View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Aquila Digital Community

The University of Southern Mississippi

The Aquila Digital Community

Dissertations

Fall 12-2012

Reification: A Process to Configure Java Realtime Processors

John Huddleston Heath

University of Southern Mississippi

Follow this and additional works at: https://aquila.usm.edu/dissertations

Part of the Computer Sciences Commons, and the Physics Commons

Recommended Citation

Heath, John Huddleston, "Reification: A Process to Configure Java Realtime Processors" (2012). *Dissertations*. 718.
https://aquila.usm.edu/dissertations/718

This Dissertation is brought to you for free and open access by The Aquila Digital Community. It has been accepted for inclusion in Dissertations by an authorized administrator of The Aquila Digital Community. For more information, please contact Joshua.Cromwell@usm.edu.

The University of Southern Mississippi

REIFICATION: A PROCESS TO CONFIGURE JAVA REALTIME PROCESSORS

by

John Huddleston Heath

Abstract of a Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

December 2012

ABSTRACT

REIFICATION: A PROCESS TO CONFIGURE JAVA REALTIME PROCESSORS

by John Huddleston Heath

December 2012

Real-time systems require stringent requirements both on the processor and the software application. The primary concern is speed and the predictability of execution times. In all real-time applications the developer must identify and calculate the worst case execution times (WCET) of their software. In almost all cases the processor design complexity impacts the analysis when calculating the WCET. Design features which impact this analysis include cache and instruction pipelining. With both cache and pipelining the time taken for a particular instruction can vary depending on cache and pipeline contents. When calculating the WCET the developer must ignore the speed advantages from these enhancements and use the normal instruction timings.

This investigation is about a Java processor targeted to run within an FPGA environment (Java soft chip) supporting Java real-time applications. The investigation focuses on a simple processor design that allows simple analysis of WCET. The processor design has no cache and no instruction pipeline enhancements yet achieves higher performance than existing designs with these enhancements.

The investigation centers on a process that translates Java byte codes and folds these translated codes into a modified Harvard Micro Controller (HMC). The modifications include better alignment with the application code and take advantage of the FPGA's parallel capability. A prototyped ontology is used where the top level categories defined by Sowa are expanded to support the process.

The proposed HMC and process are used to produce investigation results. Performance testing using the Sobel edge detection algorithm is used to compare the results with the only Java processor claiming real-time abilities.

COPYRIGHT BY

JOHN HUDDLESTON HEATH

2012

The University of Southern Mississippi

REIFICATION: A PROCESS TO CONFIGURE JAVA REALTIME PROCESSORS

by

John Huddleston Heath

A Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Approved:

____
Director

____

____

____

____

____
Dean of the Graduate School

December 2012

ACKNOWLEDGMENTS

First I would like to extend my most heartfelt thanks to Dr. Ray Seyfarth, my committee chair. If it were not for his patience, understanding and commitment to see me finish, I would have not finished this dissertation. Many times it would have been easier to quit than to keep going. Dr. Seyfarth provided the incentive and inspiration to keep me going. I also would like to thank Dr. Andrew Strelzoff for the vision, ambition and guidance he gave me. Also, Dr. Strelzoff helped me greatly in my research endeavors and I am truly thankful for his support.

Furthermore I would like to thank Dr. Chaoyang Zhang for his participation in my committee and for taking time out from his duties as director of the School of Computing. Another big thank you goes to Dr. Beddhu Murali. His wit and humor were always appreciated, although it was his critical views that made me a better student and researcher. Dr. Ras Pandey is another great professor that I would like to thank for taking the time to participate as a member of my committee. Dr. Pandey's teaching style and approach was one of the most interesting ones during my academic years. I have carried his approach and style in many classes I teach today.

Additionally and most affectionately, I want to thank my former office mate, classmate and friend, Silvia Preston. I want to thank Mrs. Preston and her family for their support and most of all for their friendship.

Many times we forget people that have made an impact in our academic career. Many times we overlook the hard work and support these folks give us. My special thanks go to Ms. Paulette Jackson, office manager for the School of Computing, and her staff Crystal McCaffrey, Sherry Smith-Davidson and student workers Gabby, Karen and Andrea. Our school is quite large and these ladies work extremely hard and this I will never forget.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS v

LIST OF ILLUSTRATIONS vii

LIST OF TABLES viii

1 INTRODUCTION **1**

2 THE PROBLEM **3**

3 REVIEW OF LITERATURE **9**

3.1 HIGH LEVEL LANGUAGE TO HARDWARE CIRCUITRY 10
3.2 JAVA SOFT CHIPS 13
3.3 Current Research 16

4 REIFICATION ARCHITECTURE **25**

4.1 Reification Concepts 25
4.2 Reification Ontology 34

5 FINDINGS **59**

5.1 Introduction 59
5.2 FPGA Usage 61
5.3 Simple Test 61
5.4 Sobel Edge Detection 64

6 Conclusions and Future Research **69**

6.1 Future research 71

APPENDIX

A VHDL INSTRCUTIONS AND BUBBLE SORT **73**

A.1 VHDL Instructions 73
A.2 Bubble Sort 81

BIBLIOGRAPHY **82**

LIST OF ILLUSTRATIONS

Figure

2.1 Basic notions concerning timing analysis of systems [34]. . . . 4

3.1 FPGA configuring an application circuit 20
3.2 FPGA And gate as an application circuit 21
3.3 CLB - containing 4 slices 22
3.4 Logic Cell [33] 22
3.5 Slice arrangement in a CLB [33] 23

4.1 Serial Communication Between Processors 28
4.2 Parallel Communication Between Processors 28
4.3 Design Folding 29
4.4 Harvard Micro Controller Function Design 32
4.5 Extension of Sows's intention category 36
4.6 Extension of Sowa's proposition category 38
4.7 Extension of Sowa's category 40
4.8 CMAP for the determine objective 41
4.9 Ontology view on knowledge base instances for translation 43
4.10 CMAP for the determine objective 45
4.11 instance view supporting the determine and resolve objective 47
4.12 the resolve objective 49
4.13 the model objective 51
4.14 the fold objective 55
4.15 measure supporting model objective 58
4.16 progress supporting model objective 58

5.1 Testing Setup 60
5.2 Soft Core Processors 61
5.3 Simple Test - demonstrates instruction folding 63
5.4 Bubble Sort Test During Start 64
5.5 Bubble Sort Test at Completion 64
5.6 Sobel Edge Detection Algorithm 66
5.7 Sobel observed WCET 67
5.8 Sobel byte codes for pixel 68

LIST OF TABLES

Table

5.1 Instruction timings 63

Chapter 1
INTRODUCTION

This study is about a process that logically synthesizes a Field Programmable Gate Array (FPGA) to support Java real-time applications on a Java soft chip processor (processor within an FPGA). Java, as many know, is too slow to meet the stringent timing demands for all real-time applications. The primary reason for Java being too slow is that Java is an interpreted language. Although many enhancements, such as just-in-time compilation, have given Java significant speed-ups, Java is not a popular choice for real-time applications.

In addition to performance requirements real time applications require stringent response times which are necessary to assure the correctness of the calculated responses [9]. In these systems the response time is just as important as the program results. Missing the response time invalidates the results. For example a chess program that calculates chess moves may be bounded by tournament rules to move within 20 seconds. Although this bound may differ from tournament to tournament the execution of the program's bound does not.

This program bound is the total time taken during the longest path though the programs instructions. This time represents the program's worst case execution time (WCET). Therefore, it is statically correct to say the program is validated for all tournaments where the move time is greater than the WCET.

The most notable of all the enhancements to Java is the Pico Java [3] processor. This processor implements the Java byte codes directly in hardware. With many other improve-

ments Java is now fast enough to support real-time demands; yet, Java still falls short as a language of choice for these applications. Specifically, the Pico Java processor implements an instruction pipe-line and provides caching which complicate timing analysis.

Given the capability of the FPGA to support coarse-grained parallel applications, the FPGA has become a popular choice for the design of time-critical applications. Many coarse-grained parallel applications are refered to as embarrassing parallel. Embarrassingly parallel [7] is a term that defines parallel tasks requiring little or no communication cost. The speedups are calculated directly by the number of parallel tasks. Although this term described parallel tasks on high performance computers, not until the late 1990's did the term become common in FPGA environments[29][6]. This makes the FPGA popular for most coarse-grained applications - in particular, applications supporting critical paths running in a PC.

This study is about a process that logically synthesizes a Java soft chip to support real-time applications. Reification is the name of this process and it begins with a translation of the Java byte codes of the real-time application. These codes are translated into a design language representing a Harvard micro controller (HMC) with a simple, uncomplicated design (no pipe-line and no cache). Once synthesized the real-time application is fast enough and allows an uncomplicated approach for timing analysis, in particular, calculating WCET.

Chapter 2
THE PROBLEM

This study is about reification - a process that translates Java byte codes into a design that represents a HMC. After producing the HMC this design is logically synthesized into an FPGA. The resulting FPGA soft processor is fast enough to support the stringent timing demands. In addition to the fast times, the timing analysis is simplified.

As mentioned earlier the Pico Java processor had many improvements, one of which was instruction pipe-lining. In particular this improvement provides a mechanism that prefetches and executes instructions in a parallel fashion. Executing an instruction requires that the instruction be fetched, decoded, executed and the results written back to memory. With pipe-lining it is possible to execute four instructions simultaneously with each operating at a different stage. It is fact that pipe-lined instructions complicate the timing analysis. In almost all cases pipe-lining is not used in the design of real-time processors. But if pipe-lining is used the WCET must be calculated using the times for instructions without pipe-lining.

Another enhancement is cache which attempts to store some of the memory needed by a process in a smaller section of very fast memory. The first access of a data item would be from main memory and subsequent accesses might be from the faster cache memory. The problem is that it is nearly impossible to predict whether a particular memory access will be using cache or main memory. In the case of Java this may be a registered structure as

compared to a stacked structure located in memory. Typically, cache in a real-time system causes problems with timing analysis. Like pipe-lining caching in real-time processors is typically not used. When cache is used the timing analysis must show the statistics concerning the cached and non-cached timings.

Figure 2.1 shows the basic notions concerning the timeing analysis of systems. With processor complexity, such as cache, the WCET cannot be observed. The more complex the design is, the wider the gap is between what is observed and the WCET. In addition the complexity complicates the analysis required to compute the WCET.

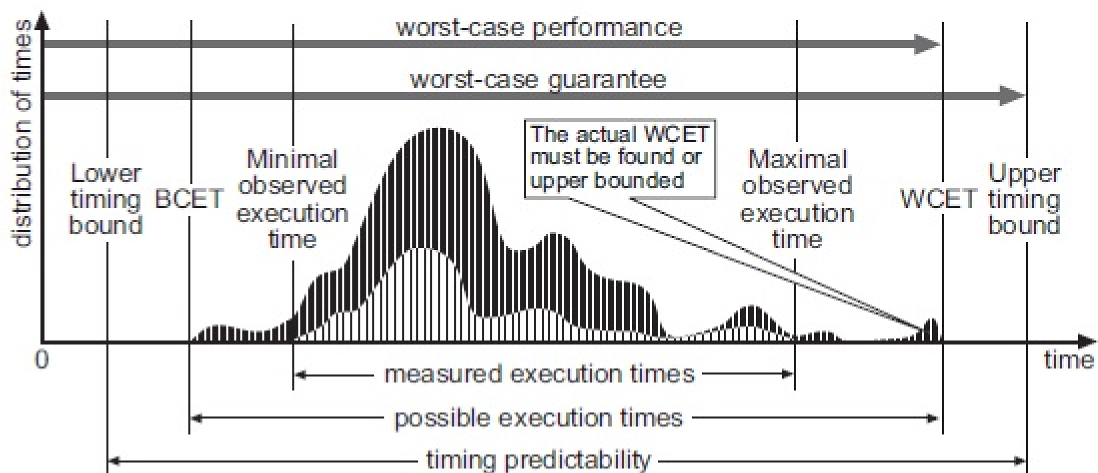

Figure 2.1: Basic notions concerning timing analysis of systems [34].

In 1999 Sun introduced the Pico Java processor design [3]. Released as a design and not as an implementation it gave third parties a challenge to produce their implementations. As part of the design a major speed enhancement was instruction folding. This feature allows for a runtime translation between the byte code's zero operand instruction to a 3 operand instruction. To support this approach a cache is necessary. This folding technique provided

a needed solution to Java's major speed problems and is the primary enhancement that gives Java the speed necessary for real-time application.

In 2003 real-time enhancements influenced the study of the Java optimized processor (JOP by Schoeberl [23]). His premise was to produce a predictable instruction set which allows simplifying WCET analysis. In the design pipe-lining was removed which provided the predictability of each instruction's execution time. In addition the design incorporated an additional cache. This new cache supported method invocation. The premise was to increase the overall statistical improvement for cache hits and to remove some restrictions on timing analysis. Instruction folding was not included in this study.

In 2006 a study shows the Pico Java processor implemented in an FPGA [20]. Although the work failed at normal FPGA clock speed the entire Java virtual machine was included. A significant contribution of this study was instruction folding. The goal of this study is to produce a Java soft chip that supports Java real time applications. The soft chip design provides no enhancements that complicate WCET. In addition the chip maintains the significant speedups provided by the enhancements.

Necessary to support these goals, this study uses a HMC model. The approach taken supports a design with abstract behavior that provides the ability to index (or address) between Java instance variables and index within specific Java methods. Like a Java abstract class, these abilities are empty by default and must be resolved during reification.

These abilities differ from traditional HMC implementations. That is the addressing nature of a traditional HMC is provided at run-time where word length defines how much one can address within the HMC. Conversely, in the HMC designed in this study, the addressing

(or indexing) is provided at compile (or design) time which requires logic synthesis for run-time implementation. Both of these approaches support generic Java applications. The study's objectives are as follows:

1. A reified HMC with the following improvements:

    (a) an abstract data unit that merges and strips data specifics with a reified process representing Java's instance variables. All indexing is supported by a coordinated effort with the control unit using the program counter.

    (b) a control unit that coordinates with a reified process that maintains the HMC's program counter. This effort requires a coordinated effort with the execution unit. Indexing is based on the current program counter.

    (c) a program unit that acquires and prepares the instruction specifics with a reified process. This unit acquires instruction parameters from the data unit. Subsequently, it prepares the instruction for execution. Like the data unit this unit requires coordination with the control unit.

    (d) an execution unit that supports a virtual set of instructions. During synthesis, the required instructions are enabled and included into the final circuit. The instruction unit supports 2 types of instructions: integer instructions and a custom set of instructions supporting a vector of 9 integers, namely, a vector multiply and sum. All instructions operate in 2 clock cycle.

2. A prototyped ontology that represents the reification process. This ontology provide the axioms, rules and instances that provides reification. The process begins by

translating the Java byte codes into VHDL design using templates that define the abstract nature of a HMC.

3. A restrictive subset of Java supporting the Sobel and Bubble Sort applications.

This study is about Java processors and their ability to support real-time applications. Historically JOP is the only Java processor capable of supporting real-time applications. In the JOP design pipe-lined instructions were removed and the claim of predicting each instruction laid the foundation for real-time abilities. The instructions are predictable but are still lacking in speed. To aid in WCET analysis, the design supports a method cache.

Lacking from the JOP design is the instruction folding technique which is critical for improving Java performance. It is this feature that gives Java the significant speedups for real-time applications. Conversely, the Java processor produced by this study has no cache, no pipelining, and supports instruction folding. Thus the WCET analysis is simple and the speedups are sufficient to support real-time applications. The author considers the key contributions of this research to be:

1. A Reifiable HMC - a modified HMC supporting Java for real-time applications. The design is uncomplicated supporting simple WCET analysis. The design supports virtual behavior in the form of design templates. In addition the design supports abstract behavior where this behavior is resolved during the reification process.

2. Reification - a prototyped ontology supporting the axioms and rules that defines a translation between Java byte codes and VHDL design templates representing abstract

behavior of a HMC. In addition it provides the mechanism to logically synthesize all behavior.

3. Cache-less Design Folding - the process supports design folding from byte codes into the design specifications of the HMC. The resulting design supports a 3 operand instruction that requires no supporting cache.

4. A Soft Bus - a structured object that interconnects circuit objects within and between each HMC unit. A simple indexing occurs where the program counter is used to synchronize each of these units. With the soft bus the word boundary limitations of the HMC are expanded. The soft bus for this study supports both a Java integer type and a vector containing 9 Java integers.

Chapter 3
REVIEW OF LITERATURE

This study is about simplifying a Java processor while maintaining speed and predictability as required for real-time designs. Simplifying the processor requires design changes such as removing pipe-lining, removing caches and supporting instruction folding without a cache. With this approach, one can design a simple, fast processor capable of supporting Java in all fashions, including real-time.

After FPGA'S became popular, many soft chips became available. The Pico Java implementation has been basically ignored with most soft chips supportimg the C-language (or a C-like language). Like the Pico Java processors, no claim is made about supporting real-time applications.

Much later, a Java soft chip ([23] JOP) became available with a claim to support real-time applications. Although, the Pico Java design was not used, the processor supports most of the Java Virtual Machine. Missing from this design is instruction folding, which as mentioned earlier, is the key component for speeding up Java. Conversely, the JOP design does include cache. To reduce the WCET analysis, the design supports a method cache.

After the first Java soft chip, the first Pico Java soft chip became available[20]. The key component of this processor is the implementation of instruction folding. Although the processor had significant speedups the implementation had some major issues. To address these issues, the clock speed had to be skewed (slowed down).

This chapter provides a review of the literature related to using FPGA's to implement soft chips. The first section looks at the high level languages supported. The second section looks at soft chips in general. Last is a section on a background describing the real-time environment for soft chips. Last it provides a section on ontologies used to support this study's methodology.

3.1 HIGH LEVEL LANGUAGE TO HARDWARE CIRCUITRY

This section reviews the languages supported by soft chips. This support comes in the form of a library used by application developers to create hardware circuits.

3.1.1 Trident

Trident is a framework that synthesizes C program's floating point operations to FPGA's. Floating point arrays are copied to off-chip memory banks where the computation takes place. To accomplish this, Trident uses schedulers and pipe-lining schemes. The goal of Trident is to allow rapid prototyping of these calculations in hardware. To achieve this goal, Trident provides libraries for scheduling and pipe-lining schemes. Trident is an open framework, and much is promised [30].

3.1.2 Handle C

Handle C provides a compiler for Fortran, C, or C++. The user requesting algorithmic calculations to be performed in an FPGA, uses Handle C's looping APIs. Handle C then compiles the code into an proprietary intermediate net-list containing coarse functions. Next Handle C optimizes this intermediate net-list before expanding to the FPGA specific net-list,

which is then compiled to the FPGA bit-stream.

Another aspect of the Handle C APIs is the use of parallel processes. The compiler creates an abstract syntax tree for each parallel process. The compiler uses these trees when creating the intermediate net-list [14].

3.1.3 Impulse C

Impulse C is a C-based development system for coarse-grained programmable hardware targets, including mixed processor and FPGA platforms. At the root of this technology are the Impulse C compiler and related tools and the Impulse application programmer interface. Impulse C can process blocks of C code, most often represented by one or a small number of C subroutines, into equivalent Verilog.

The Impulse compiler and optimizer enable the automated scheduling of C statements for increased parallelism and automated and semi-automated optimizations such as loop pipe-lining and unrolling. Interactive tools provided with the compiler let designers iteratively analyze and experiment with alternative hardware pipe-lining strategies [19].

3.1.4 SRC C

SRC C is a development environment for the Map C processor. Complete with editors, compilers, debuggers and simulators, SRC C provides a full programming environment. After the code is debugged, it can then be targeted for the Map C processor. SRC C has the following compilation modes:

- Debug - code is compiled into a Map emulator. It verifies the CPU and Map interaction.

- Simulation - supports applications written in C, Fortran, Verilog or VHDL and produces a simulation executable that supports the specific application code.

- Hardware - the code is prepared for execution in the FPGA using the Map design. In this mode, the code is optimized for parallelism by preparing loops for pipe-lining, scheduling memory references and by providing support for parallel code blocks and streams. The output is in an intermediate HDL, which is then compiled into the FPGA's bit-stream.

3.1.5 Mitrion C

Mitrion C and the Mitrion virtual processor represent a new approach to software programmability for FPGAs. The virtual processor is a massively parallel high performance processor for FPGAs that executes software written in the Mitrion C programming language.

The processor's architecture follows a cluster model, placing all processing nodes within an FPGA. The compiler and the processor configuration unit use the Mitrion C source code to create processing nodes and an ad hoc network on a chip.

The network uses point to point connections wherever possible and switches wherever required. Its latency of a single clock cycle is guaranteed, and network nodes are optimized to run a single instruction and communicate on every clock cycle. The result is a cluster with full fine-grained parallelism. Adapting the cluster to a particular program transforms the von Neumann architecture's inherently sequential problem of instruction scheduling into a parallelizable problem of data packet switching [32].

3.1.6 RC Toolbox

RC Toolbox is part of the Matlab environment. It provides a basic-like language to support circuit generation. It consists of four key components. First, RC Blockset allows the programming of sequential and iterative constructs directly related to those in C languages and includes four categories of blocks: program flow for sequential, parallel, and pipe-lined constructs; math for math functions, including floating point types; parallel memory access for global variables and memories; and RC abstraction layer for integration with various RC platforms. Designers can use the Matlab design environment to easily import third party intellectual property cores as a graphical block with inputs and outputs, and hardware experts can use it to incorporate HDLs for access to low level programming. Second RC I/O consists of hardware abstraction layer libraries optimized for each RC platform. Third, with RC Debugging Toolbox, users can validate entire applications as well as generate, collect, and visualize application data - all within the Matlab environment. Lastly the RC Platform Builder automatically generates all required logic and compiles the entire bit-stream without exposing the complex FPGA implementation tools [23].

3.2 JAVA SOFT CHIPS

Java soft chips are multiprocessor cores implemented using logic synthesis into an FPGA. These processors differ from the previous section in the fact that they implement the Java byte codes directly in hardware. There are varying types, based on whether they support full or partial byte codes. In addition some soft chips support the entire Java virtual machine. Finally some support Java with custom instructions. All processess discussed in this section

lay no claim of supporting real time applications.

3.2.1 Jiffy

JIFFY is a JIT compiler that runs in an FPGA. It is optimized for space to run within a resource constrained embedded system. Byte codes are translated into a transitional language consisting of three registers and a stack. The three registers are required for most stack operations and then optimized. Last these optimized instructions are translated into native instructions of the target architecture. JIFFY was tested in a CISC (80586) and a RISC (Alpha 21164) architecture which produced a speedup of 1.1 and 7.5 times faster than interpreting Java byte codes on the x86 architecture. The compilation time is about 10 times as fast as similar compilers [2].

3.2.2 Moon

Vulcan ASIC's Moon processor is a JVM that runs in an FPGA. Moon uses microcode to implement the JVM instructions. In addition stack folding is used to improve the stack operations. Instruction folding provides a speedup of four. [22].

3.2.3 Komodo

Komodo, is a multi-threaded version of a Java processor. If has four instruction fetch units, each with program counters and a status flag for each thread. Komodo implements complex instructions in microcode. In addition Komodo contains a hardware priority manager used for scheduling. This manager coordinates with the microcode and can select threads after each byte code instruction. Komodo run in an FPGA and implements a subset of the JVM. The processor must run at a reduced clock rate. [11].

3.2.4 Femto Java

Femto Java is a research project to build an application specific Java processor. In this project, they studied small embedded applications (50-280 byte codes) and noticed common instructions (69 instructions) for both 8 and 16 bit applications. These instructions take from 3-14 clock cycles to execute. A manual configuration process is discussed which can create a pattern of 22-69 instructions. There is no mention of instruction improvement. Given that it implements only a 16 bit version, it is not Java compliant[31].

3.2.5 Ignite

Ignite is a stack processor designed for high speed Forth applications. It implements a subset of the JVM, which is called Removed Operand Set Computer (ROSC). It varies slightly from Java byte code. The small JVM converts Java byte code into ROSC.

As most Forth processors, Ignite contains two stacks and 16 global registers. Four 8 bit instructions are fetched from the 32-bit memory. Ignite uses unique instruction formatting where both immediate values and branch offsets are right aligned. This simplifies instruction decoding. Ignite is available as an ASIC at 80 MHz and an FPGA. The processor must operate at a reduced clock rate[21].

3.2.6 Light Foot

Light Foot [23], is a 8/32 bit processor based on the Harvard architecture. The 8 bits define the program memory width and the 32 bits define the data memory width. It contains an integer ALU and two stacks. One stack is used for temporary data and is implemented using registers. The other stack, a conventional stack using memory, is for the JVM stack.

The design supports instruction folding. In particular, it introduces a typed byte code that combines a reference and push into one clock cycle.

Light Foot implements a full, interpreted version of the JVM. The speedup is 8 times better than RISC interpreters. It runs on a Vertex II and operates at a reduced clock rate.

3.3 Current Research

A real time system is defined by information processing which has to respond to externally generated input stimuli within a finite and specified period. Moreover, the correctness depends not only on the logical results but also the time it was delivered. A failure to respond within the specified period is considered as bad as a wrong response.

These systems have many characteristics such as having guaranteed response times, being extremely reliable and being efficiently implemented. Real time systems have four classifications[10]. In this document, the hard real-time classification is addressed and the response deadlines must occur correctly.

Thus guaranteed response times are critical for real-time systems; we need to be able to predict confident worst case response times. Efficiency is important but predictability is essential. There are two ways to determine the worst case time - by measurement or analysis. In many cases, processor complexity can lead to problems. Specifically processor models with caches and pipelines can complicate the analysis of worst case execution time. The hard fact for real-time systems is that worst case execution time is necessary. This fact ensures that no real time response will miss its deadline.

3.3.1 Java

The inherent goal in Java technology "write once, run anywhere" is the fact that Java programs run on the Java virtual machine[13]. Java programs are insulated from any particularities of the underlying hardware. Thus the need for Java to build translation layers into the virtual machine. Java can be implemented using interpreters and compilers. Both convert the virtual machine (byte codes) into executable instructions intelligible to the underlying CPU. The interpreter emulates the byte codes, where the compiler compiles the byte codes into native machine code. The term "just-in-time" is used to describe a Java implementation[27] where the code exists as standard Java byte codes until a method is called. At that time the byte codes are compiled into machine code and then the method is executed. Subsequent calls to the method result in direct execution of the previously compiled machine code.

The byte codes provide an image of a program that will execute on any system with a virtual machine. This was Sun's main goal for Java. These byte code images are secure and program representation is small, both of which are underlying reasons for the popularity of Java for a broad range of applications.

Conversely, the insulation Java imposes on the byte code image and the platform it runs on is somewhat complicated. Performance degrades with the dynamic translation whether interpreted or compiled. The JVM startup time can be significant. In addition a stack is used as an operand scratch pad for computations and also for local variable storage which leads to a major bottleneck.

3.3.2 Pico Java

Sun's strategy moved into the hardware arena with the introduction of another translation layer for the virtual machine. The Pico Java processor core is a highly efficient Java execution unit design that delivers speedup over the Java performance of the x86 processor architectures and over compilers. The processor core is a small, flexible microprocessor that directly executes Java byte code instructions[28].

These processors are popular for smart phones, PDAs and set-top boxes. In addition they are ideal for all embedded and network computing applications. The motto "write once, run anywhere" motivates software vendors to develop large applications entirely in Java and has them considering these new Java platforms over general purpose processors.

As mentioned, Pico Java is a microprocessor core that provides high speed, direct execution of Java byte code instructions. The architectural specification outlines a number of design innovations that speed up Java. Most notably, instruction folding gives Java significant speedups.

The Pico Java architecture consists of a RISC-style pipe-line with a Java byte code instruction set. The Pico Java core is designed on a four stage pipe-line. A 64 register set is used for a stack cache where spilling and filling is used to manage data. The core is designed for flexibility and performance over a wide rage of application areas. For example, the core can be designed with or without floating point and various types of data and instruction caches. In addition the specification does not include any memory or I/O interfaces.

Besides Java's interpretation Java has a zero operand instruction set. To do a simple add

requires 4 instructions. A 3 operand instruction format allows the use of instruciton folding to reduce instruction count and times. Instruction folding can convert the 4 instruction sequence into 1 instruction to support the add operation.

3.3.3 FPGA Architecture

Invented in 1980 by Xilinx, the FPGA is a semiconductor device that allows runtime user configured implementations of complex digital circuits[35]. The FPGA was originally designed as a quick prototyping solution to large circuit designs. Previously, the only way a designer had to debug circuits was with hardware simulators. But with large designs, these simulations quite often bogged down. Not surprisingly, the FPGA soon became the quick time-to-market solution for circuit designs and later for soft core processors.

In contrast application specific integrated circuits (ASIC) maintain their functions within the life of the chip in which they reside, whereas the FPGA, can be synthesized (reprogrammed) to perform different functions in a matter of seconds. Programming an FPGA, the designer is now free to choose between high level languages (similar to assembly and C) like VHDL and Verilog or object oriented graphical environments like Viva (similar to form designing in Visual Basic).

An FPGA contains a two-dimensioned array of configurable logic blocks (CLB), an I/O bank and a set of data transfer channels supporting each row and column of the CLB array. Both the CLBs and IOBs operate in one of two ways. First, they must support the configuration process where the FPGA receives the application circuit design. Second, these objects support the application circuit. The interconnections support each of these operations

by carrying signals to and from each object. Each CLB has two channels providing input and two for output. These channels carry both configuration data and application data.

During the configuration process, the FPGA is programmed to represent an application circuit. To begin, the configuration S/W must select a CLB. It is this CLB that will represent a gate in the application circuit. After the selection the process establishes a chain (or pathway) to this CLB. Figure 3.1 shows the configuration chain to the end point represented by CLB_3. CLB_1 and CLB_2 represent the chain. To configure each CLB requires identifying which channels have the input and to which channels to direct the output. The last piece of information is the end point representing the desired gate.

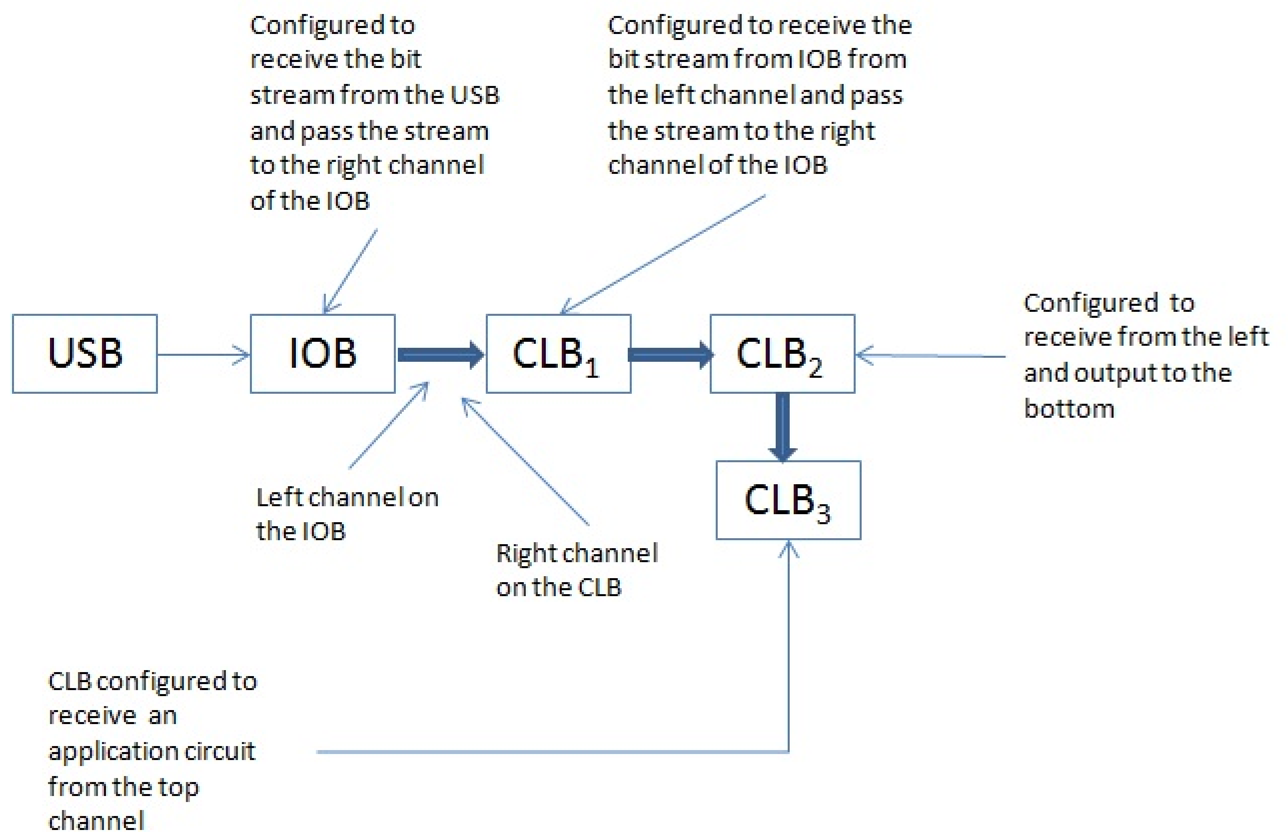

Figure 3.1: FPGA configuring an application circuit

After configuration the FPGA represents the application circuit. It responds to programmed inputs and provides appropriate output. Figure 3.2 shows how an FPGA is programmed to represent an *and* gate. As shown, the application circuit *ands* the value represented by switches where the results are shown on an *LED*. As you can see, two IOBs are connected to the appropriate switches, a CLB receives input from the top and left channels and performs the *and*. After, the CLB directs the results to the left channel. From the figure, you may easily see the last IOB accepts input from the left and directs it output to the appropriate LED.

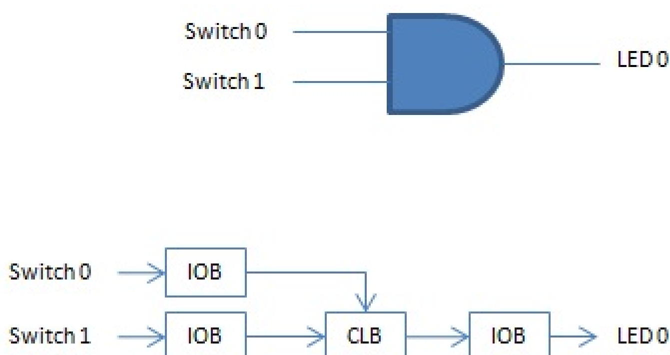

Figure 3.2: FPGA And gate as an application circuit

Each CLB contains 4 slices (see figure 3.3). These slices are used to represent the circuit, are used in configuration and provide output to the channels. Each slice contains two logic cells. Figure 3.4 shows the components that make up a logic cell. Each logic cell contains two lookup tables (LUT). Although they operate as combinatorial logic, functionally they operate like a table lookup providing proper gate outpt. They contain gate values for 3 gates (and, nand and or).

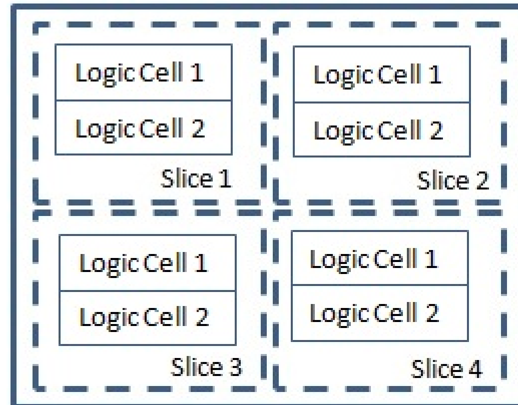

Figure 3.3: CLB - containing 4 slices

In addition to the LUTs, there is a full adder and a multiplexor that selects the proper support for the adder's role in the circuit. Also, each circuit can operate in either an asynchronous or synchronous fashion. To support synchronization a D flip-flop is used. A multiplexor is used to select between these two modes.

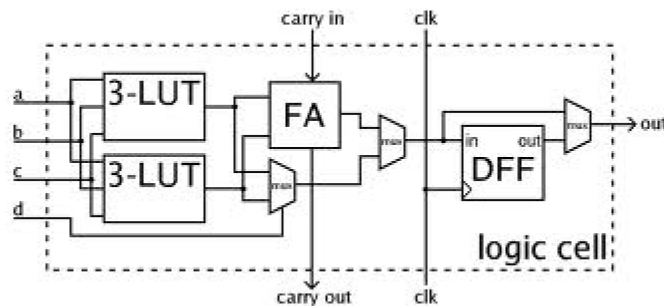

Figure 3.4: Logic Cell [33]

There are four inputs and two outputs for each cell. There are two channels, one to the right of the cell and one below the cell. Each output connects to either channel. These channels are the interconnection highway connecting each cell's output to the appropriate cell input. These channels respond to configuration tables where each intersection of a row and column channel can route.

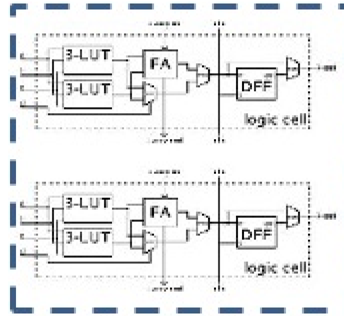

Figure 3.5: Slice arrangement in a CLB [33]

The last configurable objects are the input/output blocks (IOBs). The IOBs form a ring around the outer edge of the FPGA. These objects provide access to one of the I/O pins on the FPGA chip.

A synthesis process is required to reconfigure an FPGA. This process takes the design circuit (from HDL or Viva) through several steps that places the resulting circuit in the desired FPGA. The process takes several seconds for large designs.

3.3.4 Ontology

Webster defines ontology as “the theory of existence”. In mathematics this coincides with the symbol $\exists$ (there exists). In science and engineering the subject of ontology focuses on categories and the relationships of categories about entities that exist in a domain of interest. One can say an ontology is the product of a study of these relationships[25].

Moreover, ontologies are represented by a lattice, which define the rules of the multiple inheritances in a terminal hierarchy. The symbol $\top$ defines the universal type for a specific ontology. Conversely, the symbol $\bot$ defines the absurd type. Falling between these symbols in the lattice are the domain categories.

Most ontological studies begin with a classification of top-level categories. These categories represent comprehensive research over many centuries beginning with Greek philosophers, and, more recently, with studies in artificial intelligence. Most of the recent studies base their research on Charles Sanders Peirce[1]. His classification of firstness, secondness and thirdness are the standard in many modern studies.

Following Peirce, John Sowa [5] added to these top level categories with the concept of abstract versus concrete; and, in addition the concepts of occurrent versus continuant [24]. Occurent sub categories must respond to something; for example, responding to a button click. Once responded all supporting categories are of type continuant where they support occurrent objects. The abstract and physical categories define what is inherrited and what can be instantiated. In most all cases abstract categories define structure and do not appear as instances in a knowledge base. Whereas physical categories inherit abstract structures and they can define behavior.

When defining a prototype ontology the above norm is not followed. Given the complexities of developing specific ontolologes a bottom up approach may be used [24].

In the fields of science and engineering, developing all the axioms and definitions for an ontology requires a Herculean effort [24]. The types of ontologies where categories are not defined by the axioms and definitions are classified as a terminological ontology. In many instances, these ontology specifics are created as instances in the knowledge base. These typed ontologies are classified as prototype-based.

Chapter 4

REIFICATION ARCHITECTURE

This chapter presents the reification architecture. Reification begins with a translation of byte codes that represent a real-time application written in Java. Next the process folds the translated results into a HMC design. The process continues where the folded design is compiled, assembled, routed and synthesized on an FPGA.

In the following sections, the author begins with a discussion of architectural concepts. These concepts provide the reader with the necessary background to better understand the architecture. Next the author presents the reification architecture in detail. Presented as an ontology, this section defines and describes the axioms, rules and instances necessary to support the reification process.

4.1 Reification Concepts

This section describes the concepts necessary to support the reification process. The author begins by describing the ping-pong model. This model depicts an approach taken during circuit designs. The author shows why a model was necessary and provides a foundation supporting the model's synchronization abilities. The next section explains the communication types within the ping-pong model. The author describes and presents examples on the various communication types.

4.1.1 Ping Pong Process

The VHDL language was used to develop all circuits in this study. Although VHDL is a very sophisticated language, much care must be given when creating circuits. VHDL was primarily created to document logic circuits and describe complex behavior. Recently it has been used as input to computer aided design tools to synthesize logic devices that represent circuits in actual hardware. In addition, it is normal to analyze the circuits created from VHDL before you test. As an analogy, if one would use the C language to write a program, than one would have to analyze the assembly language to insure your logic is represented correctly.

In this study, a ping-pong approach is taken. Similar to a class of circuits labeled synchronous sequential circuits [16], previous behavior of input signals are used to generate output signals. Typically, D-flip flops are used to store all application data. On the rising edge of the clock, all sequential signals are updated. The follow π-calculus equation [15] defines this approach:

$$Sys \stackrel{def}{\equiv} \sum_{i=1}^{\infty} r_i.\overline{r_{i_{ff}}}.P_i \mid c_{t_{20ns}}.r_{i_{ff}}.\overline{r_i}.P_0 \tag{4.1}$$

The P_i and P_0 processes represent concurrent tasks in VHDL. The P_0 process represents the D-flip flop and c_t is the clock that operates at 20 nanoseconds. The P_i processes are the sequential circuits operating in an asynchronous fashion. All processors are scheduled to execute when an input signal changes. Thus, each P_i is scheduled when it's r_i signal changes.

The P_0 process supports the P_i processes by updating their registered signals (r_i) with their ($r_{i_{ff}}$). This is performed every 20 nanoseconds on the rising edge. The time necessary to update these signals is trivial, regardless of the number of P_i processes. The communication between the P_i and P_0 are in parallel, both assumed to be scheduled on each rising edge. Therefore it is evident that the P_i processes are synchronized to the 20 nanosecond clock, although they are considered asynchronous circuits. With this in mind, all future equations and diagrams remove the persistent circuit (P_0) and only imply it's existence using the *ff* convention.

Moreover, the P_i processors must respond to their r_i signals. After the r_i signal is generated and when scheduled, the P_i processor performs its associated behavior. When complete, the P_i processor must update their corresponding $r_{i_{ff}}$ signal. With this in mind, each processor must complete before the next clock. This can be assumed to be 18 nanoseconds.

4.1.2 Process Communication

There are two ways processes communicate with each other. Figure 4.1 shows three processes. The flip flop updates R_on each rising edge. At that time, only the A process is scheduled as per the sensitivity rule [16]. Upon completion, the R_ff value is modified, resulting with the B process being scheduled. For demonstration purposes, process B updates the R1_ff signal.

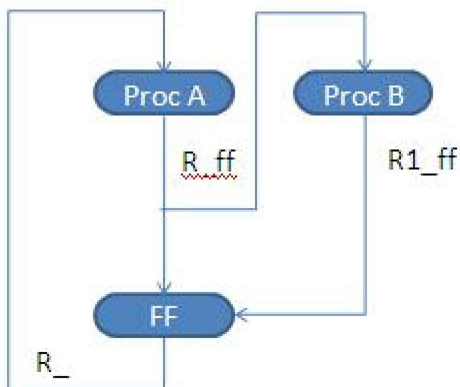

Figure 4.1: Serial Communication Between Processors

The second type of execution (see Figure 4.2), is parallel communication. Again this shows three processes. Because they both use R_, then the sensitivity list for both A and B must contain R_. Process A updates R_ff and B updates R1_ff.

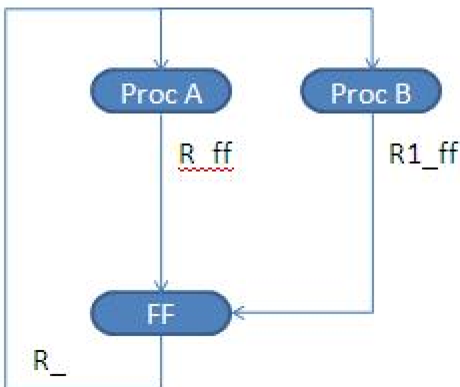

Figure 4.2: Parallel Communication Between Processors

4.1.3 Design Folding

The reification process reads the Java byte codes and translates them to specific VHDL design files. These files are then compiled, assembled, routed and used to configure the FPGA. During the translation, design specifics (presented next) are merged into design templates.

As is common in hardware designs signals are routed using multiplexors, where a selection signal is used to route desired signals through the selecting device. Also encoding signals require similar type logic designs. In VHDL similar goals are achieved using the case statement.

As shown inFigure 4.3, a case statement template is depicted where the selection parameter is used in conjunction with a series of when statements. A tag is appropriately placed in the case statement. The next tag appears in the middle of the case grouping. The reification process concatenates the specifics for each when, and inserts these into the case template.

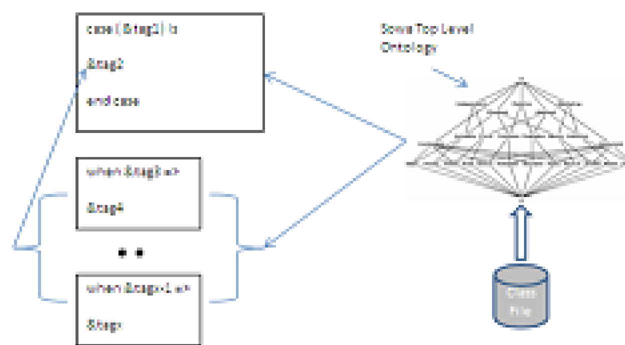

Figure 4.3: Design Folding

4.1.4 Sobel

The popular Sobel edge detection algorithm is used to demonstrate the Reification process; although by itself this algorithm may not be consider for a real time application; one could imagine a real time visual control system incorperating Sobel edge detection and other image processing functions.

In addition to choosing Sobel for demonstration, the algorithm is straight forward and

simple to implement. It consist of about a dozen instructions and contains only one path through the instructions. In other words, the BCET and the WCET are equal and observable. Also the algorithm uses two data structures. Specifically, the structures required are an integer and a pixel neighborhood containing a 3x3 integer structure.

Equations 4.2 thru 4.4 define and edge value in the x-direction. The pixel object is defined by a record structure in VHDL to contain 9 integer values of 16 bits each. Equation 4.2 shows a 3x3 multiplier which requires the use of 9 multipliers. A custom instruction is used to multiply the pixel and G_x structures in one instruction cycle.

Equation 4.3 defines the sum of the product values discussed above. Another custom instruction is used to compute the sum of the 3x3 product structure in one instruction cycle. The resulting sum is stored in the integer variable sumx. The 3x3 custom instructions demonstrate the parallel capabilities of the FPGA and the ability to mix word sizes.

$$product = \sum_{i=1}^{3}\sum_{j=1}^{3} pixel_{(i,j)} \times Gx_{(i,j)} \tag{4.2}$$

$$sumx = \sum_{i=1}^{3} product_i \tag{4.3}$$

$$sumx = |sumx| \tag{4.4}$$

The last equation (4.4) takes the absolute value of the previously summed value sumx. This instruction is part of the IJVM instruction set. With the new parallel instructions, the demonstration shows the ability to mix custom instructions with IJVM.

Equations 4.5 to 4.7 define the edge in the y-direction. This is similar to the x-direction with the only difference being the calculation of the product value. The pixel is multiplied

by the G_y structure. Like the x-direction, the sum and absolute values determine the sumy value.

$$product = \sum_{i=1}^{3}\sum_{j=1}^{3} pixel_{(i,j)} \times Gy_{(i,j)} \tag{4.5}$$

$$sumy = \sum_{i=1}^{3} product_i \tag{4.6}$$

$$sumy = |sumy| \tag{4.7}$$

The last set of equations (4.8 to 4.10 define the summed value. This value is then limited between 0 and 255. Not shown, this value is sent back to the PC over the serial port.

$$sum = sumx + sumy \tag{4.8}$$

$$sum = limitGT(sum, 255) \tag{4.9}$$

$$sum = limitLT(sum, 0) \tag{4.10}$$

4.1.5 Harvard Micro Controller

This study uses the HMC as a model where the design specifics (see section 4.1.4) from the Sobel algorithm are inserted into the modeled design files. Since the platform is an FPGA, some changes were necessary to incorporate FPGA advantages. Figure 4.4 depicts the model with the changes. The most important change is inclusion of buses between the data and program units. This change takes advantage of the interconnection capabilities of the FPGA and the flexibility of the reification process.

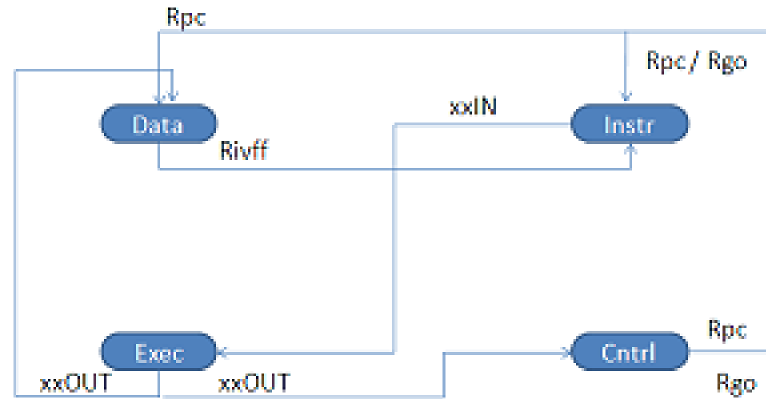

Figure 4.4: Harvard Micro Controller Function Design

Figure 4.4 the ping-pong approach for communication between the units. The approach is evident with the labels “ff” as they reflect signals being computed since the last rising edge of the clock. The following π equations depicts the HMC model:

$$Sys_{\{t_0,t_1\}} \stackrel{def}{\equiv} out_x.\overline{R_{iv_{ff}}}.D \mid R_{iv_{ff}}.R_{c_{ff}}.\overline{in_{x_{ff}}}.P \mid in_{x_{ff}}.\overline{out_x}.E \mid out_x.\overline{R_c}.\overline{R_{c_{ff}}}.C \tag{4.11}$$

The equation shows the HMC’s four units operating as parallel processes. Each process reacts to the registered (non ff signals) values which were updated during the rising edge of the clock. Moreover, the system (Sys) is defined over two clocked periods($t_0\, and\, t_1$). The t_0 period is the write and prepare cycle. Conversely the t_1 period is prepare and execute cycle. The t_0 and t_1 period require 40 nanoseconds (2 system clocks) for all instructions except input and output. These instructions involve serial data where the t_1 cycle reflect the associated serial transmission times. Equations 4.12 thru 4.18 show the reactions within each period.

The following equations show the reactions during the t_0 period. The control unit

reacts to instruction completion (4.12) by adjusting the program counter (R_c) for the next instruction (equation 4.13). Also the data unit reacts by updating the instance variables (equation 4.14). The program unit must react to both the control and data units and is shown by equation 4.15. During this reaction the program unit prepares the instruction object for the execution unit which will react during the t_1 period.

$$Sys_{t_0} \xrightarrow{out_x} \overline{out_x}.E \mid out_x.(C \mid D) \tag{4.12}$$

$$Sys_{t_0} \xrightarrow{R_{c_{ff}}} \overline{R_{c_{ff}}}.C + R_{c_{ff}}.P \tag{4.13}$$

$$Sys_{t_0} \xrightarrow{R_{iv_{ff}}} \overline{R_{iv_{ff}}}.D + R_{iv_{ff}}.P \tag{4.14}$$

$$Sys_{t_0} \xrightarrow{in_{x_{ff}}} \overline{in_{x_{ff}}}.P + in_{x_{ff}}.E \tag{4.15}$$

The next period, t_1, is depicted in reaction equations 4.16 to 4.18. It is during this cycle that the HMC prepares and executes the instruction. Initially the system reacts to the program counter (R_c). The data unit responds to the new program counter and prepares for the instruction results. In addition, the instruction variables ($R_{iv_{ff}}$) reflect any changes. The program unit reacts to the instance variables. Last the executions reacts and starts execution.

$$Sys_{t_1} \xrightarrow{R_c} \overline{R_c}.C \mid R_c.D \tag{4.16}$$

$$Sys_{t_1} \xrightarrow{R_{iv_{ff}}} \overline{R_{iv_{ff}}}.D + R_{iv_{ff}}.P \tag{4.17}$$

$$Sys_{t_1} \xrightarrow{in_{x_{ff}}} \overline{in_{x_{ff}}}.P + in_{x_{ff}}.E \tag{4.18}$$

4.2 Reification Ontology

In this section the author describes the prototyped ontology used in this study. This ontology represents the reification process. Recall the Sowa top level categories where these categories exist in 5 levels. The top level categories similiar to Peirce's are labeled Independent, Relative and Mediating. Next Sowa portrays the categories of Abstract and Physical. The next two sections provide insight into these categories.

The authors uses JESS (Java Expert System Shell) an expert system to design axioms (or templates) representing Sowa categories and rules represent the mediation categories.

In addition to Jess, the author chooses to present reification concepts using the concept graphing (or concept mapping [18]) [26]. Concept graphing is a convention of mapping logic. A concept graph shows concepts and the relationships between the concepts. Originally developed by John Sowa to support his studies on representing knowledge, concept graphing plays an important role in ontologies [17]. In the diagrams, boxes represent the concepts where the lines represent relationships between the concepts.

You may recall this study is about reification; in particular it is about a process. The author uses the studies of Singh on thematic roles [12]. These studies focus on the role of a source and a product in the support of a process. For the source, these studies define an agent and matter. An agent represent a process role as the initiator in a voluntarily fashion. In contrast, matter represents a process role as a resource. Both the initiator and the resource provide the source in a concept map.

Regarding the product, these studies define the process role of a result. Further a result

is a goal of an act of a source. In addition these studies define a theme. A theme is a product of a process that may be moved, said, or experienced, but is not structurally changed. A result supports the knowledge as a goal whereas the theme represents the essence.

The author uses a role in a dyadic relationship with the objective. The knowledge base supports these relationships with predicates for each role object. Acquiring a role object, the developer invokes the predicate with the verb that defines the objective. For example, to retrieve an agent representing a design fold would be *getAgent(designFold)*.

In concepts maps there is a formula operator (w), which translates the diagram objects to predicate calculus [4]. The concepts relate to typed variables, whereas the relationships relate to predicates with the concepts as arguments.

Finally the concept maps in this section depict concepts and thematic roles representing Sowa's mediating category as applied to the reification process. These maps refer to heuristic rules where the agent and matter represent the left hand portion of a rule and the result and theme represent the right hand portion. In addition to these roles, Sowa instances representing the History category may appear in support of either portion of a rule.

4.2.1 Abstract Categories

In this section the author describes the abstract categories of the study. As mentioned earlier, Sowa's abstract categories define the structure for form, proposition and intention.

It is in these structures (or categories) that the author begins to differ from Sowa. In his studies Sowa renamed the findings of Peirce's top level to represent more appropriate and expanded knowledge representations. The author begins by redefining the structures

of Intention, proposition and form as they relate to this study. These are discussed in the following sections.

Intention

Sowa describes intention in two ways. In the occurrent view, Sowa believes intention implies what the ontology user has in mind in a sense of accomplishment. In the continuant view, intention is more deliberate and clear in formulation.

For this study the author defines the category of goal. A goal is the ontology user's desire for specific knowledge or behavior. A goal expresses Sowa's category of occurrent. Also for intention in a more deliberate nature, the author defines the category of objective. Objective supports Sowa's belief and supports the category of continuant. Also objective supports goal in a deliberate manner. Figure 4.5 shows Sowa's category of intention with the two subcategories used in this study.

Used through this study, the author mentions a command, where command represents the goal. In addition, the author uses the term *verb*, where the author believes a verb supports a command (or goal) in a specific way.

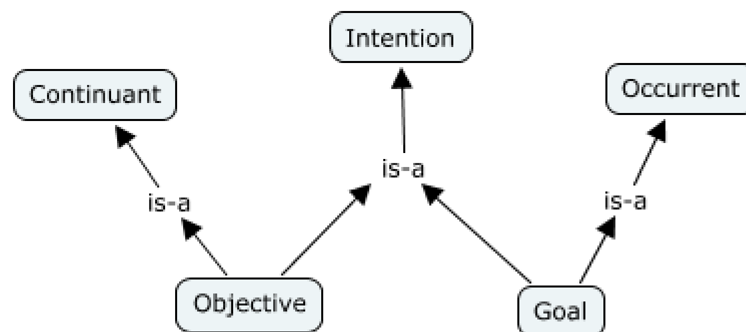

Figure 4.5: Extension of Sows's intention category

Goal. In this section, the author describes the goal category. Figure 4.5 shows this category being an intention and occurrent. As mentioned, goal is the intention of the ontology user.

A goal has one or more objectives(explained later). It is in goal that the supporting abstract categories must be initialized and the knowledge base must be maintained.

After initialization, the ontology responds to asserted goals. The following pseudocode code shows the behavior responding to an assertion goal:

```
Objective[] obj = kb[=Goal]
for each (o in obj)
    o.intention()
end for
```

Objective. In this section, the author describes the objective category. Figure 4.5 shows this category being both intention and a continuant. As mentioned, an objective is the intention of a goal. It responds to a goal by a method labeled intention.

First an objective has both measure and progress (discussed later). Measure defines the objective's role and the progress defines state behavior. In addition the objective category must have persistent state.

Last objective provides behavior that mediates the progress. The following code shows this behavior:

```
f = false
while (not f)
```

```
    State s = kb[=Objective(for state)]

    Behavior b = kb[=Progress(s)]

    f = b.exec()

end while
```

Proposition

In this section the author describes the categories that define the study's proposition. In his studies Sowa believes proposition (see figure 4.6) characterizes and describes relationships. Sowa references the work of Peirce and his characterization of secondness. As mentioned earlier, Peirce believes secondness relates and reacts to entities that represent knowledge.

For this study the author defines the category of measure and progress to further define proposition. These are discussed in the next two sections.

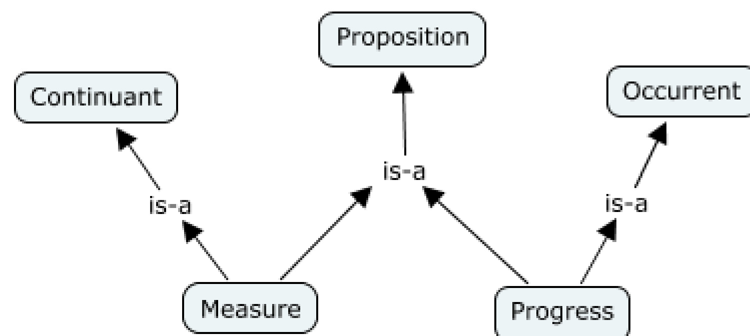

Figure 4.6: Extension of Sowa's proposition category

Measure. In this section, the author describes the category of measure. As mentioned earlier, measure supports an objective as a characterization of a *verb*. As a subcategory of proposition, it characterizes and describes the dyadic relationships of an objective to a role.

The measure category supports a structure that supports knowledge. As you may recall, this category is continuant and therefore, in the author's view, maintains a one to one relationship with objective. To support Peirce's secondness and Sowa's proposition, measure must relate entities with dyadic expressions. For this, measure supports predicates for each persisted object with getters and setters identified with an objective (a verb). For example, this may be *getagent* for objective.

Progress. In this section the author describes the category progress. Progress is a proposition in an occurrent fashion. It relates an objective to a series of state behavior. In addition it characterizes an objective by relating measure to progress results.

In addition to Sowa's relate and characterize, progress supports Peirce's findings where progress reacts. It reacts to the measured relationship by identifying knowledge behavior between the initiating roles of measure and progress. More on this behavior is discussed later.

The progress category supports a structure providing persistent knowledge. This category supports an objective in a one-to-many relationship. Like measure, progress defines dyadic expressions in the form of predicates and associated state.

Form

Form is Sowa's last second level category. It represents knowledge for proposition. It must support relationships and provide behavior for proposition categories of measure and progress.

The first category of form is the category of knowledge object. This category takes the structure of continuant. It supports proposition by acting in dyadic roles. Knowledge objects do not represent objects relating to time or time like sequences. Figure 4.7 shows the category of form and the inherited structures.

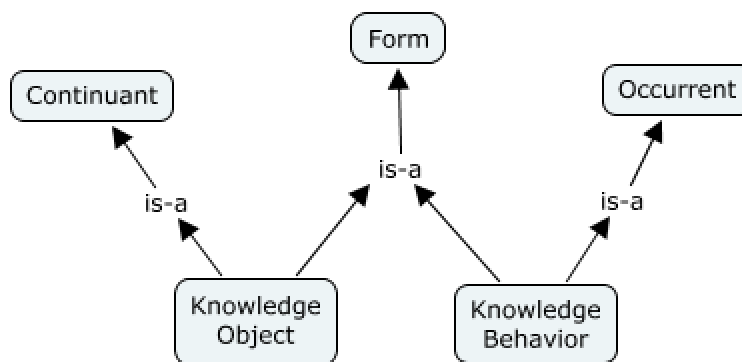

Figure 4.7: Extension of Sowa's category

The second category of form is the category of knowledge behavior. This category takes the structure of continuant. It supports proposition in the form of time-like sequences and supporting behavior.

The next two sections define these form categories in more detail.

Knowledge Object. As mentioned above, knowledge objects represent Sowa's form and take the structure of continuant. They support both measure and progress by acting in relationships as role objects. Knowledge objects are represented in a knowledge base as instances and are used in the mediating intentions of an objective.

All knowledge objects have a identification label. Much like a class name, knowledge queries use this label to acquire class like instances. Unlike class name, the identification

may change to support specific physical requirements.

Knowledge Behavior. As mentioned above, knowledge behavior represents Sowa's form and takes the structure of occurrent. It defines state and provides the behavior to support progress. In addition, this category supports Peirce's secondness where it reacts to measure by behavior that relates the measured role to the progress roles.

Figure 4.8 shows a concept map for knowledge behavior supporting Peirce's reaction. The open behavior appropriately copies the measured role of agent and matter to the states agent and matter. In contrast, the close behavior copies the result to either the measured result or to a supporting state's agent. This category identifies both open and close as abstract behaviors. Physical objects must extend this category and supply appropriate behavior.

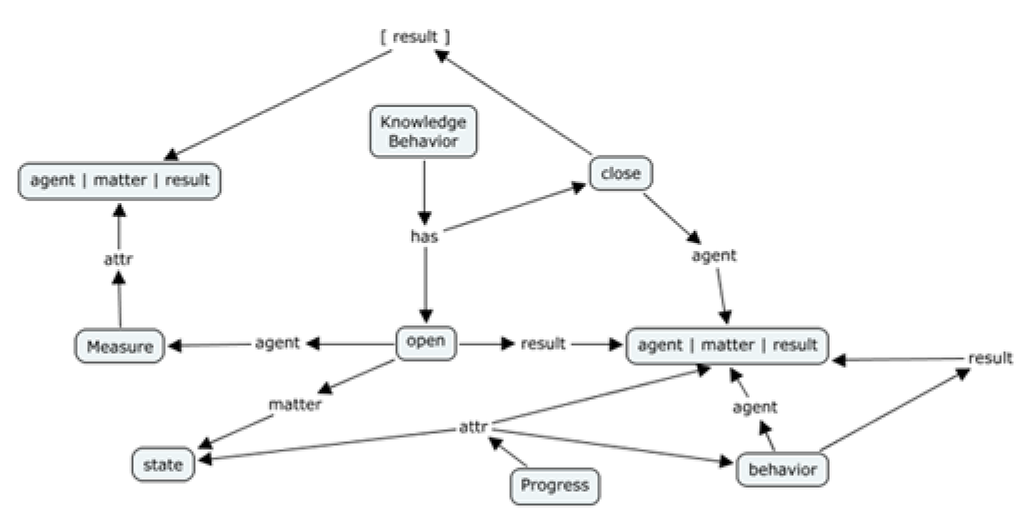

Figure 4.8: CMAP for the determine objective

The following pseudo code provides a view into this behavior:

```
Open -----
```

```
for each (State)

    KObject[] a  = kb[Measure.agent(some state type)]

    KObject[] m = kb[Measure.measure(some state type)]

    assert (Progress(State).agent(a) )

    assert(Progress(State).matter(m))

fnd for


Close -----

Knowledge ko = kb[=Progress.result]

assert(Measure.results(ko))
```

4.2.2 Physical Categories

In this section the author presents the physical categories for the reification process. This study's approach is to create a prototyped ontology. In particular this approach creates instances in the knowledge base, rather than explicit categories where these categories are defined with appropriate axioms and rules for the reification process.

To implement reification this project defines the category of reification. This category inherits the structure from the goal category described earlier. Reification represents the user's intention to reify Java byte codes into a circuit within an FPGA.

To represent the user's intention, the author describes three goals: translate, design and synthesize. Figure 4.9 shows the ontological view of the knowledge base supporting the translation goal (discussed in the next section). It shows the instances of the abstract

categories discussed earlier.

In Figure 4.9 you can see the instances of the abstract categories explained earlier. The goal, objective and matter instances are created during initialization. After initialization the user asserts (creates an instance) the byte codes. With this the instances in measure occur and the ontology begins to react with instances in progress. After reaction, the measured results reflect the appropriate reacted results.

Command

translation	determine
translation	resolve

Measure

determine	Agent: Binary[]	Matter: Pattern[]	Result: BinaryCode[]
resolve	Agent: BinaryCode[]	Matter: Symbol[]	Result: Instruction[]

Objective

determine	state
resolve	state

Progress

State	Behavior	Agent	Matter	Result
RDY	Check	Binary[]	Pattern[]	
ARG2	Check	Binary[7]	Pattern[=ARG2]	
ARG1	Check	Binary[5]	Pattern[=ARG1]	
ARG0	Check	Binary[3]	Pattern[=ARG0]	
EXIT	Check	---	---	x=specific pattern
START	Obtain	Binary[x]	x	
ASSIGN	Obtain	Binary[x]	x	
INSTR	Obtain	Binary[x]	x	
ARGS	Obtain	Binary[x]	x	
ADJUST	Obtain	---	x	Binary[x]
END	Obtain	---	---	BinaryCode

Progress

State	Behavior	Agent	Matter	Result
BEGIN	LookUp	BinaryCode	Symbol[]	
ASSIGN	LookUp	BinaryCode	Symbol[=ATTR]	i.e., a
INSTR	LookUp	BinaryCode	Symbol[=INSTR]	i.e., add
ARGS	LookUp	BinaryCode	Symbol[=ATTR]	i.e., [a] [b]
STOP	Assemble	---	---	Instruction(a=add(b,c))

Figure 4.9: Ontology view on knowledge base instances for translation

Translation Goal

This goal supports the user's intention to translate Java byte codes. To do this the byte codes must be searched to determine patterns. Much like what the Pico Java processor does in hardware, specific ontological behavior compares the byte codes to one of three

patterns supported within the study. Then the binary sequences are resolved with appropriate symbols (arguments, instructions, ect.).

As stated earlier a goal represents the user's intention. Further an objective represents the intention of a goal. A goal may have one or more objectives. For the translation goal there are two objectives: determine and resolve.

The following pseudo code describes this behavior:

```
Objective[] list = kb[=goal(translation)]
for each( Objective o  in list)
  o.intention()
end for
```

Determine Objective. The determine objective supports the translation goal and is called in the above pseudo code. Figure 4.9 shows the concept map for the determine objective. The agent is pattern that defines the structure for the byte code patterns. The matter is the binary object and describes the byte code sequences. For this study these sequences represent the Sobel algorithm.

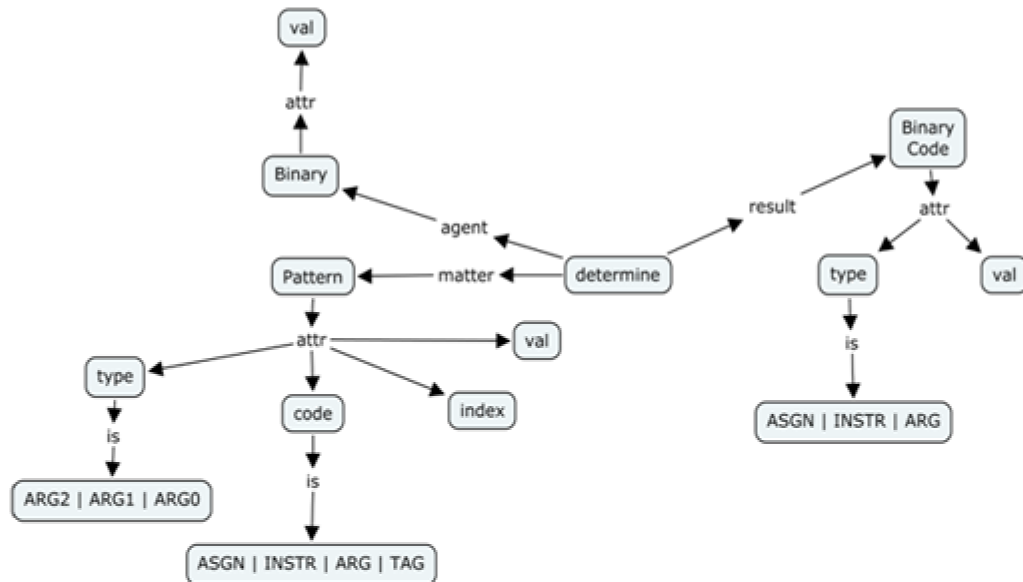

Figure 4.10: CMAP for the determine objective

Determine searches the knowledge base to determine the state of the objective. Next determine searches the knowledge base for behavior using state. Last determine then invokes the behavior (explained next sections). This continues until the last state returns true at which determine has finished.

The following pseudo code describes this behavior:

```
repeat
    State s = kb[=state(o)]   // where o is the current intention
    Behavior b = kb[behavior(s)]
    flag = b.exec()
until (flag=true)
```

Check Behavior The check progress computes and records the progress for the determine objective. The focus of the check behavior is to find out what pattern the binary sequence represents. Check supports five states (or stages). Two of the states (RDY, EXIT) support progress by recording role properties of agent and matter within each progress state and by recording the role result into the determine objective's measure.

Figure 4.10 shows the CMAP for the determine objective. It contains three knowledge objects supporting a role. For the agent the binary category contains instances representing the Java byte codes. For the matter the pattern category contains instances for each pattern type with the associated code types which include the indexes for assignments, instructions, arguments and tags (and tag value).

The check behavior focuses on the tag code type. A tag identifies the index for a unique sequenced value. For example, there are two tag values of interest: the byte code value for push (178) and pop (179). It is the position of the push and pop values within the sequence that determine a matched pattern. For example, for the pattern type ARG2, there are two pushes and one pop. Likewise, for pattern type ARG1, there are 1 push and 1 pop. And for ARG0, there is only one pop.

Figure 4.11 shows an example of instances in the knowledge base with a three byte binary sequence. The sequence represents a pattern type of ARG0. Also shown are the pattern instances for ARG0. The check behavior uses the index value for the TAG code type. For this example the index is 1 and the binary sequence of 1 has 179 as the value. Given that the sequence contains the pop value appropriately placed with the sequence, check determines a matched pattern.

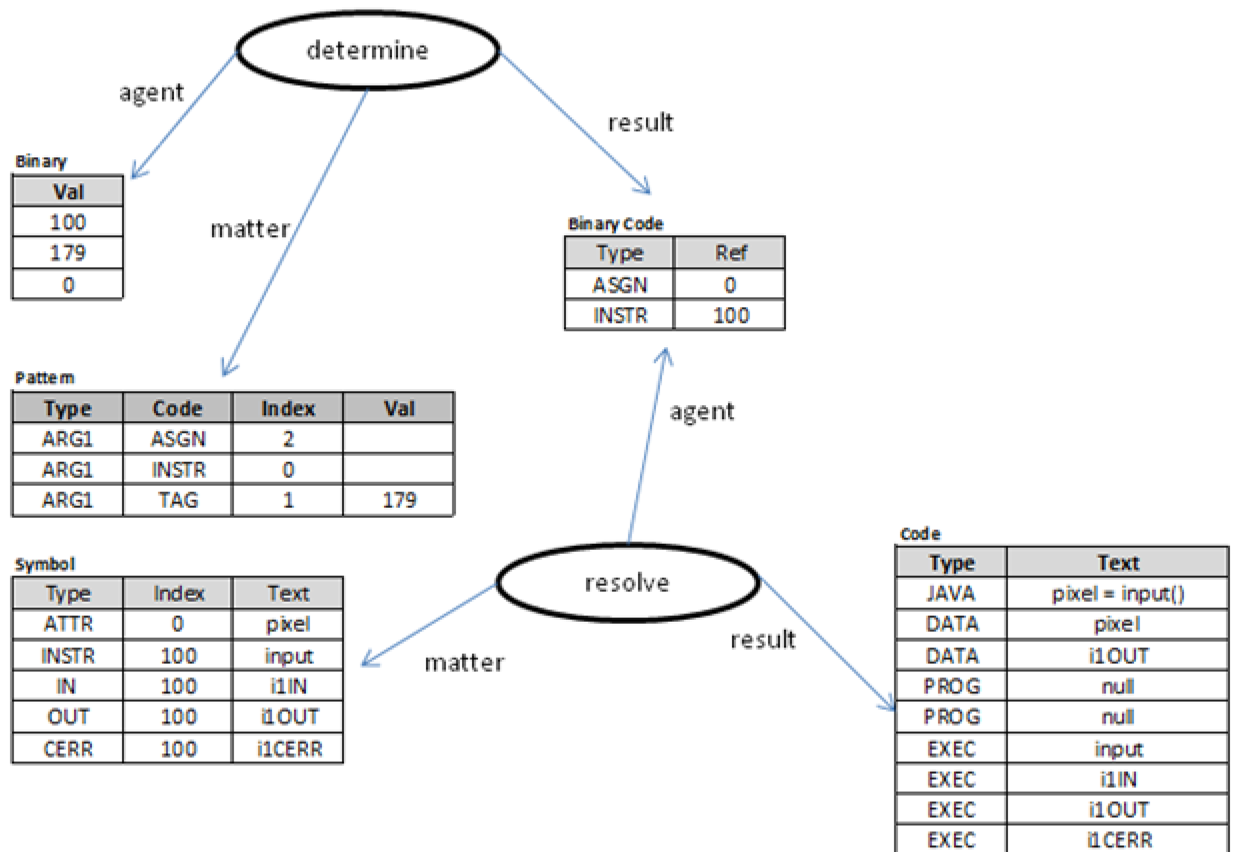

Figure 4.11: instance view supporting the determine and resolve objective

Matching a pattern check updates the progress results. As shown in the example figure the exit state contains this value. After matching, check sets the objective's state to exit. As mentioned above exit assumes the behavior for updating the determine objective's measure.

Obtain Behavior The obtain behavior attains binary sequences using the matched pattern indexes. This behavior acts on behalf of the determine objective and has six states. First the inherited knowledge behavior supporting Peirce's reaction begins by copying the measures role inputs into state's role input. Next the obtain behavior sequences through three states (ASGN, INSTR and ARGS); where each state behavior is similar. Specifically, a pattern

index is used to lookup the proper binary value.

For the example shown in figure 4.11, the binary sequence (100, 179 and 0) results in two instances in the knowledge base. As shown, the ASGN index from the pattern instance results with a 0 binary value. Likewise, the INSTR index from the pattern instance results with a 100 binary value. Each value and type is appropriately inserted in the knowledge base.

During the last state (STOP) obtain reacts by copying the progress result into the appropriate objective measure. By copying the measure results the obtain behavior completes.

Resolve Objective. In this section the author describes the second objective supporting the translation goal. The determine objective matches byte patterns to a specific Java pattern. The binary indexes are then collected and reserved in a binary pattern. Figure 4.12 shows the CMAP for the resolve objective. First the resolve objective uses the binary pattern in a symbolic lookup to determine associated text with the recognized byte code pattern. These symbolic patterns represent specific VHDL statements implementing an FPGA circuit.

Lookup Behavior The lookup behavior supports the resolve objective. The purpose of the resolve object is to use the binary code and symbol instances to obtain mnemonics. Figure 4.12 shows the resolve objective which has 3 knowledge objects. It contains the numerical values from the binary sequence that define the assignments, instruction and arguments that support the java statement represented by the byte codes.

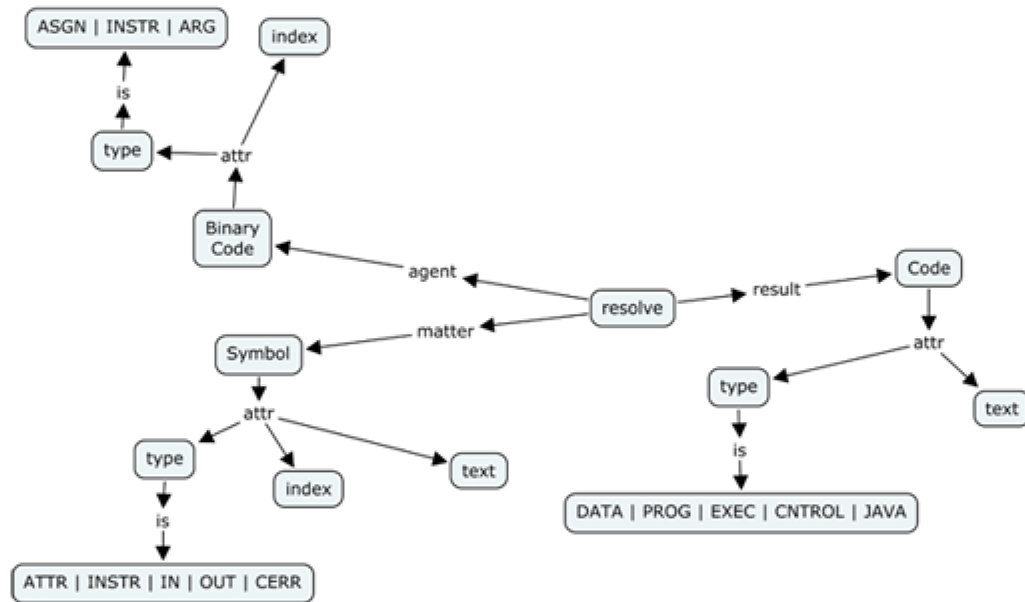

Figure 4.12: the resolve objective

In addition to the binary code there are the symbol instances representing symbols that comprise the Java attributes and instructions. Moreover, it contains key elements that are used to model VHDL code supporting the design command (more on this later).

Figure 4.11 provides an example with the binary code showing instances for assignment and instruction. The assignment index contains a zero, which is used to query the knowledge base for the attribute at zero. The text associated with this index is *pixel*. The instruction index is 100 and the text for the instruction *input*. The JAVA is:

pixel = input()

After the code type for Java, the HMC unit labels are put into the knowledge base. For a complete result refer the figure 4.11.

Design Goal

The design command supports the user's intention to create VHDL design files that directly support the real-time application. This goal has two objectives that model the application behavior into design specifics. After satisfying these two objectives these design instances are then folded into VHDL design templates that representing a HMC.

Initially the model object queries the knowledge base for instances that represent the translated application. These instances support the process where key items are parsed into design templates as instances into the knowledge base. Later these instances are gathered and folded into the HMC design files.

Model Objective. The model objective is the first objective that supports the design goal. Figure 4.13 shows the concept map for this objective. It has 3 knowledge objects. First there is the object for code which is the measured result for translation. It contains mnemonics supporting Java and VHDL design specifics.

Figure 4.15 shows knowledge instances supporting the model objective. From the agent the text field represents VHDL snippets that will be used to replace the tags in the when template representing the matter. The results are shown in the when template where the delineated areas represent a HMC unit.

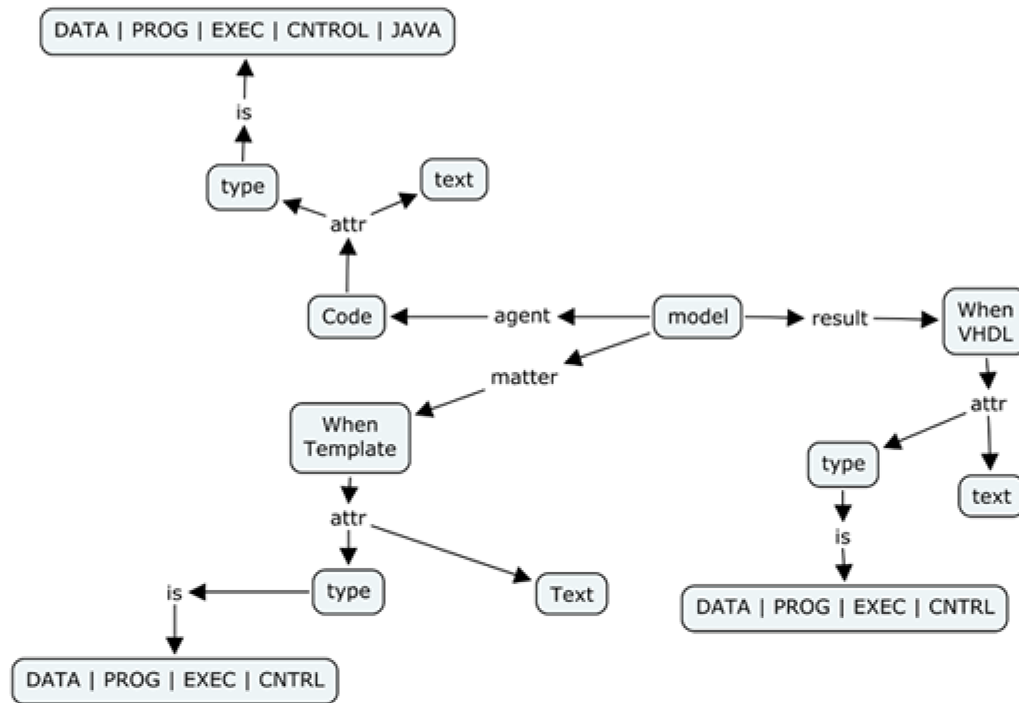

Figure 4.13: the model objective

Next there is the object for the when template. This object contains design specifics representing VHDL code. Within each template there are tags ([*]) that represent insertion points that mark areas within the template where code instances are replaced. Finally there is the object when VHDL. This object represents VHDL design specifics supporting each HMC unit. Each instance represents a VHDL when sequence, similar to a case statement in Java. Each when is indexed by the HMC program counter. In the next three sections, the author discusses the progress for the model objective.

Setup Behavior Setup establishes progress for the model objective. It is responsible for establishing the agents and matters for each state behavior. The agent represents the results from the translate goal while matter contains the when templates for each HMC unit. There

are four states that represent each HMC unit. For each state setup queries the code instances and asserts the unit's agent. Next setup queries the knowledge base for the unit's when template and asserts the unit's matter. Upon completion setup updates the objective's state.

The following pseudo code represents an example:

```
for each (Unit u in HMC)
    Code[] c = kb[=code(u)]
    WhenTemplate[] w = kb[=whentemplate(u)]
    assert(kb[=progress(u).agent( c )]
    assert(kb[=progress(u).matter(w)]
end for
assert([kb=objective(model).state(DATA)])
```

Replace Behavior The replace behavior establishes VHDL design specifics for translated code objects. Replace supports each HMC unit where the agent represents code instances and the matter represents when templates. Finally the results contain a complete VHDL design for each HMC unit.

The following pseudo code illustrates the replace behavior:

```
    Text result
    Int j=0
    Text[] x = agent(for text)
```

```
Text    y = matter(for text)

Int[] position = tags(y);

Int k = 0

for each (I in position)

    result += y(j,i)+x(k++)

    j=i+3

end for

assert(kb[=progress.result(result)])

assert(kb[=object(model).state(next unit)])
```

Transfer Behavior The transfer behavior supports the model objective, and cooperates with the replace behavior. It begins by querying the knowledge base for the replaced results and appropriately places the results as instances within the progress and measured areas.

The following pseudo code defines this behavior:

```
for each (Unit u in HMC)

   WhenTemplate[] w = kb[=Progress(u).result]

   assert (kb=[progress.result(w)])

   assert( kb=[measure.result(w)])

end for
```

Fold Objective. The fold objective defines the intention for design. Figure 4.14 shows the concept map. There are three knowledge objects that describe the objective. Acting as an agent is the when VHDL category. This category defines instances of the measure for the model objective. They contain VHDL labeling and Java specifics for the matched pattern. Also shown in the figure is the HMC template. This template contains a single knowledge base instance which contains a tagged representation of a complete VHDL program. There are four tags within the template where each tag represents a HMC unit position within the design file.

The last object is the VHDL program. This object represents the measured result for the folding objective. It contains a file image that can be assembled, compiled, routed and synthesized into an FPGA.

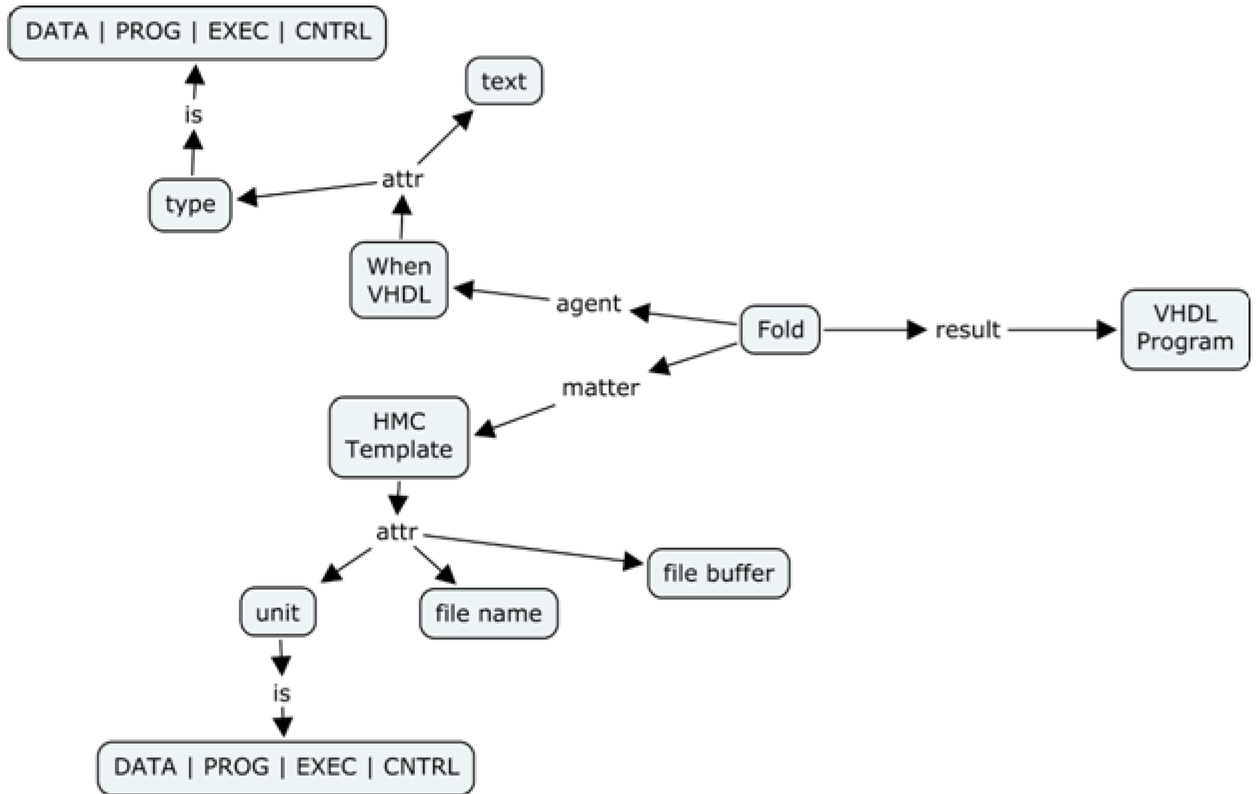

Figure 4.14: the fold objective

Setup Behavior This behavior supports the fold objective. By default this behavior responds and appropriately modifies both the agent and the matter for each HMC unit. It begins by gathering the HMC template. Next, with this instance and for every HMC unit, setup acquires the modeled results and asserts the agent. In addition it positions the HMC template into matter. Figure 4.16 shows the progress during the folding process. From this figure you can see each state with the appropriate behavior and role objects.

The following pseudo shows this behavior:

```
HMCTemplate h = kb[=HMCTemplate]

for each (unit u in HMC)
```

```
    WhenVHDL[] w = kb[=Progress(model).result(u)]

    assert(kb[=Progress(u).agent(w)]

    assert (kb[=Progress(u).matter(h)]

end for
```

Merge Behavior This behavior supports each unit of the HMC. The agent contains the instances of the unit's when VHDL objects. The matter contains the HMC template. Recall this template contains 4 tags (one for each HMC unit); these tags mark the position where the agents must be placed.

First merge responds by concatenating associated text for each agent object. This text represents the when statements associated with the VHDL when state. Similar to case in Java, the text would represent the concatenation of each case within the switch statement. In VHDL this would be each when with a case structure.

Next merge determines the position of the first tag. This position is used when merging the concatenated string into the HMC template. The resulting text after merging is asserted (similiar to inserting) into the unit's result. Finally merge must update the next unit's agent with the current result.

The following pseudo code shows this behavior:

```
Text result

Text buffer    = matter(for file buffer)

for each (WhenVHDL w in agent)

  C += w(for text)
```

```
end for

Int pos = matter(tag)

Result = buffer(0,pos) + c + buffer(pos+3,toend)

assert (kb[=progress(next state).agent(result)]

VHDLProgram vp = VHDLProgram(result)

assert (kb[=progress.result(vp)])
```

Exit Behavior Exit is the last state behavior. It acquires the VHDL program from the staged results in progress and updates the progress results. Then exit updates the measured results for fold objective.

At the start exit queries the progress results for the VHDL program. This object represents the staged results from each HMC unit. Next this behavior updates the fold's progress results with this the acquired object. Finally exit updates the measured results for the fold objective.

The following pseudo describes this process:

```
VHDLProgram vp = kb[=progress(process).result]

assert(kb[=progress.result(vp)]

assert(kb[=measure(fold).result(vp)])
```

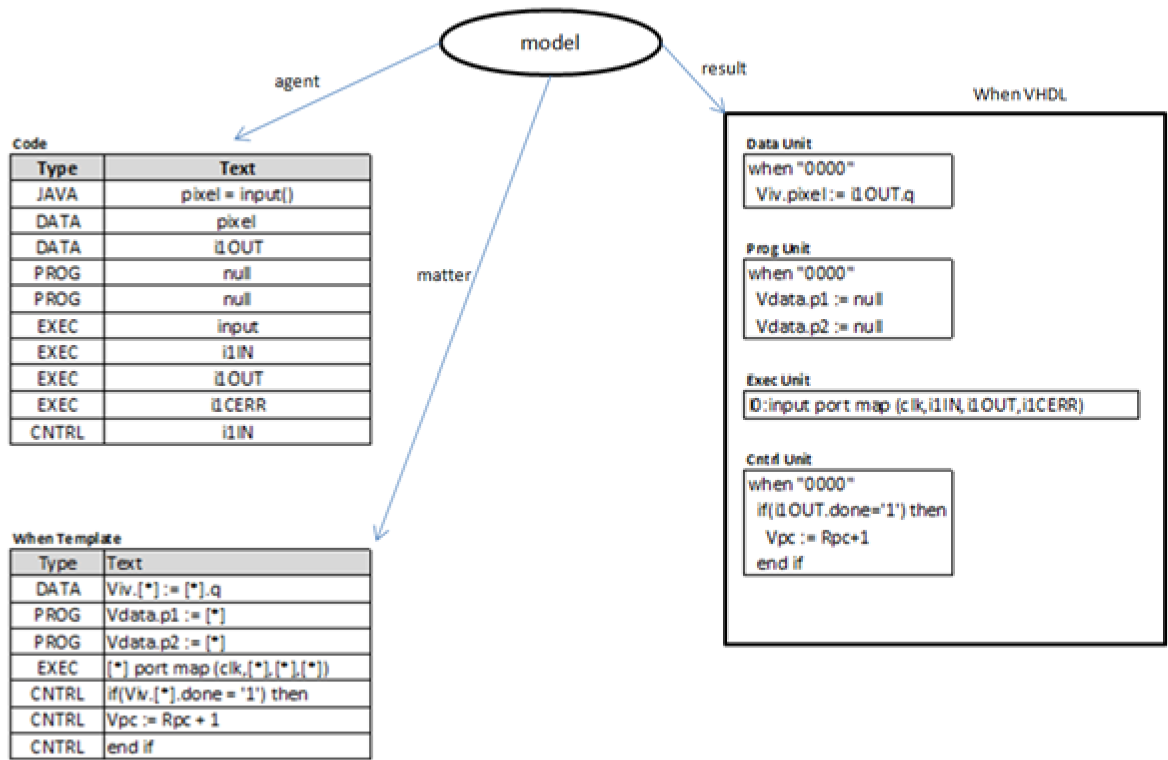

Figure 4.15: measure supporting model objective

Progress

State	Behavior	agent	matter	result
BEGIN	setup			
DATA	replace	Code	WhenTemplate	WhenVHDL
PROG	replace	Code	WhenTemplate	WhenVHDL
EXEC	replace	Code	WhenTemplate	WhenVHDL
CNTRL	replace	Code	WhenTemplate	WhenVHDL
END	transfer	kb[=WhenVHDL]		

Figure 4.16: progress supporting model objective

Chapter 5

FINDINGS

In this section the author presents the findings of the study's investigation. As mentioned in Chapter 2 the problem with Java soft chips, the designs are generally too complicated to support real-time applications. In particular these processors require cache, pipe-lining and instruction folding. The reification process in combination with an abstracted HMC design provides a simple design approach and supports instruction folding without any cache.

The test results focus on timing comparisons with other soft chips where the HMC is configured in two modes. First the HMC is configured for 16 bit mode. In this mode the HMC supports instruction folding and provides basic branching, looping and mathematical instructions. In the second mode the HMC runs in mixed mode where the HMC supports two word lengths for 16 and 144 bits.

5.1 Introduction

In the following sections the author provides the study's results. First the author describes the testing environment used to obtain these results. All circuits developed were synthesized using the Qartus II (version 9.1) integrated developement environment. This tool provided the environment to design, implement and test all circuits.

Moreover the vehicle used to support these circuits was an Altera FPGA prototyping system Cyclone II operating with a 50 MHz clock. This system supports 8 MB of SDRAM

and 512KB of SRAM. In addition to the two types of RAM, the chip also supports 4 MB of flash memory.

For display and control purposes the prototype system provides 8 seven-segment displays. In addition it supports 25 LEDs, 17 of which are red and 8 are green. There is a USB port which is used to synthesize circuits and there is a UART which is used to send and receive bit-mapped images.

To capture and display appropriate circuit signals the tool Signal Tap was used. This tool is part of the Qartus II environment. The tool provides a logic analyzer capability which allows the selection of specific signals to display. The tool uses triggers which are specific conditions which when reached the selected signals are displayed. Both the prototype and tool provide the environment to design and test all circuits. Figure 5.1 shows the the testing setup where a virtual bench sits between the reified application and the FPGA prototype system (or a simulated system). In this setup Signal Tap captures the appropiate signals much like what a logic analyzer does.

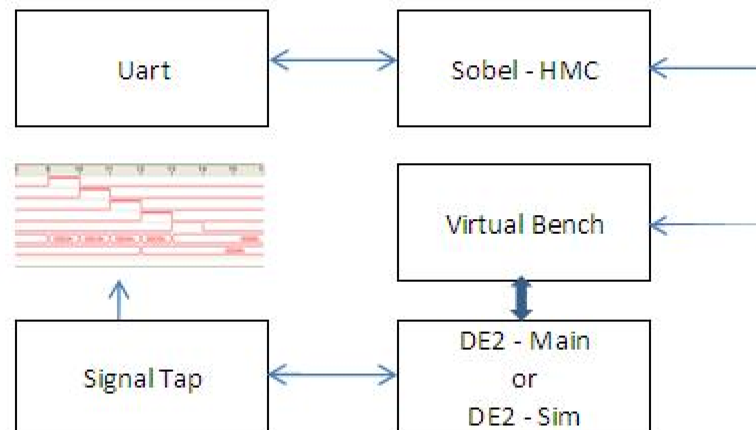

Figure 5.1: Testing Setup

5.2 FPGA Usage

In this section a resource comparison is made between other soft processors(see Figure 5.2) and the reified HMC soft processor designed in this study (noted by USM). The first two are the number of logic cells and memory used to implement a sample applicaiton.

The last comparison is the clock speed of each prototype system. It is important to note that the reified HMC is the only processor that supports Java and has no cache.

Processor	Resources (LC)	Memory (KB)	Frequency (MHz)
USM	855	4.8	50
JOP	1452	3.25	98
Lightfoot	3400	1	40
NIOS A	1828	6.2	120
NIOS B	2923	5.5	119
SPEAR	1700	8	80

Figure 5.2: Soft Core Processors

5.3 Simple Test

In this section the author presents the findings of the HMC running in 16 bit mode. There are two tests that show branching, indexing and calculating abilities of the HMC. Primarily it shows that each instruction takes 2 clock cycles. These findings show that the HMC is as fast as other chips running two clock cycles per instruction and it runs without any design enhancements.

5.3.1 Simple Test

The simple test represents a test for add, subtract and brahncing. It was used to demonstrate the 3 operand instruction of the Pico Java [3] and is similarly used in this study to show the

results from instruction folding. As mentioned above the Signal Tap tool is used to test the circuits, where key triggers are defined and appropriate signals are displayed. Figure 5.3 shows a simple Java program (shown on left side). The first line shows an assignment where *k1* is a static constant for the number 1. This simple program sets *A* to one, after *B* is given the result of adding one to *A*. The if statement fails and the program terminates with *B* is set to the results of subtracting 1 from *A*.

Signal Tap is setup to trigger when software switch is equal is set to 1. The setup continues by displaying the done flag associated with the Java statements. Figure 5.3 shows the output from signal tap immediately after the trigger is reached. The left column shows the Java program simple test code. The first five values represent the done flag and along the bottom shows the program counter. Notice the then path is not executed which is shown by the absence of the done flag, and is shown by the program counter (*PC*) skipping from three to five.

Also shown in this figure you may notice the top row where it showing the system clock. There is a black line appropriately placed when the test starts and extends through clock four. On line one there is a pulse at clock six which is the termination (done flag) at program counter equal to one. Also notice the test starts at clock four and terminates at clock six where the assignment statement takes two clock cycles to complete execution. On the rising edge of the done pulse at clock 6, the next instruction starts to execute the add instruction cycles. You will notice all instructions take two clock cycles.

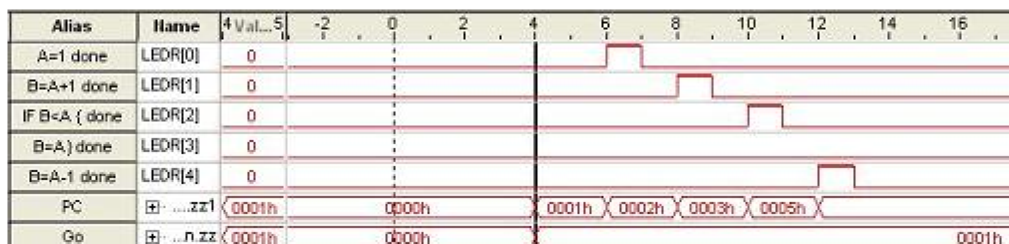

Figure 5.3: Simple Test - demonstrates instruction folding

Instruction	USM	JOP	leJOS	TINI	Komodo	JStamp	SaJe	Xint
iload iadd	2	2	836	789	8	38	8	17
iinc	2	11	422	388	4	41	11	1
if-icmplt taken	2	6	1609	1265	24	42	18	36
if-icmplt not taken	2	6	1520	1211	24	40	14	37
getstatic	2	17	1676	4463	80	102	15	40
putstatic	2	18	-	-	-	-	-	-

Table 5.1: Instruction timings

5.3.2 Bubble Sort

To examine the capability of the proposed HMC the author chose to look at the bubble sort alrogithm [8]. This program provides an example of looping and indexing in support of a 10 member array. The sort works by comparing adjacent pairs of elements and performing swaps is necessary. The worst case performance is $O(n^2)$

As mentioned above this study uses a 10 member array arranged in descending order. Figure 5.4 shows a real-time trace that shows the first swap happening around instruction number ten. The last figure (see figure 5.5) completes at instruction number 570 completing the sort.

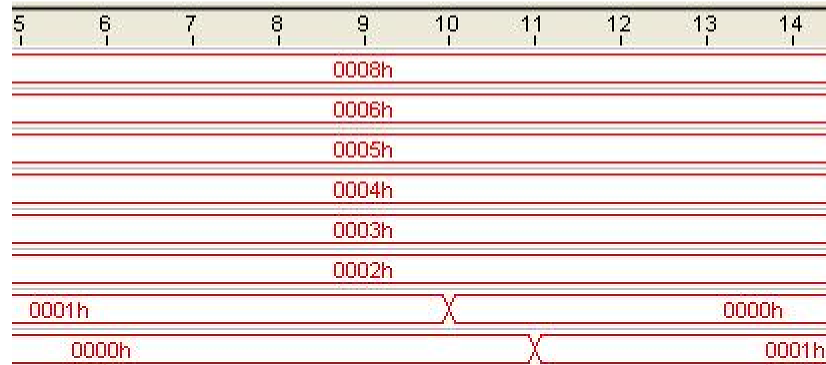

Figure 5.4: Bubble Sort Test During Start

The code for the bubble sort appears in the appendix. To perform the worst case case analysis of bubble sort we examine the code. The inner loop requires 12 instructions to complete and the total number of times through the loop is forty-five. This amounts to 540 instructions. This leaves the outer loop, which executes 10 times and has 3 instructions of overhead. Thus the bubble sort takes 570 instructions to complete.

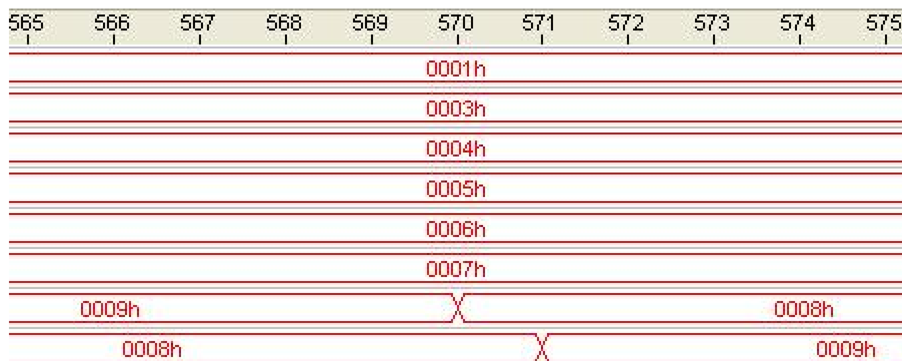

Figure 5.5: Bubble Sort Test at Completion

5.4 Sobel Edge Detection

In this section the author presents the findings of the HMC running in mixed mode. In this mode the HMC supports 16 and 144 bit word lengths. Two custom instructions are provided to support the larger bit length. For example there is a cross multiplier where corresponding

positions of two 3x3 matrices are multiplied in 2 clock cycles. The findings are compared to JOP [23] which is the only Java soft chip that which claims to support real-time applications. The WCET time for JOP is obtained from the instruction timings.

5.4.1 Reified Sobel Edge Detection

Recall the equations from 4.2 to 4.7 where they show 2 controlling loops. For this study there are two custom instructions that replace the looping instructions for these equations. The first instruction provides a 3x3 cross multiply, where each member in the matrix is multiplied by the cooresonding member in the second matrix. The result is a 3x3 matrix where the nine multiplications are performed in parallel. The total time to complete is 2 clock cycles. Next there is a 3x3 sum which is performed in parallel and the execution time is two clock cycles.

These two instructions are added to the reified insttruction set, where the reification process aligns the instruction to Java methods. Thus the two instructions replace the corresponding loops. The speedup is around five hundred when processing an image.

This section shows the WCET times for the Sobel edge detection algorithm using these two instructions. Figure 5.6 shows the code for this algorithm where the left column shows the program counter. The critical path of interest begins at pc 1 and ends with pc 9. This code represents the non I/O statements.

Figure 5.7 shows the timings of the Sobel edge detection. Notice at the bottom where it starts with 1 and then proceeds to 9. These transition represent the program counter (PC). The instructions at these locations represent the instructions between the input and output

statements. This is the critical path where the analysis is focused.

Notice the pc increments and the time that each increment takes. This is done by projecting each transition to the top of the figure where the time sequences are presented. You will notice that each instruction at these pc values takes 2 clock cycles. Thus the critical path can be observed to take 18 clock cycles.

In addition the figure shows 3 pulses. These pulses represent the completion of the multiply, sum and absolute value instructions. The first groupings represent the calculations in the x-direction and they are repeated for the y-direction. Each rising edge terminates one instruction and immediately begins execution of the following instructions.

```
pc 0 pixel = input()
pc 1 product = mul (pixel,Gx)
pc 2 sumx = sum(product)
pc 3 sumx = absval(sumx)
pc 4 product = mul (pixel,Gy)
pc 5 sumy = sum(product)
pc 6 sumy = absval(sumy)
pc 7 sum = sumx + sumy
pc 8 sum = ifgt(sum,255)
pc 9 sum = iflt(sum,1)
pc a output(sum)
```

Figure 5.6: Sobel Edge Detection Algorithm

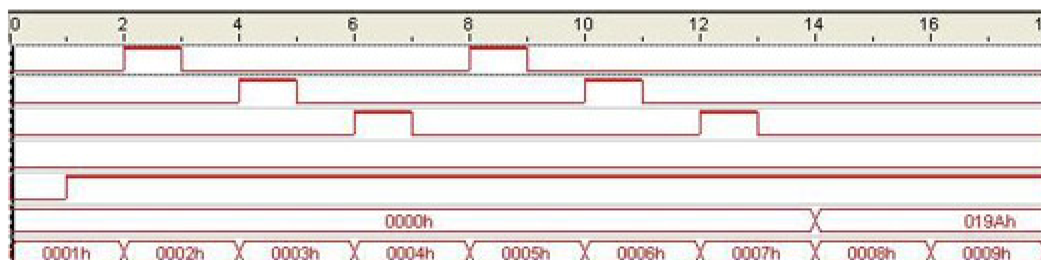

Figure 5.7: Sobel observed WCET

5.4.2 JOP - Sobel Edge Detection

The next examination takes the Sobel byte codes written in Java where the author examines the byte codes. Recall the equations from 4.2 to 4.7 where there are two main loops required to complete the critical path. Table 5.8 shows the two loops and their associated instructions cycles for each byte code. At the bottom is the total number of clock cycles required to execute the loop one time. For this study the results shows it takes 126 clock cycles to execute the loop 1 time. Thus, the total time to compute a pixel would be 1134 clock cycles.

Figure 5.7 shows that the total time for the critical path is 18 clock cycles. This critical path represents the time between reading the pixel matrix and outputting the Sobel value. This results in a speeup of 63 compared with JOP. Looking at the results from Puffitsch [20] there is an estimated 28 percent increase on performance when measured using cache in JOP. Appying this increase in performance JOP would execute in 816 clock cycltes. Thus the speedup of 45 as compared with the reified approached in this study.

Byte Code	# Clocks
12 iconst_m1	1
13 istore 6 [a]	1
15 goto 52	4
18 iconst_m1	1
19 istore 7 [b]	1
21 goto 43	4
24 iload_1 [sumx]	1
25 iload 4 [x]	1
27 iload 5 [y]	1
29 imul	35
30 iadd	1
31 istore_1 [sumx]	1
32 iload_2 [sumy]	1
33 iload 4 [x]	1
35 iload 5 [y]	1
37 imul	35
38 iadd	1
39 istore_2 [sumy]	1
40 iinc 7 1 [b]	11
43 iload 7 [b]	1
45 iconst_2	1
46 if_icmplt 24	4
49 iinc 6 1 [a]	11
52 iload 6 [a]	1
54 iconst_2	1
55 if_icmplt 18	4
Total inside loop	*126*
Total for 3x3 loop	1134

Figure 5.8: Sobel byte codes for pixel

Chapter 6

Conclusions and Future Research

This work expanded upon the Pico Java processor where there was an investigation into design complexity versus speed. The goal was to eliminate the complexity while maintaining the speed advantages. The previous chapter shows this was accomplished and surpassed the original goal where there was a significant speedup as compared to the only Java soft processor claiming real-time abilities.

The investigation contributions are as follows:

1. A cache-less HMC with instruction folding provided the speed required for real-time support without using cache or pipe-lined instructions which would complicate the computation of WCET. A primary concern during the investigation was to support the basic instruction set with 2 clock cycles per instruction. The findings show that the simple test and bubble sort test validated the maintenance of this speed. The basic instructions for comparing, branching, indexing, adding and subtracting showed consistent timings of 2 clock cycles per instruction.

2. A reified HMC was the basic design approach taken in this investigation. It provided for initialization, execution and control supporting Java applications using three operand instructions. It supported an abstract approach to data and program organization aligned with the application boundaries based on 16 and 144 bits.

3. A soft bus approach was taken during the investigation. This allowed for mixing word lengths permitting significant parallelizations. Two instructions were added that supported 3x3 matrix objects. The first instruction performed a cross multiply where two 3x3 objects were multiplied in parallel producing a 3x3 result. All nine multiplies were executed in parallel in the 2 clock cycle instruction cycle. In addition a sum instruction was added that summed the values in a 3x3 matrix producing a 16 bit result in 2 clock cycles. The soft bus allowed for this word mix and it protected the 16 bit basic instruction set integrity. This approach was demonstrated and validated by the Sobel Algorithm and compared with the only Java processor claiming real-time abilities. The findings show a significant speedup with the existing processor with cache enhancements.

4. Reification, a prototyped ontology, was used as the glue in this investigation. It glued the above items together as a unified, general purpose Java soft chip able to support real-time applications. It folded the byte codes into a reified HMC supporting parallel operations of the FPGA.

This investigation is about a Java processor operating in an FPGA environment supporting real-time applications. A subset of Java was used that closely resembles Integer Java (IJVM) . Both the subset of Java and IJVM do not support the complete Java virtual machine, though there is no technical problem with doing so.

Important Java features not included in this study include the ability to define local variables, the use of parentheses and the invocation of class methods. These features were

not necessary to prove the feasibility of supporting real-time applications in a Java soft processor.

The HMC design was modified to take advantage of the FPGA environment to support parallelization. A change was made to the addressing scheme. Rather than by referencing data position, the program counter was used to provide better alignment with the application. These changes do not represent the general purpose nature of a HMC.

The Java class byte codes were manually asserted into the knowledge base and the class files are not read directly. The class attributes were defined using static references which forced the appropiate byte codes for *getstatic* and *putstatic*. These instructions codes were used to identify instruction patterns. Last the HMC had to be manually synthesized because of development environment restrictions.

6.1 Future research

This investigation lays a solid foundation for Java real-time support in an FPGA environment. One possible improvement to the design is to expand the byte code support to the complete IJVM. Other possible improvements are to support local variables and to allow the use of parentheses.

It is possible to double the performance by eliminating the control unit and having each instruction perform aspects of the control unit. Doing so would allow for instruction execution in 1 clock cycle. In addition this would reduce FPGA resource count by about 15 per cent.

Further improvements would be to expand the process to include a reified class and a

reified group. By expanding a class each method could operate in its own processor. The class attributes would have to support arbitration so that each method could compete for shared data. Next a reified group would help organize the process and provide the ability to share data between reified classes.

This investigation describes a sensible approach to reification of a Java soft processor for supporting real-time applications. The resultant design provides substantial performance improvements of the existing real-time soft processors with a design supporting simple determination of worst case execution time. This research has identified several possible improvements which will form the basis of future research.

Appendix A

VHDL INSTRCUTIONS AND BUBBLE SORT

A.1 VHDL Instructions

A.1.1 Absolute Value

```
process (r,cin,p1OUT) is
variable v : std_logic_vector (i downto j);
variable x : std_logic;
begin

if (cin.go='1' and cin.p(i)='1') then
  x := '1';
  v := conv_std_logic_vector((-conv_integer(cin.p)),i+1);
else
  v := cin.p;
end if;
p1IN.go    <= cin.go;
rin        <= v;
cout.q     <= r;
cout.v     <= v;
cout.done  <= p1OUT.q;
cout.vdone <= x;
end process;

process (clk,rin,r99in) is
begin
    if (rising_edge(clk)) then
        r  <= rin;
        r99 <= r99in;
    end if;
end process;
cout.q    <= r;
cout.done <= p1OUT.q;
end absval_main;
```

A.1.2 Add Instruction

```
process (r,cin,p1OUT) is
variable v : std_logic_vector (i downto j);
begin
if (cin.go='1') then
  v := cin.p1+cin.p2;
else
  v := r;
end if;
p1IN.go   <= cin.go;
cout.q    <= r; -- note using v and not r.
cout.done <= p1OUT.q;
rin  <= v;
end process;

process (clk,rin,r99in) is
begin
    if (rising_edge(clk)) then
        r   <= rin;
        r99 <= r99in;
    end if;
end process;

end add_main;
```

A.1.3 Limit Greater Than Instruction

```
 process (r,cin) is
variable v : std_logic_vector (i downto j);
begin

v := r;
if (cin.go='1' and cin.p1 > cin.p2) then
 --   p1IN.go <= '1';
v := k255;
else
 --   p1IN.go <= '0';
v := cin.p1;
end if;
rin <= v;
end process;

process (clk,rin,r99in) is
begin
    if (rising_edge(clk)) then
        r   <= rin;
        r99 <= r99in;
    end if;
end process;
p1IN.go   <= cin.go;
cout.q    <= r;
cout.done <= p1OUT.q;
end limitgt_main;
```

A.1.4 Limit Less Than Instruction

```
 process (r,cin) is
variable v : std_logic_vector (i downto j);
begin

v := r;
p1IN.go <= cin.go;
if (cin.go='1' and cin.p1 < cin.p2) then
v := kzero;
else
v := cin.p1;
end if;
rin <= v;
end process;

process (clk,rin,r99in) is
begin
    if (rising_edge(clk)) then
        r   <= rin;
        r99 <= r99in;
    end if;
end process;
cout.q    <= r;
cout.done <= p1OUT.q;
end limitlt_main;
```

A.1.5 3 x 3 Cross Multiply Instruction

```
 process (r,cin,p1OUT) is
variable v : pixel;
begin

v := r;
if (cin.go='1') then
  p1IN.go <= '1';
  v.p0 := conv_std_logic_vector(conv_integer(cin.p1.p0)*conv_integer(cin.p2.p0),i+1);
  v.p1 := conv_std_logic_vector(conv_integer(cin.p1.p1)*conv_integer(cin.p2.p1),i+1);
  v.p2 := conv_std_logic_vector(conv_integer(cin.p1.p2)*conv_integer(cin.p2.p2),i+1);
  v.p3 := conv_std_logic_vector(conv_integer(cin.p1.p3)*conv_integer(cin.p2.p3),i+1);
  v.p4 := conv_std_logic_vector(conv_integer(cin.p1.p4)*conv_integer(cin.p2.p4),i+1);
  v.p5 := conv_std_logic_vector(conv_integer(cin.p1.p5)*conv_integer(cin.p2.p5),i+1);
  v.p6 := conv_std_logic_vector(conv_integer(cin.p1.p6)*conv_integer(cin.p2.p6),i+1);
  v.p7 := conv_std_logic_vector(conv_integer(cin.p1.p7)*conv_integer(cin.p2.p7),i+1);
  v.p8 := conv_std_logic_vector(conv_integer(cin.p1.p8)*conv_integer(cin.p2.p8),i+1);
else
 p1IN.go <= '0';
end if;
cout.q <= r;
cout.done <= p1OUT.q;
rin <= v;
end process;

process (clk,rin,r99in) is
begin
    if (rising_edge(clk)) then
        r   <= rin;
        r99 <= r99in;
    end if;
end process;
cout.q    <= r;
cout.done <= p1OUT.q;
end smul_main;
```

A.1.6 3 x 3 Sum Instruction

```
 process (r,cin,p1OUT) is
variable v : std_logic_vector (i downto j);
begin

v := r;
if (cin.go='1') then
  p1IN.go <= '1';
  v := cin.p.p0+cin.p.p1+cin.p.p2+cin.p.p3+cin.p.p4+cin.p.p5+cin.p.p6+cin.p.p7+cin.p.p8;
else
  p1IN.go <= '0';
end if;
rin <= v;
end process;

process (clk,rin,r99in) is
begin
    if (rising_edge(clk)) then
        r   <= rin;
        r99 <= r99in;
    end if;
end process;
cout.q    <= r;
cout.done <= p1OUT.q;
end ssmul_main;
```

A.1.7 Sobel Edge Detection

```
 process (Rpc,Riv,Rdata,i1OUT,Q,S1OUT,S2OUT,A1OUT,O1OUT,X) is
variable Vpc : std_logic_vector (3 downto 0);
variable Viv : sobelREG;
variable Vdata : dataREG;
variable z : pixel;
begin
Vpc   := Rpc;
Viv   := Riv;
Vdata := Rdata;
--z := i1OUT.q;
--z := X;
case (Rpc) is
when "0000" =>
Viv.pix := i1OUT.q;
--z := i1OUT.q;
  if (i1OUT.done='1') then
    Vpc := "0001";
    Vdata.p1 := i1OUT.q;
    Vdata.p2 := Q;
  end if;
when "0001" =>
  Viv.product := s2OUT.q;
  if (s2OUT.done='1') then
    Vdata.p1 := s2OUT.q;
    Vpc := "0010";
  end if;
when "0010" =>
Viv.sumx := s1OUT.q;
  if (s1OUT.done='1') then
    Vdata.d1 := s1OUT.q;
    Vpc := "0011";
  end if;
when "0011" =>
  Viv.sumx := a1OUT.q;
  if (a1OUT.done='1') then
    Vdata.p1 := Riv.pix;
    Vdata.p2 := Q1;
    Vpc := "0100";
  end if;
when "0100" =>
  Viv.product := s2OUT.q;
  if (s2OUT.done='1') then
    Vdata.p1 := s2OUT.q;
    Vpc := "0101";
  end if;
when "0101" =>
Viv.sumy := s1OUT.q;
  if (s1OUT.done='1') then
    Vdata.d1 := s1OUT.q;
    Vpc := "0110";
  end if;
when "0110" =>
  Viv.sumy := a1OUT.q;
  if (a1OUT.done='1') then
    Vdata.d1 := Riv.sumx;
    Vdata.d2 := a1OUT.q;
    Vpc := "0111";
  end if;
when "0111" =>
  Viv.sum := a2OUT.q;
  if (a2OUT.done='1') then
    Vdata.d1 := a2OUT.q;
    Vdata.d2 := Riv.k255;
    Vpc := "1000";
  end if;
when "1000" =>
  Viv.sum := l1OUT.q;
  if (l1OUT.done='1') then
    Vdata.d1 := l1OUT.q;
    Vdata.d2 := Riv.k1;
    Vpc := "1001";
  end if;
when "1001" =>
  Viv.sum := l2OUT.q;
  if (l2OUT.done='1') then
    Vdata.d1 := l2OUT.q;
    Vpc := "1010";
  end if;
when "1010" =>
  if (o1OUT.done='1') then
    Vpc := "0000";
  end if;
when others => null;
end case;
if (cin.reset='1') then
  Viv.k255 := k255;
  Viv.k1   := kone;
end if;
Rpcff   <= Vpc;
```

```
Rivff   <= Viv;
Rdataff <= Vdata;

Xff <= z;

end process;
```

A.2 Bubble Sort

```
    public void sort () {
      for (i=k0;i<klen;i+=k1) {
for (j=k9;j>i;j-=k1) {
  t = j-k1;
  x = get(j);
  y = get(t);
  if (x < y) {
    h = get(t);
    zz = get(j);
    put (t,zz);
    put (j,h);
  }
}
}
}
```

BIBLIOGRAPHY

[1] *The essential Peirce: selected philosophical writings.* 1992.

[2] Georg Acher. Jiffy: Portierung eines jit-compilers auf fpgas. In *Java-Informations-Tage*, pages 26–35, 1999.

[3] J Michael O Connor and Marc Tremblay. PICO JAVA I : THE JAVA VIRTUAL MACHINE IN HARDWARE. *Ieee Micro*, (April):45–53, 1997.

[4] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[5] James Geller. John sowa, knowledge representation: Logical, philosophical, and computational foundations, brooks/cole, 2000, 512 pp., $70.95 (cloth), isbn 0-534-94965-7. *Minds Mach.*, 13:441–444, August 2003.

[6] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, N. Salami, N. Kajihara, and N. Otsu. Real-world applications of analog and digital evolvable hardware. *IEEE Transactions on Evolutionary Computation*, 3(3):220–235, 1999.

[7] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[8] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[9] Hermann Kopetz. Software Engineering for Real-Time : A Roadmap Software Engineering for Real-Time : A Roadmap. *System*, 1982.

[10] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[11] J. Kreuzinger, R. Marston, and T. Ungerer. The Komodo project: thread-based event handling supported by a multithreaded Java microcontroller. *EUROMICRO*, pages 122–128 vol.2, 1999.

[12] Manfred Krifka. Thematic relations as links between nominal reference and temporal constitution. pages 29–53. CSLI Publications, 1992.

[13] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[14] S.M. Loo, B.E. Wells, N. Freije, and J. Kulick. Handel-C for rapid prototyping of VLSI coprocessors for real time systems. *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory (Cat. No.02EX540)*, pages 6–10.

[15] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1st edition, June 1999.

[16] Mehrdad Najibi, Kamran Saleh, Mohsen Naderi, Hossein Pedram, and Mehdi Sedighi. Prototyping globally asynchronous locally synchronous circuits on commercial synchronous fpgas. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, RSP '05, pages 63–69, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Ian Niles, Adam Pease, and Adam Pease (presenter. Linking lexicons and ontologies: Mapping wordnet to the suggested upper merged ontology. In *In Proceedings of the 2003 International Conference on Information and Knowledge Engineering (IKE 03), Las Vegas*, pages 23–26, 2003.

[18] Joseph D. Novak. The theory underlying concept maps and how to construct them. Technical report, 2006.

[19] David Pellerin and Scott Thibault. *Practical fpga programming in c*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2005.

[20] W. Puffitsch and M. Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 213–221. ACM New York, NY, USA, 2007.

[21] Martin Schoeberl. Design and implementation of an efficient stack machine. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 159b–159b. IEEE, 2005.

[22] Martin Schoeberl. Evaluation of a Java processor. *Tagungsband Austrochip*, 2005.

[23] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[24] J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole Publishing, Pacific Grove, 2000.

[25] J.F. Sowa. A dynamic theory of ontology. In *Proceeding of the 2006 conference on Formal Ontology in Information Systems: Proceedings of the Fourth International Conference (FOIS 2006)*, pages 204–213. IOS Press, 2006.

[26] John F. Sowa. Conceptual graphs. In *Information Processing in Mind and Machine*, pages 39–44. Addison-Wesley, 1984.

[27] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[28] L.M. Surhone, M.T. Tennoe, and S.F. Henssonow. *Picojava*. VDM Verlag Dr. Mueller AG & Co. Kg, 2010.

[29] Richard Swan, Anthony Wyatt, Richard Cant, and Caroline Langensiepen. Re-configurable computing. *Crossroads*, 5, April 1999.

[30] Justin L Tripp, Kristopher D Peterson, Jeffrey D Poznanovic, Christine M Ahrens, Christine Ahrens, and Maya B Gokhale. TRIDENT : AN FPGA COMPILER FRAMEWORK FOR FLOATING-POINT ALGORITHMS. *Synthesis*, 836, 2005.

[31] Sascha Uhrig and Jörg Wiese. jamuth: an ip processor core for embedded java real-time systems. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, pages 230–237, New York, NY, USA, 2007. ACM.

[32] Richard Wain, Ian Bush, Martyn Guest, Miles Deegan, Igor Kozin, and Christine Kitchen. An overview of FPGAs and FPGA programming ; Initial experiences at Daresbury. *Engineering*, (November), 2006.

[33] Wikipedia. Field-programmable gate array, 2011.

[34] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

[35] Wayne Wolf. *FPGA-Based System Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.