, addition, subtraction and boolean operations are linear, as expected. Because all variables in the PIC18F are stored in the data memory and loading a variable from the data memory costs the same time as loading a constant in the working register, using a constant does not involve extra costs. The costs of global variables are not linear for the PIC18F because the PIC18F is different from the other microcontrollers. Global variables are usually mapped to a different memory bank. Therefore, to access a global variable, the compiler has to switch the active memory bank before and afterwards. This is not dependent on the size of the variable. So, the costs of using a global variable on the PIC18F are fixed.

5.3.2 ATmega

The ATmega has advantage from its `MOVW` instruction and this is visible in the costs of the assignment. A 16-bit int assignment is as expensive as a 8-bit char assignment and a 32-bit long assignment is twice as expensive as an assignment, because it is twice as large. Because of this, the V_2 constant is used in the model of the assignment. Addition, subtraction and boolean operations are linear in V, as expected, and so are the costs of using a constant and a global variable.

5.3.3 MSP430

The MSP430 is a 16 bits microcontroller and its CPU core can execute operations on 8 bits of data as fast as on 16 bits. This is visible in the cycle counts in Table 5.4. For nearly all operations, the cycle count is equal for 8 bits chars and 16 bits ints. This is contained in the formula for V: V is defined as $V = \lceil \frac{N}{B} \rceil$ and for this microcontroller B is 16 instead of 8 for the PIC18F and the ATmega. This means that for both chars ($N = 8$) and ints ($N = 16$) the value of V is 1.

The costs of assignments, addition, subtraction, and boolean operations are linear in V and the costs of using a constant or a global variable are linear in V too.

5.4 Multiplication

The costs for multiplication on the PIC18F, the ATmega and the MSP430 are listed in Table 5.1. For multiplication, a quadratic relationship is expected between the size of the operands and the costs. This is because when two numbers of length nb have to be multiplied using a $b \times b$ multiplier, n^2 multiplications and some additions are needed. This is described in [13, Chapter 12.1] for hardware-implemented multipliers but also applicable to this case. This is the same principle as used when multiplying multi-digit numbers by hand on paper.

However, for the ATmega, experimental data shows no strict quadratic relationship. Figure 5.1 shows the costs, with a linear ($t = 17.214V - 15.500$) and a second order ($t = 3.262V^2 + 1.833$) fit[1].

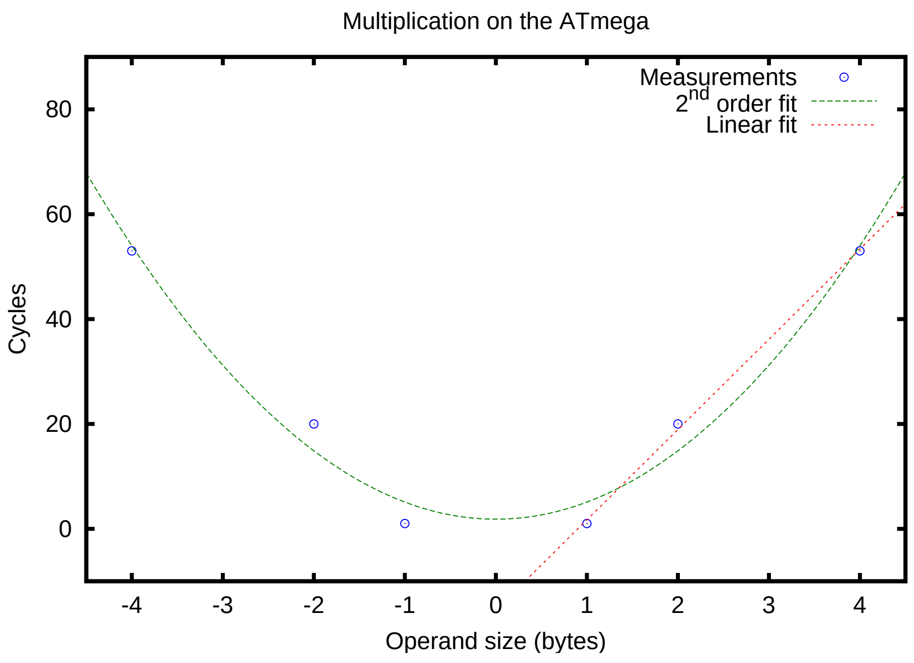


Figure 5.1: Multiplication on the ATmega

[1]Note that the coefficients of models are real-valued. The reason for this is that the values are built-up from diverse factors and ratios. This results in non-integer coefficients.

	Char 8 bits	**Int 16 bits**	**Long 32 bits**	**Model**
PIC18F	2	13	40-740	$3.881V^2 - 2.1667$
ATmega	1	20	53	$2.333V^2 + V + 3V_2 + 5.667$
MSP430	29	27	45	$5.667V^2 + 22.33$

Table 5.1: Cycle times for multiplication

The linear fit seems to be more correct than the second order fit. The reason for this is that the ATmega uses its `MOVW` instruction and the result of the multiplication is stored in a variable of the same type as the operands. As we will see the final model also contains a V_2 parameter, which means that both the linear and second order models are incorrect.

When the result of a multiplication is stored in a variable of the same size as the operands, for example when multiplying two 32-bit longs, the result, that could be 64 bits, is truncated and stored in a 32-bit long. No matter what would be in the 32 most significant bits of the result, they are ignored. From the compiler's viewpoint it is useless to calculate anything that would not be used in the 32 least significant bits so these calculations are not done at all.

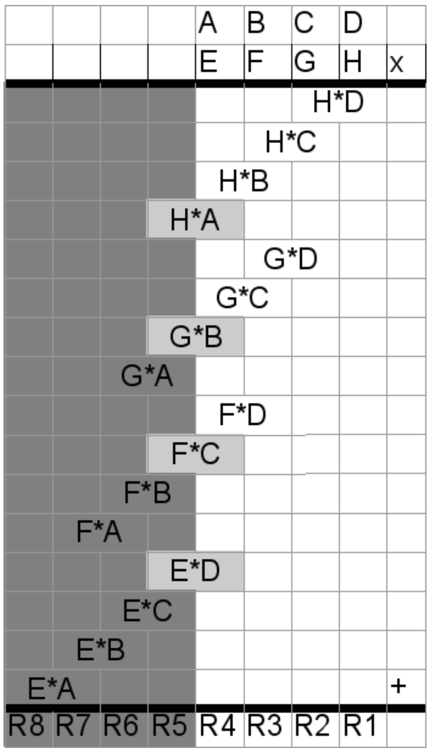


Figure 5.2: Multiplication of two four-byte numbers using a 8×8 multiplier

Figure 5.2 demonstrates how the number ABCD is multiplied by the number EFGH using a 8×8 multiplier. In these numbers, each character stands for one byte. Because the result will be stored in 32-bits (=4 byte) the calculations leading to R5-R8 are not needed and do not have to be executed. These are all calculations with a dark-gray background. The multiplications with a light-gray background have to be executed but the high byte of that result is not used. In this way, for a 32-bits long multiplication, only 10 out of 16 multiplications have to be done. For a 16 bits int multiplication, this is 3 out of 4. Putting this in a formula gives $m = \frac{1}{2}V^2 + \frac{1}{2}V$ where m is the number of multiplications and V the ratio between the size of the operands and the size of the multiplier, as given in the first paragraph of this section.

The results of the multiplications have to be added up, which involves 20 operations (move, add and add with carry) for a 32 × 32 bits long multiplication and 4 operations for a 16×16 int multiplication. This can be put in a formula as $a = \frac{4}{3}V^2 - \frac{4}{3}$, where a is the amount of addition and movement operations The results of this formula exactly correspond to the expectations: for $V = 1$, where no additions or whatsoever are needed, the result is 0, for $V = 2$ the result is 4 and for $V = 4$ the result is 20.

On an ATmega multiplications cost 2 cycles each and the addition operations cost 1 cycle each, so the costs spent on a 32 bits long multiplication are $t = a + 2 \times m = 20 + 2 \times 10 = 40$ cycles. The other 13 cycles are spent on moving operands before and after the execution of the multiplication routine, calling and returning. For a 16 bits int multiplication, the multiplication costs $3 \times 2 + 4 = 10$ cycles and the other 10 cycles are spent on moving operands, calling and returning. The operand-size dependent instructions used here are `MOVW` instructions, so the costs depend on V_2 instead of V. Putting these costs in a formula gives $t_{other} = 7 + 3V_2$. This V_2 parameter is the reason that the relation between the costs and the data size looks more linear than quadratic.

Summarizing the costs for multiplication on the ATmega: $t = 2m + a + t_{other} = 2(\frac{1}{2}V^2 + \frac{1}{2}V) + (1.333V^2 - 1.333) + 7 + 3V_2) = 2.333V^2 + V + 3V_2 + 5.667$. This formula corresponds to the results within one cycle for the 16 bits int and 32 bits long.

The PIC18F has no instruction to move multiple bytes, so the PIC18F is expected to conform better to the quadratic relation. That is the case, although the compiler chooses to do the multiplication of 32 bits longs completely using a right shift algorithm like the one presented in [13, Chapter 9.2]. The time costs of a multiplication using this algorithm vary from 40 to 740 cycles. The precise costs depend on the input data. The hardware multiplier is only used for the 8 bits char and 16 bits int multiplication.

The results of the ATmega can be used to predict the costs of multiplication on a PIC18F. Figure 5.2 will be used to illustrate this. On the ATmega the H*A, G*B, F*C and E*D related calculations cost 3 cycles. On the PIC18F the same calculations would cost 4 cycles (2 more cycles due to the W register and the instruction set, 1 less because a `MULWF` is only 1 cycle).

The H*B, G*C and F*D related calculations cost 4 cycles on the ATmega and 6 cycles on the PIC18F. The extra costs are because of an extra move or add instruction.

The calculations for H*C and G*D on the ATmega cost 5 cycles and the calculations for H*D cost 6 on the ATmega, so assuming the costs of these calculations on the PIC18F grow the same way, they would cost respectively 8 and 10 cycles. The total amount of cycles for a 32 bits long multiplication would become $4 \times 4 + 3 \times 6 + 2 \times 8 + 1 \times 10 = 60$ cycles.

With this there are three data points: $(V,t) = (1,2)$, $(V,t) = (2,13)$ and $(V,t) = (4,60)$. To transform this to a model GNU Octave is used; a software package compatible with MATLAB®. The resulting model should be a second order polynomial with a minimum at $V = 0$. Running the fitter on these three data points alone results in a wrong polynomial; the result has no minimum at $V = 0$. Therefore, the fitter needs more data to help it find the right polynomial.

A parabola with a minimum at $V = 0$ is symmetric in the line $V = 0$, therefore if the known data points are mirrored in this line, we would get more data points that would help the fitter to find the right formula. The resulting model is $t = 3.881V^2 - 2.1667$. This formula gives a nearly perfect result: $t(1) = 1.7143$, $t(2) = 13.3571$ and $t(4) = 59.9286$. Figure 5.3 gives a graphical representation.

The MSP430 has no special instructions like the ATmega's `MOVW`, so the model is quadratic. Multiplication of 8-bit chars costs 2 cycles more, because the compiler clears the upper byte of both operands. This gives two values for t when V is 1. This is solved by taking the average of both points and for the fit. Figure 5.4 shows the results: two data points for $V = 1$ and the line of the fit uses the average of 28. The resulting model is $t = 5.667V^2 + 22.33$. The size-independent part of the model is quite high. That is because the multiplications are done in a subroutine so there are extra costs involved in the call and return. The costs per multiplication are also quite high because the multiplier is implemented as peripheral, so if you want to use the multiplier you have to use quite expensive `mov` instructions to move the operands to the multiplier (4 cycles per 16-bits) and to take the result out (3 cycles per 16 bits).

5.5 Pointers

Pointers are one of the most important features of C. The ability to use a pointer is not always straightforward for microcontrollers. Some microcontrollers have no native support for pointers in their instruction set relying on special function registers for providing pointer support. Costs of using pointers can be:

- Zero, because the instruction set has the ability to use contents of registers directly as pointer.
- Fixed, because an extra operation is needed for the handling of the pointer.
- Linear in V because extra operations are needed for every B bits of the variable.

The PIC18F has three sets of indirect access registers. When the FSR*x*H and FSR*x*L ($x = 0..2$) registers are loaded with a memory address, the value of that memory address can be loaded from the INDF*x* register and some other registers providing pre/post increment/decrement. When using pointers in program code the compiler uses these registers to load or store the data pointed to by the pointer variable. Loading these registers takes quite some effort, so the fixed costs are high: the model is $t = 12.5 + 2.2V$.

The ATmega's architecture has quite good pointer support. The costs of using a pointer are one cycle more than when using a normal variable, independent of the length of the variable. The model for pointer dereferencing on the ATmega is a simple $t = 1$.

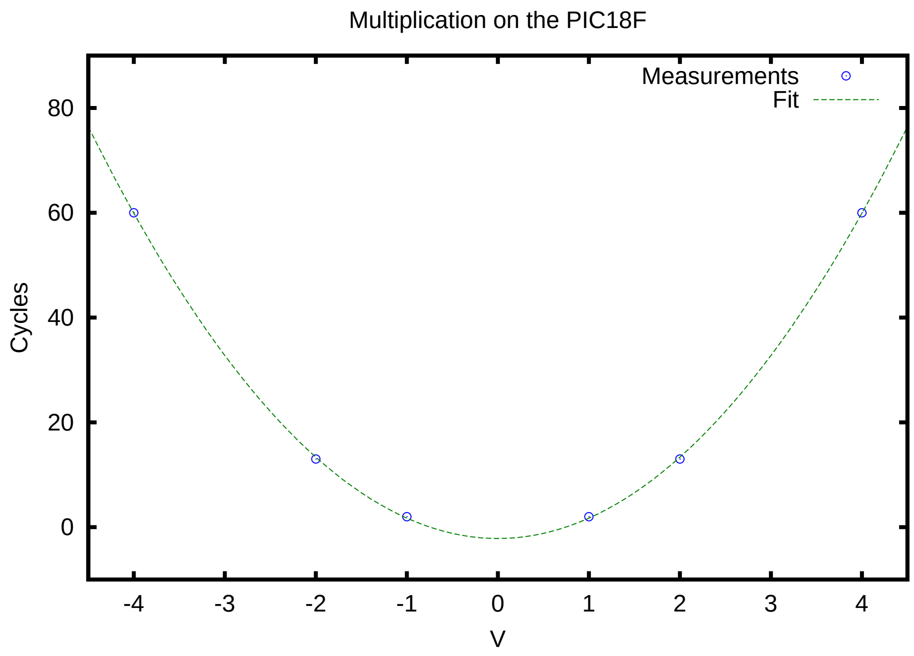


Figure 5.3: Multiplication on the PIC18F

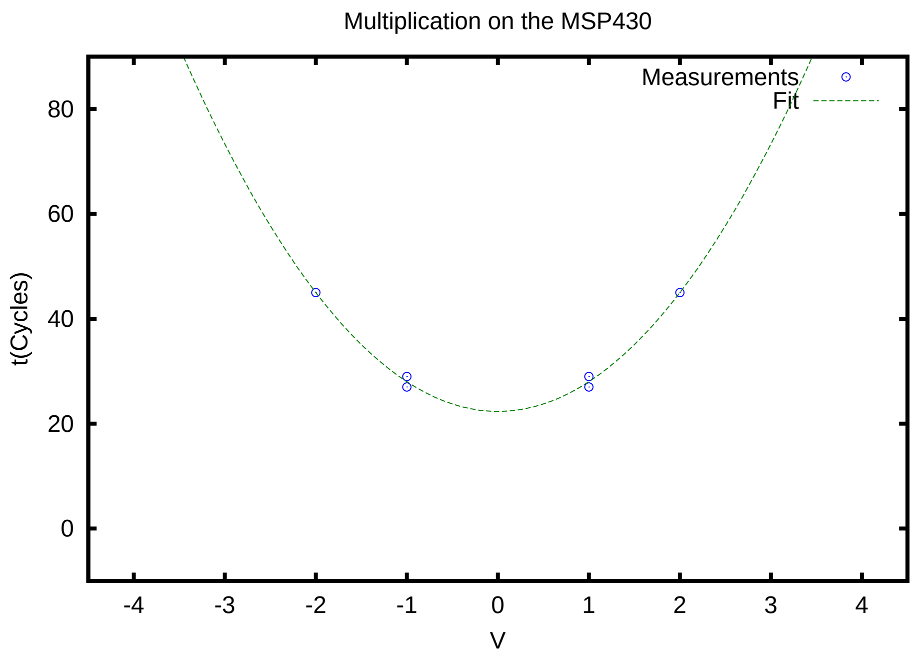


Figure 5.4: Multiplication on the MSP430

On the MSP430, the instruction set fully supports pointers; there are no extra costs for using a pointer. Therefore the model is $t = 0$ for dereferencing pointers on the MSP430.

5.6 Bit shifting

Bit shifting is used in fixed-point arithmetic and as cheap multiplication. Most microcontrollers have instructions to shift a register by 1 bit. To shift more positions the same instruction has to be executed multiple times. Therefore, a shift operation is linear in the amount of positions (p). To shift a multiple of the native data size the bit that is "shifted out" is put in the carry bit of a status register and shifted in when shifting the next part. This has to be done $V - 1$ times, so besides linear in p, the bit shifting operation is also linear in V. So the model of bit shifting will look like $t = (aV + b)(cp + d)$.

When shifting a constant amount, the compiler can optimize the shift operation. E.g.: when shifting one byte by 4 positions on the ATmega, the compiler can use the `SWAP` instruction to exchange the high 4 bits with the low 4 bits and then do a bitwise *and* with 0x0f. This costs less cycles than shifting 4 times. The drawback of this method is that the costs are nonlinear and more difficult to predict. In these cases a formula for the mean (μ) and standard deviation (σ) will be given.

5.6.1 PIC18F

For the PIC18F the model for bit shifting is $t = (V + 6)(p + 1)$. This formula works for left and right shifting of 16 bits ints and 32 bits longs. There are differences between left and right shifting, so actually it is $t = (V + 7)(p + 1)$ for right shifting and $t = (V + 5)(p + 1)$ for left shifting, but $t = (V + 6)(p + 1)$ is a good average. Figure 5.5 shows the results for ints and longs.

Bit shifting chars is highly optimized by the compiler, so this formula does not work, as shown in Figure 5.6. Because bit shifting is mostly used in fixed-point arithmetic and usually larger datatypes are used for this, an error is acceptable in this case.

5.6.2 ATmega

Determining a formula for bit shifting on the ATmega is difficult. Figure 5.7 shows a graph of the measurements. The graph shows that there is no real regularity in the measurements. The values for 8 bits chars show an alternating sequence of 2 and 3 cycles, for 16 bits ints the values float around 5 but for some shift amounts the costs are 30 to 50 cycles. The costs for 32 bits longs increase linear, except when shifting 7 to 9 positions. So the costs of the bit shift operation cannot be expressed in a formula giving exact results, but it is possible to express these costs as a random variable and give a mean value and a standard deviation. It is also possible to express these values in terms of V. The resulting formulas for the mean and standard deviation are $\mu_t = 17.221V - 17.231$ and $\sigma_t = 14.323V - 13.597$.

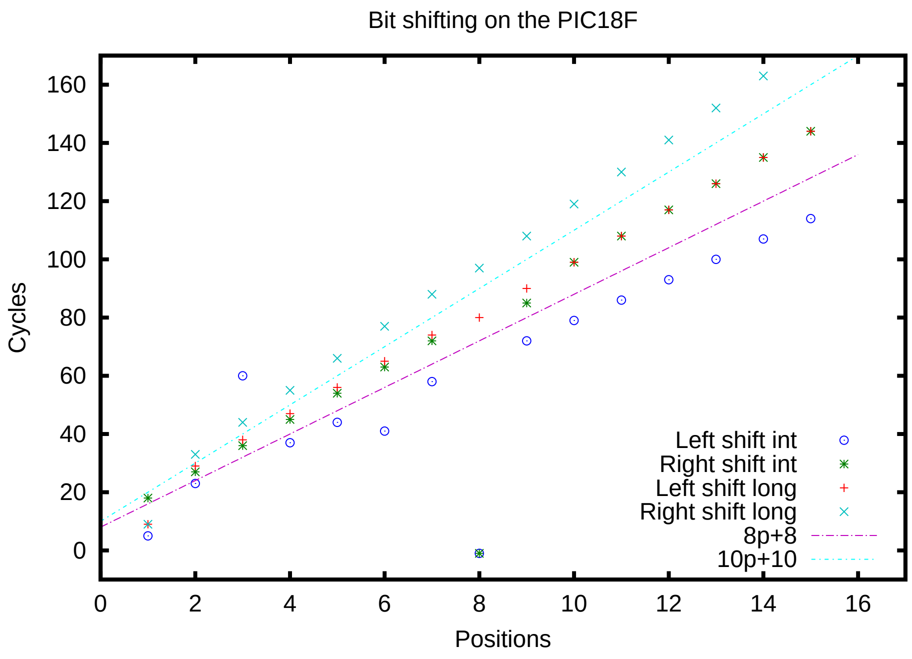


Figure 5.5: Bit shifting on the PIC18F for 16 bits ints and 32 bits longs

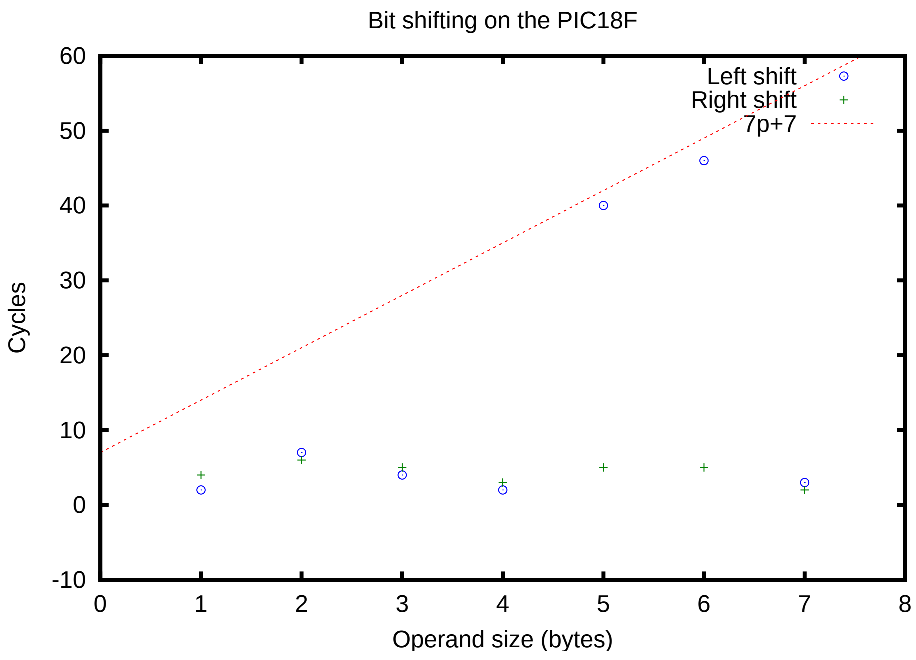


Figure 5.6: Bit shifting on the PIC18F for 8 bits chars

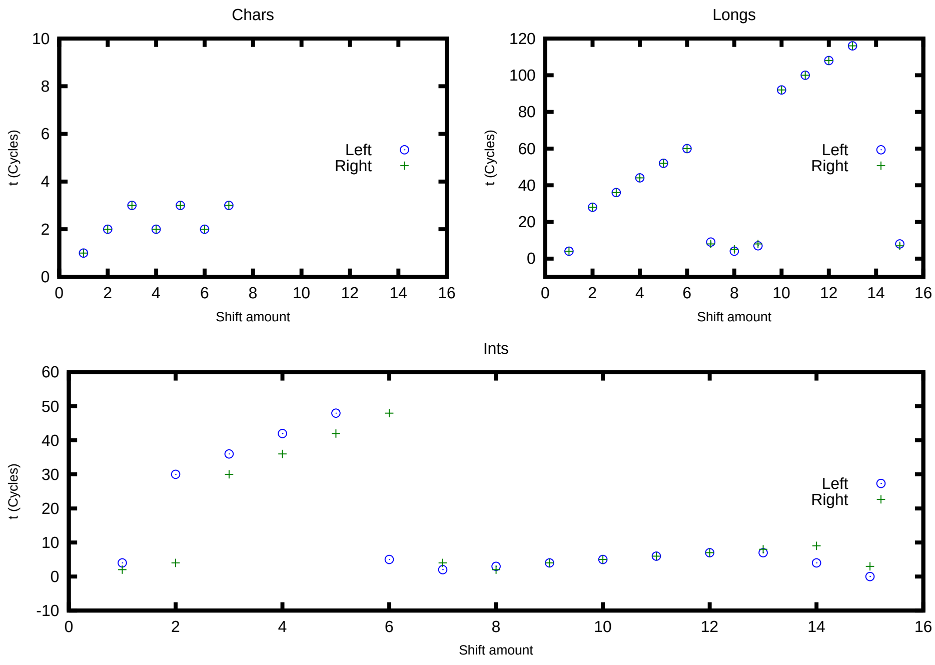


Figure 5.7: Bit shifting on the ATmega

5.6.3 MSP430

The costs of bit shifting on the MSP430 vary in the same way as on the ATmega. Figure 5.9 shows the measurements. For 8 bits chars and 16 bits ints the cycle count increments for the first 6 or 7 positions and then it jumps back to 1 or 2 and the same pattern repeats. For 32 bits longs, there is a jump between 4 and 5 and around 8. Because of these irregularities the costs of bit shifting can be expressed as mean and standard deviation, like with the ATmega. The formulas of the mean and standard deviation are $\mu_t = 15.405V - 12.176$ and $\sigma_t = 12.576V - 10.378$.

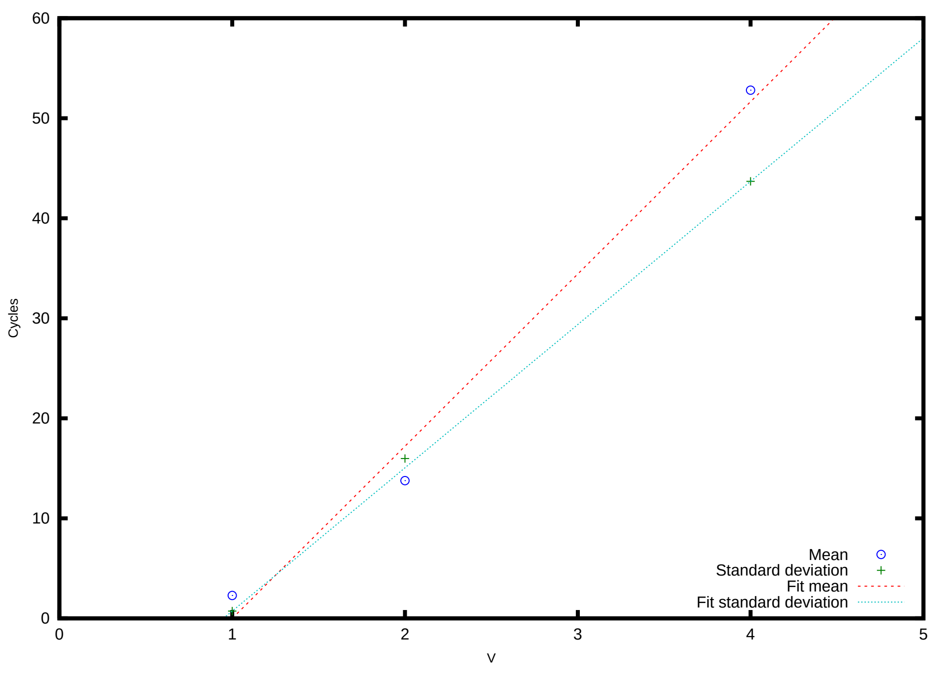


Figure 5.8: Standard deviation and mean for bit shifting on the ATmega

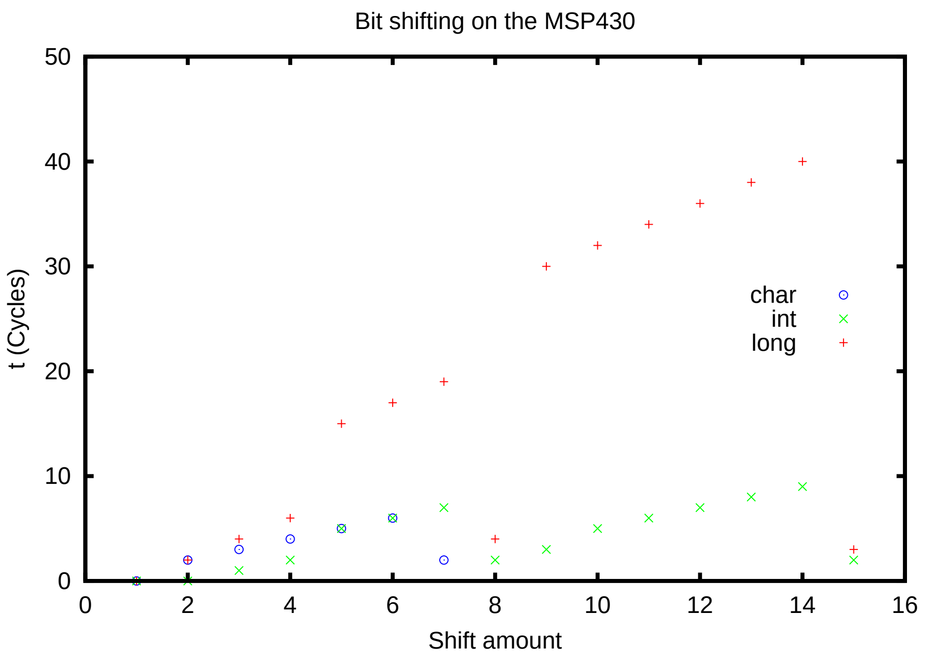


Figure 5.9: Bit shifting on the MSP430 (only left shifting shown)

5.7 Boolean expressions, if statements and loops

The evaluation of a boolean expression consists of comparing values and evaluating boolean expressions. The evaluation stops when the final result is known. So, for determining the costs of a boolean expression, the values of the involved variables have to be known. The costs of the expression depend on the amount of terms that have to be compared and evaluated. Each comparison and boolean evaluation comes with some costs, linear V. The costs of boolean and/or expressions are already given in Section 5.3.

Then there are startup costs involved, these are the fixed costs of evaluating a boolean expression and originate from the branches that always have to be executed, independent of the number of terms.

An if statement is no more than a boolean expression, except that there is some extra code after the evaluation of the expressions and before the final branch instructions. The costs of the if statement itself are zero, because all instructions to make an if statement are already present in the evaluation of the boolean expression, it is just a different arrangement.

The C language features multiple loop constructs: for, while and do...while. The costs of such a loop are a combination of the costs of the boolean expression and the costs to jump back to the beginning of the loop, based on the results of the expression. In case of a for-loop there are usually extra costs for an expression like `i++`. These have to be calculated separately and counted for each loop iteration. These are listed as “Loops per iteration” in the tables and have to be counted each iteration. The costs to jump back to the beginning of the loop are usually in the order of 1 or 2 cycles.

PIC18F

On a PIC18F the comparison is perfectly linear in V: $t = 1, 2, 4$ for $V = 1, 2, 4$. The costs of a loop iteration is 1 cycle. The boolean type is 8 bits, so boolean *and* and *or* cost 1 cycle.

ATmega

Comparison on the ATmega is mostly linear in V: $t = 1, 6, 12$ for $V = 1, 2, 4$, with $t = 3V$ as model there is an error for $t = 1$. This can be a little too high when using a 8 bits char as loop variable, but that is only 2 cycles too much per loop iteration which, is an acceptable error. A loop costs 2 cycles per iteration and booleans are 8-bit, so boolean *and* and *or* cost 1 cycle.

MSP430

Comparisons are linear in V for the MSP430; the model is $t = 3V$. The costs of a loop are 2 cycles per iteration. Booleans are 8-bits, so boolean *and* and *or* costs 1 cycle.

5.8 Function calls and interrupts

The costs of a function call depend mostly on the costs of the context save. If a function uses more local variables than fit in the scratch registers, then some saved-across call registers have to be used, which means that the current value of these registers has to be stored somewhere else (usually on the stack) and restored at the end of a function.

The cost of a context save and restore depend on the number of registers that have to be saved. Usually, when only one or two registers have to be saved, the instructions are inlined. When more registers have to be saved the compiler uses a subroutine that can be used by all functions to limit the program size. The costs of the context save are linear in the number of registers that have to be saved and restored, so the model is linear in V.

In practice, either the number of used variables is so small that no context save has to be done or a whole lot of variables is used so the complete context has to be saved. Therefore we will assume that, if a context save is done, the compiler will always use the subroutine for context saving.

Besides the context save there are also costs of calling the function and, if applicable, returning a value. These costs will have a fixed part for the call and return instructions and a part linear in V for returning a value. In case of a void function, V is zero.

If a function has arguments, there are also costs involved. A limited number of arguments can be passed in registers. If more data needs to be passed it is pushed onto the stack. As mentioned in Section 5.2, the case of too much arguments is not covered in the abstract model.

Interrupts are not much more than a normal function, except that all registers have to be saved and restored and there are some extra implementation dependent costs. More about this can be found in Section A.3.2.

PIC18F

Function calls and returning a value is linear on a PIC18F: $t = 5 + V$. Passing an argument is also linear: $t = 2V$. This is independent of the number of arguments, because all arguments are passed via the data memory. Therefore, the costs of a stack push are listed as N/A in Table 5.2.

The PIC18F only has the working register `W`, which is not saved across calls. Therefore, there is no context saving on a PIC18F, so the costs are counted as zero. An interrupt costs 11 cycles.

ATmega

The ATmega profits again from its `MOVW` instruction when returning a value or passing an argument. There is a `MOVW` instruction and a `MOV` instruction involved; therefore the model has a V and a V_2 parameter. Also, the arguments take advantage of the `MOVW` instruction, the costs of function arguments are V_2. The maximum for the arguments is 8 bytes, so if V is larger than 8 the arguments have to be pushed on the stack, which costs $2V$.

The context save and restore on an ATmega costs 29+4 cycles per register, $t = 29 + 4V$ for a maximum of 16 registers. An interrupt costs 39 cycles.

MSP430

The MSP430 shows the same figures for both 8-bits chars and 16-bits ints because of its 16 bits architecture. Both the function call and return costs and the function argument costs are perfectly linear. If more than 4 words are supplied the arguments have to be pushed on the stack which costs $3V$.

On the MSP430 the context save costs 15 cycles + 2 cycles per register, so $t = 15 + 2V$ with a maximum of 8 registers. An interrupt costs 16 cycles.

	Void/ independent 0 bits	**Char 8 bits**	**Int 16 bits**	**Long 32 bits**	**Model**
Assignment		2	6	12	$3V$
Addition, subtraction, (boolean) and, (boolean) or, xor		1	4	8	$2V$
Using a constant		0	0	0	0
Using a global variable		2	2	2	2
Multiplication		2	13	40-740	$3.881V^2 - 2.1667$
Pointer dereference		14	18	21	$12.5 + 2.2V$
Bit shifting		1-45	1-144	9-183	$(V+6)(p+1)$
Startup costs boolean expression	3				3
Comparison		1	2	4	V
Loops per iteration	1				1
Function call and return	5	6	7	9	$5 + V$
Function argument		2	4	8	$2V$
Context save	0				0
Stack push	N/A				n/a
Interrupt	11				11

Table 5.2: Cycle times for all operations on the PIC18F

5.9 ARM7

In the previous sections models are determined based on the numbers of the previous chapter, summarized in Table 4.1. This can also be done directly, skipping the step of writing down the separate values for diverse datasizes. Most relations between the operation size and the cycle count are straightforward and easily derived. This was done for the ARM7 CPU core that is used in the NXP LPC2144. The results are in Table 5.5. The ARM7 has a

	Void/ independent 0 bits	**Char 8 bits**	**Int 16 bits**	**Long 32 bits**	**Model**
Assignment		2	2	4	$2V_2$
Addition, subtraction, (boolean) and, (boolean) or, xor		1	2	4	V
Using a constant		1	2	4	V
Using a global variable		2	4	8	$2V$
Multiplication		1	20	53	$2.333V^2 + V + 3V_2 + 5.667$
Pointer dereference		1	1	1	1
Bit shifting		1-3	2-48	2-124	$\mu = 17.2V - 17.2$, $\sigma = 14.3V - 13.6$
Startup costs boolean expression	2				2
Comparison		1	6	12	$3V$
Loops per iteration	2				2
Function call and return	7	9	10	13	$7 + V + V_2$
Function argument		1	1	2	V_2
Context save	$29 + 4V$				$29 + 4V$
Stack push	$2V$				$2V$
Interrupt	39				39

Table 5.3: Cycle times for all operations on the ATmega

32 bits CPU and the largest datasize considered here is 32 bits, so the costs of using larger datatypes is not known. The use of the V parameters in the formulas is based on the analysis of the assembly code and the knowledge acquired from the previous sections. For example the function call and return operation: 2 cycles are used for the call and the return back and 1 specifically for the passing of the resulting value. So when calling a void function the costs are 2 cycles, and when calling an 8, 16 or 32 bits argument function the costs are 3 cycles.

An assignment is a `MOV` instruction: one instruction per B, so the costs are $t = V$. Addition, subtraction, (boolean) *and* and (boolean) *or* is done using respectively the `ADD`, `SUB`, `AND` and `OR` instructions, one instruction per B, therefore the cost model becomes $t = V$. Constants are generated via the 8-bit constant argument of an instruction when constants are smaller than 8-bits. For constants up to 16 bits this is complemented with an OR instruction with a constant argument and a shift. For larger instructions the LDR instructions

	Void/ independent 0 bits	**Char 8 bits**	**Int 16 bits**	**Long 32 bits**	**Model**
Assignment		3	3	6	$3V$
Addition, subtraction, (boolean) and, (boolean) or, xor		1	1	2	V
Using a constant		1	1	2	V
Using a global variable		2	2	4	$2V$
Multiplication		29	27	45	$5.667V^2 + 22.33$
Pointer dereference		0	0	0	0
Bit shifting		0-6	0-9	0-46	$\mu = 15.4V - 12.2$, $\sigma = 12.6V - 10.4$
Startup costs boolean expression	0				0
Comparison		3	3	6	$3V$
Loops per iteration	2				2
Function call and return	7	9	9	11	$7 + 2V$
Function argument		1	1	2	V
Context save	$15 + 2V$				$15 + 2V$
Stack push	$3V$				$3V$
Interrupt	16				16

Table 5.4: Cycle times for all operations on the MSP430

is used to load the constant from a location in the memory. In the case of an 8-bit constant the costs are 1 cycle, in case of 16 to 32 bits the costs are 2 cycles. The advantage of 1 cycle for 8-bit constants will be neglected and the costs will be modeled as $t = 2V$.

Using global variables costs 4 cycles for loading and storing; two for the `LDR` instruction to load the address and two for the `LDR` or `STR` instruction to load from or store to that address. Once the address is loaded, if more loads are needed, for example because the datatype is larger than 32 bits, only one `LDR` or `STR` instruction is needed because the old, already loaded, address can be used with an offset. The model becomes $t = 2 + 2V$, because two cycles are the fixed costs to load the address and then it is 2 cycles per B.

Multiplication is 1 cycle for all datatypes up to 32 bits. For larger datatypes the multiplication operation scales quadratic, therefore the model for multiplication is $t = V^2$.

Pointer dereferencing is done by using the `LDR` or `STR` instructions, in the same way as loading and storing global variables, with the difference that the address is already available in a variable and does not have to be loaded. This saves two cycles with respect to the costs of a global variable. The `LDR` or `STR` instructions have to be used every *B* bits, therefore the cost model becomes $t = 2V$.

Bit shifting is fast with an ARM7. The ARM7 has a barrel shifter that can shift every wanted number of positions in a single cycle, instead of shifting step-by-step. When the core is running in ARM mode some instructions have an extra argument for bit shifting, when in Thumb mode the bit shift instruction is a separate instruction, taking one cycle. Because Thumb mode is the default for this compiler the costs of the shift operation are modeled as $t = V$.

For an if statement, the startup costs are zero. Per comparison the costs are 2V cycles. A boolean *and* or boolean *or* costs *V* cycles, as earlier mentioned. A loop costs 1 cycle per loop iteration, a loop does not have a datatype, therefore the costs model is $t = 1$.

A function call costs $t = 2 + V$, as mentioned before; 2 cycles for the call and return and one per *B* bits of the return type. Passing arguments costs 1 cycle per *B* bits, therefore the model is $t = V$. Saving and restoring the context can be done in one instruction for all registers, 1+1=2, so the model for context save is $t = 2$. When arguments do not fit in the available registers the costs of stacking an argument are 1 cycle.

The costs of the interrupt are determined using a piece of example code that comes with the IAR workbench software. The costs are 24 cycles for the call, complete context save and return.

This method of characterizing the platform is much faster than the method presented in the previous chapter. The step of writing all numbers down, including finding out all exceptions and describing these, costs much time. This method is preferred above the method of the previous chapter.

5.10 Spartan 3 FPGA

The Spartan 3 FPGA cannot run C code. But what we can do is take the same operations and map these to FPGA hardware. So when a line in the C version of the algorithm looks like: `a = b + c * d` we can say that in an FPGA you need a multiplier to calculate `c * d` and an adder to calculate the addition and a memory element to store `a`. This will result in an FPGA design that has all operations implemented in parallel. This does not mean that the algorithm itself is parallellized. Because time costs are no issue in this methond of implementation, instead of time costs for a microcontroller implementation we will count area costs for the FPGA.

Eventually the designer could paralellize the algorithm as much as possible to gain more speed or the designer could implement a serialization mechanism that coordinates a sequential algorithm. This is completely up to the designer and will not be covered by this thesis. In this thesis only the costs of implementing all operations together in an FPGA is considered. The area costs of the coordination mechanisms will not be considered either.

5.10.1 Basic arithmetic

To calculate the hardware costs of an operation the operation was written as a VHDL entity and synthesized. Then the costs of each entity in terms of LUTs, flipflops and slices were

Operation	**Costs**
Assignment	V
Addition, subtraction, (boolean) and, (boolean) or, xor	V
Using a constant	$2V$
Using a global variable	$2+2V$
Multiplication	V^2
Pointer dereference	$2V$
Bit shifting	V
Startup costs boolean expression	0
Comparison	$2V$[a]
Loops per iteration	1
Function call and return	$2+V$
Function argument	V
Context save	2
Stack push	1
Interrupt	24

Table 5.5: Cycle times for all operations on the ARM7/LPC2144

[a]+1 per operand if $B \neq N$, for example when comparing booleans.

put in Table 5.6. These are the results when the synthesizer makes the choices related to optimizing for speed or area.

The results are more or less logical: Addition, subtraction, boolean operations and comparisons the costs are linear in the number of bits: 1 LUT and 0.5-1 slice per bit. For multiplication the available 18×18 hardware multipliers are used. If the datasize is larger than 18 bits, multiple multipliers are used and some glue-logic to connect them and add the results of the partial multiplications. The costs are quadratic in V, both for the multipliers and for the glue logic. For the multipliers the quadratic form of the model is straightforward, for every 18 bits quadratically more multipliers are needed. The LUT costs are less smooth but are clearly quadratic, as shown in Figure 5.10.

Addition and subtraction are linear in N, for each bit a fixed amount of LUTs is needed, more bits, more LUTs. The same goes for slices; each slice contains 2 LUTs, which are used dependent on the choices of the synthesizer. In some cases, the synthesizer can use both LUTs of a slice, in other cases only one. In the case of addition and subtraction the amount of LUTs needed is N and the amount of slices needed is $N/2$, so the synthesizer can use both LUTs of a slice. ‘Less than’ and ‘greater than’ operations are in principle the same as subtraction, but the result is not used; only the sign of the result is of interest. Therefore, the costs of these operations is the same as the costs of subtraction.

Or, *and* and *xor* are also linear in N but here only one of the two available LUTs per slice are used, so the slice costs are also N.

To store a number, flipflops or a location in a block-RAM is needed. In the Spartan 3 each slice contains two storage elements that can be used as flipflop or latch. Besides that,

	LUTs	Flipflops	Slices	Mults
Multiplication $8 \times 8 = 16$				1
Multiplication $16 \times 16 = 32$				1
Multiplication $24 \times 24 = 48$	55		28	4
Multiplication $32 \times 32 = 64$	79		40	4
Multiplication $48 \times 48 = 96$	193		135	9
Multiplication $64 \times 64 = 128$	509		256	16
8-bit addition	8		4	
8-bit subtraction	8		4	
8-bit bitwise or	8		8	
8-bit bitwise and	8		8	
8-bit bitwise xor	8		8	
32-bit bitwise or	32		32	
32-bit addition	32		16	
48-bit less than/greater than	48		24	
8-bit equality check	4		2	
48-bit equality check	24		12	
Storing an 8-bit number		8[a]		
Storing a 32-bit number		32[a]		

Table 5.6: Costs of basic arithmetic on the Spartan 3

[a]Can be IOB-flipflops or slice flipflops, dependent on the mapping

each IO port has a flipflop, so dependent on the location of the stored number one of these options is chosen by the synthesizer. Then there are block RAMs, which have an address- and databus, but because there is some coordination needed for the storage the block RAMs are much less suitable for storing a temporary value, therefore only flipflops are considered.

The slice cost of storing a number are chosen to be zero because in practice the result of some calculation is stored. The calculation already uses the slices for its LUTs and the storage of the number is done in the flipflops of these slices, so no extra slices are needed.

The resulting models are listed in Table 5.7.

5.10.2 Program flow

In the previous subsection we mapped arithmetical operations to hardware. Program flow constructions have no hardware equivalents. Although a for-loop could be modeled as a counter and some other logic, this counter is not able to implement the function of that for-loop in the algorithm. The function of a for-loop is usually not to do the same thing for a certain amount of time, but more to iterate over a dataset and do things with this stream of data. In an FPGA situation, either this for-loop is parallellized or a data-stream processing entity is made that processes the incoming data to generate the output.

Some program-flow related things can be modeled using a finite-state-machine. The size of such finite state machine will be negligible compared to the size of large adders and

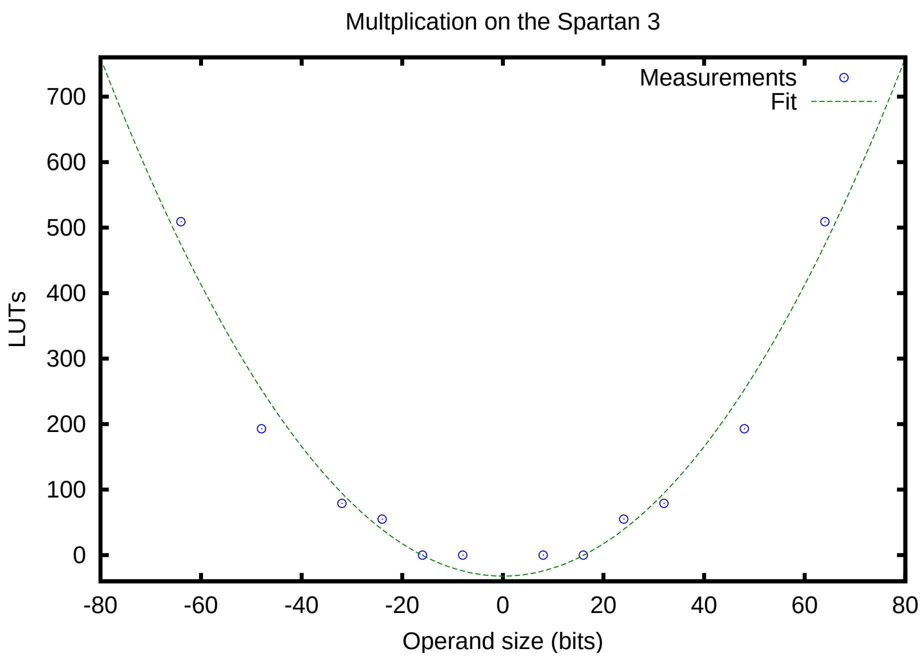


Figure 5.10: LUT costs of multiplication on the Spartan 3

	LUTs	**Flipflops**	**Slices**	**Mults**
Multiplication	$0.12359N^2 - 32.040$	0	$\frac{1}{2}(0.12359N^2 - 32.040)$	$\lceil \frac{N}{18} \rceil^2$
Addition, subtraction, less than, greater than	N	0	N/2	0
Equality check	N/2	0	N/4	0
Or, and, xor	N	0	N	0
Storing a number	0	N	0	0

Table 5.7: Models of hardware cost of basic arithmetic on the Spartan 3

multipliers.

We may conclude that the implementation of program flow related constructs could be done in so much different ways that it is impossible to determine the costs of these constructs in advance. Because of this reason the costs cannot be determined for program flow and other constructs.

5.11 Power consumption

To get the most accurate results for power consumption one has to measure on a real target for each kind of operation what the power consumption is. Real targets are not available and in this research we want to determine the power consumption analytically. Therefore, we need to use the data given by the manufacturers in their datasheets. These datasheets usually give the typical supply current drawn by the microcontroller chip for a given temperature, clock speed and supply voltage.

To get the lowest power consumption all these parameters have to be optimized; the clock frequency is chosen such that the algorithm executes just fast enough to get the wanted performance. Then the supply voltage is chosen as low as possible for the given clock speed.

The influence of the temperature on the current consumption is minimal and it cannot be controlled in practical situations. 25 °C will be used as standard temperature as all datasheets have figures for this temperature. We will assume that the minimum required clock frequency is known or determined based on analysis of the algorithm and the target, using either the detailed or the abstract model.

Power consumption caused by peripherals and current drawn from I/O pins will not be considered. The power consumption of peripherals depends on the device. However, their power consumption is not always specified in the devices datasheet. This is a problem, but with the given information it cannot be solved in this thesis. Power drawn from I/O pins will be the same for all hardware targets, so it is no basis for comparison and will not be considered.

5.11.1 Relation between frequency and power consumption

The average power consumption, P_{avg}, of a CMOS circuit can be described by:

$$P_{avg} = P_{dynamic} + P_{short-circuit} + P_{leakage} + P_{static} \tag{5.3}$$

Where $P_{dynamic}$ is the dynamic power consumption that is caused by the charging and discharging of the capacities in the transistors and wires. $P_{dynamic}$ is given by:

$$P_{dynamic} = C_L \cdot V_{DD}^2 \cdot N \cdot F \tag{5.4}$$

In this formula C_L is the capacitance of the circuit, which can be as less as a wire or an inverter or this can be the sum of all capacitances in the whole circuit when calculating the power consumption of the whole circuit. V_{DD} is the supply voltage of the circuit, N is the average number of transitions per clock cycle and F is the clock frequency.

A short-circuit happens when the state of a CMOS gate changes. During the transition, there is a moment that both the pMOS and nMOS transistors are on. At that moment, a short circuit exists between the power supply and the ground. The power consumed at that moment, $P_{short-circuit}$ is given by:

$$P_{short-circuit} = K \cdot (V_{DD} - 2V_{th})^3 \cdot \tau \cdot N \cdot F \tag{5.5}$$

Where K is a constant that depends on the technology and the transistor sizes, V_{th} is the threshold voltage of the transistors, which is technology dependent and τ is the rise or fall time of the input signal.

Then there is the power consumption due to leakage current and static power consumption. The leakage current is something static and therefore the power consumed because of this phenomena, $P_{leakage}$ is given by:

$$P_{leakage} = I_{leakage} \cdot V_{DD} \tag{5.6}$$

$I_{leakage}$ may, according to [17], be assumed equal to the reverse saturation current, which depends on material and transistor properties.

The static power consumption, P_{static} is caused by feeding degraded voltage levels (for example caused by pass gates) to the input of gates, causing transistors to be weakly on and consuming significantly more power. Another cause is the use of pseudo-nMOS logic: a gate uses a single pMOS transistor, whose gate is always grounded, acts as a pull-up resistor. Then a complex function implemented by nMOS transistors pulls the output low when the result of the function is low. In this way, there exists a direct path from V_{DD} to GND and thus continuous power consumption.

This current consumption can be modelled by the following formula:

$$P_{static} = I_{static} \cdot V_{DD} \tag{5.7}$$

where I_{static} is the current that flows because of the abovementioned reasons. In case of pseudo-nMOS logic, a probability parameter can be added to this model, or incorporated in the static current parameter.

Summarizing the total power consumption in the scope of this research, the average power consumption becomes:

$$P_{avg} = C_L \cdot V_{DD}^2 \cdot N \cdot F + K \cdot (V_{DD} - 2V_{th})^3 \cdot \tau \cdot N \cdot F + I_{leakage} \cdot V_{DD} + I_{static} \cdot V_{DD} \tag{5.8}$$

In this model the circuit-dependent parameters are: C_L, K, V_{th}, τ, $I_{leakage}$ and I_{static}. These parameters are unknown; the datasheet only gives a relation between I, V_{DD} and F.

When the power model is reduced to:

$$P_{avg} = a \cdot V_{DD}^3 \cdot F + b \cdot V_{DD}^2 \cdot F + c \cdot V_{DD} \cdot F + d \cdot V_{DD} + e \cdot F + f \tag{5.9}$$

the values of $a..f$ can be determined by fitting the information from the datasheets. This would still be a complicated task. To simplify this we define that we are only interested in the power consumption for lowest possible voltage for a specific frequency. When assuming that lowest possible voltage is linearly related to the frequency, as is stated in the PIC18F8766 datasheet (see also Section 5.11.2), the model reduces to:

$$P_{avg} = a \cdot F^4 + b \cdot F^3 + c \cdot F^2 + d \cdot F + e \tag{5.10}$$

This is the model that will be used in this section. The values of $a..e$ will be determined by fitting the data extracted from the datasheets. To help the fitter, the data points will be mirrored in the y-axis, as it has been done for multiplication in Section 5.4.

5.11.2 PIC18F

The PIC18F8722 has two variants, one named PIC18F8722 and one named PIC18**L**F8722. There is no difference between the architectures, power consumption characteristics or maximum clock speed, the only difference is that the PIC18F is the normal variant that requires

4.2 to 5.5 V and the PIC18LF is the low-voltage variant that can run on 2.0 to 5.5 V. This does not mean that the maximum clock frequency can be reached at 2.0 V, the maximum frequency at 2.0 V is 4 MHz and increases linearly with the supply voltage. The datasheet gives a formula for the maximum clock frequency at a certain voltage [11]:

$$F_{MAX} = (16.36\,\text{MHz/V})(V_{DDAPPMIN} - 2.0\,\text{V}) + 4\,\text{MHz} \tag{5.11}$$

or, expressed in terms of F_{MAX}:

$$V_{DDAPPMIN} = \frac{F_{MAX} - 4\,\text{MHz}}{16.36\,\text{MHz/V}} + 2\,\text{V} \tag{5.12}$$

For calculating the power consumption for a certain clock frequency this minimal voltage will be used. The datasheet contains diverse figures for the supply current for diverse supply voltages, clock frequencies, clock sources and running modes. For the supply voltage, the lowest possible option will be chosen. The clock source will be an external crystal oscillator or the internal RC oscillator. Using external clock sources is possible and the microcontroller will consume less power, but when taking the external clock generator's power cost into account, the resulting power consumption will be higher. The running mode considered here is called PRI_RUN, where the microcontroller is actively running.

The data in the datasheet is not fully complete; the current figures for PRI_RUN at 16 MHz are available for a supply voltage of 4.2 and 5.0 V while according to the previously mentioned formula for $V_{DDAPPMIN}$ the PIC18F can run on a voltage as low as 2.7 V. Therefore, the value of the supply current for the PIC18F8722 at 16 MHz was approximated. Figure 5.11 shows a graph containing the values that are given in the datasheet and the approximation for 16 MHz. Table 5.8 lists the current, voltage and power values for 4, 16 and 40 MHz. When a fourth order fit is applied to these figures the resulting model is $P(F) = 0.0000028653F^4 + 0.047137F^2 + 1.2451$. In this formula F is the clock frequency and not the number of cycles per second, which is a factor 4 lower because of the clock division of the PIC18F.

F has to be greater than or equal to 4 MHz because 4 MHz is the maximum clock frequency at the minimum supply voltage. A plot of this model together with the data points is given in Figure 5.12.

5.11.3 ATmega

The ATmega128 also has a low power variant, the ATmega128L. This is the same device except for the specified supply voltage and clock frequency. The normal ATmega requires a supply voltage between 4.5 and 5.5 V and can run at up to 16 MHz. The -L variant can run on a supply voltage between 2.7 and 5.5 V but its maximum clock speed is 8 MHz. Figure 5.13 (taken from [4]) gives the relation between the supply current and the frequency. Although some parts of the data in the graph are outside the given boundaries and the datasheet states that "Parts are not guaranteed to function properly at frequencies higher than the ordering code indicates" it is highly probable that the chip will work. Therefore, some safety margin was taken and Table 5.9 was derived from this graph.

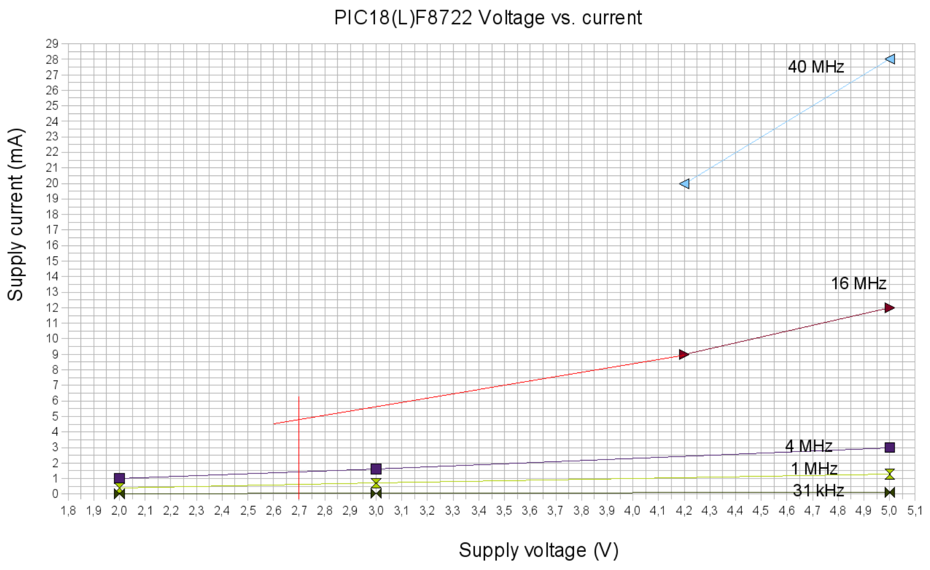


Figure 5.11: PIC18F8722 Suply voltage vs. supply current for diverse clock frequencies

Frequency (MHz)	Voltage (V)	Current (mA)	Power (mW)
4	2.0	1.0	2.0
16	2.7	5.0[a]	13.5
40	4.2	20	84.0

Table 5.8: PIC18F8722 system clock frequency and power consumption

[a]Approximation, see text and Figure 5.11

Making a fit resulted in a negative coefficient for the F^4 term. This means that for higher frequencies the powerconsumption would become negative, which is incorrect, so a datapoint P(0)=0 was added to help the fitter. The resulting model for all $F \geq 8$ MHz is $P(F) = 0.00011162F^4 + 0.48180F^2 - 3.7300$. A graph of the datapoints and the model is given in Figure 5.14.

5.11.4 MSP430

The MSP430F2619 has no special low power variant. The supply voltage has to be between 1.8 and 3.6 V during program execution and at least 2.2 V during flash memory programming. Figure 5.15 gives the minimal supply voltage for a certain clock speed. Figure 5.16 gives the relation between the supply voltage and the supply current for certain clock frequencies. At least 3.3 V is required to be able to run at the maximum clock frequency, 16 MHz.

From these graphs, Table 5.10 has been made. Note that Figure 5.15 specifies that the minimum voltage required to run at 12 MHz is 2.7 V but that Figure 5.16 does not specify

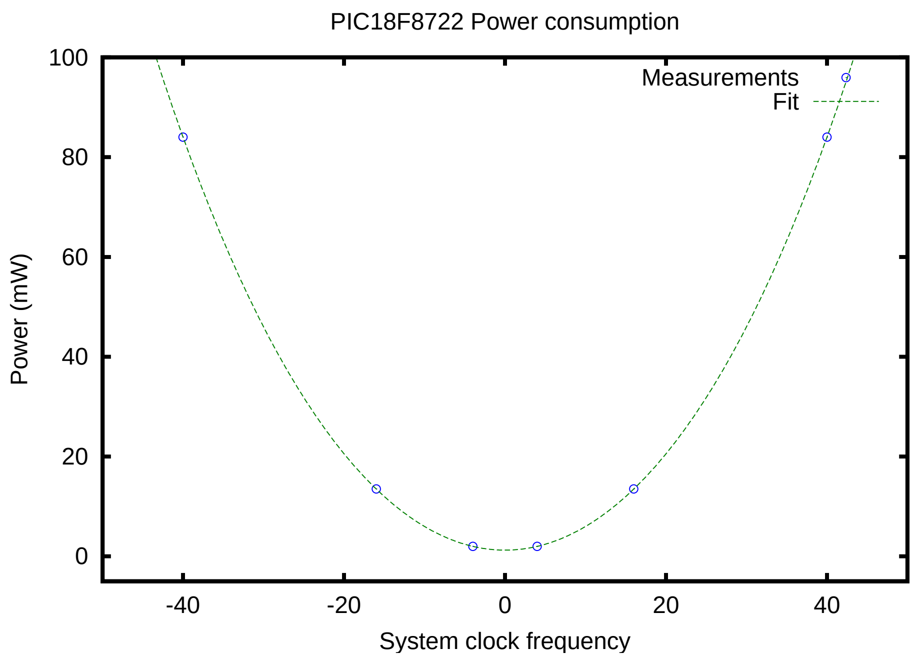


Figure 5.12: PIC18F8722 Power consumption vs. clock frequency

Frequency (MHz)	Voltage (V)	Current (mA)	Power (mW)
8	2,7	7	18,9
8,5	3	9	27
9,2	3,3	11	36,3
10	3,6	13	46,8
12	4	18	72
16	4,5	28	126

Table 5.9: ATmega128(L) system clock frequency and power consumption

a current for these values. Therefore the value the value for 12 MHz is an approximation based on the trends of the graph for 12 MHz. When graphing this and fitting the resulting model for the power consumption is: $P(F) = 0.000165F^4 + 0.0720F^2 + 1.07$ where P is in mW, F in MHz and $F \geq 4.15$ MHz. See Figure 5.17 for the graph.

5.11.5 LPC2144

The LPC2144 requires a supply voltage of 3.3 V with a margin of ± 0.3 V. Current consumption figures are given for 10 and 60 MHz while executing an endless loop and at 12 and 60 MHz with USB enabled. There are no graphs that show information for other clock frequencies. Because the supply voltage is fixed, the fourth order model from Equation 5.10 cannot be used because it relies on the assumption that the supply voltage is linear with the

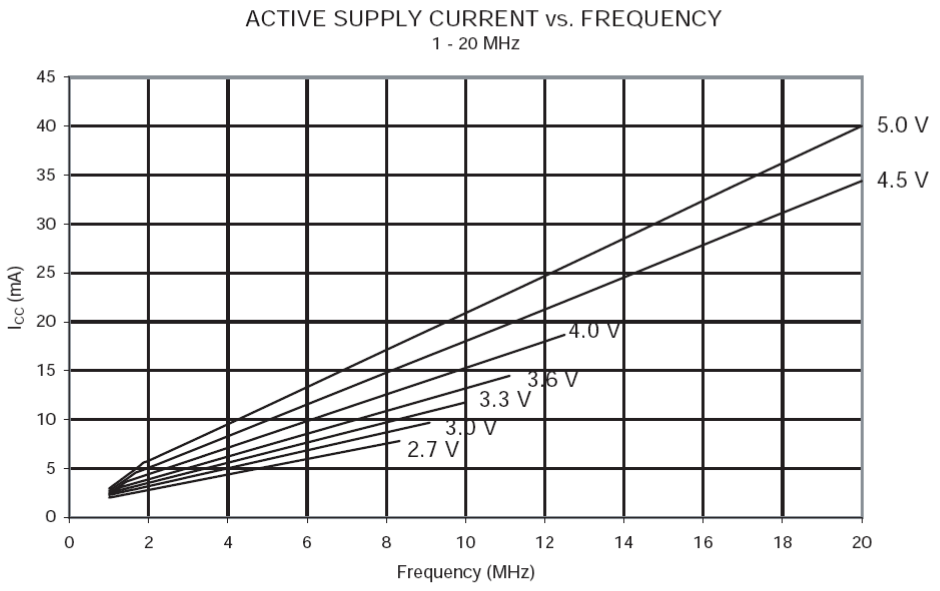


Figure 5.13: ATmega128(L) supply current vs. clock frequency

clock frequency. In this case, the model given in Equation 5.9 has to be used.

Because V_{DD} is fixed, the model for the power consumption of the LPC2144 reduces to a function linear in F. With a current consumption of 15 mA at 10 MHz at 3.3 V and a current consumption of 40 mA at 60 MHz at 3.3 V the power consumption is 49.5 mW at 10 MHz and 132 mW at 60 MHz. Based on these figures the model becomes $P(F) = 1.65 \cdot F + 33$.

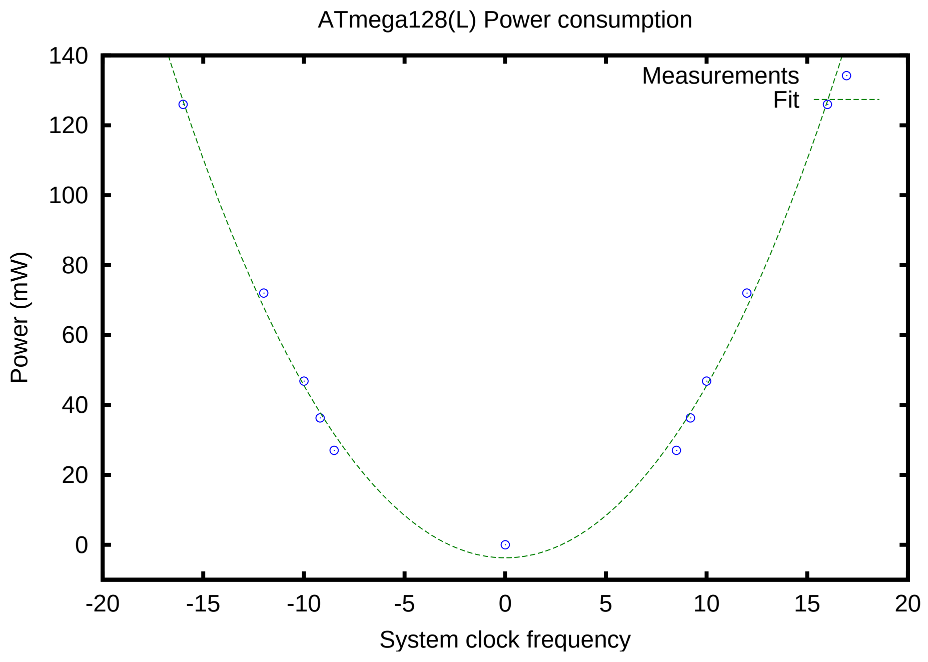


Figure 5.14: ATmega128(L) power consumption vs. clock frequency

Frequency (MHz)	**Voltage (V)**	**Current (mA)**	**Power (mW)**
4.15	1.8	1.3	2.34
8	2.2	2.9	6.38
12	2.7[a]	5.5	14.85
16	3.3	9.2	30.36

Table 5.10: MSP430 system clock frequency and power consumption

[a]Approximation, see text.

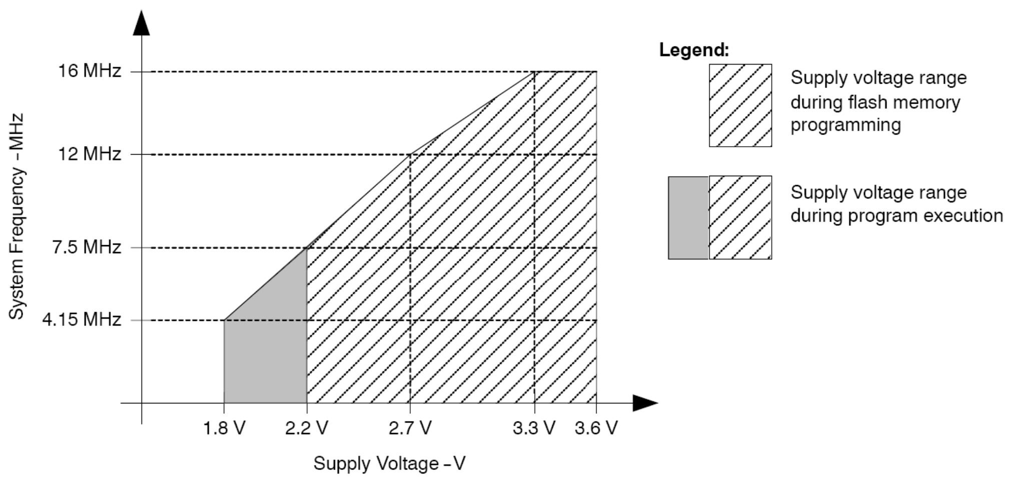


Figure 5.15: MSP430F2619 System clock frequency vs. supply voltage operating range. Source: [18]

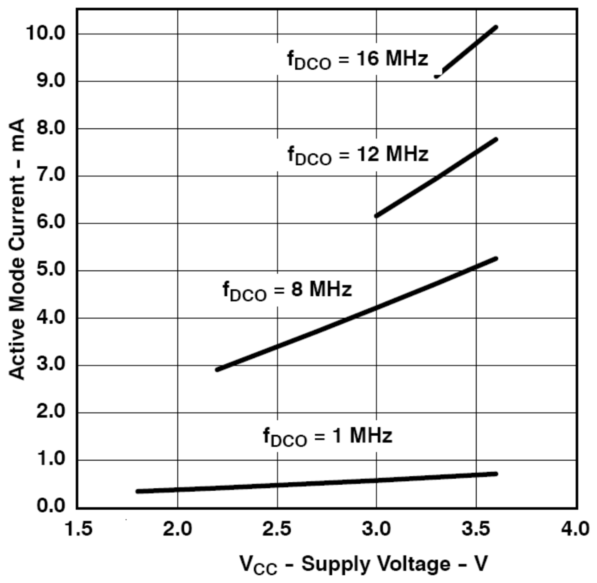


Figure 5.16: MSP430F2619 Active mode current vs. supply voltage, $T_A = 25°\text{C}$. Source: [18]

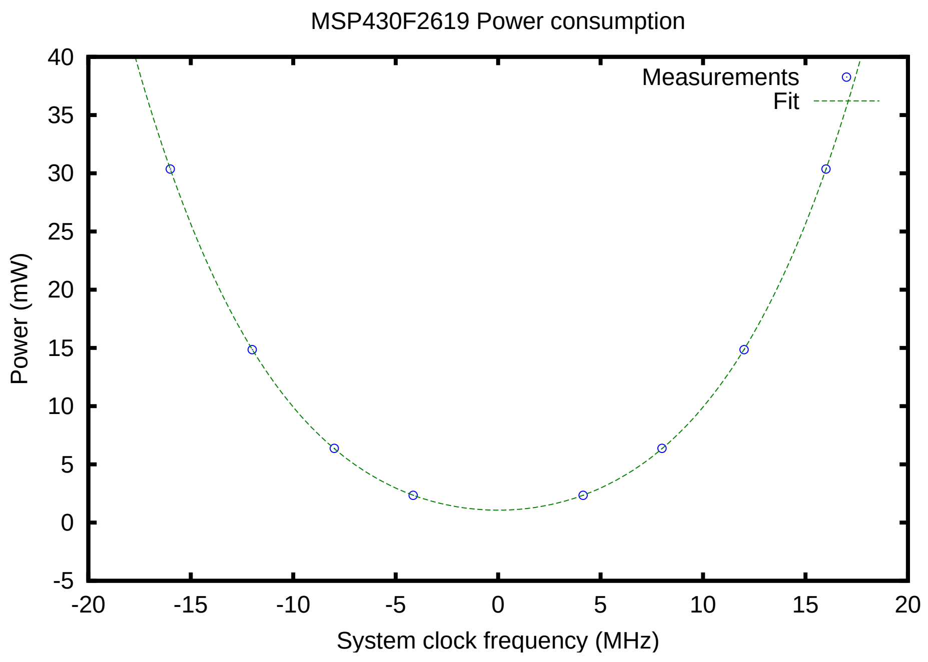


Figure 5.17: MSP430F2619 Power consumption vs. clock frequency

5.11.6 Spartan 3

The XC3S400 from Xilinx's Spartan 3 series has three different supply lines: V_{CCO}, V_{CCINT} and V_{CCAUX}. V_{CCO} supplies the power for the I/O lines. Power consumed by I/O devices is delivered by this power line. V_{CCINT} is the main power supply for the FPGA's logic. Logic constructed in the FPGA, like an adder, consumes power delivered by this power line. The last power line is V_{CCAUX}, which is an auxiliary source of power that is primarily used to optimize various FPGA functions like I/O switching. V_{CCINT} is 1.2 V, V_{CCAUX} is 2.5 V and V_{CCO} can be between 1.2 V and 3.3 V, dependent on the I/O standard that is used for communication with external devices.

Power used for the operation of the FPGA, the execution of the algorithm and the calculations will be consumed from V_{CCINT} and V_{CCAUX}. The quiescent power consumption of the XC3S400 is 18 mW on the V_{CCINT} line and 38 mW on the V_{CCAUX} line. To calculate the dynamic power consumption Xilinx has three possibilities[2]:

1. An online web form: `http://www.origin.xilinx.com/cgi-bin/power_tool/power_Spartan3` this form allows the user to fill in the chosen supply voltages, the sizes of the synthesized logic in terms of LUTs/flipflops/block RAMs/multipliers, the toggle rates of the signals and the clock frequency. With these values, the web form calculates the power consumption of the FPGA.

2. A spreadsheet for use with Microsoft Excel. This spreadsheet has the same functionality as the web form.

3. The XPower Analyser utility, part of the ISE software, the development environment. After creating the design the user feeds the XPower utility some simulation data and the XPower utility calculates the power that would be consumed during the simulation.

The web form and spreadsheet are meant for pre-implementation estimation, the XPower tool isused during and after the design. The web form does not give any information about the relation between the filled-in values; all calculations are done server-side. The spreadsheet is protected with a password and thus does not give any information either. The XPower tool can only be used with a working design and a simulation, so it cannot be used before a design is made.

5.12 Summary

After being able to determine the execution time of an algorithm in the previous chapter using detailed, complex and time intensive modeling, this chapter simplifies and reduces the model to 15 specific operations. These models are expressed in terms of V and V_2, which indicate the ratio between the size of the architecture (B) and the operation (N). Special cases and operations that happen very infrequently are neglected.

[2]More information about these possibilities is available from the Xilinx power solutions webpage: `http://www.xilinx.com/products/design_resources/power_central/index.htm`

The 15 operations are:

1. Assignment: linear in V. Microcontrollers like the ATmega, with a move-instruction that moves multiple registers, have an advantage when datatypes larger than B.

2. Addition, subtraction, (boolean) and, (boolean) or and xor: linear in V. There are no special cases when B is larger than N for addition and subtraction because all microcontrollers have an add-with-carry or subtract-with-borrow instruction for larger datatypes.

3. Using a constant: no costs or linear in V. These costs come from the limitation to use constants directly in an instruction. Constants have to be generated before they can be used using a special instruction or have to be loaded from the memory.

4. Using a global variable: in most cases linear in V. Global variables are not held in the registers but are stored in a fixed memory location. Most microcontrollers need a special instruction to load the memory variable in a register so that it can be used in a calculation. These instructions have to be executed for every B bits of the memory variable so this operation is linear in V. In case of the PIC18F, the global variables are stored in a different memory bank, causing bank switching before and after using a global variable. These costs are not related to V.

5. Multiplication: quadratic in V. Just as when multiplying two multi-digit numbers by hand on paper, multiplication on microcontrollers for numbers larger than B requires quadratic more multiplications and additions.

6. Pointer dereference: fixed or linear in V. Pointer dereferencing requires the use of a load or store instruction that can use the contents of a register as basis for the memory location to load from or store to. In some architectures these load or store instructions do not involve extra costs, in some once the address is loaded there are no further extra costs because the architecture can load or store with an offset and in others there are costs for every B bits.

7. Bit shifting: linear in both V and the number of positions to be shifted. Most simple architectures only have a shift-by-1 instruction to shift a B bits one position. This has to be repeated V times to shift a whole variable 1 postion. To shift p positions this has to be repeated p times.

8. Startup costs of a boolean expression/if statement: fixed. Evaluating boolean expressions involves jump instructions that have to be taken, regardless of the result of the expression. These are not related to the expression and are therefore fixed.

9. Comparison: linear in V. For comparing if two variables are equal they are subtracted from each other, the microcontroller sets or resets the zero flag in the status register and a jump-if-zero instruction is executed based on this flag. For other types of comparison like less than (or equal) or greater than (or equal) other flags are used but the procedure is the same. A subtraction is linear in V, so the comparison too.

10. Loops per iteration: fixed. These costs represent the jumps that are executed to get from the end of the loop back to the beginning. This has to be counted for every iteration of the loop, does not depend on the type of loop (for, do..while, while..do) and does not include the cost of the evaluation of the loop condition nor the costs of an increment of the loop variable of the for loop.

11. Function call and return: linear in V. The costs of the function call are always the same, regardless of the type of the function (void, int, short, etc. . .) but returning the result of the function involves some costs for passing the return value. These costs are linear in V. For a void function N is zero, which makes V zero too.

12. Function argument: linear in V. Each argument of the function needs to be moved to the right register or pushed onto the stack. This is linear in V. There is a limitation on the microcontroller's available registers for function arguments so if a function uses more they have to be pushed on the stack, which is usually more expensive. This operation only covers the costs of the non-stacked variables.

13. Context save (and restore): linear in V (where V is about the registers that have to be saved). Each register has to be stacked using a push-to-stack instruction, so the costs are linear in V. Some microcontrollers, like the MSP430 and the ARM7, can do this with a single instruction, thus saving program memory. On the ARM7 this really costs 1 cycle, on the MSP430 the costs are linear in V.

14. Stack push: linear in V. Every B bits need a stack instruction. As mentioned above: some microcontrollers can do this with one instruction, some even with costs independent on V.

15. Interrupt: fixed. An interrupt has nothing related to V because unlike a normal function it has to save all registers so that the interrupted code wouldn't notice being interrupted.

The resulting models for the PIC18F, the ATmega, the MSP430 and the ARM7 are in Table 5.11.

For FPGA's, in this case the Spartan 3, only mathematical operations can be analyzed for area consumption. Program flow related operations could be implemented in so much different ways that it is not possible to determine the size of these things.

To analyze the costs in terms of power of an algorithm first the costs in terms of time is analyzed. When knowing the costs of an algorithm in cycles and the required performance (the number of times the algorithm has to be executed in a certain timespan), the minimum clock speed of the microcontroller can be calculated. This minimum clock speed dictates the minimum supply voltage. The combination of the supply voltage and the clock speed results in a certain power consumption. Datasheets of microcontrollers usually give the relation between supply voltage, supply current and clock frequency using a graph or table.

Power consumption in CMOS circuits depends in third order on the supply voltage and linear on the clock frequency. When the supply voltage is fixed (LPC2144), the power consumption thus depends linearly on the clock frequency. When the supply voltage is

	PIC18F	**ATmega**	**MSP430**	**ARM7**
Assignment	$3V$	$2V_2$	$3V$	V
Addition, subtraction, (boolean) and, (boolean) or, xor	$2V$	V	V	V
Using a constant	0	V	V	$2V$
Using a global variable	2	$2V$	$2V$	$2+2V$
Multiplication	$3.881V^2 - 2.1667$	$2.333V^2 + V + 3V_2 + 5.667$	$5.667V^2 + 22.33$	V^2
Pointer dereference	$12.5+2.2V$	1	0	$2V$
Bit shifting	$(V+6)(p+1)$	$\mu = 17.2V - 17.2$, $\sigma = 14.3V - 13.6$	$\mu = 15.4V - 12.2$, $\sigma = 12.6V - 10.4$	V
Startup costs boolean expression	3	2	0	0
Comparison	V	$3V$	$3V$	$2V$
Loops per iteration	1	2	2	1
Function call and return	$5+V$	$7+V+V_2$	$7+2V$	$2+V$
Function argument	$2V$	V_2	V	V
Context save	0	$29+4V$	$15+2V$	2
Stack push	n/a	$2V$	$3V$	1
Interrupt	11	39	16	24

Table 5.11: Time cost models for all targets

allowed to scale with the clock frequency (PIC18F, ATmega and MSP430) the minimum supply voltage relates linearly to the clock frequency, resulting in a fourth-order relation between power consumption and frequency. With this information it is possible to make an accurate fit on the information given in the datasheet and derive a model that gives the power consumption based on the required clock frequency. The resulting models for the power consumption are given in Table 5.12.

The power consumption of the Spartan 3 can be calculated using a web form or a spreadsheet. No model is available or derivable from the web form or spreadsheet.

Microcontroller	**Power consumption model (mW)**	**Requirements**
PIC18F	$P(F) = 0.0000028653F^4 + 0.047137F^2 + 1.2451.$	$F \geq 4\,\text{MHz}$
ATmega	$P(F) = 0.00011162F^4 + 0.48180F^2 - 3.7300$	$F \geq 8\,\text{MHz}$
MSP430	$P(F) = 0.000165F^4 + 0.0720F^2 + 1.07$	$F \geq 4.15\,\text{MHz}$
LPC2144	$P(F) = 1.65F + 33$	

Table 5.12: Power consumption models for all targets

Chapter 6

Experimental results

This chapter describes the results of using the models presented in the previous chapters to predict the costs of a piece of sample code.

The subject of this test is a moving average filter, implemented in C, using fixed point arithmetic with 16 bits after the decimal dot. The code of this filter is the following:

C code

```c
#define fix 16
long fixmult(long a, long b) {
        return (a * b) >> fix;
}

long sx0,sx1,sx2,sx3,sx4,sx5;
long b0,b1,b2,b3,b4,b5;

long filter(long ixsample) {
        sx5=sx4;
        sx4=sx3;
        sx3=sx2;
        sx2=sx1;
        sx1=sx0;
        sx0=ixsample;
        return fixmult(b0, sx0)
             + fixmult(b1, sx1)
             + fixmult(b2, sx2)
             + fixmult(b3, sx3)
             + fixmult(b4, sx4)
             + fixmult(b5, sx5);
}
```

The filter coefficients `b0`..`b5` are loaded with pre-calculated values ($\frac{1}{6}$, because this a moving average filter). For every new incoming data sample the `filter` function is executed. The result of this function is a new sample of the filtered signal.

6.1 Detailed machine model

Analyzing the code using the detailed machine model results in Table 6.1.

Operation	**`fixmult`**	**`filter`**
Long		
Assignment	0	6
Addition/subtraction	0	5
Extra for the use of a global variable	0	23
Multiplication	1	0
Short two-term expression	1	6
Right shift	1	0
Function calls		
Returning long	1	1
Context save	0	1
Per long argument	2	1
Other called functions		
fixmult		6

Table 6.1: Analysis of the filter in terms of operations

Multiplying these values with the costs given in Table 4.1, results in the figures of Table 6.2. Note that the ARM7 is not listed because no detailed model is available. The values for `filter` include six times the costs of `fixmult`.

The table has a minimum, maximum and a modified column. This is because the costs of multiplication for the PIC18F and bit shifting for all targets are given as a range. The minimum and maximum columns use respectively the low and high value of the range of multiplication and bit shifting. These results are quite far apart and not realistic. Therefore, a modified version was made in which the costs of bit shifting are based on Tables A.6 A.7 and A.8. For multiplication of longs on the PIC18F the value of 59 cycles was taken, the costs of the multiplication when the hardware multiplier would be used.

`fixmult`	**Minimum**	**Maximum**	**Modified**
PIC18F	72	946	277
ATmega	74	194	79
MSP430	60	100	66
`filter`			
PIC18F	595	5839	1825
ATmega	687	1407	717
MSP430	511	751	547

Table 6.2: Analysis of the `fixmult` and `filter` functions in terms of time costs using the detailed model

6.2 Abstract machine model

Analyzing the code using the abstract machine model gives the results as listed in Table 6.3. For all operations $B = 32$ bits, so V(PIC18F, ATmega, MSP430, ARM7)=4,4,2,1 and V_2(PIC18F, ATmega, MSP430,ARM7)=2,2,1,1. Calculating the costs using the models from Table 5.11 results in Table 6.4.

The standard deviation values result from the costs of bit shifting. In total 6 bit shift operations are executed, all 6 with $p = 16$ and $B = 32$ so the resulting costs per operation are all the same. When seen as random variable, the values are correlated with a correlation coefficient of +1, therefore the standard deviation values have to be added up to get the complete standard deviation (instead of using $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2 + \ldots + \sigma_n^2}$ for independent variables).

For more accuracy, a modified version was made and the results are added to the table under 'Modified version'. Instead of treating the costs of bit shifting as a random variable, the real costs of a shift-by-16 operation were used.

Operation	**`filter`**	**`fixmult`**
Assignment	12×32 bits	1×32 bits
Addition, subtraction, (boolean) and, (boolean) or, xor	6×32 bits	
Using a global variable	23×32 bits	
Multiplication	6×32 bits	1×32 bits
Bit shifting	6×32 bits, $p = 16$	1×32 bits, $p = 16$
Function call and return	7×32 bits	1×32 bits
Function argument	13×32 bits	2×32 bits
Context save	1	

Table 6.3: Analysis of the `fixmult` and `filter` functions in terms of operations using the abstract model

	Mean (μ)	**Standard deviation (σ)**	**Modified version**
PIC18F	1210,58		1960.58
ATmega	1291,17	261,6	773.97
MSP430	764,39	88,8	599.99
ARM7	158		158

Table 6.4: Analysis of the complete `filter` function in terms of time cost using the abstract model

6.2.1 FPGA implementation

The operations that are of interest for the FPGA implementation are 6 multiplications, 6 additions and the storage of `sx0..sx5` and `b0..b5`; 12 32-bit numbers. Bit shifting is free in an FPGA, it is just a matter of different routing. Based on the models from Table 5.7 the resource costs have been calculated for the Spartan 3. See table 6.5 for the results. The table shows that 24 multipliers are needed while the XC3S400 has only 16. This means that a larger FPGA has to be chosen or something has to be implemented to share the multiplier between the calculations. The total use of area is 11% of the available slices.

	LUTs	**Flipflops**	**Slices**	**Mults**
Multiplication	567.1	0	283.55	24
Addition, subtraction, less than, greater than	192	0	96	0
Storing a number	0	384	0	0
Total (rounded)	759	384	380	24

Table 6.5: Costs of implementing a moving average filter on the Spartan 3

6.3 Simulation

When simulating this code some additional code was written that fills `b0..b5` with the pre-calculated value of $\frac{1}{6}$ and executes the `filter` function with a sample value. Then the code was executed in the simulator of the IAR development environment and the time to execute the function was measured using the cycle counting functionality of the simulator. The results are in Table 6.6.

The PIC18F compiler does multiplication using the software algorithm, which costs in this case on average 90 cycles per multiplication, instead of the 59 cycles used in the detailed model and 64 cycles in the detailed model, leading to an error of +156 to +186 cycles.

	`filter`
PIC18F	1938
ATmega	773
MSP430	589
ARM7	141

Table 6.6: Analysis of the complete `filter` function in terms of time cost using simulation

6.4 Comparison

Table 6.7 compares the results of the detailed model and the abstract model with the simulation. This table shows the results of the simulation, which are taken as correct and used as reference to calculate the errors of the other methods. The detailed minimum, maximum and modified numbers are from Table 6.2. The minimum and maximum values are unusable because of their large error, which was expected. The results of the modified version are quite good. The relative error is constant, 6-7%, which makes these figures really useful for making a choice between hardware platforms. The unmodified abstract model gives good results, were it not that the standard deviation is quite large for the ATmega and MSP430, leading to large errors. When using the modified model the results are much better, with even a smaller error than the detailed model. However, this fact cannot be taken as general conclusion.

Platform	**Simula-tion**	**Detailed mini-mum**	**Detailed maxi-mum**	**Detailed realistic**	**Abstract**	**Modified abstract**
Mean						
PIC18F	1938	595	5839	1825	1810,58	1888,58
Atmega	773	687	1407	717	1291,17	749,97
MSP430	589	511	751	547	764,39	575,99
ARM7	141				158	158
Standard deviation						
PIC18F						
ATmega					261,6	
MSP430					88,8	
ARM7						
Absolute error						
PIC18F		-1343	3901	-113	-127,42	-49,42
ATmega		-86	634	-56	518,17	-23,03
MSP430		-78	162	-42	175,39	-13,01
ARM7					17	17
Relative error						
PIC18F		-69%	201%	-6%	-7%	-3%
ATmega		-11%	82%	-7%	67%	-3%
MSP430		-13%	28%	-7%	30%	-2%
ARM7					12%	12%

Table 6.7: Comparison of the resulting predicted costs for all models

6.5 Power consumption

Based on the costs of the algorithm the maximum sample frequency of this filter can be determined for each target. The results are listed in Table 6.8. This table also lists the minimum frequency for which the power models are valid and the maximum frequency the microcontroller can run on. Together with the power models from the previous chapter the relation between the sampling frequency and the power consumption can be determined.

The power consumption of the Spartan 3 is determined using the web form. The chosen variant is the XC3S1000 instead of the XC3S400 because the filter needs 24 multipliers. The XC3S1000 does have 24 multpliers and consumes 92 mW in rest. 24 multipliers consume 1 mW at 1 MHz, 9 mW at 10 MHz and 90 mW at 100 MHz, which is linear in F. The logic (LUTs and flipflops) consumes 0 mW at 1 MHz[1], 4 mW at 10 MHz and 44 mW at 100 MHz, which is linear in F too. The Spartan 3 can filter 1 sample per clock cycle. So the power consumption of the XC3S1000 in mW is given by $P(F) = 92 + 1.34F$.

The power consumption of all platforms is graphed in two variants, one for the lower frequencies, Figure 6.1, and one for the higher frequencies, Figure 6.2. The graphs only show the regions where the models are valid. The power consumption of the XC3S400 is also graphed for reference, as if it had enough multipliers.

	Minimum clock frequency (MHz)	**Maximum clock frequency (MHz)**	**Minimum sample frequency (kHz)**	**Maximum sample frequency (kHz)**
PIC18F8722	4	40	0.52	5.2
ATmega128	8	16	10.3	20.7
MSP430F2619	4.15	16	7.0	27.2
LPC2144	0	60	0	425.5

Table 6.8: Clock and sample frequency ranges for which the power models are valid

The ranking of the platforms depend on the required filtering frequency. For frequencies above 425 kHz an FPGA is the only choice. For frequencies above 255 kHz the power consumption of the XC3S1000 is lower than the LPC2144 so the XC3S1000 is the best choice when looking for the lowest power consumption. On the other hand, the time-to-market could be longer, preferring a microcontroller over an FPGA.

For frequencies lower than 27 kHz the MSP430F2619 is available and is the best choice because of the low power consumption. The second best choice could be the ATmega128 or the LPC2144, dependent on the frequency. The PIC18F8722 is the worst choice in all cases.

6.6 Time spent on modeling

The time spent to make the model of the code (counting the operations) using the abstract method was less than the time spent on making a model using the detailed model of the platform. On the other hand, calculating the time cost takes a little more time, because the sizes of the operation have to be incorporated in the calculation.

In this case making a model of this code did cost one hour for the detailed model and 45 minutes for the abstract model. The time spent on processing the information to get the time costs was 30 minutes for the detailed model and 45 minutes for the abstract model.

[1]This is 0 mW due to rounding applied by the tool instead of 0.4 mW

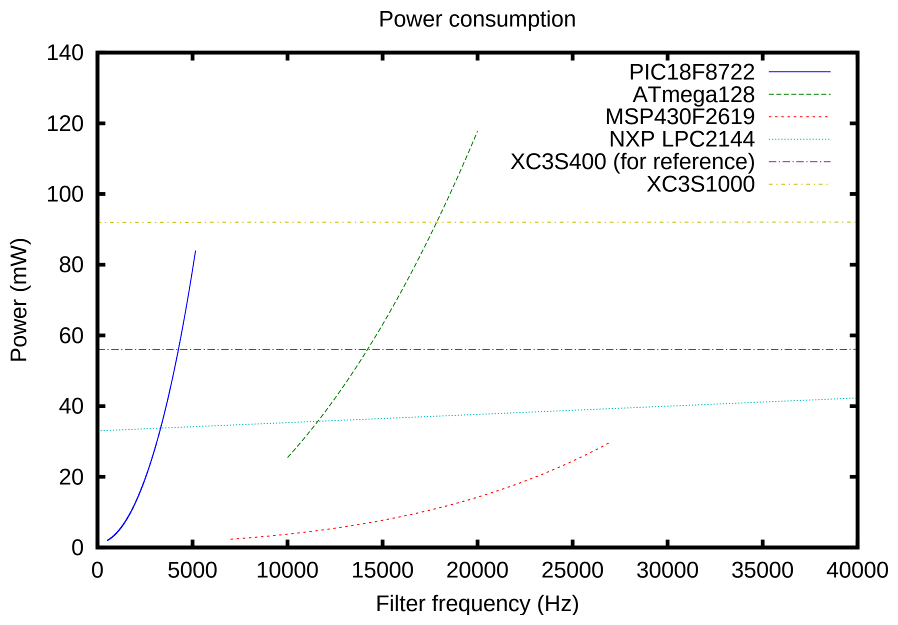


Figure 6.1: Power consumption for filter frequencies up to 40 kHz

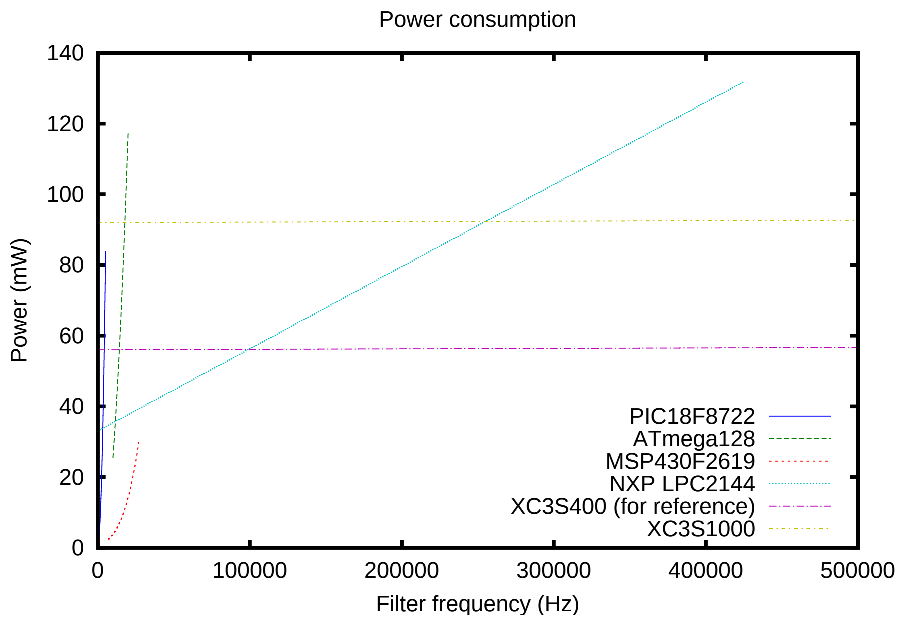


Figure 6.2: Power consumption for filter frequencies up to 500 kHz

For longer pieces of code the time spent on making a model of the code would increase linearly with the length of the code. The time for processing the results would remain the same. We can say that given the accuracy of the results, the abstract model would be the best choice to use for an electronics designer.

6.7 Conclusion

Based on the above mentioned results we can say the results of determining the time consumption of an algorithm containing bit shift operations using the unmodified detailed and abstract machine models are poor. The largest error in this case was for the ATmega: 67%. However, when the costs of the bit shifting operation are more precisely determined the error drops to 6 to 7% for the detailed machine model and 2-3% for the abstract machine model. When the ARM7 is included the error is larger because in the case of the ARM7 the error was 12%. We can conclude that if an algorithm does not contain bit shift operations both the detailed and the abstract model are able to give a quite good measure of the real-life time cost of an algorithm. If the algorithm does contain bit shift operations then the cost of the bit-shift operation have to be looked up or measured separately to get accurate results.

With the information about the time consumption of an algorithm on a specific hardware platform it is possible to calculate the power consumption for the algorithm when executed with a certain frequency. This gives a good view of the performance of the diverse platforms and is excellent for ranking the platforms.

The time spent on making a model of the code is acceptable; one hour and 45 minutes for both models, but for the abstract model less time was spent on making the model of the code. As the time spent on modeling the code increases linearly with the lenght of the code, the abstract method is the best choice.

Chapter 7

Conclusions and Future Work

The goal of this thesis was to provide a method to give an electronics engineer an instrument to help him make a well-founded choice for a certain hardware platform. A method to visualize the costs of a piece of code for a range of hardware platforms. This method should be practicable in a small amount of time and provide enough precision to get a reliable result.

We provided two ways of modeling a hardware platform: using a detailed model and using an abstract model. By then defining an algorithm in terms of operations, one could easily calculate the cost in terms of time for a certain algorithm. Also, power models have been derived to calculate the power consumption based on the running clock frequency, to calculate the power cost of an algorithm.

The detailed model was derived by looking at how the operations in C were transformed into assembly code by the compiler and what the costs were of that machine code. This resulted in a total of 73 operations and special cases. Examples of normal operations are: 8, 16 and 32 bits addition, multiplication and pointer dereferencing. Special cases, like using a constant of 0 or 0xFF, could make small difference, so these were included in the detailed model.

The abstract model parameterizes the size of the operation. It also neglects special cases. This brought the number of operations down to only 15. This is not much but proved to be sufficient for characterizing an algorithm. Experiments showed a maximum error of 7% for the detailed model and a maximum error of 12% for the abstract model, however in general, the error of the detailed model was lower than that of the abstract model.

When power is an important issue for the designer, the power models derived for the hardware platforms are a great help in showing the differences between the platforms. The power models show a fourth order relationship between clock frequency and power consumption if the supply voltage can be varied and a linear relationship if the supply voltage is fixed.

The usefulness of the power models is shown in the experiments where for certain situations one microcontroller was the best choice and for other situation another would be better. In the experiments, there was no over-all best choice.

We have also shown that an algorithm in C can be projected on an FPGA by looking at the mathematical operations happening. Models have been derived to show the costs of the mathematical operations in terms of FPGA resources. With this information, also the power consumption can be calculated using the tools given by the manufacturer. Power versus performance, unit cost and expected time-to-market would be decisive for choosing between an FPGA or a microcontroller.

All this answers the research questions about how to determine the execution time and power consumption of a microcontroller running a piece of C code and the question of how an FPGA can be compared to a microcontroller.

The time spent on calculating the time cost using the abstract model was less than when using the detailed model. The precision of the results was not much different, and was for both models definitely sufficent for visualizing the differences in preformance and power consumption of the platforms. Therefore the abstract model is preferred.

With the knowledge gathered in this research also the last two research questions can be answered:

What is the trade-off between the detail of characterization of an algorithm *and the precision of the result?* The algorithm needs to be sufficiently precise analyzed to get the right number of operations. Missing a detail, making a mistake or incorrectly categorizing an operation usually does not cause extreme errors. However, when the error is made inside of a loop that is executed multiple times the error counts every loop iteration, possibly leading to a larger error. On the other hand, errors in analyzing the algorithm does not directly lead to a favor or discrimination of a certain platform, and thus does not influence the designer's choice for a certain platform.

What is the trade-off between the detail of characterization of a platform *and the precision of the result?* There is always a trade-off between modeling corner cases and the complexity of the model. Remarkable in this research was that the experiment showed that reducing the model from 73 operations to 15 parameterized operations reduced the overall error. Errors are unavoidable, but should be minimized to avoid undeserved favor of a certain platform. On the other hand, the platforms analysed in this research are sufficiently different that an error in the model would not lead to a different choice.

7.1 Reflection

In this research I learned a lot about the deep inner working of microcontrollers. I learned about the advantages and disadvantages of certain architectures and how certain architectural choices can make a certain operation costly or cheap.

I was amazed by how bad the performance of the PIC18F architecture was. I knew that the PIC18F needed 4 clock ticks per cycle but due to the single register architecture the real performance is even lower. With the moving average filter the 8 bits ATmega architecture outperforms the same sized PIC18F architecture by a factor 10 at the same clock speed and a factor 4 when running both at their maximum speed.

Should the method presented in this research be used in practice? An experienced electronics designer most probably will not use it because he already knows that the MSP430 has the lowest power consumption and if he needs really much power he should choose an ARM7 or an FPGA. But in that vague region where it is not sure whether the MSP430 could be fast enough and the ARM7 consumes too much power the method presented in this thesis could be decisive.

I think that this research has been quite extensive on this subject. Eventually more research could be done on:

- Building a database of models of more hardware platforms.
- The creation of an automatic algorithm analyzer that analyzes the code and gives the resulting performance for all the platforms in the database.
- The mapping of algorithms on FPGA hardware. In this research only mathematical operations were considered, but the possibilities of FPGAs are much more extensive.
- Modeling the costs of bit shifting. Modeling algorithms with bit shifting results in large errors when using the unmodified models.

Bibliography

[1] ARM Limited. *Application Note 93 Benchmarking wit ARMulator*. http://infocenter.arm.com/help/topic/com.arm.doc.dai0093a/DAI0093A_benchmarking_appsnote.pdf.

[2] ARM Limited. *ARM Architecture Reference Manual*. http://www.arm.com/miscPDFs/14128.pdf.

[3] ARM Limited. *ARM7TDMI-S Technical Reference Manual Revision: r4p3*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0234b/DDI0234.pdf.

[4] Atmel Corporation. *ATmega128 and ATmega128L datasheet — 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash*. http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf.

[5] Atmel Corporation. *Atmel 8-bit AVR Instruction Set*. http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf.

[6] Lutz Bierl. *MSP430 Family Mixed-Signal Microcontroller Application Reports*. Texas Instruments. http://focus.ti.com/lit/an/slaa024/slaa024.pdf.

[7] Dennis M. Ritchie Brian W. Kernighan. *The C Programming Language — Second edition*. Prentice Hall P T R, 1988. ISBN 0-13-110362-8 (paperback) ISBN 0-13-110370-9. Online version available from http://www.cs.utah.edu/~phister/K_n_R/index.html.

[8] How many mips can the lpc2106 do? Embeddedrelated.com LPC2000 Mailing list, Dec 2003. http://www.embeddedrelated.com/groups/lpc2000/show/117.php.

[9] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. John Wiley, 2006.

[10] Jon Kirwan. MSP430. Jon Kirwan's personal webpage, July 2004. http://www.infinitefactors.org/jonk/msp430.html.

[11] Microchip Technology Inc. *PIC18F8722 Family Data Sheet*. `http://ww1.microchip.com/downloads/en/devicedoc/39646b.pdf`.

[12] NXP Semiconductors. *LPC2141/42/44/46/48 Product data sheet*. `http://www.nxp.com/acrobat_download/datasheets/LPC2141_42_44_46_48_3.pdf`.

[13] Behrooz Parhami. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press, Oxford, UK, 2000.

[14] Philips expands 32-bit arm microcontroller family for cost-sensitive designs. Philips Press Release, October 2003. `http://www.nxp.com/news/content/file_999.html`.

[15] Rafael H. Saavedra and Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, 1996.

[16] NXP semiconductor. *UM10139 Volume 1: LPC214x User Manual*. `http://www.standardics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc2141.lpc2142.lpc2144.lpc2146.lpc2148.pdf`.

[17] Dimitrios Soudris. *Designing Cmos Circuits For Low Power*. Kluwer Academic Publishers, october 2002.

[18] Texas Instruments. *MSP430x241x, MSP430x261x Mixed Signal Microcontroller*. `http://focus.ti.com/lit/ds/symlink/msp430f2619.pdf`.

[19] Texas Instruments. *MSP430x2xx Family User's Guide*. `http://focus.ti.com/lit/ug/slau144e/slau144e.pdf`.

[20] Sijmen Woutersen. The X32 softcore – a top-down approach on processor design. Master's thesis, TUDelft, 2006. `http://x32.ewi.tudelft.nl/woutersen_thesis.pdf`.

[21] Xilinx. *Xilinx Application Note XAPP477 — Embedded Processing and Control Solutions for Spartan-3 FPGAs*, August 2003. `http://www.xilinx.com/support/documentation/application_notes/xapp477.pdf`.

[22] Xilinx. *Spartan-3 FPGA Family: Complete Data Sheet*, November 2007. `http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf`.

[23] Xilinx. *Spartan-3 Generation FPGA User Guide*, February 2008. `http://www.xilinx.com/support/documentation/user_guides/ug331.pdf`.

Appendix A

Derivation of the detailed models

This appendix describes the derivation of the detailed model presented in Chapter 4 in full detail. The first section describes a very detailed way of modeling, applied to data transport statements. This way of modeling turns out to be too detailed and too labour-intensive. Then the following sections describe the more general analysis of the other operations.

A.1 Data transport

The following subsections describe operations related to data transport. Most described operations have an example with C code and the assembly code produced by the compiler. For more details about the instructions used in the assembly code, refer to [11] for the PIC18F, to [5] for the ATmega and [19] for the MSP430.

A.1.1 Assignments — PIC18F

Although all assignments in C use the '=' symbol, technically they can be different. A variable could be a local variable or a global one and sometimes the variable is not a variable but an I/O port. All these cases will be covered for all microcontrollers, starting with the PIC18F.

Assigning the contents of one variable to another

For the PIC18F there is no difference between global and local variables, they all reside in `F`, the data memory.

C code
```
a = b;
```

PIC18F assembly
```
C0 0B
F0 0A   MOVFF    b,a
```

This `MOVFF` (move from data memory to data memory) instruction takes 2 cycles.

Assigning a constant to a variable

It is possible to create a constant by using the `MOVLW` instruction:

C code
```
b = 14;
```

PIC18F assembly
```
0E 0E   MOVLW    0xe
6F 0B   MOVWF    b,1
```

These instructions take two cycles in total: 1 for `MOVLW` (move literal to `W`) and 1 for `MOVWF` (move W to the data memory). There are two special cases: when assigning zero or 0xff:

C code
```
a = 0xff;
a = 0;
```

PIC18F assembly
```
69 0A   SETF     a,1
6B 0A   CLRF     a,1
```

Assigning a zero or 0xff can be done by using a `CLRF` or `SETF` instruction and takes only one cycle.

Writing a memory variable to an I/O port

C code
```
PORTB = j;
```

PIC18F assembly
```
C0 0A
FF 81   MOVFF    j,_A_PORTB
```

This `MOVFF` (move from data memory to data memory) instruction takes 2 cycles.

Writing a constant to an I/O port

C code
```
PORTB = 12;
```

PIC18F assembly
```
0E 0C   MOVLW    0xc
6E 81   MOVWF    _A_PORTB,0
```

These instructions take two cycles in total: 1 for `MOVLW` (move literal to `W`) and 1 for `MOVWF` (move W to the data memory).

Larger datatypes and global variables

For the PIC18F there are no differences whether the variable is global or local; all variables are accessed via the `F`, the data memory. When the compiler decides to place the variable in a different bank it takes two cycles extra: one to switch to the other bank and one to switch back.

For larger datatypes the time it costs scales linearly in the number of bytes: an assignment between two ints takes 4 cycles and an assignment between two longs takes 8 cycles.

A.1.2 Assignments — ATmega

The ATmega has 32 working registers. The IAR compiler usually assigns some of these registers as ‘saved-across-call’, some as ‘scratch’ and some as call arguments. Global and static variables are usually stored in the data memory and local variables are usually saved in the saved-across-call or scratch registers, depending on whether the function contains calls to other functions or not.

Assigning the contents of one variable to another

C code
```
c = a;
```
ATmega assembly
```
2F10     MOV     R17,R16
```

Assigning between two local variables that are stored in registers takes only one cycle.

Assigning a constant to a local variable

C code
```
c = 12;
```
ATmega assembly
```
E00C     LDI     R16,0x0C
```

Constants can be 'generated' using the `LDI` instruction. This instruction has a limitation: only the registers R16 to R31 can be used as destination. An `LDI` instruction takes one cycle.

Assigning the contents of a local variable to an I/O port

C code
```
PORTB = a;
```
ATmega assembly
```
BB08     OUT     PORTB,R16
```

To write to an I/O register an `OUT` instruction is used. This instruction takes 1 cycle.

Writing a constant to an I/O port

C code
```
PORTB = 12;
```
ATmega assembly
```
E00C     LDI     R16,0x0C
BB08     OUT     PORTB,R16
```

First an immediate value (0x0C = 12) is loaded into a register and then the contents of a register are written to the I/O register using the `OUT` instruction. An `LDI` instruction takes 1 cycle, and an `OUT` instruction too, so in total this takes 2 cycles.

Non-local variables

There are two methods to use the data memory: using the `LDS` and `STS` (Load Direct from Data Space and Store Direct to Data Space) instructions or using one of the pointer registers.

Examples of `LDS` and `STS`:

C code
```
b = nonlocal;
```
ATmega assembly
```
9130 0060 LDS  R19,nonlocal
```

C code
```
nonlocal = b;
```
ATmega assembly
```
9320 0060 STS  nonlocal,R18
```

`LDS` and `STS` instructions take two cycles, so comparing with a local assignment an assignment concerning a global variable takes one cycle more. When both variables are non-local an assignment takes four cycles:

C code
```
nonlocal = nonlocal2;
```
ATmega assembly
```
9100 0141 LDS  R16,nonlocal2
9300 0140 STS  nonlocal,R16
```

The compiler can choose to assign via a pointer register:

C code

```
b = nonlocal;
```

ATmega assembly

```
E6E0     LDI     R30,0x60
E0F0     LDI     R31,0x00
8130     LD      R19,Z
```

This takes 4 cycles; 1 per `LDI` instruction and 2 for the `LD`. One `LDS` instruction would be faster (2 cycles) but there could be a reason for the compiler to use a pointer register, for example in the following situation:

C code

```
a = arr[i++];
```

In this situation an `LD Rx,Z+` instruction could be used to read from the memory and increment the Z pointer register afterwards in only two cycles.

Larger datatypes

In principle larger datatypes scale linearly in the number of bytes but the amount of local variables is limited by the registers. When there are too many local variables, the compiler maps some variables to the data memory. The ATmega also has a `MOVW` instruction that can do assignments between register pairs. This saves cycles in assignments between local variables that are ints or longs. Example (`b` and `b2` are `int`, `c` and `c2` are `long`):

C code

```
b = b2;

c = c2;

```

ATmega assembly

```
01CD     MOVW    R24,R26

01A8     MOVW    R20,R16
01B9     MOVW    R22,R18
```

For large non-local variables the compiler can use the `LDD` instruction (Load Indirect Data with Displacement). In this example `c` is a local long and `nc` is a non-local long:

C code

```
1  nc = c;
```

ATmega assembly

```
E4E5     LDI     R30,0x45     1
E0F1     LDI     R31,0x01     2
8300     ST      Z,R16        3
8311     STD     Z+1,R17      4
8322     STD     Z+2,R18      5
8333     STD     Z+3,R19      6
```

In line 1 and 2 the pointer register Z (R31:R30) is loaded with the address of `nc` (0x0145) and then the contents of registers R16 to R19 are transferred to `nc` using `ST` and `STD` instructions. This assignment takes 1+1+2+2+2+2=10 cycles.

A.1.3 Assignments — MSP430

The MSP430's addressing modes make it possible to use only the `mov` instruction for all kinds of sources and destinations. The amount of cycles depends on the addressing mode. The complete overview of the cycle counts of all different cases can be found in [19, table 4-10]

Assigning the contents of one local variable to another

Assuming that the local variables are stored in registers, a `mov` instruction moves the data from one register to another:

```
C code
a = b;
```

```
MSP430 assembly
4F4E           mov.b  R15,R14
```

This is an instruction with a register as source and destination. It takes 1 cycle. If the destination is a memory variable, the instruction has the address as extra parameter. The `mov` instruction here has a `.b` suffix, meaning that it operates on only 8 of the 16 bits. This does not influence the execution times.

```
C code
nl = a;
```

```
MSP430 assembly
4EC2 1106      mov.b  R14,&nl
```

With a memory location as destination, this instruction takes 4 cycles. There is a difference when the memory is the source, as in the following example:

```
C code
a = nl2;
```

```
MSP430 assembly
425E 1107      mov.b  &nl2,R14
```

In this case the `mov` instruction takes 3 cycles, so it clearly depends whether the memory is the source or the destination. It is also possible to move from the data memory to the data memory, without using a register:

```
C code
nl = nl2;
```

```
MSP430 assembly
42D2 1107 1106 mov.b  &nl2,&nl
```

In this case the `mov` instruction takes 6 cycles, 1 less than when moving via a register. This shows that using global variables can cause is a serious performance penalty, using a global variable can cost 3 to 6 times more cycles.

Using I/O ports and special cases

As described in Section 3.3.1, I/O ports are memory mapped, so writing to an I/O port is the same as writing to a memory location. The following example demonstrates writing a local variable (`j`), a constant (12) and a global variable (`nl`) to the I/O port:

```
C code
P1OUT = j;
P1OUT = 12;
P1OUT = nl;
```

```
MSP430 assembly
4CC2 0021      mov.b  R12,&P1OUT
40F2 000C 0021 mov.b  #0xC,&P1OUT
42D2 1106 0021 mov.b  &nl,&P1OUT
```

These `mov` instructions take respectively 4, 5 and 6 cycles, which is exactly the same as for a global variable.

For assignments on the MSP430, there are no real special cases. Writing a 0 to a register or I/O port looks like a special case:

```
C code
P1OUT = 0;
```

```
MSP430 assembly
43C2 0021          clr.b   &P1OUT
```

But this is no special case, because the `clr.b &P1OUT` instruction is a pseudo instruction that executes `mov.b #0, &P1OUT`. It's only to make the assembly code better readable.

Larger datatypes

The MSP430 is a 16-bit microcontroller, all instruction work as well on 8-bit data as on 16-bit data at the same execution speed. There is no difference in execution speed whether the code uses a 8 bits char or a 16 bits int. The execution time for larger datatypes, like long, scales linearly in the number of words.

A.1.4 Data transport summary

A summary of the analysis of the execution time of possible kinds of assignments is listed in Table A.1. This table already contains 10 cases, but these are not even all possibilities, because, for example using an I/O port as source is not covered. Neither are other data sizes than 8 bits.

As this method looks at every different kind of combination, it should provide the most precise approximation of the execution time but due to the enormous amount of possible cases it is unusable in practice. Defining all possible cases is impossible and characterizing a microcontroller based on these cases would cost far too much time. Modelling a hardware platform this precise costs too much time and is therefore not usable..

Source	**Destination**	**PIC18F**	**ATmega**	**MSP430**
Local	Local	2	1	1
Constant	Local	2	1	2
Constant 0 or 0xff (special case)	Local	1	1	2
Global	Local	4	2	3
Local	I/O port	2	1	4
Constant	I/O port	2	2	5
Global	I/O port	4	3	6
Local	Global	4	2	4
Constant	Global	4	3	5
Global	Global	4	4	6

Table A.1: Cycle counts for different 8-bit assignments

A.2 Arithmetic

Looking at the results from the costs of data transport in Table A.1, there is much regularity: for the PIC18F an assignment is nearly always 2 cycles and always 4 cycles when a global variable is involved. For the ATmega an assignment is 1 cycle and an extra cycle is added for each I/O port, constant or global variable involved. For the MSP430 an assignment between local variables is 1 cycle, using a constant adds 1, using an I/O port adds 2, a global as source adds 2 and as destination 3 cycles. This generalizes the cycle count for the assignment a little. But it can be done more general.

Assignment only instructions are not frequently used. In most cases, some calculation is done and the result is assigned to variable. A typical line of code containing an assignment can look as follows:

C code
```
h = a + b - c + d;
```

A compiler could translate this to something like:

Assembly pseudocode
```
; a in R5, b in R6, c in R7, d in R8 and h in R9
mov R5 to R9
add R6 to R9
sub R7 from R9
add R8 to R9
```

This code contains one `mov` instruction, two `add`s and one `sub`. This is exactly what the original code contained: one '=', two '+' signs and one '-'. So what are the costs of an addition? One add instruction. What are the costs of an assignment? One `mov` instruction. Isn't this method of looking at the costs of an algorithm in cycles much better? Yes it is, this chapter shows the results. This chapter builds on the assumption that there is a direct relation between the operands and the cycle count for code lines.

The method used to determine the cycle count of C code lines has less focus on the assembly representation generated by the compiler but more on the differences between two code lines with a different number of operations. Consider for example the following code lines with their cycle count:

C code
```
h = a + b;              // 2 cycles
h = a + b + c;          // 3 cycles
h = a + b + c + d;      // 4 cycles
h = a + b + c + d + e;  // 5 cycles
```

When this would be the result of the cycle counting it is safe to decide that an assignment costs 1 cycle and each addition also 1 cycle. Care has to be taken that the compiler doesn't do unexpected things (like placing a variable in the data memory instead of keeping it in a register), which would lead to a wrong conclusion. Also not all situations are as straightforward as this one, the PIC18F for example has a `SUBWF` instruction to subtract `WREG` (the working register) from `F` (a value in the data memory) but no instruction to subtract `F` from `WREG`. The effects on the cycle count are demonstrated in the following example:

C code
```
1  h = a - b + c + d; // 5 cycles
2  h = a + b - c + d; // 7 cycles
3  h = a + b + c - d; // 7 cycles
4  h = a + b + c + d; // 5 cycles
```

In line 1 and 4 everything is fine, 2 cycles for the assignment and 1 for the addition or subtraction. In line 2, the compiler lets the microcontroller calculate `a + b`, store the result in the memory, load `c` in `WREG`, subtract the result of `a + b` and then add `d`. The same goes for line 3. Cases like this will be mentioned as exception.

The following sections describe the results of this approach for standard arithmetic on the PIC18F, the ATmega and the MSP430.

A.2.1 Arithmetic on the PIC18F

Researching the cycle count of arithmetic on the PIC18F resulted in the figures listed in Table A.2.

	Char **8 bits**	**Int** **16 bits**	**Long** **32 bits**
Assignment	2	6	12
Addition/subtraction	1 or 3[a]	4	8
Or/and/xor	1	3	7
Extra for the use of a constant $\leq$ 0xff	0	-1	-2
Extra for the use of a constant $\leq$ 0xffff	n/a	0	-1
Extra for the use of a constant $\leq$ 0xffffff	n/a	n/a	0
Extra for the use of a constant $\leq$ 0xffffffff	n/a	n/a	0
Extra for the use of a global variable	2	2	2
Extra for a short, assignment-only instruction	0	0	0
Multiplication	2	13	40 to 740
Short two-term expression	0	-2	-2
Multiplication only expression	0	-8	0
Parentheses	4	4	4
Pointer	14	18	21

Table A.2: Arithmetic on the PIC18F

[a]See text.

The costs for subtracting are 1 cycle when the subtraction is the first operation or a constant is involved and 3 otherwise. This is because of the `SUBWF` instruction that can only subtract a value in the data memory from the working register's value and not the other way around, so the compiler has to move some data around to get the right results.

Using a memory variable from a different bank, as is the case with global variables, costs 1 cycle to switch forth and 1 to switch back. When operating on global variables only, bank switching is usually not needed. However, global variables are usually used sparingly so it was decided to count 2 cycles for the use of a global variable.

Using parentheses comes with some costs if the order of operation is changed. The IAR compiler does not optimize the expression to overcome these costs so they have to be counted. The costs only occur if order of operation is changed. For example when calculating $(a+b)*c$ the parentheses have to be counted while when calculating $a+(b*c)$ they don't have to be.

Assigning constants can save cycles in some cases, for example when assigning a constant smaller than or equal to 0x00ff to an int variable. In this case, the compiler uses the `CLRF` instruction to directly assign 0x00 to the upper byte and a normal `MOVLW`/`MOVW` (move literal to `W` and move `W` to `F`) instruction combination for the nonzero byte.

In some other cases the compiler can also do some optimization, such as when the expression has only two terms (`a = b + c;`) or when the expression is a multiplication only expression (`a = b * c * d;`).

Multiplication is costly, especially for longs. This is because the IAR compiler decides to do multiplication of longs in completely software instead of using the hardware multiplier. If the hardware multiplier would be used a multiplication of longs would take 59 cycles.

Although the PIC18F datasheet states that the PIC18F has a C compiler optimized architecture because of an "Optional extended instruction set designed to optimize re-entrant code" the costs of using a pointer (an important feature of the C programming language) are disproportional when comparing with other operations or other platforms.

A.2.2 Arithmetic on the ATmega

The cycle counts of arithmetical operations on chars (the native data size for the ATmega) are listed in Table A.3.

	Char 8 bits	**Int 16 bits**	**Long 32 bits**
Assignment	2 or 1[a]	2	4
Addition/subtraction	1	2	4
Or/and/xor	1	2	4
Extra for the use of a constant $\leq$ 0xff	0	-1	4
Extra for the use of a constant $\leq$ 0xffff	n/a	0	4
Extra for the use of a constant $\leq$ 0xffffff	n/a	n/a	4
Extra for the use of a constant $\leq$ 0xffffffff	n/a	n/a	4
Extra for the use of a global variable	2	4	8
Extra for a short, assignment-only instruction	0	0	0
Multiplication	1	20	53
Short two-term expression	0	0	0
Multiplication only expression	0	-1	-2
Parentheses	1	1	2
Pointer	1	1	1

Table A.3: Arithmetic on the ATmega

[a]See text.

The assignment of chars is 1 cycle instead of 2 when the target is the first term of the equation. In the assignment of ints the ATmega has benefit by its `MOVW` instruction: although the bitsize increases from 8 to 16 bits, the costs of an assignment stay the same.

When working with longs in an algorithm the amount of registers limits the practical amount of local variables. With the IAR compiler the maximum amount of local long-typed variables is limited to about 4. When more than 4 local longs are used the compiler stores them on the stack. This means that those variables must be considered global when counting the cycles.

A.2.3 Arithmetic on the MSP430

The native size of the MSP430 is 16 bits. Because all instructions have a 16-bit and a 8-bit variant (`.b` suffix) the cycle count for chars is in nearly all cases the same as for ints. Table A.4 lists the cycle counts of all operations.

	Char 8 bits	**Int 16 bits**	**Long 32 bits**
Assignment	3 or 0[a]	3 or 0[a]	6
Addition/subtraction	1	1	2
Or/and/xor	1	1	2
Extra for the use of a constant $\leq$ 0xff	1	1	-1
Extra for the use of a constant $\leq$ 0xffff	n/a	1	1
Extra for the use of a constant $\leq$ 0xffffff	n/a	n/a	2
Extra for the use of a constant $\leq$ 0xffffffff	n/a	n/a	2
Extra for the use of a global variable	2	2	4
Extra for a short, assignment-only instruction	-2	0	0
Multiplication	29	27	45
Short two-term expression	0	-2	0
Multiplication only expression	0	0	0
Parentheses	1	1	2
Pointer	0	0	0

Table A.4: Arithmetic on the MSP430

[a]See text.

The structure of the MSP430 CPU has much similarities with the CPU of the ATmega. Just like the ATmega, assigning to the first term of the expression is also cheaper than assigning to another variable: 0 cycles instead of 3.

Multiplying chars costs 2 cycles more because the compiler sees a char-char multiplication as an int-int multiplication with the most significant bytes zero and thus executes two extra instructions. Multiplying two longs is done by using the 16$\times$16 hardware multiplier four times, just like when multiplying two two-digit numbers by hand on paper.

Like with the ATmega, when there are too many local variables to fit in the registers, some variables need to be stored on the stack. For each use of these variables, 3 cycles extra are needed.

A.2.4 Bit shifting

A common operation when using fixed-point arithmetic is bit shifting. In fixed-point representation the amount of digits after the point is fixed to a predefined value. When for example a 16-bit datatype is used and the position of the point is defined as between the 12th and 13th digit (so there are 4 numbers after the point) the number can look as follows: 000000010100.0100. This number represents the value of $0 \times 2^{11} \ldots 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} = 20.25$.

Addition and subtraction of fixed-point numbers is exactly the same as with normal numbers. Multiplication is different: after the multiplication the number has to be shifted to the right with the amount of numbers after the point. Also to convert normal data to fixed-point data these numbers have to be shifted to the left by the amount of numbers after the point. So, bit shifting is a frequently used operation.

When shifting with a fixed amount the compiler usually tries to use a method that costs as little time as possible. This results in non-regular varying cycle times for the bit shifting operation. The summarized results are listed in Table A.5 and the exact results are listed in Tables A.6, A.7 and A.8.

Left shift	**Char**[a] **8 bits**	**Int**[b] **16 bits**	**Long**[b] **32 bits**
PIC18F	1 to 45	-1 to 114	9 to149
ATmega	1 to 3	2 to 48	4 to 124
MSP430	0 to 6	0 to 9	0 to 40
Right shift			
PIC18F	2 to 6	-1 to 144	9 to 183
ATmega	1 to 3	2 to 48	4 to 124
MSP430	1 to 5	0 to 7	0 to 40

Table A.5: Cycle times for bit shifting with a fixed amount

[a]Cycle counts are for a shift amount of 1 to 7 postions.
[b]Cycle counts are for a shift amount of 1 to 16 positions.

The figures in these tables show that the cycle count of a bit shift operation has a linear tendency, with lower values when the shift amount is a multiple of 8. Sometimes the values 'jump', for example in Table A.6 for leftshifting a char: shifting by 4 costs 3 cycles, shifting by 5 costs 41 cycles. This is because the compiler decides to no longer inline the commands for the bit shift but to execute a subroutine that handles the shifting.

Shifting with a variable argument is linear. Table A.9 lists the formulas that give the relation between the cycles and the shift amount for the various datatypes. In the formulas, n is the shift amount.

	Left shift			Right shift		
Shift amount	**Char**	**Int**	**Long**	**Char**	**Int**	**Long**
1	1	5	9	4	18	9
2	6	23	29	6	27	33
3	3	60	38	5	36	44
4	1	37	47	3	45	55
5	39	44	56	5	54	66
6	45	41	65	5	63	77
7	2	58	74	2	72	88
8		-1	80		-1	97
9		72	90		85	108
10		79	99		99	119
11		86	108		108	130
12		93	117		117	141
13		100	126		126	152
14		107	135		135	163
15		114	144		144	174
16		-2	149		144	183

Table A.6: Cycle times for bit shifting on the PIC18F

	Left shift			Right shift		
Shift amount	**Char**	**Int**	**Long**	**Char**	**Int**	**Long**
1	1	2	4	1	2	4
2	2	4	28	2	4	28
3	3	30	36	3	30	36
4	2	36	44	2	36	44
5	3	42	52	3	42	52
6	2	48	60	2	48	60
7	3	5	9	3	4	8
8	-1	2	4	-1	2	5
9		3	7		4	8
10		4	92		5	92
11		5	100		6	100
12		6	108		7	108
13		7	116		8	116
14		7	124		9	124
15		4	8		3	7
16		0	4		3	5

Table A.7: Cycle times for bit shifting on the ATmega

	Left shift			**Right shift**		
Shift amount	**Char**	**Int**	**Long**	**Char**	**Int**	**Long**
1	0	0	0	1	0	0
2	2	0	2	1	0	2
3	3	1	4	2	1	4
4	4	2	6	3	2	6
5	5	5	15	4	4	15
6	6	6	17	5	5	17
7	2	7	19	2	6	19
8	-2	2	4	-2	1	3
9		3	30		2	30
10		5	32		2	32
11		6	34		3	34
12		7	36		4	36
13		8	38		6	38
14		9	40		7	40
15		2	3		2	3
16		-2	0		2	2

Table A.8: Cycle times for bit shifting on the MSP430

Left shift	**Char 8 bits**	**Int 16 bits**	**Long 32 bits**
PIC18F	$12+7n$	$15+7n$	$23+9n$
ATmega	$15+6n$	$14+6n$	$16+8n$
MSP430	$12+7n$	$15+7n$	$23+9n$
Right shift			
PIC18F	$12+9n$	$15+9n$	$23+11n$
ATmega	$15+6n$	$14+6n$	$16+8n$
MSP430	$12+9n$	$15+9n$	$23+11n$

Table A.9: Cycle times for bit shifting with the variable amount n

A.3 Program flow

This section covers program flow related constructions in C. They are responsible for a significant share of the execution time of an algorithm. Program flow related constructions like function calls, if and loops constructs can sometimes take more time than real calculations of the algorithm. This section will cover function calls, interrupts, reading the AD-converter, boolean expressions in if statements and loop conditions, for and while loops.

A.3.1 Function calls

A function call usually consist of a *jump* or *branch* to a different code location and after execution a *return* back to the calling code. In both the calling code and the called code, the registers of the CPU could be used to contain local variables. These registers must be saved before executing the function code and restored afterwards. There are two frequently used schemes for this: caller-saves and callee-saves. In the caller-saves scheme, the calling code saves the registers that are still needed after the call and the called code may use all registers, in the callee-saves scheme the called code has to save the registers it wants to use. The caller-saves scheme usually requires fewer register saves and restores during run time but may require more instruction space [9, page 515]. This might be a reason why the IAR compilers use the callee saves scheme.

Time spent on a function call can be divided into 7 steps:

1. Preparing the arguments: as much as possible in registers, the rest on the stack.
2. Jumping to the function code
3. Saving the context: all registers that may not be harmed are pushed on the stack.
4. Execution of the function code.
5. Putting the return value in place.
6. Restoration of the context.
7. Returning to the calling code.

For counting the cycles, these steps are represented in numbers as three different numbers: one for the call and return (steps 2, 5 and 7), one for the arguments (step 1) and one for the context save and restore (step 3 and 6). These numbers are listed in Table A.10.

The following code illustrates the use of this table:

C code
```
int examplefunction(char c, int i, int j){

        //Some code with much local variables etc.

        return j;
}
int main() {
        int b;

        // some code

        b = examplefunction(a,1,g);

        // some more code

}
```

In this code `examplefunction` is a function that returns an int, has one char argument, two int arguments; that is 5 bytes or 3 words. One int argument is a constant. A full context save has to be done at the start of `examplefunction` because the function has many local variables. The rows that apply to this situation are: function call returning int, context save (r=16 for the ATmega and r=8 for the MSP430), 1 char argument and 2 int arguments for the ATmega and the MSP430, 5 bytes on the stack for the PIC18F and one 2-byte constant argument.

For the PIC18F this will cost $7+0+2\times5+0=17$ cycles for the call and 2 for the assignment (from Section A.2.1), so in total 19 cycles. On the ATmega context save takes much cycles: $10+(29+4\times16)+1+2\times1+2\times1=108$ and on the MSP430: $9+(15+2\times8)+1+2\times1+2\times1=42$ cycles. This shows that having only one working register can save much time on context saves. This does not mean that the PIC18F will be faster because the penalty of having only one working register can be larger than the benefits of

Function call	PIC18F	ATmega	MSP430
Returning void	5	7	7
Returning char	6	9[a]	9[a]
Returning int	7	10[a]	9[a]
Returning long	9	13[a]	11[a]
Extra when the return value is a constant	0	1/byte	1/word
Context save [b]	0	$29+4r$ [c]	$15+2r$ [d]
Arguments			
Per char argument	n/a	1	1
Per int argument	n/a	1	1
Per long argument	n/a	2	2
Max amount of arguments not on the stack	0	8 bytes	4 words
Costs for arguments on the stack	2/byte	2/byte	3/word
When an argument is a constant	0	1/byte	1/byte

Table A.10: Cycle times for function calls

[a]Including the assignment of the function result.

[b]r is the amount of registers to be saved

[c]$r=[1,16]$

[d]$r=[1,8]$

not having to save the context.

A.3.2 Interrupts

All microcontrollers have one or more interrupts. Dependent on the architecture of the microcontroller there is an interrupt table that contains (a jump instruction to the) addresses of the interrupt service routines (ISRs) or the microcontroller jumps to a fixed location that contains code that determines which peripheral caused an interrupt. The way this is implemented has impact on the time it costs to execute the ISR. Directly jumping to the right location is much faster than jumping to a routine that determines the interrupt cause and then executes the specific ISR.

An ISR is in principle no more than an ordinary function. The only thing is that all registers should be saved and restored, including registers that are normally not saved across calls.

The cycle counts for the execution and return of an ISR are on the PIC18F: 11 cycles, on the ATmega: 39 cycles + context save, and on the MSP430: 16 cycles + context save. The number of cycles that are required for the context save depend on the amount of local variables used in the interrupt service routine.

A.3.3 Reading the A/D converter

There are two methods of reading an A/D converter: start the conversion and wait until the conversion is finished by polling a finished bit in a register or enable an interrupt that executes an ISR when the execution is finished.

PIC18F

This can be the code when not using interrupts:

C code
```c
int adcread() {
  // Initalisation, values are for illustrative purposes
  ADCON1 = 0x12;
  ADCON0 = 0x34;
  ADCON2 = 0x34;
  ADON = 1;                 // Enable the A/D converter

  GO_DONE = 1;              // Start the conversion
  while (GO_DONE == 1);     // Wait until the conversion is done

  return *((int*)&ADRESL);  // Return the result
}
```

The line `return *((int*)&ADRESL);` is a trick to let the compiler return the registercombination ADRESH:ADRESL as int. This code was much faster than `return ADRESH << 8 | ADRESL;`.

The cycle count of this function depends on the time that `GO_DONE` is 1, which depends on the configured sampletime of the A/D converter. If the sample frequency for example is 1 kHz, then the execution time of this function would be $1/1000 = 100$ ms plus some cycles, which are negligible compared with the 100 ms spent, waiting for the conversion to finish. This can be done in the same way, with the same costs on the ATmega and MSP430.

When using the A/D converter interrupt the CPU can sleep to save power or do something else while waiting for the A/D converter. The following code for the PIC18F demonstrates the use of the interrupt for the A/D conversion:

C code

```c
void adcstart() {
  // Initialize and start the AD converter
  ADCON1 = 0x12;
  ADCON0 = 0x34;
  ADCON2 = 0x34;
  ADON = 1;
  ADIF = 0;
  ADIE = 1;
  GIEH = 1;
  GIEL = 1;
  GO_DONE = 1;
}
// adcvalue is volatile because it is changed by the interrupt code and used
// by non-interrupt code.
volatile int adcvalue;

#pragma vector=0x8
__interrupt void interrupt() {
  if (ADIF == 1) {  // the interrupt was an ADC interrupt
    adcvalue = *((int*)&ADRESL);  // store the value
    ADIF = 0;       // reset the ADC interrupt flag
    GO_DONE = 1;    // start a new conversion
    return;         // return normal execution
  } else {
    // other interrupt
  }
}
```

The PIC18F two interrupt vectors, one for high and one for low priority interrupts. The code in the ISR has to check what triggered the interrupt. This code takes 11 cycles however only when the interrupt-source check checks `ADIF` first. Reading `adcvalue` must be done with interrupts disabled, otherwise the interrupt could be triggered between reading the first a second byte, leading to wrong results. Enabling and disabling the interrupts takes 2 cycles in total. Therefore, the costs of reading a sample from the A/D converter takes $11 + 2 = 13$ cycles.

ATmega

The ATmega has a free running mode for the A/D converter: it keeps on sampling, the code does not have to start the A/D converter each time. In combination with the ability to enable the A/D converter interrupt code can be executed for each sample.

The ATmega has a protection mechanism that prevents an interrupt routine from corrupting the reading of the A/D converter result registers. When the low byte is read the high byte cannot be overwritten until it is read too.

The following code is an example of using the A/D converter in free running mode in combination with an ISR that stores the information:

C code

```c
 1  void adcstart() {
 2    /* turn off analogue comparator */
 3    ACSR |= (1 << ACD);
 4
 5    /* select PA0 */
 6    ADMUX = 0;
 7     * Set ADC prescaler to 128, enable ADC, and start conversion */
 8    ADCSRA = (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0) // divide clock by 128
 9           | (1 << ADEN)  // adc enable
10           | (1 << ADFR)  // free running
11           | (1 << ADSC)  // start now
12           | (0 << ADIE)  // interrupt disable for now
13           | (0 << ADIF); // clear interrupt flag
14
15    /* wait until bogus first conversion finished */
16    while (ADCSRA & (1 << ADSC));
17
18    // interrupt enable, clear flag
19    ADCSRA &= ~(1<< ADIF);
20    ADCSRA |= (1<< ADIE);
21  }
22
23  volatile int adcvalue;
24  #pragma vector=ADC_vect
25  __interrupt void adcinterrupt(void) {
26    // store value
27    adcvalue = ADC;
28    // clear interrupt flag
29    ADCSRA &= ~(1<< ADIF);
30  }
```

The ATmega has an interrupt vector table that contains jump instructions to the correct ISRs. When an interrupt occurs the CPU jumps to the specific location in the interrupt vector table and then to the ISR. This process and the ISR in the above code costs 48 cycles, where 28 cycles are for saving and restoring the context and only 9 for the code in lines 27 and 29.

When the algorithm does not depend on a fixed sample rate and missing a sample is no problem then it is enough to start the A/D converter in free running mode and just read the value of the `ADC` register pair. This will cost 2 cycles per sample.

MSP430

The MSP430's A/D converter has 4 sample modi: single channel single conversion, sequence of channels, repeat single channel and repeat sequence of channels. With this functionality, it is possible to let the microcontroller read multiple channels without interrupting the program execution. The values are stored in the `ADC12MEM0` to `ADCMEM15` registers. By setting bits in the `ADC12IE` register, an interrupt triggers when the specified `ADCMEMx` register is changed. In the ISR the `ADC12IFG` register indicates which channel triggered the interrupt. The interrupt flag is reset automatically when the matching `ADC12MEMx` register is read.

The interrupt is a normal MSP430 interrupt and takes 16 cycles to execute and return + the costs of code in the ISR.

A.3.4 Boolean expressions and if statements

The compiled version of an if statement consists of an evaluation of a boolean expression followed by a branch based on the result of the expression.

The following example illustrates a standard if statement and its representation in assembly code.

C code

```
1  if(b==a) {
2          <statements1>
3  } else {
4          <statements2>
5  }
6  <the rest of the code>
```

PIC18F assembly

```
1  0000A4   51 0B   MOVF     a,0,1
2  0000A6   19 0A   XORWF    b,0,1
3  0000A8   E1 03   BNZ      0xb0
4           <statements1>
5  ...      ...     ...      ...     ...
6  0000AE   D0 02   BRA      0xb4
7  0000B0   <statements2>
8  ...      ...     ...      ...     ...
9  0000B4   <the rest of the code>
```

In line 1 of the assembly code the value of `a` is loaded in `W`, in line 2 the value of `W` is xor-ed with the value of `b`, if that result is not zero (meaning that `a` and `b` are different), the `BNZ` instruction jumps to address 0x0b where the else part is executed. If `a` and `b` are equal the `BNZ` instruction is not executed, the BRA instruction is executed to skip the else part. In this case, the costs of the boolean expression are 3 cycles. If the expression evaluates to false the `BNZ` instruction will be executed and will cost 2 cycles instead of 1. The same goes for the ATmega and the MSP430, with slightly different cycle counts. Table A.11 gives an overview of all figures related to boolean expressions and the if statement.

To interpret this table some knowledge about evaluation of boolean expressions in C is required. In C it is defined that "expressions connected by `&&` or `||` are evaluated left to right, and it is guaranteed that evaluation will stop as soon as the truth or falsehood is known" [7]. Therefore, in the expression in the following example, if `c` contains the lowercase character

	PIC18F	**ATmega**	**MSP430**
Startup costs	3	2	0
Per evaluated comparison (==,! =,<,>,=>,=<) char or boolean	1	1	3
Per evaluated comparison (==,! =,<,>,=>,=<) int	2	6	3
Per evaluated comparison (==,! =,<,>,=>,=<) long	4	12	6
Comparison with a constant	0	char 0, int 1, long 3	1/byte
Expression evaluates to false	1	1	0
If statement evaluates to true and has an else part	2	2	2
Extra costs for a function call	-1	-1	0
Boolean and/or	1	1	and 3, or 5
Short two-term expression	-1	-2	0

Table A.11: Cycle times for boolean expressions and if statements

'a' then the result of this boolean equation will be true. The rest of the equation will not be evaluated, because it is already known that the result will be true. The table also contains 'extra costs' that are negative, for example when using a function call. This means that when a function call is used in a boolean expression on a PIC18F or an ATmega one cycle is saved with respect to normal function calls. The same goes for short two term expressions like `if (a==1)`: an expression with only two terms, `a` and `1` in this case, can be optimized by the compiler and that saves 1 or 2 cycles with respect to the normal situation.

C code
```
...
if (c == 'a' || c == somefunction() || c == c2) {
...
}
...
```

For evaluating the costs of a boolean expression the result of the expression should be known so that it is known what part will be evaluated and what not. So say that in the above example `c` is not equal to 'a' but is equal to the result of `somefunction`. In that case, the costs of the if statement are formed by the startup costs, two times an evaluated char comparison, one comparison with a constant, the costs of the function call, the extra costs for a function call and the costs of one boolean or. For the PIC18F the costs will be $3+2*1+0+$[costs of `somefunction`]$-1+1=5$ cycles plus the costs of `somefunction`, for the ATmega $2+2*1+0+$[costs of `somefunction`]$-1+1=4$ cycles plus the costs of `somefunction` and for the MSP430 $0+2*3+1+$[costs of `somefunction`]$+0+5=12$ cycles plus the cost of `somefunction`.

A.3.5 For and while loops

For and while loops can from the compilers viewpoint be seen as a combination of if statements and jumps. For example the following for loop and its representation with goto statements:

Original version

```
int i;
for (i=0;i<1000;i++){
// some code
}
```

Version with goto statements

```
int i;
i=0;


a:
if (i<1000) goto b;

// some code
i++;
goto a;
b:
```

A while loop works the same way, but does not have an initialisation and increment parameter.

The version with goto statements illustrates how the compiler converts the C code into assembly code for the microcontroller. So the costs of a for loop are made up of the initial `i=0` statement, the if statement each iteration, the increment each iteration and the goto statements.

The costs of the `i=0` and `i++` statements can be calculated according to Sections A.2.1, A.2.2 and A.2.3 and the costs of the if statements are calculated according to Section A.3.4. The costs of goto parts of the loop are 1 cycle for the PIC18F, 2 for the ATmega and 2 for the MSP430. The PIC18F also needs 2 cycles as startup costs.

A.4 Summary

This chapter described the derivation of the detailed models of the PIC18F, the ATmega and the MSP430. In Chapter 4, Table 4.1 lists all 73 operations and exceptions together with and their costs on the three platforms.