Automatic Synthesis of Application-specific Processors

CHARLES MUTIGWE

Thesis submitted in fulfilment of the requirements for the Degree

DOCTOR TECHNOLOGIAE: ENGINEERING: ELECTRICAL

in the

School of Electrical and Computer Systems Engineering
Faculty of Engineering and Information Technology

at the

Central University of Technology, Free State

Promoter: Prof. F. Aghdasi, Ph.D. (Eng.)
Co-promoter: Prof. J. Kinyua, Ph.D. (Eng.)

BLOEMFONTEIN
October, 2012

DECLARATION WITH REGARD TO INDEPENDENT WORK

I, CHARLES MUTIGWE, identity number [redacted] and student number 207072035, do hereby declare that this research project submitted to the Central University of Technology, Free State for the Degree DOCTOR TECHNOLOGIAE: ENGINEERING: ELECTRICAL, is my own independent work; and complies with the Code of Academic Integrity, as well as other relevant policies, procedures, rules and regulations of the Central University of Technology, Free State; and has not been submitted before to any institution by myself of any other person in fulfilment (or partial fulfilment) of the requirements for the attainment of any qualification.

SIGNATURE OF STUDENT	DATE

I would like to dedicate this thesis to my parents and my grandmother who, by word and example, have inspired me to never stop learning and to always strive for academic excellence.

Acknowledgements

I would like express my deepest thanks to my advisors, Professor Farhad Aghdasi and Professor Johnson Kinyua, for their support, guidance and understanding during my doctoral studies. I thank them for giving me this wonderful opportunity to pursue the idea of automatic generation of processors and for agreeing to continue advising me after they both left CUT. I would like to thank Prof. Aghdasi for helping me secure funding for the first 3 years of my studies.

As a part-time student not based in South Africa or in my home country, Zimbabwe, the administrative requirements associated with continued enrollment at CUT were many and quite demanding. In this regard, I am especially grateful to my brother, Bruce, who is based in Bulawayo, Zimbabwe, for the many trips to Harare that he made on my behalf, and for all the other necessary administrative documents he helped me to secure. He was very supportive of my studies in other ways, including as a 'bouncing board' for some of my ideas. I would also like to thank the many dedicated administrative staff at CUT who helped me successfully navigate the administrative maze.

Above all, I am grateful to my wife, Sabely, my daughter, Chelsea and my son, Simbarashe who have all patiently put up with and supported me through the seemingly endless years spent on this research project.

Summary

This thesis describes a method for the automatic generation of application specific processors. The thesis was organized into three separate but interrelated studies, which together provide: a justification for the method used, a theory that supports the method, and a software application that realizes the method. The first study looked at how modern day microprocessors utilize their hardware resources and it proposed a metric, called core density, for measuring the utilization rate. The core density is a function of the microprocessor's instruction set and the application scheduled to run on that microprocessor. This study concluded that modern day microprocessors use their resources very inefficiently and proposed the use of subset processors to execute the same applications more efficiently. The second study sought to provide a theoretical framework for the use of subset processors by developing a generic formal model of computer architecture. To demonstrate the model's versatility, it was used to describe a number of computer architecture components and entire computing systems. The third study describes the development of a set of software tools that enable the automatic generation of application specific processors. The FiT toolkit automatically generates a unique Hardware Description Language (HDL) description of a processor based on an application binary file and a parameterizable template of a generic microprocessor. Area-optimized and performance-optimized custom soft processors were generated using the FiT toolkit and the utilization of the hardware resources by the custom soft processors was characterized. The FiT toolkit was combined with an ANSI C compiler and a third-party tool for programming field-programmable gate arrays (FPGAs) to create an unconstrained C-to-silicon compiler.

Opsomming

Hierdie tesis beskryf 'n metode vir die outomatiese generasie van die aansoek spesifieke verwerkers. Die tesis is georganiseer in drie afsonderlike maar verwante studies, wat saam verskaf: 'n regverdiging vir die metode wat gebruik word om 'n teorie wat die metode ondersteun, en 'n sagteware program wat die metode besef. Die eerste studie het gekyk na hoe die hedendaagse mikroverwerkers gebruik hulle hardeware hulpbronne en dit 'n statistiek, die sogenaamde kern digtheid voorgestel, vir die meet van die benutting koers. Die kern digtheid is 'n funksie van die mikroverwerker se opdrag stel en die aansoek wat geskeduleer is om uit te voer op daardie mikroverwerker. Hierdie studie het tot die gevolgtrekking gekom dat die moderne dag mikroverwerkers gebruik om hul hulpbronne baie ondoeltreffend en voorgestel dat die gebruik van die subset verwerkers dieselfde toepassings meer doeltreffend uit te voer. Die tweede studie het probeer om 'n teoretiese raamwerk vir die gebruik van die subset verwerkers deur die ontwikkeling van 'n generiese formele model van die rekenaar argitektuur te verskaf. Die model se veelsydigheid te demonstreer, is dit gebruik om 'n aantal van rekenaarargitektuur komponente en die hele rekenaar stelsels te beskryf. Die derde studie beskryf die ontwikkeling van 'n stel van sagteware gereedskap wat in staat stel om die outomatiese generasie van die aansoek spesifieke verwerkers. Die Fit toolkit genereer outomaties 'n unieke Hardware Beskrywing Taal (HDL) beskrywing van 'n verwerker wat gebaseer is op 'n aansoek binre ler en 'n parameterizable sjabloon van 'n generiese mikroverwerker. Area-new en prestasie-optimale persoonlike sagte verwerkers is gegenereer deur gebruik te maak van die Fit toolkit en die benutting van die hardeware hulpbronne deur die persoonlike sagte verwerkers is gekenmerk. Die Fit toolkit is gekombineer met 'n ANSI C compiler en 1/3-party hulpmiddel vir die programmering van veldprogrammeerbare hek skikkings (FPGAs) te skep van 'n onbeperkte C-tot-silikon samesteller.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

As the number of transistors on a silicon chip continue increase exponentially, as predicted by Moore's Law, the design of electronic systems is moving away from the situation where discrete chips were assembled on a circuit board to one where entire systems or networks of systems are placed on a single chip. The design of these systems-on-chip (SoCs) and networks-on-chip (NoCs) demands new ways of viewing the electronic systems design process and new design tools. The research described in this work focuses on one aspect of designing SoCs and that is the automated design systems based on one class of microprocessors.

We believe that the work presented in this thesis contributes a small, but important, step in facilitating the paradigm shift that is required to design the SoCs of the future. This work not only provides a reference implementation and a theoretical framework for some new designs, but it also shows the shortcomings of existing microprocessor implementations.

In this chapter we begin by looking at the problems that the research presented in this thesis attempts to answer. Using some scenarios, we highlight real computer engineering issues that motivated our research. This is followed by a description of the objectives of the research. Next, the research methodologies that were used are outlined and then a description of the research's limitations is given. The chapter concludes with a summary of how the rest of the thesis is organized.

1.2 Problem Description

The basic question addressed by this work is, *how can the properties of an application be used to automatically generate an optimized custom processor, which will later be used to execute the application?* Application properties, in this context, means the assembly instructions used by the application - their usage frequencies, their length in bits and their cycle times. Optimized, in this context, means that for an application set to run on the processor, each resource in the processor is scheduled at least once for the successful execution of the application. In other words, there are no unused resources in the processor. The microprocessor generating system developed to answer this question is easily extended to synthesize microcontrollers, and this leads to a more general question related to the program-store von Neumann model [1]: *can a generic program-architecture-store model be developed to represent these microcontrollers that have a one-to-one relation between the application and the processor?*

Other sub-questions arising from these, which are addressed as part of this work, are:

- *How are processors generated automatically?*
- *How efficiently do applications use the resources on traditional microprocessors?*
- *How efficiently do applications use the resources on the microprocessors that are generated as part of this work?*
- *Can the processor generating system developed be extended into an unconstrained C-to-silicon compiler?*

Any effective answer to the last sub-question is of much interest to the electronic design automation (EDA) community [2].

1.3 Motivation for the Research

Consider the following scenarios, which are typical of those that have motivated this research.

- **Platform Vendor versus App Developers:** A device platform vendor may want to provide some performance guarantees for applications hosted on the device, while at the same time improving the platform's popularity by courting as many third-party developers to write apps for the platform. The third-party developers may each optimize the performance and functionality of their applications without regard to any other applications that may be co-located on the device. If the device is implemented with a limited number of microprocessors and uses context-switching-based multitasking, then as the number of applications hosted on the devices rises, clearly the performance guarantees are in jeopardy. For example, early releases of the iPhone did not permit any apps to run in the background, causing outcries from the developer community [3]. iPhone multitasking was introduced with OS 4.0. However, even with this release, applications are categorized and then the allocation of computing resources to these categories is prioritized [4].

- **Application Isolation:** According to some estimates, 80 percent of automobile innovations come from computer systems [5]. Premium-class automobiles now use around 100 million lines of code, and that number is expected to reach 300 million in the near future [6]. As the amount of software increases, so does the complexity of the systems. Complexity leads to reliability concerns. The challenge is to continue to increase the functionality by adding new software applications (i.e. more lines of code), while improving reliability by minimizing the potential for unwanted interactions between these applications.

- **Legacy Application Support:** The obsolescence of technology is one of the primary drivers for modernizing legacy information systems [7]. However, according to Sneed [8], more than 50 percent of these modernization projects fail; he recommends an upgrade method that minimizes the risks of failure. The proposed method leaves the data and the business processes unchanged, and focuses on updating the technology components piecemeal-wise.

- **Time to Market:** Raising the level of programming abstraction improves the productivity and performance of the applications developers [9, 10] by allowing developers to more easily describe what a desired application should do, instead of having them focus on the details of how it should do it. The productivity of the product developers is important since, according to Cohen *et al.* [11], many technology-driven firms now compete on product-development cycle time.

The two common themes running through the examples are: complexity, and features. Each example involves the relationship between a computer system's complexity and its supported feature set. The features are encapsulated in software applications. As more applications are scheduled to run on a computer system, the complexity of the system increases due to (1) the interaction among the applications and (2) the varying demands made by each application to the shared computing resources. One way, which is the focus of this work, to manage the latter cause of computer system complexity is to try and guarantee computing resources for each application. The main resource in most modern computer systems is the microprocessor, so the general direction of this work is to show how we can transparently allocate to each application a dedicated microprocessor whose resources are tailored to the application.

1.4 Research Objectives

The objectives of our research were to:

1. Study how existing microprocessors utilize their hardware resources when running software applications. The results of this study should help identify any inefficiencies or missing features that require further research.

2. Create and evaluate a tool to automatically generate custom microprocessors. This tool is a new, practically useful system to perform tasks that previously could not be achieved.
3. Develop a formalism to model computer architecture. The practical applications of the computer architecture models are that they enable us to better understand, control and modify our custom microprocessors.

1.5 Research Methodology

In this section we use the terminology by Amaral *et al.* [12] to describe the research methods used in this work.

The *experimental methodology* was used to measure how existing microprocessors utilize their hardware resources as a function of the microprocessors' instruction sets. Industry standard benchmarks from the Standard Performance Evaluation Corp. [13, 14] were used to provide the source code. The most popular open source compilers were used to compile the source code and to profile compiler outputs.

The *model methodology* was used to develop the 'instructions-resources-data triplet model, which is an abstract model of real and virtual computer architectures. The triplet model was constructed from first principles as a recursive, set-theoretic model. The triplet model was used to describe custom microprocessors, which were the primary focus of this research. However, the triplet model is generic enough and is able to model general computer components and general computing systems, as shown by the examples that were given in this work.

The *build methodology* was used to build, in the C programming language, a software application that demonstrated the feasibility of automatically generating custom microprocessors. We called this software application the FiT toolkit. The automatically generated custom microprocessors were then implemented on a reconfigurable computing device. The utilization of hardware resources by the automatically generated custom microprocessors was then studied using the *experimental methodology*. Third-party tools provided by Xilinx Corp. [15], the vendor of the reconfigurable computing device, were used to study the generated custom microprocessors.

1.6 Limitations of the Research

While the method of generating custom microprocessors presented in this work raises the abstraction level and so improves the productivity of system designers. The quality of the custom microprocessors that are produced may depend on the quality of the executable file that is generated by the compiler. We tested for this dependence using two popular optimizing compilers and found it not to exist, however in order to be more conclusive, tests using a more diverse set of compilers are needed.

The example applications of the formal model and the FiT toolkit that were both developed as part of this research target the k85 microprocessor. The k85 is a very simple microprocessor that is code-compatible with the first popular microprocessors, the Intel 8080 and 8085. This research needs to be extended to target modern microprocessors, which are much more complex than the k85.

1.7 Thesis Organization

The rest of this thesis is organized around three self-contained studies. The studies are related and they are arranged so as to build upon one another.

Chapter 2 reviews previous work on instruction set usage, formal models of computer architecture, and subset processors.

Chapter 3 presents the first study in which the core density metric is developed and compared to another instruction set usage measure. The study then describes the instruction set usage experiments. The results of the experiments are also discussed and compared to instruction set usage results from other researchers.

We propose the use of subset processors in order to generate application-specific systems that have an optimal core density.

Chapter 4 presents the second study where a generic computer architecture model is developed. We give a number of examples to demonstrate how the model can be used to describe computer components as well as complete computer systems. In this study the computer architecture model is used to describe the subset processors that were proposed in the previous study.

Chapter 5 presents the third study, which describes the FiT toolchain. FiT is an application-specific processor generator that accepts an application binary file as an input, and automatically creates an HDL specification for an 8-bit subset processor. The HDL processor specifications are then realized on an FPGA development board. Two approaches to generating the sub-set processors are presented and compared as to their hardware resource utilization. In the study, we use some scripts to combine the FiT toolchain, an ANSI C compiler from the Amsterdam Compiler Kit and the Xilinx ISE WebPACK to create a C-to-silicon compiler. This C-to-silicon compiler takes an application described in unconstrained ANSI C and automatically implements the application in hardware as a microcontroller that is driven by an 8-bit subset processor.

Chapter 6 summarizes the findings of this research, highlights the contributions of this research and presents some recommendations for future research work.

Chapter 2

Related Work

2.1 Introduction

The research presented in this work is focused in the area of customizable processors, also referred to as application-specific instruction processors (ASIPs). Several researchers have described ASIPs as the next evolutionary step for microprocessors [16, 17]. The general research approaches in this area have been to construct one or more of the following [18]: (i) Parameterizable processors (ii) Extensible processors (iii) Custom processor development tools. As part of this research, a custom processor development tool for parameterizable processors was created.

In this chapter we review earlier work that laid the foundation for our research on a particular group of ASIPs called subset processors. We also note any gaps in the earlier work that have directed our research.

2.2 Instruction Set Usage

Foster *et al.* [19] describe two types of instruction set usage analyses. In the first type, the static case, the frequency counts of the instructions used to specify the logic of the problem are collected and analyzed. In the second type, the dynamic case, the frequency counts of the instructions used to execute the logic of the problem are collected and analyzed [20]. Foster *et al.* [19] proposed two measures for instruction usage.

The first measure is based on information theory and it calculates the average number of bits of information contained in each opcode for each application that is analyzed. It states that if there are T instructions and p_i is the probability that the i-th instruction is used, then the average number of bits of information contained in each instruction, I, is

$$I = -\sum_{i=1}^{T} p_i(log_2 p_i). \tag{2.1}$$

I_{max} the maximum value of I occurs when the usage of any instruction is equally probable and it is

$$I_{max} = log_2 T. \tag{2.2}$$

The relative difference between I and I_{max} is a measure of the utilization of any particular set of instructions.

The second measure estimates the effort needed to recode an application when the number of opcodes available to the compiler or assembly programmer is reduced to the N most popular ones after an initial unconstrained compilation. To arrive at this measure the instructions used in the application are ordered from the most frequently used to the least frequently used. Let C_k equals the number of times the k-th instruction was used where $C_k \geqslant C_{k+1}$ for all k and let P equal the total number of instructions in the original application. The fraction of all instruction occurrences included in the set N is given by the function $f(N)$ where

$$f(N) = \frac{1}{P}\sum_{k=1}^{N} C_k \tag{2.3}$$

and $g(N)$ defined as:

$$g(N) = 1 - f(N) \tag{2.4}$$

is then a measure of the effort needed to recode the original application onto a machine that only uses the top N opcodes. Both of the measure mentioned above applied to static and dynamic instruction usage. Foster *et al.* also found that the hand-assembled code has higher static opcode usage than machine-compiled code, and that there are no significant differences in dynamic opcode usage between the hand-assembled and machine-compiled code.

Hennessy and Patterson [21, 22] conducted instruction set usage experiments on several Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC) ISAs using the SPEC CPU92 benchmark applications. They found that on average 90% of the instruction execution comes from 10% of the instructions in the integer programs and 14% of the instructions in the floating-point programs. Fig. 2.1 (sourced from [21]) shows the percentage of instructions that are responsible for 80% and for 90% of the instruction executions. The total bar height indicates the fractions of instructions that account for the 90% of the instruction executions and the dark portion indicates the fraction of instructions responsible for the 80% of the instruction executions. For the x86 architecture Hennessy and Patterson [22] found that the top 10 instructions were responsible for 96% of the instructions that were executed as shown in Table 2.1. These 10

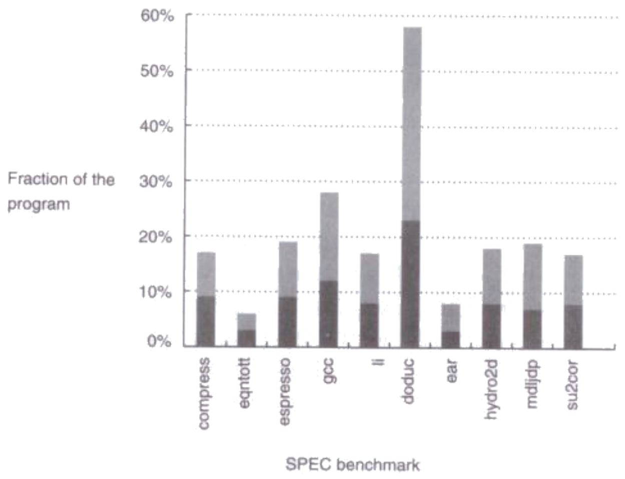


Figure 2.1: Instructions (%) responsible for 80% and 90% of instruction executions.

Table 2.1: Top 10 instructions for the x86.

Rank	**80x86 instruction**	**Integer average (% total executed)**
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	**96%**

instructions represent 25 opcodes in the x86 ISA. Using the TI TMS320C540x Digital Signal Processor (DSP), Hennessy and Patterson [22] found that the top 20 instructions account for 97.2% of all the instructions executed as shown in Table 2.2.

Other empirical studies by Adams and Zimmerman [20], and Huang and Peng [23] on the x86 instruction set architecture (ISA); and El-Kharashi *et al.* [24] on the Java Virtual Machine (JVM) have also shown that modern applications spend 80-90% of their time accessing only 10-20% of the ISA.

2.3 Formal Models of Computer Architecture

The term *computer architecture* was first used to describe the attributes of the IBM System/360 as seen by the programmer [25, 26]. Today this aspect of a computer's design is commonly known as its Instruction Set Architecture (ISA). Over time, the concept of computer architecture has grown to be more encompassing. Mudge [26] defines computer architecture as the ISA together with its implementation using hardware components. He adds that computer architecture influences and is influenced by the existing technology, the applications targeted

Table 2.2: Mix of instructions for TMS320C540x DSP.

Instruction	**Percent**
store mem16	32.2%
load mem16	9.4%
add mem16	6.8%
call	5.0%
push mem16	5.0%
subtract mem16	4.9%
multiple-accumulate (MAC) mem16	4.6%
move mem-mem 16	4.0%
change status	3.7%
pop mem16	2.8%
conditional branch	2.6%
load mem32	2.5%
return	2.5%
store mem32	2.0%
branch	2.0%
repeat	2.0%
multiply	1.8%
NOP	1.5%
add mem32	1.3%
subtract mem32	0.9%
Total	**97.2%**

to run on the computer, and other constraints such as costs, compatibility and the marketplace. Hennessy and Patterson [27] define computer architecture as the design specifications for a computer, which include the description of its: (i) ISA, (ii) microarchitecture, also known as computer organization, and (iii) hardware. These design specifications (or blueprints), when implemented, should result in a computer that maximizes performance while subject to constraints, such as costs and power. In this work we used the definition of computer architecture by Hennessy and Patterson. When dealing with physical machines, the 'program' and 'data' components of our proposed model relate to the ISA part of this definition, while the 'resources' part of our model relates to the microarchitecture and the hardware descriptions.

A constructive computation-based theoretical framework for modeling the underlying structures of computer architecture is presented by Albrecht [28]. While this framework is generic, it has some limitations in that it is not intuitive and it is mainly focused on modeling the operations of the components. Furthermore, it is only accessible to computer architects with advanced mathematical training in formal models.

Within the literature, the architecture of physical computers and the architecture of virtual computers are treated as a separate subjects [27, 29]. Given the growing importance of virtualization in the computer industry, we are of the view that a framework which seamlessly handles both physical and virtual computer architectures will be advantageous. Chen *et al.* [30] proposed a Virtual Machine (VM) model that extends an existing model that is used for real machines. They model both the source computer system and the destination computer system (virtual machine) as Turing Machines, M^S and M^T, such that

$$M^S = \left(S^s, I^S, \delta^S, {s_o}^S, {S_f}^S\right) \tag{2.5}$$

where

- S^S denotes the set of all the states the source machine (computer system) may have,
- I^S denotes the set of all the instructions the source machine can provide,

- $\delta^S : S \times I \mapsto S \times I$ is the execution operator of an instruction of I,
- ${s_o}^S$ denotes the initial state of a machine, and
- ${S_f}^S$ denotes the set of all the possible final states of a machine.

For M^T, simply change all the superscripts in Equation (2.5) from 'S' to 'T'. To handle the emulation they extend the execution operator to

$$\delta : S \times \{I\} \mapsto S \times I. \tag{2.6}$$

This results in a state machine-based model that does not easily lend itself to modeling the architectures such as those of application-specific processor cores on a reconfigurable computing fabric, since the set of all states may be too large to easily display and keep track of.

Smith and Nair [29] model virtual machines graphically. Fig. 2.2 shows an example of one of their models for a system VM that supports multiple Operating System (OS) environments on the same hardware. Such models are most useful when working with a small set of virtual machines or when the level of virtualization is low. These graphical models cannot be used to analyze virtual machines algebraically or recursively.

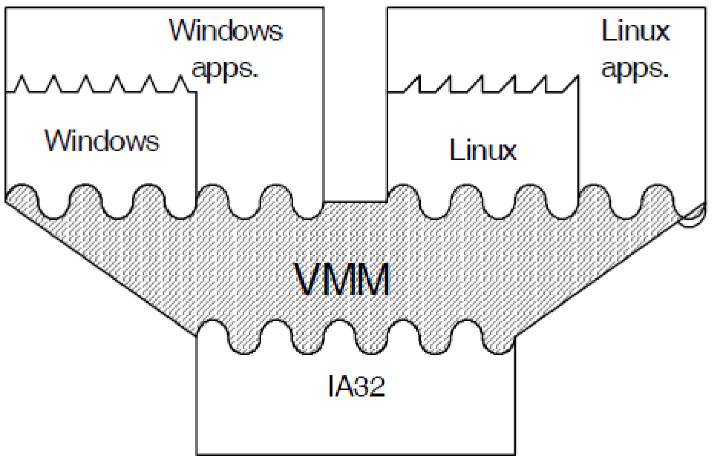


Figure 2.2: A VM that supports multiple OS environments.

Reconfigurable computing refers to “systems incorporating some form of (run-time) hardware programmability, - customizing how the hardware is used using a number of physical control points. These control points can then be changed periodically in order to execute different applications using the same hardware” [31]. We are looking for a computer architecture model that is generic enough to take into account the possible use of reconfigurable computing resources. Sima *et al.* [32] put forward an architectural-based taxonomy for field-programmable devices. Their taxonomy introduced a recursive formalism (similar to Flynn’s requestor/server formalism [33]) that is based on microcode in order to abstract away any references to a particular ISA. This formalism defines a computing machine (CM) as a doublet, consisting of a microprogram (μP), and a set of resources (R). That is

$$CM = (\mu P, R). \tag{2.7}$$

Our proposed computer architecture model extends this formalism in Equation (2.7) by adding a third component that will be used to model the data processed by the computing machine. This data component will facilitate the modeling of virtual devices and operations on inputs with different data types.

2.4 Soft Processors

A *soft processor* is a hardware description language (HDL) model of a microprocessor core that is implemented on a reconfigurable device. In a soft processor a HDL, such as VHDL or Verilog, is used to describe the structure and behavior of the microprocessor. Vendor-specific logic synthesis and place & route tools are then used to implement HDL model on a reconfigurable device, such as an FPGA. Most of the complex soft processors that have been implemented using VHDL and Verilog models have been based on the RISC architecture. In a RISC ISA, the reduced number of instructions and their fixed format lend themselves to a simpler design for the microprocessors control unit [34, 35]. One of the earliest complete VHDL models of a commercial processor was LEON, which modeled the SPARC V8 microprocessor [36]. The LEON required 5,300 look-up tables

(LUTs) to implement on a Xilinx XCV300E-8 FPGA. Other RISC-based VHDL models of complete 32-bit and 64-bit microprocessors continue to be developed and used as research tools [37, 38]. Lu *et al.* developed a complete VHDL model of the CISC-based Pentium microprocessor, which was implemented on an FPGA and used to replace the original microprocessor on a motherboard [39]. Lu *et al.* were able to install and run Windows XP and other applications on this soft processor-driven computer system. However, given the complexity of the x86 architecture, 65,612 LUTs were required to implement the Pentium VHDL model on a Xilinx Virtex-4 FPGA. In all the cases above the resources required to implement each soft processor, as measured by the number of LUTs used, are fixed. However, as we will show in Chapter 3 most of these resources will not be used by the applications running on the soft processors and so in order to more efficiently utilize the limited resources on the reconfigurable devices any unused resources should not be implemented.

2.5 Subset Processors

The study of subset processors is part of the broader study of ASIPs. Jain *et al.* [40], in their overview of the issues and techniques in ASIP design note that the techniques for instruction set generation fall into one of two categories: instruction set synthesis or instruction selection from a pre-existing superset of instructions. Our FiT toolkit uses the instruction selection technique. Soft processors have size, performance and power disadvantages when compared to their packaged counterparts. Bilski *et al.* [41] discuss some of ways that these shortcomings were mitigated in the design of the Xilinx MicroBlaze soft processor. They did this as follows:

- Limited the number of pipe stages to three, because adding more pipe stages increased the number and size of the multiplexors in the processor.
- Matched the supported number logic instructions in the MicroBlaze's instruction set to the input size of the Look-Up Tables (LUT), which in the case of the Xilinx devices was 4.

- Implemented all timing-critical functions using a carry chain in order to reduce logic delays.
- Used the shift register mode of the LUT to implement First-In First-Out (FIFO) and instruction fetch buffers, whenever possible.

As a result of their optimizations the MicroBlaze has a fixed floor plan and any customizations, such as those proposed in Chapter 3, would disturb the optimal floor plan. Customization in the MicroBlaze is achieved by adding or removing carefully selected pre-configured modules, such as the floating-point unit. These result in predetermined optimal fixed floor plans.

Other soft processors, such as the NIOS II from Altera [42], support extending the instruction set by allowing users to create custom instructions. FiT does not use a fixed floor plan scheme. To facilitate portability, the HDL model of our custom soft processors does not take advantage of any vendor-dependent optimizations or pre-compiled synthesis libraries. As will be discussed in Chapter 3, we do not use instruction set extensions, as these increase the core density and require that the source code or the tools that compile the source code be updated to handle the new instructions.

Bush [43], in his Ph.D. thesis, proposed the synthesis of subset processors. However, since this was incidental to his primary research focus, he did not address the theory and applications of subset processors. However, more recently Yiannacouras *et al.* [44] have taken a more substantive stab at the subject; they have proposed a method similar to ours which they referred to as ISA subsetting, together with an application called the SPREE toolkit that generated subset processors. Unlike FiT, SPREE only uses a single structural processor template, similar to the one described in Section 5.2.1 and it does not have an automated end-to-end flow using either the source code or the object code as input to generate the custom soft processor. Yiannacouras *et al.* [44], also did not provide an overarching theoretical framework for subset processors.

2.6 Summary

Earlier research on instruction set usage showed low instruction set utilization, however this research failed to show the relationship between instruction set utilization and microprocessor hardware utilization. While, previous work on subset processors lacked a theoretical framework, which made the analysis and comparison of subset processors difficult. To overcome these problems this work established the relationship between instruction set utilization and microprocessor hardware utilization and proposed a way to measure the relationship. Our work, also developed a theoretical framework for subset processors. The FiT toolkit was created to automatically synthesize custom subset processors that are based on our theoretical framework. These custom subset processors optimize the hardware utilization.

Chapter 3

Instruction Set Usage Analysis

3.1 Introduction

The instruction set of a processor serves as an interface between the processor's hardware and the software applications seeking to run on that hardware. To the software engineer, the instruction set exposes a processor's functionality, while to the hardware engineer it is a measure of the hardware resources that will need to be implemented in the processor. Given two processors with the same bus size, but with different architectures, the one with a larger instruction set will expose more features or operations to the software applications. In addition, it will often require more hardware resources (as will be shown later in this study) and design effort to implement.

The software applications that execute on a processor are created by compiling source code that is written in a high-level programming language, such as C or Java, or they are hand-crafted using assembly instructions and an assembler. In both cases, the end result is an object file consisting of an ordered sequence of operations that are to be performed by the processor.

A number of empirical studies on instruction set usage, including the one presented in this work, have shown that most of the instructions in any instruction set are rarely used by the applications [19, 24, 20, 23]. If a processor's instruction set is a measure of the hardware resources needed to implement the processor, then these results from the empirical studies suggest that most of the hardware resources on processors with a fixed instruction set architecture (ISA) are highly underutilized.

In this chapter we discuss the relationship between a processors instruction set and the hardware resources needed to support that instruction set, and define the *core density* measure. We also describe our instruction set usage experiments, present the results of these experiments and discuss their implications.

3.2 Instruction Sets and Core Density

3.2.1 Introduction

In this study we attempt to make a direct connection between instruction set usage and processor hardware requirements. We conducted instruction set usage experiments across a variety of existing ISAs using the SPEC CPU2006 [13] and SPECjvm2008 benchmark [14] applications written in C and Java. For the C applications, we used two compilers and it took into account different compiler optimizations. The results from these experiments confirm those of previous

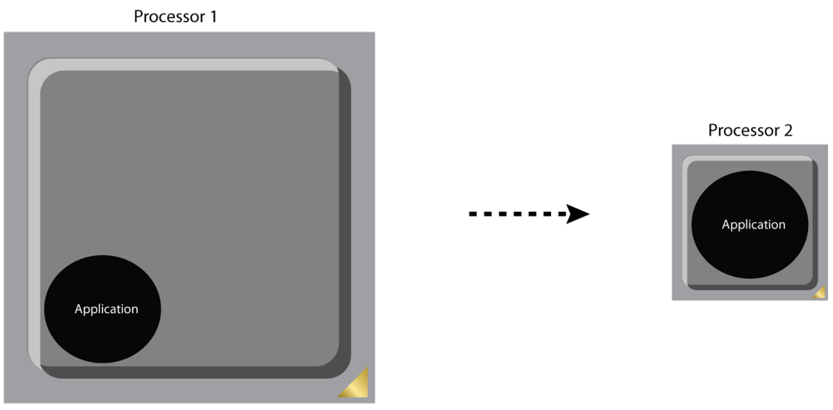


Figure 3.1: Transforming a fixed ISA processor into an exact ISA processor.

studies. We describe a way to measure the resource underutilization using a single instruction set usage measure, *core density*, in place of the two measures by Foster *et al.* We then propose a more efficient method of allocating hardware resources to support a given instruction set. The proposed technique generates application-specific processors that we term *exact processors*, where each processor's resources are mapped one-to-one to the processing needs of the applications that are set to execute on the processor, as shown by Processor 2 in Fig. 3.1.

3.2.2 Instruction Sets and Hardware

A processor instruction is a directive to the processor that specifies the following: the instruction format; the operation to be performed, also known as the opcode; the source operands; the result operand; and the next instruction to be executed [45]. The instruction layout is illustrated in Fig. 3.2.

The instruction format is often expressed implicitly. The opcode encodes the type of transformation that is to be performed. There are four basic types of transformations for discrete information [46]:

- data transfer in space (for example, from one register to another),
- data transfer in time (storage),
- arithmetic operations, and
- logical operations.

The addresses of the source operands, result operand, and the next instruction may be expressed in several ways: by coordinate addresses, by implication, as immediate variables, or by association [45]. From now onwards, the terms instruction and opcode will be used interchangeably.

Let R be all of the hardware resources of the processor represented as functional units and I be the *processor instruction set*, where

$$R = \{r_1, \ldots, r_M\} \quad \text{and} \quad I = \{i_1, \ldots, r_N\}. \tag{3.1}$$

Let P be the power set of R. For any i where $i \in I$, let R_i represent the set of resources needed to implement the instruction where

$$R_i \in P \quad \text{and} \quad R_i \subset R. \tag{3.2}$$

Let R_I represent the resources needed to implement all the instructions in I, then

$$R_I = \bigcup_{i=1}^{N} R_i. \tag{3.3}$$

Resources that are not directly related to the implementation of any instruction will be referred to as a constant M_{const}, where

$$M_{const} = |R - R_I|. \tag{3.4}$$

One example of such a resource is the hardware needed to implement pipelining.

As the number of shared resources decreases, that is as

$$\left| \bigcap_{i=1}^{N} R_i \right| \to 0, \tag{3.5}$$

the processor's performance improves. An example of this relationship is the improved performance of directly-implemented processors versus their microprogrammed versions.

With regards to hardware requirements, in directly-implemented processors there is a one-to-one function f, such that $f : I \mapsto P$. As the number of instructions in I increases, so does $|R_I|$.

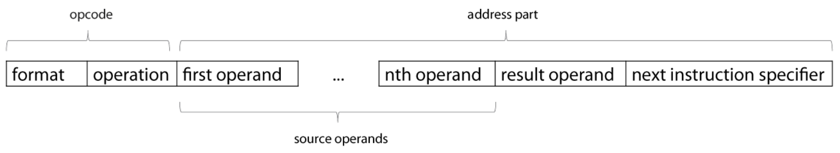


Figure 3.2: The Instruction Vector.

In the case of microprogrammed processors, f is not one-to-one; several instructions may map onto one set of resources. However, each instruction represents a distinct operation, and this distinction is captured in the microinstructions used to describe it. All the sets of microinstructions and the lookup tables matching them to their corresponding instructions are stored in the control memory. Wisniewska *et al.* [47] found that as the number of microinstructions was increased by a factor of 3, the amount of control memory required to store them increased by a multiple of 54, suggesting an exponential growth in hardware requirements. Jian-Lun [48] notes that for some processors (including the Intel x86), the control memory takes up 50% of the area on the chip.

We conclude that regardless of whether a processor is directly-implemented or it is microprogrammed,

$$|I| \propto |R_I|. \tag{3.6}$$

3.2.3 Instruction Subsets

Applications that execute on a processor use a set of instructions, which we will refer to as the *application instruction set*. Let A represent this set. The application instruction set is a subset of the processor instruction set, that is

$$A \subset I. \tag{3.7}$$

The number of possible application instruction sets, N, is

$$N = 2^{|I|}. \tag{3.8}$$

The resources required to implement R_A the application-specific processor for A are,

$$R_A = \bigcup_{i=1}^{|A|} R_i \tag{3.9}$$

where,

$$R_A \subset R_I. \tag{3.10}$$

From Equation (3.7), we have

$$|A| \leqslant |I| \tag{3.11}$$

and from Equation (3.10), we have

$$|R_A| \leqslant |R_I|. \tag{3.12}$$

The results of the experiments presented below show that for the benchmark applications,

$$A \ll I. \tag{3.13}$$

Now using Equations (3.6), (3.12) and (3.13) we have

$$|R_A| \ll |R_I|. \tag{3.14}$$

That is, the average application-specific processor requires much less hardware resources than the processor that implements the complete instruction set. The next Section describes a measure that can be used to compare the resources between the two processors in Equation (3.14).

We will refer to the processor that implements the complete processor instruction set as a *general-purpose core* (**GPC**), while the processor that implements only the application instruction sets that will target it, we will call an *exact processor core* (**EPC**). There is another type of core that allows for run-time extensions to the GPC, which we will refer to as the *extensible processor core* (**XPC**). The Xtensa processor from Tensilica provides an example of an XPC [49]. Since, in every case, the $GPC \subseteq XPC$, the ISA utilization of the XPC is at best only equal to that of the GPC. Therefore, we will only address the relationship between the GPC and the EPC in this study.

3.2.4 Core Density

Given a set of applications that have been compiled to run on a given ISA, we define the applications' core density (η) as

$$\eta = \frac{\text{Amount of hardware resources needed to implement a GPC}}{\text{Amount of hardware resources needed to implement an EPC}} \tag{3.15}$$

For example, assume that we have an application that uses 22% of the target ISA and $M_{const} = \varnothing$, then the application's core density $\eta = 100/22 = 4.5$. That is, the hardware resources of four and a half EPC modules are equivalent to the resources of a single GPC module that implements the target ISA.

Before discussing the relationship between our core density metric and the first measure on opcode usage proposed by Foster *et al.* [19], we will briefly describe the latter. The number of bits, B_{op}, used to encode any opcode from the instruction set I has to be

$$B_{op} \geqslant log_2|I|. \tag{3.16}$$

Let p_k represent the probability that the k-th opcode is used by the processor, then the average number of bits of information contained in each opcode, b_I, is

$$b_I = -\sum_{k=1}^{|I|} p_k(log_2 p_k) \tag{3.17}$$

If all opcodes are equally probable, then

$$p_k = \frac{1}{|I|} \qquad \text{and} \qquad b_{I-max} = log_2|I| \tag{3.18}$$

From Equations (3.16) and (3.18) we see that

$$b_{I-max} \leqslant B_{op} \tag{3.19}$$

According to Foster *et al.* [19], the relative difference between b_I and b_{I-max} is a measure of the utilization of any particular set of opcodes.

Now, let us consider an application with an instruction set A that is set to execute on the application-specific processor, in this case:

$$b_A = b_{A-max} \tag{3.20}$$

since all the opcodes in A are utilized, and thus are equally probable. Also,

$$b_{I-max} = \alpha b_{A-max} \tag{3.21}$$

where $\alpha \geqslant 1$, since $|I| \geqslant |A|$ more bits are required to represent each opcode in the processor instruction set than in the application instruction set. The core density, η, is given by

$$\eta = \frac{\alpha|I| + M_{const}}{\alpha|A| + M_{const}} = \frac{\alpha(2^{b_I}) + M_{const}}{\alpha(2^{b_A}) + M_{const}} \tag{3.22}$$

and from Equations (3.20), (3.21) and (3.22) we have

$$\eta = \frac{\alpha(2^{b_I}) + M_{const}}{\alpha(2^{(\frac{b_{I-max}}{\alpha})}) + M_{const}}. \tag{3.23}$$

From Equation (3.23) we see that core density and the first measure in Foster *et al.* [19] are both related to b_I and b_{I-max}.

The second measure proposed by Foster *et al.* [19] estimates the effort needed to recode an application when the number of opcodes available to the compiler is reduced after an initial unconstrained compilation. Let us consider an application instruction set, A, and a compiler that is constrained to use only the N most frequently used opcodes in A to recompile the application. We will call the resulting application instruction set A_c. Clearly $A_c \subseteq A$ and $|A_c| = N$. The second metric proposed by Foster *et al.* [19] measures the effort required to recompile the code that uses instructions in set A to code that only uses instructions in set A_c. They determined that the effort required is proportional to $e^{-\beta N}$, where β is a constant.

Jones [50] and others have demonstrated abstract processors that have an instruction set consisting of only one opcode. In other words, a compiler can be constrained until $N = 1$ if the compiler supports this universal opcode.

To illustrate the process of constraining a compiler, let us look at an example. Suppose we have an application, which when compiled by an unconstrained compiler has an application instruction set, A_3, consisting of three opcodes; *IF... THEN*, *NOT*, and *OR*, with the first opcode in this list as the most frequently used. For ease of manipulation, we will represent these opcodes by the logic symbols $\Rightarrow$, $\neg$, and $\vee$, respectively. So,

$$A_3 = \{\Rightarrow, \neg, \vee\} \tag{3.24}$$

We further suppose that our compiler is set to recognize the following equivalences:

$$\begin{aligned} (p \Rightarrow q) \qquad &\textit{iff} \qquad (\neg p \vee q) \\ \neg p \qquad &\textit{iff} \qquad (p \Rightarrow \textit{false}) \end{aligned} \tag{3.25}$$

If the compiler is now constrained to $N = 1$, then using the equivalences in Equation (3.25) the object code using A_3 can be recompiled into code that uses application instruction set A_1, where

$$A_1 = \{\Rightarrow\}. \tag{3.26}$$

Note that the transformation from A_3 to A_1 often comes at the cost of more instructions (larger program size) and/or slower execution, since the opcodes that are stripped out of A_3 implement their operations faster than the transformed *IF... THEN* instruction. We will discuss later how by increasing η, this performance penalty can be mitigated.

The core density for an application using a compiler that is constrained to only use the N most popular opcodes and targeting a processor with an instruction set I is

$$\eta = \frac{\alpha|I| + M_{const}}{\alpha N + M_{const}} \tag{3.27}$$

Table 3.1: Architectures Studied

Architecture	Machine Type	Implementation	Unique Opcodes
JVM	Stack	Software	204
MIPS64	Register	Hardware	1182
PowerPC	Register	Hardware	533
x86	Register	Hardware	659
x86-64	Register	Hardware	1101

3.3 ISA Usage Experiments

In our experiments we only considered opcodes and their static usage. We varied four factors; the benchmark applications, the ISA of the target processors, the compilers, and the compiler optimizations. Below is a description of how the experiments were setup.

3.3.1 Architectures and Compilers

A virtual machine with 64-bit version of the Linux (kernel 2.6.32) Ubuntu 10.40 LTS distribution was used as the platform for all the experiments. We used the GNU Compiler Collection (*gcc*) and the Portable C Compiler (*pcc*) to build the C benchmark applications. For the Java applications we used the pre-compiled class files that came with the JVM benchmark applications. We also used the GNU Compiler for Java (*gcj*) to compile the JVM benchmark applications' source code to directly target the x86-64 platform.

We used gcc version 4.4.3 with the m32 flag set and pcc version 1.0 as the native C compilers. The crosstool-ng was used to build the gcc cross-compilers for the MIPS64, PowerPC, and x86-64 architectures, which we will refer to as the hardware processors. We determined the number of unique opcodes for each of the hardware processors in Table 3.1 from the manufacturers' product manuals.

Table 3.2: C Benchmark Applications

Application	**Description**	**Type**
401.bzip2	Data compression tool	Integer
403.gcc	C language optimizing compiler	Integer
429.mcf	Combinatorial optimization tool for single-depot vehicle scheduling	Integer
433.milc	Quantum chromodynamics simulation tool	Floating Point
456.hmmer	Gene sequence database search application	Integer
458.sjeng	Game playing (chess & variants) and pattern recognition application	Integer
464.h264ref	Video compression	Integer
470.lbm	Computational fluid dynamics using Lattice Boltzmann method	Floating Point
482.sphinx3	Speech recognition system	Floating Point
999.specrand	Pseudorandom number generator	Floating Point

3.3.2 Benchmark Applications

The benchmark applications were divided into two categories: the C applications and the Java applications. The ten C benchmark applications are listed in Table 3.2 and they are all part of the SPEC CPU2006 benchmark suite [13]. The applications in this category are further divided, according to their primary computation type, into two groups: floating-point or integer.

The eight Java benchmark applications are listed in Table 3.3 and they are all part of the SPECjvm2008 benchmark suite [14]. Only the class files from this benchmark suite have been used; we did not compile any of the source code.

3.3.3 Method

All the C applications were compiled three times with the gcc compilers, and with each compilation a different optimization scheme was used together with the *-S* option to generate assembler files instead of binary files. In the first iteration there is no optimization, in the second the applications were optimized for size (*-Os* flag), and in the final iteration they were optimized for speed (*-O3* flag). All

Table 3.3: Java Benchmark Applications

Application	Description
Compiler	Compiles a set of .java files using the OpenJDKfront end compiler
Compress	Compresses data using a modified Lempel-Ziv method (LZW)
Crypto	Encrypt & decrypt samples using AES, DES and RSA protocols
MPEGaudio	MP3 audio decoding
Serial	Serializes and deserializes primitives and objects
Startup	Starts each benchmark for one operation
Sunflow	Tests graphics visualization
XML	XML.transform and XML.validation

the C applications were also compiled twice with the pcc compilers, and with each compilation a different optimization option was used. In the first pcc iteration there was no optimization; in the second, the applications were optimized for speed (-*O* flag).

In the case of the Java applications, the classfiles shown in step 3 of Fig. 3.3, provide the starting point for the data generation process. The classfiles were disassembled using the javap utility into JVM assembly files.

The opcodes are extracted by parsing the assembly files for each run and collating them into a single flat file. The data analysis of the raw data files and the generation of the tables and charts were performed using Microsoft Excel.

3.4 Results and Discussion

In this Section we present the results of the ISA utilization experiments and discuss their implications.

3.4.1 Compilers

For the physical architectures, the same source code was used with the gcc and pcc compilers. These two compilers follow different compilation strategies, as

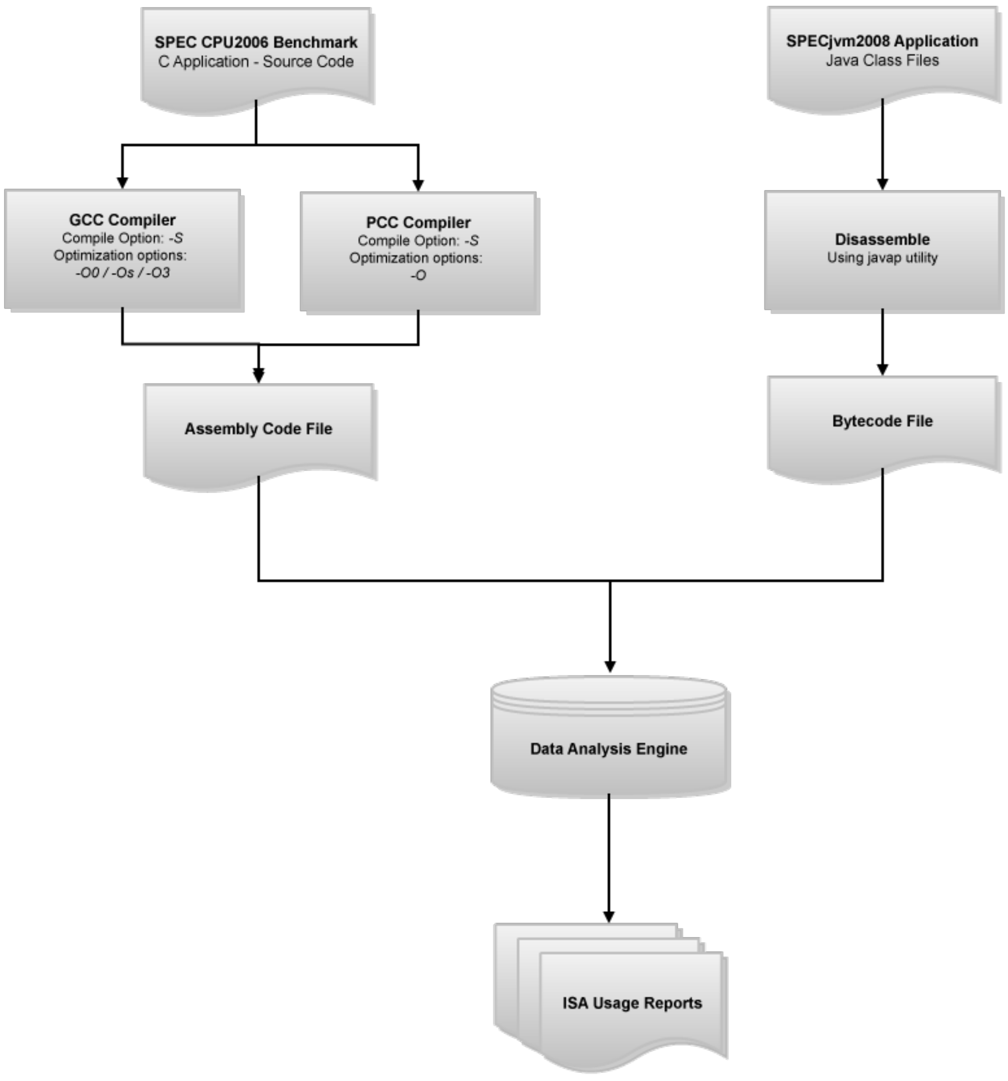


Figure 3.3: Opcode extraction and analysis process for each application.

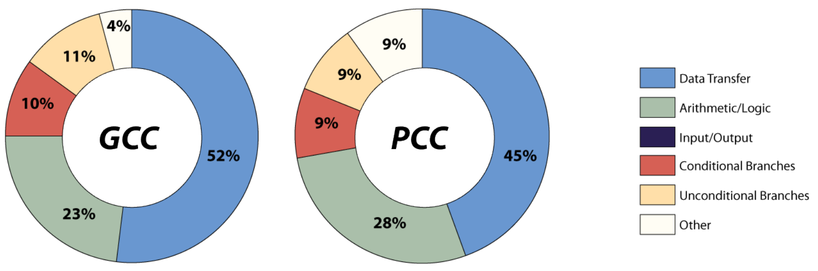


Figure 3.4: Instruction usage by instruction type and compiler.

displayed by the relative differences in the instruction types of the compiled code in Fig. 3.4.

However, with regards to the relationship between instruction set utilization and architecture, the choice of compiler did not make a difference, as seen in Fig. 3.5. The average utilization for all the applications in the C benchmark set is 2% higher that when the applications are considered individually. The PowerPC and x86 architectures have higher utilization rates compared to the MIPS and x86-64 architectures. The latter architectures have larger instruction sets.

Given the relative invariance of the instruction set utilization when different compilers are used, we will, for the remainder of the study, only considers benchmark data generated using the gcc compiler.

3.4.2 Optimizations

The impact of the compiler optimizations on the size of the application is shown in Fig. 3.6. The optimization for speed reduced the average application size for all the hardware processors when compared with the un-optimized applications. The optimization for size results are as expected; across all the platforms, the size of the applications is significantly reduced when compared to both the un-optimized case and the case in which the applications are optimized for speed. No compiler optimization experiments were performed for the software processor since, for this processor, we only used the class files that were provided with the SPECjvm2008 benchmark suite.

While the compiler optimizations affect the program size, we observed that they did not have a significant impact on ISA utilization, as shown in Fig. 3.7. With the exception of the PowerPC architecture, the differences between any pair of results in the ISA utilization-versus-compiler optimizations all fall within a 3% margin. The margin for the PowerPC architecture is 6%. Based on this observation, from this point forward we will only discuss the results for the applications that were optimized for size.

3.4.3 Instruction Set Usage

The number of instructions used by each application is shown in Table 3.4 for the hardware processors and in Table 3.6 for the JVM or the software processor. The x86 and the x86-64 platforms are what were traditionally called CISC-based architectures that use more instructions than the other two hardware processors,

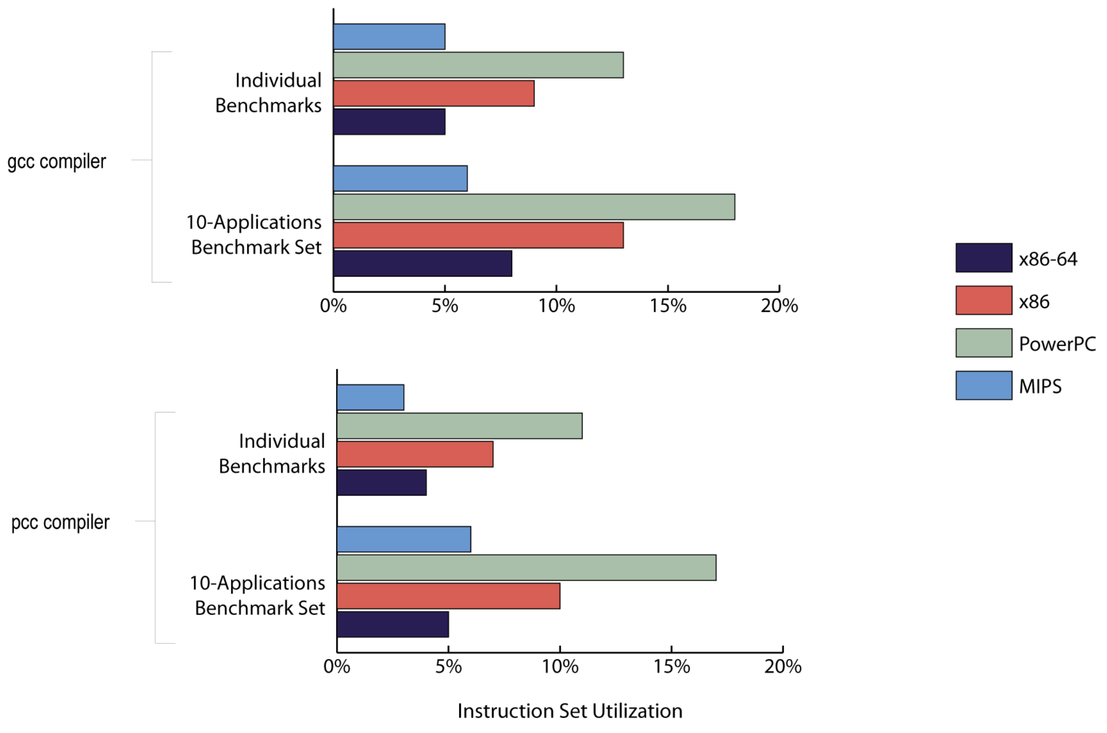


Figure 3.5: Average ISA utilization by target architecture and compiler.

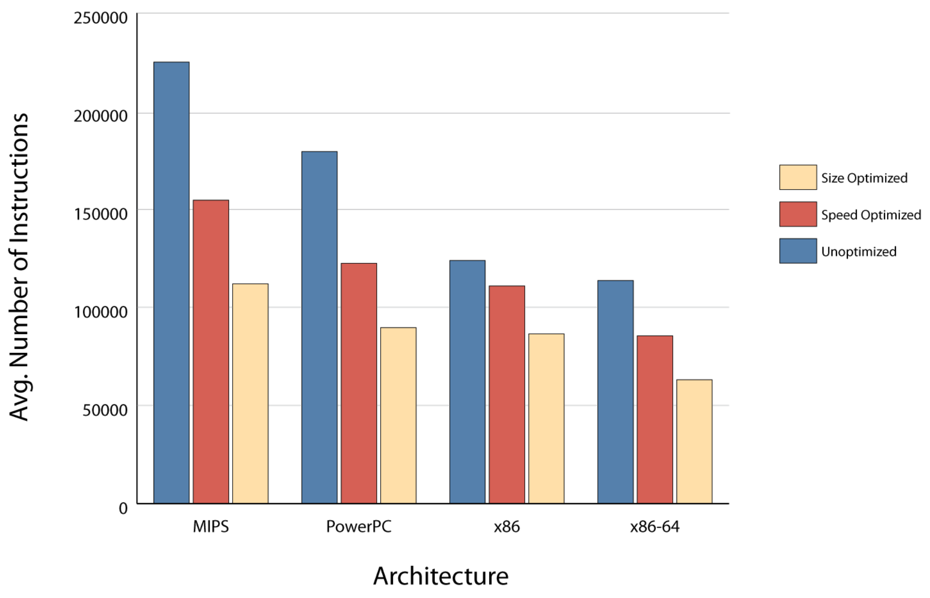


Figure 3.6: Average instruction counts by compiler target and optimization.

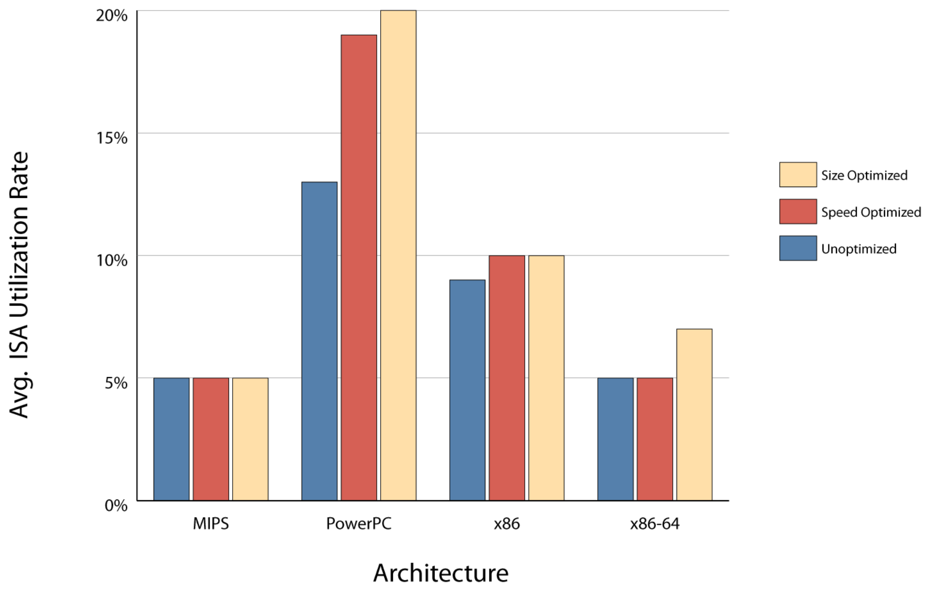


Figure 3.7: Average ISA utilization by compiler target and optimization.

Table 3.4: Number of instructions (opcodes) used

Application\Architecture	**MIPS**	**PowerPC**	**x86**	**x86-64**
401.bzip2	17,543	15,104	12,465	10,095
403.gcc	801,853	630,836	616,909	438,257
429.mcf	2,371	2,135	2,003	1,562
433.milc	27,755	21,279	20,954	16,191
456.hmmer	68,736	56,292	57,601	43,726
458.sjeng	29,999	25,391	19,620	14,331
464.h264ref	124,775	107,945	98,543	77,030
470.lbm	3,953	2,274	2,275	1,875
482.sphinx3	42,780	35,486	34,450	27,557
999.specrand	150	121	108	77
Average number of instructions	**111,992**	**89,686**	**86,493**	**63,070**

traditionally called RISC-based. This result is at odds with one of the widely held views that CISC-based applications require fewer instructions than RISC-based ones. However, there may be two potential explanations for this result. Firstly, the distinction between RISC and CISC in most modern processors is no longer clear-cut. For example, RISC architecture should have fewer opcodes, but a look at the number of opcodes for each processor in Table 3.1 reveals that the MIPS64 processor has more opcodes than the x86-64. Secondly, the discrepancy may be due to the fact that the compiler produces more compact code for the CISC-based processors. The CISC-based processors have been more popular in the market place and hence more resources may have been committed to optimizing compilers for the processors.

In order to estimate M_{const}, we assume that 50% of the processor consists of instruction-dependent hardware. This estimate is in line with findings of Jian-Lun [48]. Using this approximation, η reduces to

$$\eta = \frac{|I|}{|A|}. \tag{3.28}$$

The ISA utilization results for the hardware processors, expressed in terms of core density, are presented in Table 3.5.

Table 3.5: Application Core Densities

Application\Architecture	**MIPS**	**PowerPC**	**x86**	**x86-64**
401.bzip2	20.38	5.03	10.14	19.32
403.gcc	16.42	3.37	7.40	13.59
429.mcf	30.31	8.20	16.07	29.76
433.milc	20.74	5.38	8.45	18.35
456.hmmer	16.19	4.26	7.57	13.43
458.sjeng	17.91	5.03	9.98	18.35
464.h264ref	16.19	3.73	7.01	13.59
470.lbm	23.64	8.46	12.43	25.60
482.sphinx3	16.65	4.33	8.04	14.68
999.specrand	39.40	15.68	24.41	47.87
Average η	**21.78**	**6.35**	**11.15**	**21.45**
η/10-App Set	**15.76**	**3.05**	**6.72**	**11.35**
Max. η/10-App Set	**39.40**	**15.68**	**24.41**	**47.87**

Considering the case of the x86-64 processor, we note that if it were executing any one of the ten applications, its average core density would be 21.5. However, if it were only executing the 999.specrand benchmark application its core density would be 47.9. This indicates that for this application, the traditional processor model (or GPC) fails to utilize more than 98% of the processor resources, since only 1 exact core is required. However, resources equivalent to 48 exact cores are deployed in an existing traditional processor. Based on these results, a multi-core solution may be pursued in order to improve utilization and performance by several orders of magnitude, while using the same hardware resources as a traditional processor. For example, consider an embedded system that executes only the sphinx3 application. By designing an EPC for this application, forty eight such cores can be deployed using the same amount of hardware resources as one existing x86-64 processor. We also note that when using the exact processor model, resources do not need to be optimized for the average case; they can be optimized for the individual case and the resource utilization rate of a multi-core system is equal to the average of the resource utilization rates of its component cores. So, by using exact cores whose individual resource utilization rates have been maximized, the resulting multi-core system will have the maximum resource utilization rate.

If however, all ten applications were to run on the x86-64 processor, then let S be the set of all the applications. That is

$$S = \left\{ \begin{array}{l} \mathit{401.bzip2}, \mathit{403.gcc}, \mathit{429.mcf}, \mathit{433.milc}, \mathit{456.hmmer}, \\ \mathit{458.sjeng}, \mathit{464.h264ref}, \mathit{470.lbm}, \mathit{482.sphinx3}, \mathit{999.specrand} \end{array} \right\}.$$

The application instruction set for S is

$$\bigcup_{k \in S} A_k. \tag{3.29}$$

where A_k is the application instruction set of the k-th application generated by the compiler targeting the x86-64 processor. So for S, the core density, shown in the row for η/10-App Set in Table 3.5, is determined as

$$\eta = \frac{|P_{x86-64}|}{|\bigcup_{k \in S} A_k|} = 11.4. \tag{3.30}$$

This indicates that as more varied applications run on the processor, the utilization rate of the resources improves. However, even in this case, more than 90% of the processor's resources are not utilized.

Instruction-set usage statistics for the software processor are presented in Table 3.6. The average core density for the software processor is lower than that for the hardware processors, indicating better resource utilization by this processor. This may be due to the fact that the JVM has the least number of unique opcodes of all the processors in this study. As the number of opcodes in a processor approaches 1, the core density also approaches 1. In other words, ISA utilization approaches 100%.

3.4.4 Top N Instructions

The top 25 instructions account for more than 89% of all the instructions used by the applications, as shown in Fig. 3.8. This result confirms the locality of reference property of applications and is in line with the results of Adams and Zimmerman [20], and Hennessy and Patterson [22].

Table 3.6: JVM instruction-set usage statistics

Application	**Core Density**	**Opcodes Used**	**Top50 Opcodes**
Compiler	2.83	1357	98%
Compress	2.49	1041	97%
Crypto	3.52	939	100%
MPEGaudio	4.43	149	100%
Serial	2.58	2041	97%
Startup	4.16	363	100%
Sunflow	6.38	57	100%
XML	2.76	1645	96%
Average	**3.64**	**949**	**98%**
η /10-App Set	**1.77**		
Max. η /10-App Set	**6.38**		

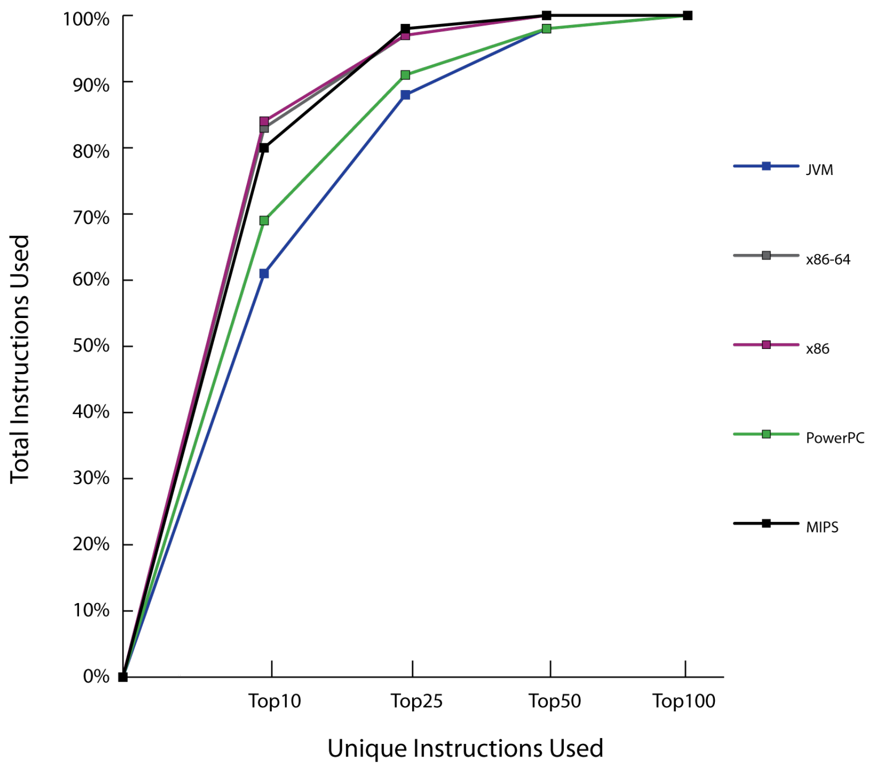


Figure 3.8: TopN opcodes used as a percentage of total opcodes used.

Selecting for the constrained compiler, we get the core densities shown in Table 3.7, and these core densities are in the range of the maximum cases in Table 3.5 and Table 3.6.

The top 100 instructions account for all the instructions used by all the processors. This leads us to question *why most current processors have more than 100 instructions*, which is a reframing of the CISC-RISC debate. A follow-up question is, *which 100 instructions are needed*? Our answer is to keep any existing ISAs, but to provide two options to create application-specific processors. In the first option, an instruction set usage analysis of the application binaries that are set run on the processor is performed to get the set of instructions used. The application binaries may be pre-existing ones or they may be new ones created from source code. An application-specific processor is then synthesized using the set of used instructions; this processor will be referred to as a *subset processor*. In the second option, a processor that supports a subset of the ISA is implemented without knowing beforehand the applications that are going to use it. A constrained compiler targeting this processor is then developed. Finally, the applications set to run on the subset processor are (re)compiled using the new constrained compiler.

Table 3.7: Core densities using the Top25 opcodes.

	MIPS	**PowerPC**	**x86**	**x86-64**	**JVM**
Core Density	118.2	53.3	65.9	110.1	20.4

3.5 Significance of this Study

Our study results indicate that the average resource utilization of modern fixed ISA microprocessors is in the 5-20% range. We have shown that on average, nine heterogeneous exact cores can be placed on the same microprocessor chip in which a single core resides today. As part of this study we defined a new measure, the core density, to measure this property. Resource utilization is application-dependent and the results show that by using exact processor cores, the resulting system can be as much as 48 times more efficient than today's processors.

The results show that maximum resource utilization can be achieved by constraining the number of processor instructions available to the compiler. Using this technique resource utilization can be improved by up to 118 times. This option suggests a new paradigm in compiler design, where the compiler will influence the final instruction set of the processor that will execute the compiled code. It also suggests a change in instruction set design where a universal instruction is available, or a set of primitive instructions is used to build the rest of the instruction set.

Chapter 4

A Computer Architecture Model

4.1 Introduction

The theory of computing is now a well-established area in computer science. A wide array of abstract computing machine models, such as finite-state machines, random-access machines, push-down automatons and Turing machines, can now be proposed, compared, and otherwise analyzed. However, the same cannot be said when it comes to the theoretical foundations for the engineering areas involved in the design and realization of these computing devices. The design of general-purpose computers has, until recently, been largely a qualitative exercise [51]. The work by Flynn [52] and Hennessy and Patterson [27] has helped start a reversal of that trend by putting the design of computers on a quantitative footing. Along those lines, in this work we propose to formalize the description of computer architectures. We develop a generic model for computing architecture that has the following three features:

- **Fidelity:** The model accurately represents the structural parts of a given architecture and how they are related, while at the same time it allows for abstract computer models to represent the operations of these parts and their interactions.
- **Accessibility:** The model is an intuitive, algebraic model that can be used

by computer architects and other designers of digital systems with little training in formal mathematical methods.

- **Extensibility:** The model can be used to model entire systems or the individual components of such systems.

This study adds more details to our earlier presentation of the model in [53]. Our development of the model primarily focuses on the static aspects of a computer's architecture; that is, the associations between its components. The dynamic aspects, which describe how the connected components interact in order to perform the computations, are discussed briefly.

In this chapter we develop the simple model of computer architecture, together with an example of its application. We then extend the simple model into the general model. Applications of the general model to microprogramming, reconfigurable computing, microprogrammed processors, general computing systems and virtualized systems are presented. Finally, a model of the subset microprocessors that were created as a part this research is developed.

4.2 Formulating the Simple Model

For the development of the first iteration of our model we are going to use, as an example, a simple microprocessor that has a hard-wired control unit without pipelining and without any other sophisticated microarchitecture. Ways to add new microarchitectural features to the processor model will be discussed later, in the Applications section. We will call this simple microprocessor the ***k85***. The simple model will follow along the lines of the one proposed by Sima *et al.* [32]; however, we make significant extensions to that model in order to facilitate later generalizations. Our model uses a triplet instead of a doublet as the primary structure and we introduce the use of relations to describe the relationship between the triplet elements.

Let us assume that we have a computing machine (CM) which consists of a set of N computing resources (R) that use a set of Q operations or instructions (I)

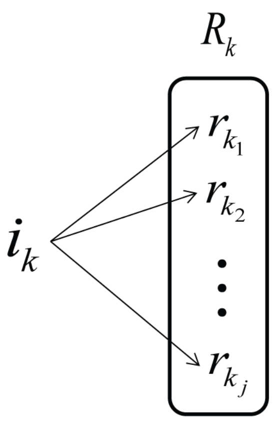


Figure 4.1: Instruction to resources relation.

to operate on a set of P data types (D). An implementation of a CM can be formalized by means of a triplet:

$$CM = (I, R, D) \tag{4.1}$$

For the instruction set,

$$I = \{i_1, i_2, \ldots, i_Q\}. \tag{4.2}$$

Each instruction, i_k, where $1 \leqslant k \leqslant Q$, is typically represented by an instruction mnemonic or an opcode. Similarly, for the computing resources or functional units,

$$R = \{r_1, r_2, \ldots, r_N\}. \tag{4.3}$$

A resource, r_k, where $1 \leqslant k \leqslant N$, may represent a component of the microprocessor, such as an adder or a register.

Each instruction, i_k, controls one or more resources, as shown in Fig. 4.1. That is, each instruction controls a set of resources, R_k, where

$$R_k \subset R$$

$$|R_k| > 0. \tag{4.4}$$

We introduce a set of triggers, T. These triggers are state transitions that are used to initiate other processes. An example of such triggers is a set of X sequential rising edges of a clock signal that are numbered and represented as $\{clock_1, clock_2, \ldots, clock_X\}$.

$$T = \{t_1, t_2, \ldots, t_X\} \tag{4.5}$$

Let U_k represent the relation between i_k, R_k, and T_k, where $T_k \subset T$ and it represents the set of triggers needed to implement instruction i_k. We are now able to model U_k as a set of triplets, where each instruction's resource pool can now be modeled up to the point in which each element in the pool is required. That is,

$$U_k = \{(i_k, r_{k_1}, t_{k_1}), (i_k, r_{k_2}, t_{k_2}), \ldots, (i_k, r_{k_N}, t_{k_X})\}. \tag{4.6}$$

Following a similar line of development, we now considering the data types

$$D = \{d_1, d_2, \ldots, d_P\}. \tag{4.7}$$

Each data type, d_k, where $1 \leqslant k \leqslant P$, represents a data format or addressing mode. Data types may represent *immediate data* or *indirect data*. Immediate data is embedded in the instruction and as such is available for immediate processing by the computing resources. Indirect data represents a location where the computing resources can find the data to be processed as part of the instruction execution.

Each resource, r_k, can operate on zero or more data types, as shown in Fig. 4.2. That is, each resource can process a set of data types, R_k, where

$$D_k \subset D$$

$$|D_k| \geq 0. \tag{4.8}$$

Let V_k represent the relation between r_k, D_k and T_k. That is,

$$V_k = \{(r_k, d_{k_1}, t_{k_1}), (r_k, d_{k_2}, t_{k_1}), \ldots, (r_k, d_{k_N}, t_{k_X})\}. \tag{4.9}$$

Putting the component of the triplet together, as shown in Fig. 4.3, we see that the composition $U_k \circ V_k$ allows us to express the relationship that exists between the instruction i_k and data of type d_k whenever there exists a suitable resource r_k at interval t_j:

$$i_k\,(U_k \circ V_k)\,d_k \Leftrightarrow \{\exists r_k \exists t_j | (i_k U_k r_k) \wedge (r_k V_k d_k)\} \tag{4.10}$$

Suppose bOp is a generic binary operation, such as addition, and that i_k is a specific instance of that operation acting on two data items of types d_{k_1} and d_{k_2}. An example of i_k would be the addition of an unsigned integer and a floating point number. We have $i_k \in bOp$ and

$$[i_k(U_k \circ V_k)d_{k_1} \wedge i_k(U_k \circ V_k)d_{k_2}] \Rightarrow (d_{k_1}\; bOp\; d_{k_2})\,. \tag{4.11}$$

Generalizing the result above to i_k that is an n-ary operation (nOp) acting on n data items, we have

$$\begin{bmatrix} i_k(U_k \circ V_k)d_{k_1} \\ \wedge\ \ldots\ \wedge \\ i_k(U_k \circ V_k)d_{k_n} \end{bmatrix} \Rightarrow nOp\,(d_{k_1}, \ldots, d_{k_n})\,. \tag{4.12}$$

The distribution of processor specifications in a format consistent with relation-

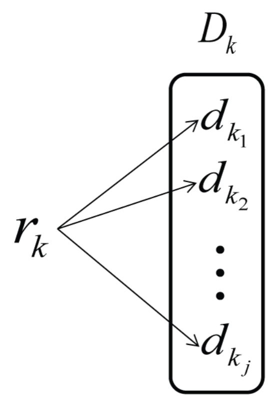


Figure 4.2: Resources to data relation.

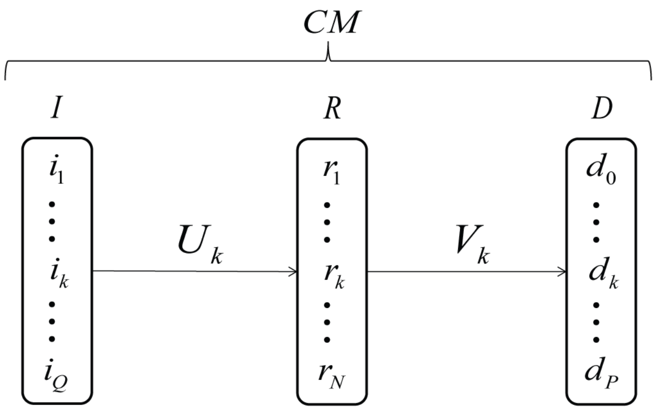


Figure 4.3: Graphical representation of the triplet model.

ships in Equations (4.10), (4.11), and (4.12) should facilitate automated compiler construction for new architectures.

Next, we present a way to model the dynamic aspects of a computer's architecture. The resources needed to execute each instruction, as shown in Fig. 4.1, are often marshaled using a sequence of microoperations that are ordered by the elements of the trigger set T. We introduce a set of microoperations M where

$$M = \{m_1, m_2, \ldots, m_Y\}. \tag{4.13}$$

For each instruction, i_k, there is a corresponding set of microoperations, M_k, where $M_k \subset M$. Each microoperation, m_{k_i}, where $m_{k_i} \in M_k$, accesses at most two resources, $r_{k_{i1}}$ and $r_{k_{i2}}$, for inputs. Each microoperation places its result in one resource, r_{k_o}, at most. Now, we are able to model W_k, the operations of each instruction, and i_k over the interval T_k as a sequence of triplets shown in Equation (4.14).

$$W_k = \begin{pmatrix} (m_{k_1}, R^w_{k_1}, t_{k_1}), (m_{k_2}, R^w_{k_2}, t_{k_2}), \\ \ldots, (m_{k_X}, R^w_{k_X}, t_{k_X}) \end{pmatrix} \tag{4.14}$$

where

$$R^w_{k_i} = (r_{k_{i1}}, r_{k_{i2}}, r_{k_o}).$$

4.2.1 Example of the Simple Model

In this section we apply the model to the k85 which is a simple 8-bit microprocessor that is binary-compatible with the Intel 8080 and the Intel 8085. The architecture diagram of the k85 microprocessor is shown in Fig. 4.4. We assume that the controller for our processor is hardwired and not microprogrammed. At the top level we describe the processor model as

$$CM_{k85} = (I_{k85}, R_{k85}, D_{k85}) \tag{4.15}$$

where,

$$I_{k85} = \{ANDA, MOV, CALL, \ldots, XHL\} \tag{4.16}$$

using instruction mnemonics or using opcode templates

$$I_{k85} = \left\{ \begin{array}{l} 10100XXX, 01XXXXXX \\ 11001101, \ldots, 11100011 \end{array} \right\}. \tag{4.17}$$

The processor has 59 types of instructions; that is,

$$M = |I_{k85}| = 59. \tag{4.18}$$

The width of the data bus is 8 bits; we will use that as the default size of each resource. Any resource with a different size will be shown with its size in parenthesis next to the resource name. Using Fig. 4.4, we can put together R_{k85} as

$$R_{k85} = \{ALU, \textit{Control Unit}, \textit{Register File}, \ldots\} \tag{4.19}$$

where *Register File* is a macro for an array of all the registers in the processor. That is,

$$\begin{aligned} \textit{Register File} = \; & A, B, C, D, E, \textit{Flags}, H, L, \textit{IR}, . \\ & \textit{PC(16)}, \textit{SP(16)}, \textit{Temp}, W, Z \end{aligned} \tag{4.20}$$

Some registers or combinations of registers are directly available to I_{k85}, and these represent data types. In the case of our processor, *REG*, is the set of available 8-bit registers, while *REG16* consists of overlapping register pairs, or other 16-bit

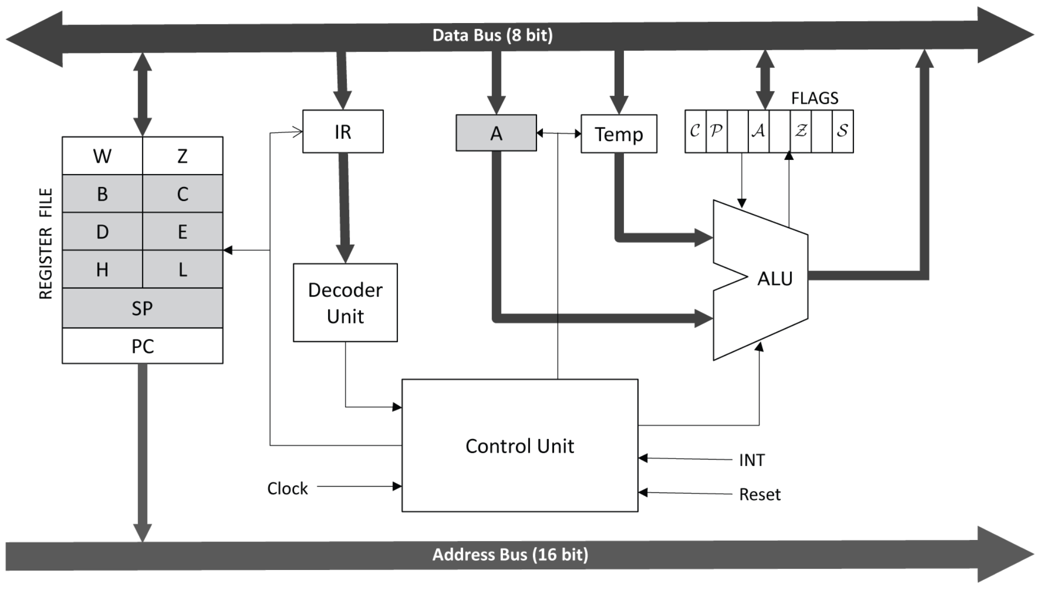


Figure 4.4: The k85 microprocessor architecture block diagram.

registers that are available to I_{k85}.

$$REG = \{A, B, C, D, E, H, L\} \tag{4.21}$$

and

$$REG16 = \{BC, DE, HL, PC(16)\}. \tag{4.22}$$

Now we can put together the data types for our processor.

$$D_{k85} = \begin{array}{l} REG \ \cup \ REG16 \cup \\ \{address(16), data, data(16), port\} \end{array} \tag{4.23}$$

where *data* and *port* are any 8-bit numbers representing data or a port respectively, while *data(16)* and *address(16)* are any 16-bit numbers representing data or a memory address respectively.

Next, the relations S and T are specified. Let us consider the **ANDA** instruction group. That is, $k = ANDA$ in Equations (4.6) and (4.9).

The **ANDA** instruction has two options:

ANDA *register*	–	the *register* is ANDed with the A register and the result is stored in A.
ANDA *M*	–	the data in the memory location pointed to by the contents of the *HL* register is ANDed with the A register, and the result is stored in A.

In Register Transfer Level (RTL) notation, the actions performed by the **ANDA** instruction are described as follows:

$$\begin{aligned} \mathit{ANDA\ register} : t_1 \ & A \leftarrow A \wedge \mathit{register} \\ & \mathcal{Z} \leftarrow \mathit{1} \\ & \mathcal{S} \leftarrow \mathit{1} \\ & \mathcal{P} \leftarrow \mathit{1} \\ & \mathcal{C} \leftarrow \mathit{0} \\ & \mathcal{A} \leftarrow A\langle \mathit{3} \rangle \vee \mathit{register}\langle \mathit{3} \rangle \end{aligned} \tag{4.24}$$

where *register* is replaced by M for the second option, and the number between each angled bracket represents a bit position. $\mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S}$ and $\mathcal{Z}$ represent the individual flags in the *Flags* register. We can now model how this instruction operates as

$$U_{ANDA} = \left\{ \begin{array}{l} \mathit{(ANDA,ALU)}, \mathit{(ANDA,REG)}, \\ \mathit{(ANDA,HL)}, \mathit{(ANDA,Control\ Unit)}, \\ \mathit{(ANDA,IR)}, \mathit{(ANDA,Decoder\ Unit)} \end{array} \right\} \tag{4.25}$$

$$V_{ANDA} = \left\{ \begin{array}{l} \mathit{(ALU,REG)}, \mathit{(ALU,data)}, \mathit{(ALU,Flags)}, \\ \mathit{(PC,address(16))}, \mathit{(Control\ Unit,REG)}, \\ \mathit{(Decoder\ Unit,data)} \end{array} \right\} \tag{4.26}$$

$$W_{ANDA} = \left(\begin{array}{l} (\wedge, (A, \mathit{register}, A), t_1), \\ (, (\mathit{1}, , \mathcal{Z}), t_1), \\ \qquad \qquad \cdots \\ (\vee, (A\langle \mathit{3} \rangle, \mathit{register}\langle \mathit{3} \rangle, \mathcal{A}), t_1) \end{array} \right). \tag{4.27}$$

4.3 The General Model

We generalize our model by transforming Equation (4.1) into a recursive formalism. Our simple model may be viewed as a computing object consisting of three related components: the program or ordered set of *instructions* that direct some computing *resources* to act on some *data*. The computing object can be represented by the triplet

$$\mathit{computing\ object} = (\mathit{instructions}, \mathit{resources}, \mathit{data}). \tag{4.28}$$

In the general model, parts of *I*, *R* and *D* from Equation (4.1) may be replaced by models for computing objects that each have a format corresponding to Equation (4.28). We represent the general model of the computing machine using the following triplet

$${}^{\mathit{level}}_{\mathit{architecture}}CM = \left\{ \begin{array}{ll} {}^{\mathit{(level+1)}}_{\mathit{architecture}}I, & {}^{\mathit{(level+1)}}_{\mathit{architecture}}R, \\ {}^{\mathit{(level+1)}}_{\mathit{architecture}}D & \end{array} \right\} \tag{4.29}$$

and the relations as

$${}^{\mathit{(level+1)}}_{\mathit{architecture}}U \quad \text{and} \quad {}^{\mathit{(level+1)}}_{\mathit{architecture}}V. \tag{4.30}$$

The *implementation reference level* (IRL) is defined as $\mathit{level} = 0$, as shown in Fig. 4.5. The IRL can be set arbitrarily; however, it is preferable to set the IRL as close to the primary computing device being modeled. In this way the components of the device being modeled will appear at $\mathit{level} > 0$ and any aggregate structures or networks using the device will appear at $\mathit{level} < 0$.

4.3.1 Example of the General Model

Let us consider the ALU from our previous example as an assembly of a multiplexor together with six 8-bit circuits that implement the ***Adder***, ***Shifter***, ***AND***, ***OR***, ***XOR*** and ***NOT*** functions, as shown in Fig. 4.6. The MUX, along

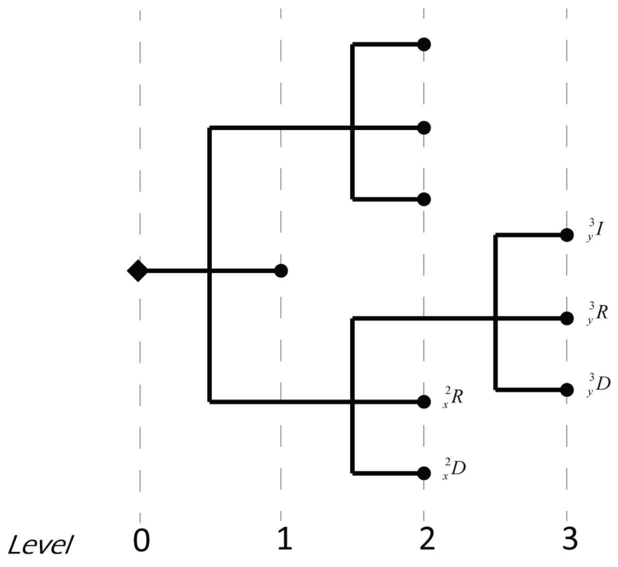


Figure 4.5: Tree-based representation of the generic model.

with its input and output signals, form the processor's *control plane*. The 8-bit circuits along with their data input and output form the processor's *datapath*.

The ALU can now be modeled as

$$ {}^{2}_{k85}ALU = \left(\begin{array}{l} {}^{3}_{MUX}OpSelLUT, \\ {}^{3}_{MUX}OpModules, \\ {}^{3}_{MUX}ControlSignals \end{array} \right) \tag{4.31}$$

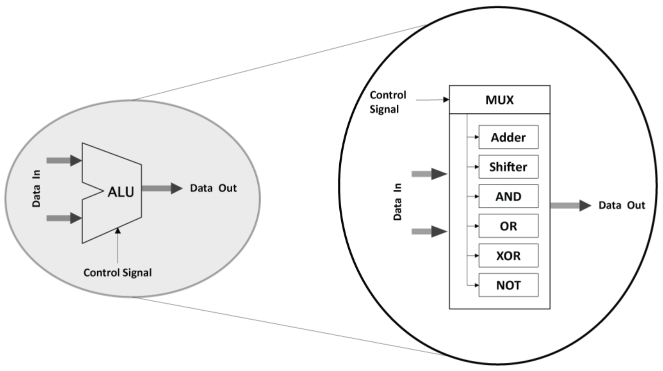


Figure 4.6: The arithmetic logic unit (ALU).

where ${}_{MUX}^{3}OpSelLUT$ is the set of codes used to direct the MUX to select the appropriate function circuit,

$$ {}_{MUX}^{3}OpModules = \left\{ \begin{array}{l} Adder,\ Shifter,\ AND, \\ OR,\ NOT,\ XOR,\ MUX \end{array} \right\} \tag{4.32} $$

The ${}_{MUX}^{3}ControlSignals$ emanate from the Control Unit shown in Fig. 4.6. We can now rewrite Equation (4.29) as

$$ {}_{k85}^{0}CM = ({}_{k85}^{1}I, {}_{k85}^{1}R, {}_{k85}^{1}D) \tag{4.33} $$

where

$$ {}_{k85}^{1}R = \left\{ {}_{k85}^{2}ALU, {}_{k85}^{2}ControlUnit, \ldots \right\}. \tag{4.34} $$

4.4 Applications of the Model

Our proposed model can be used to either:

1. Describe the architecture of an entire computing machine, as the virtualization application example below shows, or to
2. Describe an optimized part of an existing machine, as demonstrated by the microprogramming application example below.

An existing lower (numerical) level model may be extended by 'plugging' into it the model of a new higher level component, as shown in Fig. 4.5.

In general the proposed model can be used in the following manner: Firstly, identify a potential processor application or optimization scheme. Next, generate a triplet covering the whole application area. If required, interface this new model to an existing higher- or lower-level model. Finally, define the relationships between the triplet components.

In addition to the application of the triplet model to describe a fixed processor that we have used throughout the study in modelling the k85 microprocessor, we are also going to provide five more examples to demonstrate the applications of this model.

4.4.1 Microprogramming

While hard-wired processors, like the one in our microprocessor example above may offer a performance advantage over their microprogrammed counterparts, many commercial microprocessors today are microprogrammed. Microprogramming offers the following advantages when compared to hard-wired architectures: ease of development and maintenance, flexibility, and lower costs [54].

A microprogram is a sequence of microinstructions that are not directly accessible to the programs running on the machine. Each microinstruction corresponds to a primitive operation that the machine can perform, often referred to as a micro-operation. The microinstructions are often described using RTLl. A processor's programmer-visible instruction can then be described by a microprogram, as the example of the **LHL** instruction from the k85 architecture shows.

$$\begin{array}{rll} \mathit{LHL\ addr}: & t_1 & Z \leftarrow M[PC] \\ & & PC++ \\ & t_2 & W \leftarrow M[PC] \\ & & PC++ \\ & t_3 & L \leftarrow M[WZ] \\ & & WZ++ \\ & t_4 & H \leftarrow M[WZ] \end{array} \tag{4.35}$$

The microprogrammed control unit can be implemented using control memory, a control address register, and a next address generator unit [55], as shown in Fig. 4.7. Each microinstruction is stored as a word in the control memory.

We model the microprogrammed control unit as the triplet:

$$^{2}ControlUnit = \begin{pmatrix} {}^{3}I_n, \{{}^{3}R_a, {}^{3}R_m\}, \\ \{{}^{2}D_i, {}^{3}D_n, {}^{2}D_o\} \end{pmatrix} \tag{4.36}$$

where $^{2}D_i$ and $^{2}D_o$ map one-to-one with the Control Unit input and output

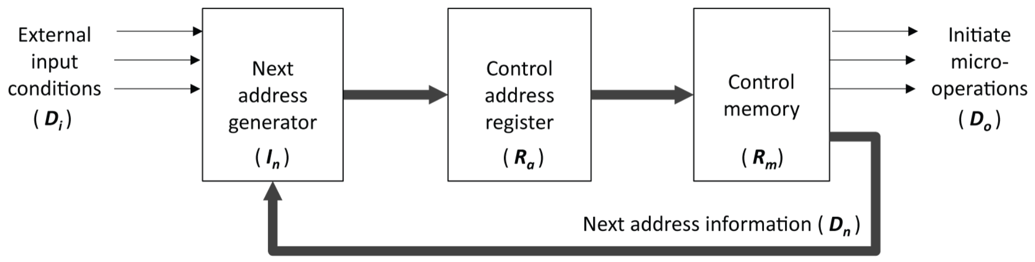

External input conditions (D_i)
Next address generator (I_n)
Control address register (R_a)
Control memory (R_m)
Initiate micro-operations (D_o)
Next address information (D_n)

Figure 4.7: Microprogramming control unit organization.

signals from Fig. 4.6, respectively. ${}^{3}I_n$ is a hardware circuit that generates the next address to be latched into ${}^{3}R_a$ based on ${}^{2}D_i$ and ${}^{3}D_n$. Now, the model in Equation (4.36) can be substituted into the model in Equation (4.34).

The discussion above, as well as the model in Equation (4.36), can be applied to co-designed virtual machines. By way of contrast, Chen *et al.* provide a state machine-based model [30] of such machines. In a co-designed virtual machine the source architecture, that is the one visible to the binary applications running on the machine, is emulated on a target architecture. One of the most well-known co-designed virtual machines is the Transmeta Crusoe processor [56] which uses a 'code morphing' (CMS) layer [57] to transparently run Intel IA-32-based software, which serves as the source architecture, on an underlying very Long instruction word (VLIW) architecture. This serves as the target architecture. Using our model, this CMS layer can be implemented in ${}^{3}I_n$ in order to maximize performance or it can be implemented in ${}^{3}R_m$ in order to maximize design flexibility and post-production maintenance.

4.4.2 Reconfigurable Computing

Given a reconfigurable system RC, the amount of physical hardware (R_{flex}) remains constant; however, the logic functions $F(x)$ that are implemented on this hardware change over the lifetime of the system, or even the lifetime of the set of applications, App, that is scheduled to run on the system. In addition to the applications, configuration software S_{config}, that is used to program the logic functions onto the hardware, is also input into the system. We can now model this system as

$$^{0}RC = (^{1}F(S_{config}), ^{1}R_{flex}, ^{1}D) \tag{4.37}$$

where the data member of the model

$$^{1}D = \{App, S_{config}\} \tag{4.38}$$

and the model's program member is mapped directly onto the hardware, since

$$^{1}F(S_{config}) \quad \mapsto \quad App. \tag{4.39}$$

The net effect is that the application will be directly executed by the hardware thus sidestepping the fetch-decode-execute cycle of traditional microprocessors [31].

4.4.3 Microprogrammed Processor

The microprogrammed processor model can be generated from the model of the fixed processor by replacing the *hard-wired control unit* in Equation (4.34) with the *microprogrammed control unit* from Equation (4.36).

4.4.4 Basic Computer System

A basic computer system (BCS), in general, consists of one or more microprocessors, a ROM module, a RAM module, and an IO interface module. All these modules are connected to the data, address, and control buses [55]. Attached to this system are the IO devices, such as a keyboard and a hard-disk drive. All the above-mentioned items form the R member of the triplet model for the BCS. Each of the elements of R may be replaced by a higher-level triplet that describes its functions in more detail. For example, the microprocessor in R may be replaced by a triplet model of a microprocessor, like the one shown in Equation

(4.33). We model the basic computer system as

$$ {}_{arch}^{0}BCS = ({}_{arch}^{1}I, {}_{arch}^{1}R, {}_{arch}^{1}D) \tag{4.40} $$

where

$$ {}_{arch}^{1}I = \left\{ \begin{array}{l} \textit{Process Control, Interrupt Handlers,} \\ \textit{Device Drivers, Memory Management, \ldots} \end{array} \right\} $$

and

$$ {}_{arch}^{1}D = \left\{ \begin{array}{l} \textit{Bootsrap Program, System Program,} \\ \textit{User Programs, System Calls, \ldots} \end{array} \right\}. $$

4.4.5 Virtual Computer System

Virtualization is formally described as a one-one homomorphism between a 'real' system and a 'virtual' system, with respect to all the operators in an instruction sequence set [58]. That is, for any state transformation in the 'real' system, an equivalent transformation can be performed in the 'virtual' system. One realization of virtualization is through virtual machines (VM). A VM is a software layer that emulates a desired machine's architecture [29]. The VM executes (runs) on a real machine whose architecture may or may not be the same as that emulated by the VM.

We will model the generic virtual machine as

$$ {}_{architecture}VM = \left(\begin{array}{l} {}_{architecture}I, {}_{architecture}^{soft}R, \\ {}_{architecture}D \end{array} \right), \tag{4.41} $$

while the physical or 'real' machine is modeled as

$$ {}_{architecture}PM = \left(\begin{array}{l} {}_{architecture}I, {}_{architecture}R, \\ {}_{architecture}D \end{array} \right) \tag{4.42} $$

where ${}_{architecture}^{soft}R$ is a set of programs that emulate corresponding elements of ${}_{architecture}R$ in the model in Equation (4.42). Note that there are no physical components in ${}_{architecture}VM$.

The model for a physical machine that is hosting a VM is

$$_{arch\_1}CM = \Big(_{arch\_1}VMM, {}_{arch\_1}PM, {}_{arch\_2}VM \Big) \tag{4.43}$$

where VMM is the virtual machine monitor or hypervisor.

Traditionally, if $arch\_1 \neq arch\_2$, the host machine model above is said to represent a *simulation*, otherwise, the model represents *virtualization*.

The host machine model presented above allows us to more easily propose and describe extensions to existing virtualization technologies. An example is a *multi-tenancy hypervisor* that can support multiple virtual machines with different architectures. Such a system can be modeled by replacing $_{arch\_2}VM$ in Equation (4.43) with $\{_{arch\_1}VM, \ldots, {}_{arch\_n}VM\}$. In this case, we can have, say, an x86 hypervisor that supports ARM, PowerPC, and x86 virtual machines modeled as

$$_{x86}CM = \left(\begin{array}{l} _{x86}VMM, {}_{x86}PM, \\ \{_{ARM}VM, {}_{PowerPC}VM, {}_{x86}VM\} \end{array} \right). \tag{4.44}$$

We note that all the elements of the model triplet can be implemented as hardware or software, depending on the purpose of the model. For example, the I triplet-component in the microprogramming example in Equation (4.36) is implemented in hardware, while the I, R, and D triplet components in the virtualization example in Equation (4.41) are all implemented in software.

4.5 Reconfigurable Subset Processor Models

One of the goals of this thesis is to develop an automated system to synthesize subset processors and implement them on a reconfigurable computing platform. In this section we use the ideas developed so far in this study to outline a theoretical framework for the subset processors on a reconfigurable computing fabric.

Using the same notation that was presented in Section 3.2 for the processor instruction set (I) and the application instruction set (A), we model $_{app}SP$, the subset processor for the application stored in the binary file ${}_{app}D$, as

$$ {}_{app}SP = \left(A, R_A, {}_{app}D \right) \tag{4.45} $$

where from Equation (3.7), A is a subset of I. In order to place the subset processor onto a reconfigurable platform, the model from Equation (4.45) is now used as the configuration program for the reconfigurable device, and Equations (4.38) and (4.39) are rewritten as

$$ {}^{1}D = \{ {}_{app}D \ , \ {}_{app}SP \} \tag{4.46} $$

and

$$ {}^{1}F({}_{app}SP) \quad \mapsto \quad {}_{app}D. \tag{4.47} $$

The reconfigurable platform hosting the subset processor can now be modeled by replacing Equations (4.38) and (4.39) in Equation (4.37) with Equations (4.46) and (4.47), respectively

$$ {}^{0}RC = ({}^{1}F({}_{app}SP), {}^{1}R_{flex}, {}^{1}D). \tag{4.48} $$

In order to model a set of B subset processors placed on a single reconfigurable device, such as might be found on multicore systems, we introduce an index, k, where $1 \leqslant k \leqslant B$ and ${}_{app}SP_k$ is the k-th subset processor on the reconfigurable device. We now rewrite Equations (4.45), (4.46) and (4.47) for the k-th subset processor as

$$ {}_{app}SP_k = \left(A_k, R_{Ak}, {}_{app}D_k \right) , \tag{4.49} $$

$$ {}^{1}D_k = \{ {}_{app}D_k \ , \ {}_{app}SP_k \}, \tag{4.50} $$

and

$$ {}^{1}F({}_{app}SP_k) \quad \mapsto \quad {}_{app}D_k \tag{4.51} $$

The multiple subset processor version of the triplet model's data component is now represented by

$$ {}^{1}D = \bigcup_{k=1}^{|B|} {}^{1}D_k. \tag{4.52} $$

while its instructions component is now represented by

$$ {}^{1}F({}_{app}SP) = \bigcup_{k=1}^{|B|} {}^{1}F({}_{app}SP_k). \tag{4.53} $$

Equations (4.52) and (4.53) can now be substituted into Equation (4.48) in order to produce the multi-subset-processors model.

4.6 Significance of this Study

In this study, we have developed a formal model of computer architecture and shown how this model can be used to describe physical and virtual computer systems together with their components, such as hard-wired and microprogrammed processors. We have also shown how reconfigurable computing systems may be modeled using our proposed framework. The triplet components of the model can be used to represent programs, hardware/resources, and the data of the system under design to an arbitrary level of detail as required by the designer. The model requires an understanding of some basic set theory and Boolean logic operators, both of which are almost universally accessible to computer architects and other digital designers. The level notation combined with the recursive nature of the formalism allows the model to be extended by including detailed sub-component triplets or by adding the primary model's triplet into that of a larger super structure. We believe that this model combined with the quantitative architecture tools mentioned in the introduction, can help take the design of new computers architectures from an art form to a science form.

Chapter 5

Matching Processors to Applications

5.1 Introduction

Driven by Moore's law, the number of transistors available to system designers continues to grow exponentially resulting in the exponential fall in the average cost of each transistor. Designers continuously take advantage of this dynamic by adding more functions into the hardware which has enabled the recent trends, like SoCs and multiple-processor SoCs (MPSoCs). As more functions are added to electronic systems, their designs get more complex. This complexity increases the risks in both the design project and the final product. To deal with this complexity and it consequences, the design process has been abstracted away from the hardware layer to progressively higher software layers. System design has evolved from transistor level to schematic level and then to hardware description language (HDL) level.

The challenges in electronic system design now relate to the design of SoCs and MPSoCs. There are technical and non-technical issues that need to be considered as part of the design process. The key technical issues revolve around the need to balance the performance of the system with its size on the chip and its power consumption/dissipation. Meanwhile, the principal non-technical issues relate to

- time to market
- costs
- availability of skilled personnel for product development, and
- post-sales/in-field product support, maintenance, and upgrades.

The complexity involved in the design of SoCs using hardware description language makes it difficult to meet the challenges mentioned above - particularly the non-technical ones. A new higher level of abstraction, the application level or electronic system level (ESL), is now taking shape to address the design of SoCs. This work presents one such ESL tool.

SoCs by definition have at least one microprocessor or processing element. In Chapter 3, using the core density measure, we showed how existing microprocessors use their resources inefficiently. In this study, we seek to develop an application that optimizes the number of resources that are utilized by microprocessors. We refer to this application as the **FiT** toolkit and it uses the following steps to automatically generate a custom processor. Firstly, FiT analyzes the instruction set usage of all the programs that are set to run on the microprocessor. And as we found out in Equation (3.13) the set of instructions that are used by the programs is much smaller than the set of instructions that are supported by the processor. Next, FiT generates a VHDL model of the subset processor that only supports the instructions found in the first step. This model is optimized for area, by not implementing those functions that will not be needed by the programs. FiT uses the model presented Equation (4.45) to develop the VHDL model of the subset processor. Each instruction in the instruction set has some VHDL code associated with it, when launched FiT builds a tree structure that stores these associations and when an instruction is used by a program the associated VHDL code is added to the VHDL model. Other custom processor generators, such as the one by Trajkovic *et al.* [59], focus on generating custom processors that are optimized for performance. They do so by analyzing the C source code of the programs set to run on the custom processor and identifying any parts of the program that can be parallelized. In this chapter we use FiT to generate custom soft processors that implement subsets of the Intel 8080/8085 instruction set architecture (ISA) [60].

5.2 Toolkit Description

FiT automatically generates custom soft processors based on the theory of subset processors proposed in chapters 3 and 4. The toolkit takes as its input one or more application binary files and the template for the general-purpose processor template. The toolkit outputs an HDL model of the custom soft processor, which is then implemented on a reconfigurable computing fabric. In this study, the recon-

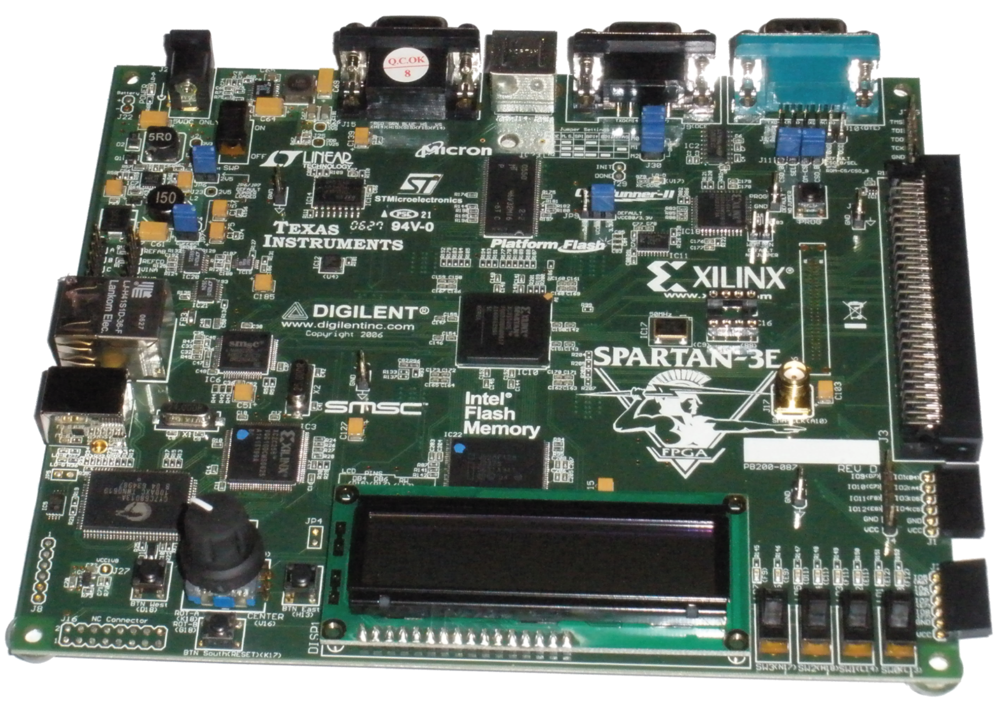


Figure 5.1: The Xilinx Spartan 3E Development Board.

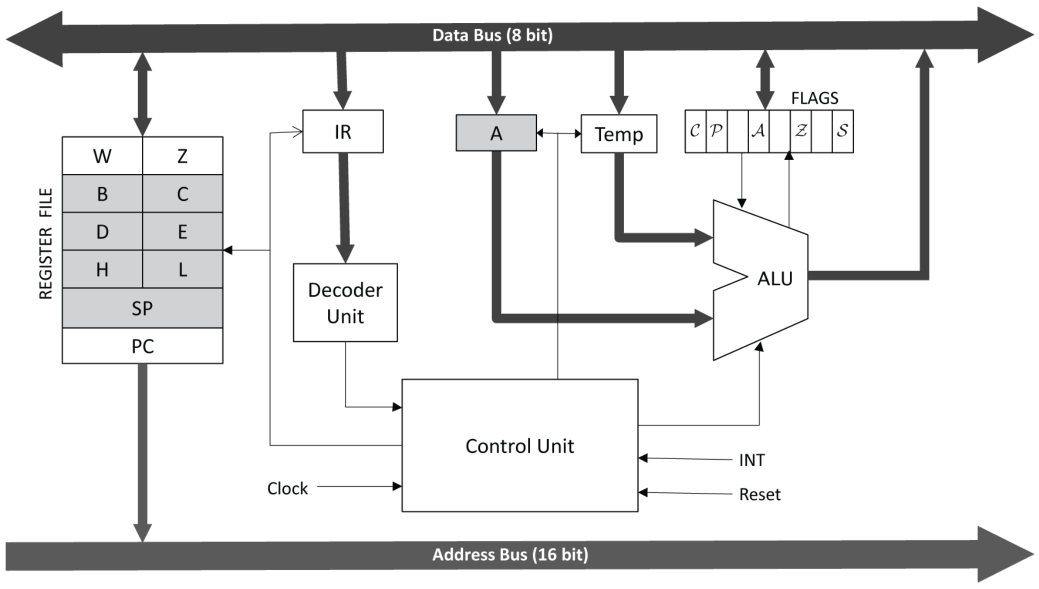


Figure 5.2: The k85 microprocessor architecture block diagram.

figurable computing fabric consisted of a Xilinx Spartan 3E field-programmable gate array (FPGA) on a Digilent development board, as shown in Fig. 5.1.

5.2.1 General-Purpose Processor Templates

Two types of general-purpose processor templates are used as input to the FiT toolkit. The first, which we refer to as the structural template, combines models of all the components of the physical processor. The structural template emulates the datapath and control of the physical processor shown in Fig. 5.2. The only major architectural difference between the soft processor generated from the structural template and the physical processor is that the control module for the soft processor is hard-wired and not micro-coded. In the FiT toolkit, only the control module is customizable when using structural template. The structural template results in an area-optimized implementation of the processor where only one instruction can be executed in each clock cycle.

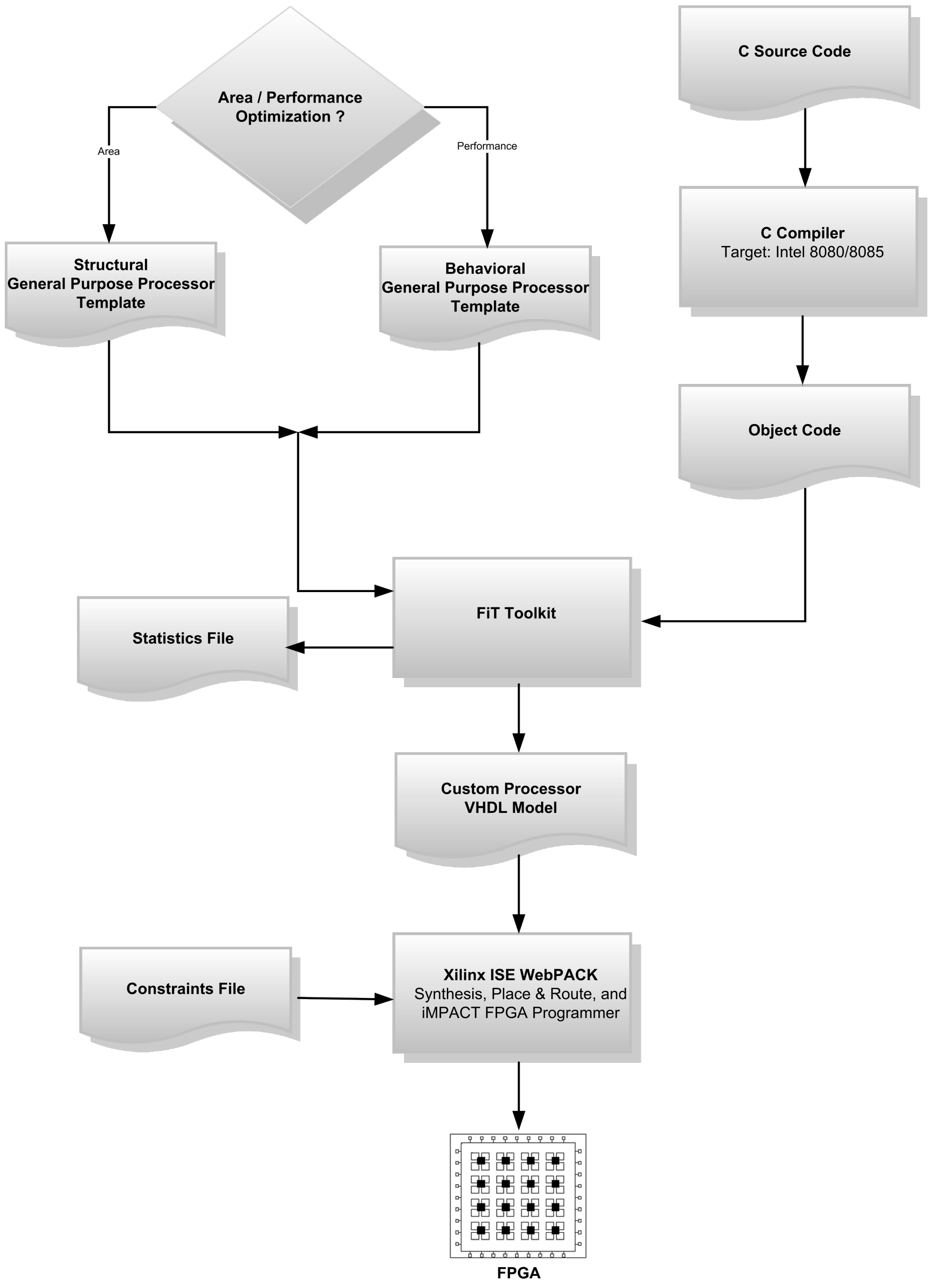
The second general-purpose processor template type, which we refer to as the behavioral template, is generated from a behavioral description of the physical processor. Using this template type with FiT, both the datapath and the control circuitry are customized to suit the ISA subset. The control module of the resulting custom soft processor is also hard-wired. The behavioral template takes advantage of the parallelism offered by FPGAs to produce a performance-optimized implementation of the processor. Using the behavioral template, the bus width can be adjusted so that the processor executes one instruction in each clock cycle. Furthermore, when there are no data dependencies in the instructions and the bus is appropriately sized, the performance-optimized processor can be designed to execute multiple instructions concurrently.

5.2.2 Operation

Before using FiT to generate a custom soft processor, the applications that will execute on the custom soft processor need to be compiled using a compiler that is targeted at a general-purpose processor. This general-purpose processor's ISA is a superset of the custom soft processor's ISA. In this study, the general-purpose processor is the k85 microprocessor which is a soft processor that implements all of the Intel 8080/8085 ISA.

Fig. 5.3 shows the data flow in and out of the FiT toolkit. For each application binary file, FiT disassembles the file and extracts the unique opcodes used into an application instruction set. A set union of all the application instruction sets is performed to produce the custom soft processor instruction set. The custom soft processor instruction set is used together with the general-purpose processor template to produce a VHDL model of the custom soft processor. The Xilinx ISE WebPACK tool [15] is used to synthesize, place & route, and program the FPGA with the custom soft processor.

FiT has a command line *–no-disassemble* option which causes the toolkit to bypass the disassembling of the application binary files and to instead read the opcodes needed for the custom soft processor description from a text file. This option is used to generate custom soft processors for arbitrary sets of opcodes and is especially useful for the processor verification.



Area / Performance Optimization ?
Area
Performance
Structural General Purpose Processor Template
Behavioral General Purpose Processor Template
C Source Code
C Compiler Target: Intel 8080/8085
Object Code
FiT Toolkit
Statistics File
Custom Processor VHDL Model
Constraints File
Xilinx ISE WebPACK Synthesis, Place & Route, and iMPACT FPGA Programmer
FPGA

Figure 5.3: The FiT toolkit flow diagram.

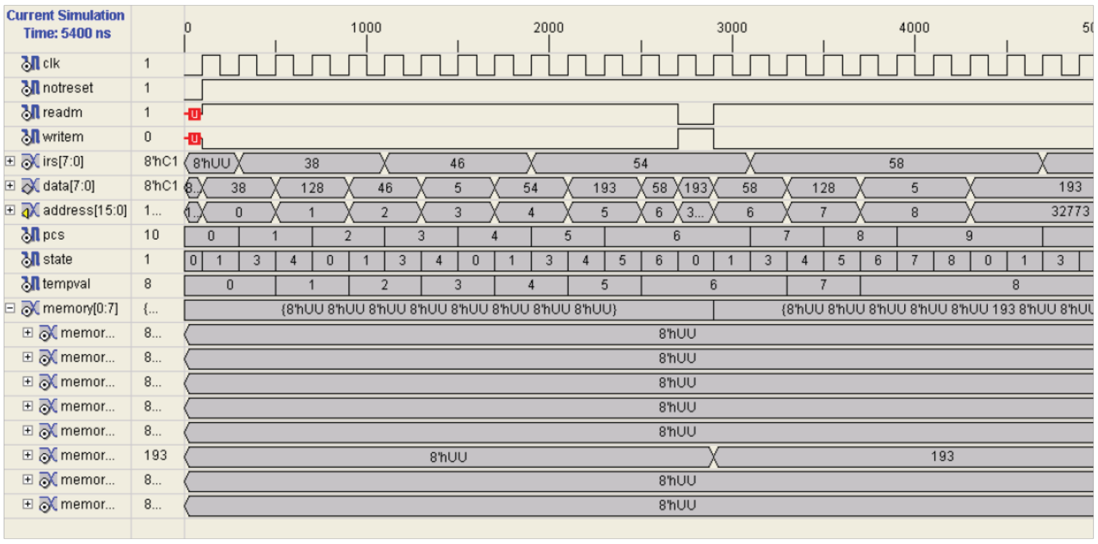


Figure 5.4: Timing diagram for the Processor, ROM, and SRAM modules.

5.2.3 Verification

The operations of the custom soft processors were verified using VHDL test-benches that produced timing diagrams, as well as using the visual output from the applications when available. Fig. 5.4 shows a timing diagram for the micro-controller consisting of the Processor, ROM and SRAM modules in Annexure A. This was generated using the Xilinx ISE Simulator.

Fig. 5.5 shows a microcontroller on the FPGA that has its ROM loaded with the ‘Hello World!’ application and a soft video controller. The code in the ROM is executed by the custom soft processor and the results are displayed on the monitor.

5.3 Benchmarks

The C source code for the benchmark applications, listed in Table 5.1, is available in [61] and was compiled using the Amsterdam Compiler Kit (ACK) [62] with the Intel 8080 ISA as the target. The 8080 structural and behavioral templates were used to generate an area-optimized custom soft processor and a performance-

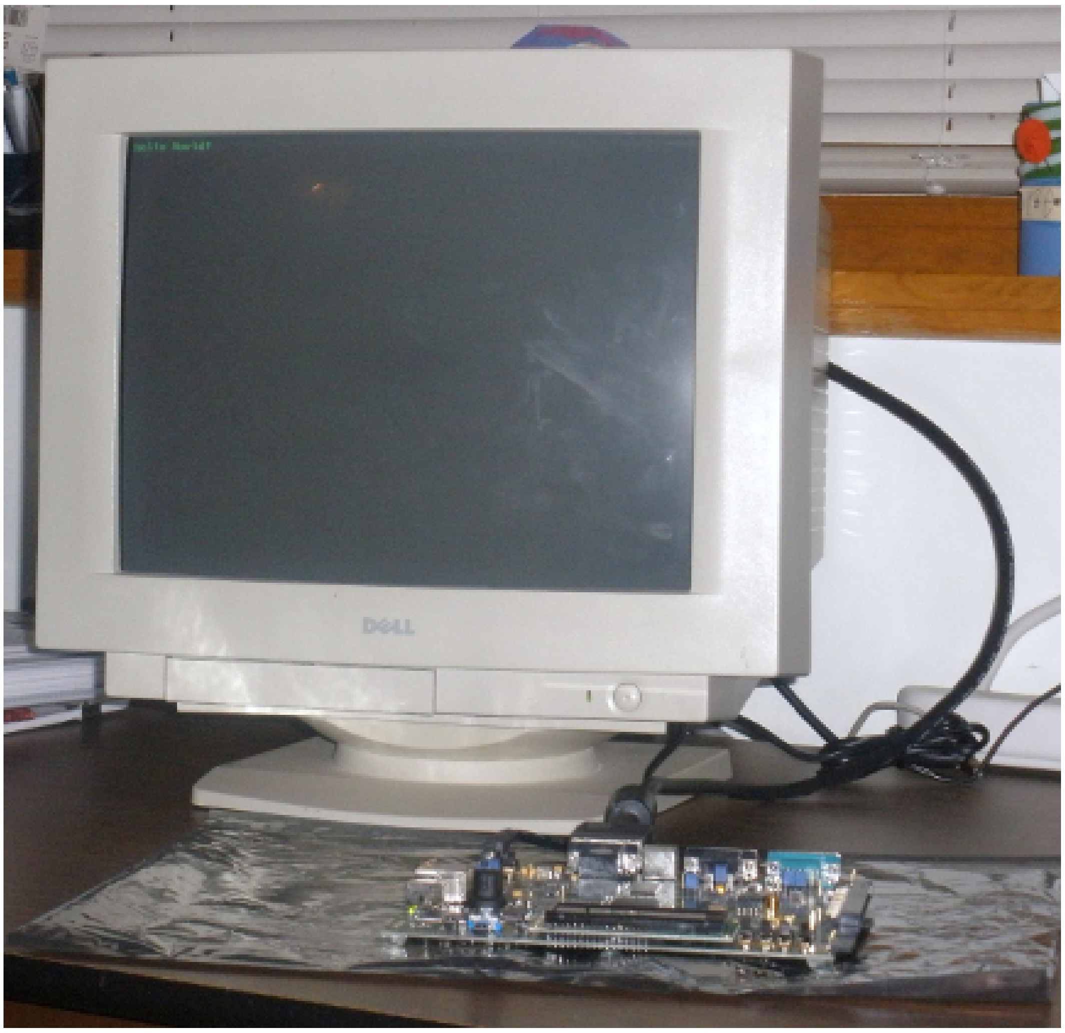


Figure 5.5: Running a ‘Hello World!’ application.

optimized custom soft processor, respectively, for each benchmark application.

The core density measure presented in the first part of this work only took into account the presence of each unique opcode in the application instruction set. In this part we also look at the length, in bytes, and the clock cycles for the instruction associated with each unique opcode in the application instruction set. We put together the instruction length (IL) index and instruction cycle (IC) index and these are defined as follows: Given an application instruction set A, let l_k and t_k represent the length and clock cycle, respectively, for the instruction associated with the k-th opcode in A then,

$$IL\ index = \sum_{k=1}^{|A|} l_k \tag{5.1}$$

and

$$IC\ index = \sum_{k=1}^{|A|} t_k \tag{5.2}$$

5.4 Results

Using FiT with the *–no-disassemble* option and an input file with all the opcodes for the Intel 8080 ISA, we obtain the results in Table 5.2. These results represent the resources required to implement a complete Intel 8080-compatible soft processor and serve as the base for the core densities of the benchmark applications. The basic reconfigurable unit in an FPGA is a look-up table (LUT). In this study

Table 5.1: FiT Benchmark Applications.

Application	**Description**
anneal	Traveling salesman problem by simulated annealing
fitexy	Fit data to a straight line, errors in both x and y
mglin	Linear elliptic PDE solved by multigrid method
pearsn	Pearson's correlation between two data sets
spctrm	Power spectrum estimation using FFT

we used the number of LUTs needed to implement a design on an FPGA as a measure of the resources needed to implement said design.

Table 5.2: Results for the complete 8085 ISA opcode set.

Unique Opcodes in 8080 ISA	244
Instruction Length (IL)-index	314
Instruction Cycles (IC)-index	383
LUTs used - Structural Template	1047
LUTs used - Behavioral Template	3440

The number of unique opcodes in the Intel 8080 is comparable to that of the Java Virtual Machine as shown in Table 3.1, and as a result, the core densities for these two processors are similar. The performance-optimized soft processor is on average 3.3 times larger than the area-optimized one, based on the number of LUTs required to implement either type of soft processor for the benchmarks.

5.4.1 Area-Optimized Custom Soft Processors

Fig. 5.6 shows the core densities of the area-optimized custom soft processors for the benchmarks.

Based on these results and the average core densities in Table 5.3, we observe that the average actual core density does not match the average core density for the unique opcodes. We also note that the actual core density does not scale with the sizes of the benchmark application instruction sets. This is contrary to what is predicted using Equation (3.6). Several factors may account for this discrepancy. Below are two possible causes.

- The resources required to implement the datapath are equal to or greater than those required to implement the controller. As a result, our decision to only customize the controller then produces custom soft processors that do not scale. Custom soft processors do not scale when their actual core densities approach 1, regardless of the application instruction sets as seen in Fig. 5.6.

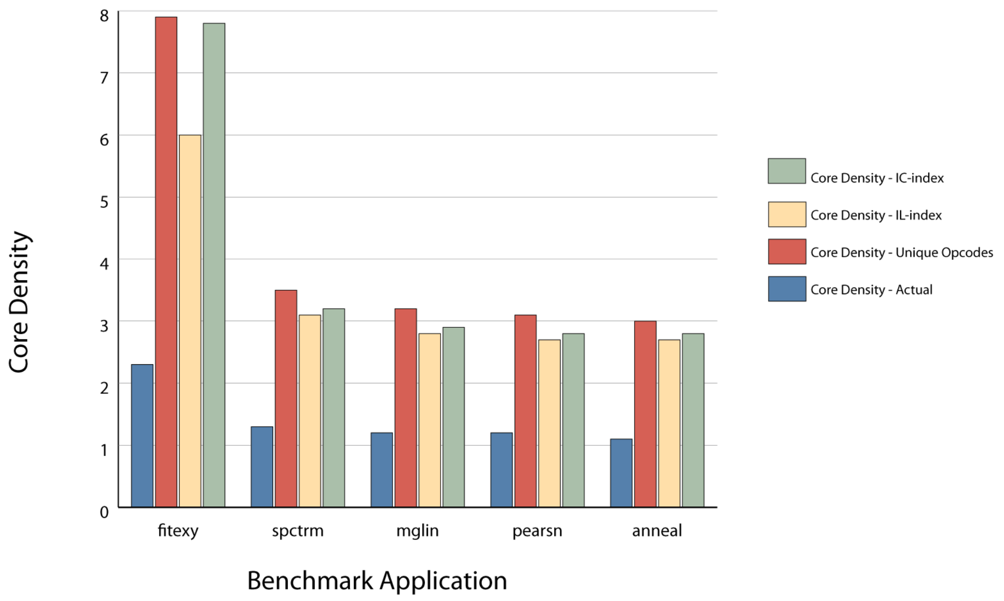


Figure 5.6: Core densities for area-optimized custom soft processors.

- The structural template uses a MUX-based design that saves on area at the expense of speed and flexibility. Since the complete Intel 8080 ISA has 244 unique opcodes and only 14 registers, it is possible that very small subsets of the unique opcodes will use up all of the available registers. This will result in the actual core densities approaching 1 in all of those cases.

A customizable datapath and controller for the structural template will need to be designed to test if either of the factors mentioned above is the cause of the contradiction. Based on the results from SPREE [44], it would appear that the contradiction is due to the fact that the datapath was not being customized.

Table 5.3: Average core densities of area-optimized custom soft processors.

Core Density	**Average**
Actual	1.4
Unique Opcodes	4.1
IL-index	3.5
IC-index	3.9

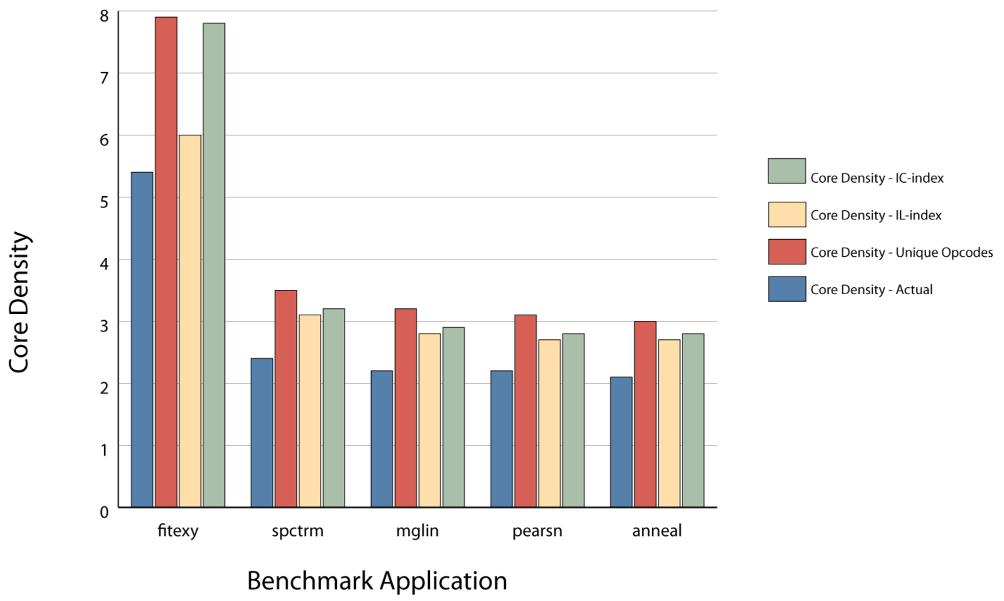


Figure 5.7: Core densities for performance-optimized custom soft processors.

5.4.2 Performance-Optimized Custom Soft Processors

The core densities of the performance-optimized custom soft processors for the benchmarks are shown in Fig. 5.7. We see that the actual core density of the performance-optimized custom soft processors scales linearly with the core density based on the unique opcodes. This result is in line with Equation (3.6). The actual core density of the performance-optimized custom soft processors also scales linearly with the other two core densities.

Table 5.4 shows the average core densities of the performance-optimized custom soft processors. We observe that the average IL-index core density more closely approximates the average actual core density.

Based on these results, we conclude that a core density measure based on the unique opcodes used provides a good approximation of resource utilization. A better approximation of resource utilization is provided by a core density measure that is based on the IL-index, that is on a combination of the unique opcodes used and their bit-lengths.

5.5 Application

In this section we demonstrate two applications of the FiT toolkit. First, we show how with some minor modifications, the FiT toolkit was modified to generate custom soft microcontrollers or SoCs. Fig. 5.8 is based on the custom 8080-compatible soft processors. The following modifications were made to the FiT

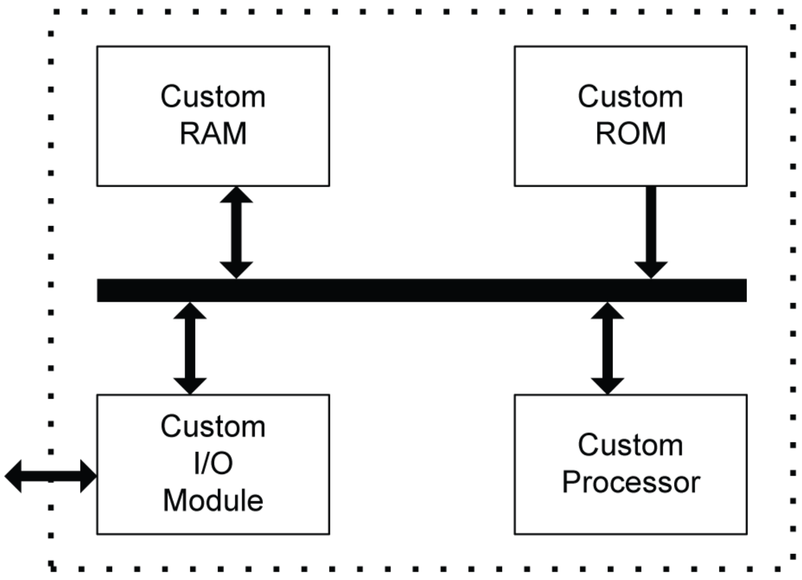


Figure 5.8: Custom 8085-based microcontroller.

toolkit.

1. As part of the disassembly of the application executables, a VHDL model of the ROM with only the instructions in the application executable was

Table 5.4: Average core densities of performance-optimized custom soft processors.

Core Density	**Average**
Actual	2.9
Unique Opcodes	4.1
IL-index	3.5
IC-index	3.9

created.

2. If any of the disassembled instructions was an IO instruction, then a VHDL model of the 8080-based IO module that implements only the ports required by the instructions was created.

3. A 256 kilobyte dual-port RAM VHDL model was created.

4. The three models above together with the custom soft processor model were combined into a single model for the microcontroller.

Next, we combined the FiT toolkit together with a C compiler and a third-party FPGA synthesis and place & route toolkit to create a C-to-silicon compiler, as shown in Fig. 5.9. Although we completed this step manually, it can however be scripted using the TCL scripting language, which is compatible all of the three components. The C-to-silicon compiler took a C source file and compiled it into an executable file. FiT created a VHDL model of the custom microcontroller with a custom soft processor tailor-made for the executable file and the executable file was loaded into the custom ROM. The third-party tool, in this case the Xilinx ISE WebPACK, synthesized the microcontroller's VHDL model and programmed it into the FPGA.

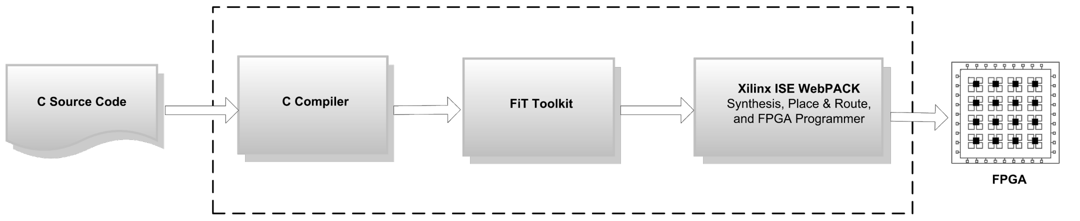


Figure 5.9: C-to-silicon Compiler.

5.6 Significance of this Study

In this study we presented a tool, FiT, for automatically generating custom soft processors which is supported by a resource usage theory. We showed that the custom soft processors can be optimized for area or for performance. The resource usage results of the performance-optimized custom soft processors are in line with the resource usage theory's predictions. Based on the results, we suggested an update to the theory to make it more consistent with the observed results. We also gave possible reasons as to why the results for the area-optimized custom soft processors were not as predicted by the theory.

Two applications of the FiT toolkit were presented. The first showed how a custom microcontroller can be generated by adding custom ROM, RAM and I/O modules to the custom soft processor generated using FiT. Second, we showed how a C-to-silicon compiler was assembled with FiT as the central component.

To the best of our knowledge, this work is the first that presents an automated toolkit, FiT, for generating custom soft processors with only the object code as input. FiT does not require its users to learn a new programming language or to use only a subset of an existing programming language. Furthermore, the application on which a custom soft processor is based does not need to be profiled, as a first step, to identify any hot spots that are candidates for optimization. We believe that FiT is an ESL tool that helps address most of the non-technical design issues mentioned above.

Chapter 6

Conclusion and Future Research

6.1 Summary

An assertion that modern day microprocessors use their resources inefficiently was made in Chapter 3. The core density metric to measure this inefficiency was proposed. The core density metric was shown to be related to a measure of instruction set usage that was proposed earlier by other researchers. Subset processors were proposed as a more efficient way to utilize a microprocessor's resources. Experiments were undertaken to quantify, in terms of core density, how efficiently modern day microprocessor utilize their resources. The processors studied were the JVM (if realized in hardware), MIPS, PowerPC, Intel x86, and Intel x86-64. The primary results of these experiments were:

- The larger the instruction set, the more inefficiently the processor utilizes its resources (the processor has a core density). This result was independent of the processor's design philosophy. That is, it did not matter whether the processor was a CISC design or a RISC design, nor if it was a virtual or a physical processor.
- Neither the choice of compiler nor the compiler optimizations that were used to generate the benchmark applications changed the result mentioned above regarding the impact of the instruction set size on processor resource

utilization.

A theoretical formalism to describe the subset processors, in particular, and computer architecture, in general, was developed in Chapter 4.

The FiT toolkit has been constructed and its operation was described in Chapter 5. FiT is an application that automatically generates VHDL models of custom soft processors from an application binary file. The custom soft processor VHDL models generated by FiT were then implemented on a Xilinx FPGA using the FPGA vendor's synthesis tools. The resulting synthesized processor designs were then evaluated and compared to the assertions and relationships derived in Chapter 3. The primary results were:

- Area-optimized soft processors are on average 3.3 times smaller (i.e. use fewer transistors) than performance-optimized soft processors.
- The number of unique opcodes is linearly proportional to the hardware resources used by the microprocessor for the performance-optimized soft processors.
- The number of unique opcodes is not linearly proportional to the hardware resources used by the microprocessor for the area-optimized soft processors.
- A core density metric that comprises the number of unique opcodes, used together with the bit length of the used opcodes more accurately measures a processor's resource utilization than the originally proposed core density metric, which only uses the number of unique opcodes.

By combining the FiT toolkit with an ANSI C compiler and the Xilinx ISE WebPACK, an unconstrained C-to-silicon compiler has been constructed.

6.2 Contributions

Our research makes the following contributions:

- We develop a model of computer architecture that extends the traditional von Neumann *program-data* model by storing a hardware *architecture* description of the computer, along with the *program* that it will execute and the *data* that it will use.

- We confirm the low instruction set usage rate results, found by several other researchers, through experiments that were broader than those done by most of the earlier researchers. Our experiments have as variables the: benchmark applications, target microprocessor architectures, compilers, and compiler optimization options.

- We propose the *core density* metric to measure the hardware resource utilization of an application on a given microprocessor platform. Using the instruction set usage experiments, we show that current microprocessor architectures result in high core densities.

- We present approaches to lowering the core density to its optimal value of 1.

- We contribute an implementation of an application-specific processor generator toolchain, called *FiT*, whose output processors all have a core density of 1.

- We show how the combination of the FiT toolchain with a traditional ANSI C compiler and a vendor-specific synthesis tool produces an *unconstrained C-to-silicon compiler.*

6.3 Future Research

The research work presented in this thesis can be extended in the following areas:

6.3.1 Extending the Study on Instruction Set Usage

Further research is needed to determine the optimum number of instructions that will be needed to constrain a compiler, so as to minimize the recompiling effort and the size of the generated code, while maximizing the performance of the resulting processor. The opcode statistics for the C benchmark applications were gathered using source code, another research approach would have been to compile the benchmarks applications into an executable (binary) file and then disassemble the binary files to collect the opcode statistics. This is the approach used by the FiT toolkit.

6.3.2 Extending the Computer Architecture Model

Further research is needed to show that the triplet model can be used to describe modern processors, like the Intel Pentium processor. A software application to store, input and visually manipulate triplet models is needed as designs get more complex.

6.3.3 Extending the FiT Toolchain

Future work includes investigating larger processors and generating a more customizable structural template. Another avenue will be to explore ways to automatically generate networks of custom soft processors on a single FPGA chip. Future research may target popular programming languages in order to create other high-level-language-to-silicon compilers, such as Fortran-to-silicon compiler or Java-to-silicon compiler.

6.4 List of Scientific Publications

The research work presented in this thesis has been presented, discussed and published in various national and international forums as indicated below.

1. C. Mutigwe, "Toward a Theory of Computer Architecture with Applications to Microprogramming and Virtualization," in *Proc. 10th Michael L. Gargano Annual Research Day*, Pace University, White Plains, USA, pp. A2.1-A2.7, May 2012. [Online]. Available: http://csis.pace.edu/ ctappert/srd2012/a2.pdf

2. C. Mutigwe, J. Kinyua, and F. Aghdasi, "Instruction Set Usage Analysis for Application-Specific Systems Design," *2nd International Conference on Emerging Trends in Computer & Information Technology (ICETCIT 2013)*, Kuala Lumpur, Malaysia, 15-16 January 2013.

3. C. Mutigwe, J. Kinyua, and F. Aghdasi, "A Model of Computer Architecture with Applications," *4th International Conference on Computer Modeling and Simulation (ICCMS 2013)*, Rome, Italy, 24-25 February 2013.

4. C. Mutigwe, J. Kinyua, and F. Aghdasi, "Instruction Set Usage Analysis for Application-Specific Systems Design, *International Journal of Information Technology and Computer Science*, vol. 7, no. 2, pp. 99-103, 2013, ISSN: 2091-1610.

5. C. Mutigwe, J. Kinyua, and F. Aghdasi, "A Model of Computer Architecture with Applications, *International Journal of Computer Theory and Engineering*. (to appear in 2013).

6. C. Mutigwe, J. Kinyua and F. Aghdasi, "On the Design of Application-specific Processors using the Core Density Metric," *ACM Transactions on Computer Systems*, Submitted.

References

[1] A. W. Burks, H. H. Goldstine, and J. von Neumann, *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*, 2nd ed. Princeton, USA: Institute for Advanced Study, 1947.

[2] R. Wilson. (2011, Nov. 20,) C to Silicon. Really? EE Times. [Online]. Available: http://www.eetimes.com/discussion/other/4213003/C-to-silicon--Really [Nov. 23, 2011].

[3] A. Wolfe. (2008, Jul. 16,) Steve Jobs decision behind iPhone apps 'achilles' heel. InformationWeek. [Online]. Available: http://www.informationweek.com/global-cio/interviews/steve-jobs-decision-behind-iphone-apps-a/229210628 [Apr. 6, 2011].

[4] J. Diaz. (2010, Apr. 8,) How multitasking works in the new iPhone OS 4.0. GIZMODO. [Online]. Available: http://gizmodo.com/5512656/how-multitasking-works-in-the-new-iphone-os-40 [Oct. 12, 2011].

[5] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proc. IEEE*, vol. 95, no. 2, pp. 356–373, Feb. 2007.

[6] R. Charette, "This car runs on code," *IEEE Spectrum*, Feb. 2009. [Online]. Available: http://www.spectrum.ieee.org/feb09/7649 [Mar. 15, 2011].

[7] A. J. McAllister, "The case for teaching legacy systems modernization," in *Proc. 8th Intl. Conf. on Information Technology: New Generations (ITNG'11)*, Las Vegas, USA, 2011, pp. 251–256.

[8] H. M. Sneed, “An incremental approach to system replacement and integration,” in *Proc. 9th European Conf. on Software Maintenance and Reengineering*, Manchester, UK, 2005, pp. 196–206.

[9] D. E. Bernholdt, J. Nieplocha, and P. Sadayappan, “Raising level of programming abstraction in scalable programming models,” in *Proc. 10th IEEE Intl. Conf. on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, Madrid, Spain, 2004, pp. 76–84.

[10] S. Jayadevappa, R. Shankar, and I. Mahgoub, “A comparative study of modelling at different levels of abstraction in system on chip designs: a case study,” in *Proc. IEEE Computer Society Annual Symp. on VLSI*, Tampa, USA, 2004, pp. 52–58.

[11] M. A. Cohen, J. Eliashberg, and T.-H. Ho, “New product development: The performance and time-to-market tradeoff,” *Management Science*, vol. 42, no. 2, pp. 173–186, Feb. 1996.

[12] J. N. Amaral, M. Buro, R. Elio, J. Hoover, I. Nikolaidis, M. Salavatipour, L. Stewart, and K. Wong. (2010, Sep.) About computing science research methodology. [Online]. Available: http://webdocs.cs.ualberta.ca/~c603/readings/research-methods.pdf [Oct. 21, 2012].

[13] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[14] “SPECjvm2008 Benchmarks,” Standard Performance Evaluation Corp., 2008. [Online]. Available: http://www.spec.org/jvm2008/docs/benchmarks/ [May. 7, 2012].

[15] ISE WebPACK Design Software. Xilinx Inc. [Online]. Available: http://www.xilinx.com/tools/webpack.htm [Aug. 18, 2008].

[16] S. Borkar and A. A. Chien, “The future of microprocessors,” *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[17] M. J. Bass and C. M. Christensen, "The future of the microprocessor business," *IEEE Spectrum*, vol. 39, no. 4, pp. 34–39, 2002.

[18] P. Ienne and R. Leupers, "From prêt-à-porter to tailor-made," in *Customizable Embedded Processors*, P. Ienne and R. Leupers, Eds. San Franscisco, USA: Morgan Kaufmann, 2007, ch. 1, pp. 3–9.

[19] C. C. Foster, R. H. Gonter, and E. M. Riseman, "Measures of op-code utilization," *IEEE Transactions on Computers*, vol. 20, no. 5, pp. 582–584, 1971.

[20] T. L. Adams and R. E. Zimmerman, "An analysis of 8086 instruction set usage in MS DOS programs," *ACM SIGARCH Computer Architecture News*, vol. 17, no. 2, pp. 152–160, 1989.

[21] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco, USA: Morgan Kaufmann, 1996.

[22] ——, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, USA: Morgan Kaufmann, 2002.

[23] I. J. Huang and T. C. Peng, "Analysis of x86 instruction set usage for DOS/Windows applications and its implication on superscalar design," in *Proc. Intl. Conf. on Computer Design (ICCD'98)*, Austin, USA, 1998, pp. 566–573.

[24] M. El-Kharashi, F. ElGuibaly, and K. Li, "A quantitative study for Java microprocessor architectural requirements. Part II: High-level language support," *Microprocessors and Microsystems*, vol. 24, no. 5, pp. 237–250, 2000.

[25] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 87–101, 1964.

[26] T. Mudge, "Strategic directions in computer architecture," *ACM Computing Surveys*, vol. 28, no. 4, pp. 671–678, 1996.

[27] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco, USA: Morgan Kaufmann, 2011.

[28] R. F. Albrecht, "Modelling of computer architectures," in *Proc. 1st Intl. Conf. on Massively Parallel Computing Systems*, Ischia, Italy, 1994, pp. 434–442.

[29] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. San Francisco, USA: Morgan Kaufmann, 2005.

[30] W. Chen, W. Xu, Z. Wang, Q. Dou, Y. Wang, B. Zhao, and B. Wang, "A formalization of an emulation based co-designed virtual machine," in *Proc. 5th Intl. Conf. Innovative Mobile and Internet Services in Ubiquitous Computing*, Seoul, South Korea, 2011, pp. 164–168.

[31] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[32] M. Sima, S. Vassiliadis, S. Cotofana, J. T. van Eijndhoven, and K. A. Vissers, "Field-programmable custom computing machines - a taxonomy," in *Proc. Reconfigurable Computing Is Going Mainstream, 12th Intl. Conf. Field-Programmable Logic and Applications*, Montpellier, France, 2002, pp. 79–88.

[33] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[34] W. Stallings, "Reduced instruction set computer architecture," *Proceedings of the IEEE*, vol. 76, no. 1, pp. 38–55, 1988.

[35] P. Bhosle and H. K. Moorthy, "FPGA implementation of low power pipelined 32-bit RISC processor," *International Journal of Innovative Technology and Exploring Engineering*, vol. 1, no. 3, pp. 66–71, 2012.

[36] P. Clarke. European Space Agency launches free SPARC-like core. [Online]. Available: http://www.eetimes.com/electronics-news/4151500/European-Space-Agency-launches-free-Sparc-like-core [Mar. 6, 2000].

[37] B. Valli, A. U. Kumar, and B. V. Bhaskar, "FPGA implementation and functional verification of a pipelined MIPS processor," *International Journal of Computational Engineering Research*, vol. 2, no. 5, pp. 1559–1561, 2012.

[38] M. Imran and K. Ramananjaneyulu, "FPGA implementation of a 64-bit RISC processor using VHDL," *International Journal of Engineering Research and Applications*, vol. 2, no. 3, pp. 2544–2549, 2012.

[39] S.-L. L. Lu, P. Yiannacouras, T. Suh, R. Kassa, and M. Konow, "A desktop computer with a reconfigurable Pentium," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 1, pp. 1–15, 2008.

[40] M. K. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: Survey and issues," in *Proc. 14th Intl. Conf. on VLSI Design*, Bangalore, India, 2001, pp. 76–81.

[41] G. Bilski, S. Mohan, and R. Wittig, "Designing soft processors for FPGAs," in *Customizable Embedded Processors*, P. Ienne and R. Leupers, Eds. San Franscisco, USA: Morgan Kaufmann, 2007, ch. 17, pp. 425–441.

[42] (2011, Jan.) NIOS II custom instruction user guide. Altera Corp. [Online]. Available: http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf [Jul. 3, 2012].

[43] W. R. Bush, "The high-level synthesis of microprocessors using instruction frequency statistics," Ph.D. dissertation, University of California at Berkeley, 1992.

[44] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *Proc. ACM/SIGDA 14th Intl. Symp. on Field Programmable Gate Arrays*, Monterey, USA, 2006, pp. 201–210.

[45] M. J. Flynn, J. D. Johnson, and S. P. Wakefield, "On instruction sets and their formats," *IEEE Transactions on Computers*, vol. 34, no. 3, pp. 242–254, 1985.

[46] B. Kagan, *Computers, Computer Systems and Networks*, 2nd ed. Moscow, Russia: Mir Publishers, 1988.

[47] M. Wisniewska, M. Adamski, R. Wisniewski, and W. A. Halang, "Application of hypergraphs in microcode length reduction of microprogrammed

controllers," in *Proc. 2nd Intl. Workshop on Nonlinear Dynamics and Synchronization (INDS'09)*, Klagenfurt, Austria, 2009, pp. 106–109.

[48] S. Jian-Lun, "Researches on the technology of high performance microprogrammed control," in *Proc. Intl. Conf. on Educational and Information Technology (ICEIT'10)*, San Francisco, USA, 2010, pp. V2:20–V2:24.

[49] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[50] D. W. Jones, "The Ultimate RISC," *ACM SIGARCH Computer Architecture News*, vol. 16, no. 3, pp. 48–55, 1988.

[51] G. A. Blaauw and F. P. Brooks, *Computer Architecture: Concepts and Evolution.* Reading, USA: Addison-Wesley, 1997.

[52] M. J. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design.* Boston, USA: Jones and Bartlett, 1995.

[53] C. Mutigwe, "Toward a theory of computer architecture with applications to microprogramming and virtualization," in *Proc. 10th Michael L. Gargano Annual Research Day*, Pace University, White Plains, USA, 2012, pp. A2.1–A2.7.

[54] T. G. Rauscher and P. M. Adams, "Tutorial: microprogramming and firmware engineering." Piscataway, USA: IEEE Press, 1989, ch. Microprogramming: a tutorial and survey of recent developments, pp. 2–20.

[55] M. M. Mano, *Digital Logic and Computer Design*, eastern economy ed. New Delhi, India: Prentice-Hall, 1996.

[56] A. Klaiber, "The technology behind Crusoe processors," Transmeta Corp., Tech. Rep., 2000.

[57] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proc. Intl. Symp. Code Generation and Optimization: Feedback-directed and Runtime Optimization*, San Francisco, USA, 2003, pp. 15–24.

[58] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[59] J. Trajkovic, S. Abdi, N. G., and D. D. Gajski, “Automated generation of custom processor core from C code,” *Journal of Electrical and Computer Engineering, 26 pages*, 2012.

[60] *8080/8085 Assembly Language Programming Manual*, Intel Corp., 1978.

[61] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge, UK: Cambridge University Press, 1992.

[62] The Amsterdam Compiler Kit. [Online]. Available: http://tack.sourceforge.net/ [Feb. 17, 2008].

ANNEXURE A

.1 Processor Module

This section shows hand-crafted VHDL code for a soft processor that is a subset processor of the k85 microprocessor. This only implements 5 opcodes. The FiT toolkit automatically generates for the performance-optimized soft processors, VHDL code (model) similar to the one presented here. The area-optimized soft processor VHDL models are automatically generated in a slightly different manner since they are assembled at the functional unit level. A soft microcontroller is generated by combiining this processor module, together with the ROM module in Annexure A.2 and the SRAM module in Annexure A.3.

```
------------------------------------------------------

-- Engineer:           Charles Mutigwe

-- Create Date:        7:36:25 06/07/2009

-- Design Name:        8080_Computer

-- Module Name:        PROCESSOR

-- Project Name:       D.Tech Research Project

-- Target Devices:     Xilinx xc3s500e-5fg320

--                     (Spartan-3E Starter Kit)

-- Tool versions:      Xilinx WebPACK ISE 9.2.04i
--                      Xilinx ISE Simulator
-- Description:
-- This module setup the  8080 Datapath and Control Logic
-- that implements five opcodes.
--
-- Dependencies:
--
-- Revision:            0.01 - File Created
--
-- Additional Comments:
--
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity PROCESSOR is
       -- size of EPROM
       generic (numInstr : integer := 9);
  port (clk :            in  std_logic;
       notReset :        in  std_logic;
       int :             in  std_logic;
       data :            inout  std_logic_vector (7 downto 0);
       address :         out  std_logic_vector (15 downto 0);
       intAck :          out  std_logic;
       readM :           out  std_logic;
       writeM :          out  std_logic;
       readIO :          out  std_logic;
       writeIO :         out  std_logic);
end PROCESSOR;
architecture Behavioral of PROCESSOR is
signal state:   integer range 0 to 8 ;
-- Debugging Signals --
-- Program Counter signal
signal PCs:     integer range 0 to (numInstr - 1) := 0;
-- Instruction Register signal
signal IRs:     std_logic_vector(7 downto 0);
begin
     data_transform:            process(clk,notReset,int,state)
     variable addressv:         std_logic_vector(15 downto 0);
     variable datav:            std_logic_vector(7 downto 0);
     variable intAckv:          std_logic := '0';
     variable readMv:           std_logic := '0';
     variable writeMv:          std_logic := '0';
     variable readIOv:          std_logic := '0';
     variable writeIOv:         std_logic := '0';
     -- Instruction Register
     variable IR:               std_logic_vector(7 downto 0);
     -- match EPROM
     variable PC:       integer range 0 to (numInstr - 1):= 0;
```

```
   -- A Register
   variable A:                 std_logic_vector (7 downto 0);
   -- H Register
   variable H:                 std_logic_vector (7 downto 0);
   -- L Register
   variable L:                 std_logic_vector (7 downto 0);
   -- W Register
   variable W:                 std_logic_vector (7 downto 0);
   -- Z Register
   variable Z:                 std_logic_vector (7 downto 0);
begin
     if (notReset = '0') then
       state <= 0;
       PC := 0;
       addressv := (others => '0');
     elsif (rising_edge(clk)) then
       if (int = '1') then
         state <= 1;
        end if;
       -- initialize all outbound variables
       datav := (others => 'Z');
       if (PC < numInstr) then
         addressv := conv_std_logic_vector(PC, 16);
       end if;
       -- handle the opcodes
```

```
case state is
  when 0 =>    -- instruction fetch
    IR := data;
    PC := PC + 1;
    readMv := '1';
    writeMv := '0';
    state <= 3;
  when 2 =>    -- interrupt service routine
    PC := PC - 1;
    intAckv  := '1';
    IR := conv_integer(data);
    state <= 3;
  when 3 =>    -- wait to read/write to SRAM
    state <= 4;
  when 4 =>    -- state s2
    -- instruction decode
    case conv_integer(IR) is
      when 0 =>        -- NOP
        -- do nothing
        readMv := '1';
        state <= 0;
      when 38 =>       -- MVI H data
        H := data;
        PC := PC + 1;
        readMv := '1';
```

```
          state <= 0;
        when 46 =>        -- MVI L data
          L := data;
          PC := PC + 1;
          readMv := '1';
          state <= 0;
        when 54 =>        -- MVI M data
          W := data;
          PC := PC + 1;
          readMv := '1';
          state <= 5;
        when 58 =>        -- LDA addr
          Z := data;
          PC := PC + 1;
          readMv := '1';
          state <= 5;
        when others =>
          -- do nothing
          state <= 0;
      end case;
    when 5 =>    -- wait to read/write to SRAM
      state <= 6;
    when 6 =>    -- state s3
      -- instruction decode
      case conv_integer(IR) is
    when 54 =>       -- MVI M data
      datav := W;
      addressv := H & L;
      writeMv := '1';
      readMv := '0';
      state <= 0;
    when 58 =>       -- LDA addr
      W := data;
      PC := PC + 1;
      readMv := '1';
      state <= 7;
    when others =>
      -- do nothing
      state <= 0;
  end case;
when 7 =>    -- wait to read/write to SRAM
  state <= 8;
when 8 =>    -- state s4
  -- instruction decode
  case conv_integer(IR) is
    when 58 =>       -- LDA addr
      A := data;
      addressv := Z & W;
      readMv := '1';
      state <= 0;
```

```
              when others =>
                -- do nothing
                state <= 0;
            end case;
          when others =>
            state <= 0;
        end case;
        -- set the final values of the outbound signals
        if (writeMv = '1') then
          data <= datav;
        else
          data <= (others => 'Z');
        end if;
        address <= addressv;
        IRs <= IR;
        PCs <= PC;
        intAck <= intAckv;
        readM <= readMv;
        writeM <= writeMv;
        readIO <= readIOv;
        writeIO <= writeIOv;
      end if;
end process data_transform;
end Behavioral;
```

.2 ROM Module

This section shows hand-crafted VHDL code for the ROM module which is loaded with the executable file to be used by the Processor module in Annexure A.1. The FiT toolkit automatically generates the VHDL model for the application-loaded ROM module when building the microcontroller.

```
-----------------------------------------------------
-- Engineer:            Charles Mutigwe
-- Create Date:         14:36:25 04/09/2008
-- Design Name:         8080_Computer
-- Module Name:         EPROM
-- Project Name:        D.Tech Research Project
-- Target Devices:      Xilinx xc3s500e-5fg320
--                      (Spartan-3E Starter Kit)
-- Tool versions:       Xilinx WebPACK ISE 9.2.04i
--                      Xilinx ISE Simulator
-- Description:
--  This module implements the ROM
--
-- Dependencies:
--
-- Revision:            0.01 - File Created
--                      0.02 - asynchronous ROM tests
--
-- Additional Comments:
--
```

```
-------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity EPROM is
generic (
       bits :       integer := 8; -- # of bits in a word
       words :      integer := 9;  -- # of words in ROM
       databus :    integer := 15  -- address bus width
       );
  port (
       nCS :         in  std_logic;
       nOE :         in  std_logic;
       address :     in std_logic_vector(databus-1 downto 0);
       data :        out std_logic_vector(bits-1 downto 0)
       );
end EPROM;
architecture behavioral of EPROM is
signal tempval: integer range 0 to words-1;
type vector_array is array (0 to words-1) of \
     std_logic_vector(bits-1 downto 0);
constant memory: vector_array :=
( -- for the test
"00100110", --(26H/38) MVI H, data
"10000000", --(80H/128)
"00101110", --(2EH/46) MVI L, data
"00000101", --(5H/5)
"00110110", --(36H/54) MVI M, data
"11000001", --(C1H/193)
"00111010", --(3AH/58) LDA addr
"10000000", --(80H/128)
"00000101" --(5H/5)
);
begin
tempval <= conv_integer(address) when (nCS = '0') \
     and (nOE = '0');
data <= memory(tempval) when (nCS = '0') and \
     (nOE = '0') else
     (others => 'Z');
end behavioral;
```

.3 SRAM Module

This section shows hand-crafted VHDL code for the SRAM to be used by the Processor module in Annexure A.1. The FiT toolkit automatically generates the VHDL model for the SRAM module when building the microcontroller.

```
------------------------------------------------------
-- Engineer:            Charles Mutigwe
-- Create Date:         14:36:25 04/09/2008
-- Design Name:         8080_Computer
-- Module Name:         SRAM
-- Project Name:        D.Tech Research Project
-- Target Devices:      Xilinx xc3s500e-5fg320
--                      (Spartan-3E Starter Kit)
-- Tool versions:       Xilinx WebPACK ISE 9.2.04i
--                      Xilinx ISE Simulator
-- Description:
--  This module implements the SRAM
--
-- Dependencies:
--
-- Revision:            0.01 - File Created
--                      0.02 - Synchronous RAM tests
--                         - Inferred Distributed/Block RAM
-- Additional Comments:
--
```

```
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity SRAM is
     -- # of bits per word
        generic (
       bits:       integer := 8;
       -- # of words in the memory
       words:      integer := 8;
       -- address bus width
       databus :   integer := 15
       );
  port (
       nCS :      in  std_logic;
       nOE :      in  std_logic;
       nWE :      in  std_logic;
       clk :      in  std_logic;
       address:   in std_logic_vector(databus-1 downto 0);
       data :     inout  std_logic_vector(bits-1 downto 0)
       );
end SRAM;
architecture Behavioral of SRAM is
signal temp: integer range 0 to words-1;
```

```
type vector_array is array (0 to words-1) of \
     std_logic_vector(bits-1 downto 0);
signal memory: vector_array;
begin
      temp <= conv_integer(address) when (nCS = '0');
      process(clk)
      begin
        if (rising_edge(clk)) then
          if (nOE = '1') and (nWE = '0') then
            if (nCS = '0') then
              memory(temp) <= data;
            end if;
          end if;
        end if;
      end process;

      -- LUT-based RAM
      data <= memory(temp) when (nOE = '0') and \
         (nWE /= '0') and (nCS = '0') else
      (others => 'Z');

end Behavioral;
```