Towards Efficient Execution of a GALS MoC Based System Level Language

Muhammad Nadeem

Supervisors: Dr. Morteza Biglari-Abhari
Prof. Zoran Salcic

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Electrical and Electronic Engineering, The University of Auckland

Abstract

Embedded systems include a broad range of electronic systems, from household appliances to safety critical systems such as medical systems, automotive electronics and avionics. Due to growing complexity of the applications, these systems consist of a number of computational intensive units running concurrently. They also interact with each other and with environment, repeatedly reading inputs, doing computations and responding appropriately. These computational units may have different response times; hence, they may need to run concurrently at different speeds. These systems may be considered as GALS (Globally Asynchronous Locally Synchronous), which typically consist of a collection of components that execute concurrently and communicate using possibly slow or unreliable channels. SystemJ is a system level programming language based on GALS model of computation allowing the asynchronous coupling of synchronous reactive modules at the top level, which execute at different speeds. It extends Java with Esterel-like constructs for the synchronous concurrency and reactivity, and CSP-like constructs for the asynchronous concurrency. SystemJ targets a large range of heterogeneous embedded systems that combine data-intensive and control-dominated computations (heterogeneous) in addition to synchronous and asynchronous concurrency.

Although, the problem of modeling complex systems has largely been solved by raising level of abstraction, there is still need for efficient execution platforms to realize such applications. While, there have been efforts towards supporting heterogeneous applications, they primarily focused on the reactive part of applications. In summary, developing architectural support for heterogeneous embedded applications has been the main focus of this research.

This thesis proposes improvements to some existing architectures as well as developing new architectures that make use of the formal underlying structure of the language to achieve higher execution efficiency in the GALS paradigm. These architectures execute control and data-driven operations along with asynchronous and synchronous concurrent processes in an efficient way. Our novel solutions range from extensions to a single CPU architecture to multiprocessor architectures, all while considering computational demands and resource constraints. We started with the deployment of Java Optimized Processor (JOP) to execute SystemJ programs compiled to Java and improved it by extending its architecture to include the reactive features. We suggested novel architectures which efficiently execute the SystemJ programs compiled by separating control and data-operations on a single core by embedding control operations inside the data-operation represented in Java by translating them to custom bytecodes. We refined this approach by providing two separate modes of execution for control and data-oriented operations. This approach is further extended to multiprocessor architecture to speed up the execution and meet the computational demands required by high-end embedded systems. We experimentally evaluated all proposed architectures to validate their effectiveness. Better performance, lower code density and resources usage have been achieved compared to previous approaches for SystemJ execution, thus proving its suitability for heterogeneous embedded applications.

To my parents and family

Acknowledgements

First and foremost I offer my sincerest gratitude to my supervisors, Dr. Morteza Biglari-Abhari and Prof. Zoran Salcic, who have supported me thoughout my thesis. Their time, expertise, support and guidance have been invaluable and appreciated. I attribute the level of my Doctor of Philosphy degree to their encouragement and effort and without them this thesis, too, would not have been completed or written. The joy and enthusiasm they had for the research was contagious and motivational for me, even during tough times.

My time at University of Auckland was made enjoyable in large part due to the many friends and groups that became a part of my life. I am grateful for time spent with roommates, friends, and many other people.

Last but not least, my deepest thanks to my parents and family for their continuous support throughout these years. Their understanding and support ecouraged me to do my best in Ph.D research.

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

AGRC	Asynchronous Graph Code
AJT	Assembly to Java Translator
ASIC	Application Specific Integrated Circuit
ATT	Average Tick Time
APS	Asynchronous Protocol Stack
BCL	Bytecode Engineering Library
CA	Control Array
CAB	Control Array Base
CD	Clock-domain
CISC	Complex Instruction Set Computer
cl	combinational lock
CM	Control Memory
CMP	Chip Multiprocessor
CP	Control Processor
CRCF	Concurrency and Reactive Control Flow
CSP	Communicating Sequential Processes
CVM	Control Virtual Machine
dl	demoloop
DM	Data Memory
DP	Data Processor
DSP	Digital Signal Processing
EOT	End of Tick
ER	Environment Ready
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GALS	Globally Asynchronous Locally Synchronous
GPP	General Purpose Processor
JAN	Java Action Node

JCF	Java Control Flow
JOP	Java Optimized Processor
JPC	Java Program Counter
JTG	Jump Table Generator
JVM	Java Virtual Machine
J2ME	Java 2 Platform, Micro Edition
LE	Logic Element
LT	Level Tracker
LTD	Level Tracker Decoder
MCA	Main Continuous Address
MoC	Model of Computation
PC	Program Counter
PM	Program Memory
RAM	Random Access Memory
ReCOP	Reacive CO-Processor
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RF	Register File
RJOP	Reactive Java Optimized Processor
SOC	System-on-Chip
TOS	Top Of Stack
TP	Tandem Processor
TVM	Tandem Virtual Machine
VLIW	Very Large Instruction Word
WCET	Worst Case Execution Time
WCRT	Worst Case Reaction Time

Chapter 1

Introduction

We are living in a world which is evolving at a faster pace than ever before. This is for a large part, happening since we stepped into the digital revolution marking the beginning of the information age changing the analog mechanical and electronic technology into digital technology. Central to this never-ending era is the mass production of the digital logic circuits which is at the heart of the computer. Almost every modern home and workplace contains many computers. Some of these computers are immediately recognizable as laptops and desktop PCs, but many are hidden within intelligent devices that use software to implement functionality. These computers are embedded systems.

An embedded system is a combination of computer hardware and/or software, and perhaps additional mechanical or other parts, designed to perform a dedicated function [1]. The cellular smart phones and tablets are couple of recent examples which have widespread use and significant impact on our daily lives. There are almost no areas of modern technology which could do without embedded systems. They appear in all areas of industrial applications and process control, in cars, in home appliances, entertainment electronics, cellular phones, video and photo cameras, and many more places. Embedded systems are becoming increasingly prevalent in everyday life. The growth rate in embedded systems is more than 10% per annum and it is forecast that by 2020 there will be over 40 billion devices (5 to 10 embedded devices per person on earth) worldwide. Today 20% of the value of each car is attributed to embedded electronics and this will increase to an average of 35-50% by 2020. The market for embedded computer systems, which already generates more than US$1 trillion in revenue annually, will double in size over the next four years, according to a report released by research company IDC [2].

The data in Figure 1.1 shows that the embedded computers dominate the computer market and have already surpassed traditional PC market share in terms of number of units.

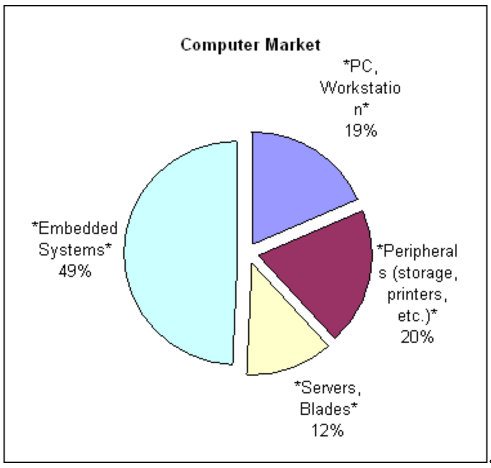


Figure 1.1: ASIC prototypes and verification on applications destined for the embedded systems market vs the traditional PC market

1.1 Characteristics of Modern Embedded Systems

The embedded systems are deployed to execute a wide range of ever changing applications. The nature of application is a ever changing phenomenon and new applications keep on emerging, including many that have not yet been envisioned, with features which were never the characteristics of the traditional embedded system. Recently, application complexity has increased significantly and traditional design approaches are unable to cope with these demands.

1.1.1 Computation Power

The semiconductor technology has grown rapidly resulting in larger die sizes and shrinking feature sizes largely surfing on Moore's law [3]. This enormous rise in computation power has enabled the designer to realize the computational intensive applications on the embedded systems that were not possible a few years ago. Compute-intensive is a term that applies to any computer application that demands a lot of computation. Previously, embedded systems were limited in computational capability, memory size, and power consumption, the focus of the research was to make the best use of the limited system resources. The system performance issues, such as execution time, were traded off with system resources. Modern embedded devices are usually equipped with significant computation capabilities. With more available computational capability in embedded system devices, and more complicated requirements demanding more intensive computation, the most critical design concerns are changing in some important application domains. Now, it is possible to produce chips with much more logic tremendously increasing the computational power available in the appliances. But to realize this performance, we need to apply smarter methods to improve system execution time.

1.1.2 Interaction with the Environment

Modern embedded systems interact with the environment more than ever making them reactive in nature, which are different from transformational and interactive systems.

Transformational Systems

The transformational systems, on providing the input, compute the output and then stop. These systems acquire some data as input at the beginning of execution and produces some data as a result, on termination. A compiler is a typical example of transformational system.

Interactive Systems

The interactive systems communicate with the environment at their own speed. They get the input from the environment and compute the results. The environment has to wait for the result before entering the input. An operating system is a typical example of interactive system.

Reactive Systems

The reactive systems [4] are also interactive but slightly differ in the sense that they have to respond at a speed dictated by the environment and cannot make it wait. The system reads the inputs and performs the computation and makes the result available before the next set of inputs is available. The examples of such systems are lift controller and vehicles automation systems. The reactive systems can be divided into two categories: data-dominated and control-dominated. The reaction time is less important for the data dominated systems, which contains intense data computations, operating on samples arriving in regular intervals from the environment. On the other hand, the control-dominated systems do not perform intensive data computation. The systems which have mix of control and data are termed as heterogeneous systems. Criticality, parallelism and determinism are some of the most essential features of a reactive system [5]. The reactive systems are usually modeled using synchronous languages which express reactive control flow patterns in a concise manner, with a clear semantics that imposes deterministic program behavior under all circumstances.

The examples of typical target areas for such systems are: automotive (e.g. engine controllers, anti-lock brake controllers); consumer electronics (e.g. microwave ovens, digital cameras, compact disc players); plant control (e.g. robots, plant monitors, airplane control systems, missile guidance systems) and telecommunications (e.g. telephone switches, cellular phones).

1.1.3 Real-time Processing

Real-time systems are those systems in which the correctness of the system depends not only on the logical results of computation but also the time at which the results are produced [6]. A real-time embedded system that must meet at least one hard deadline is called a hard real-time embedded system or a safety-critical real-time embedded system. Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment. Missing a hard deadline means a total system failure. If no hard deadline exists, then the system is called a soft real-time embedded system. The design of a hard real-time system is fundamentally different from the design of a soft real-time system. While a hard real-time computer system must sustain a guaranteed temporal behavior under all specified load and fault conditions, it is permissible for a soft real-time embedded system to miss a deadline occasionally. Timing correctness requirements in a real-time system arise because of the physical impact of the controlling system's activities upon its environment. The most common timing constraints for tasks are periodic, aperiodic, and sporadic. A task is an execution path through address space. A periodic task is one that is activated every T units. The deadline for each activated instance may be less than, equal to, or greater than the period T. An aperiodic task is activated at unpredictable times. A sporadic task is an aperiodic task with the additional constraint that there is a minimum interarrival time between task activations. Real-time systems need a time-predictable execution platform so that the Worst Case Execution Time (WCET) can be estimated statically. Predictable system is one in which timing requirements can be guaranteed a priori. WCET of a computational task is the maximum length of time the task could take to execute on a specific hardware platform. Knowing worst-case execution times is of prime importance for the schedulability analysis of hard real-time systems. The distributed nature of the system makes it easier to design but wiritng applications for such systems becomes harder.

1.1.4 Distributed Nature

New application areas of embedded systems such as networked computing are composed of several computing units thus making them distributed systems in nature. There are various reasons to distribute embedded systems for example: the high performance enabled by the use of several computation units for a better response time, sometimes the geographical delocalization of the system elements or the replication of systems for fault tolerance. A distributed system consists of a collection of autonomous components, connected through a network and distribution middleware, which enables them to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility. The software part may run as concurrent processes on different processors which may have multiple points of failure and control. The systems can also be decomposed into smaller specific units carrying out a particular task efficiently and run concurrently offering different services at the same time. They may run at the same or different clock speed in which case they are termed as multi-clock systems. Applications modeling for such distributed systems is hard but they are easy to test and maintain.

1.1.5 Globally Asynchronous Locally Synchronous Systems

A wide range of embedded systems today consist of a number of computational intensive units running concurrently. They also tend to interact with each other and with the environment making them reactive by repeatedly reading inputs, doing computations and generating outputs. These computational units have different response times; hence, they may need to run concurrently at different speeds avoiding the higher operating frequencies which result in reduced power consumption. These systems are called GALS (Globally Asynchronous Locally Synchronous) systems [7] and typically consist of a collection of sequential, deterministic components that execute concurrently and communicate using slow or unreliable channels. Hence, a GALS system consists of two or more processes (or processors) running concurrently and asynchronously, i.e., they run at unrelated clock speeds, at the top-level. Each of these asynchronous processes can themselves consist of one or more processes all running concurrently but synchronously, i.e., in lockstep with the parent process clock speed. GALS paradigm is used both in software and hardware. In software GALS systems, different processes run concurrently whereas in hardware GALS systems different physical processing modules run concurrently. The GALS paradigm unites the advantage of synchronous designs (for determinism) and asynchronous designs (for flexibility) to obtain more efficient and powerful system descriptions and implementations. Examples of GALS applications are distributed control applications, industrial process controllers, aircraft and automobile controllers, ATM networks and multi-agent robotics etc.

Let us consider the example of a track controller controlling the tranportation of multiple trains onto the track divided into multiple blocks as shown in Figure 1.2. The trains can enter and exit these blocks. The trains cannot enter any block without permission and cannot exit any block without indication. The safety requirements demand that no two trains can be present in the same block at any time. The trains can move from one block to another. The problem is to design appropriate controllers for each block. The solution of such a problem is a GALS system having distributed network of controllers with one controller per block. Each controller is locally reactive and can sense entry/exit of trains and exchanges signals with the train on the block. The adjacent controllers can talk to each other for controlling entry/exit of trains.

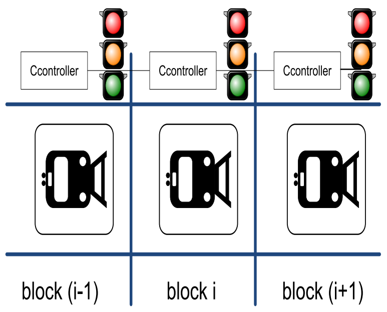


Figure 1.2: Exampe of GALS application: track controlling

1.2 Modelling GALS System

Modelling GALS applications typically requires the use of non-standard control flow constructs, concurrency and exception handling. The technological advancements certainly opens up vast possibilities for innovation, but only provided that the attendant complexity can be managed. It should be kept in mind that if product complexity is measured on a linear scale, Moore's law. In order to avoid exponentially increasing development costs, it is suggested that overall productivity of designed functions must itself improve exponentially on the time axis. Given the importance of embedded systems, it is important to overcome bottlenecks represented by current development practices in the face of increasing complexity [3].

1.2.1 Modelling with Traditional Languages

Among the programming languages, C, C++ and Java are the most commonly used in embedded systems and are familiar to designers. Java's expressive power is comparable to C++; it is a much simpler language, which reduces the difficulty of program analysis and optimization. It has the edge over C due to standard language and library support for concurrency. Its treatment of arrays permits better static and dynamic error checking than is conveniently feasible in C and C++. Using Java's widespread adoption by the science and engineering community promises a large base of support in the form of compilers, debuggers, development environments, and class libraries. Java's higher level of abstraction allows for increased programmer productivity. It is relatively secure and has support for dynamic loading of new classes, component integration and reuse, which provides platforms to support application portability as well as distributed applications. But, these traditional languages lack high level parallelism and use of asynchronous parallelism typically leads to non-deterministic program behavior with respect to time and functionality [8]. It is hard to certify the correctness of the programs modelled in such languages which is a must for safety critical systems. The lack of statements for modelling reactive control structures and frequent occurrence of context switching to support concurrency reduces their efficiency.

1.2.2 Modelling with System Level Design Languages

Modelling GALS applications using traditional languages is complex, time consuming and expensive thus leading to large designer productivity gap. Thus, we need to move to higher level of abstraction in order to manage the ever increasing complexity and heterogeneity in the embedded systems. The new design challenges can be tackled by moving to the higher level of abstraction, called system level. The designer specifies a model at the highest available level of abstraction which can be translated to a lower level subsequently. The lower levels such as behavior, register-transfer, circuit and physical (in descending order) differ from each other by their execution semantics often referred to as a model of computation (MoC) and the modeling details [9]. For example, the MoC for the logic level may be thought as including Boolean equations to describe the data-path and Finite State Machines (FSMs) to describe the control. The MoC of the RTL may be a combination of FSMs with data-path [10] or as a micro-program to describe the register transfer level operations. The semantics of the behavioral level has often been that of discrete event systems [9]. The system level distinguish itself from the lower levels of abstractions by providing separation of computation from communication, mix of a range of semantics or models of computation, behavioral hierarchy, support for exceptions and exception handling, mix of data-dominated and control-dominated processing and support for formal verification. The system level approach provides an abstract way to capture many important features such as modularity, hierarchy and concurrency, which can be, both behavioral and structural, external communication between the modeled system and its environment and, finally, communication between concurrent processes within the designed system. The use of system level design languages is vital as it enables us to specify the model at an abstract, system level, and reduce productivity gap and time-to-market pressure.

SystemC [11], SystemVerilog [12], SpecC [13] are examples of system level design languages which provide an abstract way to model embedded systems. These languages use a well-known syntax with powerful constructs, enabling the modelling and simulation of complex systems. But, they are informal languages as they are not based on formal mathematical model of computation and do not follow formal semantic rules. It is hard to prove the correctness of designs modelled using these languages which is an essential property of the embedded systems thus making them less suited to model such systems. Although a subset of these languages can be formally described but they are unable to describe complex heterogeneous systems [8].

1.2.3 Modelling with Synchronous Languages

The synchronous languages [14] such as Esterel [15], lustre [16], signal [17], statecharts [18], sml [19] synccharts [20], argos [21] and SR [5] are system level design languages with synchronous semantics. Esterel is imperative, while *lustre* and *signal* are declarative in style; *state-*

charts and *argos* are graphical languages that allow one to program by constructing hierarchical automata. They support control flow constructs and express it in a user friendly manner. They have simple formal model and are based on synchrony hypothesis, which states in essence that a system responds in zero time to environmental requests [15]. The key advantage of synchronous languages is that the synchronous approach has a rigorous mathematical semantics which allows the programmers to develop critical software faster and easier. But, synchronous languages show poorly when performing data computation and communicating asynchronously.

1.2.4 Modelling with Asynchronous Languages

The asynchronous languages (also called multi-clock languages) such as CSP [22], OCCAM [23] and ADA [24] are capable of handling concurrency as well as asynchronous communication but they suffer from same drawback of poor data handling. Communicating Sequential Processes (CSP) is a formal language for describing patterns of interaction in concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi. OCCAM [23] is built on CSP [22] and shares many of its features. ADA is an extension of PASCAL and other languages. It is capable of handling data computation unlike other asynchronous language but is an informal language.

Due to inability of system level design languages to model multi-clock heterogeneous systems, the productivity gap remains there which requires a way to model such complex systems with ease and quickly. An ideal programming language should have the capability of efficiently handling intensive data computation, concurrency and asynchronous communication.

1.2.5 Modelling with SystemJ: A System Level Design Language

SystemJ [25, 26] is a system level programming language based on the Globally Asynchronous Locally Synchronous (GALS) model of computation and allows the asynchronous coupling of synchronous reactive modules at the top level, which execute at different speeds. It extends Java with Esterel-like [15] constructs for the synchronous concurrency and reactivity, and CSP-like [22] constructs for the asynchronous concurrency. The Java has been used in the past for writing reactive programs such as PureSR [27], Java-Time [28], Jester [29] and Junior [30] which are the reactive extensions of Java. SystemJ goes one step further and we view SystemJ as an ideal environment for specifying, modelling, implementing and formally verifying GALS based embedded systems.

SystemJ targets a large range of heterogeneous embedded systems that combine data-intensive and control-dominated computations in addition to synchronous and asynchronous concurrency. SystemJ is based on the synchrony hypothesis which makes the assumption that the computer is infinitely fast, each reaction is instantaneous and atomic, dividing time into a sequence of discrete instants, different reactions cannot interfere with one another, and a system's reaction to an input appears at the same instant as the input. The synchrony hypothesis is a generalization of the synchronous model used for digital circuits where each reaction must be finished during one clock *tick*. The perfect synchrony paradigm of synchronous languages considers the computation of reactions to be infinitely fast, so that the reaction appears at the same point of time when the action requests for it. This is achieved by an idealized view where the execution of most statements does not require time. Consumption of time must be explicitly programmed: every atomic statement consumes either none or exactly one unit of a logical time. Hence, by the semantics of these languages, all threads run synchronously to each other, since they automatically synchronize at the next time consuming statement.

Our work focuses on the SystemJ language, which combines the data-computation with control and asynchronous communication to model the complex and heterogeneous multi-clock embedded systems.

1.3 Motivation

SystemJ is a system level design language which models applications targetting Globally Asynchronous Locally Synchronous (GALS) heterogeneous systems. SystemJ models the control computations using a combination of both the synchronous and the asynchronous model of computations. SystemJ consists of asynchronous processes running concurrently at individual speeds at the top level, similar to CSP, while each of these asynchronous processes may consist of one or more synchronous processes running in lock step, similar to Esterel. SystemJ uses Java to describe data computations because of its powerful constructs for traditional sequential programming. A system described using SystemJ language communicates with the environments through the signals. GALS MoC implements both synchronous and asynchronous concurrency and also has the built in feature of signal broadcasting and rendezvous.

The languages based on GALS MoC can be implemented as software programs to run on traditional general purpose processors to achieve the desired behavior. This kind of implementation may provide better data handling but, these traditional processors and classical programming languages do not have similar structures or statements (instructions) to handle the corresponding synchronous and asynchronous features. Hence, implementation on commercial off-the-shelf (COTS) processors is problematic since it must be simulated. Therefore, a general purpose processor based software solution can hardly enhance the application performance or reduce resource usage. Furthermore, the compilation techniques targeting such processor or languages suffers from the drawbacks such as large generated code size [31] and reduced runtime performance.

Since traditional processors have difficulties to handle synchronous and asynchronous features of GALS MoC, it gives rise to a natural question whether reactive processors can be deployed as alternative. Reacitve processors [32–36] and the Esterel Virtual Machine (EVM) [31] are a few examples of execution platforms which provide support for reactivity and concurrency. While all these approaches provide efficient execution of reactive control parts and modest support for concurrency, they provide very poor support or have practically no support for complex data-driven operations. There have been efforts towards supporting heterogeneous applications [37], but they primarily focus on reactive part of applications. Therefore, the architectural support for heterogeneous embedded applications based on GALS MoC is lacking and demands a serious attention.

Furthermore, embedded systems are different from traditional desktop computers in the sense that they are constrained by the size, weight, power, cost and response time. Most embedded systems execute in response to external events, both periodic and aperiodic. The correctness of operation often depends on the response time staying within a given time limit. Also, the suitability of the target platform depends upon the nature of the applications. For example, data-dominated embedded systems such as video games often require high computation performance and large program memories. The mid-end embedded systems may be for example engine control systems typically demand a medium level of computing performance and memory. Low-end embedded systems are typically low cost and high volume consumer devices. They have low computing requirements and a program memory of some kilobytes. The growth in complexity may have a significant impact on implementation constraints such as cost, size, performance as well as power. Cost is, with few exceptions, an important issue in embedded systems. Many embedded systems are produced in large quantities; the need to reduce costs is a major concern. Embedded systems often have significant energy constraints, and many are battery-powered. As a result of these constraints, embedded systems use a slow processor and small memory size to minimize costs and energy consumption thus making it hard to deploy in embedded system with stringent response time requirement. Therefore, embedded-computing applications involve unique challenges, as they must be performed in real time in the face of resource constraints.

In summary, we need a target platform which could efficiently handle GALS MoC and at the same time should be very competitive with other implementations in terms of performance and resource usage. Thus motivation of this thesis is to explore and develop an efficient execution platform for SystemJ based applications. The architecture should be capable of efficient execution of control and data-computations as the embedded applications are heterogeneous in nature and consist of a mix of control and data parts. The solution should minimize logic resources so that it is economical enough cost wise. Furthermore, the capability of being time-predictable is essential in real-time systems. With a lot of functionality being added, the need for high performance in embedded systems has become inevitable. The architecture must be scalable towards multiprocessors so that it is able to support increasingly complex applications, since it is not possible to rely on single core frequency improvement.

1.4 Research Contributions

The focus of this thesis is the development of an efficient architecture for the execution of GALS programs. The project was driven by the desire to achieve predictable, competitive execution speeds at minimal resource usage, in terms of processor size. This work has resulted in proposing several novel processor architectures that improve the performance and efficiency while consuming less resources and guaranteeing the time predictability offered by the base processor architecture.

In summary, the contributions of this research are:

- *Accelerated and Time Predictable Execution of GALS Programs:* We demonstrate the effectiveness of a new predictable execution of GALS program on GALS programs described in SystemJ on a Java processor, called Java Optimized Processor (JOP). Previously, the SystemJ programs were translated to Java which are further compiled for execution on the targeted general purpose processor. This approach was slow due to interpreting nature of Java Virtual Machine and require large memories. However, the use of JOP eliminates the overheads of the interpreter as it is done natively by the processor itself. This intermediate step in the design process with SystemJ enables us to prototype and to verify the specifications on a predictable architecture.
- *A Reactive Java Processor for the Execution of GALS Programs:* We also propose a novel, high performance and low cost execution architecture for SystemJ. The new core, which is called RJOP (Reactive JOP), facilitates efficient execution of both data dominated and control dominated embedded applications described in SystemJ. It extends JOP to efficiently handle reactivity and signal manipulation. It also maintains the time-predictable execution of the applications intended for real-time embedded systems and calculation of Worst Case Reaction Time (WCRT) as provided by the original core.
- *A Heterogeneous Tandem Processor Architecture for GALS Programs Execution*: We propose a high performance solution in the form of a tandem processor with JOP, or TP-JOP, in which control processor (CP) and JOP work together to implement control flow and data operations, separated during compilation of GALS programs, respectively. TP-JOP is built on TVM/TP approach [38, 39] which are based on the idea of tandem execution of two processors, one that controls the flow of SystemJ program (CP, control processor) and one that executes operations that are within our GALS MoC (which are

considered instantaneous and expressed in standard Java). The data oriented part of the code which is in Java was executed on a general purpose processor with JVM in previous implementation of the TP. We demonstrate the effectiveness of the architecture through experimental evaluation against a range of suitable benchmark.

- *Efficient Merging of Control and Data-computations:* Multiple heterogeneous cores have their own design complexity issues. Special purpose cores have significant impact on the memory hierarchy of the system, and require specially designed communication protocols for fast data exchange among them. A major challenge is the design of a suitable high-performance and flexible communication interface between them. We propose a novel GALS-JOP processor where JOP and the CP functionalities are merged into a single processor by enriching JOP with some key constructs and abstractions for efficient implementation of SystemJ GALS Programs. The concurrency and reactivity control flow presented as special instructions is translated to Java statements which are mapped to custom bytecodes. This core has acceptable performance considering that it is more economical in terms of resource usage compared to the previous implementations.

- *Execution of Control and Data computations with Distinct Modes of Executions:* The focus of the research has been the trade-off between performance and complexity of the microarchitecture. Devoting precious silicon area to special purpose execution hardware or processor does not lead to an optimal solution. We have developed a solution which extracts the desired functionality and execution speed without consuming precious resources. We have presented an approach to efficiently mix Java with asynchronous and synchronous concurrency and execute it on a specialized Java processor extended with capabilities for concurrency and reactivity. A new processor, which uses JOP (Java Optimized Processor) as its base, executes concurrent programs that comply with GALS formal model of computation by clearly distinguishing between concurrency and reactivity control flow and Java control flow. The new processor, called JOP-Plus, can be used for embedded and even real-time applications in which the majority of application code is written in Java and the overall programs specified and structured in SystemJ system-level concurrent programming language. We have implemented a processor core where two virtual processors share one data-path. The virtual processors operate independently although not in parallel.

- *A Homogeneous Multiprocessor Architecture for Concurrent Execution of GALS Programs:* A new, scalable, multiple processor architecture is proposed for execution of SystemJ programs based on GALS model of computation. The proposed architecture GALS-CMP, based on JOP-Plus, is suitable for implementation of embedded systems that contain reactive or control dominated parts that interact with external environment. It allows concurrent execution of clock-domains improving the response time.

- *Compiler Modifications:* We have made several modifications to the SystemJ compiler back-end to produce compatible Java codes for the new target platforms. We also proposed several optimizations to produce more efficient code in future.

- *Experimental evaluation:* We evaluate the different architectures and validate their effectiveness by running benchmarks on them. Better performance, code density and resources usage has been found and compared to previous approaches for SystemJ execution, thus proving its suitability for heterogeneous embedded applications.

1.5 Thesis Organization

The remainder of this thesis is organized as shown in Figure 1.3. Chapter 2 gives an overview of the SystemJ language. The chapter starts with the arguments on the need of such a language and introduces its syntax and structure through an example program. Next, the compilation strategies used for target platforms are discussed in detail. Finally, the execution of code produced by the SystemJ compiler on existing platforms is discussed.

Chapter 3 presents a detailed description of Java Virtual Machine (JVM) specifications and its structure. Then, we discuss different hardware solutions from both academia and industry for accelerating Java in embedded systems. The details of a Java processor, called Java Optimized Processor (JOP), and its architecture are also presented. JOP, which is the basic tool of this research, is introduced as the execution platform for GALS programs and its effectiveness is demonstrated through experimental results. This chapter also describes Reactive-JOP, an enhanced version of JOP which incorporates reactive features. This chapter ends with short description in the form of summary.

Chapters 4 presents two different execution platforms for the efficient execution of SystemJ compiled in a way that separates control from data-computations. The heterogeneous multiprocessor architecture TP-JOP is an improved version of existing TP and uses two specialized processors for native execution of both control and data-computations. The novel uniprocessor approach, called GALS-JOP, executes both control and data-computations by translating the programing model of control into the program model of the data-computations processor. This chapter contains a detailed architecture description of both processors and their implementations. The experimental results are provided to evaluate the performance of the proposed architectures.

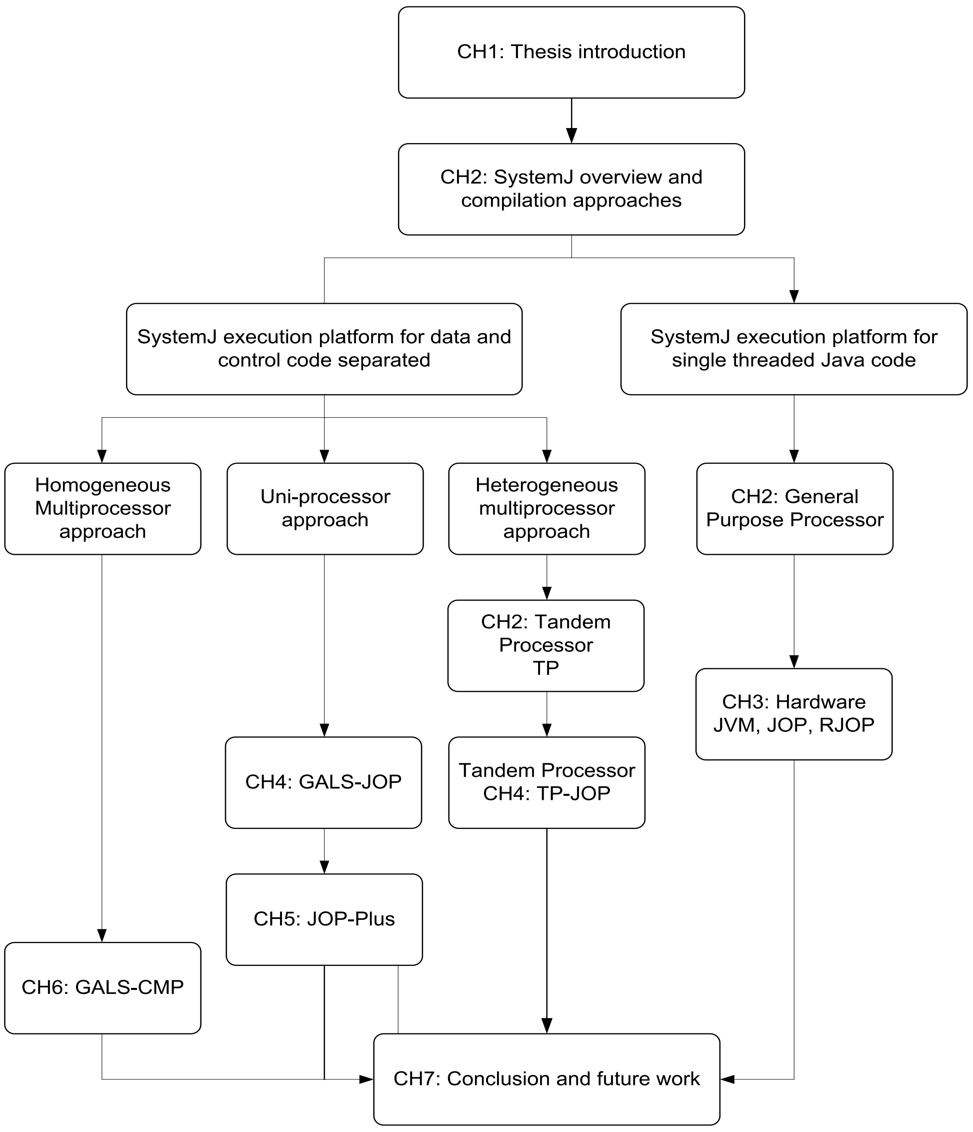

CH1: Thesis introduction
CH2: SystemJ overview and compilation approaches
SystemJ execution platform for data and control code separated
SystemJ execution platform for single threaded Java code
Homogeneous Multiprocessor approach
Uni-processor approach
Heterogeneous multiprocessor approach
CH2: General Purpose Processor
CH2: Tandem Processor TP
CH3: Hardware JVM, JOP, RJOP
CH4: GALS-JOP
Tandem Processor CH4: TP-JOP
CH5: JOP-Plus
CH6: GALS-CMP
CH7: Conclusion and future work

Figure 1.3: Thesis structure

Chapter 5 also uses a single processor for the execution of control and data computations similar to GALS-JOP. The proposed processor, called JOP-Plus, has two distinct execution modes for executing control and data parts. This chapter describes the background for the proposed approach, related work and their shortcomings. Next, it gives a description of the strategy to remove these shortcomings. JOP-Plus design flow, memory organization and implementation details are presented next. Finally, the evaluation of JOP-Plus's performance through experimental set up is discussed.

Chapter 6 presents a homogeneous multiprocessor system in order to meet the processing power requirement and response time constraints imposed by the modern applications. This chapter starts with the discussion on performance limits of uniprocessor approach and dwells on the ways to boost performance or processing power beyond the uniprocessor limits. Then we describe GALS-CMP architecture and its compilation and execution flow with the help of an example. The experimental results and related discussion are presented towards the end of chapter.

Finally, Chapter 7 gives a short discussion on the implementation platforms developed during the course of this research and their performance comparison is carried out. In the end, concluding remarks and possible future directions for the research are presented.

Chapter 2

SystemJ Overview

The main goal of SystemJ is to provide a specification mechanism, a language, that allows a system designer to use well-known programming constructs. SystemJ covers all the important criteria of a system level design language. SystemJ is based on formal semantics, which may allow partial automated formal verification of a SystemJ program. The SystemJ is expressive enough to incorporate both data and control flow information and includes a well known programming language Java that is easy to use. This chapter gives an overview of the SystemJ language. It introduces the SystemJ Model of Computation (MoC), syntax, synchronous and asynchronous kernel statements. It also presents a SystemJ program example, different approaches to compile it and existing execution platforms.

2.1 The SystemJ MoC

The system-level programming language SystemJ [25, 26] extends Java with synchronous and asynchronous concurrency and reactivity, making it suitable for designing complex embedded programs. The language allows use of full Java and discourages the use of Java concurrency (threading library). Instead, SystemJ provides its own concurrency model based on the formal Globally Asynchronous Locally Synchronous (GALS) model of computation (MoC).

A SystemJ program consists of multiple asynchronous processes, called *clock-domains* (CD), which are described at the top design level. The clock-domains are composed together with the asynchronous parallel operator (><). Each clock-domain consists of a number of synchronous concurrent processes, called *reactions*, which execute in lock-step, driven by a logical clock, called *tick*. Transitions in Esterel are also based on the same logical and discrete lock event. A synchronous program reacts to its environment in a sequence of ticks, and computations within a tick are assumed to be instantaneous, i.e., as if the processor executing them was infinitely fast. The reactions communicate within a clock-domain, as well as with the external environment (*input/output*) through *signals*, which are broadcast and present within the current tick and comply with synchronous reactive MoC [15, 40–42]. The reactions are represented as concurrent processes within a clock-domain using the synchronous parallel operator (||). Communication between reactions in different clock-domains, which are asynchronous each to the other, is carried out through the exchange of messages over *channels*, which are semantically the same as channels used in CSP MoC [8, 22]. Besides operations on signals and channels, SystemJ allows free use of Java data objects and statements in its reactions, and those statements are considered instantaneous in terms of logical time (i.e. they do not consume logical time or ticks). Control flow of a SystemJ program incorporates scheduling of all reactions and clock-domains, as well as communication between reactions, and communication with the external environment. The data-driven computations and transformations are performed in Java. The SystemJ is amenable to verification as it is based on mathematical semantics and can be deployed in mission critical systems.

2.2 The SystemJ Entities

The SystemJ has three types of high-level design entities called system, clock-domain and reaction.

2.2.1 System

The *system* is the top level entity through which a SystemJ program sees its environment and vice verse. In the declarative part of the system entity, the interface with the environment (which are input/output signals) and channels are declared. The clock-domains are declared inside the body of the system. The channels serve as the communication medium between the clock-domains. Multiple clock-domains, which are asynchronous processes, are also declared in the body of this entity. Within a system, each clock-domain executes at its own tick, and any two clock-domain ticks are unrelated.

2.2.2 Clock-domains

A SystemJ program consists of a set of clock-domains executing at unrelated logical clock ticks. Each clock-domain consists of a number of synchronous concurrent processes called reactions. The clock-domain can be given some name or can be unnamed. The clock-domains can communicate with each other and with the environment using their own set of channels and signals, respectively. Two clock-domains synchronizing with each other using CSP style communication are called partner clock-domains. The data structures cannot be shared among the reactions and among the clock-domains using mutual exclusion as it will provide another source of communication which is not safe. This prohibition allows communication only through the signals and channels, which is safer and easier to validate.

2.2.3 Reactions

Reactions are combined and controlled within a clock-domain using the synchronous parallel operator (||). Reactions can be either named or unnamed and may have child reactions. Reactions use sequential and concurrent statements to specify their operation and those statements are from both SystemJ and Java repertoire. Reactions communicate within a clock-domain, as well as with the external environment (*input/output*) through signals, which are broadcast and present within the current tick if *emitted*, otherwise they are *absent*. Communication between reactions in different clock-domains is carried out through the exchange of messages over channels, which are semantically the same as channels used in the CSP MoC [8,22]. When any of the time consuming statements is executed by a reaction, it consumes one tick of time. Hence, by the semantics of synchronous languages, all reactions with in a clock-domain run synchronously to each other, since they automatically synchronize at the next time consuming statement. The reaction stops its own execution after executing time consuming statement and waits until other synchronous reactions in the same clock-domain consume their own tick. Behaviors of the reactions fully comply with the synchronous reactive (SR) MoC [5,9,15,41,42].

2.3 SystemJ Objects

The SystemJ has three main types of objects namely *signals*, *channels* and *Java objects*. The channels and signals are used for inter clock-domain and intra clock-domain communication, respectively.

2.3.1 Signals

The signals are used for communication among reactions within a clock-domain as well as with the external environment (*input/output*). The signals communicating with the environment, whether input or output, are called *interface* signals and are declared in the interface part of the body of the system. The signals which are used to communicate among the reactions in the clock-domain are termed as *local* signals and are declared within the body of a clock-domain. The signals are always broadcast and *present* within the current tick if *emitted*, otherwise they are *absent*. A signal has a default status of *unknown* only during the first pass of program execution. After the first pass, its default status is absent and not the unknown. The signal can be either a *pure* signal or a *valued* signal. A pure signal has a status only, while a valued signal has both status as well as value of any Java data-type. A signal can be emitted by the multiple reactions of the same clock-domain.

We use signals for communication with enevironment and not the channels because clock-domains in SystemJ are complete deterministic but their asynchrnous composition leads to non-deterministic behavior. Therefore, if we use channel for communication with the environment, it will introduce non-determinsim.

2.3.2 Channels

As mentioned earlier, communication between reactions of different clock-domains is carried out through the exchange of messages over channels. They are used to synchronize the clock-domains. The channels can be of any Java type and are declared in the interface body of the system. The clock-domains are synchronized through channels using CSP style rendezvous mechanism. The idea is that two clock-domains rendezvous at a point of execution, and neither is allowed to proceed from that point until both have arrived, but they can proceed to execute other reactions of the same clock-domain if needed. The channels are point-to-point i.e., for every sending channel port there needs to be a corresponding receiving channel. The channels are unidirectional, therefore, they are always declared in pairs (input and output). No two input channel can read the same output channel or vice versa. Every channel has status and value buffers which are used to implement the rendezvous communication. The input and output channels are composed of signals *channel_receive* and *channel_sent* respectively and both are of type *integer*. If the sending channel's clock-domain is faster than the receiving channel's clock-domain, it sends the data through the channel and increments the *channel_sent* signal. The receiving channel samples the *channel_sent* signal at the start of tick, if it is greater than the *channel_receive*, it receives data and increments the *channel_receive* signal which is an acknowledgment for the sending channel. The sending channel samples the *channel_receive* at the start of tick. If both are equal, then the sending channel sends the data again, otherwise sending channel blocks producing internal ticks. If the receiving channel's clock-domain is faster than the sending channel's clock-domain, the receiving channel will block for *channel_sent* signal. Only the reaction receiving data from the channel is blocked producing internal ticks whereas other sibling reactions run normally. The implementation of rendezvous communication in SystemJ is based on blocking reads and writes i.e., they wait for each other to synchronize and get ready to exchange data.

2.3.3 Java Objects

Besides signals and channels, SystemJ allows free use of Java data objects in the program. The SystemJ objects declared are global but they cannot be used for the communication among the clock-domains as it is allowed through channels only.

The pictorial explanation of a simple SystemJ example program that consists of three clock-domains CD1, CD2 and CD3 is presented in Figure 2.1. The green shaded rectangles indicate the clock-domains while blue shaded areas show the reactions inside the clock-domain. The arrows indicate the interface signals, local signals and channels as labeled in Figure 2.1.

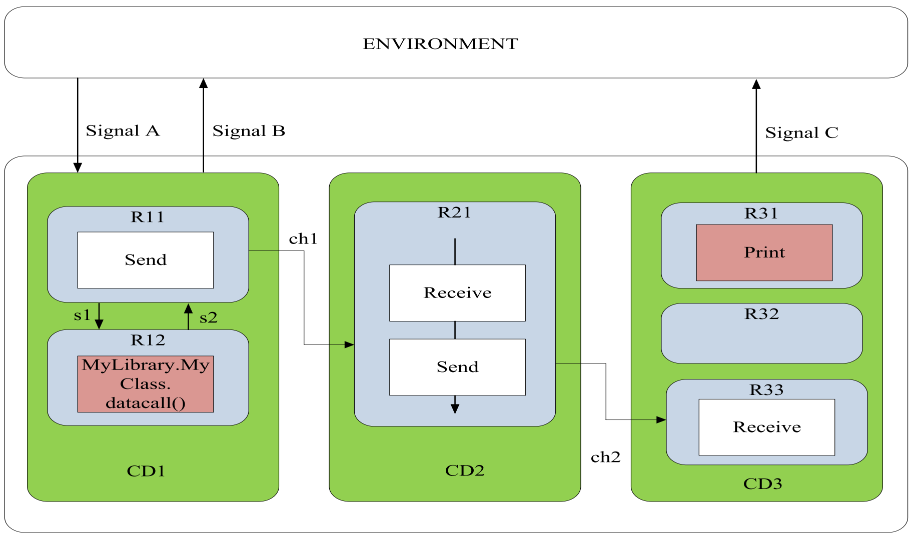


Figure 2.1: A graphical equivalent of SystemJ example program

2.4 The SystemJ Kernel Statements

The SystemJ kernel statements comprise of both the synchronous and asynchronous statements which deal with the synchronous and asynchronous parts of GALS MoC, respectively. The complex statements can be derived by combining the kernel statements.

2.4.1 Synchronous Kernel Statements Descriptions

The description of the synchronous kernel statements is given below.

Dummy statements: There are two dummy statements, ';' and *pause*. The ';' does not perform any operation. The pause statement also does not perform any operation but differs from the ';' statement as pause consumes one tick and the following instruction is executed only in the next tick. It is the only explicit statement which consumes a tick. If a reaction does not have a pause statement; it consumes a tick by default after it has executed all the instantaneous statements. The reaction will be executed again in the next tick.

Signal declarations: The statement for the declaration of signals in a SystemJ program is given below:

$$[input]\ [output]\ [type]\ signal\ SignalName$$

The signal statement may have a prefix of input or output and a type associated with it. If the signal statement has input or output as prefix, then the signal *SignalName* is declared as the interface signal to communicate with the environment. In the absence of these prefixes, the signal is declared as a local signal and is used for communications between reactions in the same clock-domain. If the signal has a prefix type, which can be any Java type, then it is a valued signal otherwise it is a pure signal which has status only. When a signal is declared, its status is set to unknown ($\perp$) and value is initialized to zero.

Signal emission: The signal emission statement is used to emit the status of signal as well as the value, if it is a valued signal the value can be of any Java type. The syntax of the kernel statement for signal emission is given below:

$$emit\ SignalName$$

The status of the signal can either be present (+), absent () or unknown($\perp$).The default status of a signal is absent and the value (for valued signal) can be determined in any instant, even if the signal is absent. The signals remain absent until they are explicitly set to present. Whenever signal emitting instruction is executed, it sets the status and value of the signal. A signal can be emitted multiple times during the same tick whose value can be combined by using the *hook* provided by SystemJ. This hook can call the function attached to it which is provided by the designer to set the value of the emitted signal at every emission.

Sequential statements: The SystemJ allows the combining of synchronous reactive statements by using the *';'* operator, making them execute sequentially one after the other. Here p1 and p2 represent arbitrary SystemJ statements.

$$p1\ ;\ p2$$

Infinite loop: The SystemJ programs run forever and this feature is implemented by using *while* statement which provides an infinite loop mechanism. All the finite or infinite loops enclosing the synchronous reactive statements need to consume at least one logical tick to avoid schizophrenic signal behavior. It occurs when compound statements can be run and exited or terminated and reentered inside the same reaction within the same tick [15,41,42]. This may result in signals involved ending up having two distinct occurrences in the same reaction, with differing values.

$$while\ (true)\ p$$

Conditional statements: The *present* statement checks the status of a signal expression at any given instant of time. The signal can only be present if it is emitted or it is coming from the environment. The syntax of the present statement is given below:

$$present\ SignalName\ p1\ else\ p2$$

Preempt watchdogs: The *abort* statement acts as watchdog and provides an exception mechanism based on signal expression. If the watchdog signal expression is evaluated to true in any given tick, this statement preempts the body of the abort construct. The abort syntax is

$$[weak]abort[immediate]SignalName\{p\}$$

The abort can be performed on *immediate* or *non-immediate* signals. In the former case, the expression is evaluated in the very first tick when the statement is executed; otherwise the checking of signal expression is delayed until the next tick. The abort can also be weak or strong. In case of weak abort, the preemption of the enclosed computation occurs at the end of logical tick; otherwise the enclosed computation is preempted even before entering the node.

Halt watchdog: The *suspend* statement acts as a halt watchdog and is very similar to abort except that this statement causes its body to pause instead of preempting the body if the signal is present. The suspend statement can also be weak and strong and accompanied by the immediate signal predicate as shown by its syntax.

$$[weak]suspend([immediate]SignalName\{p\}$$

Trap exception mechanism: The *trap* and *exit* statements are a user controlled preemption unlike the *abort* and *suspend* preemptions which are based on the signal. The trap statement is similar to the Java's try-catch but has different semantics as Java's *try-catch* semantics is incompatible with SystemJ semantics. The computations are enclosed inside the *trap* statement, which can be preempted by the *exit* statement. If no exit statement is executed then the *trap* statement finishes processing the enclosing block and continues forward. The *trap* statements can be nested where highest priority is given to outermost *trap* construct.

$$trap\ (T)\ \{p\}$$

The exit from the trap is made by using the exit statement given below:

$$trap\ (T)\{P\ exit\ (T)\}$$

Parallel statements: The synchronous concurrency causes reactions to run in parallel and in lockstep. This is done by using the synchronous concurrency primitive construct (||). Both p1 and p2 are not necessarily instantaneous. p1 and p2 may have Java statements for data computation and several "pause" statements allowing them to synchronize on tick boundaries. The synchronous statements also consist of statements for Java data-driven computations.

$$p1 \, || \, p2$$

Each reaction in the clock-domain finishes with a termination code having an integer value ranging from 0 to infinity (∞). A reaction terminates with the code of 0 if it is completed and will never run again. A termination code of 1 indicates that the reaction has paused and will resume execution in the next tick. All reactions which have not finished execution due to unresolved signal dependency are assigned termination code of ∞ (represented by special code depending on the platform) and are used for the cyclic scheduling of the reactions. The rest of the termination codes are reserved for the traps. Higher the termination code, higher is the priority of the trap. The collective termination code of the reactions combined using the synchronous parallel operator is the termination code of the reaction having maximum value. For example, if reaction *p1* completes with a termination code of '*m*' and *p2* with '*n*' then the composition *p1*|| *p2* completes with a termination code of *max(m, n)*. Each clock-domain is both deterministic and causal by construction.

Java data-driven computation: As mentioned earlier, the program may have Java data-driven computation in the following form:

$$jterm(p)$$

The write-write concurrency of Java data-driven computation is prohibited while read-write concurrency is allowed.

Obtaining signal value: The signal values can be of any Java type.

$$jterm(p) = \#SignalName$$

Obtaining a channel value: The channel can be of any Java type object.

$$jterm(p) = \#ChannelName$$

2.4.2 Asynchronous Kernel Statements Descriptions

The asynchronous statements given below are specific to SystemJ and represent the asynchronous Model of Computation implemented in SystemJ.

Sending channel declaration: The syntax of the statement used for declaring the output channels in SystemJ is implemented using Java variables. The output channels are declared only inside the interface of a SystemJ program.

$$output\ [type]\ channel\ ChannelName$$

The output channels can be considered as a composition of value only. The output channel is used to synchronize and send data of any Java type to input channels of other clock-domain. The SystemJ requires a pair of input and output channels to have the same name. The output channel can only send the data as it is a simplex channel.

Receiving channel declaration: The syntax of the statement used for declaring the input channels in SystemJ is given below. The input channels can be considered as a composition of valued-only signals implemented using Java variables. The input channels are declared only inside the interface of a SystemJ program.

$$input\ [type]\ channel\ ChannelName$$

The input channel is used to receive data of any Java type from the output channel of same name but from other clock-domain. The input channel can only receive data as it is a simplex channel.

Asynchronous parallel: The clock-domains are spawned out by using the asynchronous parallel operator, ><, and run at their individual clocks. Each clock-domain samples the input signals from the environment at the start of the tick. The synchronous parallel reactions inside a clock-domain run simultaneously. They compute the output and emit it instantly in zero time and then wait for the next tick. In other words, each clock-domain behaves like a synchronous Finite State Machine model. When a clock-domain has finished transition, it is called *End of Tick*. The SystemJ clock-domains can be executed using different scheduling disciplines and cyclical is default, one after the other, according to an integral speed ratio.

$$p1 >< p2$$

sending data on channel: The *send* statement is used to send data over the output channel between the partner clock-domains by using rendezvous mechanism. The syntax for the *send* statement is shown below, where data is being sent over an arbitrary output channel *C*.

$$send\ C([exp])$$

The *send* statement is blocking and waits for output channel to get ready for the exchange of the data. Once the send statement is executed, the reaction that contains the send statement, just blocks. It waits for an acknowledgment from the partner clock-domain and consumes internal ticks meanwhile. The other reactions of the clock-domain may proceed in a normal way. When acknowledgment is received from the receiver channel indicating that it is ready, the blocking is released and the next sequential statement after the send statement is executed. A send statement can only be used once on a channel in any given clock-domain as a channel cannot be used by two concurrent reactions to send data at the same time.

Receiving data on channel: The *receive* statement receives the data from the input channel sent by the output channel. The receive statement, like send statement, waits for the other input channel to synchronize and get ready to exchange data.

$$receive\ C()$$

If the receiving clock-domain is faster than the sending clock-domain, then receiver is blocked and produces internal ticks unless it receives signal from the corresponding sending channel that data had been sent. After receiving this acknowledge signal it will unblock and execute next sequential statement. A *receive* statement can only be used once on a channel in any given clock-domain.

2.5 SystemJ Example Program

The application code for SystemJ example described in Figure 2.1 is given in Figure 2.2. The *system* entity is declared on line 3 followed by the environment signals *A*, *B*, *C* and channels *ch1* and *ch2* (line $5-9$). Next, clock-domains CD1, CD2 and CD3 are initialized (line 12). The clock-domain CD1 (line $15-17$) contains two unnamed synchronous parallel reactions identified as R11 (line 20) and R12 (line 27) in the comments. The reaction declaration contains the input signal (*A*), output signal (*B*) and channel (*ch1*) passed to the clock-domain by the system interface. Next, the signals local to the clock-domain (*s1* and *s2*) are declared (line 18). The reaction R11 sends value 2 to its partner clock-domain via its output channel (line 22). Once the reaction has completed the data transfer to the CD2, it emits the local signal s1 (line 23) and then waits for the signal *s2* (line 24). The reaction *R12* waits for the signal s1 to be emitted (line 23) and upon receiving this signal, R12 does some data computation (line 30) and emits signal s2. Meanwhile, reaction *R12* waits for the input signal *A* from the environment. If *A* is present (line 34), it emits B with the value of 2; otherwise, it emits B with the value of 5 and is terminated (line $35-36$). The reaction *R11* is terminated as soon as *s2* is emitted in *R12*. The details of the clock-domain CD2 and CD3 are abstracted out.

```
//example.sysj
import MyLibrary.*;
   system {
      interface{ //environment interface
            input signal  A;
            output int signal B, C;
            //channel points
            input int channel ch1,ch2;
            output int channel ch1,ch2;
      }
      //initializing clock-domains
      CD1(A,B,ch1)><CD2(ch1,ch2)><CD3(C, ch2)
   }   // end of system entity
 // clock-domain 1
 reaction CD1(input Signal  A,
     output int signal  B,
     output int channel ch1){
     Signal  s1,s2;  //local Signals

      //Reaction R11
      {
        send ch1 (2);
        emit s1; //emit signal  s1
        await (immediate s2);
      }
      ||    //synchronous parallel operator

     //Reaction R12
      {
        await (s1);
        //some data computations
        MyLibrary.MyClass.dataCall();
        emit s2;
        present(A)
        emit B(2);
        else emit B(5);
       } // end of reaction R12
  }    // end of CD1
```

Figure 2.2: SystemJ example program

2.6 The SystemJ Compilation Approaches

The SystemJ programs can be compiled using various approaches given in Figure 2.3.

We will discuss these approaches in detail in the next sections.

2.6.1 Library Based Compilation

SystemJ's compilation target is Java source code as Java is used for data-driven computations and transformations in SystemJ and, hence, Java is the most natural compilation target. Furthermore, it is highly portable and Java compilation allows the use of Java compiler for further optimizations and checking. It provides SystemJ with the ability to use standard Java tools and execute exclusively in a Java run-time environment (unlike Jester [29]). Therefore, the necessary functionality had to be implemented as Java libraries. A Java package, named TReK (True Reactive Kernel) implements a kernel of reactive constructs with identical semantics as in Esterel (unlike PureSR [27]). The compiling of SystemJ program is merely the translation to Java

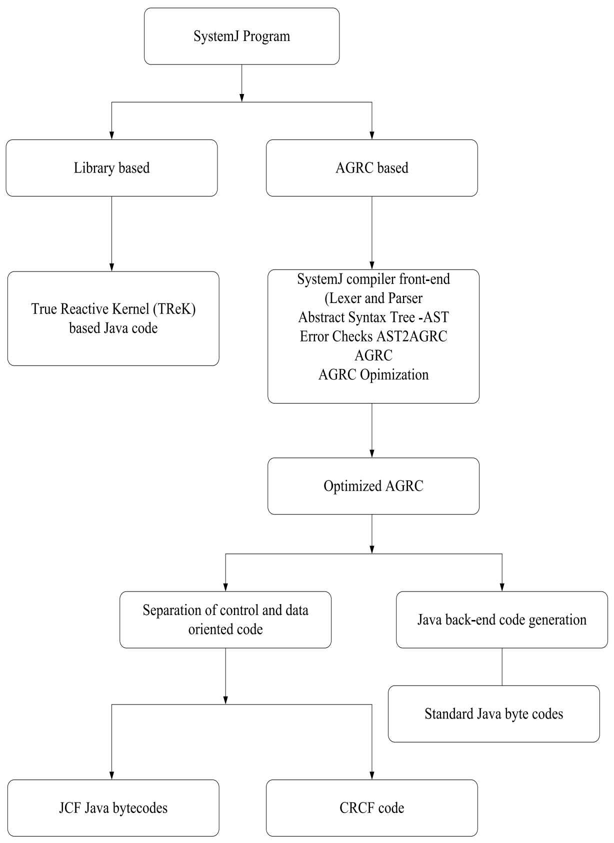

SystemJ Program
Library based
AGRC based
True Reactive Kernel (TReK) based Java code
SystemJ compiler front-end (Lexer and Parser Abstract Syntax Tree -AST Error Checks AST2AGRC AGRC AGRC Opimization
Optimized AGRC
Separation of control and data oriented code
Java back-end code generation
Standard Java byte codes
JCF Java bytecodes
CRCF code

Figure 2.3: Compilation strategies for SystemJ programs

and TReK calls [25]. TReK comprises only sixteen classes and it employs Java thread scheduling, *try-catch* mechanism and generics. In this approach, support for some reactive constructs was missing and dependence on Java multithreading introduced undesirable non-determinism.

2.6.2 AGRC Approach

In this approach [26], the "front-end" of SystemJ compiler transforms a SystemJ program to an intermediate representation called Asynchronous Graph Code (AGRC) which extends the GRaph Code (GRC) format for compiling Esterel [43] with asynchrony and, thus, has similarities with the GRC intermediate format. During the translation into AGRC, structural translation rules [26] are followed and each statement is translated into one or multiple nodes of AGRC. The resulting sub-graphs are then composed together to build the complete AGRC representation of SystemJ source code. The detailed description of AGRC can be found in [38]. AGRC based compilation approach adopts two different ways to generate the code:

- Standard Java code
- A mixture of Java and Control Virtual Machine (CVM) assembly constructs [38]

We will discuss both of these approaches in detail in the next sections.

2.6.3 Standard Java Code

The SystemJ compiler back-end [26] generates the pure Java code. The threads are scheduled using the algorithm proposed in [28, 44] resulting in reduced code size when compared to statically scheduling synchronous threads. It also makes the back-end code portable to multiple processor architectures in future work. The compiler back-end generates the single threaded Java code which is much more efficient in terms of memory consumption and execution time compared to using Java threads like in TReK [25]. The generated code is operating system (OS) dependent as Java threads are mapped to native threads and thus the implementation depends upon the underlying OS scheduler. The verification techniques like model-checking can be applied more easily to a single threaded Java program as compared to a multi-threaded one.

```
system {
    interface{
       input signal C;
       output signal F;
  }
   {
     {
        signal A;
       {
        present (C)
        emit A;
        pause;
       }
       ||
       {
        emit F;
        System.out.println("data computations");
        pause;
       }
      }

         ×

     {
     }
  }
}
```

Figure 2.4: Simple SystemJ code showing syntax

A simple SystemJ program shown in Figure 2.4 is compiled using the Java only approach. The resultant Java code is shown in Figure 2.5. The *system* entity is compiled to Java *class* that bears the same name as SystemJ code file. The signals are compiled to Java objects. The clock-domains are compiled into the *methods* (line 28 – 34) which are called by the *main* method (line 46 & 50) recursively inside a *while* (this is Java *while* loop which is different from SystemJ *while* loop as mentioned earlier) loop as SystemJ programs are reactive in nature and run forever. The clock-domain *CD0* consists of two reactions shown as *reaction0* and *reaction1,* respectively. The synchronous concurrency of the reactions is emulated using Java *switch-case* statements. The *switch* statement (line 9) has three cases: the *case 0* (line 13) represents the very first transition of the reaction, *case* 1 (line 19) represents all other transitions and *case*2 (line 10) represents termination of the reaction. During the execution, it enters first into the *case* 0. Here it checks the presence of signal C and if it is present, A is emitted and reaction pauses with termination-code 1 as indicated by array *ends*[0] (line 17). The signal A is emitted by setting its status to true (*A.setPresent()*). The program then communicates with the environment by emitting the output signals to the environment. The Java program then proceeds to carry out the next tick. The second clock-domain is empty and it is necessary as SystemJ syntax does not allow to declare a system with one clock-domain only.

It should be pointed out that the different instances of the same signal A (example *A0* and *A1*) are used to avoid schizophrenic behavior. The compiler also generates signal and Channel *classes* to support the generic (Object type) data transfers and emissions. The clock-domains specified in SystemJ code are scheduled in a round robin fashion. Weighted round robin scheduling can also be applied where designer chooses a single clock-domain as a reference and all other clock-domains run at an integral multiple of this clock-domain's tick. The clock-domains can be deployed to run at unrelated speeds if they are executed on separate machines.

The single threaded Java code suffers from slow execution speed. During the compilation of the intermediate AGRC format into back-end Java code both synchronous and asynchronous concurrency is compiled away to produce single threaded Java program. The synchronous concurrency is emulated using state variables whereas asynchronous concurrency is emulated by using the scheduling schemes stated above. In case the clock-domains are scheduled cyclically, any unfinished clock-domain will have to wait for the other clock-domains to finish their execution thus causing unnecessary delays. The synchronous concurrency of the reactions is emulated by cyclic scheduling, which requires jump in the back-end generated Java code. Due to absence of *goto* in Java, this behavior is emulated using *if-else* and *switch* statements resulting in large code size. Also, Java is compiled to bytecodes and generally executed by interpreting JVM running on a processor, which have slower execution as they do not run natively in general. This demands an improvement in the compilation technique so that the above mentioned demerits in pure Java compilation approach may be avoided.

2.6.4 Separation of Control and Data-computations

In order to overcome the above mentioned problems in Java only approach, another approach is adopted which is inspired from earlier works on reactive processors such as REMIC [33], REFLIX [32], KEP [35] and the Esterel Virtual Machine (EVM) [31]. Both REMIC and KEP provide hardware support for the reactive programs. The SystemJ code is translated to AGRC and then data-driven operations and control-driven operations are separated [38]. The former is translated to Java, whereas the latter uses custom instructions to provide direct reactive support. Both can be executed by using the Java Virtual Machine (JVM) and the Control Virtual Machine

```
public class example {
  ...
  private static Signal F;
  private static Signal C;
  private static Signal A0;
  private static Signal A1;
    …
  reaction0(tdone,ends, gotocond){
   switch(S0){
     case 2:
       S0=2;
       break;
     case 0:
     A0.setClear();
     if(C.getStatus())
     A0.setPresent();
     ends[0]=1;S0=1;
     break;
     case 1:
     A1.setClear();
     ends[0]=0;S0=2;
     break;
   }
  }
  reaction1(tdone,ends, gotocond){
   ...
  }
  public static void CD0(){
    reaction0(tdone,ends, gotocond);
    reaction1(tdone,ends, gotocond);
  }
  public static void CD1(){
   ...
  }
  public static void main(String args[]){

    F = new Signal();
    C = new Signal();
    A0 = new Signal();
    A1 = new Signal();

    while(true){
    ...
      {
        C.gethook();
      cd0();
      }
    ...
      {
      cd1();
      }
   }
}
```

Figure 2.5: The single threaded pure Java code generated by the AGRC based SystemJ compiler for SystemJ example given in 2.2

(CVM), respectively. The control-driven operations represent the concurrency and control flow (CRCF) and data-driven operations represent control flow of Java data-computations described in Java (also called JCF). The CRCF always leads the JCF and calls JCF as and when required. This approach, thus, not only reduces the code size drastically, but it also improves the run-time efficiency.

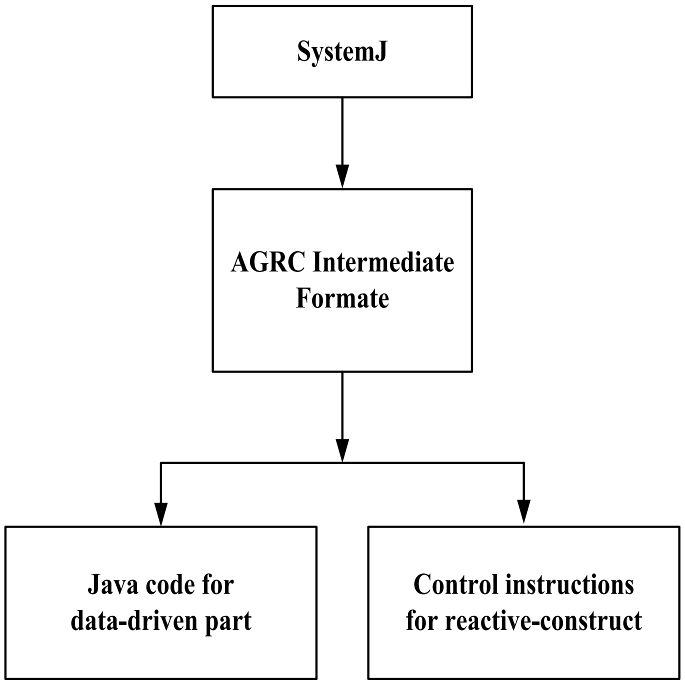

SystemJ
AGRC Intermediate Formate
Java code for data-driven part
Control instructions for reactive-construct

Figure 2.6: A SystemJ compilation approach separating control from data-computations

When building the AGRC, all "action nodes" [38] with pure Java data-computations are marked as Java data-driven action nodes or simply Java Action Nodes (JANs). Each JAN is given the unique identity code (JAN_ID). The *emit* statements are decomposed into two action nodes: one setting the value (if any) and the other setting the status of the signal. The former is marked as a Java action node. All Java action nodes which are not separated by the control flow are grouped together as one action node called a *macro action node* (or simply Java action node). During back-end code generation, for each clock-domain, a separate Java *switch-case* statement encapsulated by a Java method is generated where the *case-number* represents the macro action nodes to be executed and the method represents the clock-domain being executed.

Both asynchronous clock-domains and synchronous reactions are executed in a cyclic manner. Additional test-nodes called data-locks are inserted after each macro action node. When a control point comes across a macro action node, it makes a call to the JVM with the required case-number. Once all the sibling reactions are either suspended (with a termination code of ∞) or completed (with termination code of 1 or 0), the current reaction is checked again to see if the data-lock condition has been satisfied. The signals checked in the data-lock test-nodes are emitted by the JVM once it has completed processing the requested case-number. The JVM also sends back the result of the completed case-number processing.

The data structure for the CRCF is implemented in CVM data-memory (DM) which is arranged in a unique manner for fast computations. The input/output signals statuses (1-bit each) are stored first in the data memory and are word (16−bit) aligned, thus if we have 16 or less signals we use at least one memory space (word) in the DM. The internal signals are stored next. Internal signals can be emitted from multiple synchronous reactions in a given clock-domain and thus we assign 1-bit lock status for each signal per synchronous reaction. Next, data-locks are stored which inform if the calls made for data-computations (Java action node in AGRC) to the JVM have been returned. A complete DM word (16-bits) is used for data-locks and program counters (PC) for the various synchronous reactions in a clock-domain which are stored in the following locations. Then up to four terminate nodes, four-bits each, are stored in a single DM word. The switch nodes used for state selection of the currently executing SystemJ program are stored next. The switch nodes can have varying number of children/branches, each indicating the next state of the program. The switch nodes and their children are stored together with children arranged in an ascending order after the switch node. This arrangement of DM is repeated for each clock-domain [38]. Figure 2.7 below shows the data structures for the CVM.

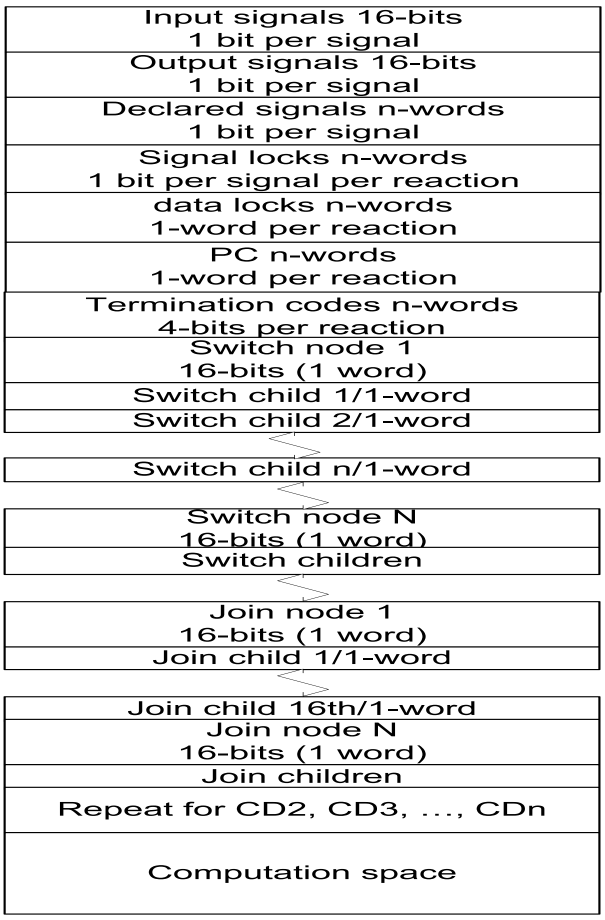


Figure 2.7: CVM data structure

The CVM has fourteen instructions namely *load* (LDR), *store* (STR), *add* (ADD), *subtract* (SUB), *jump* (JMP), *and* (AND), *or* (OR), *clear flag* (CLF), *present*, *switch*, *seot*/*ceot*, *sendata* and *chkend*. The instruction set has four addressing modes: immediate (instruction has an immediate operand value), inherent (instruction does not contain any operand or register), registered (instruction has one or more registers as operand), and indirect (instruction contains an operand as a pointer into the memory). The description of some specific instructions is provided while others are regular instructions.

- *present:* The present instruction checks the status of the signal. If status is true, the program executes the next instruction, otherwise the program counter gets reset to the false branch. All present and preemptive statements (*abort* and *suspend*) of SystemJ are mapped to this present instruction (test nodes in AGRC representation). The *present* instruction also provides direct support for the environment signal (mapped to memory) checks and thus, reduces the code size.
- *sendata:* This instruction is used to make Java data calls to the JVM by providing the case number and a pointer into memory which holds the data lock position.
- *chkend:* This instruction is a special instruction which acts as the synchronizer for the || operator. It compares the termination codes of the synchronous concurrent reactions within a clock-domain and makes the decision regarding the continuation context.
- *switch:* The switch instruction is used to directly decode the switch nodes in the AGRC. Like present, this instruction reduces the code size.
- *seot/ceot:* The *seot/clfeot* are special instructions used to implement the environment communication semantics of clock-domains and setting the boundaries of instants of time i.e., tick. The seot instructions sets the register indicating the end of tick. The ceot instruction clears it before starting next tick.

Figure 2.8 shows the result of compilation of SystemJ code in the previous example for the approach in which control and data-computations are separated. The data-computations are translated to Java and control is compiled into assembly code. The Java code does not contain any data-computations but the compiler inserts a data-call in order to synchronize with the machine executing the data-computation. This is done during the initialization. The compiled Java class contains a method CD0, which represents the 'test' clock-domain. It contains a *switch* statement whose case-numbers represent the various Java instantaneous action-nodes. The control instructions are responsible for the complete control-flow of SystemJ program. Assuming that multiple clock-domains are synchronized, the output signals located at memory location *$0* are initialized first. Next, we check for the presence of the input signal. If present signal *A* is emitted, otherwise it jumps to the *BB0* and skips the emission of signal. After that the reaction pauses with termination code of *1* and the resume address of the reaction *1* indicated label *AA0* is stored. Once, we are finished with the execution of *reaction1*, execution of *reaction2* starts which emits the signal *F*. The data-lock position and the case-number (each one byte long) are concatenated to be sent to the JVM. SENDATA instruction sends this word at the output. The second reaction also pauses with termination code of *1* which is stored in the second nibble of the memory location. After storing the resume address of *reaction2*, the instruction CHKEND checks for the termination code of the reactions and, if any reaction has not completed (terminated or paused), it is executed again.

Java Code

```
public class test{

public static void main(){

while(true){
/*polling on native C method
to get the casenumber*/
 int casenum = poll();
 boolean ret = CD0(casenum);
 /*sending result using C*/
 send_res(ret);
  }
 }
public boolean CD0(int cnum){
 boolean ret = false;
 switch(cnum){
  case 0:
// dummy method inserted
// before CD starts execution
   System.out.println("cd0");
   break;
    }
return ret;
  }
}
```

Control Assembly Code

```
  LDR R1 #0;
  STR R1 $1; output signals
  // reaction 1
AA0 LDR R0 $0;
  AND R0 R0 #32768;
  PRESENT R0 BB0; checking signal C
  LDR R0 $6; loading from mem
  OR R0 R0 #32768; signal mask
  STR R0 $6; emit local signal A
BB0 LDR R0 #1;
  LDR R1 $a
  AND R1 R1 #$FFF0
  OR R1 R1 R0
  STR R1 $a; reaction terminates with 1
  LDR R0 $1;
  LDR R0 #AA0; store PC
  STR R0 $8; stored in memory
//reaction 2
FF1 OR R0 R0 #32768 ; signal mask
  STR R0 $1; emitted the signal F
  LDR R0 #2; lock position
  ADD R1 R6 #0
  STR R1 #0;
  SENDATA R0; data call
  LDR R0 #16; reaction terminates
  LDR R1 $a
  AND R1 R1 #$FF0F
  OR R1 R1 R0
  STR R1 $a;
  LDR R0 #FF; store PC
  STR R0 $9; stored in memory
  CHKEND R2 R0;
  ...
```

Figure 2.8: SystemJ compilation to mixture of Java and assembly

2.7 The SystemJ Execution Platforms

In the last section, we have introduced the SystemJ compilation approaches which are divided into two categories: one which compiles SystemJ program to Java only, and another one separates the data-computations from control computation and translate them to Java and assembly code respectively. The resulting code from these two categories can be executed on different platforms ranging from desktop to multiprocessor based embedded systems. We will restrict our discussion to the execution of SystemJ programs on embedded platforms only.

2.7.1 Execution Platforms for Standard Java Approach

In Java only compilation approach, SystemJ is fully compiled to Java. The translated Java code is compiled into an intermediate representation called Java byte-code. The Java byte-codes are not compiled directly to platform-specific machine code due to the portability reasons which provides the capability to the computer programs written in Java language to run similarly on any hardware/operating-system platform. Java byte-code instructions are analogous to machine code, but they need to be interpreted by a virtual machine (VM) written specifically for the host hardware. The simplest realization of the JVM is a program that interprets the byte-code instructions.

A Java virtual machine starts execution by invoking the method *main* of some specified class, passing it a single argument, which is an array of strings. The Java virtual machine dynamically loads, links, and initializes classes and interfaces. Initialization of a class or interface consists of executing the class or interface initialization method. Initialization consists of execution of any class variable initializers and static initializers of the class, in textual order. But before class can be initialized, its direct superclass must be initialized, as well as the direct superclass of its direct superclass, and so on, recursively. The class initialization is a recursive process that initializes all superclasses in the inheritance hierarchy.

Embedded systems have traditionally been differentiated from desktop systems on the basis of functionality: desktop systems provide a wide spectrum of technologies to serve a broad range of application needs, while embedded devices are fitted with just enough software to handle a specific application. Unlike desktop systems, embedded systems use different user interface technologies; have significantly smaller memories and screen sizes; use a wide variety of embedded processors; and have tight constraints on power consumption, user response time, and physical space.

In order to run SystemJ programs translated to Java on an embedded system implementing a JVM, we need Java API package, and associated native-code libraries alongside the JVM. All these add to the overall size of platform to be resided in the memory. Current Java API packages tend to be large, so the specific API selected will significantly impact its size. Added components can also affect platform size. Desktop JVMs usually can't execute Java code directly from ROM. Normally; Java classes are first loaded into RAM, verified, and then executed by the JVM. This approach is impractical for many embedded systems because it increases the use of expensive RAM beyond the cost constraints of the embedded system. A tool called *ROMizer* [45] can be used to create ROM-based executable images of Java classes. To execute Java code out of ROM instead of RAM, a ROMizer utility processes Java class files into a run-time format that can be run directly out of ROM or flash memory by the JVM. The ROMizing process frees the JVM from the class-file loading and byte code–verification phases, and improves the performance and start-up time of Java applications.

In the current implementation, SystemJ applications compiled to Java are executed on KVM port of an interpreting JVM running on a general purpose Nios II Soft-Core Processor [46, 47]. The processor has a 32-bit Instruction Set Architecture, 32 general-purpose registers, single-instruction 32x32 multiply and divide operations, and dedicated instructions for 64-bit and 128-bit products of multiplication. It has Harvard memory architecture and is a Reduced Instruction Set Computer (RISC) processor core widely used with Altera FPGAs and SOPC Builder. The processor has three versions: economy, standard and fast. These versions have varying number of pipeline stages; instruction and data cache; memories and hardware components for multiply and divide operations. In addition, each version varies in size and performance depending on the features that are selected. Adding peripherals to the Nios II Processors is done through the Avalon Interface Bus [48] which contains the necessary logic to interface the processor with other components. Furthermore, it is hard to predict the worst case execution time due to the complex nature of general purpose processor running JVM.

An interpreting JVM is simple to implement as it imposes no perceivable interruption and does not need to store the code compiled to native code during execution. However, interpreted JVMs can be orders of magnitude slower than compiled code and waste significant resources, e.g., CPU, memory, battery etc. In addition to the standard interpreted JVM implementation, a variety of execution techniques have been proposed to reduce the execution time of Java programs. These accelerating techniques along with the JVM structure are explained in the Chapter 3.

2.7.2 Execution Platforms for Separation of Control and Data-computations Approach

The SystemJ program compiled using the approach where control computations and data-computations are separated, can be executed in two ways. It can be executed by using virtual machines or processor. Both these approaches are discussed next.

Tandem Virtual Machine

The execution approach which uses virtual machines for the execution of SystemJ programs is called Tandem Virtual Machine as two virtual machines, Java Virtual Machine (JVM) and Control Virtual Machine (CVM), are operating in tandem [38]. The JVM executes data-computations and can be either a full standard Java virtual machine (J2SE) or a standard Java micro edition (J2ME) running on microprocessor platforms. The CVM performs control computations and consists of a program memory, data memory, the ALU, and the instruction sequencer. The program memory contains the compiled and assembled SystemJ control statements that are executed by CVM. The program memory of CVM is implemented as a linked-list with each node capable of holding an assembler word of 32-bit each. The ALU performs only the most basic instructions like addition, subtraction, and certain custom instructions providing direct support for AGRC execution. The CVM data structure resides in the data memory as explained in Section 2.6.4.

Both JVM and CVM are run as two operating system processes. CVM is responsible for the control flow and starts the processing always leading the JVM. The JVM, after starting the execution, always polls the CVM for the data calls. Once a data call is received, it performs the required computation and sends the result back to the CVM. The communication interface between the CVM and the JVM is shown in Figure 2.9. As mentioned earlier in the text, various data-computations that need to be processed by the JVM are enclosed in a *switch-case* statement wrapped in a method. The data call is made through the *sendata* instruction which provides the case-number and a pointer into memory which holds the data lock position. During processing of the sendata instruction, the CVM clears memory pointed to by the data lock position to ensure that the current reaction does not proceed further without the completion of the requested data call. CVM uses two pointers into the FIFO buffer called the “tail pointer” (*tptr*) and the “head pointer” (*hptr*) which indicate the data call made and data call in process, respectively. CVM checks *tptr*, if it is pointing to the memory location in data-memory, *tptr* is reset to the start of the FIFO, otherwise the contents of data calls are transferred into the data-memory pointed to by the *tptr* and the *tptr* is incremented (effectively implementing a circular FIFO).

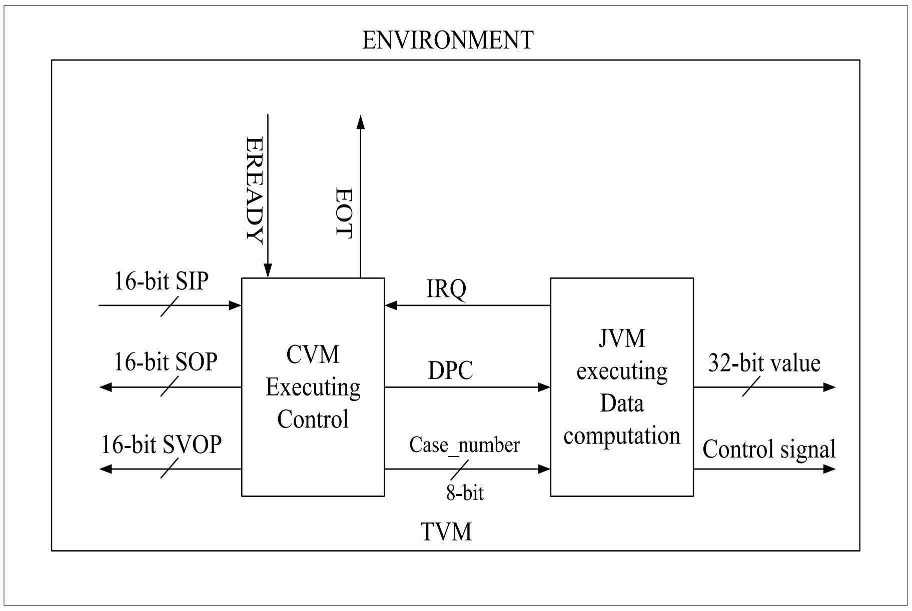


Figure 2.9: TVM execution platform with communication interface between virtual machine and with the environment

CVM communicates with the JVM through DPC, DPCR, and the IRQ registers. When CVM makes a data call to JVM by providing the case-number and data-lock position in the DPCR register. The DPC register is set high indicating to the JVM that a data-call request has been made. When JVM finishes the requested computation, it sets the IRQ register high. The JVM and CVM are two system processes executing asynchronously and time for the result from JVM is unknown to CVM, therefore, result returned by the JVM is managed through interrupt. The CVM fetches next instruction from the memory only if it has not made a data call or there is no pending call. If there is a pending call, DPC is set, then it checks IRQ bit which is set if JVM has entertained the data call. If IRQ is high, CVM reads the result from JVM and writes into the data-memory location pointed to by the *hptr*. Next the *hptr* is checked to see if it has reached the last memory position, if so it is reset otherwise it is incremented. Then the *hptr* and the *tptr* are compared to check if there are any pending calls in the FIFO. If they are equal, all data-calls are considered finished else the next data call is made to the JVM. If the IRQ bit is not set, then the data call is made to the JVM and the procedure returns. SIP and SOP registers hold the statuses of input and output signals. The SVOP register holds the status of valued signals wheresa values are handled by the JVM. EOT regsiter indicates the end of tick to the environment.

Drawbacks of TVM Approach

The TVM execution platform discussed above has many advantages such as smaller memory footprint and executes the compiled SystemJ programs faster than Java only approach. But this execution platform is only suitable for desktop computers and high-end embedded systems. The TVM platform uses two virtual machines which imposes a high penalty on the memory and computing power requirement. Thus, this approach is not suitable for the low-end embedded systems, which cannot meet the aforementioned requirements. Furthermore, the CVM and JVM communicate through Java Native Interface (JNI) sharing the same virtual address space on the same processor which is a hurdle in multiprocessor chip (MPSoC) implementation. These problems led to development of new execution platform, called Tandem Processor (TP), which is well suited to the low-end embedded systems.

Tandem Processor

TP [39] is based on the hardware implementation of TVM, where CVM and JVM are replaced by the Control Processor (CP) and Data Processor (DP), respectively. They run in tandem and perform control and data-computations. The main components of the CP are the program and data memories, the ALU, the MAX unit, the register file and the instruction sequencer. The data-memory is arranged in the same way as the CVM. The ALU is 16-bit and performs simple arithmetic operations. The MAX unit is a special hardware unit and is used to find the maximum termination code. The program memory of the CP holds the compiled and assembled SystemJ assembly instructions. CP is a hardware implementation of CVM and shares the instruction set architecture (ISA) with the CVM. Only few instructions are added to ISA which are required to communicate with the environment as give below:

- *LSIP:* This instruction loads the input signals statuses from the environment.
- *SOP:* Present output signal statuses to environment.
- *SVOP:* Present output signal values to environment.

The CP guides the control flow of SystemJ program. It starts the processing and always leads the DP. The Java based data-driven operations are executed on the JVM running on the DP which, once started, keeps on polling the CP for the data calls. After receiving the data call, DP computes the results and responds by providing the result back to the CP. The Java data-computations are wrapped in a *switch-case* statement where each case number represents a Java macro action node. The CP sends the case number to be processed to the DP along with the data lock address using the *SENDATA* instruction which provides the case-number and the data lock address in a register as upper and lower bytes respectively. The currently executing clock-domain number (four bit) is initialized in a register (16-bits) at the start of processing a clock-domain. During the execution of the sendata instruction, the sendata instruction register and the clock-domain register are concatenated together. Upon receiving the request, the DP uses the clock-domain number to find the appropriate (clock-domain) method and the case-number to execute the correct macro action-node. The result of this computation is a two bit result vector 0bX1, where the zero bit shows the completion of this data call and the first bit is the result of the computation. Upon availability of the result, CP is halted and result vecotr is directly stored in the data-memory of CP without requiring any FIFO. The The data-lock address, bits [0..7] of the sendata register, is used for this store operation.

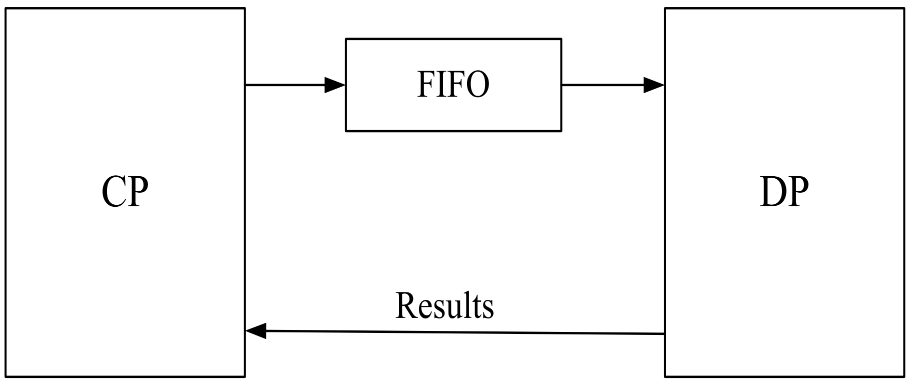


Figure 2.10: TP execution platform with FIFO as communication interface between control and data processor

In SystemJ each clock-domain communicates with the environment once at the start of its tick. The CP reads the input signal statuses from the environment and emits the output signal statuses to the environment at the end of tick (EOT). The DP reads the input signal values and emits the

Instruction	**Register Transfer**	**Mode**
AND Rz Rx Operand	$Rx\&\&Operand \rightarrow Rz$	*immediate*
	$Rx\&\&Rz \rightarrow Rz$	*indirect*
OR Rz Rx Operand	$Rx\|\|Operand \rightarrow Rz$	*immediate*
	$Rx\|\|Rz \rightarrow Rz$	*indirect*
ADD Rz Rx Operand	$Rx + Operand \rightarrow Rz$	*immediate*
	$Rx + Rz \rightarrow Rz$	*indirect*
SUBV Rz Rx Operand	$Rx - Operand \rightarrow Rz$	*immediate*
SUB Rz Rx Operand	$Rx - Operand$	*immediate*
LDR Rz Rx Operand	$Operand \rightarrow Rz$	*immediate*
	$M[Rx] \rightarrow Rz$	*indirect*
	$M[Operand] \rightarrow Rz$	*direct*
STR Rz Rx Operand	$Operand \rightarrow M[Rz]$	*immediate*
	$Rx \rightarrow M[Rz]$	*indirect*
	$Rx \rightarrow M[Operand]$	*direct*
JMP Rx	$Operand \rightarrow PC$	*immediate*
	$Rx \rightarrow PC$	*direct*
PRESENT Rz Operand	$ifRz(0) = 1thenOperand \rightarrow PCelseNEXT$	*immediate*
SENDATA Rx	$Rx \rightarrow FIFO$	*indirect*
CHKEND Rz Rx	$MAX\{Rx[15:12], Rx[11:8], Rx[7:4], Rx[15:12], Rz[3:0]\} \rightarrow Rz$	*indirect*
SWITCH Rz Rx	$M[Rx] \rightarrow Rz, Rz + Rx + 1 \rightarrow Rz, M[Rz] \rightarrow PC$	*indirect*
SZ Operand	$ifZ = 1thenOperand \rightarrow PCelseNEXT$	*immediate*
CLFZ	$0 \rightarrow Rz$	*inherent*
CER	$0 \rightarrow ER$	*inherent*
CEOT	$0 \rightarrow EOT$	*inherent*
SEOT	$1 \rightarrow EOT$	*inherent*
LER Rz	$ER \rightarrow Rz$	*indirect*
SSVOP Rx	$Rx \rightarrow SVOP$	*indirect*
LSIP Rz	$SIP \rightarrow Rz$	*indirect*
SSOP Rx	$Rx \rightarrow SOP$	*indirect*
NOOP	$No\,Operation$	*inherent*

Table 2.1: Control processor instructions and their description

values of the output signals to the environment. The environment communication interface in CP consists of the *Signal Input Port* (SIP) and *Signal Output Port* (SOP), used to communicate the status of the signals and *Signal Value Output Port* (SVOP) for indicating to the environment that the emitted output signal is a valued signal. The CP also utilizes the *EOT* and *EREADY* (Environment Ready) ports as control signals to coordinate the communication with the environment. Current implementation of CP consists of a single set of SIP, SOP and SVOP ports for all the clock-domains implemented on this CP. The maximum number of the environment interface signals is restricted to 16 in the complete SystemJ system. All the clock-domains share the interface ports and data-memory locations 0x0, 0x1 to store their interface signals statuses. The signals statuses loaded and stored from/to the interface ports for different clock-domains are separated using bit masking. The signals are stored in the ports/memory locations in their order of appearance in SystemJ program from the least significant to the most significant bit. The system also has the draback of not being predictable.

2.8 Summary

In this chapter, we have provided an overview of important features of a system level design language for modeling the GALS systems. It described SystemJ MoC which is actually GALS MoC, SystemJ program entities and its kernel statements. The SystemJ programming model was explained in detail with help of a sample program. The SystemJ program can be compiled either using library or AGRC based approach. The latter can produce a standard Java code or splitting the source code into CRCF code and Java based JCF code. Different execution platforms along with pros and cons for the resultant codes are discussed. The SystemJ program compiled to single threaded pure Java code is executed on general purpose processor. The SystemJ code compiled separating control and data-computations are executed on a heterogeneous multiprocessor architecture resulting in improved performance and reduced memory footprint.

Chapter 3

Java and SystemJ Execution

The SystemJ compiler translates the programs described in SystemJ language to Java. The Java compiler reads Java language source (*.java*) files, translates the source into Java bytecodes, and places the bytecodes into class (*.class*) files. The compiler generates one class file per class in the source. The Java bytecodes runs on top of Java Virtual Machine. This chapter gives an introduction of the Java Virtual Machine (JVM), its implementation approaches, Java processors and, in particular, Java Optimized Processor (JOP). The JOP is introduced as an execution platform for SystemJ program. Then a new core, called RJOP is described which extends JOP by incorporating reactivity for more efficient execution of the SystemJ programs. The performance evaluation is presented in the end to show the effectiveness of the approach.

3.1 Java Virtual Machine

The Java Virtual Machine (JVM) is a definition of an abstract computing machine that executes bytecode programs. The JVM specifications [49] defines an instruction set, called bytecodes, and the meaning of those instructions. It also defines the class file format which consists of the bytecodes, a symbol table and other ancillary information and an algorithm to verify whether a class file contains valid programs or not. The instruction set of the JVM is stack-based and can be seen as CISC (complex instruction set computer). Unlike processor of contemporary design which use registers a stack machine uses stack. Stack is a LIFO (last in first out) storage with two abstract operations : push, pop. Push will put an item into stack at the top. Pop retrieve an item at the top of stack. Stack doesn't need addressing as it is implicit in the operators which use stack. All operations take their arguments from the stack and put the result back onto the stack. Addition operation takes top two elements from stack, adds them and push the result back to stack. Store operation takes one value and one address from stack and store value to address.

3.1.1 JVM Instruction Set

Each JVM instruction is one byte in length, although some require parameters, resulting in some multi-byte instructions. They usually do not contain any information about the location of the operands as they are bound to be the top two elements of the stack. Most of the JVM instructions operate on one specific operand type and all such instructions have the type they operate on encoded in their name. The JVM instructions can be categorized in the following way:

Constant Loading Instructions

These instructions are used to load constants onto the top of the stack. Constants can be of type *long*, *integer*, *float*, *string* and *null*. There are unique instructions for each basic type (*int*, *long*, *float*, *double* and *reference*).

Variable Access Instructions

Variables are located in the run-time stack frame of a JVM method stack. Most of them take two bytes except for *double* and *long* values, which take four bytes. In JVM, there are not many instructions which deal directly with the variables (the only exception is *iinc*). The values of variables are loaded onto or stored from the top of the stack by using *load* and *store* instructions.

Array Operation Instructions

Arrays are objects in the JVM and *Array elements* can have all Java types and can also be *boolean*, *byte*, *char* and *short*. JVM has dedicated instructions to deal with their attributes and data elements. They create a new array, load an array component from memory onto the operand stack, store a value from the operand stack to an array component and provide the length of array.

Data Member Access Instructions

A JVM class can have *class-wide* (static) and *instance-wide* (non-static) data members.

Type conversion

The type conversion instructions perform numerical conversions between all Java types.

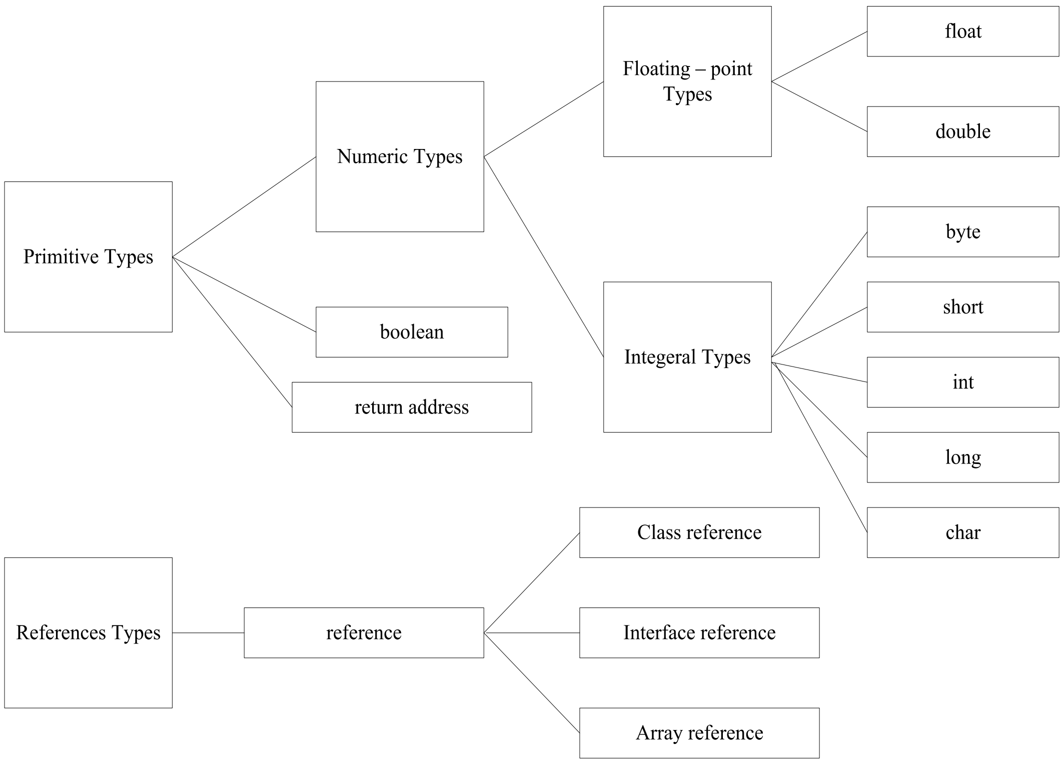


Figure 3.1: Java Data types

Object Creation Instructions

The instruction *new ClassName* creates an instance of that class on the stack. It must be initialized by an explicit call to one of its constructors.

Arithmetic and Logical Instructions

These instructions do arithmetic calculations on the parameters from the stack and store the result back on the stack. There are arithmetic instructions for *int*, *float* and *double*. There is no direct support for *byte*, *short* or *char* types. These values are handled by *int* operations and have to be converted back before being stored in a local variable or an object field.

Stack Manipulation Instructions

There are a number of instructions to manipulate the stack top of JVM and, therefore, JVM is stack based and uses no registers; these instructions are helpful in speeding up certain operations.

Method Invocation Instructions

Four instructions are used to invoke different kinds of methods: *invokestatic* is used to call static methods, *invokeinterface* is used to call interface methods, *invokevirtual* is used to call methods of object instances; and *invokespecial* is used to call special methods such as constructors or methods of the super classes.

Un-conditional Jump Instructions

The program execution can be unconditionally changed to a location that may not be the next instruction in the flow. JVM adds the offset to the current program counter and resulting address must point to an instruction in the current method.

Conditional Jump Instructions

The JVM provides a complete set of branch conditions for *int* values and references. They provide integer comparison with *zero*, comparison of two *integers* and comparison of *longs*, *floats*, and *doubles*. Branch target addresses are specified relative to the current address with a signed 16-bit offset.

Other Instructions

The *nop* instruction does nothing. It can be used as a placeholder for testing purposes. There are two synchronization instructions, *monitorenter* and *monitorexit*, to implement object-based synchronization. They both take an object on the stack as their parameter. Method synchronization is denoted by the *synchronize* attribute. *athrow* instruction throws an exception; note that other JVM instructions can also throw exceptions when they detect an abnormal condition.

The JVM bytecode set has 212 opcodes, with 4 more reserved for future use/expansion. For full details of all the different types of constants, as well as the instruction set, refer to the Java Virtual Machine Specification [49]. A complete list of JVM bytecodes is provided in Appendix B.

3.1.2 Class File Format

The Java class file contains everything the JVM needs to know about one Java class or interface. Information stored in it often varies in length as actual length of the information is available only during the loading of the class file. For instance, the number of methods listed in the methods component can differ among class files, because it depends on the number of methods defined in the source code. Such information is organized in the class file by prefacing the actual information by its size or length. This way, when the class is being loaded by the JVM, the size of variable-length information is read first. Once the JVM knows the size, it can correctly read in the actual information. The order of class file components is defined and is also strict. It provides the JVM with a way to find the location of different components during the loading of a class file. The first eight bytes of a class file contain the *magic* and *version numbers*. The items in the *ClassFile* structure are shown in Figure 3.2. The *constant pool* starts on the ninth byte followed by the access flags. Then follows the constant pool because the constant pool has variable-length and doesn't know about the exact location of the access flags until it has finished reading in the constant pool.

```
ClassFile {
      u4 magic;
      u2 minor_version;
      u2 major_version;
      u2 constant_pool_count;
      cp_info c  onstant_pool[constant_pool_count-1];
      u2 access_flags;
      u2 this_class;
      u2 super_class;
      u2 interfaces_count;
      u2 interfaces[interfaces_count];
      u2 fields_count;
      field_info fields[fields_count];
      u2 methods_count;
      method_info methods[methods_count];
      u2 attributes_count;
      attribute_info attributes[attributes_count];
     }
```

Figure 3.2: Class file format showing all components in orderly fashion

3.1.3 JVM Run-time Data Areas

When JVM runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects of the program instantiated, parameters to methods, return values, local variables, and intermediate results of computations. The JVM organizes the memory it needs to execute a program into several run-time data areas. Some of these areas are shared between threads, whereas other data areas exist separately for each thread. Although the same run-time data areas exist in some form in every JVM implementation, their specification is quite abstract. Many decisions about the structural details of the run-time data areas are left to the designers of individual implementations. The internal architecture of JVM with all the run-time data areas is given in Figure 3.3.

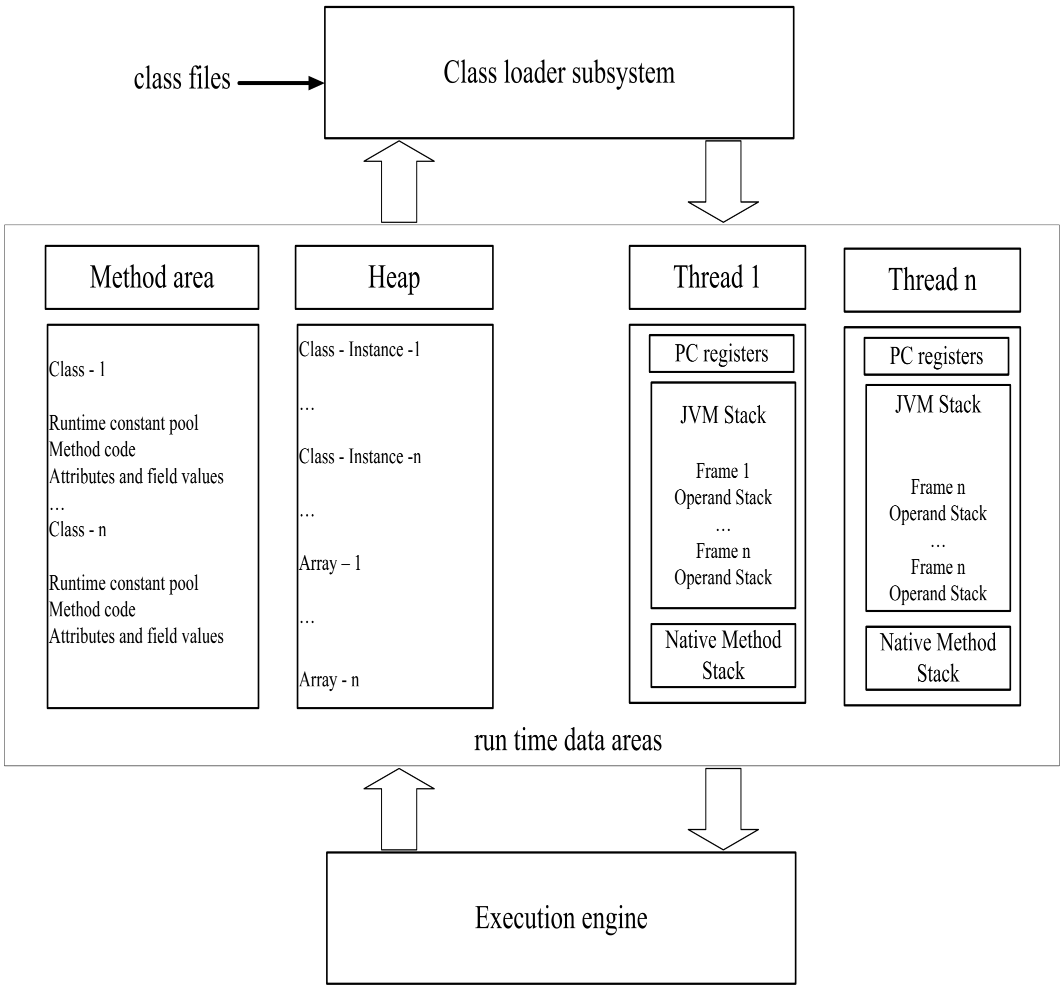


Figure 3.3: The internal architecture of the JVM

Stack

The JVM creates a new Java stack for each new thread launched. A thread's stack stores the state of Java method invocations for the thread and includes its local variables, invoking parameters, its return value, and intermediate calculations as shown in Figure 3.3. The *stack frame* for each method call is placed in the overall stack in the last-in, first-out fashion. Each stack frame in turn consists of the operand stack for that method call, a section for arguments and local variables, and some other data. The area in the stack where the operands are stored is called *operand stack*. This is the area on which the method's instructions operate. All the *local variables* and *arguments* for the method are also stored in the local variable section of a stack frame with the arguments stored first and then the locals. The arguments and locals are stored in the order of their declaration. The Frame Data section of a stack frame contains a pointer to the caller's stack frame. This enables return to the caller when the called method finishes, and enables it to put the return value, if any, into the caller's stack frame. The JVM performs only *push* and *pop* operations directly on Java stacks. The data on a thread's Java stack can only be accessed by that thread and other threads cannot access or alter the Java stack.

Method Area

Method area is shared among all the threads. It contains static class information such as *field* and method data, the code for the methods and the *constant pool*. JVM restricts the maximum size of a method to 64K bytes. The classes used for the execution program are stored in the method area which includes bytecodes of the methods of the class. For each type it loads, the JVM stores the fully qualified name of the type, the fully qualified name of the direct superclass, the information that a type is a class or an interface, the type's modifiers, an ordered list of the fully qualified names of any direct superinterfaces, the constant pool for the type, field information, method information, class (static) variables declared in the type, except constants, reference to class ClassLoader and reference to class *Class*. The constant pool is a per-class table, containing various kinds of constants such as numeric values or method and field references. The constant pools contain the strings and numeric literals used by the program and field references. The method area size may or may not be fixed. When Java application runs, the JVM can increase and decrease the method area to meet the application's needs. Since the method area is shared with all the threads, access to the data structures of method area should be thread-safe. The type information stored in the method area must be organized in a way that it can be accessed quickly. It may include other data structures such as method table to speed up the access. A method table consists of an array of direct references to all those instance methods that can be invoked on a class instance, and includes instance methods which are inherited from superclasses.

Heap

Heap is the run-time data area where Java objects exist and memory to all class instances and arrays are allocated from here. Whenever an object is created as a result of the invoking of Java *new* operation, the necessary memory for the object is allocated within the heap. This space is shared among all threads as there is only one heap in JVM. Heap memory for objects is reclaimed by an automatic memory management system which is known as a *Garbage Collector* (GC). The heap may be of a fixed size or may be expanded and shrunk, depending on the strategy of GC.

3.2 JVM Implementations

JVM can be implemented in following different ways [50]:

3.2.1 Interpreter

The simplest realization of the JVM is a program that interprets the bytecode instructions. Interpreting JVM is simple to implement, imposes no perceivable interruption and does not need to store the code compiled to native code during execution. However, interpreted JVM can be orders of magnitude slower than compiled code and wastes significant resources, e.g., CPU, memory, battery etc. The CACAO [51] and Squawk Virtual Machine [52] are the two examples of two open source interpreters for embedded systems.

3.2.2 Just-In-Time Compilation

JIT compiler translates Java bytecodes to native instructions during run-time. A Just-In-Time (JIT) compiler translates Java bytecode into native machine language. It does this while it is executing the program. Just as for a normal interpreter, the input to a JIT compiler is a Java bytecode program, and its task is to execute that program. But, as it is executing the program, it also translates parts of it into machine language. The translated parts of the program can then be executed much faster than they could be interpreted. Since a given part of a program is often executed many times as the program runs, a JIT compiler can significantly speed up the overall execution time. Typical JIT and Dynamic Adaptive Compiler (DAC) solutions are highly complex pieces of software and require significant effort to tune for the target platform, increasing cost of ownership. KAFFE JIT compiler [53], CACAO JIT [51], AJIT [54] and FAJITA [55] are few examples.

3.2.3 Batch Compilation

Java can be compiled, in advance, to the native instruction set of the target. Like traditional high-level language compilers, a direct Java compiler starts with an application's Java source code (or, alternatively, with its bytecode) and translates it directly into the machine language of the target processor. Unlike JIT compilation, batch compilation is done statically and can apply time-consuming techniques which are not possible in former case. The benefit is the increased performance as system relies more on the mature native compiler rather than bytecode based Java compiler. Although it provides good performance, it can be extremely costly in terms of memory code bloat. Also, for device updates we have to rely on native code patches to selectively update certain aspects of the application. Caffeine [56] and JaNi [57] are two examples which translate the bytecode directly into machine language.

3.2.4 Hardware Implementation

A Java processor is the implementation of the JVM in hardware where bytecodes serve as the native instructions set. The bytecode interpretation is performed in hardware without incurring any of the extra execution overheads and memory usage, as compared with software solutions, thus making it an interesting execution system for embedded systems programmed in Java. This hardware implementation of JVM is also referred to as Java processor and executes bytecodes faster than the interpreter JVM.

3.3 Java Processors

Two different approaches can be found to improve Java bytecode execution by hardware. In the first type, a Java coprocessor is introduced in the instruction fetch path of a general purpose processor which translates the Java bytecodes to sequences of instructions for the host CPU or directly executes basic Java bytecodes whereas complex instructions are emulated by the main processor. In the second category, all the Java bytecodes are executed by the Java processor. We will discuss some of the important processors proposed by the industry and academia till date.

picoJava-I

picoJava-I [58] is a configurable processor core that supports the JVM specification. It has a RISC-style pipeline executing the JVM instruction set. However, only the most common instructions that most directly impact the program execution are implemented in hardware. Some moderately complicated but performance critical instructions are implemented through microcodes. The remaining instructions are trapped and emulated in software by the processor core. The hardware design is thus simplified since the complex instructions are implemented in software as they are not executed frequently. It does not have any instructions cache.

picoJava-II

The picoJava-II core [59, 60] is the successor to the picoJava-I processor and augments the bytecode instruction set with a number of extended instructions which manipulate the caches, control registers, and absolute memory addresses. These extended instructions are useful for non-Java application programs that are run on this core. The programs are compiled to Java bytecodes first as these bytecodes are the processor's native instruction set. The pipeline is extended to 6 stages compared with the 4 stages in the picoJava-I pipeline. It does not have instructions cache either.

microJava

The Sun microJava 701 microprocessor [61] is based on the picoJava-II core. It is supported by a complete set of software and hardware development tools. It also has a six-stage pipeline and it can execute instructions without stalls. Also added is the folding ability that lets the processor combine up to four instructions.

Ignite, Patriot PSC1000

This microprocessor [62] is a general-purpose 32-bit, stack-oriented architecture whose instruction set is very similar to the JVM bytecodes and it can efficiently execute Java programs. It is one of the family of low-power, low-cost, stack architecture processors targeted specifically for embedded applications. The PSC1000 CPU instruction sets are hardwired and most of the instructions are executed in a single cycle without using the pipelines or superscalar architecture. The PSC1000 family also runs C and C++ efficiently as they are semantically similar to Java. It can also run stack-architecture based languages such as Forth [63] and Postscript™. The PSC1000 was later renamed to Ignite.

aJ-100

The aJile Systems aJ-100 [64] is the first device in the family of single-chip Java microcontrollers that directly execute JVM bytecodes, real-time Java threading primitives and a number of extended bytecodes for embedded operations. This processor operates at 100 MHz, uses 32-bit direct execution, incorporates five 8-bit discrete general-purpose I/O ports, and integrates 48 Kbytes of SRAM and a memory controller for ROM, SRAM or flash memory.

Komodo

This core [65] features the direct execution of Java bytecode, a zero-cycle context switch overhead and hardware support for scheduling and garbage collection. It is designed for embedded applications; therefore, the target architecture has a simple pipelined processor kernel, which is able to issue one instruction per cycle. The multithreading is supported through a hardware event handling mechanism that allows handling of simultaneous overlapping events with hard real-time requirements. Real-time Java threads are used as interrupt service threads (ISTs) instead of interrupt service routines (ISRs) for event handling. It has zero-cycle context switching allowing the Komodo microcontroller to react very fast on external events. ISTs are triggered without the typical overhead of ISRs in conventional processors. This helps in embedding the hardware scheduler late in the processor pipeline. It enables the scheduling on an instruction-per instruction basis due to a hardware-implemented real-time scheduling scheme.

Jamuth

Jamuth [66] is a Java multithreaded processor core for embedded real-time systems and is an enhancement of the Komodo [65]. It features a real-time capable incremental garbage collection, integrated real-time scheduling schemes and full compatibility to Java CDC standard. Due to its design as an IP core for Altera's System-on-Programmable-Chip (SoPC) environment, it can easily be combined with other (peripheral) components to a whole system on a single chip. In addition, the usage of Java decreases the effort of software development and maintenance in a significant way.

femtoJava

This [67] is a stack-based microcontroller that executes Java bytecodes. Its key attributes are reduced instruction set, Harvard architecture and small size. It was designed specifically for the embedded system market. It has both multi-cycle and pipelined versions. The multi-cycle version takes three to fourteen cycles to execute an instruction. The pipelined architecture [68] has five stages: instruction fetch, instruction decoding, operand fetch, execution, and write back. The operand stack and local variable pool are implemented using registers.

MAJC

The Sun Microsystems MAJC architecture [69] exploits the parallelism in multiple levels: instruction, data, thread and process, through vertical and speculative multithreading, chip multiprocessing and VLIW. Other VLIW processors aimed at DSP were developed, like Viper [70], Fujitsu FR500 [71] and Texas TMS320C6x [72].

Cjip

The Cjip [73] is a CISC/WISC processor for embedded applications featuring native Java byte code execution as well as Assembler/C/C++ support. Java byte code is to a very large extent implemented directly in microcode, providing native Java execution speeds. The Cjip uses 72 bit wide microcode instructions, providing a high degree of parallelism to efficiently control all the processor's hardware logic. The microcode in the internal ROM and RAM controls the processor hardware logic and resources. The application program is in DRAM, external to the Cjip chip.

Moon

The Moon processor core, designed by Vulcan ASIC Ltd. [74, 75] is a stack-based, von Neumann architecture. This is a synthesizable core and implements Sun Microsystems' JVM, designed to run ROM-driven embedded system-on-chip (SOC) applications. The core has been optimized for the area through a unique partitioning of Java bytecode for direct, microcoded and external execution allowing it to easily fit into CPLD devices for both production and prototyping solutions

Lightfoot

The Lightfoot 32-bit core [76], is a hybrid 8/32-bit stack-based processor executing Java bytecodes in the hardware. It is 3-stage pipelined Harvard architecture. The instruction memory is 8 bits wide and data memory is 32 bits wide. It has an integer ALU, a barrel shifter, and a 2-bit multiply step unit. The processor architecture supports three different instruction formats: soft bytecodes, non-returnable instructions, and single-byte instructions that can be folded with a return instruction. Special instructions are provided for supporting the complex JVM bytecodes. This group includes instructions for creating stack frames. There are two different stacks, data stack and return stack, with the top elements implemented as registers and memory extension. The data stack is used to hold temporary data and the return stack holds return addresses for subroutines and can also be used as an auxiliary stack. The core is available in VHDL and can be implemented in less than 30K gates. Lightfoot is now part of the VS2000 Typhoon Family Microcontroller [77].

LavaCORE

LavaCORE [78] is a 32-bit configurable processor core developed by Xilinx AllianceCORE, partner of Derivation Systems, Inc.; and is targeted at Xilinx FPGA architectures. The LavaCORE processor has 32-bit address and data buses designed to add optional modules including local memory, a floating point unit, DES encryption engine, and garbage collector. The processor core consists of an integer unit, programmable timers, register file, and interrupt controller. Configuration options allow cache sizes, and 32, 16, or 8-bit data widths. It also allows selecting Java bytecode instructions which are required to be omitted or moved from hardware to software. The processor also incorporates three 8-bit instruction registers, an instruction pre-fetch buffer to reduce memory access, and a 32-bit ALU to compute integer and logical operations.

jHISC

jHISC [79] is a RISC based processor with some Object-Oriented (OO) feature enhancements. It also provides object manipulation instructions to handle the related operations. Excluding 64-bit operation instructions, Java bytecodes are fully supported, with 91% of bytecodes and 75% of OO related bytecodes implemented in hardware directly. The other performance sensitive bytecodes not implemented in hardware are executed through software traps.

Azul

Azul Systems provides an impressive multiprocessor system for transactions oriented server workloads [80]. A single Vega chip contains 54 64-bit RISC cores, optimized for the execution of Java programs. Up to 16 Vega processors can be combined to a cache coherent multiprocessor system with 864 processors cores, supporting up to 768 GB of shared memory.

JOP

JOP [81]is a hardware implementation of the JVM targeted for small real-time embedded systems where instruction set of JVM (bytecodes) becomes the native instructions of the processor. It should be noted that the JOP does not implement all the bytecodes in hardware as some of the bytecodes are too complex to be implemented in the hardware e.g., *new*. Therefore, their functionalities are emulated using the Java. In fact, JOP has a single native instruction set, the so-called microcode. During execution, every Java bytecode is translated to either one, or a sequence of microcode instructions. This translation merely adds one pipeline stage to the core processor and results in no execution overheads. The JOP is implemented as a small soft core that fits in an FPGA. It executes Java bytecodes much faster without JIT-Compiler [50]. As JOP is used as basis for research, it is discussed in more details in Section 3.4.

SHAP

The embedded Java multi-core architecture is based on JOP and enhances it with a hardware object manager. SHAP also implements the method cache [82]. The access to the shared heap is provided through a full duplex bus with pipelined transactions. Each core is equipped with local on-chip memory for the Java operand stack (8 KB) and the method cache (2 KB) to further reduce the memory bandwidth requirements. It supports synchronization on a per object basis.

3.4 Java Optimized Processor-JOP

Numerous important Java processors from both industry and academia were mentioned in the previous section, but most of them are either not available or do not provide enough tool support to be used as basis for the ongoing research. Of these, it was found that JOP is the most suitable as it is available as the open source, technology independent and most importantly easy customizable making it perfectly suitable for use in our research. The most significant and appreciable features of the JOP are:

- Availability as open source
- Extendibility
- Time predictability

The open source nature of JOP makes it an attractive choice for the research. The extendibility is essential to support the convergence of multiple functionalities to the existing core. In a lot of cases fixed hardware blocks are inadequate to provide required performance because of their lack of flexibility, reusability and ability to deal with multiple modes and standards. For a lot of specific tasks standard processors have challenges of their own in terms of meeting the performance and power consumption requirements. The extendibility allows to add special registers, specialized execution units that efficiently perform task-specific algorithms and customized system specific I/Os that can connect directly to neighboring blocks of dedicated hardware. This design approach does not affect clock rates and keeps energy consumption low.

3.4.1 JOP Overview

The JOP architecture consists of the processor core, a memory interface and a number of IO peripherals as shown in Figure 3.4. The processor core interacts with the memory through a memory interface. The processor core contains the three pipeline stages i.e., microcode fetch, decode and execute and an additional translation stage bytecode fetch. The JOP has a stack cache as a substitution for the data cache and a method cache to cache the instructions. The method cache [83] is organized such that it caches entire Java method. A cache fill from main memory is only performed on a miss on method invocation or return. Therefore, all other bytecodes have a guaranteed cache hit. The default configuration is 4 KB, divided into 16 blocks of 256 bytes. The memory interface provides a connection between the main memory and the processor core. The request for a method to be placed in the cache is performed through the extension module, but the cache hit detection and load is performed by the memory interface independently from the processor core (and therefore concurrently).

The IO interface contains peripheral devices, such as the system time and timer interrupt, a serial interface and application specific devices. JOP uses a simple and efficient system-on-chip interconnection called SimpCon [84] to connect the memory controller and peripheral devices to the processor pipeline. SimpCon is a fully synchronous standard for on-chip interconnections. It is a point-to-point connection between a master and a slave device.

3.4.2 Instruction Set

A Java program is translated to bytecode instructions upon compilation. These bytecode instructions are executed by the JVM which does not assume any particular implementation technology. Java processors usually do not execute Java bytecodes directly, because some instructions are too complex to be implemented in hardware. Therefore, JOP translates the bytecodes into its own RISC based instruction set called microcodes. The JOP microcodes are 10-bits long, opcode is 8-bit and two extra bits (labeled as *opd* & *nxt*) are added for fetching the immediate operands embedded in microcode and indicating the end of microcode sequence for the current bytecode so that new Java bytecode could be fetched.

The instruction set contains different types of microcodes [50]. The bytecode equivalent microcodes are direct implementations of bytecodes and result in one cycle execution time for the bytecode (except *st* and *ld*). They are: *pop*, *and*, *or*, *xor*, *add*, *sub*, *st<n>*, *st*, *ushr*, *shl*, *shr*, *nop*, *ld<n>*, *ld*, and *dup*. The local memory access microcodes are used to access the first 32 locations of the internal stack RAM which contains the internal variables and the constants. The microcodes are: *stm*, *stmi*, *ldm*, *ldmi* and *ldi*. The register manipulation microcodes are used to manipulate the contents of the registers in the core. The contents of the stack pointer register, the variable pointer register and the Java program counter registers are accessed and modified by

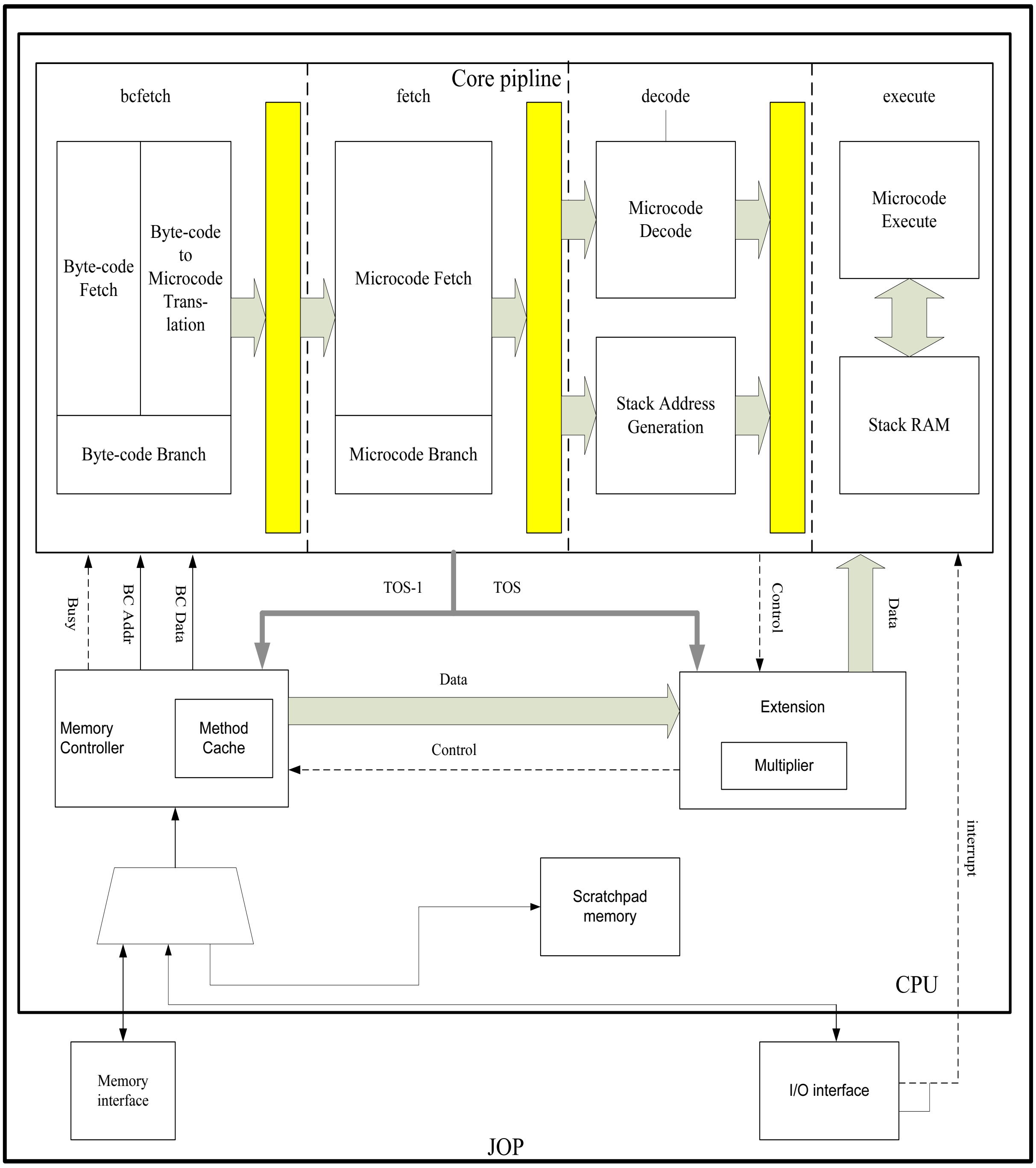


Figure 3.4: JOP architecture

using *stvp*, *stjpc*, *stsp*, *ldvp*, *ldjpc*, *ldsp* and *star* microcodes. The bytecode operand microcodes are used to load operands from the bytecode RAM. They are converted to a 32-bit word and pushed on the stack by using the *ld_opd_8s*, *ld_ opd_8u*, *ld_opd_16s* and *ld_opd_16u* microcodes. The external memory access instructions such as: *stmra*, *stmwa*, *stmwd*, wait *ldmrd*, *stbcrd*, *ldbcstart*, *stald*, *stast*, *stgf*, *stpf* and *stcp* are used to access memory subsystem and the IO subsystem. The *mul* microcode accesses the optional hardware multiplier. Two branch microcodes provide the conditional jumping using *bz* and *bnz*. The branch bytecodes are mapped to one microcode: *jbr*. The complete microcode set along-with the operations performed by it is given in [50].

3.4.3 Translation of Bytecodes to Microcode

The JOP translates the CISC JVM bytecodes into RISC stack based sequence of microcodes. The translation of bytecodes to microcodes takes exactly one cycle and microcodes are executed in 3 stage pipeline. The next bytecode is fetched only when the current bytecode has finished its execution. No time dependencies between bytecodes result in a simple processor model for the low-level WCET analysis [85]. The mapping between the Java bytecode and the JOP microcode is done using a translation-table generated during application building. Each bytecode acts as an address for the jump-table and the corresponding location contains the start address of microcode sequence for that bytecode. This address is loaded into the JOP program counter for every bytecode executed. Every bytecode is translated to an address in the microcode ROM that implements the JVM. If there exists an equivalent JOP microcodes for the bytecode, it is executed in one cycle and the next bytecode is translated. For more complex bytecodes, JOP just continues to execute microcode in the subsequent cycles. The end of this sequence is coded in the microcode (as the *nxt* bit) as shown in Figure 3.5. Interrupts are implemented as special bytecodes. These bytecodes are inserted by the hardware in the Java instruction stream in the translation stage as special bytecodes and are transparent. Interrupts and exceptions are handled by redirection of the microcode address to the handler code.

The JOP is not a pure stack machine. Method parameters and local variables are defined as locals. These locals can reside in a stack frame of the method and are accessed with an offset relative to the start of this local area. Additional local variables (16) are available at the microcode level. These variables serve as scratch variables, like registers in a conventional CPU. However, arithmetic and logic operations are performed on the stack. For optimum use of the available memory resources, all microcode opcodes are 8 bits long. There are no variable-length microcode opcodes and every microcode, with the exception of *wait*, is executed in a single cycle.

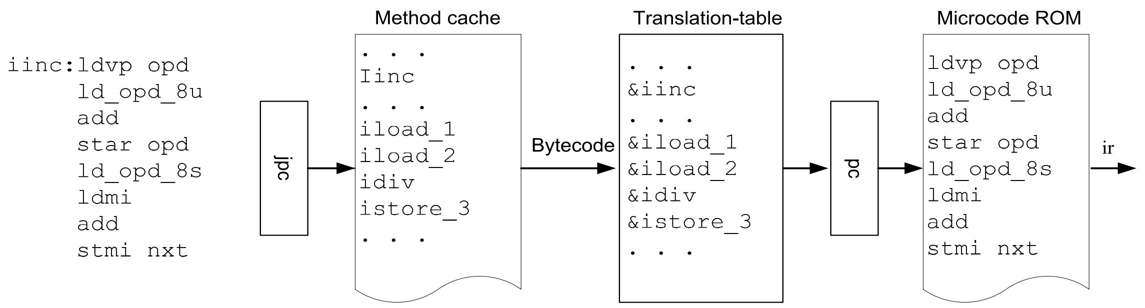


Figure 3.5: Bytecode to microcode mapping example and microcode fetching mechanism

3.4.4 Memory Organization

The JOP consists of a number of physical memories to store different types of data as discussed below.

Main Memory

The largest memory is the main memory where the application program is stored and remaining space is allocated to the heap for object storage. The JOP is connected to off-chip main memory via SimpCon [86] interface.

Stack Cache

In a JVM, stack is a heavily accessed memory region and it is placed in the upper level of the memory hierarchy to provide good performance and is referred to as stack cache. The stack cache is used both as an operand stack and as the storage place for local variables placed deeper inside a stack. The stack cache [87] of JOP is organized at two levels: first level as two discrete registers for the top two elements of the stack; second level as on-chip memory with one read and one write port. With the two top elements of the stack as discrete registers, these values are read, operated on and written back in the same cycle. Read and write access to a local variable is also performed in the same pipeline stage. The top element of the stack (TOS) resides in register named as *A* and next to top element (TOS-1) resides in the register named as *B* in Figure 3.6. Whenever an ALU operation is performed, the operands are *A* and *B* which are the top two elements of the stack. The result of the operation is stored in *A* and the next element in the stack is read and stored in the *B*. Similarly, when we store a local variable on the stack, the contents of *B* is transferred to *A* and, *B* is written with the contents of next element in the stack. When we push some value on the stack, the new value is stored in *A* and old value of the *A* of the is written to *B*, and old value of the *B* is written to the stack.

Method Cache

The JOP has a novel instruction cache [83], called method cache, which holds one complete method at any given time. This is possible because typical Java programs comprise of short methods with no branches out of the method and all branches inside are relative. Hence it is possible to load bytecodes of a complete method into the cache from *main* memory on method invocation and the return. Only filling the cache on method invocation and return simplifies WCET analysis [85] and removes another source of uncertainty, as there is no competition for the main memory access between instruction cache and data cache. In the method cache, several cache blocks (similar to cache lines) are used for a method. The main difference from a conventional cache is that the blocks for a method are all loaded at once and need to be consecutive. The time needed to load a complete method is calculated using the memory properties (latency and bandwidth) and the length of the method. On method invoke, the length of the invoked method is used, and on a return, the method length of the caller is used to calculate the load time. The full loaded method and relative addressing inside a method also result in a simpler cache. Tagged memory and address translation are not necessary.

Microcode ROM

The microcode is the native instruction of the JOP. Microcode ROM is used to hold the microcode sequences for each bytecode and has a single cycle access latency. Apart from the microcode sequence for the bytecodes, the microcode ROM also contains microcode sequence for loading the Java application program and booting the JOP.

3.4.5 JOP Datapath

The JOP datapath shown in Figure 3.6, has four pipeline stages: Bytecode Fetch, Fetch, Decode and Execution (or Stack). In the first stage, the Java bytecodes are fetched from the internal RAM, which serves as the instruction cache, from the address stored in *jpc* register. The bytecode is mapped through the translation-table into the address (*jpaddr*) for the microcode ROM. The fetched bytecode results in an absolute jump in the microcode (the second stage). The microcode ROM address provided by the translation-table is stored in a register, named *pc*. The bytecode is also stored in a register for later use as an operand (requested by signal *opd*) as shown in Figure 3.6. If the bytecode is a complex one having more than one microcodes, JOP continues to execute those microcodes. At the end of this microcode sequence, the next bytecode, and therefore the new jump address, is requested (signal *nxt*).

The second pipeline stage fetches JOP microcode from the internal microcode memory and executes microcode branches. The program counter *pc* is incremented during normal execution.

If the microcode is labeled with *nxt* a new bytecode is requested from the first stage and *pc* is loaded with *jpaddr* which is the starting address for the implementation of that bytecode. The label *nxt* is the flag that marks the end of the microcode instruction stream for one bytecode. Another flag, *opd*, indicates that a bytecode operand needs to be fetched in the first pipeline stage. Both flags are stored in a table that is indexed by the program counter.

In the decode stage, microcode is decoded and control signals are generated. The address for the stack RAM is also generated in the same stage. When an address relative to the stack pointer is used (either as read or as write address, never for both) the stack pointer is also decremented or incremented in the decode stage. Stack machine instructions can be categorized from a stack manipulation perspective as either *pop* or *push*. This allows us to generate fill or spill TOS-1 addresses for the following instruction during the decode stage

In the execution stage, arithmetic or logical operations are performed. The operands for the arithmetic/logical operations are the contents of registers *A* and *B* and the result is stored back in register *A*. All load operations (local variables, internal register, external memory and periphery) result in a value being loaded into register A without requiring a write-back pipeline stage. Only the data in the register *A* is stored into the memory during a store operation. Register *B* is never accessed directly. It is read as an implicit operand or used for stack spill on push instructions. It is written during the stack spill with the content of the stack RAM or in the case of stack fill with the contents of register *A*.

3.4.6 Boot Up

The application start-up process involves configuring an FPGA with an image containing JOP processor and downloading the application. The FPGA can be configured via a download cable (with JTAG commands) performed within the IDEs from Altera and Xilinx or with command line tools such as *quartus_pgm* or *jbi_32*. For automatic boot-up on power-up, the configuration can be stored in nonvolatile flash memory. When the configuration has finished, an internal reset signal is generated. After reset, microcode instructions are executed starting from address *0*. At this stage, application program (Java bytecode) has not been loaded yet. The first sequence of microcodes in ROM performs this task. The Java application can be loaded from an external memory or Flash memory, via a PC serial line, or a USB-port. For VHDL simulation in ModelSim, the Java application is loaded by the test bench instead of JOP. In the next step, a minimal stack frame is generated and the special method *Startup.boot()* is invoked, even though some parts of the JVM are not yet setup. From now on JOP runs in Java mode. The method *boot( )* sends a greeting message, detects the size of the main memory, initializes the data structures for GC, initializes *java.lang.System*, invokes the static class initializers in a predefined order and finally invokes the *main* method of the application class.

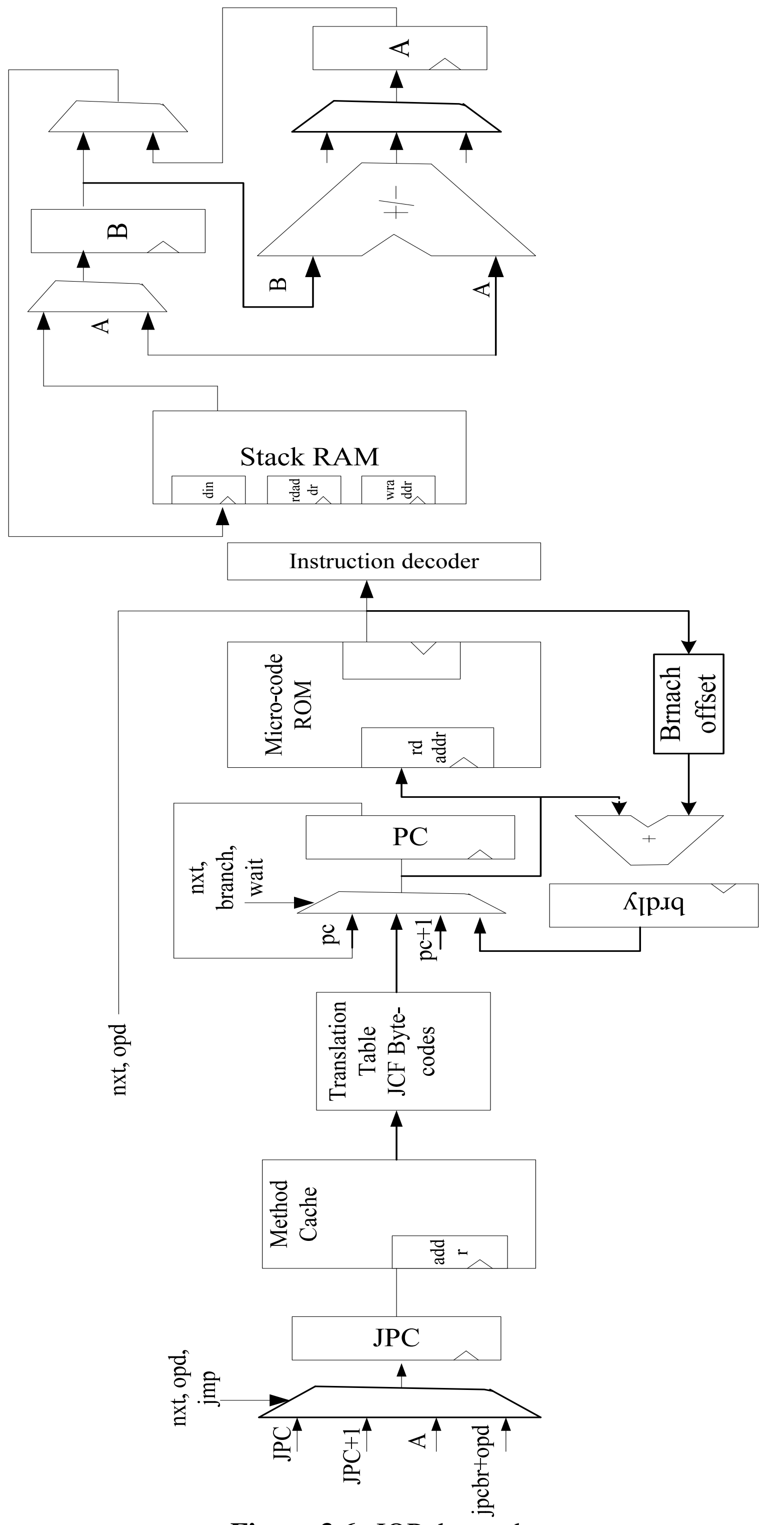


Figure 3.6: JOP datapath

3.4.7 Extending JOP

Java bytecode is the form of instructions that the JVM executes. Each bytecode opcode is one byte in length, there are possible 256 opcodes and not all of them are used. In fact, Sun Micro systems have set aside 3 values to be permanently unimplemented. The instruction set of the JVM contains 201 different instructions [49]. Of the remaining 52 bytecodes, around 20 opcodes have been used by JOP to implement specific functions. The remaining opcodes can be used to implement the further user-specific bytecodes. These bytecodes are used for data transfer among the existing and extended hardware components and perform arithmetic and logic operations.

The native language of JOP is microcode. The newer microcode could be introduced inside the *instruction.java* file in the tools. This is accompanied by VHDL file editing to modify the functionality to support the microcode. A native method is implemented in JOP microcode. The interface to this native method is through a special bytecode. The mapping between native methods and the special bytecode is performed by JOPizer, a JOP tool which converts the class file generated by the Java compiler into a format compatible with JOP. The real microcode is added in *jvm.asm* file (microcode implementation of JVM) against the label for the bytecode. The code in *Native.java* file provides a method signature for the *Native* method and the mapping between this signature and the name is provided in *jvm.asm* and in *JopInstr.java* files. The native method is accessed by the method provided in *Native.java*. The native method gets substituted by JOPizer with a special bytecode. The detailed information about the JOP related tools can be found in [50].

The JOP also allows the addition of new peripheral devices which involves some VHDL coding. All peripheral components in JOP are connected using the SimpCon interface. For a device that implements the Wishbone [88] bus, a SimpCon-Wishbone bridge is provided.

3.5 JOP as SystemJ Execution Platforms

Previously, SystemJ programs compiled to Java were executed on JVM running on the NIOS II processor. Executing SystemJ compiled on an interpreting JVM is slow, thus lacking the performance required to run the more complex applications. The fastest way to execute SystemJ programs compiled to Java bytecodes is by using a Java Processor such as JOP which brings the following benefits [50]:

- The hardware JVM shows much better performance and can be up to 500 times faster than the programs running on an interpreting JVM [50].

- JOP is the smallest hardware implementation of the JVM available to date. This fact enables low-cost FPGAs to be used in embedded systems. The resource usage of JOP can be configured to trade size against performance for different application domains.
- The JOP is capable of calculating the worst-case execution time (WCET) estimates of tasks which is crucial for designing real-time systems. This information also helps to schedule clock-domains using various strategies.

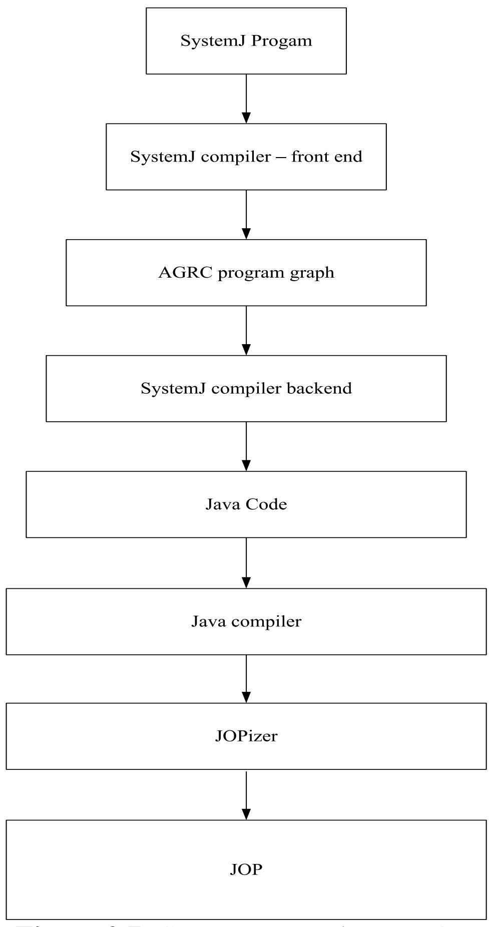


Figure 3.7: SystemJ execution on JOP

Figure 3.7 illustrates the design flow of SystemJ on JOP. The SystemJ program is transformed to an intermediate format called Asynchronous GRaph Code (AGRC) and the compiler back-end produces a single threaded Java code [8] where concurrency and reactivity is emulated in Java. The SystemJ programs also interact with the environment through signals which can be either Boolean or valued as mentioned earlier. The synchronous reactive constructs operate on these signals. All Java variables and *signals* declared in the SystemJ source are global variables in the generated Java source code. JOP interacts with the environment through the IOs provided.

3.6 JOP Performance Evaluation for SystemJ

We evaluate the performance of JOP for executing the SystemJ program by running benchmark examples on it.

3.6.1 Benchmarks

There is no standard benchmark suite available covering both the GALS and synchronous models with heavy data-computations. The benchmarks used here have been developed by [8] . Although we have large number of SystemJ benchmarks, we have selected only those which have been used in all other implementations. The benchmarks include both synchronous and asynchronous examples as shown in Table 3.1.

The *demoloop* (dl) example represents the synchronous MoC of SystemJ and has the following features:

- It has the two input signals and four output signals.
- The output signals are emitted in an orderly fashion when a particular input is present and abort if the other input signal is present.
- Hence, this example is control dominated with very little data-computations.
- It consists of a single clock-domain comprising of four reactions.

The *runner* benchmark gives the behavior description of a runner and has the following features:

- It belongs to the heterogeneous class of applications as not only it contains reactive statements, but also includes arithmetic and logical data handling.
- It consists of a single clock-domain and four reactions combined using || operator.

The synchronous examples were borrowed from the Esterel test-bench suite [89]. The asynchronous case is represented by asynchronous protocol stack (*aps*) taken from [90]. The *aps* example is discussed in detail in chapter 6. The features of all chosen benchmarks are given in Table 3.1. The benchmark example codes are given in Appendix D.

3.6.2 Hardware Platform

All presented data has been collected from the experiments carried out by using the cycle-accurate *ModelSim* simulator and executing them on *Altera* Cyclone II FPGA with 70k logic

Examples	**Lines of code**		**Total number of synchronous reactions**	**Number of CDs**
	SystemJ	**Java**		
dl	51	12	4	1
runner	91	10	4	1
cl	85	0	4	1
aps	154	31	6	2

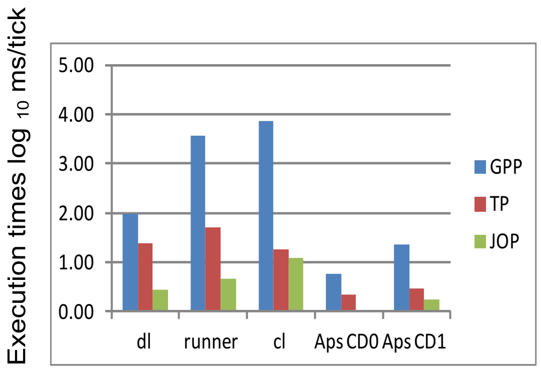
Table 3.1: Features of experiment set

elements, 2MB of RAM and running at 50 MHz clock. The system is capable of running at 100 MHz but the results presented are for 50 MHz clock for fair comparison with earlier published results.

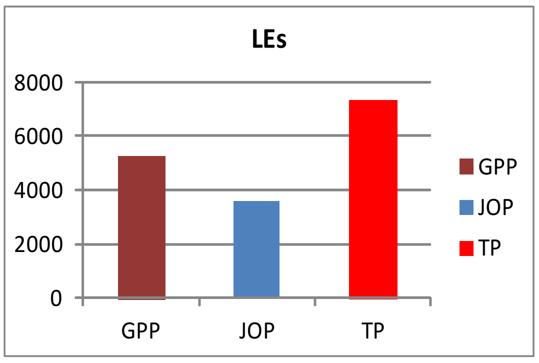
3.6.3 Execution Time Comparison

The execution speed comparisons are given in terms of the average execution time between two consecutive ticks for each clock-domain, which are indications of the throughput of the platforms. The average tick execution time is obtained by averaging one million ticks. Figure 3.8a shows the comparison of execution time in terms of average tick times for a Java Optimized Processor (JOP) against general purpose processor implementing a interpreting JVM against and the TP platform. The results are normalized to the smallest value and their logarithmic values are presented. The clock-domain aps *cd0* has smallest execution time which is normalized to 1 and its logarithmic value will be zero. An interpreting JVM is easy to implement but the execution speed suffers from interpreting and is much slower than the JOP as shown in Figure 3.13a. This clearly demonstrates the benefit of use of a Java processor as SystemJ execution platform when compared with GPP.

The JOP out-performs the TP and is upto 11 times faster than the TP despite the fact that the TP benefits from the compilation approach where control-oriented operations are separated from the data-oriented operations. The JOP performs better than the TP system due to two reasons. First, it is due to data-calls by CP to DP in a TP system as the TP system is loosely coupled and incurs large communication overhead. Second, all the data computations are performed by the data-processor (NIOS II in this case) which is a general-purpose processor and has a software interpretation of bytecodes which is much slower than the hardware implementation. Hence, JOP is expected to outperform the TP system when running applications involving data-dominated computations as the extensive data calls would slow down the system. In the case of TP, the scheduling of the reactions takes place in CP which is more efficient than the scheduling in single threaded Java code.



(a) Execution times comparison



(b) Resource usage comparison

Figure 3.8: JOP performance comparison

3.6.4 Resource Usage Comparison

Figure 3.8b shows the resource utilization for different target platforms in terms of the logic elements used when implemented on an Altera Cyclone II FPGA. The JOP uses 32% and 104% fewer resources when compared with GPP and TP, respectively. These results presented show the superiority of a Java processor over the interpreting JVM running on a general purpose processor and TP and uses fewer resources at the same time. This clearly suggests that it should be deployed as an execution platform for SystemJ applications compiled to Java.

3.7 Limitations of SystemJ Execution on JOP

Although using a Java processor results in higher performance, it also inherits some limitations. The JOP has a special kind of instruction cache, called method cache, as discussed in Section 3.4.4. The default configuration for this cache is 4KB, providing optimal performance. This puts a constraint on the SystemJ application compiled to Java for execution on it. The size of all Java method should be less than 4KB. This requires some minor modifications in compiler back-end to generate the Java code with method size of less than 4KB.

In JOP, the heap area resides inside the main memory located external to JOP which is usually slower and requires multiple cycles to access it. All signals, valued or non-valued, are stored in this heap. The signals in the heap are accessed in the following cases:

- When reading input signals from the environment at the end of the tick, the signals are read and stored in the heap.

- When sending the output signals to the environment, the signals are read from the heap and sent to the output port.
- The heap is also accessed when status of a local signal is set.
- The signals are also accessed to check the status of the input and control signals during the control flow of the program.

The compiler creates a Signal class that is used to hold signal values and statuses. The signal and channel classes provide methods which are called while communicating with the environment and updating channels statuses. The following table shows the methods of signal class which are used to manipulate the signals.

Java methods	**Description**
setStatus()	It sets the status of an input or a local signal to true
getStatus()	It reads the status of an input or a local signal
sethook()	It writes the output signals to the environment
gethook()	It reads the input signals from the environment
setClear()	It sets the status of an input or a local signal to false
tick();	It indicates the end of tick to environment

Table 3.2: Signal manipulation methods in single threaded Java code

Let us consider the example where we set the status of a signal S. This is done by using the Java statement *S.getStatus()* in the *main* method. As a result of execution of this statement, the method cache is filled with the bytecode for *getStatus()* method and object signal is accessed from the heap. When returning from the *getStatus()* method, the method cache is again loaded with bytecodes of main method from the main memory of the JOP. High access latency to external main memory and frequently loading of cache results in the degradation of performance. This cannot be afforded as reactive systems are bound to react at a pace determined by the environment.

The reactive programs constantly interact with the environment and often access the signals resulting the substantial Java code used in performing these operations. The frequent occurrence of these constructs, which are slow to execute, make them candidates for acceleration because by making the common case fast, significant improvement in performance can be achieved.

3.8 Reactive-JOP - RJOP

The use of JOP has resulted in higher performance and predictability of execution time but support for reactivity and control flow is missing. JOP lacks the architectural features to efficiently execute the signal manipulation and control constructs required to implement the reactivity and control flow in synchronous part of SystemJ. The RJOP [91] is a novel, high performance and low cost execution architecture based on a customizable processor core aimed at providing better support for concurrency and reactivity of SystemJ. It also maintains the time-predictable execution of the applications intended for real-time embedded systems and calculation of Worst Case Reaction Time as provided by the original core. The JOP, inherently suited to data-driven transformational operations, is extended to efficiently execute the reactive constructs in SystemJ. As such, the new core becomes suitable for both data-dominated and control-dominated applications in embedded system domain. The benchmark results show significant performance improvement and lower resource requirements over the existing architectures used for the SystemJ execution. It allows calculating the WCRT of synchronous clock-domains, which further can be used in efficient scheduling of full multiclock-domain GALS SystemJ programs.

3.8.1 Related Work

The idea of extending processor to support reactivity and control is not new and a number of instances exist where support for reactivity is incorporated into general purpose processors. The traditional general purpose microcontroller core is combined with a custom hardware block that extends the instruction set of the traditional microcontroller by certain new instructions to support reactivity. The existing processors such as RePIC [92], ReMIC [33] and ReFLIX [32] can handle reactivity, but they require the specific compiler for the target and porting of JVM to execute SystemJ. Doing so, often leads to a very poor performance of SystemJ applications. To avoid that, same technique could be applied to systems that have to offer both Java capability and high performance such as Java processor.

3.8.2 Reactivity and RJOP

The synchronous reactive constructs operate on signals which can have status only or both value and status. The placement of these signals in data structures is a critical issue. Currently, the signals are translated to objects of Signal Class and the status of the signals is set by calling methods of this class, incurring calling overheads which results in slow execution. The signals are stored in heap and any operation on the signals requires memory access which can incur multi-cycle latency depending on the hardware platform and memory used. Also, it does not perform ALU operations on the memory operands, so they need to be loaded into some temporary storage which increases the latency. In addition, access to the large memory blocks has always been expensive in terms of power consumption, thus should be avoided.

The reactive systems are characterized by the frequent interaction with the environment or with other clock-domains through signals. The frequent access to the signals requires that they should be cached locally. In the new proposed approach, the signal statuses are stored in an array of processor registers, called *Signal-File* (SF), for easy, symmetrical and less power hungry access. It is parameterized upto 256 signals. Similarly, if there exists any signal dependency in SystemJ reactions, the execution of current reaction is suspended and another reaction is scheduled. Once the signal dependency is resolved, the reaction is resumed from the same location where it left. This is done by using the signal locks. When compiled to Java, these locks are translated to array which require access to the main memory. They are also moved to the Signal-File as they need to be accessed quite frequently.

3.8.3 RJOP Architecture

The JOP architecture is extended by introducing new specific bytecodes to efficiently support the signal manipulation and control flow in SystemJ. Each bytecode opcode is one byte in length. The instruction set of the JVM contains 201 different instructions [9]. Few additional opcodes have been used by the JOP to implement specific functions and the remaining can be used to implement the desired custom bytecodes. The introduction of the new bytecodes requires modification of *javainstr.java* and *Native.java* as mentioned earlier in Section 3.4.7. Table 3.3 lists the new bytecodes introduced in the RJOP and their description.

Bytecode	**Task**
jopsys_emit	Asserts the signal in a Signal-File
jopsys_demit	Clear a signal in Signal-File
jopsys_initsf	Initializes the Signal-File contents if required
jopsys_present	Checks the presence of the signal
jopsys_lsip	Loads input signal from port to Signal-File
jopsys_lsop	Loads output Signal from Signal-File to port
jopsys_seot	Sets the register indicating end of logical tick
jopsys_ceot	Clears the end of logical tick register
jopsys_cer	Clears environment ready signal

Table 3.3: List of extended bytecodes specific to RJOP

The existing set of microcodes is also extended with new microcodes to support these bytecodes as shown in Table 3.4. These microcodes are cost-effective, because they share the resources of the existing processor. They are used for data transfers among the existing and extended hardware components and perform arithmetic and logic operations. Here *T1* and *T2* are two temporary registers introduced as an alternative to registers *A* and *B* in JOP data-path which hold top two elements of the stack.

Microcode	**Task**	**Register Transfer**
readsf	Loads the contents of SF into register T1	$SF[A] \rightarrow T1, B \rightarrow A, stack[sp] \rightarrow B, sp-1 \rightarrow sp$
loadormask	Loads the "or" mask into T2	$ormask \rightarrow T2$
reor	ORing of operands from T1 & T2	$T1 \vert\vert T2 \rightarrow T1$
writesf	Write the contents of T1 into Signal-File	$T1 \rightarrow SF[A], B \rightarrow A, stack[sp] \rightarrow B, sp-1 \rightarrow sp$
loadandmask	Loads the "and" mask into T2	$andmask \rightarrow T2$
reand	ANDing of operands from T1 & T2	$T1\&\&T2 \rightarrow T1$
loadopd	Loads contents from A into T1and T1into T2	$A \rightarrow T1, T1 \rightarrow T2, B \rightarrow A, stack[sp] \rightarrow B, sp-1 \rightarrow sp$
pushonstack	Load from T1 to top of stack	$T1 \rightarrow A, A \rightarrow B, B \rightarrow stack[sp+1], sp+1 \rightarrow sp$
popfromstack	Store top of stack in T1	$A \rightarrow T1, B \rightarrow A, stack[sp] \rightarrow B, sp-1 \rightarrow sp$

Table 3.4: Register transfer description of RJOP specific microcodes

Figure 3.9 shows the mapping of a new bytecode to the microcodes. The method *Native.emit* is replaced by the *jopsys_emit* bytecode which is mapped to a sequence of six microcodes and the last one containing the *nxt* field indicates the fetching of a new bytecode. This bytecode is used to set the status of a signal to true. The *loadopd* microcode loads an operand in the register. The operand contains the signal location comprising of register address in the Signal-File and the bit number which corresponds to the signal emitted. The *readsf* reads the Signal-File into register T1. The *loadormask* loads the mask into T2 to set bit representing the emitted signal and OR operation is performed on the operands in T1 and T2 by *reor* microcode. The result is stored back to Signal-File by *writesf* microcode with the bit corresponding to signal set to high.

```
jopsys_emit:
        loadopd
        readsf
        loadormask
        reor
        writetesf nxt
```

Figure 3.9: Mapping of *emit* bytecode to microcode

The JOP is extended with a custom hardware unit, called Reactive Unit to support new microcodes as shown in Figure 3.10. The decoder logic is modified to generate appropriate control signals for new microcodes and the ALU in the Stack unit is used to perform arithmetic and logic operations. This extension allows the RJOP to efficiently deal with the reactive applications along with the data-dominated computationally intensive applications. The bytecode fetch and microcode branch units are modified to provide branching for the new unit.

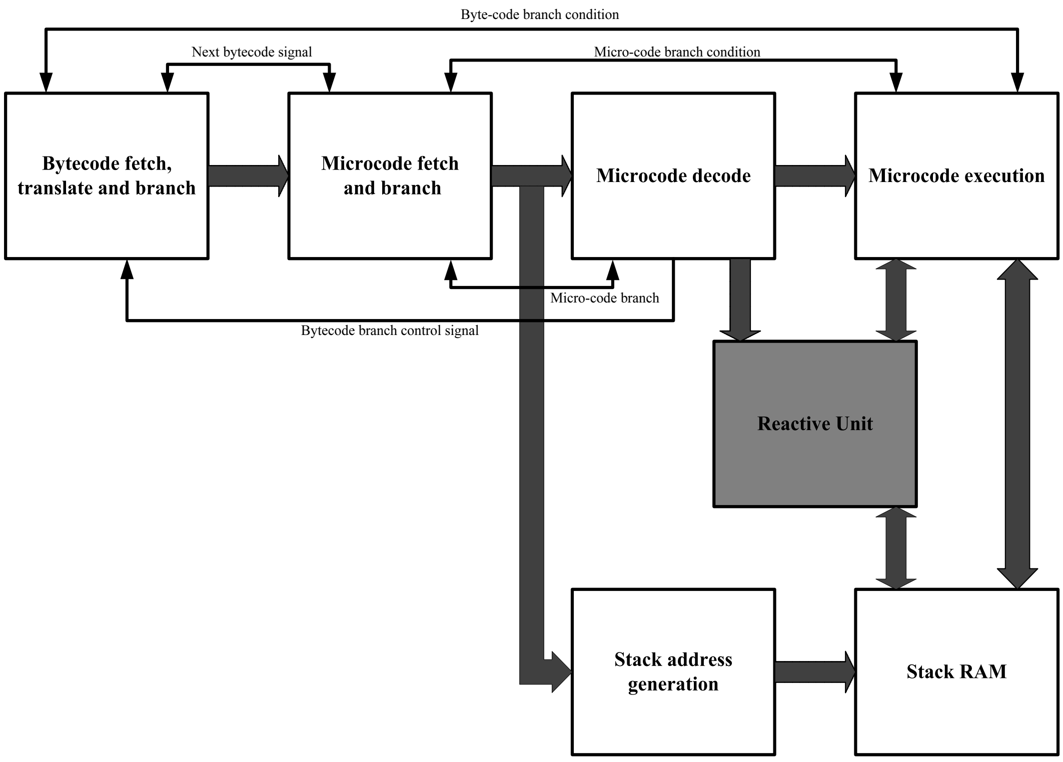

Byte-code branch condition
Next bytecode signal
Micro-code branch condition
Bytecode fetch, translate and branch
Microcode fetch and branch
Microcode decode
Microcode execution
Micro-code branch
Bytecode branch control signal
Reactive Unit
Stack address generation
Stack RAM

Figure 3.10: RJOP architecture with reactive unit shown as Grey

3.8.4 Compiler Modifications

The SystemJ compiler back-end is modified to generate the Java code compatible for execution on RJOP. The modified compiler, still generating single threaded Java code, translates the pure SystemJ signals into simple Java variables instead of objects of a class and they are assigned a unique identity code. The signal emission and signal status checking statements, which resulted in Java method calls in earlier version of compiler, are translated into Java methods with the *Native* prefix recognizable by the JOP tool and are subsequently replaced by custom bytecodes. All of the control flow constructs are translated into *if* - *else* statements which check the presence (true status) of the signal or the signal expression. The signal checks are carried out on these signal variables without invoking the methods of another class. Communication with the environment is provided through memory mapped IO. This includes loading input signals from the environment, emitting output signals and indicating end of tick etc. In the current implementation, acceleration of scheduling and asynchronous communication is omitted and is carried out in the existing way. The translation of SystemJ reactive constructs into Java statements using the original and the modified compiler is shown in Table 3.5.

SystemJ Constructs	Standard Java	RJOP Java
system {}	*public class system {}*	*public class system {}*
signal S	*Signal S = new Signal();*	*int S;*
emit S	*S.setStatus()*	*Native.emit(S);*
present S	*if(S.getStatus()) term(p) else;*	*Var=Native.present(S);*
abort S	*if(S.getStatus()) term(p) else;*	*Var= Native.present(S);*
suspend S	*if(S.getStatus()) term(p) else;*	*Var= Native.present(S);*
Reaction p(:){term(q)}	*public static void p(){}*	*public static voidp(){}*
synchronous/concurrency	*Switch{case...}*	*Switch{case...}*
environment	*S.sethook();*	*Native.cer(id,0);* *Native.lsop(id)*
environment	*S.gethook();*	*Native.lsip(id);*
resetting	*S.setClear()*	*Native.demit(S);*
logical tick	*tick();*	*Native.seot(id,1);* *Native.ceot(id,0);*

Table 3.5: Translation of SystemJ reactive constructs into Java statements

Figure 3.11 shows the example of translation of SystemJ code to the standard Java source and the Java source compatible with the RJOP. A small SystemJ code example is given which checks for the input signal *A* and if it is present then it emits signal *B*, otherwise *C* is emitted. The equivalent Java code is given next where *input* signals are declared as objects of the *Signal* class. The checking and setting of signals is done by calling the methods of the class. The modified Java code for the RJOP is provided on the right where signals are declared as *integers* and the setting and checking of the signals is performed by the respective newly introduced bytecodes.

```
// SystemJ Code
system{
 interface{
 input Signal  A;
 output Signal B,C;
 }
{
present (A)
  emit B;
else
  emit C;
...
```

```
// Java source for JOP
import  Signal;
public class ... {
private static Signal A;
private static Signal B;
private static Signal C;
...
static void main(){
  A = new signal();
  ...
  if (A.getStatus()){
   B.setStatus();
  }
  else {locks[]=1;}
...
// starting new tick
  A.setClrear();
...
```

```
// Java source for RJOP
Import com.jopdesign.sys.*;
public class  ... {
private static int A=0;
private static int B=1;
private static int C=2;
...
static void main(){
int flag = Native.present(A)
  if (flag > 0){
    Native.emit(B);
  }
  else {
    Native.emit(lockid);
  }
...
// starting new tick
   Native.demit(A);
...
```

Figure 3.11: SystemJ program compiled targeting JOP and RJOP

3.8.5 Reactive Unit

Figure 3.12 shows the internal architecture of the Reactive Unit. It is composed of the Signal-File to store signal statuses, temporary registers to hold intermediate values (also act as an alternative for stack registers *A* and *B*) and a mask generation unit to mask or unmask a particular bit of a signal word. The masks are used for the setting or resetting the status of a signal. The Reactive Unit is interfaced with decoder, stack and internal stack cache. All of the signals handling bytecodes modify the status of the signals. The signal *id* is passed as a parameter in the *Native* method which is replaced by the corresponding bytecode. The arguments are available in the stack from where they are read into register *A*. The least significant byte is loaded from *A* into T1 by the microcode *loadopd*. The *lsb 5* bits give the byte address whereas msb 3-bits indicate the location of the signal inside the byte and are also used to generate the mask for that signal. The signal word and mask are loaded into temporary registers T1 and T2 by *readsf* and *loadormask* microcodes respectively. The contents of both registers are *ORed* by *reor* microcode and the result is stored back into T1. The contents of T1 are written back to the Signal-File by the microcode *writesf*. The destination address is available in the register A which holds the top element of the stack.

In the case of the control flow instructions, the signal address is provided, and the signal statuses are loaded into T1. The *andmask* is loaded into T2 which sets all other bits of T1 to zero and status of the required signal is extracted. T1 is then checked for the presence of the signal. If T1 is not zero, the signal is present, else it is absent. The communication with the environment is carried out using both the existing and new microcodes. The data to be sent is pushed onto the stack using *pushonstack* from where it is stored in the data port register. The address is passed as a parameter and loaded into the port address register. Similarly, when reading the statuses of the input signals at the end of tick, the address is provided and the statuses are loaded into the register T1. The statuses are stored at the required location into the Signal-File in the same way as mentioned above. The details of RJOP specific bytecodes along with register transfer descriptions are provided in the Appendix C.

3.9 Performance Evaluation

In order to evaluate the RJOP performance, we use the same benchmark examples as used for JOP in Section 3.6.1. All presented data has been collected from the experiments carried out by using the cycle-accurate *ModelSim* simulator and executing them on *Altera* Cyclone II FPGA with 70k logic elements, 2MB of RAM and running at 50 MHz clock. The system is capable of running at 100 MHz but the results presented are for 50 MHz clock for fair comparison with earlier published results.

environment

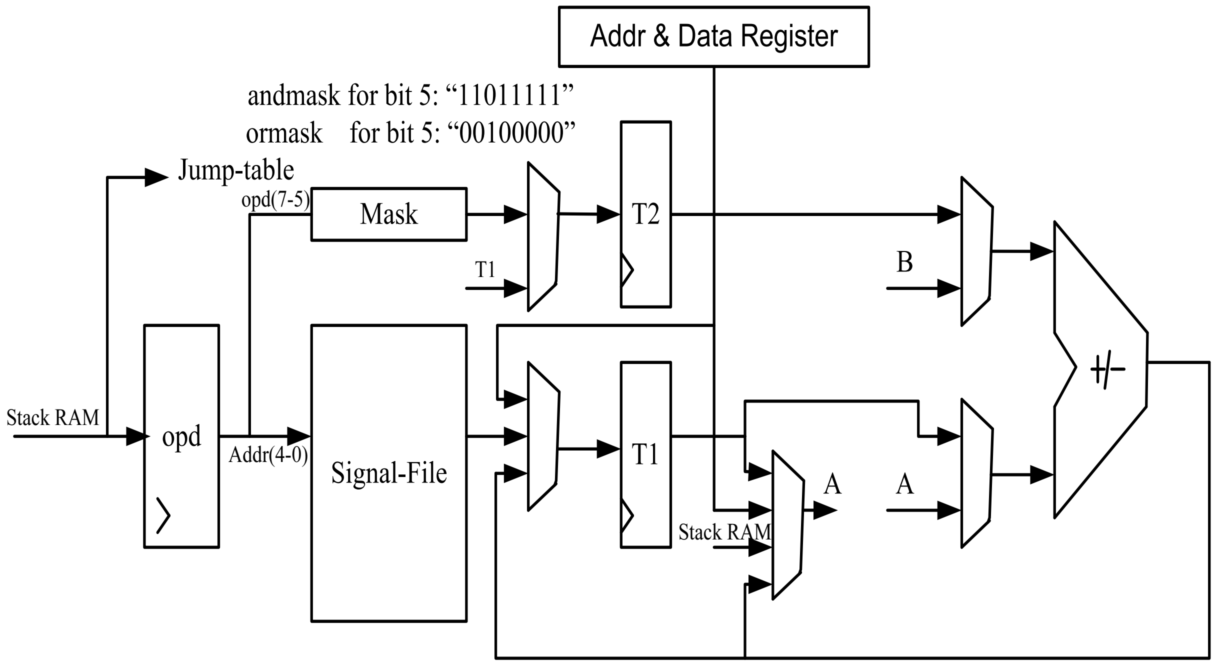
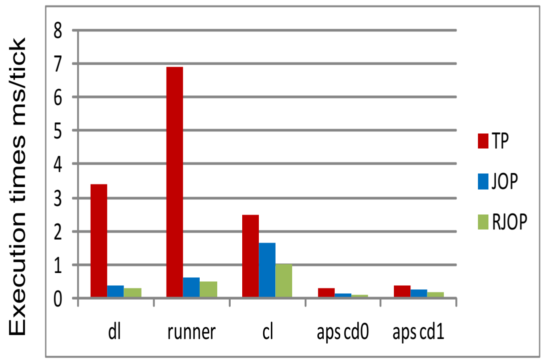


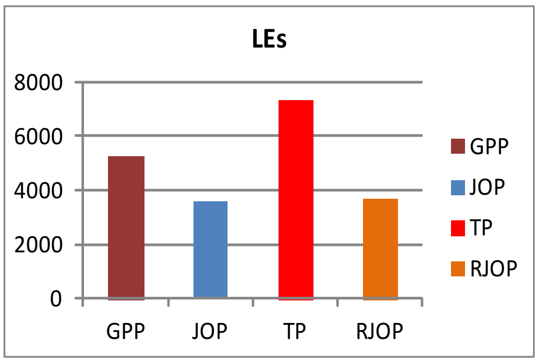
Figure 3.12: Detailed architecture of the reactive unit of RJOP

The performance of RJOP [91] is compared against the JOP. The results in previous section showed that the JOP is a better execution platform for Java-based data-dominated computationally intensive applications in embedded systems. But, the SystemJ programs involving extensive signal emission and signal check statements tend to execute slower. The RJOP incorporates the capability to handle both the control flow and signal manipulation. The results in Figure 3.13a show that RJOP on average is 20% faster than JOP. The RJOP will perform even better if the application becomes more control-oriented. Hence, RJOP is guaranteed to provide better performance than JOP regardless of whether the application is control-dominated or data-dominated, and supports our idea of providing the hardware support for the reactive constructs of SystemJ language to harness its potential performance. The RJOP uses slightly less than 3% resources (in terms of logic elements) when compared with JOP. The resource usage is found out by synthesizing the VHDL code usgin Quartus tool for target FPGA mentioned above.

New RJOP bytecodes that support reactivity all have bounded execution times similar to JOP, therefore the program worst case execution times (WCET) can be found for RJOP. However, as RJOP executes SystemJ programs and reactions in lock-step with logical tick, it allows calculating the worst case reaction time (WCRT), which is the maximum time between any two consecutive ticks in any SystemJ clock-domain, which at the same time represents the minimum allowed time between two consecutive events from the environment.



(a) Execution times comparison



(b) Resource usage comparison in terms of Logic Elements (LE)

Figure 3.13: JOP performance comparison

3.10 Summary

Java is a unique combination of the language definition, a rich class library and a runtime environment. A Java program is compiled to bytecodes that are executed by a Java virtual machine. The intermediate bytecode representation simplifies porting of Java to different computer systems. The Java Virtual Machine (JVM) is a definition of an abstract computing machine that executes bytecode programs. An interpreting JVM is easy to implement and needs few system resources. However, the execution speed suffers from interpreting. A Java processor avoids the slow execution model of an interpreting JVM and the memory requirements of a compiler, thus making it an interesting execution system for Java in embedded systems. A Java processor is the implementation of the JVM as a concrete machine. JOP [81] is a hardware implementation of the JVM targeted for small real-time embedded systems where instruction set of JVM (bytecodes) becomes the native instructions of the processor. The most significant and appreciable features of the JOP are: availability as open source, extendibility and time predictability. The JOP is used as a target platform for executing SystemJ programs compiled to Java. The performance results suggest that the JOP is a suitable platform for execution of SystemJ programs. JOP lacks the architectural features to efficiently execute the signal manipulation and control constructs required to implement the reactivity in synchronous part of SystemJ. The JOP, inherently suited to data-driven transformational operations, is extended to efficiently execute the reactivity constructs in SystemJ. The benchmark results validated the idea of incorporating reactivity to achieve the performance improvement.

Chapter 4

GALS-JOP

The aim of this research is to explore and develop an efficient embedded platform for SystmJ program execution capable of carrying out work in minimal time with minimal resource usage. A SystemJ program is translated to either pure Java or a mix of Java and special instructions where control-driven operations are separated from data-driven operations in the form of concurrent reactive control flow (CRCF) and Java control flow (JCF) as demonstrated in [38]. The code generated as a result of split compilation approach is executed on TP architecture, where both CRCF and JCF are executed on two separate processors concurrently. The existing TP platform [39], briefly discussed in Chapter 2, shows the advantage of split compilation strategy adopted over pure Java compilation approach. The TP architecture deploys a custom control processor (ReCOP - REactivity and Concurrency Processor) to handle the CRCF in a better way but JCF suffers from poor execution due to the use of a general purpose processor (interpreting JVM) as discussed in Chapter 2. The long interpretation time is an important drawback, at least several times longer than in case of the program native execution. It is also hard to estimate execution cycle time due to complex nature of underlying micro-architecture components. One can greatly reduce execution time of the code by replacing software-implemented virtual machine with its hardware equivalent. The JOP processor introduced in Chapter 3 is a suitable hardware platform for executing the programs described in Java [93]. This promptly led us to replace general purpose processor with JOP in TP approach to efficiently handle the JCF. The resulting architecture will not only benefit from the split compilation strategy and but also efficient hardware platform. This, being a two processor approach, uses too many logic resources. This problem is solved by introducing a novel approach which efficiently executes both CRCF and JCF on a single processor and also brings economy in terms of resource usage.

This chapter starts with the discussion on the deficiencies of existing approaches and highlights the problem areas. Next, we present a solution in the form of TP-JOP and its design flow, architecture and expected performance outcomes. Then, we present a uniprocessor solution in the form of GALS-JOP processor for efficient execution of SystemJ programs. We introduce the global aspects of the proposed approach, which positions the presented work and its contributions. The new processor is described and qualitative comparisons with related processors and execution models are presented. The details about compilation strategy, tools developed, architecture, implementation details and performance evaluation are also discussed in the later part of the chapter.

4.1 Introduction

One of the key goal in embedded systems design is efficient exploitation and implementation of concurrency as the way of reducing design complexity of the system and at the same time ability to deal with the requirements of timely response to the events coming from external environment. Significant research efforts have been made in tailoring Java and its execution environment to facilitate its use in embedded systems [94]. However, dealing with concurrency is still left to the inefficient Java thread model. Java threads, besides low efficiency, are relatively unsafe model that demands programmers to deal with low-level details of thread synchronization and communication resulting in programs with very little guarantees on worst case execution time. Also, Java memory model makes it difficult to implement it on simple processors.

4.2 Deficiencies in the Existing Approaches

In its original implementation, SystemJ was compiled to Java and then executed on a processor that has a JVM or some variation of it, such as J2ME. The JVM is normally interpretive in nature resulting in slower execution. In order to overcome this problem, we deployed a Java processor, JOP, to exacerbate the execution of SystemJ programs compiled to Java as discussed in Chapter 3. Furthermore, we presented RJOP [91] approach to enhance the efficiency of execution of SystemJ programs by extending the instruction set of JOP [93] to support reactive statements of the language and abstraction of signals used in synchronous parts of SystemJ programs. It maintains portability of SystemJ compiler generated code, and at the same time, exhibits higher execution speed for a typical SystemJ programs. It is important to mention that back-end compiler used for RJOP does not take advantage of AGRC and does not address concurrency directly as defined in SystemJ. Furthermore, SystemJ's powerful concurrency model in that case is implemented in Java and inherits some deficiencies of Java, particularly those related to the lack of efficient program flow control in the form of *goto* mechanism.

The disadvantages associated with the implementation of concurrency and reactivity in SystemJ program compiled to Java can be averted by separating the SystemJ program concurrency and reactive control flow (CRCF) from the ordinary Java control flow (JCF) and then implementing them on two disjoint machines, such as Tandem Virtual Machine [38] or Tandem Processor [39]. These approaches have significant curtailment of memory footprint and boost the performance of SystemJ program execution when compared to JVM only approach. These implementations rely on the use of JVM on a standard processor to execute Java computations, and a specialized processor for execution of CRCF. The non-native execution of Java bytecode on a general purpose processor suffers from slower execution speed.

We exploit JOP's ability to efficiently execute the Java and replace the existing general purpose processor within TP with JOP. The tandem processor architecture is now based on JOP, called TP-JOP, integrates JOP with the existing control processor (ReCOP) to speed up the execution of SystemJ programs. The approach exactly follows the existing idea of combining Java Virtual Machine (JVM) and Control Virtual machine (CVM) into a Tandem Virtual Machine (TVM) [38], but replaces the interpreting JVM with JOP, a hardware implementation of JVM. We go a step farther and propose an approach that results in a significant breakthrough towards the use of SystemJ for embedded applications and is discussed in detail in Section 4.5.

4.3 TP-JOP

TP-JOP is a solution that is implemented based on the idea of tandem execution of two processors [39], one that controls the flow of SystemJ program (CP, control processor) and the other performs data-computations. The data driven part of the code (JCF) which is in Java was executed on a general purpose processor in previous implementation of the TP. The TP-JOP, as the name indicates, uses JOP, a full Java processor which replaces general purpose processor without interfering with native execution of CRCF on the CP. The deployment of JOP facilitates the native execution of the data-computations presented in Java. The native execution of both CRCF and JCF code in resulting TP-JOP architecture removes the performance bottleneck due to non-native execution of data-computations in existing TP architecture.

4.3.1 TP-JOP Design flow

The TP-JOP design flow is akin to the TP except TP-JOP makes a step forward by introducing JOP instead of interpreting JVM for Java computations. TP-JOP has strict separation of control (CRCF) from data-computations (JCF) as shown in Figure 4.1. Both are executed on two different processors with CRCF leading JCF. At any time, each reaction as described in Chapter 2, can be executing either pure CRCF statements or waiting on the result of Java action node to be communicated to it by Data Processor (JOP in this case) which executes this node. However, if a reaction is waiting for the result of data-computations (JCF) from the DP, the execution point of control can be transferred to another reaction, thus actually not blocking the execution of CRCF. The only point when control flow can be suspended occurs only if all the reactions of the system have requested the execution of JCF and none of them has received the result of data-computation. This suggests that communication mechanism should be capable of holding the requests in orderly fashion. This is achieved through a queue (FIFO) which stores the Java call requests and releases them in the order of their occurrences once DP becomes ready to execute them. The effectiveness of the mechanism has been demonstrated on both virtual and physical implementation of tandem operation of CRCF and JCF.

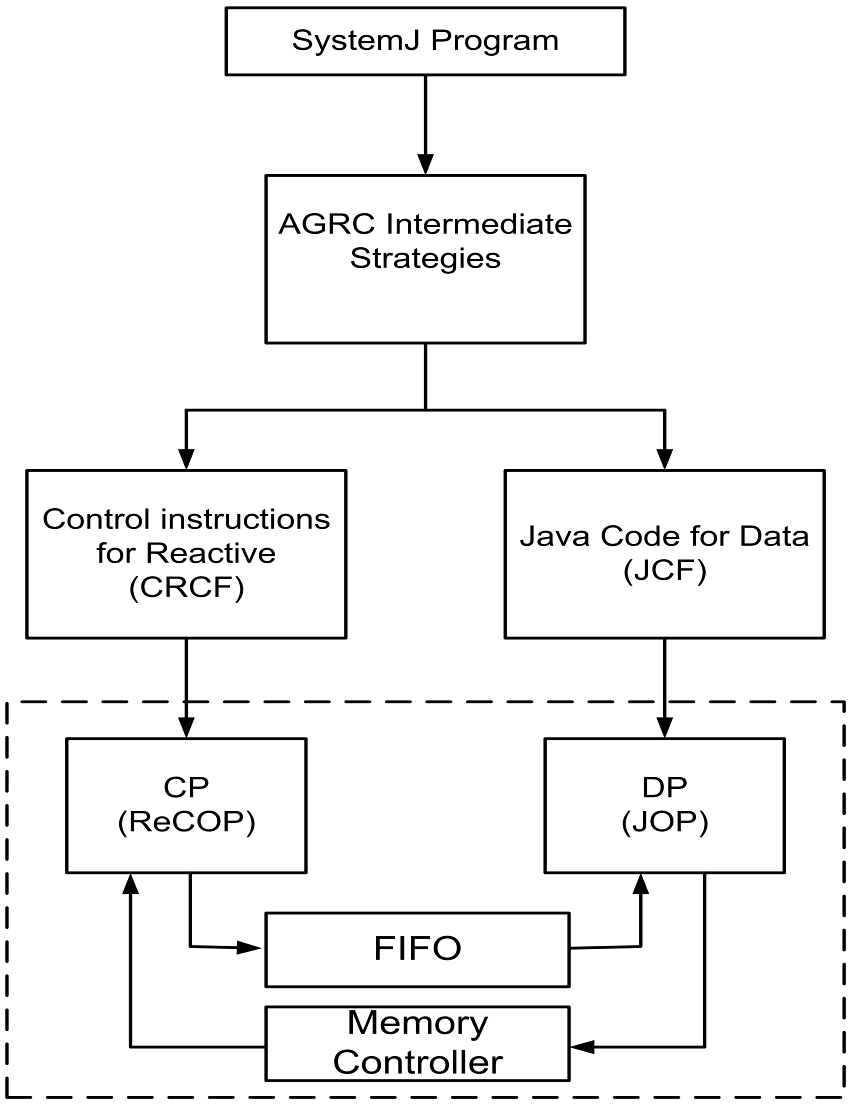


Figure 4.1: Abstract view of TP-JOP design flow and communication

4.3.2 TP-JOP Architecture

The TP-JOP architecture is shown in Figure 4.2 and resembles to the TP architecture explained in the Chapter 2 with few modifications. In case of tandem processor, the Control Processor (CP), Data Processor (DP) and FIFO are connected through Avalon bus [48]. In TP-JOP case, CP (ReCOP) and DP (JOP) communicate through the FIFO and memory controller without employing the Avalon bus. The memory controller is instilled to control the access to the data memory of the CP which is shared with the DP to write the result of the computation. The FIFO is connected to memory-mapped IO of the JOP and also has connection with the CP where the former can only read and the latter can write. The FIFO is 32-bit wide and depth is parametrized depending on the maximum number of *data calls* which can be made by the CP at the same time (within in a single tick). The FIFO has no special way to indicate a *data call* to the JOP and this is achieved by embedding 1-bit information, called *call-bit*, in the data call itself. When CP makes a data call, the *case-number*, *data-lock* location, *clock-domain* number and a call-bit are concatenated and written in the FIFO. The case-number and data-lock represent the *case* to be executed and the memory location of the CP where result will be written, respectively. The data-computation for each clock-domain reside inside the JCF, wrapped in *switch-case* statement. The case-number, together with the clock-domain number, help in identifying the case containing the required computation.

The JOP polls the FIFO all the time by reading its location pointed to by the read pointer of the FIFO and checks for the call-bit. The presence [95] of call-bit informs that a data call has been made and JOP performs the corresponding data-computations. The JOP concatenates 1-bit *Result* of the data-computations, 1-bit *result_available* indicating availability of result, and data-lock pointer and writes them into the DPRR (Data Processor Result Register) register. The memory controller receives the contents of DPRR and writes the result of the data-computations provided by the DP (JOP) to the data memory location of CP pointed to by data-lock in the FIFO by halting the CP. The JOP uses existing bytecode to read the FIFO contents and write the result to the DPRR connected to IOs. Both FIFO and DPRR are mapped to same address with former is read only and latter is write only.

4.3.3 Compiler Modification

The compiler requisites minor modifications to generate the code compatible with TP-JOP execution. The JOP makes use of specific bytecodes, instead of using utility function provided by the DP [39], to communicate with the FIFO. Figure 4.3 shows the JCF code generated for the JOP with the data-computations wrapped in a *switch-case* statement. There is no change in assembly code representing CRCF. The boolean variable *result* is the result of the computation (line 12) and *resword* contains the completion bit and the result of data-computation. The *resword* is concatenated with the data-lock pointer and stored in the DPRR register. If the result of computation is *true*, then *dprr* variable has value 3 (line 31) as both *result* and *result-available* are *true*. If the result is *false*, the *dprr* is written with the value 2 (line 32). The *dprr* variable is written to DPRR register (line 33). The contents of the DPRR (2-bit) are stored in the data memory of control processor at location pointed to by the data-lock pointer.

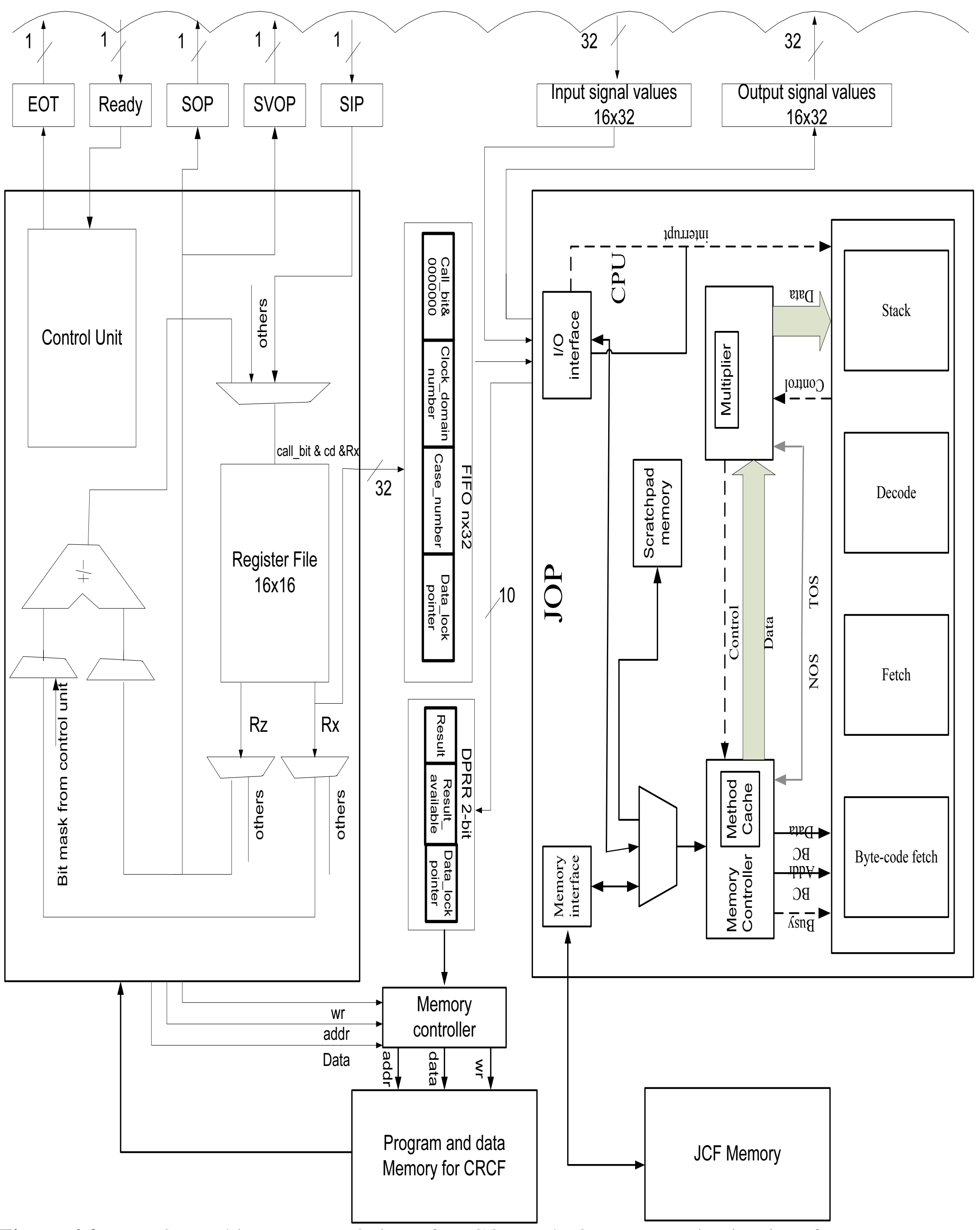


Figure 4.2: TP-JOP architecture consisting of ReCOP and JOP, communication interface components FIFO and DPRR, memories and CRCF memory controller

```
import java.util.*;
import java.io.*;
import com.jopdesign.sys.Const;
import com.jopdesign.sys.Native;

public class exampletpjop {

public static void main(String args[]){

  while(true){ //polling

   boolean result = false;
   int resword, dprr=0;
   int data_call_word = Native.rd(Const.FIFO_ADDR);
   int data_call = (data_call_word >> 20)&0x001;

   if(data_call == 1){

     int data_lock  = (data_call_word) & 0xFF;
     int case_number = (data_call_word >> 8) & 0xFF;
     int clock_domain = (data_call_word >> 16) & 0x000F;

      switch(clock_domain ){
       case 0:
         result = cbackcall0(case_number);
       break;
       case 1:
         result = cbackcall1(case_number);
       break;
      }
      if (result==true){ resword = 0x0003;}
      else { resword = 0x0002;}
      dprr = (data_lock << 2)& resword;
      Native.wr(dprr, (Const.FIFO_ADDR));
    }
   }
 private static boolean  cbackcall0(int var){

    switch(case_number){
      case 0 :
     ...
    case 1 :
     ...
    }
 }
}
```

Figure 4.3: TP-JOP example code in Java representing data-computation

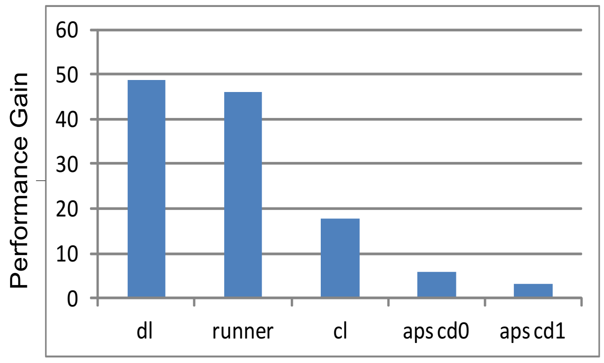
4.3.4 TP-JOP Performance Evaluation

We demonstrate the effectiveness of the proposed architecture by running benchmark examples on it. Although we have a large number of benchmark examples, but we have selected only those SystemJ benchmarks which have been used in previous implementations. The characteristics of the benchmarks have been described in Chapter 3 and the SystemJ codes are given in the Appendix D.

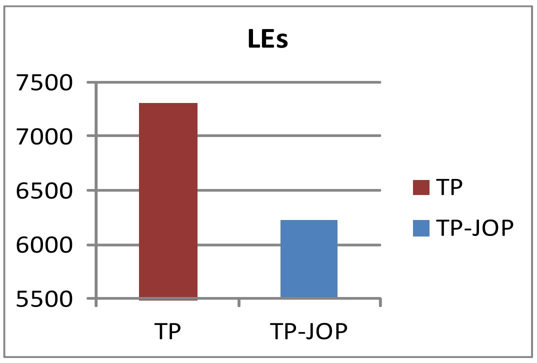
All presented data for evaluation has been collected from the experiments carried out by using the cycle-accurate *ModelSim* simulator targeting *Altera* Cyclone II FPGA with 70k logic elements and running at 50 MHz clock. The system is capable of running at 100 MHz but the results presented are for 50 MHz clock for fair comparison with earlier published results.

The solution presented in the form of TP-JOP outperforms the TP thanks to the use of JOP (hardware implementations of JVM). The JOP performs better than the software JVM implementations [50], because the bytecodes are directly executed as native instructions in hardware. In TP, the control computations (CRCF) were run natively on a control processor whereas the data-computations (JCF) were run on an interpreting JVM and therefore exhibited inadequate performance. The deployment of JOP removes this bottleneck as both CRCF and JCF are run on processors natively. The extent of the overall performance gain depends on the amount of data-computation performed by JOP which is a function of JCF embedded in the SystemJ program.

The results in Figure 4.4a show that TP-JOP [95] is 6 to 50 times faster than the TP for the given set of the benchmarks. Both TP and TP-JOP use same compilation approach which separates control from data-computation (CRCF and JCF). Both approaches use the same control processor (CP) to execute the CRCF and different processor to handle JCF (data-computation). The former uses interpreting JVM and the latter uses hardware JVM. It means both will try to match each other for applications with no data-computations involved at all. The disparity in the performance grows with the introduction of data-computation and is directly proportional to the amount of data-computations involved. The *combinational lock* is more control-oriented; therefore, the performance gain is less compared to *demoloop* and *runner* which have reasonable amount of data-computations. The resource usage comparison given in Figure 4.4b shows that TP-JOP uses 15% fewer resources than TP. Therefore, TP-JOP is more efficient platform when compared to TP both in terms of execution times and resources used.



(a) TP-JOP execution time comparison



(b) Resource usage comparison

Figure 4.4: TP-JOP performance evaluation

4.4 Limitations of TP-JOP

TP-JOP has strict separation of execution of control (CRCF) from data-computations (JCF), which inevitably results in duplication of some of the computation resources. The two major downsides to this approach are (1) increased hardware complexity since there is little design reuse between the two types of processors and (2) poor resource utilization when the application mix contains a balance different than that ideally suited to the underlying heterogeneous hardware. Although the CP (ReCOP) used in TP is very simple, it duplicates basic instructions for data movement and arithmetic operations, and also requires two program memories in which CRCF and JCF part of programs are stored. Also, major data-structures that represent clock-domains, reactions and signals, are contained in the data memory of CP. Only handfuls of instructions are specialized instructions that operate on these objects or are dedicated to support control operations related to AGRC-based control flow. This was a major motivation to look more closely at how functions of the program control flow and interactions with the environment could be merged with JOP's processor functionalities resulting in a more economical and efficient implementation. Also, TP-JOP as a two processor approach suggests that data must transfer between the processors over some sort of interconnects. It is exorbitant and incurs performance overheads due to adopted communication mechanism. Furthermore, there are cases when only one processor is busy and the other one is not doing any useful task. These cases may arise when JOP is polling on CP for data call or CP is waiting for results of data-computations, resulting in under-utilization of processors and unnecessary consumption of power.

The need of a peculiar CP can be avoided as its functions can be built right into the JOP processor itself - eliminating inter-processor data transfers over a slow communication medium. We analyze the TP-JOP and carefully merge a minimal set of features of the CP into JOP by extending JOP's instruction set, memory model and data-path. The CP instructions are mapped to the unused bytecodes, the hardware support for the new bytecodes and its associated control is provided by extending the JOP's data-path. As mentioned in Chapter 3, JVM has only 256 bytecode and JOP uses around 230 of them. This number of free available bytecodes is less than the instruction set of CP. Therefore, it requires some thinking to map the ISA of CP to available bytecodes The new processor, called GALS-JOP, facilitates efficient execution of synchronous and asynchronous concurrency and reactivity (CRCF) and Java oriented data-computations (JCF) by merging best of both worlds at low cost. Importantly, the design approach does not need any essential modifications in the compilation flow of SystemJ [26], which is based on a formal semantics, giving advantages over non-formal programming languages and their compilation approaches.

4.5 GALS-JOP

The GALS-JOP subjugates the impediments offered by the two processor approaches as discussed in Section 4.4. It can be viewed as a merger of JOP and CP for the improved SystemJ program execution on a single processor. The GALS-JOP approach is thus a common answer to questions from two different worlds and offers the following improvements when compared to the TP-JOP approach:

- It takes the advantage of already demonstrated compilation approach where CRCF and JCF are separated from each other.
- Executes them on a single processor without the need of a separate CP.
- It is a single processor approach, hence, eliminates the communication between the processors and associated communication hardware.
- Saves the logic resources used to implement the CP and communication among the processors.
- Significantly raises the cost-effectiveness of the resulting solution compared to TP and TP-JOP approach.
- Achieves the goal of more conducive execution of SystemJ programs on small, embedded platform.
- This approach has better utilization of available time when compared to multiprocessor approach as the processor is always busy

The approach to use JOP as a base of the new processor utilizes the fact that JOP is a complete Java processor and allows extensions of its instruction set, as well as the data-path. Also, the decision to preserve SystemJ compiler and JOP tools in their entirety has been another motivation for this approach. The goal of this system is to achieve performance and efficiency approaching that of application specific systems.

4.6 GALS-JOP Compilation and Execution Strategy

Figure 4.5 shows the entire design flow adopted for compiling SystemJ program and executing it on GALS-JOP. The SystemJ compiler front-end takes the SystemJ application program (*.sysj*) as input and produces the AGRC graph of the compiled program. The compiler back-end identifies and separates the CRCF and JCF. The CRCF is generated in the form of assembly code (*.asm*) in a usual fashion. The generation of JCF differs from the one generated for the TP and TP-JOP as the data-computations are no longer wrapped in the *switch-case* statements. Instead, the data-computations for each case is enclosed in Java methods (*.java*). The original SystemJ compiler back-end is modified to produce the JCF code in the required form. In order to execute the CRCF, the JOP is extended by utilizing the unused bytecodes (bytecodes are the instruction set of JVM). This requires the modification to the JOP tool chain as well as to its hardware.

Each CRCF instruction (or a group of instructions explained in detail in the next sections) is mapped to a single bytecode (or more than one bytecode in some cases). If there exists a Java bytecode, performing the same operation as described by the CRCF instruction, then the CRCF instruction is mapped to the existing bytecode (or bytecodes). If no such bytecode exists, then an unused bytecode is used to map the CRCF instruction and hardware support is provided to implement the functionality described by the CRCF instruction which is mapped to that particular bytecode. The data calls launched by the CP to DP in TP and TP-JOP approach are replaced by conventional method calls to those particular methods. It should be noted that the data computations no longer require the *case_number* as they are recognized by their unique name as in conventional Java code. The JOP tool chain describes the methodology to implement an unused bytecode. These custom bytecodes are used as native methods in Java code (with *Native* prefix). These native methods are replaced by the special bytecodes by the JOP tools. The field values embedded in a CRCF instruction can be passed as arguments to that method. Hence, each CRCF mapping to bytecode results in a Java statement, which is always a method and might or might not have a *Native* prefix depending on it if it is conventional method call or is meant for a bytecode mapping. The data-structures of CP is implemented in the heap in the form of array object; hence, a Java statement declaring arrays is also the part of the code.

The outcome of CRCF translation is a Java program comprising of array declaration statement, native methods representing CRCF instructions and the Java methods embedded in between the CRCF instructions performing data-computations. Now we have a Java class with a *main* method and all the Java statements generated from CRCF translation are placed inside this *main* method. The methods representing data-computations are also declared and implemented in the same class having the same name as the SystemJ application. The details of the tool translating CRCF into Java are described in Section 4.6.1 and an example which guides through the procedure is given in Section 4.6.2.

The Java class (containing SystemJ application compiled to Java targeted for GALS-JOP) is compiled by the *Javac* compiler which reads class and interface definitions and compiles them into bytecode class file. This code cannot be executed as it contains the *Native* methods which are not real functions and are substituted by special bytecodes on application building with *JOPizer*. The JOPizer is a JOP specific tool based on open source *BCEL*, which links a Java application and converts the class information to the format that JOP expects (*.jop*). *Native.Java*, a JOP system class, consists of native methods for low-level functions – the code we want to avoid in application code as they require operating system or the Java native interface (JNI) [96] support. Some JNI implementations add significant overheads, and are not suitable for Java processor based embedded platforms [93].

The CRCF instruction set also contains both jump and branch instructions. Since, Java does not have *goto* statement, therefore, this functionality is incorporated in GALS-JOP with the help of a jump-table. A tool, called Jump-Table Generator (JTG), generates the jump-table. It scans the compiled application file generated by *JOPizer* and generates a jump-table implemented in VHDL comprising of the target addresses for the jump instructions. This tool is discussed in detail in Section 4.6.3. As an alternative, the jump-table can also be generated by modifying the existing tool set of JOP.

The method codes for all the methods reside in a main memory external to the JOP. The JOP uses a *method* cache to hold the method code which has a conventional size of 4KB. The *main* method of the Java class, containing the CRCF code of a SystemJ application translated to Java statements as mentioned above, produces too big code upon compilation to fit in the conventional sized cache. The GALS-JOP overcomes this problem by introducing a separate memory which holds the CRCF code translated to Java statements permanently and has an additional advantage as it does not need to be loaded into *method* cache prior to its execution. This memory is called Control Memory (CM) and it is initialized during the start up of the SystemJ program on GALS-JOP as discussed in Section 4.12. Once initialized, this memory is locked and its contents cannot be modified. The other methods, apart from the *main* method, are loaded into the *method* cache from the main memory prior to their execution. Hence, program execution switches between the CRCF and JCF and the next bytecode is fetched either from the CM or *method* cache. As a result, the return to *main* method (CRCF code residing in CM) is different from the return to other Java methods as it does not need to be loaded into cache. Once, the *main* method is completely executed, it returns to *boot* method and executes the *JVM exit* method.

4.6.1 AJT-Assembly to Java Translator

Assembly to Java Translator (AJT) tool converts the assembly code of CP to native method as shown in Table 4.1. Upon compilation for JOP, the native methods are replaced by the special bytecodes by the *JOPizer* tool. Our translator parses the assembly code and translates each assembly instruction into one or more Java native methods. We further illustrate this with the help of an example of CP instruction. It performs a logic AND operation between a register operand and an immediate operand provided in the instruction.

$$Rz \leftarrow Rx\ AND\ Operand$$

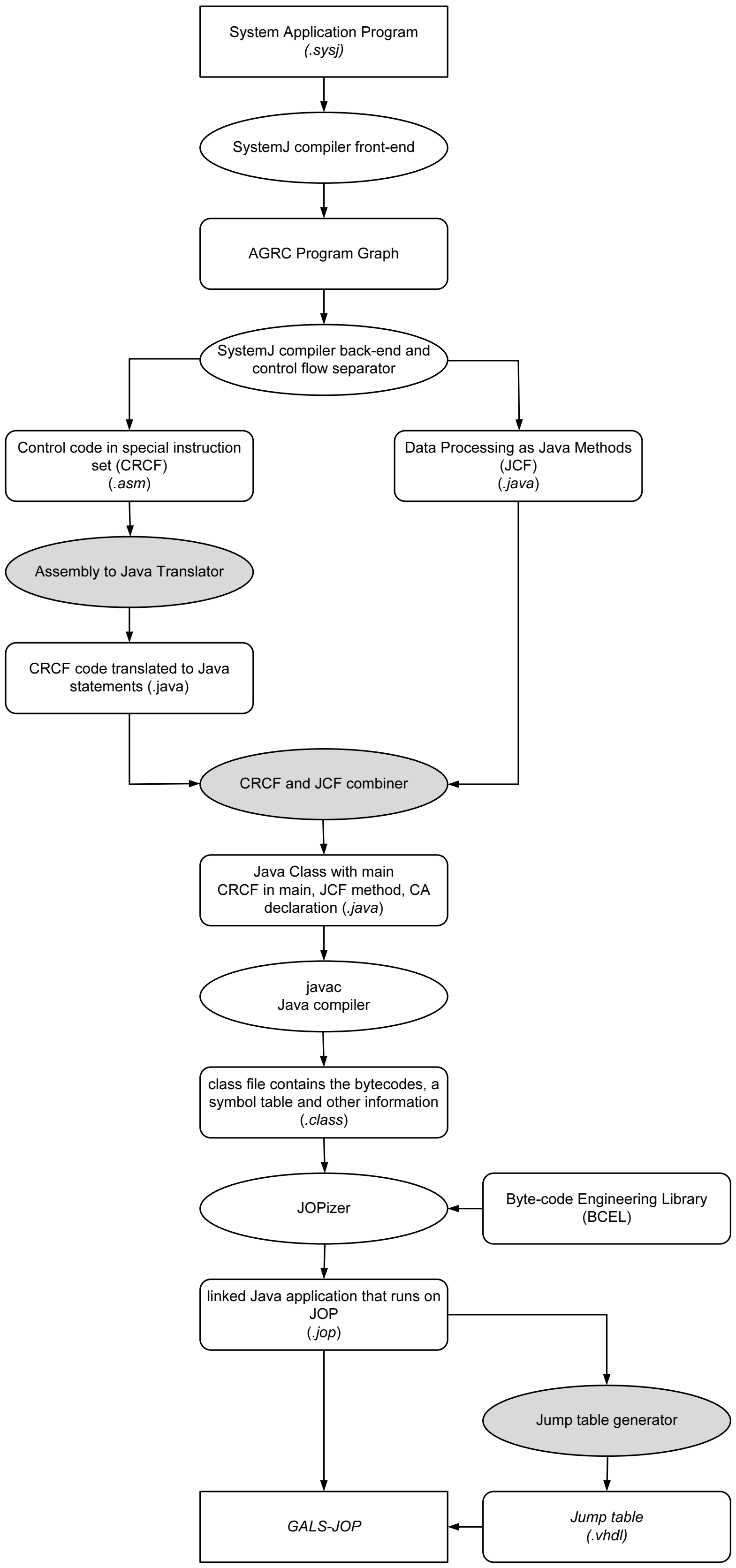

System Application Program (.sysj)
SystemJ compiler front-end
AGRC Program Graph
SystemJ compiler back-end and control flow separator
Control code in special instruction set (CRCF) (.asm)
Data Processing as Java Methods (JCF) (.java)
Assembly to Java Translator
CRCF code translated to Java statements (.java)
CRCF and JCF combiner
Java Class with main CRCF in main, JCF method, CA declaration (.java)
javac Java compiler
class file contains the bytecodes, a symbol table and other information (.class)
JOPizer
Byte-code Engineering Library (BCEL)
linked Java application that runs on JOP (.jop)
Jump table generator
GALS-JOP
Jump table (.vhdl)

Figure 4.5: GALS-JOP design flow

Instruction	**Java Translation**	**Mode**
AND Rz Rx Operand	*Native.aluimm(Rz, rand, Rx, Operand)*	*immediate*
	Native.aluind(Rz, rand, Rx, Rz)	*indirect*
OR Rz Rx Operand	*Native.aluimm(Rz, ror, Rx, Operand)*	*immediate*
	Native.aluind(Rz, ror, Rx, Rz)	*indirect*
ADD Rz Rx Operand	*Native.aluimm(Rz, radd, Rx, Operand)*	*immediate*
	Native.aluind(Rz, radd, Rx, Rz)	*indirect*
SUBV Rz Rx Operand	*Native.aluimm(Rz, rsub, Rx, Operand)*	*immediate*
SUB Rz Rx Operand	*Native.subimm(Rz, Rx, Operand)*	*immediate*
LDR Rz Rx Operand	*Native.ldrimm(Rz, operand)*	*immediate*
	Native.ldrind(Rz, operand)	*indirect*
	Native.ldrdir(Rz, Rx)	*direct*
STR Rz Rx Operand	*Native.strimm(Rz, operand)*	*immediate*
	Native.strind(Rz, operand)	*indirect*
	Native.strdir(Rz, Rx)	*direct*
JMP Rx	*Native.jmpimm(label_index)*	*immediate*
	Native.jmpdir(Rx)	*direct*
PRESENT Rz Operand	*Native.present(Rz, label_index)*	*immediate*
SENDATA Rx	*Method_id()*	*indirect*
CHKEND Rz Rx	*Native.chkend(Rz, Rx)*	*indirect*
SWITCH Rz Rx	*Native.wr(Rz, Rx,1)*	*indirect*
SZ Operand	*Native.sz(label_index)*	*immediate*
CLFZ	*Native.clfz();*	*inherent*
CER	*Native.wr(0, CONST.IO_ER)*	*inherent*
CEOT	*Native.wr(0, CONST.IO_EOT)*	*inherent*
SEOT	*Native.wr(1, CONST.IO_EOT)*	*inherent*
LER Rz	*Native.ldio(Rz, CONST.IO_ER)*	*indirect*
SSVOP Rx	*Native.wr(Rx, CONST.IO_SVOP)*	*indirect*
LSIP Rz	*Native.ldio(Rz, CONST.IO_SIP)*	*indirect*
SSOP Rx	*Native.strsop(Rx, CONST.IO_SOP)*	*indirect*
NOOP	*Native.noop();*	*inherent*
LABEL	*Native.label();*	*inherent*

Table 4.1: Translation of CP instructions to Java methods representing user specific bytecodes

This instruction performs logical AND operation on the contents of the register Rx with the immediate operand, and stores the result in the register Rz. The information provided in the instruction consists of source operand register address, immediate operand and destination register address. When translating to Java, for each combination of assembly instruction operation and mode, there exists a Java method which is mapped to new or existing bytecode during compilation. The instruction fields are passed as the arguments in the Java statement. An example of translation of assembly instruction to Java is given below where instruction fields Rz, Rx and operand are passed as arguments.

$$Native.andimm(Rz, Rx, Operand);$$

The translation of the assembly instruction performing immediate jumps is more complicated as these instructions need the target address for their execution. In the assembly code, the targets are labels. The AJT tool parses the assembly code and makes an index of all the target (those which appear) labels in the order of their appearance. The addresses specified by labels are available only after the processing of *class file* by *JOPizer*. The index of the label is passed as the argument during the translation of the immediate jump and conditional branch instructions. The index is used as address for the jump-table which consists of the target jump addresses. It is worth mentioning that each label is also translated to a Java statement and helps to build the jump-table as explained in Section 4.6.3. The Table 4.1 shows the translation of each CP's machine instruction to corresponding Java statements.

It is necessary to translate the whole assembly code into Java before it can be executed. All Java statements generated by AJT tools are placed in the *main* method of the Java class containing the SystemJ application program compiled for GALS-JOP.

4.6.2 GALS-JOP Example Code

The code generated for GALS-JOP is illustrated with the help of a simple example in Figure 4.6. It has only one clock-domain comprising of two reactions (starting at line 7 and 18, respectively), two pure output (status-only) signals (declared at line 4) and one local signal (line 8). The first reaction checks the local signal (line 10), if it is present, one of output signals is emitted (line 11) and a statement is printed out (line 12). Then other signal is emitted (line 14) and variable is assigned some value (line 15). The reaction ends with the pause statement (line 16). In the second reaction local signal is emitted (line 19) and the value of the variable is printed out (line 20). Next, code generated by the modified SystemJ compiler is shown in Figure 4.7 where JCF (line $1-17$) is separated from CRCF (line $18-38$) and given in the form of Java and assembly code respectively. The Java data-computations wrapped in a *switch-case* statement in the former case are broken into small methods now. Only an excerpt of the CRCF assembly code is shown as the purpose is to illustrate the code generation for GALS-JOP rather than explaining the functionality. Three different blocks in AGRC are identified as data-computations and are encapsulated in three different methods (line $5-14$). The assembly code loads the local signal (line 26) into a register and checks if it is present (line 27). If the signal is not present, the control flow of the program is shifted to some other location pointed to by the label. If local signal is found present, then data-lock position in the memory is locked by storing a 0 there (line 27 – 30). Next, a data call is made (line 31) and the code checks for the result of the computation (line 32 – 36) and the flow of the program is shifted accordingly.

```
system {
      interface {

        output signal RE,TE;
    }
    {
        {
           signal pipi;
           {
           present(pipi) {
               emit RE;
               System.out.println("GALS-JOP");
           }
           emit TE;
           int u=5;
           pause;
        }
        ||
        {
           emit pipi;
           System.out.println(u);
           pause;
        }
     }
     ><
     {

     }
  }
}
```

Figure 4.6: SystemJ example code

```
//Java code for data computation
Public Class galsjop{
 public static int u=0;
   ...
 private static method_0(){
    System.out.println("GALS-JOP");
 }

 private static method_1(){
    u=5;
 }
 private static method_2(){
    System.out.println (u);
 }
}

// Assembly code for Control
L1 SEOT
CER
LDR R0 $0001
AND R0 R0 #$f000
SSOP R0
LSIP R0
AND R0 R0 #$0
...
LDR R1 $0002
AND R1 R1 #$0001
PRESENT R1 L4
LDR R0 #3
ADD R1 R6 #0
STR R1 #0
SENDATA R0
ADD R4 R4 #1
L2 LDR R0 R1
CLFZ
SUBV R0 R0  #0
SZ L3
JMP L7
L3 CLFZ  ;
...
```

Figure 4.7: SystemJ program compiled by separating the CRCF and JCF represented as assembly instructions and Java methods, respectively

The code generated for GALS-JOP target for the above example is shown in Figure 4.8 in GALS-JOP Java class (line 2) consisting of JCF methods (line $4 - 13$) and CRCF code is inside the *main* method (line 15). The *main* method starts with the array declaration which reserves the space in the heap where the data-structure of the CRCF resides and, in fact, emulates the data memory. As mentioned earlier, each label of CRCF is translated to a native method (line 19) and the corresponding bytecodes are looked for after compilation to build the jump-table. The contents of the environment registers are written using the existing native methods (bytecodes) specific to JOP. The PRESENT statement (line 29) is provided with the index of target label as argument and performs conditional jump. The data calls implemented using SENDATA are reduced to conventional Java method calls (line 34).

```
1 //code for GALS-JOP
2 public class galsjop {
3  int u=0;
4  private static void methodcall_0( ) {
5    System.out.println("GALS-JOP");
6    Native.ldrimm(15, 1 );
7    Native.strdir(15, 0x2);
8  }
9  private static void methodcall_1( ) {
10    u=5;
11    Native.ldrimm(15, 1 );
12    Native.strdir(15, 0x2);
13 }
14   ...
15 public static void main(String[] args) {
16   int[] CA = new int[CAsize];
17   Native.cabaseaddr();
18   ...
19   Native.label();  //L1
20   Native.wr(1,Const.IO_EOT);
21   Native.wr(0,Const.IO_ER);
22   Native.ldrdir(0,0x1);
23   Native.aluimm(0,rand,0,0xf000);
24   Native.strsop(0,Const.IO_SSOP)
25   Native.ldio(0,Const.IO_SIP);
26   Native.aluimm(0,rand,0,0x0);
27   Native.ldrdir(1,0x2);
28   Native.aluimm(1,rand,1,0x0001);
29   Native.present(1,4);
30   Native.strdir(0,0x0);
31   Native.ldrimm(0,2);
32   Native.aluimm(0,radd,0,0x0);
33   Native.strimm(1,0x0);
34   Methodcall_0();
35   Native.aluimm(4,radd,4,1);
36   Native.label();
37   Native.ldrind(0,1);
38   Native.clfz();
39   Native.subv(0,0);
40   Native.szjump(3);
41   Native.label();
42   Native.jmpimm(7);
43   Native.clfz();
...
```

Figure 4.8: Example code ready for execution on GALS-JOP generated by AJT tool translating the CRCF instructions to Java

4.6.3 JTG - Jump-Table Generator

The CRCF instruction set also consists of both conditional and unconditional jump instructions. The jump in assembly is performed with the help of labels, which are symbols representing the memory addresses of instructions or data. The assembler calculates the address of a label relative to the origin of the section where the label is defined during assembling. Java has no *goto* statement and GALS-JOP is enriched with this functionality by using special bytecodes. These CRCF instructions are mapped to special bytecodes which are provided with the hardware support in the form of a jump-table to emulate the jumps. The Jump-Table Generator (JTG) generates a jump-table which contains the target addresses of all the labels as shown in Figure 4.9. Apart from CRCF instructions, the labels appearing in CRCF assembly code are also translated to native methods as shown in Table 4.1 and are replaced by special bytecodes by the *JOPizer*. The bytecodes for each method are available in the JOP compatible file generated by JOP along with the start address of the method. The JTG tool scans it and looks for the *main* method which contains all the special bytecodes representing the CRCF instructions including the bytecodes for labels. The JTG looks for the bytecodes representing the source label and, each time it comes across such a bytecode, its corresponding address is calculated by adding its location number to start address of the method. The absolute addresses of the labels are saved in the jump-table in the order of their appearance. The final table size depends on the number of the label entries. Hence each jump bytecode is accompanied by an index of the label as a parameter which is then used to read the target address from the jump-table. The jump-table is implemented in an on-chip RAM and initialized at the start.

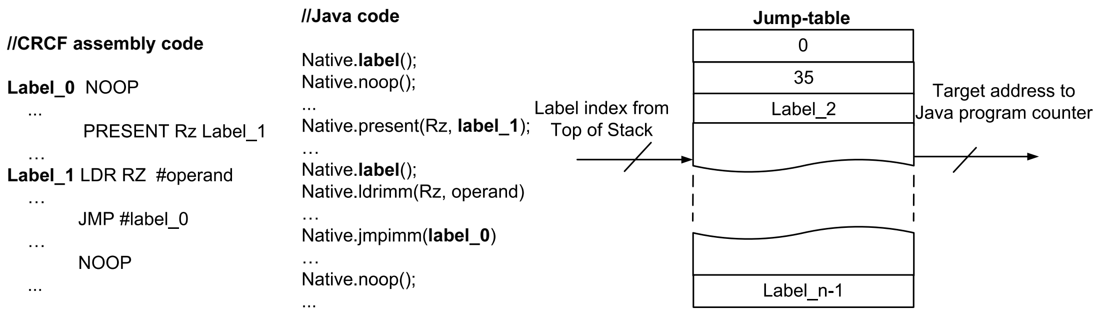


Figure 4.9: Jump-table generation procedure illustration

4.7 GALS-JOP Architecture

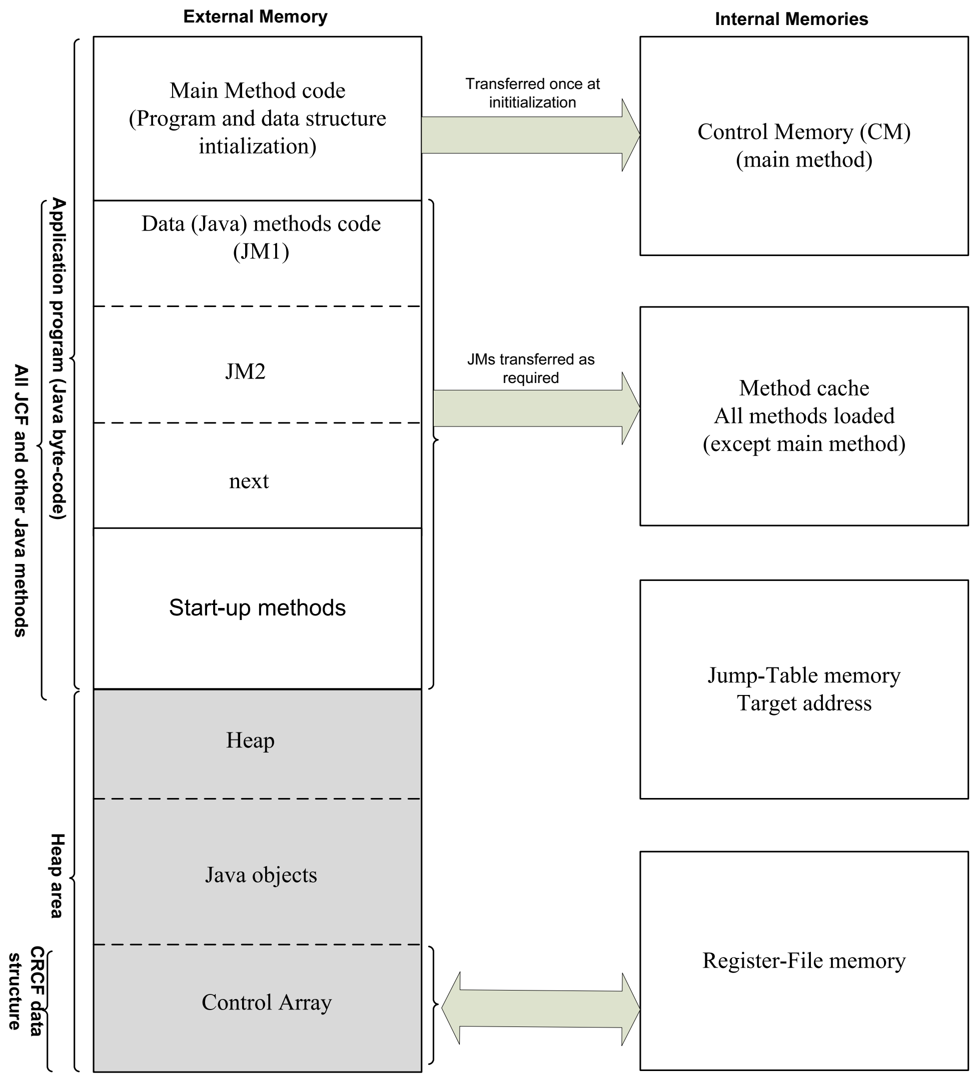
The GALS-JOP executes the code where both CRCF and JCF are merged into a single class. The JOP architecture is augmented with memories, register and other components to provide architectural support for the CRCF operation at register transfer level.

4.7.1 Memory Organization

GALS-JOP has a number of functionally-specific memories, which contribute to the faster SystemJ program execution.

Main Memory

The main memory is inherited from JOP and it is where SystemJ application code compiled for GALS-JOP resides. The size of main memory is 2 MB and, once application code is loaded, the remaining space is reserved for the heap where all the objects reside as shown in Figure 4.10. It contains all the initialization code together with CRCF and JCF code prior to their execution. The CRCF data-structures are implemented in an array, called Control Array (CA), and reside in the heap part of the main memory. The CRCF data-structure is used to implement asynchronous (clock-domains) and synchronous concurrency (reactions). The data-structures of the CRCF program has been discussed in details in Chapter 2.


External Memory
Internal Memories
Main Method code (Program and data structure intialization)
Transferred once at inititialization
Control Memory (CM) (main method)
Data (Java) methods code (JM1)
JM2
next
Start-up methods
JMs transferred as required
Method cache All methods loaded (except main method)
Jump-Table memory Target address
Heap
Java objects
Control Array
Register-File memory
Application program (Java byte-code)
All JCF and other Java methods
Heap area
CRCF data structure
Figure 4.10: GALS-JOP memory organization

Besides main memory, GALS-JOP has a number of additional memories for faster access to the program instructions. The *method* cache is inherited from JOP and it always contains currently executing Java data method except the *main* method. On invocation of another Java method, the *method* cache is loaded with the new method code to be executed. Also, not shown in Figure 4.10 , is the stack that serves as a temporary local storage for all data methods.

Control Memory

This memory is used to hold the method code of the *main* method of the Java class targeting the GALS-JOP. The *main* method of the application class, which contains the CRCF code translated to native methods, has an exception as it is not loaded into the *method* cache prior to its execution. The reason for adopting this approach is twofold. Firstly, conventional *method* cache is unable to hold the large CRCF code due to the size restriction. Secondly, whenever a method is called, it is loaded into cache and on return, the *method* cache is loaded with caller method. SystemJ program flow is controlled from the CRCF part of the program residing in the *main* method and calls JCF as and when required. Suppose, if the cache size is modified to hold the whole CRCF code, it will be loaded to the *method* cache whenever it returns after executing JCF code enclosed in a method. The frequent loading of *method* cache with relatively large *main* method is expensive and slows down the program execution. Therefore, CRCF code is stored in a separate memory, called CRCF memory or Control Memory (CM). This memory is initialized with the *main* method's code during the start-up. Unlike *method* cache, this memory is locked after initialization and does not change its content during the program execution. Depending on the execution flow, the bytecodes are fetched from the CM or *method* cache for CRCF and JCF code execution, respectively.

Register File

The CRCF instructions define a set of registers which are used to stage data between memory and the functional units. In its original implementation, the register file is implemented as multiport memory with two read ports and single write port. In case of GALS-JOP, the reading of the operand from the register file is sequentialized, therefore, expensive multiple read port is not needed any more. Hence, the register file incorporated in GALS-JOP is single port in contrast to original and allows reading only operand at a time. The register file is implemented as an array of registers having dimensions of 16x16.

Jump-table

Another memory incorporated in the GALS-JOP is jump-table which is implemented using a ROM. The jump-table is essentially used to compensate a lack of *goto* in Java (and JVM). The details of jump-table along with it functionality have been discussed in Section 4.6.3.

It should be noted that all GALS-JOP memories, except the main memory, are implemented as FPGA internal memories and can be customized to the necessary application-specific size.

A number of registers are also introduced which are used to hold different addresses and some are more important which are discussed below.

Temporary Registers

Two temporary register *T1* and *T2* are used to provide the interface between new components added and the base JOP processor without interfering the stack. They are a sort of alternate to the top two element of the stack. The data transfer between Top of Stack and *T1* is also allowed.

MCA Registers

This register is used to hold the contents of *jpc* register when *main* method invokes another mehtod. Under normal circumstances, whenever main method invokes another method, the relative address of next bytecode to be executed available in Java program counter (jpc) is pushed onto the stack. Upon return, the caller method is loaded into the *method* cache by the memory subsystem. The memory subsystem also provides the start address of the method in the cache. The relative address is popped from the stack and added to calculate the address of the next bytecode for execution. In case of GALS-JOP, the *main* method is not reloaded upon return, therefore, start address of the *main* method is not available and also relative address is not pushed on the stack. Instead, the absolute address of the next bytecode of the main method is stored in a register. The *main_continue_address* (MCA) is used to hold the address of next bytecode to be executed when *main* invokes any method. When returning to the *main* method, the address in the MCA is loaded into Java program counter to fetch the next bytecode.

CAB Register

This register is used to hold the base address of the array situated in heap implementing the data-structure of the CRCF. Figure 4.11 shows the mechanism handling the array access. The reference of the array object stored in the heap is read when accessing an array. The array reference points to a handle area and the first element in the handle area points to the first element of the array and the length of the array can be found at the offset 1 in the handle area.

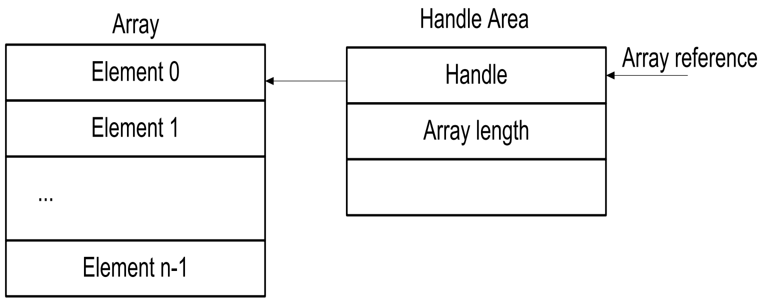


Figure 4.11: Accessing array elements from the heap

In order to access the array element, it is required to fetch the array reference and handle from the main memory. This requires multiple access to the external main memory and is therefore expensive. The access to CA is simplified and accelerated by using the *control array base* (CAB) register that stores the base address of array to provide direct access to the CA. The index of memory element is simply added to the array base address before performing the memory read or write operation providing fast and direct access to the arrays. Otherwise; each CA access will go for high latency read of object reference and CA base address. This is important as the CRCF code frequently access the data-structures implemented in the CA.

Level Tracker Register

The Level Tracker register is the part of Level Tracker circuitry that keeps the track of depth of nested calls of the methods. The Level Tracker provides the following information:

- It informs the source of next bytecode will either be CM or *method* cache.
- It indicates that whether a return is being to *main* method or some other method.

This information is used to make the decision about the loading of caller method into *method* cache upon return. The LT register is implemented as a counter. It is reset upon *main* method execution and increments whenever a method is invoked. It is decremented upon return from a child method to the parent method. This is used to evade the loading of the *main* method into *method* cache.

4.7.2 GALS-JOP Instruction Set

It should be noted that we are dealing with three different instruction sets: bytecodes, microcodes and CRCF instructions. The CRCF instructions are translated to Java. The bytecodes are the instructions that make up a compiled Java program and are executed by a Java Virtual Machine as JVM does not assume any particular implementation technology. The microcode is native instruction set of JOP and bytecodes are translated into JOP microcodes during their execution.

4.7.3 Bytecode Extension

The additional bytecodes incorporated into existing JOP's instruction set functionally correspond to the instructions of the Control Processor (CP), which are now adopted, and modified where necessary, into GALS-JOP instruction set. However, as the CP has certain number of instructions with identical or similar functionality to the existing JOP's bytecodes, those instructions are not included into GALS-JOP, thus resulting in a reduced total number of instructions compared to TP-JOP. There are 31 CRCF instruction which need to be mapped to 23 free available byte-codes. This is achieved by merging the instructions which perform similar function. For example, all *alu* instructions with *immediate* mode are merged and the function to be performed is passed as an argument. In total, GALS-JOP has 21 more bytecodes when compared with the JOP. The native methods are replaced by the special bytecodes and naming convention for the special bytecodes is *jopsys_name*. For example, *Native.xyz ()* method will be replaced by *jopsys_xyz* bytecode.

Bytecode extension		
jopsys_ldrind	*jopsys_strimm*	*jopsys_aluimm*
jopsys_ldrdir	*jopsys_jumpimm*	*jopsys_aluind*
jopsys_strio	*jopsys-subv*	*jopsys_cabaseaddress*
jopsys_label	*jopsys_present*	*jopsys_ldio*
jopsys_jumpind	*jopsys_sz*	*jopsys_clfz*
jopsys_strdir	*jopsys_chkend*	*jopsys_initctrl*
jopsys_strind	*jopsys-switchjump*	*jopsys_label*

Table 4.2: List of bytecodes introduced to support CRCF

4.7.4 Microcode Extension

The microcodes are used to implement register transfers in JOP's data-path. We introduce new microcodes to implement the functionality of new bytecodes. These new microcodes fit within the existing microcode ROM of JOP without using any extra resources [50]. Careful analysis of these register transfers resulted in the need of only 23 new microcodes as given in Table 4.3. The register transfer of each microcode is provided in the Appendix E.

4.7.5 GALS-JOP Start-up and Program Flow

The boot up sequence of GALS-JOP is shown in Figure 4.12. The initial part of start-up procedure is very similar to JOP; therefore, its details are omitted. When JOP is the target, all the Java methods are loaded into *method* cache first for faster access. During the start-up procedure, the *method* cache is loaded with the bytecodes of the *Startup.boot()* method. A similar approach

Microcode	**Description**
stdmbaseaddr	*Stores the base address of CA into CAB register*
popfromstack	*Loads the contents of A into T1*
wrrf	*Writes the content of T1 int to the regiter file*
rdrf	*Reads the contents of resiter file into T1.*
lockmain	*Indicates that main has been loaded*
ldmaincontaddr	*Load the address of next instruction to be executed into T1*
ldmaininvokcheck	*Loads the flag indicating that method is being invoked by main*
ldmaininvokedcheck	*Loads the flag telling that main is being invoked*
wrctrl	*Loads the contents of main method into control memory*
settick	*Sets the bit when clock-domain has completed the tick*
reslevel	*Resets the level tracker*
inclevel	*Increments LT*
declevel	*Decrements LT*
stmaincontaddr	*Stores the address of next instruction to be executed by main into MCA*
ldmainreturncheck	*Checks wheher returning to main*
aluop	*Performs arithmetic logic operation*
pushonstack	*Pushes content of T1 into A*
clrzf	*Clears the zero flag*
dmaddr	*Calculates the address for CA*
srzf	*Sets the zero flag*
ctrlinitfinished	*Indicates that control initialization is finished*
findmax	*Finds the maximum nibble*
loadjmpaddr	*Loads contents of jump-table into T1*

Table 4.3: Description of extended microcodes incorporated to implement the functionality of new byte-codes

applies to the GALS-JOP with the exception of the *main* method, comprising of control flow instructions (CRCF) and calls to data-computations methods (JCF), which are not loaded into the cache. The main method is permanently loaded into another fast memory called Control Memory (CM) during the initialization phase. Once start-up is completed, the *main* method is invoked. Invocation of the *main* method does not require access to the *method* cache and its bytecodes are fetched from the CM, which is another source of bytecode now together with the *method* cache. All the methods invoked by the *main* method are loaded into and executed from the method cache until the control returns to *main* method.

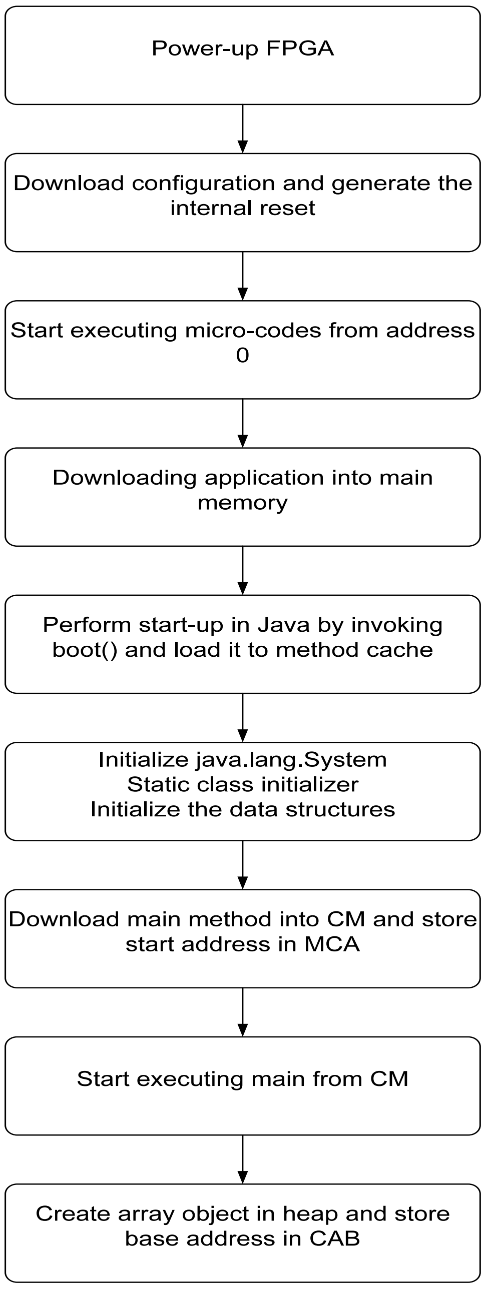

Power-up FPGA
Download configuration and generate the internal reset
Start executing micro-codes from address 0
Downloading application into main memory
Perform start-up in Java by invoking boot() and load it to method cache
Initialize java.lang.System
Static class initializer
Initialize the data structures
Download main method into CM and store start address in MCA
Start executing main from CM
Create array object in heap and store base address in CAB

Figure 4.12: GALS-JOP start-up and execution flow

The management scheme for the bytecode fetch from two spatially distinct sources is shown in Figure 4.13. Whenever *main* method invokes a Java method, the address of the next bytecode is stored in MCA register. This address is loaded into JOP's program counter (*jpc*) to resume the execution of the *main* method after returning from JCF method. Similarly, return to the *main* method is different from the return to other methods as it does not require *main* method to be loaded into the cache, which saves the time otherwise consumed in loading the cache. Hence, we have different strategies for invoking *main* and other methods which requires keeping track of method call nesting by using a Level Tracker (LT). The LT is enabled as soon as CM is initialized and *main* method is invoked. Any method invocation increments the Level Tracker, and return decrements it. If Tracker is enabled and a method is being invoked with level 0, it means the *main* is being invoked and level 1 means a current executing method has been invoked by the *main*. Similarly, when returning, the level is first decremented, and then it is checked. The return with level 1 means returning to the *main* method. The Level Tracker Decoder (LTD) generates different check signals such as *main_invoked_check*, *main_invokes_check* and *main_returned_check* which are used to carry out different invoke and return strategies.

4.7.6 GALS-JOP Data-path

The GALS-JOP data-path is presented in Figure 4.14. The JOP architecture consists of the processor core, a memory interface and a number of IO peripherals. The JOP core has four pipeline stages: Bytecode Fetch, Fetch, Decode and Execution (or Stack) explained in detail in Chapter 3. In the Bytecode Fetch stage, the Java bytecodes are fetched from the internal RAM. In the Fetch stage, microcodes are fetched from the memory. In the Decode stage, microcodes are decoded and control signals are generated. The address for the stack RAM is also generated in the same stage. In the Execution stage, arithmetic/logic operations are performed. The operands are top two elements of the stack stored in registers *A* and *B* and the result is stored back in register *A*. The GALS-JOP design extends JOP's data-path in an upward compatible fashion.

The additional components introduced in the data-path are: a register file memory, a jump-table, LT, MAX unit, various registers and multiplexers shown as shaded in Figure 4.14. The MAX unit finds out the maximum nibble from the word fed to it. Most of the registers are used to store address information, whereas registers *T1* and *T2* are two general purpose temporary registers used for data manipulation. The memory for CA is allocated on the heap by simply declaring the arrays in Java. The bytecode *jopsys_cabaseaddr*, stores the base address of array in the CAB register, to provide direct access to the CA. Otherwise, each CA access will require expensive read of object reference and CA base address. Loading an immediate value into a register is the most frequently occurring operation. This is implemented by *jopsys_ldrimm*

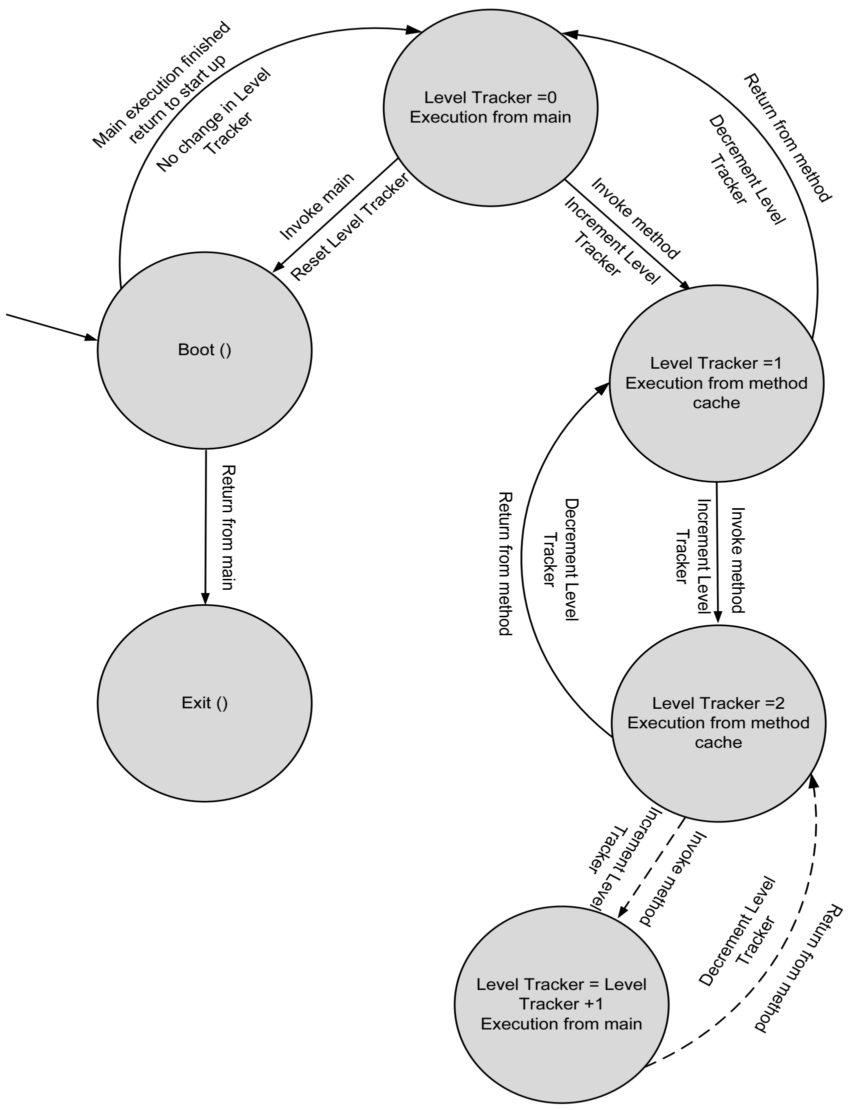


Figure 4.13: GALS-JOP instruction source management scheme

instruction which needs only two cycles for execution. The address of a register in the register file memory and the contents to be written are provided as TOS (top of stack) and TOS-1. The loading of data from the CA to register file is done by using the *jopsys_ldrdir* and *jopsys_ldrind*. All the CA addresses are relative to the base address and thus are added to the base address to find the physical address. The data is stored in CA using *jopsys_strimm*, *jopsys_strind* and *jopsys_strdir*.

In case of jump or branch instructions, the target addresses are available in the jump-table. The index is provided on the TOS as argument which is used to read the required target address into T1. The conditional jumps are implemented through the *jopsys_sz* and *jopsys_present* where target address is loaded to T1 and data to be checked is pushed on TOS. The program counter is loaded with target address from T1 if the condition is satisfied. The *jopsys_switchjump* is the most complex and exorbitant instruction. The pointers to all the cases are stored in the contiguous memory locations in ascending order following the switch node. The address of switch node is read from the register file memory. The switch node value read from the memory contains the number of the case to be executed. This value is added to switch node address and physical address is calculated. The value read from the memory is loaded into the *jpc* to jump to the desired location. The *jopsys_chkend* instruction loads the operands from register file into T1 and T2. The four nibbles in Rx and *msb* nibble of Rz are compared and the maximum nibble is stored in the T1. The GALS-JOP interacts with environment through memory-mapped IOs containing a set of registers. The input signals and *environment ready* signals are loaded from the IO registers using *jopsys_ldio* instruction and contents loaded are directly written into register file.

4.8 Experimental Results

We demonstrate the effectiveness of the proposed architecture by running benchmark examples on it. Although we have large number of benchmark examples, we have selected only those SystemJ benchmarks which have been used in previous implementations. The characteristics of the benchmarks have been described in Chapter 3. The SystemJ codes of benchmarks are given in the Appendix D.

All presented data has been collected from the experiments carried out by using the cycle-accurate *ModelSim* simulator targeting *Altera* Cyclone II FPGA with 70k logic elements, 2MB of RAM and running at 50 MHz clock. The system is capable of running at 100 MHz but the results presented are for 50 MHz clock for fair comparison with earlier published results.

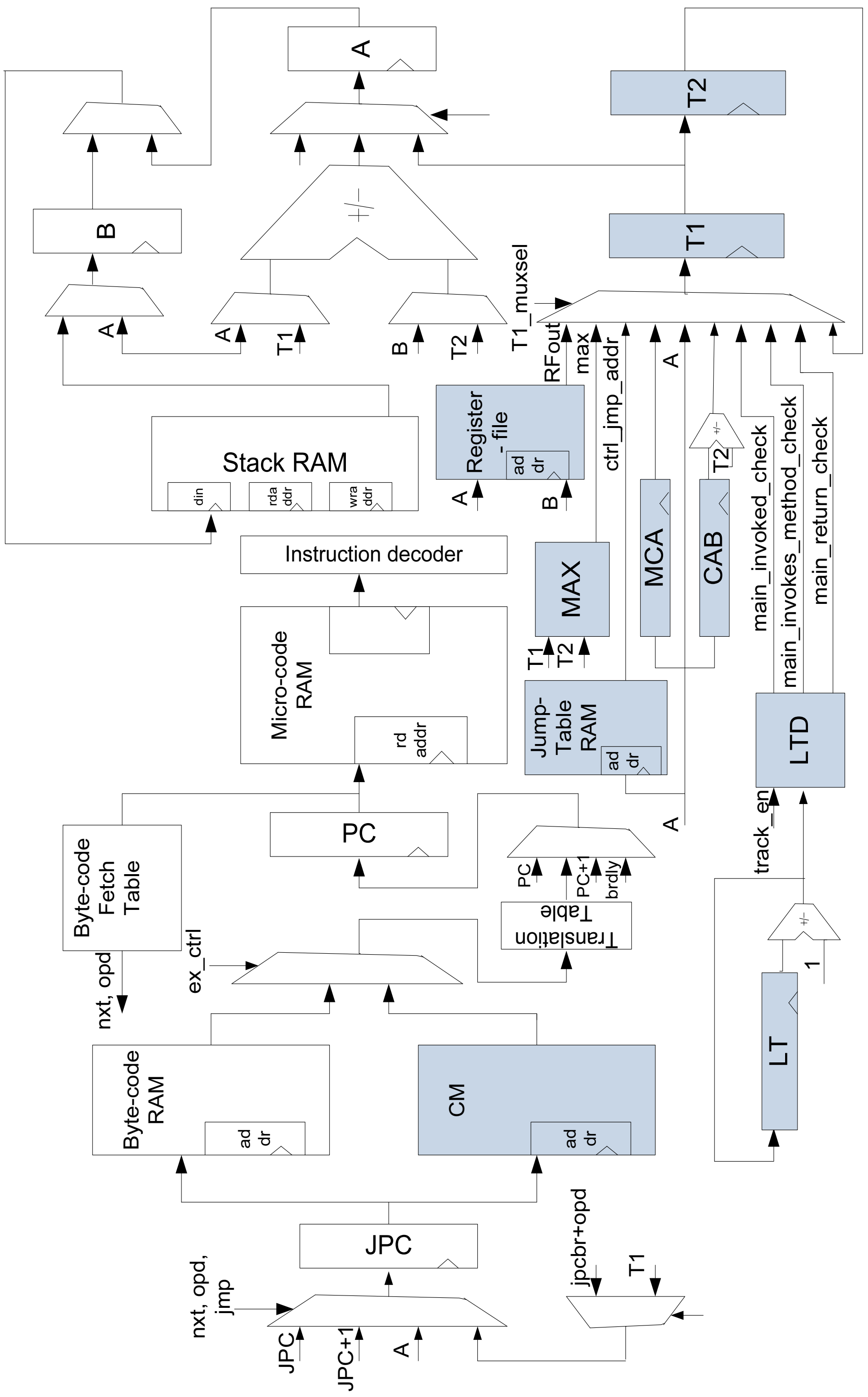
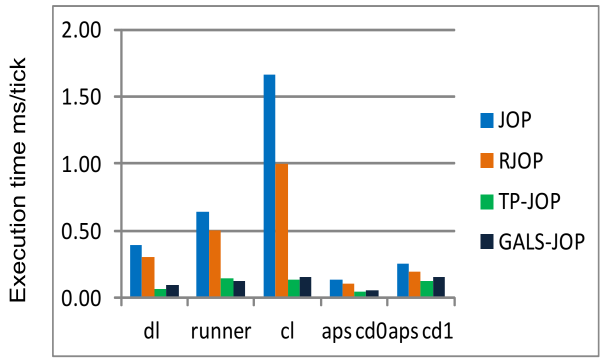


Figure 4.14: GALS-JOP data-path

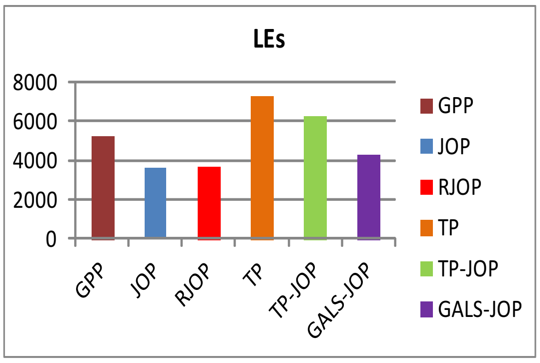
4.9 GALS-JOP Performance Evaluation

The GALS-JOP performance, given in Figure 4.15, is very close to the TP-JOP performance as might be expected. TP-JOP performs better than GALS-JOP due to concurrent and native execution of both control-oriented and data-oriented operations. The TP-JOP should perform way better than GALS-JOP and especially when executing control-dominant applications. The GALS-JOP translates the control-operations represented in assembly code to Java statements representing special bytecodes. The functionality of these bytecodes is implemented through a sequence of microcodes culminating in the execution times that vary between 1 and 20 cycles, which are worse than the 3 to 4 cycles taken by the non-pipelined custom CP to execute CRCF instructions. The GALS-JOP is 2 to 10 times faster than the JOP, 3 to 53 times faster than the TP.

The performance of GALS-JOP being very close to the TP-JOP, despite the fact that it is single processor, can be attributed to a number of factors. First, it can be attributed to the absence of the communication interface, therefore no communication overheads are incurred in terms of time and logic resources. Second, some of the most frequently occurring control instructions are optimized and take fewer cycles than custom CP, thus making the execution of control faster. Third, the data-computations are decomposed into methods instead of wrapping them into *switch-case* statement which is expensive due to inefficient implementations inherited from JOP. These data-computations are now decomposed into small methods (each *case* is wrapped into a method) and need less time for loading them into the cache prior to their execution as compared to a large single method containing all the data-computations. Finally, GALS-JOP efficiently handles return to the *main* method by avoiding its otherwise frequent loading into *method* cache.



(a) GAL-JOP execution time comparison



(b) Resource usage comparison

Figure 4.15: GALS-JOP performance results

Figure 4.15b shows the resource utilization for different target platforms in terms of logic elements executing the code generated by compiler where control (CRCF) is separated from the data-computations (JCF). The TP is the most expensive in terms of real state usage. When compared with the TP and the TP-JOP, GALS-JOP uses 41% and 31% fewer resources, respectively.

The GALS-JOP approach is preferred as it uses a single processor to achieve the same results as TP-JOP (two processor approach) which is more costly. This single processor realization is faster by an order of magnitude when compared to other single processor counterparts such as GPP (NIOS II) and JOP itself. As expected, it is typically slightly slower than TP-JOP, which hustles through code by using significantly more silicon (logic elements when implemented in FPGA).

4.10 Summary

In this chapter, we introduced two new execution platforms for SystemJ GALS programming language. The first one is the JOP based tandem processor, TP-JOP, that uses two processors to execute control (CP) and data dominated Java (JOP) parts of SystemJ programs, respectively. This implementation served as the starting point to arrive to the ultimate goal of a single processor suitable for efficient and prudent execution of SystemJ programs. The GALS-JOP processor merges the features of control processor (CP) into Java processor (JOP) and, so far, represents the compact execution platform for SystemJ. A modified programming and memory allocation model, together with the enhanced instruction set based on the original JOP processor, resulted in very efficient implementation. Also, this implementation preserves all features of the original JOP and all new features built into the GALS-JOP preserve ability to calculate execution times of resulting programs, making possible the analysis of SystemJ programs in terms of the worst case reaction times (WCRTs), where the WCRT is the longest time of any individual tick of a clock-domain within SystemJ program. Those times can be calculated for each clock-domain separately and used in analysis of real-time features of SystemJ programs. GALS-JOP also preserves existing SystemJ compilation and design flow with minimal additions to the back-end of the compiler. Its current implementation is meager in terms of required resources (in FPGA implementation). As GALS-JOP was a starting point, a better refined approach is presented in the Chapter 5.

Chapter 5

JOP-Plus

The focus of this research is to develop an efficient execution platform capable of executing the programs specified and structured in SystemJ, faster by using minimal logic resources at the same time. We obtained the performance improvements in program execution by separating the control flow from data-computation and executing them in parallel on a combination of custom and general purpose or domain specific processors better optimized for the SystemJ based applications. The resulting heterogeneous multiprocessor architectures discussed in Chapters 2 and 3, called TP and TP-JOP, used too many logic resources. In order to overcome the problem of high resource usage, we executed the control flow and data-computations on a single processor instead of using two processors. The success of such an approach was demonstrated in Chapter 4 in the form of GALS-JOP processor. But, this approach required the translation of concurrency and control flow (CRCF) programming model represented as assembly instructions to the data-computation programming model represented as Java statements (also known as JCF - Java control flow). This high level translation resulted in inefficient code generation and needed extra hardware resources in the form of jump-table which could be avoided.

We are proposing a novel approach which does not require the translation of one programming model into another programming model. Instead, it provides support for both programming models in a single processor with two execution modes. Each mode executes one programming models independent of the other. The new processor, called JOP-Plus [97], can be used for embedded and even real-time applications in which the majority of code is written in Java, and the overall programs specified and structured in SystemJ [8] system-level concurrent programming language. The combination of processing attributes enables JOP-Plus to perform equally well in both data-dominated and control-dominated applications-in many cases deleting the requirement for separate heterogeneous processors.

This chapter presents an approach to efficiently mix Java with asynchronous and synchronous concurrency and execute it on a specialized Java processor extended with capabilities for concurrency and reactivity as a separate mode of operation. The background for the proposed approach is described first. Next, execution flow of SystemJ programs is discussed in detail and its effects on JOP-Plus design are given. Finally, the details of JOP-Plus implementation and evaluation of its performance are provided. During this chapter, we will come across different instructions being executed on the same processor. We will use terms *byte-code*, *microcode* and *instructions* while referring to JVM instructions, JOP native instructions and CRCF assembly instructions, respectively.

5.1 CRCF Instruction Set Architecture

CRCF program model shares the instruction set architecture (ISA) with the CVM and ReCOP but in a slightly different way. The CVM and ReCOP had 23 instructions with varying addressing modes. The complete instruction set of ReCOP along with its precise description is provided in Appendix A. The JOP-Plus has 33 instructions and each instruction has one of the following addressing modes: *immediate*, *direct*, *indirect* or *inherent*. The CRCF ISA specifies the size of CRCF program memory as 64K words, 16 registers and 16-bit instruction-word. CRCF instruction set has a variable instruction length i.e., a single word or two words. The instructions that require a target address or immediate value make use of the second word. The CRCF instructions has a single encoding format as shown in Figure 5.1. The instruction is broken up into fields of the different sizes. The first 8-bits are used to represent the opcode; currently CRCF utilizes 33 out of 256 possible opcode values. The next 8-bits are used as references to the two registers Rz and Rx (4-bits each one) in the Register-File.

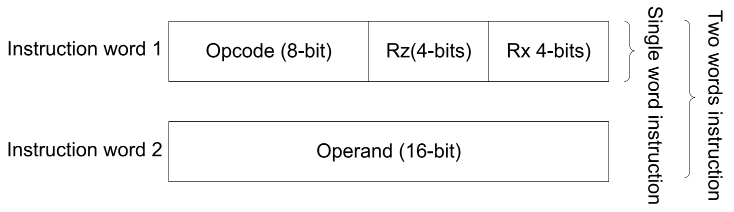


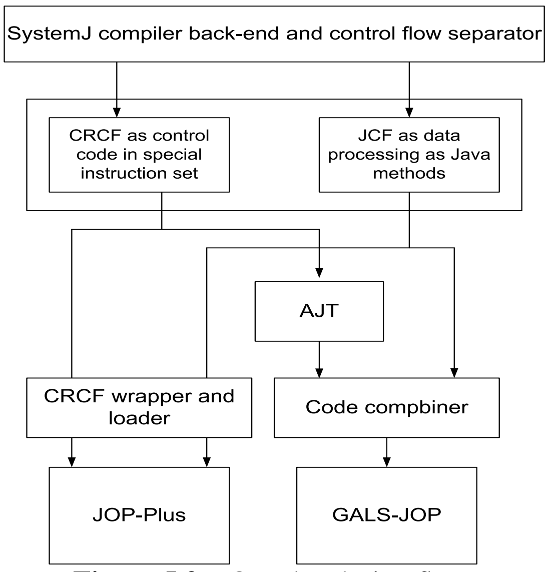
Figure 5.1: CRCF instruction format

The instructions can be organized into different categories such as arithmetic-logic, transfer between memory and registers, transfer between environment and registers, flow control and special purpose instructions.

5.2 Improving GALS-JOP Approach

Although, JOP [93] itself as a target is possible but resulting performance is fairly poor. Some improvement of performance was achieved with Reactive-JOP (RJOP), where reactivity was directly supported by hardware [91]. However, the separation of CRCF and JCF offers a larger space of SystemJ [8] execution strategies. Both TP-JOP, based on the idea of tandem processor execution, and GALS-JOP have demonstrated advantages of separation of control flows as they have different execution patterns and requirements. While, TP-JOP uses two processors (JOP and ReCOP) to implement the two control flows (CRCF and JCF), the GALS-JOP extends JOP [93] to provide hardware support for CRCF code execution along with Java code execution. This sophisticated merging of CRCF and JCF execution results in efficient implementation. The later requires some further modifications of the back-end of the compiler. The GALS-JOP [95] approach produces performance results close to the existing TP-JOP platform for executing SystemJ programs while using far few logic resources. But, this approach has the following short-comings which could be refined and improved:

- *Need of AJT tools:* GALS-JOP processor cannot execute the code generated by the SystemJ compiler back-end as such therefore, needs modifications. The modifications introduced involve a translation tool, called Assembly to Java Translator (AJT), which translates the CRCF assembly instructions to Java statements as shown in Figure 5.2. The JOP is unable to interpret the control flow given in assembly code; therefore, translation of assembly code to Java statements is necessary for its realization. If we could devise a mechanism to support control flow directly on the Java processor, it will make the AJT tool redundant and can be avoided.


SystemJ compiler back-end and control flow separator
CRCF as control code in special instruction set
JCF as data processing as Java methods
AJT
CRCF wrapper and loader
Code compbiner
JOP-Plus
GALS-JOP
Figure 5.2: JOP-Plus design flow

- *Inefficient translation:* The translation of control flow (CRCF) given in assembly instructions to Java statements is inefficient. When translating assembly code to Java, the information provided by different instruction fields, except opcode, is passed as arguments. Upon compilation, the Java statements carrying the arguments produce multiple

Java byte-codes which push the operands on the stack and perform the required operation. For example, an ALU instruction with immediate mode requires the immediate value, operand source register address, destination register address and ALU operation information to be passed as arguments as described in Chapter 4. This Java statement is compiled to 5 byte-codes majority of which just push the arguments on top of the stack as shown in the Table 5.1.

CRCF Assembly	**Translation to Java**	**Byte-codes generated upon compilation**
AND R0 #0	*Native.aluimm(0,rand,0,0x0)*	*iconst_0* *iload* *iconst_0* *iconst_0* *jopsys_aluimm*

Table 5.1: CRCF assembly instruction translated to Java and compiled to bytecodes

- *Inefficient execution:* During the execution, every Java byte-code is translated to either one, or a sequence of microcode instructions. Each of this byte-code requires one to multiple clock cycles for execution depending on whether the argument is a constant or fetched from external main memory. Furthermore, the data structure of CRCF is stored as an array in heap area of main memory situated external to the core. CRCF instructions perform operations on this data structure and frequently access it. The slow main memory has high access latency resulting in reduced instruction throughput.

- *Constraints on available byte-codes:* Due to constraints on available byte-codes, a number of assembly instructions are mapped to a single more general byte-code as shown in Table 5.2. This flexibility is achieved at the cost of performance.

CRCF Assembly	**Translation to Java statement**
AND Immediate *ADD Immediate* *XOR Immediate* *SUB Immediate* *OR Immediate*	*jopsys_aluimm*

Table 5.2: Mapping multiple assembly instruction to a single bytecode

- *Jump-table:* GALS-JOP requires an extra on-chip memory in the form of a jump-table to implement the control flow. The on-chip embedded memories are expensive and add to the cost of the system.

We present an integral attempt to address the concurrency and reactivity of SystemJ by using existing JOP in a novel way culminating in a new processor named JOP-Plus [97]. Instead of translating the CRCF assembly code to Java byte-codes, the support for assembly program model is provided thus completely eliminating the need of the AJT tool. Also, it is not required to pass any information as argument during translation to Java as the CRCF instruction set is preserved and information is available as part of the instruction. It has an efficient way to invoke a method and return from the method without using the Level Tracker (LT) and its associated logic as mentioned in Chapter 4.

5.3 JOP-Plus - A Refined Approach

This work combines and extends the ideas of TP-JOP and GALS-JOP onto a new processor core JOP-Plus. The JOP-Plus processor uses JOP as its base, executes concurrent programs that comply with Globally Asynchronous Locally Synchronous (GALS) formal model of computation compiled in a way clearly distinguishing between concurrency and reactivity control flow (CRCF) and Java control flow (JCF). The main idea is to provide support for the two separate components of the control flow in SystemJ programs, CRCF and JCF separated during compilation, by extending the instruction set of the original JOP while using single execution unit and data-path.

It works by introducing certain new components such as Register-File (RF), MAX unit and registers but not duplicating the main execution resources of JOP. This allows resultant JOP-Plus core to appear as two "logical" processors allowing it to execute two different program models. At any given time, the programming model under execution uses all the resources of processor, and only the current programming model can invoke the other programming model. In this process it switches the programming mode resulting in suspension of current program model and the resources become available for the other programming model. The program models do have independent execution, but they are called by each other. These control flows can be supported in the execution platform by extending JOP. The separation of the CRCF and JCF is maintained by storing those two parts of SystemJ program in two different memories. The execution capabilities of JOP are extended with a number of new byte-codes, as well as with the microcodes needed for their implementation. The new core has single instruction decoding and execution unit and performs very efficient switching between CRCF and JCF when necessary. The SystemJ program execution is guided by the CRCF, which in turn activates JCF whenever Java action nodes need to be executed. When JCF execution is complete, it returns to the CRCF. At the same time the existing compiler does not require any major modifications as it just needs to organize the code in a way which is compatible with JOP-Plus. This is done with the help of a simple tool which processes the information generated by compiler and assembler.

Furthermore, this approach deploys a very efficient technique to call the data-computations which is different from TP-JOP and GALS-JOP. In case of TP-JOP, each processor supports one of the two different programming models; CP supports assembly program model and JOP supports Java program model. The data-computations are performed by the JOP which decodes the information sent by the CP to select the required computation. In case of GALS-JOP, we translate one programming model into other and then data-computations enclosed inside a method are invoked in conventional way. The JOP-Plus is unique as it does not run both models at the same time and does not need to pass information to call data-computation like TP-JOP. Furthermore, it does not need any translation tool like GALS-JOP to translate CRCF assembly code to JCF format i,e,. Java. The JOP-Plus is capable of executing both programming model and, while being in one programming mode, it can directly invoke the required data-computation given in the different programming model. Hence, invoking a method containing data-computation is accompanied by a program model invocation. This seamless integration completely eliminates the need of translator and communication medium by executing them on a single component in time multiplexed manner based on demand with CRCF leading the execution. The execution pattern of JOP-Plus at any point of time is shown in Figure 5.3

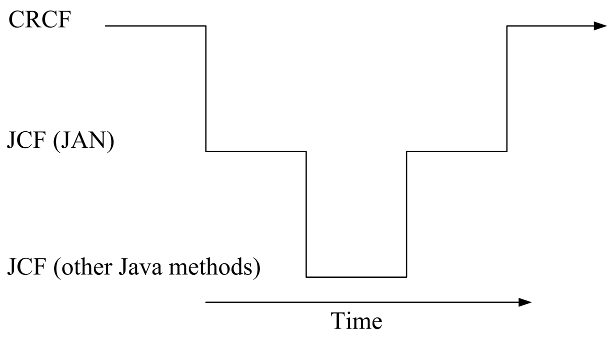
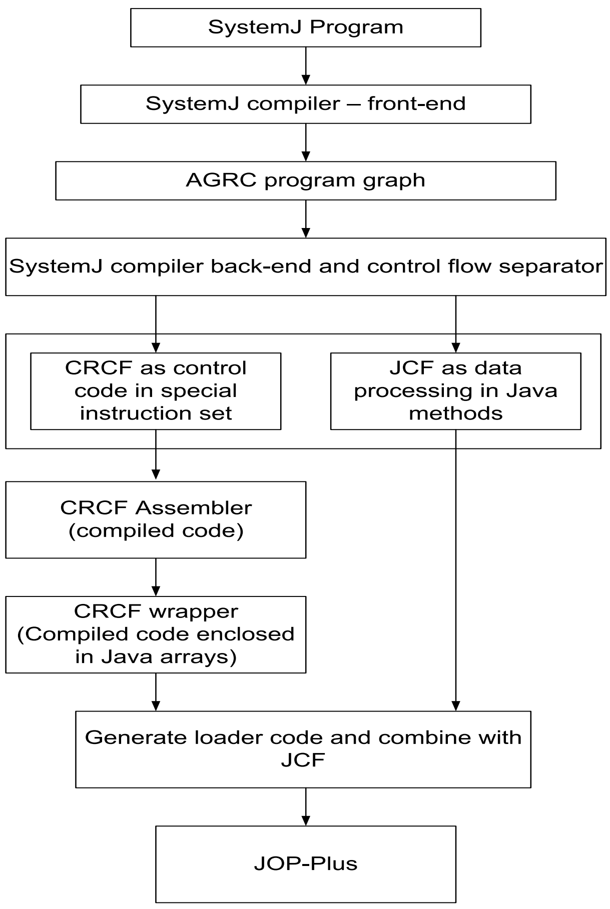


Figure 5.3: Control flow of SystemJ program

5.4 Compilation and Execution Strategy

Front-end of the SystemJ compiler [38] transforms a SystemJ program to an intermediate representation called Asynchronous Graph Code (AGRC) from which back-end of the compiler can target different execution platforms. Figure 5.4 illustrates compilation and execution strategies of interest for the JOP-Plus as the target execution platform. The SystemJ back-end separates the concurrency and reactivity control flow from the Java control flow. The CRCF is generated as assembly code and JCF is generated as Java code. The JCF code consists of variable declaration and methods implementing the data-computations corresponding to Java action nodes (JAN). The CRCF assembly code is assembled using ReCOP assembler producing machine code. This machine code, represented as *hex* values, is enclosed in an array and stored in the heap section of main memory from where it is downloaded into CRCF program memory during start-up. Java code performing this task is called *CRCF loader* code. We generate a Java class which contains both the JCF code and *CRCF loader* code with later being the part of *main* method. This Java class is compiled using *javac* compiler and the resulting class file is processed by *JOPizer* [93] tool to convert to a format compatible with the JOP-Plus execution platform. On start-up, JOP is initialized and *main* method is invoked which downloads the compiled code and switches to the CRCF program execution. We intend to remove the CRCF loader code in future by directly downloading the *hex* code into the CRCF memory during the boot up which will help in reducing the size of application.


SystemJ Program
SystemJ compiler – front-end
AGRC program graph
SystemJ compiler back-end and control flow separator
CRCF as control code in special instruction set
JCF as data processing in Java methods
CRCF Assembler (compiled code)
CRCF wrapper (Compiled code enclosed in Java arrays)
Generate loader code and combine with JCF
JOP-Plus
Figure 5.4: JOP-Plus compilation and execution strategy

5.5 JOP-Plus Significant Features

The JOP-Plus is a refined approach with some clear advantages over the previously adopted strategies. Some of more important features of the JOP-Plus are discussed next.

5.5.1 CRCF Memory Concept

Mapping SystemJ programs [38] onto CRCF and JCF is followed by the code generation which uses a special instruction set for CRCF and standard Java for JCF. Instead of fitting the special CRCF instructions into limited number of unused bytecodes as was the case with GALS-JOP, we preserve the instruction set. They are compiled and stored into a separate program memory, called CRCF program memory, which is in addition to the main memory where JCF code resides. Now there are two instruction sources: the CRCF program memory and the *method* cache which holds JCF code loaded from main memory prior to its execution. The instruction can be fetched from either of the two sources depending on the working mode of the processor. JOP-Plus execution unit is capable of decoding these instructions and execute them using microcodes stored in microcode ROM.

5.5.2 Preserving SystemJ Compiler

The JOP-Plus is similar to the GALS-JOP in the sense that both use a single processor approach to execute the SystemJ programs but JOP-Plus has an edge over the GALS-JOP as it does not require the translation of control flow to Java prior to its execution. The JCF is still broken into small methods as was the case with GALS-JOP [95]. JOP-Plus does not require any major modification in SystemJ compiler. All it needs is to generate the code organized in a way suitable for execution on JOP-Plus. This is achieved by changing the compiler back-end (only few lines of code) or by using a simple post processor tool. The later approach is preferred and adopted.

5.5.3 Two Virtual Processors on a Single Physical Processor

The JOP-Plus implementation can be considered a custom processor as the instruction set and data path have been tailored exclusively for the processing of SystemJ programs. Actually, our concept embraces two integrated execution modes; one Java mode and one ReCOP [39] mode. We have implemented a processor core where two virtual processors share one datapath. The processors may operate independently although not in parallel. We do not rely on translations or extensions to the Java format, but offer two pure programming models. The two programming models offer two logical views of the processor, a) Java programming model and b) CRCF programming model. Both programming models are orthogonal each to the other (in this sense: mutually exclusive upon operation). The user may exploit only the pure Java or CRCF programming model. This system can also be described as a system with two running modes. The switching between the execution modes takes place at the byte-code boundaries and is similar to invoke and return from a method.

5.5.4 Efficient Implementation of Data-computations

As discussed in Chapter 4, wrapping up of data-computations in *switch-case* statements was not efficient due to the implementation approach adopted by the original JOP to handle them. The data-computations are broken into small Java methods resulting in reduced time to load them into *method* cache. These methods are invoked directly by the CRCF without incurring any other overheads like the TP-JOP.

5.5.5 No Jump-table

Since Java does not have *goto* mechanism, this mechanism was provided and supported using the jump-table in the GALS-JOP. The JOP-Plus uses more efficient strategy to implement the jumps without deploying jump-tables. All instructions which affect the control flow of program contain the target address information in a register or as immediate value in the instruction. Since, we preserve CRCF instruction set in JOP-Plus, this information is available as a part of the instruction. The CRCF instructions modifying the control flow of the program and having target address as part of instruction are shown in Table 5.3.

CRCF Assembly	**Branch**	**Bytecodes generated upon compilation**
PRESENT	conditional	Opcode | Rz | Rx Target address
SZ	conditional	
JMP_IMMEDIATE	unconditional	

Table 5.3: Summary of branch instructions

5.5.6 Efficient implementation of CRCF Data Structure

In GALS-JOP approach, the CRCF data structure was implemented as an array object stored in heap located in external memory and, being further away from the core, it is more expensive to access this memory. CRCF instructions operate on this data structure and frequently access the elements of data structure. Since, CRCF data structure occupies only a fraction of CRCF program memory; therefore, in JOP-Plus it is implemented in the same memory as CRCF program memory which is situated locally to the core. The memory can be accessed for fetching instruction and read/write data without any conflict as the access to data structure and program memory is orthogonal. The details of data-structure implementation are given in Section 5.7.4.

5.5.7 Efficient Invoking of JCF

The JCF methods are directly invoked by the CRCF and they return to CRCF after finishing the execution. They return without the need of reloading *method* cache with the caller method. This is performed without introducing extra hardware resources except the CRCF memory.

5.5.8 Resource Sharing

The aggressive resource sharing between the CRCF execution mode and Java execution mode produces area-efficient and competitive design. It keeps the physical distance between the components small and reduced interconnect structure decreases both cost and complexity.

5.6 SystemJ Code Example

Figure 5.5 shows the SystemJ example code and generated code targeting JOP-Plus. The SystemJ example code presented in Chapter 4 is repeated in Figure 5.5a for readability with slight modification as it prints different message now. The assembly code for CRCF and Java codes for JCF both generated by SystemJ compiler, are shown in Figure 5.5b and Figure 5.5c, respectively. The functionality of example program presented has already been discussed in Chapter 4, therefore, details are omitted here to avoid repetition.

The Java code shown in Figure 5.5c consists of *main* method (line $15 - 23$) and other JCF methods (line $5 - 13$). The *main* method comprises of the Java code for initializing the CRCF program memory, referred as *loader code*, prints message (line 24) and a *native* method. *Native.switchmode*, represents a special bytecode and shifts the control of execution from JCF to CRCF execution mode. The execution of this bytecode results in fetching of instruction from CRCF program memory instead of the bytecode from *method* cache. The JCF methods, *method_0*, *method_1* and *method_2*, contain the data-computations represented by Java action nodes. These methods appear in the Java class file in the ascending order with respect to their *id* for each clock-domain. This helps in calculating the address of the data structure of the JCF method to be invoked stored in the main memory. The JCF method code compiled to Java bytecodes resides in the method area of the main memory and loaded into *method* cache for execution upon invocation by CRCF.

```
// SystemJ example code
system {
      interface {

        output signal RE,TE;
    }
    {
        {
           signal pipi;// localsignal
           {
           present(pipi) {
               emit RE;
               System.out.println("JOP-Plus");
           }
           emit TE;
           int u=5;
           pause;
        }
        ||
        {
           emit pipi;
           System.out.println(u);
           pause;
        }
     }
     ><
     {

     }
   }
}
```

(a) SystemJ example code

```
// Assembly code for CRCF
L1 SEOT
   CER
   LDR R0 $0001
   AND R0 R0 #$f000
   SSOP R0
   LSIP R0
   AND R0 R0 #$0
...
   LDR R1 $0002
   AND R1 R1 #$0001
   PRESENT R1 L4
   LDR R0 #3
   ADD R1 R6 #0
   STR R1 #0
   SENDATA R0
   ADD R4 R4 #1
 L2 LDR R0 R1
   CLFZ
   SUBV R0 R0  #0
   SZ L3
   JMP L7
   L3 CLFZ  ;
...
```

(b) Assembly code for CRCF

```
//Java code for JCF
public class Jopplus{
  public static int u=0;
  ...
  private static method_0(){
    System.out.println("JOP-Plus");
  }

  private static method_1(){
    u=5;
  }
  private static method_2(){
    System.out.println (u);
  }
  public static void main() {
   crcf code = new crcf();
   int length = code.cc.length;
   for (i=0; i < length; i++){
    crcf_instr=code.cc[addr];
    Native.initctrl(addr, crcf_instr);
    addr=addr+1;
    }
    ...
   System.out.println("CRCF initialized!");
   Native.switchmode();
  }
}
```

(c) Java code for JCF targeting JOP-Plus

Figure 5.5: SystemJ compilation example separating CRCF from JCF with data-computations decomposed into methods targeting JOP-Plus

The CRCF assembly code produced as the output of the compiler targeting ReCOP, deployed in TP and TP-JOP, is used without any modifications as shown in Figure 5.5b. Each instruction word is 16-bit and some instructions may have up-to two words. The precise details of control instructions set has been provided in Appendix A.

The assembly code is compiled to the hex format and enclosed in an array object which is stored in the heap area of the main memory. The array elements, initialized with the compiled hex code, are declared inside a class as shown in Figure 5.6. The array comprising of CRCF compiled code is not declared in the *main* method as the size of the compiled program gets too large (of the order of many KB), resulting in large *main* method size. The JOP [93] does not allow the oversized methods due to restricted *method* cache size. Therefore, the program is fragmented and placed in large number of relatively small arrays to keep the method size less than the 4KB as this is the default configuration used in our experimental set-ups throughout the research.

```
//crcf.java

public class crcf{
 int a;
 public static int[] cc = {
  0x3400,0x4100,0x0000,0x4090,
  0xFFFA,0x40A0,0xFFFF,0x4070,
  ...
 }
}
```

Figure 5.6: Java array object initialized with CRCF compiled code

5.7 Memory Organization

The memory organization of JOP-Plus is illustrated in Figure 5.7. Besides the new memories introduced shown as shaded, JOP-Plus has a number of memories inherited from JOP such as main memory, *method* cache, stack cache, microcode ROM, and translation-table. The new memories introduced include Register-File, CRCF program memory, CRCF data memory and a number of registers. The JOP-Plus has improved memory organization as compared to GALS-JOP as it does not need jump-table to implement jumps as the target addresses are available as part of the CRCF branch instructions. The description of the main memory and CRCF specific memories is given next.

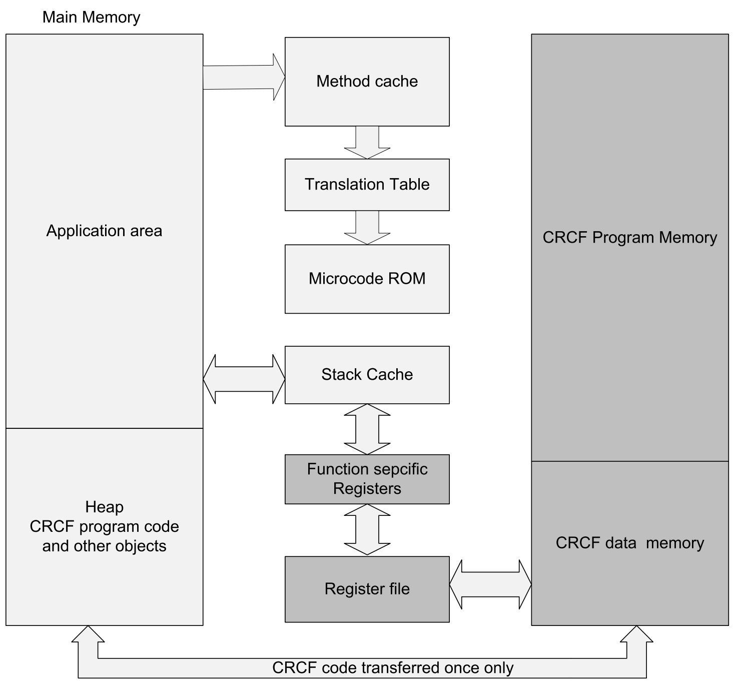


Figure 5.7: JOP-Plus memory organization

5.7.1 Main Memory

The main memory can be divided mainly into two parts: 1) application area and 2) heap. The application area consists of per-class structures such as run-time constant pool, field and method data, and the code for methods and constructors, as well as interned Strings. The heap is a run-time data area from which memory for all class instances and arrays are allocated. The detailed organization of main memory is given in [93].

The SystemJ program [8] starts as Java program, which is compiled and initially loaded into the main memory. The size of the Java application is given in the very first word of the memory location which also marks the start address of heap as shown in Figure 5.8. Second word in the memory is the address to s*pecial pointers* which are used in initializing the JOP-Plus and invoking the *main* method. The *static fields*, *static reference fields* of classes, codes for all the methods and addresses to method structures are stored next in the same order. The information stored in the structure is used to invoke a method by fetching the method code into *method* cache and pushing the arguments in newly created method frame in the stack cache. Once the application size is found, the rest of the main memory space is regarded as heap, which is used to store all Java objects including arrays, SystemJ valued signals and channels. The memory space used by inactive objects is reclaimed by garbage collector. The size of main memory is 2 M-byte for the configuration used in the experiments.

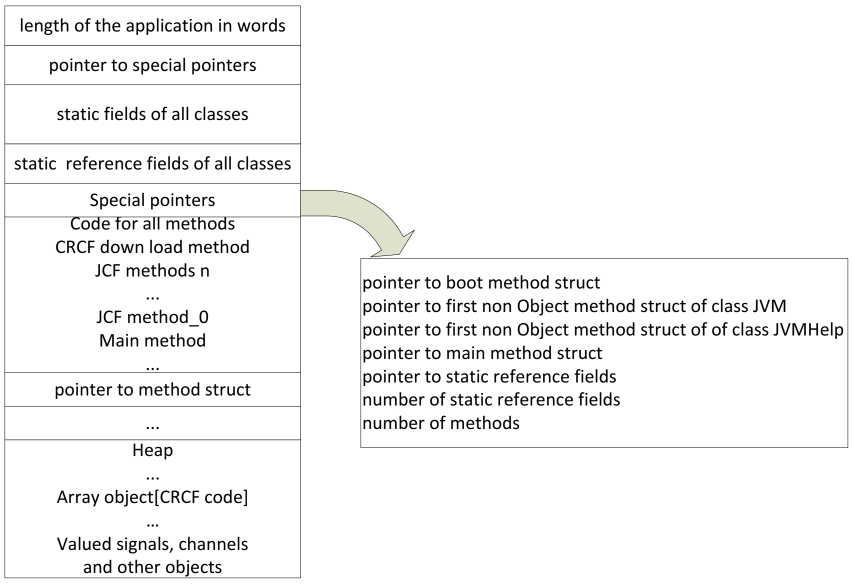

length of the application in words
pointer to special pointers
static fields of all classes
static reference fields of all classes
Special pointers
Code for all methods
CRCF down load method
JCF methods n
...
JCF method_0
Main method
...
pointer to method struct
...
Heap
...
Array object[CRCF code]
...
Valued signals, channels
and other objects
pointer to boot method struct
pointer to first non Object method struct of class JVM
pointer to first non Object method struct of of class JVMHelp
pointer to main method struct
pointer to static reference fields
number of static reference fields
number of methods

Figure 5.8: Main memory organization

5.7.2 CRCF Program Memory

The program memory of the CRCF holds the compiled and assembled code implementing concurrency and control flow of a SystemJ program. The memory is 16-bit wide RAM, which is enough to store an instruction without operand. The memory depth is parametrized depending upon the code size but maximum size is limited to 64K, the maximum number addressed by a 16-bit program counter. There are two sources of program instructions in JOP-Plus: CRCF program memory and *method* cache. Therefore, CRCF memory shares the program counter register with the *method* cache. The *method* cache serves as the instruction cache for JCF. The full code of a method is loaded into the cache before execution.

5.7.3 Register-file

In our architecture, the Java and CRCF modes have different temporary storage area for the operands. The Register-File (RF) is used for temporary storage of the operands when executing CRCF and is functionally the same as the Register-File in TP [39] and TP-JOP [95]. The Register-File cannot be viewed by the Java mode. The JOP-Plus architecture supports a flat Register-File implemented in an on-chip multi-port memory with single cycle access latency as shown in Figure 5.9. The Register-File is organized as 16x16 bit registers. These registers can be used for data storage, arithmetic operations, logic operations or address registers for accessing data memory locations.

The RF has a multiplexer at the input which is used to select one of the two data sources, the 16-bit operand of the *crcfopdreg* and top element of stack in A. The RF is implemented as an array of sixteen 16-bit signals, initialized to 0. The read ports Rx and Rz are asynchronous while the write port z is synchronized with the system clock. Data is written in the Register-File on the rising edge of the system clock.

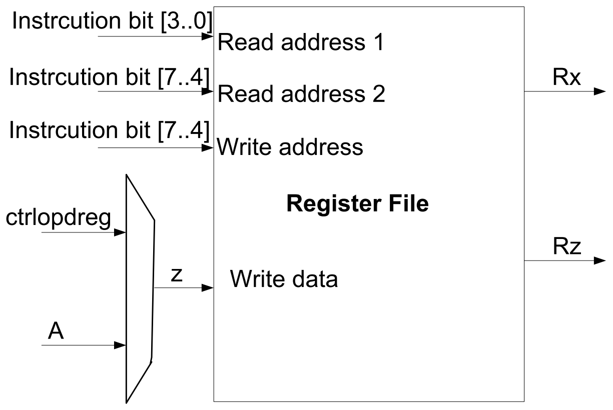


Figure 5.9: CRCF Register-File with data input mux

5.7.4 CRCF Data Memory

The CRCF data memory (DM) contains the data structure for the concurrency and control flow statements of the SystemJ program (CRCF). The data memory layout for JOP-Plus is shown in Figure 5.10. The DM is 16-bits wide and the depth is not fixed and maximum size depends on the number of clock-domains. The size of data memory increases with the increase in number of clock-domains, reactions and control statements. The CRCF instructions operate on data structure elements by fetching them into the Register-File, performs operation on them, and stores them back into the data memory. In case of GALS-JOP, data structure for CRCF is implemented as control arrays (CA) in the main memory. As CRCF instructions operate on this data structure, therefore, it requires frequent access to the main memory. The main memory is located off-chip, and access time is long compared to on-chip memory due to the limited bandwidth, resulting in slow execution of CRCF. It has been observed that CRCF data memory is only a fraction of the CRCF program memory. Hence, the CRCF data structure can be stored in the same physical memory as CRCF code or different physical on-chip memory to improve average data transfer performance. The CRCF data structure has already been described in the Chapter 2 and repeated here again.

The input/output signals statuses (1-bit each) are stored first in the data memory and are word ($16-$bit) aligned, thus if we have 16 or less signals we use at least one memory space (word) in the DM. The internal signals are stored next. Internal signals can be emitted from multiple synchronous reactions in a given clock-domain and thus we assign 1-bit lock status for each signal per synchronous reaction. Next, data-locks are stored which inform if the data calls made for data-computation (Java action node in AGRC) to the JVM have been returned. A complete DM word (16-bits) is used for data-locks and program counters (PC) for the various synchronous reactions in a clock-domain which are stored in the following locations. Then termination codes of reactions, four-bits each, are stored. Switch nodes used for state selection of the currently executing SystemJ program are stored next together with the children. This arrangement of DM is repeated for each clock-domain [38].

The JOP-Plus architecture supports tightly-coupled memory for both CRCF instruction and CRCF data to provide guaranteed low, fixed latency access.

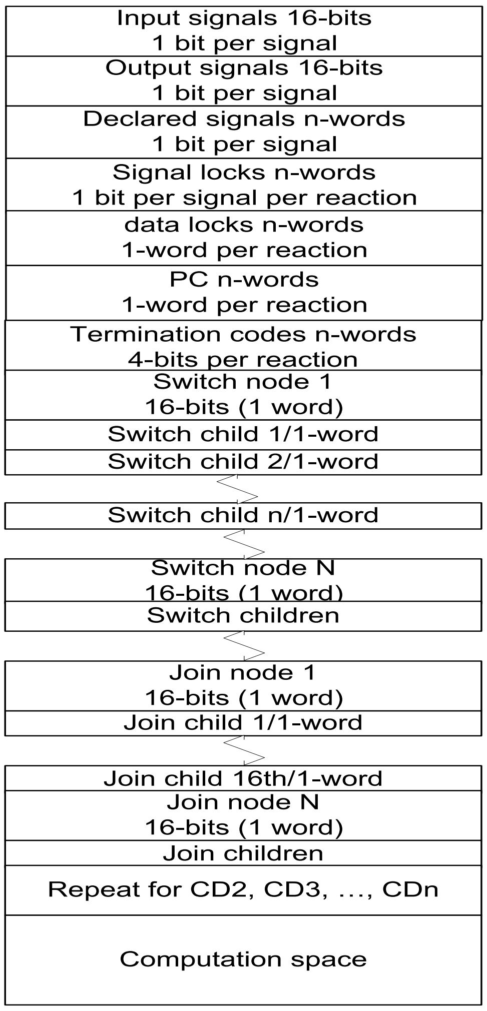


Figure 5.10: CRCF data memory

5.7.5 Registers

The JOP-Plus architecture extends the JOP with a number of function specific registers. These special registers are closely tied to some CRCF execution and they are writable only by special instructions. Some of the registers are used to store the values that are accessed repeatedly and frequently. The temporary register T is the most important register which acts as an interface between the two programming models. The contents of CRCF execution specific registers are placed into this register before passing them to execution unit through stack and fetched into this register from stack after execution. The temporary register could be avoided by implementing one of the following techniques:

- Feed the operands directly to ALU bypassing the stack registers
- Directly push the operands onto stack instead
- Load operand into T and then push onto stack

In first two cases, a large multiplexer is required in the critical path of the processor degrading the operating frequency of the system. In the second approach, we need to introduce more microcodes to push the on the stack. The third approach does not affect critical path and adds only one more input to the existing multiplexer. Therefore, this approach is preferred and implemented in JOP-Plus. The multiplexer at the input of register T is used to select one of the seven data sources: the 16-bit operand of the CRCF, Rz register, Rx register, the CRCF data memory output, the Rx output, the ALU output, the MAX unit output, *crcfpcreg*, environment ready (ER) register extended with zeros, signal input (SIP) register and the CRCF target address. Some of the important inputs to this multiplexer are shown in Figure 5.15. The description of all the function specific registers for CRCF program mode is given in Table 5.4. The *EOT* register, *ER* register, the Signal Output Port register (*SOPREG*), the Signal Value Output Port register (*SVOPREG*) and the Signal Input Port register (*SIPREG*) are used to communicate with the environment. For *SOPREG* and *SIPREG*, each bit indicates if a signal is present, and for *SVOPREG* each bit indicates if the signal is valued. The *ER* register indicates if the environment has finished reading signal values/statuses. The EOT regsiter indicates the end of tick to the environment. Both *EOT* and the *ER* registers (1-bit each) are used as control registers while communicating with the environment. These registers are 16-bits each, thus we are limited to 16 output/input signal declarations. All the registers communicating with the environment are connected to memory-mapped IOs. The *mode_control* is a 1-bit register indicating the current mode of operation. It is reset when JOP-Plus first starts execution in JCF mode and toggles whenever JOP-Plus switches the execution mode.

5.8 Start-up and Control Flow

The start-up and execution flow of JOP-Plus is shown in Figure 5.11. The start up is similar to JOP [93], therefore, the steps common to both are omitted to avoid repetition. Upon start up, the JOP-Plus will boot in Java mode and invoke the *main* method. The *main* method is loaded into the *method* cache and execution starts in Java mode. After initializing the CRCF program memory, it switches from Java mode to CRCF mode and starts executing the CRCF program.

Register	**Description**
T	*This is the temporary register and is used to communicate between stack and CRCF specific components.*
crcfpcreg	*This registers holds the address of the instruction following the CRCF instruction making a data call to JCF. On return from JCF, the content of this register are loaded into Java program counter. This register is initialized to zero at start up.*
jcfbasereg	*This register hold the pointer to method data structure with JAN_ID=0*
crcfopdreg	*This register holds the second word of instruction which is an operand, if required*
zfreg	*It is 1-bit flag using for branching in CRCF.*
mode_control	*This 1-bit register is used to select between the instruction from CRCF memory or Java bytecode from method cache.*
EOT	*This 1-bit register is used for communication with the environment and indicates the end of tick.*
ER	*This 1-bit register indicating if environment has finished reading signal values/statuses.*
SIPREG	*This 16-bit register holds the status of the input signals from the environment.*
SOPREG	*This 16-bit register holds the status of the output signals for the environment.*
SVOPREG	*This 32-bit register holds the status of the valued signals to/from the environment.*

Table 5.4: CRCF execution specific registers

The CRCF leads the execution of SystemJ program and calls JCF when required. When executing Java action nodes (JAN) in JCF mode, the program can call other Java methods or return to the CRCF. These two transitions are implemented in different ways. While transitions between Java methods are using standard Java mechanisms (invoke and return), transitions between CRCF and JCF are performed with the support of hardware as explained in Section 5.12.

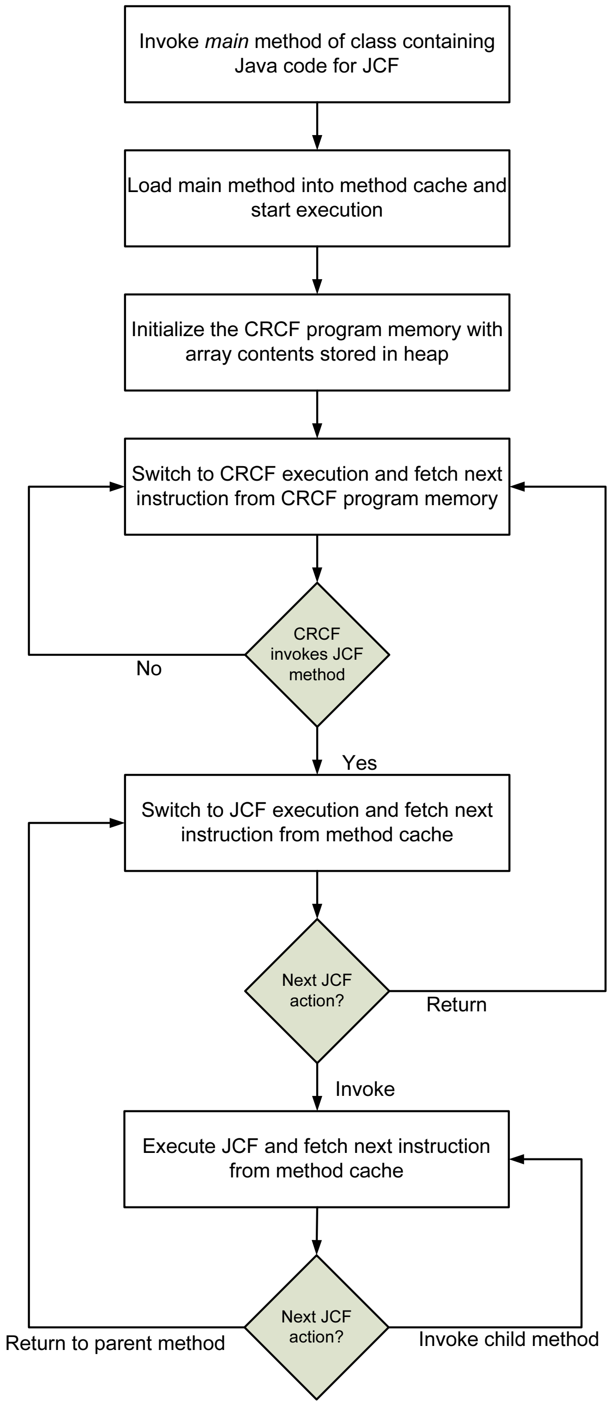

Invoke *main* method of class containing Java code for JCF
Load main method into method cache and start execution
Initialize the CRCF program memory with array contents stored in heap
Switch to CRCF execution and fetch next instruction from CRCF program memory
CRCF invokes JCF method
No
Yes
Switch to JCF execution and fetch next instruction from method cache
Next JCF action?
Return
Invoke
Execute JCF and fetch next instruction from method cache
Next JCF action?
Return to parent method
Invoke child method

Figure 5.11: Start up and program execution flow

5.8.1 CRCF Program Memory Initialization

The initialization of CRCF program memory is performed during the execution of *main* method which is considered part of JOP-Plus start-up therefore JOP-Plus has extended start-up procedure of JOP. The *main* method contains the Java code for initializing the CRCF program memory and switching the mode of processor from JCF execution to CRCF execution. The CRCF compiled assembly code is enclosed in arrays and stored in the heap, located in main memory. The array elements are read from the main memory one by one and stored in the CRCF program memory in the contiguous memory locations. The program memory contents are written through *Native.stcrcf(addr, crcf_instr)* method, representing the new special bytecode. The address and the instruction code are passed as arguments. They are available as top two elements of stack and the instruction code is stored at a memory location pointed to by the address.

Once the CRCF program memory is initialized, the array objects become dead objects as they will never be accessible by the application but have not been collected yet by the garbage collector. The heap memory occupied by dead objects is collected by the garbage collector. The *main* method performs only two major tasks: 1) initializes the CRCF program memory and 2) shifts the program flow to CRCF execution mode.

5.8.2 Instruction Fetch

The instruction fetch mechanism for the JOP-Plus is shown in Figure 5.12 as it has two different program sources. The program memory source for the next instruction to be fetched is controlled through *mode_control* flag, and defines the mode of operation. After initializing of the CRCF program memory, *Native.switchmode( )* statement in the *main* method is executed which sets the *mode_control* flag. When *mode_control* flag is set, the next instruction to be executed is always fetched from the CRCF program memory. On the other hand, the *mode_control* flag is reset when the CRCF instruction directly invokes JCF method and results in fetching of the next instruction to be executed from the *method* cache which holds the *method* code fetched from the main memory. During the return from the JCF method, the *mode_control* flag is again set and next instruction to be executed is fetched from the CRCF program memory. Both memories share the program counter (*jpc*) which is incremented on each bytecode/instruction execution. The *mode_control* bit is concatenated with the bytecode value or instruction opcode to form address of the translation-table. The corresponding entry, pointer to start address of bytecode implementation in microcode ROM, is read to fetch the microcodes for execution.

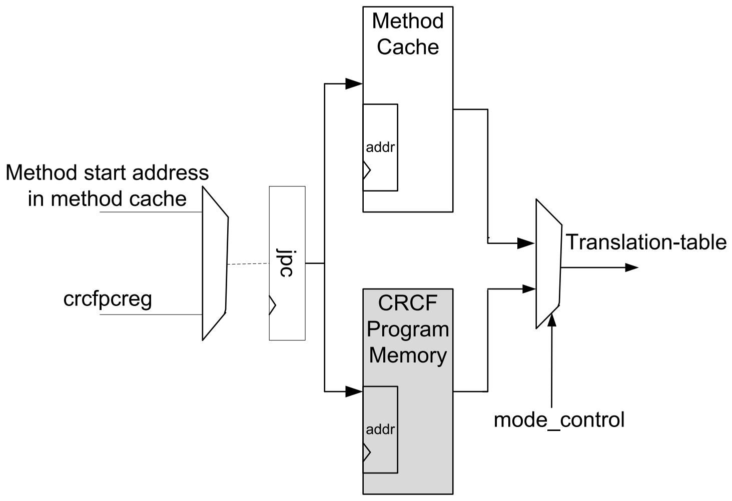


Figure 5.12: Switching between CRCF and JCF

5.8.3 Invoking JCF

The JOP-Plus has an efficient mechanism to invoke the JCF methods stored in the method area of the main memory in an organized fashion as shown in Figure 5.13. The JCF contains the Java action nodes (JAN) and each of these nodes has a unique identity code (JAN_ID) which is an integer ranging from *0* to *n-1* where *n* is the total number of Java action nodes in the program. The Java computations contained in each JAN are wrapped inside a method. In order to perform computations of a particular JAN, we need to invoke its corresponding method. The invoking of a method requires the start address of method structures stored in the main memory. The method structure is a table of method records each containing the encoded information about the method *code length*, the *start address* of method code, *variables*, *arguments* and *constant pool* address. Each method structure comprises of two 32-bit words i.e., 8 bytes. Therefore, the indices of table, the address of two consecutive methods structures, differ by two memory words. The method structures are arranged in ascending order with respect to JAN_ID and, if the address of the structure of method with JAN_ID=0 is known, then the address of the structure of any method can be calculated using its JAN_ID. If N is the address of a method structure having JAN_ID=0, then the address of any method structure with JAN_ID = n is given by $N + n * 2$. The address of the method structure with JAN_ID=0 is stored in a base register, called *jcfbasereg*. When invoking JCF, the JAN_ID information is provided in the CRCF instruction and the respective method is invoked.

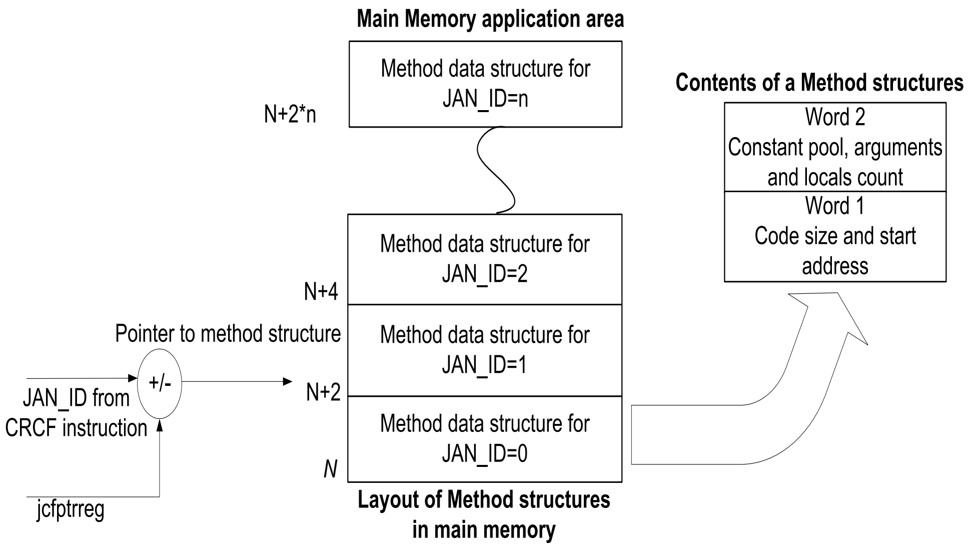

Main Memory application area
Method data structure for JAN_ID=n
N+2*n
Contents of a Method structures
Word 2
Constant pool, arguments and locals count
Word 1
Code size and start address
Method data structure for JAN_ID=2
N+4
Pointer to method structure
Method data structure for JAN_ID=1
+/-
N+2
JAN_ID from CRCF instruction
Method data structure for JAN_ID=0
N
jcfptrreg
Layout of Method structures in main memory

Figure 5.13: JAN organization in main memory

Invoking JCF is slightly different from conventional invoking of the methods in JOP [93]. The JOP-Plus does not need to store much of the information including the pointer to data structure of the calling method for loading it into *method* cache again upon return. The following are steps involved in invoking a JCF method:

- Store the address of next CRCF instruction to be executed in the *crcfpcreg*
- Load the address of method structure with JAN_ID=0 from base register *jcfbasereg* and push it onto stack
- Read the data call word comprising of the JAN_ID and data-lock pointer from Register-File into register T and pushes it onto stack
- Calculate the address of structure of the method to be invoked by loading the JAN_ID, add JAN_ID to itself and then add to base address
- Read the second word of the method structure containing constant pool address, argument count and variable count
- Extract these values and store in respective registers
- Read first word of method structure containing method start address and code length
- Extract the information and pass it to memory subsystem for loading the method code into cache
- Reset the *mode_control* flag making the *method* cache default memory for read/write operations

- Start loading the cache with method code from the main memory
- The start address of the code in the *method* cache is provided by memory subsystem which is loaded into Java program counter
- Wait for method code to be loaded into *method* cache completely
- Fetch next bytecode from *method* cache and start executing the JCF method

5.8.4 Return to CRCF

The return from JCF to CRCF program mode is performed through the *Native.rtc(result)* method which is replaced by a special bytecode during compilation. Each JCF method called by CRCF ends with this statement. The result is passed as argument and available on top of stack. The execution of special bytecode results in return to CRCF execution mode after performing the following operations:

- The result is available as the top element of stack which is written to the CRCF data memory at a location pointed to by data-lock pointer. The data-lock pointer is provided in the data call word is still available in temporary register T where it is stored prior to the switching of execution mode. The contents of CRCF program registers are never modified during the JCF program execution.
- The result is written to the CRCF data memory at a location pointed to by data-lock position.
- The address of the next CRCF instruction to be executed, which was stored in *crcfpcreg* before invoking JCF method, is loaded into *jpc* (Java program counter) register
- The *mode_control* flag is set
- The next instruction from the CRCF program memory is fetched

When returning to CRCF, reloading of *method* cache CRCF program code is not needed as it permanently resides in the CRCF program memory.

5.9 Translation-table Extension

The translation-table contains the start addresses for the bytecode implementation in microcode. As mentioned in Chapter 3, JOP has a single native instruction set in the form of microcodes. Every Java bytecode is translated either to a single microcode or a sequence of microcodes during the execution. The fetched bytecode acts as an index for the translation-table. This address is loaded into the JOP program counter for every bytecode executed. Mapping between bytecodes and, microcodes implementing the bytecodes, is provided in the JVM assembly code. The labels appearing in the assembly code represent the bytecode and mark the entry point for the bytecode implementation and are used to generate the translation-table. The translation-table is generated in the form of VHDL file during the assembly of JVM code. The JOP's translation-table contains 256 entries, for all possible bytecodes, in the order of their value. The mapping between the CRCF instruction opcodes and microcodes is also provided in the JVM assembly code.

The translation-table is extended with new logical area to accommodate the CRCF instruction set and has 512 entries now. The translation-table contains two logic areas now where low address space (0-255) still gives the mapping for bytecodes and microcodes and the higher address space (256-511) provides mapping for CRCF instruction opcodes and microcodes. Both JVM bytecode and CRCF instruction opcodes are 8-bit wide which can address only 256 entries. Therefore, fetched bytecode/opcode is concatenated with *mode_control* flag to form the 9-bit index of translation-table as shown in Figure 5.14. During CRCF execution, the *mode_control* flag is set and the address points to new logical area (high address space 256-511) provides the mapping for CRCF instruction opcode. The microcodes which implement JVM and CRCF instructions are located in the same microcode ROM.

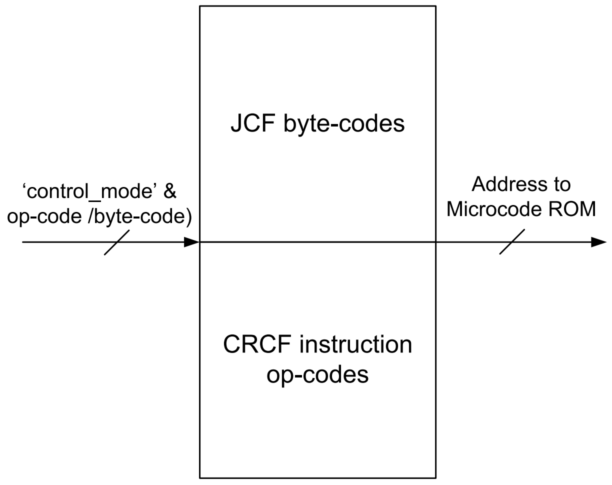

JCF byte-codes
'control_mode' & op-code /byte-code)
Address to Microcode ROM
CRCF instruction op-codes

Figure 5.14: Translation-table extension

5.10 JOP-Plus Architecture

This section describes the architecture of the JOP-Plus processor, including a discussion of all the functional units and the fundamentals of the processor hardware implementation.

5.10.1 CRCF Instruction Set Support

In addition to JVM bytecodes, the JOP-Plus architecture also provides support for the CRCF instruction set. The CRCF instructions shown in Table 5.5 are mapped to microcodes resulting in a number of new microcodes and extension in translation-table as illustrated in Section 5.9. It also results in addition of new functional units and registers discussed in the previous Sections. The precise description of CRCF instruction set, microcode mapping and register transfer operation is provided in Appendix F.

5.10.2 Bytecode Extension Summary

In addition to the functionalities performed by JOP, JOP-Plus needs to perform following specific functions when operating in the JCF program mode:

- Writing to CRCF program memory during initialization
- Switching from *main* to CRCF execution mode
- Writing result to CRCF data memory when performing data-computations in JCF mode
- Returning from JCF to CRCF execution mode
- Storing the address of method structure in a base register

The JOP-Plus extends the basic instruction set (bytecodes) of JOP [93] with the custom bytecodes to provide support for the above functions. The JOP-Plus requires 4 new bytecodes, in addition to bytecodes implemented in original JOP [93], to carry out these functionalities. The new bytecodes corresponding to JCF are added to JOP's unused bytecode space. The bytecodes are selected by keeping in view the micro-architectural constraints and with a goal to keep the clock frequency same as offered by the JOP [93]. The description of extended bytecodes is provided in Table 5.6.

Each bytecode and CRCF opcode is mapped to a single or a sequence of existing and new microcodes. The precise description of JCF bytecode for JOP-Plus is provided in Appendix F.

5.10.3 Extended Microcode Summary

JOP-Plus involves the extension of an existing instruction-set by means of a limited number of microcodes in function of the specific requirements of the application. All custom bytecodes and CRCF instructions are implemented in microcode, which is the native instruction set

Category	Instructions	Comments
Arithmetic/logic	AND_IMMEDIATE, OR_IMMEDIATE, ADD_IMMEDIATE, SUBV_IMMEDIATE, SUB_IMMEDIATE	The contents of Rx and Operand are ANDed and the result is stored in Rz except SUB_IMMEDIATE which affects zero flag only
	AND_INDIRECT, OR_INDIRECT, ADD_INDIRECT	The contents of Rx and Rz are ANDed/ORed/added and the result is stored in Rz
Data Movement	LDR_IMMEDIATE, LDR_INDIRECT, LDR_DIRECT	Load Rz with the content of immediate value / memory location pointed to by Rx or Operand
	STR_IMMEDIATE, STR_INDIRECT, STR_DIRECT	Store the content of Rx / immediate value, into memory location pointed to by Rz / direct address
Control flow	PRESENT, SZ,	Jump to address location if Rz (0)=0/Z=1
	JMP_IMMEDIATE, JMP_INDIRECT	Jump to address location unconditionally
	SWITCH	Switch execution to memory location pointed to by addition of content of Rx with memory location pointed to by Rx plus 1
Environment	CER,CEOT, SEOT	Clear/set environment control registers
	SSVOP, SSOP	Load SVOP/SOP with the content of Rx examples: SSVOP Rx, SSOP Rx
	LSIP, LER	Load Rz with the content of SIP/ER example: LSIP Rz, LER Rz
Special Instruction	CHKEND	Compare 4 bit memory blocks in Rx with Rz(3-0) and store the largest value back in Rz CHKEND Rz Rx
	SENDATA	Calls JCF example: SENDDATA Rx
	CLFZ	Clear Zero Flag
	NOOP, INIT, ESL	No operation

Table 5.5: CRCF instructions categorized based on their functionality

Bytecodes	Description
jopsys_rtc	*This bytecode causes JCF to return directly to CRCF without reloading method cache and also writes the result of computation back into CRCF data memory.*
jopsys_stcrcf	*This bytecode stores the address of structures of the method corresponding to JAN_ID =0 during the start-up. This information is used to invoke the desired JCF method by CRCF.*
jopsys_wrcrcf	*This byte code is used to initialize the CRCF memory during the execution of the main method. The instruction and address are pushed on TOS and TOS-1.*
jopsys_switchmode	*This bytecode shifts the execution control to CRCF. It stores the execution address of main for a future use. The CRCF address is loaded into program counter and control flag is toggled.*

Table 5.6: Extended bytecodes

of JOP [93]. Basically, we identify all those operations that are implemented by existing microcodes and the ones that are left, need new microcodes. The register transfer between existing hardware components is performed by using existing microcodes. However, new microcodes are introduced for the register transfer between new hardware components and, between new and existing components. In all, 16 new microcodes are introduced in the micro-architecture for both CRCF programming model and new JCF program bytecode. A summary of extended microcodes is provided in the Table 5.7.

5.10.4 JOP-Plus Data-path

The JOP-Plus data-path is organized as a 4-stage single-issue pipeline including the bytecode fetch stage. Figure 5.15 shows detailed data-path of the JOP-Plus processor with all the components. It implements the bytecode, microcode fetch, decode and execute stage. The flexible data-path of JOP has been extended to execute CRCF instruction set. The composite data-path yielded, enables all transfers to execute both Java bytecodes and CRCF instructions. The components added or extended to support CRCF execution are shown as shaded. It includes, Register-File, function specific and temporary register, MAX unit, memories, translation-table etc. By taking advantage of efficient CRCF processing, the JOP-Plus architecture shows a significant improvement in execution times.

The execution of microcodes is pipe-lined and takes up-to three cycles to execute a microcode. Both programming modes share the same ALU therefore the operands from CRCF programming mode are pushed onto stack prior to their execution. The result of ALU is popped from the stack and stored in the desired location. The operands for CRCF mode are 16-bit wide; therefore a string of zeros is appended at higher order word before feeding it to ALU which

Microcodes	**Description**	Register Transfer
stcrcfpc	*Stores the address of next crcf instruction in crcfpcreg*	$crcfpcreg \leftarrow jpc$
stjcfbasereg	*Loads the jcfbasereg with the pointer to method data structure with JAN_ID=0 available on stack*	$jcfbasereg \leftarrow A$, $B \leftarrow A$, $stack[sp] \leftarrow B$, $sp \leftarrow sp - 1$
ldt	*Pushes the value of register T with the value of Top of stack*	$A \leftarrow T$, $stack[sp + 1] \leftarrow B$, $sp \leftarrow sp + 1$
ldrxt	*Loads the contents of register Rx into register T*	$T \leftarrow RF[Rx]$
switchmode	*Toggles the mode_control flag*	$ctrl\_mode\ flag \leftarrow not\ ctrl\_mode\ flag$
ldcrcfopdt	*Loads immediate operand of CRCF instruction stored in ctrlopdreg into T.*	$T \leftarrow crcfopdreg$
ldjcfbaset	*Pushes the value of jcfbasereg into register T*	$T \leftarrow jcfbasereg$
ldcrcfpct	*Loads address of next crcf instruction given in crcfpcreg into T.*	$T \leftarrow crcfpcreg$
ldrzt	*Loads the contents of register Rz into register T*	$T \leftarrow RF[Rz]$
wrcrcfdm	*Writes the contents of CRCF data memory.*	$CRCFDM[B] \leftarrow A$, $B \leftarrow A$, $stack[sp] \leftarrow B$, $sp \leftarrow sp - 1$
wrimmcrcfdm	*Writes immediate value into CRCF data memory.*	$CRCFDM[ctrlopdreg] \leftarrow A$
ldcrcfdmt	*Reads value from CRCF DM and stores it into T.*	$T \leftarrow CRCFDM[A]$
wrrf	*Writes the Register-File.*	$RF[Rz] \leftarrow T$
stzf	*Sets the 1-bit zero flag register if zfreg if Z=1*	$ZFREG \leftarrow zF$
ldmaxt	*Loads the result of MAX hardware unit into T.*	$T \leftarrow MAX$
clzf	*Clears the 1-bit zero flag register.*	$ZF \leftarrow 0$

Table 5.7: Extended microcodes summary

is 32-bit. The CRCF programming mode involves only the most basic computations like addition, subtraction, logic AND and logic OR and utilizes the existing ALU. The Z flag register indicates if the result of the operation executed by the ALU is zero or not. It is updated by each arithmetic operation and is used for conditional change of program flow. The *zfreg* register holds this value for future use. The MAX unit is a dedicated hardware comparator block which compares four-bit memory blocks from RF outputs and outputs the maximum nibble which is stored back into RF.

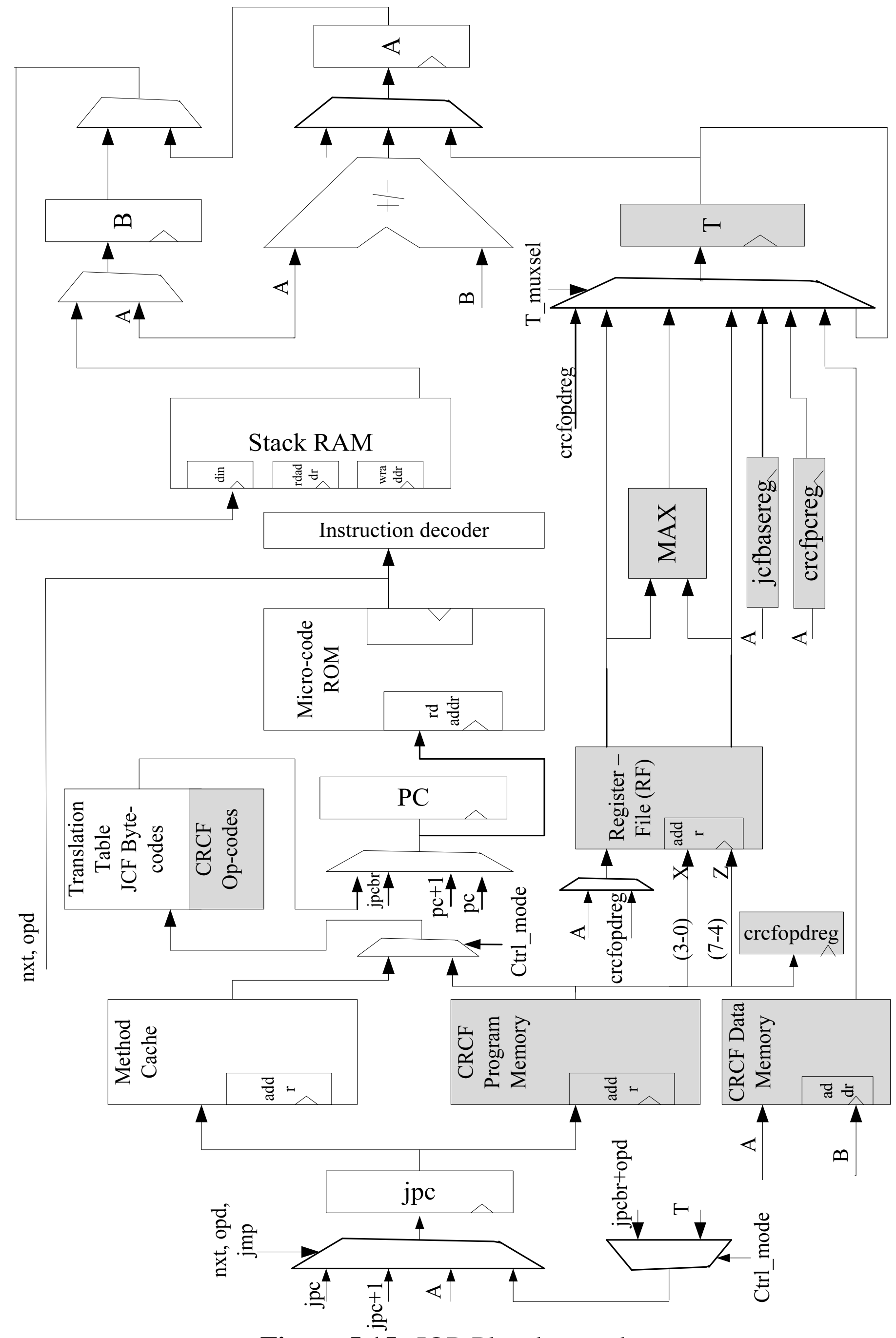


Figure 5.15: JOP-Plus data-path

5.10.5 Instruction Execution Overview

The execution of a bytecode/instruction involves calculating the address of the current instruction, bytecode/instruction fetching, microcode fetching, decoding, and executing that instruction. Depending on whether the processor is working in CRCF or JCF mode, the CRCF instruction or bytecode is fetched from the CRCF program memory or the *method* cache in the *bytecode fetch* stage, respectively. The value of the fetched bytecode/instruction is used as the address in the translation-table and reads out the start address of the microcode sequence of that particular bytecode. This address is used to fetch the microcode instruction from the microcode ROM. If the processor is in the CRCF program execution mode, then the address read from the translation is for the CRCF instruction. The microcode fetched during this (*Fetch*) stage is fed to the *Decode* stage for generating the control signals. The stack addresses are calculated during the same stage. The operations are performed on top two elements of the stack stored in two discrete registers: *TOS* and *TOS-1*, labeled *A* and *B*. Each arithmetic/logical operation is performed with registers A and B as the source, and register A as the destination. This holds both for JCF and CRCF. In case of CRCF instruction execution, the Register-File is read and written in the same stage using microcode instructions.

When in CRCF mode, PRESENT, SZ, JUMP_IMMEDIATE, JUMP_INDIRECT and SWITCH instructions change the flow of control and modify the contents of *jpc* register. All the branch and jump instruction always provide the absolute target address and do not use relative offset addressing.

5.10.6 CRCF Instruction Set Execution

All the JCF bytecodes are executed in the conventional way as in JOP and details of execution of Java bytecodes are available in [93]. In this section, we will discuss the execution instructions in CRCF programming mode. During the CRCF instruction execution, the operands are pushed from Register-File onto the stack and the results are written back into the Register-File with data available in top of the stack, and register address is available as part of the CRCF instruction. In Java mode, ALU operations take one or two inputs from registers *A* and *B*, and store a result back in register *A*. In CRCF mode, the operands from the RF can be directly fed to ALU by-passing registers A and B, it might increase the propagation delay since the ALU falls in the critical path, therefore, the operands are loaded into a temporary register first and then pushed onto the top of the stack. All the *LDR* (load register) and *STR* (store register) instructions handle all data movement between registers and CRCF data memory. During load/store of the data from the CRCF data memory (*LDR_IMMEDIATE*, *LDR_DIRECT*, *LDR_INDIRECT*, *STR_IMMEDIATE*, *STR_DIRECT* and *STR_INDIRECT*), the address is always provided in A, and data in B during a store operation. If the address/data is an immediate value, it is fetched from the CRCF program memory and pushed on the stack. All the immediate arithmetic/logical (*AND_IMMEDIATE*, *OR_IMMEDIATE*, *ADD_IMMEDIATE*, *SUBV_IMMEDIATE* and *SUB_IMMEDIATE*) instructions perform operations between the immediate value and content of the register Rx and store the result into the register Rz except *SUB_IMMEDIATE* which does not store the result back into RF. The content of Rx register and operand values are pushed onto the stack and the operation is performed. The result available on TOS is written back into RF to a location pointed to by the instruction Rz field. In case of instructions with indirect mode (*AND_INDIRECT*, *OR_INDIRECT* and *ADD_INDIRECT*), operation is performed on Rx and Rz. The contents of these registers are pushed into T and then into *A*. The result available in TOS is stored in *A*.

The *LSIP*, *SSOP*, *SSVOP*, *SEOT*, *CEOT* and *LER* are environment instructions responsible for data movement to/from environment registers connected to memory-mapped IOs. The address of the environment register is pushed on the stack and the contents of the registers are read and stored in Register-File. When writing to environment register, data to be written is also provided.

The *CHKEND* instruction finds out the maximum of Rx and Rz{3..0} nibbles and results is stored in Rz. The contents of Rz and Rx are fed to special hardware unit which finds the maximum nibble which is written to Register-File.

The *JUMP* instructions result in an unconditional jump to the target address provided as immediate (*JUMP-IMMEDIATE)* value or as the contents of a register (*JUMP-INDIRECT*). The operand/register contents are read into A via T and stored into *jpc* and the next instruction is fetched from the address pointed to by the immediate value. The *PRESENT* (jump if value is not present) and *SZ* (jump if zero flag is set) CRCF instructions perform conditional jumps. The *SWITCH* instruction is a complex instruction and results in an unconditional jump together with couple of memory read operations. It is used to decode execution path in switch nodes in AGRC. The contents of register Rx are pushed on the stack and the memory location pointed to by its contents (parent node) is read into register T. It contains the number of particular child node we want to execute. This is added to parent node incremented by one (child nodes are stored next to parent nodes) to get the pointer to the selected child node in CRCF [8] and unconditional jump is performed.

The *SENDATA* instruction is used to directly invoke the desired data-computations (JAN) presented as a method in JCF from within the CRCF. This implementation is different from GALS-JOP where invoking the desired Java method and return from Java method is carried out in conventional way. The JAN_ID, together with data-lock position pointing the memory location for result, is provided as immediate operand and pushed on the stack. The address of method structure with JAN_ID zero in *jcfbasereg* register is used to calculate the address of the structure of the desired method to be invoked. All the methods in the JCF are ordered by their IDs which makes it possible to calculate their structure address by knowing JAN_ID. The CRCF PC is saved in *crcfpcreg* register. The structure of the method, which is two words long, is read. The first word of method structure contains the information about the method code such as the start address and code length. This information is passed on to the memory subsystem which loads the desired method into the *method* cache and the execution starts. The second word contains the number of arguments, local variable count and constant pool address which are extracted and stored in appropriate registers. The *jopsys_rtc* (return to CRCF) is a new custom bytecode added to support direct return to CRCF from JCF, without reloading *method* cache and also writes the result of computation back into memory.

While returning from JCF to CRCF after performing the data-computations, the result is returned on top of stack (A). The JAN_ID and data-lock position are provided in the Register-File. The register contents, still available from Register-File, are pushed on the stack. The data-lock position is extracted and result is written to memory location pointed to by data-lock during return to CRCF. The description of the complete CRCF instruction set can be found in [39].

5.10.7 Communication with the Environment

In SystemJ each asynchronous clock-domain communicates with the environment only once the end of tick (EOT) is set high and all the environment input signals are read and the required signals including their values are emitted to the environment. The JOP-Plus communicates with the environment through a set of registers and instructions mentioned above. The ER register is set high by the environment indicating that it has finished reading signal values/statuses from the previous instant of time. Once the ER signal is received, this register value is set to low and the EOT register is set indicating to the environment to stop reading signals from SOP and SVOP and stop writing into SIP. The environment is expected to instantaneously respond to this request and maintain the register integrity (no read/write) while the EOT is high. Then input signal statuses available in SIP are stored into the data memory and output signal statuses from the previous instants processing are loaded into SOP register from CRCF data memory. The SVOP register is updated to indicate the type of the emitted signals (either pure or valued signals). A high bit represents a valued signal and a low bit a pure signal. The valued input/output signals are read/written in Java programming mode. Lastly, the EOT register is set low to indicate that environment can start reading and writing again. These steps are carried out at start of every clock-domain (EOT).

5.11 Experimental Results

In this section, we present results of experiments conducted to evaluate and compare our proposed architecture to execute the SystemJ with a number of other approaches.

5.11.1 Benchmarks

The benchmarks include both synchronous and asynchronous examples. The synchronous benchmarks include a runner's behavioral description (runner), a simple combination lock (*cl*) and demoloop (*dl*). The asynchronous case is represented by an asynchronous protocol stack (aps) example [5]. Each of *dl*, *runner* and *cl* described in SystemJ consists of a single clock domain comprising of four reactions whereas *aps* has two clock domains and six reactions in total. The SystemJ code for benchmark examples is provided in Appendix D.

5.11.2 Experimental Set up

All presented data have been collected from the experiments carried out by using the cycle-accurate *ModelSim* simulator targeting *Altera* Cyclone II FPGA with 70k logic elements, 2MB of RAM and running at 50 MHz clock. The system is capable of running at 100 MHz but the results presented are for 50 MHz clock for fair comparison with earlier published results.

5.11.3 Performance Comparison

The execution speed comparisons are given in terms of the average execution time between two consecutive ticks for each clock-domain, which are indications of the throughput of the platforms. The average tick execution time is obtained by averaging one million ticks. Time between two logical ticks, unlike time between two real clock ticks, has a variable duration. Within a system, each clock-domain runs at its own tick, and any two clock-domain ticks are unrelated. We compare the average tick times of various benchmarks on all the platforms developed during this research.

The results in Figure 5.16 show that JOP-Plus outperforms all the existing platforms. The JOP-Plus platform wins the race against the fastest platform TP-JOP and is up-to 55% faster while being far more economical. The TP-JOP executes each control (CRCF) instruction in a constant time (3 to 4 cycles), while JOP-Plus has variable execution time for CRCF instructions. In case of CRCF execution only, TP-JOP will outperform JOP-Plus, but it loses the advantage in JCF execution and transfer of control between JCF and CRCF. It is only 32% faster than the TP-JOP when executing the benchmark application *cl* which is more control oriented. The

JOP-Plus is 44% and 55% faster while executing the application with data-computation such as *runner* and *dl*, respectively. JOP-Plus improves JCF execution by decomposing it into smaller methods for more efficient execution. The JOP-Plus is up-to 125% faster than GALS-JOP while being at par in resource usage. GALS-JOP suffers from poor CRCF execution due to adopted mechanism where the CRCF instructions are translated to Java and all the operands and Register-File addresses are passed as arguments. It performs even better when executing the benchmarks with data-computation due to efficient mechanism adopted to invoke the JCF.

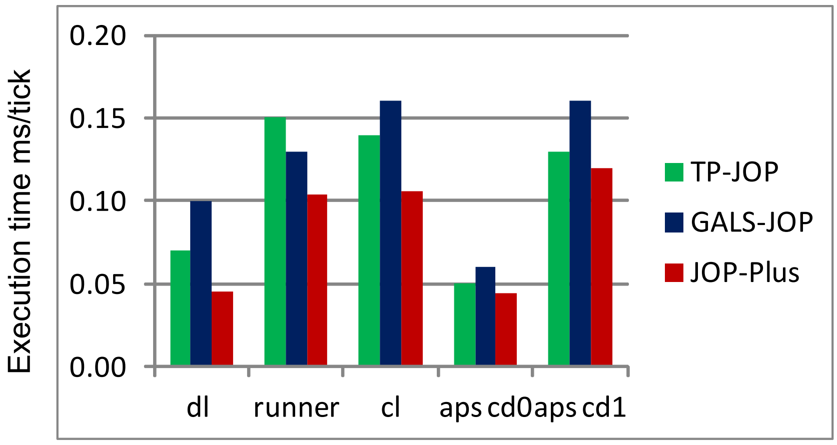


Figure 5.16: JOP-Plus comparison with other implementations

The TP-JOP uses the same CRCF assembly code and uses a custom processor for its execution. It executes the CRCF code and JCF code concurrently on two separate processors, ReCOP and JOP, respectively. The JOP-Plus is expected to perform closely to or even better than the TP-JOP due to following reasons:

- TP-JOP executes each CRCF instruction in 3 to 4 cycles, whereas JOP-Plus has variable execution time for CRCF instructions. In case of CRCF execution only, TP-JOP will outperform JOP-Plus, but it loses the advantage in JCF. When compared with GALS-JOP, the JOP-Plus has efficient CRCF execution. GALS-JOP [95] suffers from poor CRCF execution due to adopted mechanism where the CRCF instructions are translated to Java and all the operands and RF addresses are passed as arguments. As a result, each CRCF instruction is mapped to a custom bytecode to execute the specific functionality which requires executing few more bytecodes to push the arguments on the stack. In JOP-Plus, CRCF instruction set and format is preserved. The immediate operand values, target addresses and operand register addresses are available in the instruction.

- The JOP-Plus organizes the data-computations in a better way which helps in executing them efficiently in JCF mode. The TP-JOP requires invoking a method to select the clock-domain and another method to select and execute the desired data-computations. The control processor makes a data call to JOP (which executes the main and polls the control processor). The JOP extracts the case number and clock-domain fields from the data call word. First, it calls a method consisting of clock-domain code wrapped in a *switch-case* statement to select the required clock-domain. The selected clock-domain invokes another method containing the data-computations also wrapped in *switch-case* statements. The JOP [93] has a special *method* cache which holds a complete method at a time. Invoking of a method and returning to a method results in cache miss and requires the cache to be loaded again, which incur execution time cost. The higher call depth results in multiple loading of the cache on both invocation of methods and return from the method.

- Furthermore, the data-computations in JOP-Plus are decomposed into small methods, each corresponding to a JAN. It requires fewer cycles to fetch the method code from the main memory into *method* cache prior to execution. In TP-JOP, the data-computation are presented in the form of *switch-case* statements enclosed in a single large method thus requiring more time to load it into the cache. The GALS-JOP also benefits from the same approach of decomposing data-computations into small methods.
- Another feature where JOP-Plus can gain in performance is the implementation of an efficient mechanism to invoke the data-computation. It invokes the JCF methods directly from the CRCF without invoking *main* method as is the case with TP-JOP. The return to CRCF is direct without mediation of any other Java method, so no need to load caller method on return to the CRCF. The GALS-JOP [95] also has the similar characteristics but has more expensive invoke and return compared to JOP-Plus in terms of resources used and clock cycles required.
- Another improvement in the performance of JOP-Plus comes from the local storage of frequently accessed CRCF data structures requiring only single cycle to load and store data.

The JOP-Plus approach is amenable to worst case execution time (WCET) analysis as the CRCF instruction are implemented in a fashion similar to JOP with no time dependencies between instructions result in a simple processor model for the low-level WCET analysis.

5.11.4 Effectiveness

An ideal execution platform would have both very small execution times and very small resource usage at the same time. But, performance gain is not for free and usually comes at the price of extra resource usage which adds to the cost. Performance and silicon property form a design trade off - improve one and you degrade the other! Hence, performance only is not the true figure of merit for the system. To quantify how effective, or efficient an execution platform is in terms of execution time and resource usage, we should take into account both execution times and logic elements used.

We compared the effectiveness of different approaches by defining a performance measure that correlates average tick times (*ATT*) with resource usage in terms of the number of logic elements (*NLE*) used for implementation. The performance indicator is specified as $K/(ATT * NLE)$, where K is a scaling constant. The platform effectiveness is inversely proportional to the tick time and the resources used. The results in Figure 5.17 indicate that the JOP-Plus is the most effective execution platform for all benchmark examples.

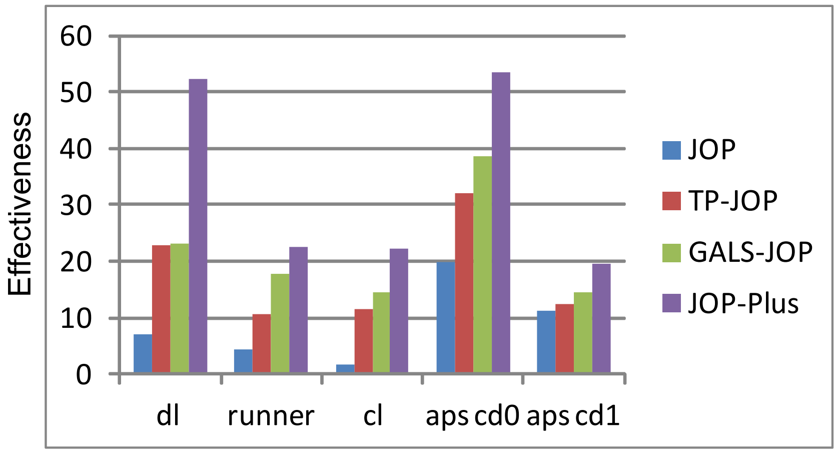


Figure 5.17: Comparison of effectiveness

In summary, the proposed architectures shows better execution times with significantly fewer hardware resources compared to the latest platform, without any addition to the memory footprint.

5.12 Summary

In this Chapter, we have proposed a new JOP-Plus processor which is capable of executing the SystemJ programs compiled in a way separating the concurrency and reactivity control flow from Java control flow. This is achieved by extending the original JOP processor, to support CRCF instructions. The GALS-JOP approach being single processor is an economical and gave reasonable performance boost. But at the same time, it has shortcomings of extra resource usage in the form of jump-table and inefficient CRCF to JCF translation mechanism. The JOP-Plus overcome these problems and seamlessly integrates the CRCF and JCF by providing two distinct modes of operation. The benchmark results show that JOP-Plus not only reduces the execution time but also avoids few of execution resources deployed by the GALS-JOP by making the efficient use of the base processor. Another performance criterion introduced correlates speed and resource usage giving a single measure of effectiveness of the platform. All the claims are validated through results obtained by running benchmarks on experimental set up.

Chapter 6

GALS-CMP

Embedded systems are typically heterogeneous requiring interacting hardware and software components, which can be locally synchronous and globally asynchronous and combine both control and data dominated blocks. The applications need to make good use of minimal hardware resources within embedded real-time systems. The GALS-JOP and JOP-Plus approaches presented in Chapter 4 and Chapter5, respectively, achieve this goal. But, today's modern applications demand ever-increasing processing power and have shifted from conventional low performance products to high throughput and computation intensive products. Limits on power consumption and temperature make it impractical to rely on increasing clock frequencies to boost processor performance. Compounding these problems is the simple fact that with the immense numbers of transistors available on today's microprocessor chips, it is too costly to design and debug ever-larger processors every year or two. Chip Multiprocessors (CMPs) avoid these problems by filling up a processor die with multiple, relatively simpler processor cores instead of just one huge core [98].

This chapter presents a novel multi-processor architecture for concurrent execution of programs that follow the Globally Asynchronous Locally Synchronous (GALS) formal model of computation. Programs are specified using the SystemJ concurrent programming language, suitable for modeling heterogeneous embedded applications that contain reactive and control driven parts and interact with the external environment. The proposed architecture is based on the compilation approach where control-driven and data-driven operations are separated. They are then executed on distinct JOP-Plus cores which are capable of supporting both types of operations, implemented as two modes within the single core. The GALS programs are partitioned at clock-domain boundaries and they are allocated to different cores. Each core can switch between two modes without any overhead. The JOP-Plus core as the basic building block of the multiprocessor extends Java Optimized Processor (JOP), suitable for data-driven transformational operations, with control-oriented constructs that implement concurrency, reactivity, and control flow in SystemJ. The resulting multiprocessor system, called GALS-CMP, is suitable for the execution of comlex heterogeneous embedded applications. Experimental evaluation over a range of benchmarks shows significant performance improvements over the existing platforms developed for the execution of the SystemJ program.

This chapter starts with the discussion on performance limits of uni-processor approach and explore the ways to increase the performance or processing power of the system. We then give a brief overview of some popular multiprocessor systems developed in industry or academia. Then we outline GALS-CMP system features alongwith the benefits which it brings into when compared with existing execution platform discussed in the previous chapters of the thesis. We also present the architectural details and the memory organization. The compilation and execution flow of the SystemJ program on the proposed multiprocessor architecture is explained with the help of an example. Finally, we validate the effectiveness of proposed system through experimental results.

6.1 Uniprocessor system shortcomings

A SystemJ program consists of multiple clock domains which run at unrelated ticks. The JOP-Plus approach to execute GALS programs is the most efficient and economical for a single processor system. This approach has some downsides as discussed below:

- *Response time:* JOP-Plus approach shows poor response time due to cyclic scheduling of the clock domains one after the other. They respond to the environment only once at the start of clock-domain execution. Hence, for each clock-domain, system will be able to interact with the environment only when it resumes the execution of same clock-domain after executing all other clock-domains. The inability of the clock-domain to respond to the environment at the end of its execution significantly elevates the response time.
- *Exploitation of application parallelism:* The SystemJ application programs offer high degree of parallelism which could be exploited to boost the performance of system. Unfortunately, uni-processor approach is unable to exploit to this parallelism. They can only extract a limited amount of parallelism from a typical instruction stream using conventional techniques [99]. Furthermore, clock-domains are scheduled in a cyclic executive manner and the clock-domains are made to run one after the other.

The response time can be reduced by making the processor run faster. To obtain this, designers have relied on better circuits (more integration, faster logic) and parallel execution (e.g. superscalar processing). Unfortunately, current architectures are approaching the limits of known technology in both respects [100].

6.2 Harnessing the performance

There are many ways to improve the performance of microprocessors. In previous Chapters, we demonstrated the performance improvement of the SystemJ programs execution in terms of avarage tick time of clock-domain through architectural modifications. Another way to achieve more performance out of a piece of silicon is by ram-ping up the clock but this usually results in more power consumption. Advances in manufacturing processes allowed decrease in feature size, thus allowing the packaging of more transistors in the same die area. We can't build microprocessors with ever increasing die sizes due to the power constraints. Although extensive research has been carried out on the optimization of power in individual components of a system, yet a considerable amount of power is consumed by the system thus causing reliability issues and increasing the cooling cost. Alternatives to traditional way of increasing throughput for microprocessors are being sought. Low-power circuit and micro-architecture techniques, on-die L2 Caches, Single Instruction Multiple Data Instruction Set Architecture extensions, multithreading, and multiprocessing are among other ways to increase throughput of microprocessors [101]. Most of these methods have already been applied in the current microprocessor architectures. According to Hennessy and Patterson [102], we are now reaching the limits of exploiting ILP efficiently. Olukotun et al [103] present an interesting study that argues that multiprocessor solution is a better path to high performance than going to higher level of instruction level parallelism.

The recent trend in computer design is chip-multiprocessors (CMPs) with increasing number of CPU cores per chip. The parallel processors allow to continue to scale chip level performance, but give rise to another difficult issue: generating parallel code to run on these machines. Multiprocessors in general are notoriously difficult to program. The complicated problem of partitioning a program becomes easier in application-specific domains as parallelism is easier to indentify and exploit. This is because much more is known about the computational structure of the functionality. The GALS applications employ a specialized computation model where application parallelism is explicitly specified in the form of clock-domains. The GALS applications described in SystemJ can be partitioned at clock-domain level and run in parallel that can take the advantages of CMPs. The Amdahl's law [104] states that the speedup of such parallel program is limited by the portion of sequential code in the program. The nature of SystemJ program suggests that the clock-domains run concurrently, hence, they are ideally suited for multiprocessor execution environment. Besides parallelism, memory bandwidth and memory management schemes are reported to be limiting factors in performance that can be obtained from these multiprocessors [104]. The increasing level of on-chip integration, together with a slowing rate of voltage supply reduction, exacerbates the power constraints in microprocessor design. According to Wolf [105], CMPs combine the significant advantages of embedded systems: increased performance, lower power consumption, and cost efficiency. The cost of designing a multiprocessor system-on-chip, where the processors work at moderate speeds and system throughput is multiplied by multiplicity of the processors, is smaller than designing a single processor which operates at much higher clock speed.

The JOP-Plus core proposed in the Chapter 5 only executes a single clock-domain at any given time. Although, uniprocessor execution of SystemJ programs is more efficient and economical in terms of resources (as is the case with GALS-JOP and JOP-Plus), but it can results in poor response time due to cyclic scheduling of the asynchronous behaviors called clock domains [5]. These programs respond to the environment events only once at every logical clock cycle of the clock-domain execution. Hence, for each clock-domain, system will be able to interact with the environment only when it resumes the execution of the same clock-domain after executing all other clock-domains. The SystemJ programs offer high degree of concurrency at clock-domain level which could be exploited to boost the performance of the system if clock domains are executed in parallel. The single processor approach is unable to exploit this parallelism and fails to make efficient use of it. Furthermore, achieving higher performance out of a piece of silicon by ramping up the system clock usually results in higher power consumption

6.3 Embedded Multiprocessors

In the embedded system domain, there are two different types of CMP architectures:

(1) heterogeneous multiprocessors and (2) homogeneous multiprocessors.

6.3.1 Heterogeneous Multiprocessors

Multiprocessors with a heterogeneous architecture combine a core CPU for controlling and communication tasks, and additional special processing elements, which are often tailored to specific applications. Some examples of heterogeneous multiprocessors include the ST Nomadik [106], designed for mobile multimedia applications; the Philips Nexperia PNX-8500 [107], aimed at digital video entertainment systems; or the TI OMAP family [108], designed to support 2.5G and 3G wireless applications; HiBRID [109], designed for stationary as well as multimedia applications. The TP and TP-JOP are two heterogeneous multiprocessor architecture developed specifically for the execution of SystemJ based applications. The TP architecture makes use of a custom and a general purpose processor to execute the control operations and data operation respectively. TP-JOP is an improvement over TP where heterogeneous processors execute both control operations and data operations natively.

6.3.2 Homogeneous Multiprocessors

The homogeneous multiprocessors consist of two or more similar CPUs sharing a main memory. The ARM11 MPCore [110] is a homogeneous multiprocessor and introduces a preintegrated symmetric multiprocessor consisting of up to four ARM11 micro-architecture processors [111]. It has 8-stage pipeline architecture, independent data and instruction caches, and a memory management unit for the shared memory. Gaisler Research AB designed and implemented a homogeneous multiprocessor system called LEON3-FT-MP [112]. It consists of one centralized shared memory and four LEON3-FT processor cores that are based on the SPARC V8 instruction set architecture [113]. All the CPUs, additional IO controllers and memory controllers are connected using two AMBA-specified advanced high-performance buses (AHB) [114]. One AHB runs at the CPUs' frequency and connects the processors to the shared memory controller. The low-speed bus connects all other peripheral devices. MicroBlaze-based CMPs can be designed with the Xilinx Embedded Development Kit (EDK). MicroBlaze is a 32-bit reduced instruction set computer (RISC) optimized for FPGA implementation [115]. The pipeline length of the CPU can be configured to either three or five stages. It implements the Harvard architecture with separate instruction and data buses. The CPU can be tailored to the individual application needs (i.e., peripheral controllers or cache sizes). Memory and peripheral devices are connected via the on-chip peripheral bus (OPB) [116]. Xilinx provides an OPB bus arbiter [117] that can integrate up to 16 masters into the system. A newer version of the MicroBlaze, supported in both Spartan-6 and Virtex-6 implementations, as well as the 7-Series, supports the AXI specification.

Altera's NIOS II [47] and the System-on-a-Programmable-Chip (SOPC) Builder [118] support the design and implementation of CMPs in Altera's FPGA technology. The NIOS RISC architecture implements a 32-bit instruction set similar to the MIPS instruction set architecture. NIOS II can be customized to meet the application requirements: three different models, from nonpipelined up to a 6-stage pipeline. Avalon [48] is the SoC bus used by the SOPC Builder. It connects the master and slave components to the System Interconnect Fabric. For multiprocessor systems, the System Interconnect Fabric integrates an arbitration module [48]. The arbitration logic can be configured in the SOPC Builder. The arbitration schemes include fairness-based, round-robin scheduling, burst transfers, and minimum share value.

MPOC [119], Daytona [120], Texas Instrument TMPS320C6474 [121] are few other examples of homogeneous multiprocessor developed for embedded DSP applications.

JopCMP [122] is a symmetric shared-memory multiprocessor, and consists of up to eight Java Optimized Processor (JOP) cores, an arbitration control device, and a shared memory. All components are interconnected via a system on chip bus. The arbiter synchronizes the access of multiple CPUs to the shared main memory. The JopCMP is designed for maximum time predictability, where simple and accurate WCET analysis is more important than good average-case performance. Furthermore, it is open-source, customizable, configurable, technology-independent (like LEON) and has been ported to FPGAs from Altera, Xilinx, and Actel thus, avoids a lock-in to a single FPGA vendor, as is the case for MicroBlaze and NIOS.

GALS-CMP, which is proposed and described in this chapter, is a homogeneous chip-multiprocessor system which uses the JOP-Plus as the base node, and each node is capable of executing any of two programing models, CRCF or Java, at any given point of time. The GALS-CMP follows the methodology adopted by the time predictable multiprocessor system JopCMP [122]. It executes GALS application described in SystemJ having multiple clock-domains. All the clock-domains can be run in a truly concurrent fashion on this parallel architecture.

6.4 GALS-CMP

A SystemJ program exhibits local synchronous concurrency and global asynchronous concurrency. This can be exploited by introducing hardware platforms consisting of multiple cores to provide parallel and faster execution. To achieve efficient execution, we divide programs into parallel operations at the clock-domain boundaries. The programmmer or system designer is offered the choice to allocate clock-domains during compilation and these parallel operations are then distributed onto the cores. In other words, each part of the compiled program to be executed on a core comprises of the CRCF and JCF code for a complete clock-domain.

We propose new, scalable homogeneous chip-multiprocessor architecture for executing programs described in SystemJ. The multiprocessor is based on the use of multiple cores, where each core supports both types of operations, implemented with two execution modes within the single core. The GALS-CMP follows the approach adopted by the time predictable multiprocessor system JopCMP [100] and instead of JOPs, uses the JOP-Plus core. It allows all clock-domains to run in a truly parallel fashion.

The architecture is suitable for FPGA prototyping as it uses some of the features of current FPGA devices like distributed SRAM memory blocks, but easily fits to the System-on-Chip approach and ASICs.

The major benefits of GALS-CMP approach are:

- A new, scalable, multiple processor architecture for supporting GALS model of computation. The proposed architecture is suitable for execution of SystemJ based heterogeneous application that contain data and control dominated parts that interact with external environment.

- This GALS-CMP is based on JOP-Plus core, therefore, it is capable of handling both concurrency and reactive control flow (CRCF) and Java control flow (JCF). It is capable of working in either of the CRCF mode or Java mode only.

- It allows parallel execution of clock-domains that can be run on a number of physical JOP-Plus processor cores. The exploitation of parallelism reduces the response time of the clock-domains as well as application execution time.

- The GALS-CMP is based on JOP-Plus core which is very compact resulting in economical multiprocessor architecture having smaller die size and cost. It is expected to be energy efficient as JOP-Plus core consumes less energy when compared to TP-JOP or TP which use the same compilation strategy. Since JOP-Plus core contains fewer resources than the its counter parts and takes fewer cycles to execute the application, therefore, it is likely to consumes less energy.

- GALS-CMP gives improved hardware performance because the base processor's customized hardware fits the GALS MoC better. It stores the CRCF program and data in a local memory resulting in no load on shared main memory bandwidth when executing the CRCF program. It also gives faster and economical access to program and data memory of CRCF.

- When compared with the scaled version of TP-JOP to handle multiple clock-domains concurrently, the CMP based on JOP-Plus is homogeneous in nature and does not required any communication between control and data execution processors as both CRCF and JCF are executed on the same processor. Also, it uses very simple and efficient mechanism to calls data-computations encapsulated in JCF methods when executing CRCF without the need of any special communication infrastructure. This keeps communication infrastructure very simple and does not explode when scaling it to multiprocessor like the TP [39] and TP-JOP approaches. It does have perofrmance issues related to the access to shared memory.

- The clock-domain allocation to different JOP-Plus cores does not need to take into account the amount of communication between clock-domains communicating using channels in SystemJ making the clock-domain allocation simple. All the clock-domains communicate using shared memory, therefore, locating two clock-domains communicating with each other on a same core or different cores will not have much effect on the program execution except when multiple cores do it at the same time.

- The JOP-Plus core is amenable to the analysis of worst case execution time.

6.5 GALS-CMP System

The GALS-CMP follows the methodology adopted by the time predictable multiprocessor system JopCMP [100]. It consists of multiple cores, connected to the shared memory through an arbiter using a SoC bus as shown in Figure 6.1. Each core uses a local *method* cache, stack cache and the CRCF memory, which contains the control code of a SystemJ clock-domain, to reduce the shared memory accesses. Furthermore, the depicted GALS-CMP architecture shows a synchronization unit which has the responsibility to coordinate access to the shared objects by a mutual exclusion mechanism. It makes other cores to wait until the core accessing the shared object finshes its job. On-chip IO devices, such as a controller for real-time Ethernet or a real-time field bus, may be mapped to shared memory addresses and are connected via the memory arbiter.

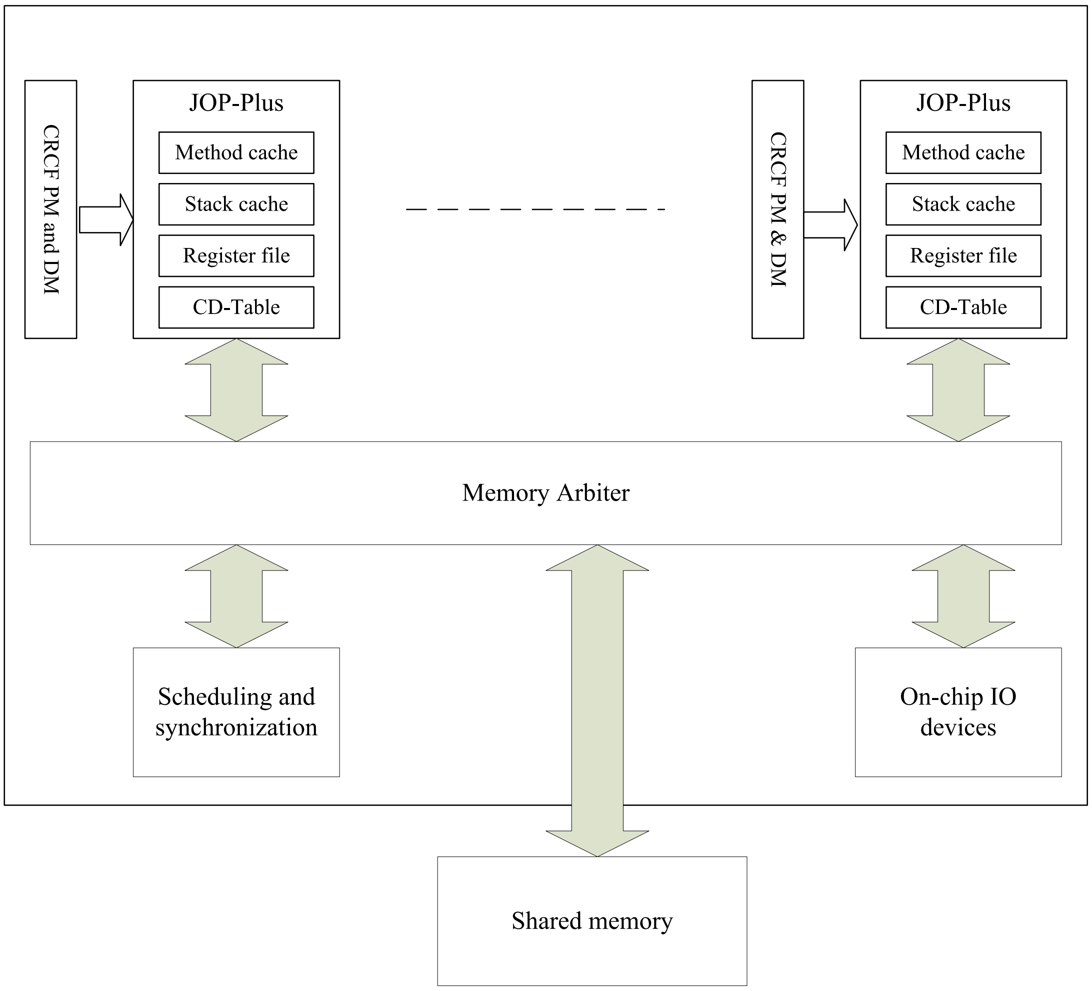


Figure 6.1: GALS-CMP architecture

GALS-CMP implements a Shared Memory Model where all the cores are connected to a single global physical memory. The JVM run time data areas are implemented in this physical memory which is equally accessible to all processors. This memory holds the application program code as well as the data in the form of objects. The memory area holding objects is termed as heap. This enables simple data sharing through a uniform mechanism of reading and writing shared structures in the common memory.

6.5.1 Interconnect Fabric

All the cores are connected into the system with shared memory via simple SoC interconnect (SimpCon) [86, 100] which provides point-to-point interconnections between components. All of the cores communicate with each other through the shared memory. SimpCon does not support synchronization of connecting multiple masters to a shared slave, therefore, a central arbiter is introduced to resolve possible emerging conflicts of parallel accesses to the shared memory. The master (core) starts the transaction by placing the read or write request and address which are valid for one cycle. The data by the slave (memory) is also valid for one cycle. If the slave needs the address or data longer than a cycle, it has to store it in a register. Consequently, the master can continue to execute its program until the result for a read operation is needed.

6.5.2 Arbitration

Multiple masters (cores) may try to access the shared memory at the same time resulting in a possible conflict as the specification does not support synchronization of connecting multiple masters to a shared slave. The possible emerging conflicts of parallel accesses to the shared memory are resolved by using an arbiter which controls the access of the various cores to the shared memory. The arbiter is connected to the core (a master) and the slave memory through the SimpCon interface. The arbiter acts as slave for each core and as a master for the shared memory. The arbiter introduces zero cycle latency for a transaction. Transaction is an individual, indivisible operation which must succeed or fail as a complete unit; it cannot remain in an intermediate state. The JopCMP implements both dynamic and static arbitration policies in the form of fixed priority, fair and time division multiplexing. The fixed priority as a dynamic arbitration policy resolves simultaneous access at runtime. The fair arbiter distributes a workload evenly among all the processors. The time division multiple access (TDMA) scheme is a static arbitration policy and strictly defines the access pattern and does not require any arbitration during the execution time. The TDMA based policy guarantees a constant bandwidth to each processor and is well suited for time predictability in multiprocessor systems. Each processor gets an allocated time slot for accessing the shared memory due to which it becomes possible to predict the memory access pattern and hence the execution time [123]. We have opted for the TDMA based arbiter as we intend to provide a time-predictable execution environment for embedded real-time systems in future. This memory arbitration scheme allows for a calculation of upper bounds of Java application worst-case execution times, depending on the number of CPUs, the time slot size, and the memory access time. The GALS-CMP inherits the attribute of predicting the execution time for JCF execution and can be extended to include CRCF as well.

6.5.3 Distribution of Processing

Both control-dominated and data-dominated processing is split at the clock-domain boundaries. This means that an entire clock domain must execute on the same processing core. However, a single processing core may execute any whole number of clock-domains. The allocation of clock-domains to the processors is done statically at the compile time. The designer is offered with the choice of allocation of clock-domains to different processors. Once WCET becomes available in future, the clok-domains distribution can be made based on the outcomes.It should be noted that an entire clock-domain must be scheduled on the same core as splitting clock-domain over several cores due to compiler constraints. Furthermore, no mechanism is specified for communication between reactions of same clock-domains splitted over several cores.

6.5.4 Inter Clock-domain Communication

The distribution of clock-domains on different cores gives rise to the problem of communication between the clock-domains. In SystemJ, all communication between clock-domains must take place through channels, which are implemented as Java objects shared by the respective clock-domains. The exchange of data using channels on traditional JVM based processors is implemented by passing Java objects between Java methods and classes on the same JVM, which entails passing references. All the cores of GALS-CMP are connected to the single shared memory and the communication between clock-domains is handled by passing object references of channels in Java. The shared memory also contains channel status signals to implement the rendezvous used in data exchange. The inter clock-domain communication in GALS-CMP is more efficient than the one adopted by the multiprocessor system in [39]. The two clock-domains communicating each to the other and executing on different processor have more expensive communication mechanism in terms of execution time and memory requirements as channel objects need to be transferred physically instead of passing references. In case of GALS-CMP, channel communication takes place by passing object reference to the other clock-domain irrespective of whether they are allocated to the same core or on different cores.

6.5.5 Accessing Shared Channel Objects

The inter clock-domain communication takes place through channels which are implemented as shared objects and reside in a single shared memory. In order to ensure the data consistency, the simultaneous access to the shared objects and class variables is managed by protecting access to shared objects. This is achieved through the synchronization unit by using the locking mechanism. There is one global lock available for the heap. If a core needs to access the shared object implementing the channel, it requests for the lock. In case no other core is accessing the heap, the lock is granted. Otherwise, it must wait until the other processor completes accessing the shared object. Once core acquires the lock, it resides in the critical section and cannot be interrupted. After the core has processed the shared object, it releases the lock. This lock can be acquired by a waiting core which had been rejected the grant of lock previously as it was being used by the other core.

6.6 GALS-CMP Architecture

This Section gives the description of GALS-CMP architecture including the memory organization, JOP-Plus core and clock-domain table (CD-table).

6.6.1 Memory Organization

Each GALS-CMP core consists of a number of memories; *method* cache, stack cache, microcode ROM, and translation-table as shown in Figure 6.1. Other memory components such as Register-File, CRCF program memory, CRCF data memory and a number of registers are part of the CRCF execution mode. The CRCF program memory holds the compiled and assembled code implementing concurrency and control flow of a SystemJ program. The memory is 16-bit wide RAM, which is enough to store an instruction word without operand. The memory depth is parameterized depending upon the code size, but maximum size is limited to 64K and this is the maximum number which a 16-bit operand in the second word of instruction can hold as target address. There are two sources of program instructions: CRCF program memory and *method* cache. Therefore, the CRCF program memory shares the program counter register with the *method* cache. The *method* cache serves as the instruction cache for JCF. The byte-codes for a complete method are loaded into the cache before execution.

The CRCF data memory (DM) contains the data structure for the concurrency and control flow statements of the SystemJ program (CRCF) as shown in Figure 6.2(c). The DM is 16-bits wide and the depth is not fixed and maximum size depends on the number of clock-domains. The Register-File (RF) is used for temporary storage of the operands when executing CRCF and is functionally the same as the Register-File in [38, 39]. The Register-File consists of 16 16-bit registers, which are not visible in the JCF mode. The *method* cache holds a complete Java method prior to its execution. The clock-domain table holds the address of the record of a particular methods and is described in details in Section 6.6.3.

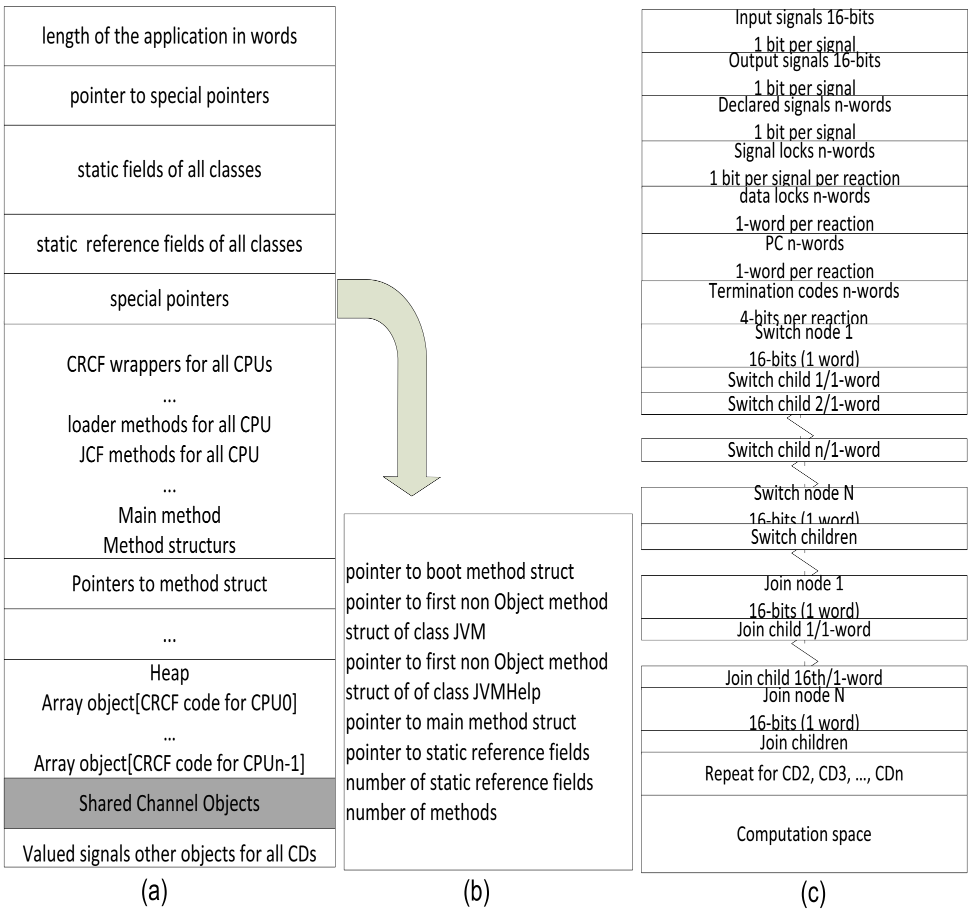

length of the application in words
pointer to special pointers
static fields of all classes
static reference fields of all classes
special pointers
CRCF wrappers for all CPUs
...
loader methods for all CPU
JCF methods for all CPU
...
Main method
Method structurs
Pointers to method struct
...
Heap
Array object[CRCF code for CPU0]
...
Array object[CRCF code for CPUn-1]
Shared Channel Objects
Valued signals other objects for all CDs
pointer to boot method struct
pointer to first non Object method struct of class JVM
pointer to first non Object method struct of of class JVMHelp
pointer to main method struct
pointer to static reference fields
number of static reference fields
number of methods
Input signals 16-bits 1 bit per signal
Output signals 16-bits 1 bit per signal
Declared signals n-words 1 bit per signal
Signal locks n-words 1 bit per signal per reaction
data locks n-words 1-word per reaction
PC n-words 1-word per reaction
Termination codes n-words 4-bits per reaction
Switch node 1 16-bits (1 word)
Switch child 1/1-word
Switch child 2/1-word
Switch child n/1-word
Switch node N 16-bits (1 word)
Switch children
Join node 1 16-bits (1 word)
Join child 1/1-word
Join child 16th/1-word
Join node N 16-bits (1 word)
Join children
Repeat for CD2, CD3, ..., CDn
Computation space
(a)
(b)
(c)

Figure 6.2: GALS-CMP memory organization (a) main meory layout (b) is a snapshot of special pointer area of main memory (c) layout of CRCF data memory

The main memory is situated external to the core and can be divided into two parts: application (program) area and heap as shown in Figure 6.2(a). The application area consists of per-class structures such as run-time constant pool, field and method data, and the code for methods and constructors. The heap is a run-time data area from which memory for all class instances and arrays are allocated. The detailed organization of the main memory is given in Figure 6.2(a). The application area also contains the code for JCF methods, loader method and CRCF wrapper for each CPU and the main method. The channels are implemented as the Java objects and reside in the heap part of the main memory. These objects are shared among the clock-domains running on the same or different cores. The access to the objects by the multiple concurrent clock-domains on different cores is synchronized by using the lock which guarantees the mutual exclusion.

6.6.2 JOP-Plus Core

The core used as basis for the GALS-CMP provides a seamless integration of both control and data execution in one processor [39]. The control execution is capable of invoking the data-computations in Java, which then returns to the control directly. The main idea is to provide support for the two separate components of the control flow in SystemJ programs, Concurrency and Reactive Control Flow (CRCF) and Java Control Flow (JCF), by extending the instruction set of the original JOP while using single execution unit and data-path. This allows resultant core to appear as two "logical" processors, or, alternatively, as a processor executing in two different modes of execution. At any given time, the processor executes SystemJ program in either of the two modes and uses all the resources of processor. The code parts related to executing CRCF and JCF are stored in two different memories. The CRCF can call a JCF method to perform some data-computations. The data call information contains the data-lock position and *id* of the Java action node which is used to invoke the corresponding method. After data-computations are performed, the JCF returns the result of computations to the CRCF which is written to memory location pointed to by data-lock position.

Each JOP-Plus has a set of IO devices needed for runtime system and only serial interface for program download and a *stdio* devices are connected to the first core. Additional IO devices can either be connected to one core locally which does not put any demand on main memory bandwidth or they can also be shared by all/some cores in that case they consume the bandwidth. They can be connected to the main memory arbiter in the same way as the memory controller as IO devices are memory mapped and standard synchronization for the access is needed. An interrupt line of an IO device can be connected to a single core or to several cores. Theses hardware interrupts, apart from the timer interrupt, are the asynchronous events with an associated thread which are normal schedulable objects, subject to the control of the scheduler. Interrupt handlers are implemented as special bytecode resulting in a call of a JVM internal method in the context of the interrupted thread. This mechanism implicitly stores almost the complete context of the current active thread on the stack.

6.6.3 CD-table

The CD-table holds the addresses of the first JCF method structure of each clock-domain which helps in calculating the address of any method being invoked directly from CRCF. In the GALS-CMP architecture, each processor core executes different clock-domain(s); therefore, each processor needs to store the base address of corresponding JCF. All these addresses are stored in a CD-table. The clock-domain number acts as the index of the table and each entry in the table consists of the address of the structure of the first method of that clock-domain. All the JCF methods, each representing a Java action node, are arranged in ascending order making it possible to calculate the address of structure of any method provided its offset from the base method is given. This offset is provided during the data-call. The CD-table, implemented in a RAM has parameterized number of entries equal to the number of clock-domains. The clock-domain number acts as the address for the RAM and address of the structure of base method for that clock-domain is read which is used to calculate the address of the desired method to be invoked.

6.7 GALS-CMP Compilation and Execution flow

Given the CRCF and JCF code generated by SystemJ compiler and a clock-domain to JOP-plus core mapping, the CDA tool integrated into compiler backend, takes the code generated by the original compiler and partitions it into multiple CRCF and JCF codes, one for each core. One sequential CRCF code and JCF code is generated for each processor. For multiple clock-domains mapped onto one core, the corresponding clock-domains are executed in a round-robin fashion.

6.7.1 SystemJ Example

Figure 6.3 gives an abstract representation of a SystemJ program that implements an asynchronous protocol stack. This program is readily runnable on any processor with JVM. It has three clock-domains; the first clock-domain models the packet generation from a network device, while the second and third clock-domains implement the main functionality of protocol stack [90], which can be extended to model complete communication stack like TCP/IP. There are three synchronous parallel reactions in each *protocol_stack* [25] clock-domain: *Assemble*, *Checkhdr* and *Stack*. These reactions assemble the incoming packets, check the header information of the sent packet and perform CRC check, and finally parse the packet and do computations, respectively.

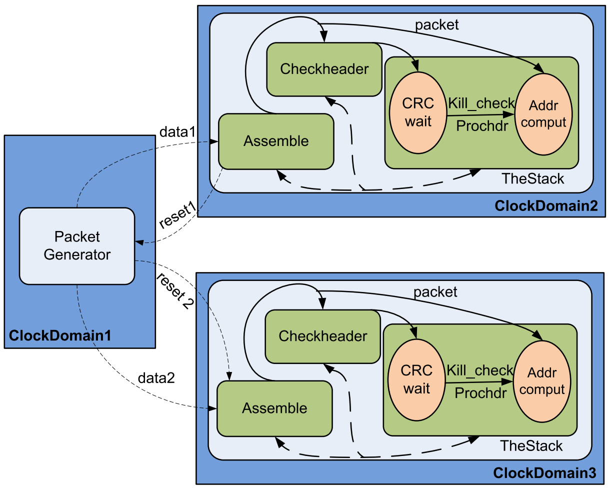


Figure 6.3: Pictorial representation of SystemJ example

The SystemJ code for a protocol stack clock-domain is shown in Figure 6.4. The second stack and packet generation itself have been abstracted out. The packet generation clock-domain, which creates a test-bench for the protocol-stacks, first resets the running protocol stack. Then, it initializes a byte array (*tosend*) with some packets and sends the packets to the protocol stack.

```
system{
  interface{
   input int channel reset1;
   output int channel reset1;
   output byte channel data1;
   input byte channel data1;
   input int channel reset2;
   output int channel reset2;
   output byte channel data2;
   input byte channel data2;
  }
  {
   {TestBench(reset1,data1) }
   ><
   //TheStack(reset1,data1)
   {
    signal packet1,kill_check1;
    signal res11,res12,res13;
    Integer signal crc_ok1;
    Asproto buffer1 = null;
    {
       while(true){
       receive reset1;
       int u1= 0;
       u1 = #reset1;
       if(u1==1){
         emit res11; emit res12;
         emit res13;
       }
       pause;
      }
    }
    || //Assemble
    {
       int cnt1=0;
       buffer1 = new Asproto();
       while(true){
         abort(res11){
          int e1 =0;
          trap(T){
           int len12 = Asproto.PKTSIZE;
           if(e1 == len12){
               exit(T);
            }
            else{
             receive data1;
             byte t1 = 0; t1 = #data1;
             e1=e1+1;
             buffer1.setRaw(e1,t1);
            }
          }
          emit packet1;
          }
         pause;
        }
      }
    ||
    {
      int crc1=0;
      while(true) {
        abort(res12){
          await(packet1);
         crc1 = buffer1.computeCRC();
          int val1 = 0;
         val1 = (crc1 == buffer1.getCRC()) ? 1:0;
         emit crc_ok1(val1);
        }
        pause;
      } // while
    }
    ||//Checkcrc
    {
      int match_ok1=0;
      while(true){
        abort(res13){
          await(packet1);
          {
             abort(kill_check1){
            //Some length computation
            match_ok1 = 1;
            pause;
            pause;
            pause;
            pause;
            }
         }
         ||
         {
            await(crc_ok1);
           int re1 = 0;
           re1= #crc_ok1;
           if(re1==0){
             emit kill_check1;
           }
           }
          int there1 = 0;
          here1 = #crc_ok1;
          if(there1==1 && match_ok1==1){
           System.out.println("Address match1!");
          }
        }
        pause;
        }
      }
     }
     ><
      //Second protocol stack
     }
     }
    }
```

Figure 6.4: SystemJ code for Asynchronous Protocol Stack

Upon compilation, the CRCF is separated from the JCF. The JCF code is presented in Java and consists of a number of methods performing some data-computations. Each JCF method represents a Java action node in AGRC and has a unique *id*. The methods are organized in JCF with respect to their *id*. The method corresponding to the first Java action node of a particular clock-domain is referred to as base method of that clock-domain which acts as reference to calculate the position of the other JCF method in the main memory.

6.7.2 Compilation Flow

The compilation and execution flow for GALS-CMP is shown in Figure 6.5. The GALS programs described in SystemJ are compiled by using the approach where CRCF is separated from JCF. The CRCF code is compiled first; the resulting code is wrapped in an array and stored in heap. On start up the compiled CRCF code is stored in the program memories of the respective processor cores. The allocation of clock-domains to the processors is done statically at the compile time. The designer is offered with the specific choice of executing the clock-domain(s) separately on different processors. The allocation takes place at in the main method of the core with zero *id*, responsible for downloading the application code into the main memory. When the other cores are enabled, they run the clock-domain method as their main method.

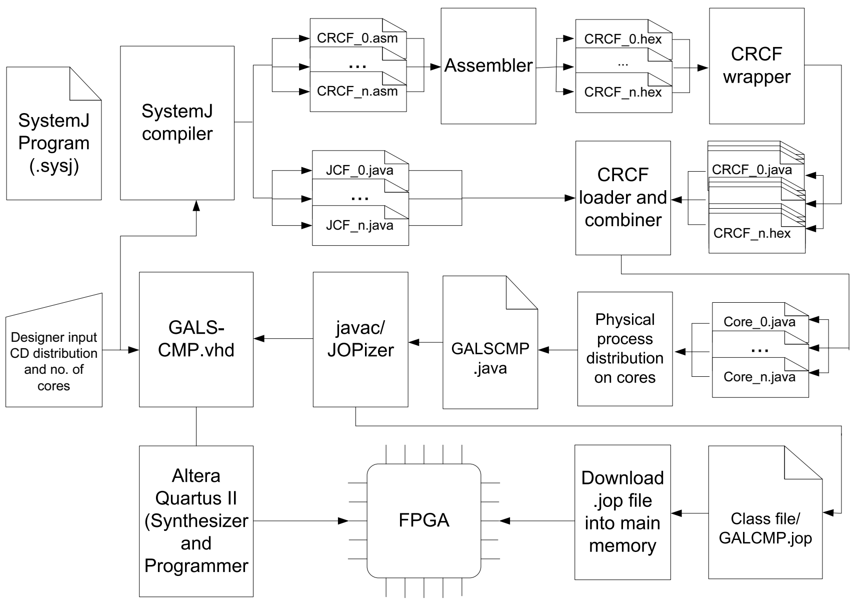


Figure 6.5: SystemJ compilation and execution flow

The compiler generates separate CRCF (*CRCF.asm*) and JCF (*JCF.java*) code for each processor consisting of the CRCF and JCF codes for the clock-domains assigned to this particular processor as shown in Figure 6.6. The CRCF assembly code (crcf.asm) is shown in Figure 6.6a and JCF Java code is shown in Figure 6.6b.

```
  // Assembly code for crcf.asm
 L1 SEOT
    CER
    LDR R0 $0001
    AND R0 R0 #$f000
    SSOP R0
    LSIP R0
    AND R0 R0 #$0
...
   LDR R1 $0002
   AND R1 R1 #$0001
   PRESENT R1 L4
   LDR R0 #3
   ADD R1 R6 #0
   STR R1 #0
   SENDATA R0 // calling Java action node
   ADD R4 R4 #1
...
   L2 LDR R0 R1
   CLFZ
   SUBV R0 R0  #0
   SZ L3
   JMP L7
   L3 CLFZ  ;
…
```

(a) CRCF assembly code

```
//Java code for JCF (Jcf.java)
Public Class Jcf{

 public static int u=0;
  …
//JCF method corresponding to Java
  action node
 private static method_0(){
    System.out.println("cd0");
 }

 private static method_1(){
     u=5;
}
private static method_2(){
     System.out.println (u);
}

…

private static method_n(){
      System.out.println (u);
 }

}
```

(b) Java action nodes as JCF methods

Figure 6.6: SystemJ code compiled by separating data-computation and control in the form of JCF Java code and CRCF assembly code

The assembler generates the CRCF machine code and generates *.hex* file (*CRCF.hex*). The CRCF wrapper fragments this code and wraps them into Java arrays (*CRCF.java*) as shown in Figure 6.7.

```
//CRCF.java
// The CRCF machine code encapsulated
// in array objects and stored in heap
// before downloading into CRCF program memory
// each array contains
public class CRCF{
 int a;
 public static int[] cc = {
  0x3400,0x4100,0x0000,0x4090,
  0xFFFA,0x40A0,0xFFFF,0x4070,
  …
 }
}
```

Figure 6.7: CRCF machine code encapsulated in arrays

The Java code responsible for downloading the CRCF code, called CRCF loader, is generated and combined with the *Jcf.java* to produce the code to be executed on each core (*core.java*) as shown in Figure 6.8. All the signals and variables local to these clock-domains being executed on the same core are declared here.

```
//Core.java
// contains the CRCF and JCF code
//to be executed on a core
public class Core {

//local signals and variable declarations

 private void jcf() {
  System.out.println("cd0_m0");
  Native.rddatacall(1);
 }

 ...
 }
// code for downloading CRCF
 public static void corestart() {
//create an instance of CRCF code enclosed
// in array
  code = new CRCF();
  int length = code.cc.length;
  for (i=0; i < length; i++){
    crcf_instr=code.cc[addr];
// read instruction from heap and stored it in
// CRCF program memory
    Native.initctrl(addr, crcf_instr);
    addr=addr+1;
  }
  System.out.println("initialize CRCF PM!");
 // CRCF program download complete
 // switch the execution mode
  Native.switchmode();
 }
}
```

Figure 6.8: Java code to be executed on a core comprising of code for downloading CRCF machine code into CRCF program memory and JCF code representing Java action nodes

Finally, all the clock-domains are assigned to the respective cores in the main method of class *Galcmp.java* as shown in Figure 6.9. The shared channel objects, through which the clock domains communicate, are declared inside this class.

```
// Galscmp.java
 Public Class Galscmp{
//shared channel objects are declared here
  public static void main(String[] args) {

    SysDevice sys = IOFactory.getFactory().getSysDevice();
    // start the other CPUs after boot up
    sys.signal = 1;
    // get cpu id
    cpu_id = Native.rdMem(Const.IO_CPU_ID);
    if (cpu_id == 0x00000002){
     // each core execute respective clock-domain(s)
        core2.corestart();
     }
    if (cpu_id == 0x00000001){
        core1.corestart();
    }
    core0.corestart();
    }
 }
}
```

Figure 6.9: Allocation of clock-domains to cores in main method of the class

The application class is compiled by *javac* and produces the class file which is further processed by the JOP specific tool to produce the GALS-CMP final code. It also generates the VHDL code for CD-table. The application is downloaded into main memory by the core with *id* zero. Once the start up is completed, all the cores start executing respective clock-domains. Each core starts in Java mode and downloads the CRCF code from the heap into CRCF program memory. After the CRCF program memory is initialized, the core switches the mode and starts executing in CRCF mode by fetching the instructions from the CRCF program memory.

6.7.3 Boot Sequence

In GALS-CMP, each processor is assigned a unique identity number (CPUid). The boot-up process is similar to JOP-Plus and is the same for all processors until the generation of the internal reset and the execution of the first microcode instruction. From that point on:

- Only one processor performs the initialization steps and processor (*id=0*) is designated to do all the boot-up and initialization work.
- The other CPUs have to wait until CPU0 completes the boot-up and initialization sequence.
- At the beginning of the booting sequence, CPU0 loads the Java application.
- Meanwhile, all other processors are waiting for an initialization finished signal from CPU0. This busy wait is performed in microcode.
- The additional CPUs will invoke a system method assigned to them in the main method. Then each CPU initializes its CRCF memory with the respective CRCF code from the heap.
- Once initalization is completed, the processor switches the execution mode
- The execution of the application starts in the CRCF mode.

6.7.4 Instruction Fetch

There are two different memories which hold the program code for two different modes: CRCF program memory and *method* cache for the CRCF and JCF modes, respectively. The program memory source for the next instruction to be fetched is controlled through *mode_control* flag, which defines the mode of operation. If *mode_control* flag is set, the next instruction to be executed is always fetched from the CRCF program memory. On the other hand, resetting of the flag results in fetching of the next instruction to be executed from the *method* cache. This flag is set and reset while switching from the CRCF mode to JCF or vice versa. Both memories share the program counter; therefore, its value is saved when switching to the JCF mode. The switching from JCF to CRCF mode does not require storing the address as memory subsystem provides the method start address in the cache every time a JCF method is loaded into it.

6.7.5 Transfer of Control to/from JCF

When invoking a JCF method to perform data-computation, the address of next (returning) CRCF instruction is stored in a register and the address of the structure of the base method of the clock-domain to be executed is fetched from the CD-table as discussed in Section 6.6.3. The address of the structure of the method to be invoked is calculated using the base addresse provided in the CD-table. The address calculation mechansim for invoking a method is similar to JOP-Plus and has been discussed in detail in Chapter 5. The structure record of a method resides in main memory and holds the information such as constant pool address, argument count, variable count, method start address and code length, in the encoded form. The method start address and code length are extracted and the information is passed to the memory subsystem for loading the method code into the cache. At the same time, the *mode_control* flag is reset making the *method* cache default memory for read/write operations. The next byte-code is fetched from the *method* cache and JCF method execution starts. When returning from the JCF to the CRCF mode, the result is available as the top element of stack. The result is written to the CRCF data memory at a location pointed to by data-lock position. The address of next CRCF instruction to be executed is stored in a register; it is loaded into Java program counter and *mode_control* flag is set. The next instruction is fetched from the CRCF program memory, and *method* cache does not need to be loaded with CRCF program code as it permanently resides in the CRCF program memory.

6.8 Performance Evaluation

This section presents the results of experiments conducted to evaluate and compare our proposed GALS-CMP multiprocessor architecture with a single processor approach, called base core, to execute the SystemJ programs.

6.8.1 Experimental Setup

All presented data have been collected from the experiments carried out by using the cycle-accurate ModelSim simulator for 2,3 and 4 core systems all running at 50 MHz clock. The system is capable of running at frequency higher than 50 MHz but the results presented are for 50 MHz clock for fair comparison with earlier published results. The benchmarks are selected to show the effectiveness of the approach for the GALS program execution. The benchmarks include asynchronous examples which are heterogeneous in nature both with and without involving any clock-domain communication.

6.8.2 Benchmarks

The benchmarks without any inter clock-domain communication include the two and four clock-domain version of *demoloop* (*dl*) [89], *dl2* and *dl4*, respectively, which effectively means running two or four synchronous programs in parallel. It consists of two or four identical clock-domains each comprising of four reactions and these clock-domains run independently without involving any communication between them. These benchmarks have minimal data-computations, and fully parallelized application without any accesses to shared data structures and synchronization needs. The asynchronous case is represented by an *asynchronous protocol stack* (*aps*) [90] and *pump controller* examples. The *aps* example has two versions: 1) *aps* has two clock-domains and 2) a variant *aps3* has three clock-domains. In both cases, the first clock-domain is used to model the packet generation process, and second clock-domain implements the stack itself as mentioned previously. The *aps3* implements two protocol stacks presenting a case where network generator sends data at a rate higher than what can be handled by a single protocol stack. Therefore, the generator sends to two different stacks alternately. The *pump controller* example consists of two clock-domains and nine reactions in total. It models the control of a pump inside a mine which may have high methane levels. The pump pumps out water whenever the water level exceeds the desired level and is turned on only if methane level is below a certain limit. Whenever methane level goes above that limit, the controller must stop the pump and wait until right methane level is restored. If methane level goes too high, then the pump is stopped immediately and an ALARM is generated.

6.8.3 Performance Parameters Definitions

The execution speed comparisons are given in terms of the average response time of the clock-domain and the application execution time. The clock-domain response time is defined as the average time taken by the clock-domain to respond to the environment at the end of its logical tick. A logical tick is the time interval between two logical time consuming statements and may have variable time depending on the amount of computation enclosed between these two statements. The application response time is defined as the time between the application input sampled and final output generation and may involve multiple logical ticks. It takes into account the time needed by the cores and in addition the time taken for communication between the processing elements to exchange the information, if needed.

6.8.4 Reponse Time Comparison

The results in Figure 6.10 show that GALS-CMP outperforms the single processor execution in terms of response time. The GALS-CMP with 2 cores has between 2 and almost 4 times reduction in response time when executing the benchmarks which do not involve any clock-domain communication and have minimal data-computations. This can be attributed to the following factors: firstly, the use of multiprocessor system and secondly, the local storage of the CRCF program as provided in the base core. The control dominated programs contain minimal data-computations, so they do not need to access the main memory to fetch the JCF method into the cache. Therefore, the gain is almost proportional to the number of processor cores. Increasing the number of clock-domains increases the response time, which is evident from the results for the benchmarks *aps* and its variant *aps3*. The improvement in response time is achieved through parallel execution of the clock-domains. The clock-domains run independently and respond to the environment at the end of each of their logical ticks which is in contrast to the single processor approach where clock-domains are executed cyclically one after the other. The clock-domains in the later case are able to respond to the environment signals only after all the clock-domains have finished their execution resulting in increased response time. The GALS-CMP is expected to have degradation in the clock-domain tick time due to the shared memory as the bandwidth is divided equally among all the nodes of the GALS-CMP. In case of control-dominated applications with minimal data-computation, the tick time is not expected to degrade by much as the code implementing the concurrency and control flow is stored in a separate local memory and, therefore, it is not affected by the constraints on memory bandwidth due to shared memory.

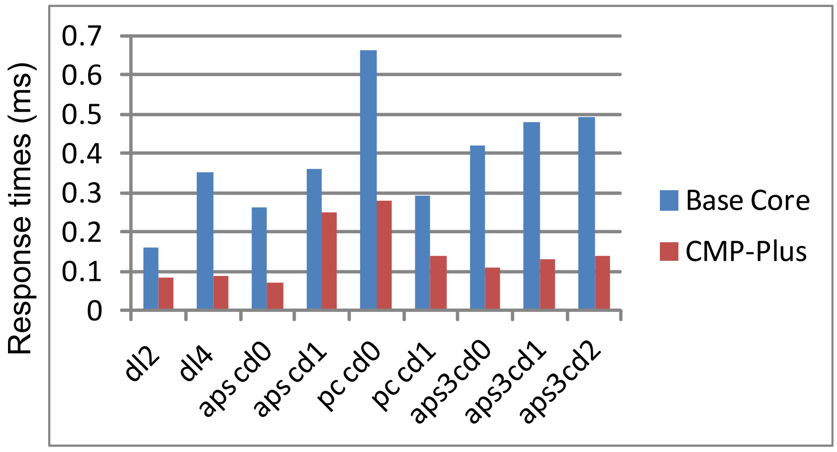


Figure 6.10: Comparison of clock-domain response time

6.8.5 Application Execution Time Comparison

The application execution time means time required to fully execute an application from start to end once. Figure 6.11 shows the reduction in application execution times for benchmark examples against the single processor execution. The results indicate that the application execution times are improved by almost 100% when we migrate from single core to two core system when executing control dominated applications such as *dl2*. The reason for this improvement is the concurrent execution of clock-domain as compared to single core execution which can execute only one clock-domain at any time, therefore, clock-domains are scheduled cyclically resulting in larger execution times. But this gain is not linear when going from 2 to 4 core systems as evident from *dl4* benchmark. This is due to the constraints on shared memory bandwidth. It should be noted that *dl2* and *dl4* are run on a two and four core systems, respectively. The *aps* and *pump controller* examples are 85% and 89% faster, respectively. Both of the examples involve the channel communication; therefore, some of the time is consumed in physical transfer of data over channels. Further addition to this time is the fact that the channel objects reside in heap, which is implemented in shared main memory resulting in delayed access due to memory bandwidth constraints.

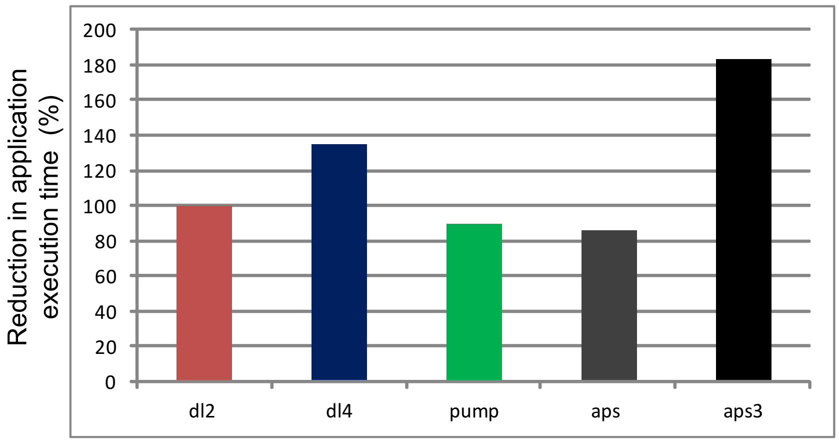


Figure 6.11: Comparison of application execution time of GALS-CMP and and base core JOP-Plus

6.9 Summary

We have described a new multiprocessor platform, GALS-CMP, for the execution of concurrent programs written in the GALS programming language SystemJ. The multiprocessor arhcitectur is based on the JOP-Plus core developed during the course of this research and described in chapter 5. The GALS-CMP platform fits well with the GALs MoC based applications described in SysetmJ where a system comprises of multiple clock-domains running independently. The communication between clock-domains is performed through shared memory by passing references. The synchronization to shared objects is achieved through locks. We demonstrated the effectiveness of the approach by running different benchmark examples and comparing the measurements against the single processor approach. This processor outperforms other execution platforms for SystemJ in terms of clock-domain response-time and the overall application response time. We believe that this solution will fullfil the requirements laid down by the high-end embedded market.

Chapter 7

Conclusion and Future Work

The embedded systems are becoming more and more complex, heterogeneous and distributed in nature. They are being deployed in critical applications and demand real time operation. There has been a surge for efficient modeling of such systems on one hand, and their execution platforms on the other hand. The problem of modeling such systems is being tackled to some extent by raising the level of abstraction. The system level language called SystemJ targets such systems and is based on the Globally Asynchronous Locally Synchronous (GALS) model of computation. It provides tight coupling of control and data-driven transformations, and represents both asynchronous and synchronous concurrent processes in an abstract way. Contrary to the modeling, the efficient execution of systems described using GALS MoC is a research problem which has largely been overlooked. It is required to be explored and needs serious attention. This thesis focuses on the exploration of the architectures for efficient execution of application described using SystemJ.

This chapter starts with a concise summary of the major goals described in this thesis. Furthermore, the research course is outlined and the main findings and results are presented. The conclusion will demonstrate the relevance of this thesis to current scientific work and give some ideas for future research.

7.1 Overview

The main goal of thesis is to propose new architectures, as well as improving upon to the existing architectures, to efficiently execute the programs described in the language targeting the Globally Asynchronous Locally Synchronous (GALS) paradigm. These architectures execute control and data-driven operations along with asynchronous and synchronous concurrent processes in an efficient way. The architectures developed are amenable to the Worst Case Reaction

Time calculation. The goal of this thesis is to achieve performance and efficiency approaching that of application specific systems.

7.2 Results Discussion

Figure 7.1 shows the embedded platforms deployed for the execution of SystemJ programs. The platforms shaded greys are the 6 new platforms introduced and five of them were the outcome of this research. A number of execution platforms have been adopted or developed for SystemJ program compiled using two approaches: 1) pure Java where SystemJ program is compiled to standard Java 2) split approach where control is separated from data-computation. Previously, SystemJ programs compiled to pure Java were executed on a general purpose processor running an interpreting JVM. As a first step towards the development of efficient platform, we introduced a Java processor in the form of JOP. Being hardware JVM, JOP performs better than the general purpose processor which interprets the Java byte-code through a JVM running on it.

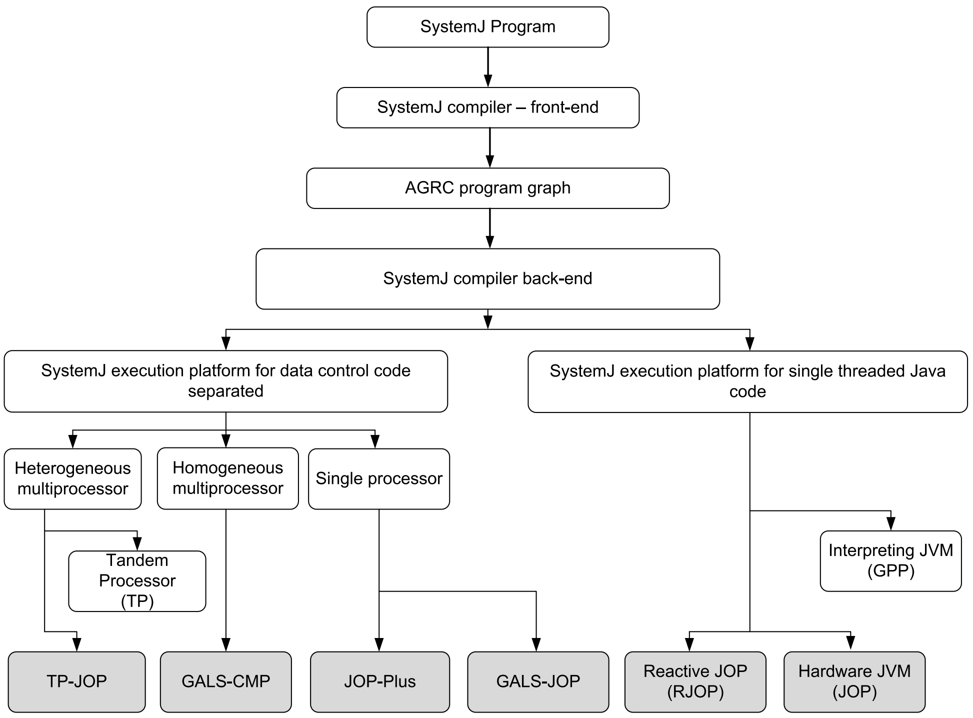
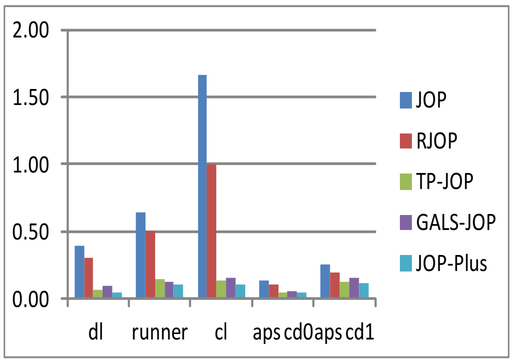

SystemJ Program
SystemJ compiler – front-end
AGRC program graph
SystemJ compiler back-end
SystemJ execution platform for data control code separated
SystemJ execution platform for single threaded Java code
Heterogeneous multiprocessor
Homogeneous multiprocessor
Single processor
Tandem Processor (TP)
Interpreting JVM (GPP)
TP-JOP
GALS-CMP
JOP-Plus
GALS-JOP
Reactive JOP (RJOP)
Hardware JVM (JOP)

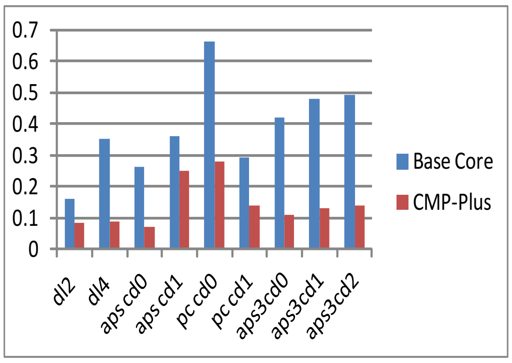
Figure 7.1: SystemJ program compilation and execution approaches. The shaded areas represent the approaches suggested in this thesis

The separation of control from data-computation not only improves the execution time of programs but also reduces the memory footprint [38]. SystemJ programs compiled using such an approach generated control code as a special instruction set and data-computation code was generated in Java. They were executed on a custom control processor and executed on a general purpose processor, respectively. This dual or two-processor approach, called TP, made use of heterogeneous elements to perform control-oriented and data-oriented computations concurrently. The results provided in Figure 7.2a indicate that JOP outperforms the TP and is up-to 11 times faster than the TP despite the fact that the TP benefits from the compilation approach where control-oriented operations are separated from the data-oriented operations. We achieved further improvement in the execution of SystemJ program through Reactive-JOP (RJOP) - a variant of JOP obtained by enriching it with reactivity and signal manipulation in the hardware, which are the key features of the GALS applications described using SystemJ. The results show that RJOP on average is 20% faster than JOP as given in Figure 7.2a.

A hardware JVM performs better than interpreting JVM, therefore, this prompted to improve the TP architecture by replacing the general purpose processor with the Java optimized processor for the execution of Java code. This resulted in a more efficient execution platform, called TP-JOP, capable of executing both control-oriented and data-oriented operations natively thus removing the TP's bottleneck of slower execution of data-computations represented in Java due to interpreting JVM. The results in Figure 7.2a show that TP-JOP [95] is 6 to 50 times faster than the TP for the given set of the benchmarks.



(a) Execution times comparison



(b) Response time comparison for multiprocessor approach

Figure 7.2: Performance comparison

Although, TP-JOP was able to execute the SystemJ programs faster, but being a dual processor approach, used too many logic elements when implemented in FPGA. Also, frequent communication between control and data processors required a more efficient interconnection. The frequent communication together with the communication infrastructure posed many problems [39] while scaling to multiprocessor architecture. The execution flow of SystemJ application also suggested that the resources executing control and data-computations are underutilized as they block waiting for each other.

To overcome these issues, we proposed a single processor execution platform in the form of GALS-JOP. It translated the control program to Java statements. It gives acceptable performance while using fewer logic resources. It does not require communication infrastructure as both control and data-computations are executed on a single processor. The GALS-JOP is 2 to 10 times faster than the JOP, 3 to 53 times faster than the TP. TP-JOP performs better than GALS-JOP due to concurrent and native execution of both control-oriented and data-oriented operations. The compiler generates the control (assembly) and data-computations (Java) in two orthogonal programing models. The execution of both on GALS-JOP required the translation of one programing model (assembly) into the other programing model (Java) understandable by the base processor (JOP). We presented a solution to this problem by extending the base processor to support both programming models. The JOP-Plus processor provides a seamless integration of both programing models into one processor. One programing model is capable of invoking the other programing model and return to it directly. The JOP-Plus is 44% and 55% faster while executing the application with data-computation such as *runner* and *dl*, respectively. JOP-Plus improves data-computations represented in JCF execution by decomposing it into smaller methods for more efficient execution. JOP-Plus is up-to 125% faster than GALS-JOP with comparable resource usage.

The single processor approach can provide only a limited processing power. In order to achieve high computation power, we proposed GALS-CMP which is a shared memory homogeneous multiprocessor architecture comprising of many cores. This architecture is capable of exploiting the parallelism offered by the SystemJ applications at clock-domain level. The clock-domains run concurrently at their own on logic tick rate in truely GALS fashion. The results in Figure 7.2b indicate that the application execution times speed up by almost 100% when we migrate from single core to two core system when executing control dominated applications such as *dl2*. But this gain is not linear when going from 2 to 4 core systems as evident from *dl4* benchmark. This is due to the constraint on shared memory bandwidth. The *aps* and *pump controller* examples are 85% and 89% faster, respectively. Both of the examples involve the channel communication; therefore, some of the time is consumed in physical transfer of data over channels. Further addition to this time is the fact that the channel objects reside in heap, which is implemented in shared main memory resulting in delayed access due to memory bandwidth constraints.

Figure 7.3 shows utilization of resources for different target platforms. The JVM approach runs both control processing and data-driven processing on a single JVM. The three different platforms implementing the JVM are GPP (NIOS II), JOP and RJOP. GPP also includes all Avalon fabric with required arbiters, multiplexers and decoding logic to connect FPGA's internal and external memories. The GPP is the most expensive option and uses 32% more logic elements than JOP. The RJOP uses only 3% more resources than the JOP and is up-to 20% faster. For split approach, the TP-JOP is the most expensive and JOP-Plus is the most economical in terms of real state usage. When compared with TP, the TP-JOP, GALS-JOP and JOP-Plus use 15%, 41% and 42% fewer resources, respectively.

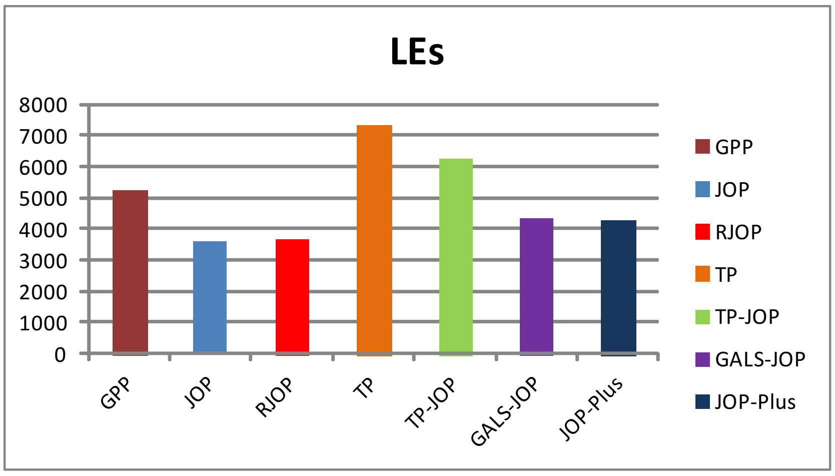


Figure 7.3: Resource usage comparison for all execution approaches

7.3 Main Contributions

The main contributions described in this thesis are:

1. *Accelerated and Time Predictable Execution of GALS Programs*: SystemJ uses Java to perform data-driven computations. It is also uses a compilation approach where concurrency and reactivity is compiled into single threaded Java code. Although Java provides a lot of advantages such a type safety, portability and automated garbage collection, it suffered from a major drawback. Java achieves these advantages at the expense of execution speed, especially, in the case of embedded implementation. As a first step towards accelerating SystemJ execution, we introduced the Java Optimized Processor (JOP) which is hardware implementation of JVM. It overcame the drawbacks of increased execution time due to interpreting JVM running on a complete traditional processor. The time predictable execution made it possible to deploy SystemJ in real-time systems.

2. *A Reactive Java Processor for the Execution of GALS Programs:* The JOP, inherently suited to data-driven transformational operations, is extended to efficiently execute the control constructs and control flow of SystemJ. The new core, which is called RJOP (Reactive JOP), efficiently executes both data dominated and control dominated embedded applications. It also maintains the time-predictable execution of the applications intended for real-time embedded systems and calculation of Worst Case Reaction Time (WCRT) as provided by the original JOP core. The results showed significant performance improvement and lower resource consumption over the existing architectures used for the SystemJ execution.

3. *A Heterogeneous Tandem Processor Architecture for GALS Programs Execution*:: The TP-JOP architecture is capable of executing both control and data computations natively.

This is an improvement over the TP execution platform which is used to execute the SystemJ program where control flow, which includes concurrency, is separated from the ordinary Java computations. The TP used a control processor in conjunction with a traditional processor. The interpreting JVM along with a complete traditional processor results in increased execution time and also higher logic usage. The TP-JOP, as the name indicates uses a full Java processor, for the execution of Java computations.

4. *Efficient Merging of Control and Data-computations:* A new processor, called GALS-JOP, makes very fine merger of tandem processor approach into a single and more economical processor. It facilitates efficient execution of synchronous and asynchronous concurrency and reactivity (control flow) and Java oriented data computations by merging the best of both worlds at low cost. Importantly, the design approach does not require any essential modification of the SystemJ compilation flow, which is based on a formal semantics, giving advantages over non-formal programming languages and their compilation approaches. GALS-JOP guarantees finding the worst case execution times for any program segment, and in particular case of SystemJ programs the worst case reaction times (WRCT). This approach required the translation of concurrency and control flow (CRCF) programming model represented as assembly instructions to the data-computation programming model represented as Java statements explained earlier.

5. *Execution of Control and Data computations with Distinct Modes of Executions:* We present a solution for low resource usage for executing programs compiled by separating the control and data-driven operations. These programs are executed on a single JOP-Plus core without requiring the translation of one programming model into another programming model. Instead, it provides support for both programming models in a single processor with two execution modes. This allows resultant core to appear as two "logical" processors, or, alternatively, as a processor executing in two different modes of execution. At any given time, the processor executes SystemJ program in either of the two modes and uses all the resources of processor. It can switch between two modes without any overheads.

6. *A Homogeneous Multiprocessor Architecture for Concurrent Execution of GALS Programs:* We presented a GALS-CMP multi-processor architecture for concurrent execution of programs that follow the Globally Asynchronous Locally Synchronous (GALS) formal model of computation. It consists of multiple JOP-Plus cores, connected to the shared memory through an arbiter. The SystemJ programs offer high degree of concurrency at clock-domain level which is exploited to boost the performance of the system. The GALS programs are partitioned at clock-domain boundaries and they are allocated to different cores. The exploitation of parallelism reduces the response time of the clock-

domains as well as application execution time. The multiprocessor system is suitable for the execution of heterogeneous complex embedded applications.

7. *Compiler Modifications:* The SystemJ compiler back-end is modified to target code generation for various platforms. We have provided a complete design flow for each target platforms that we have designed for efficient execution of GALS languages in general and SystemJ in particular.

8. *Experimental evaluation:* We evaluate the different architectures and validate their effectiveness by running benchmarks on them. Better performance, code density and resources usage compared to previous approaches for SystemJ execution, thus making it more suitable for heterogeneous embedded applications.

Thus, overall this thesis describes new execution platforms and development environment that can be used for efficient execution of GALS languages in general and SystemJ in particular.

7.4 Recommendations for Further Research

This thesis introduced a number of novel processor architectures for executing GALS program described in SystemJ. We presented the implementations of the architectures, components, and also validated the correctness of them via the evaluation. However, these architectures are initial prototypes. There is still some room to optimize or enhance these architectures and the related compiler as follow:

1. *Compiler optimizations:* The outcomes of the research suggest that SystemJ compiler can be optimized to produce more efficient code. These optimizations can be readily incorporated into the compiler. For example, the sequential execution of the control and data computations does not require the data-locks and testing of these locks in the code. The removal of the data-locks will simplify the code and reduce the memory footprint.

2. *Integrating SystemJ compiler within JOP tool chain:* In this research we have maintained the separation between the SystemJ compiler and JOP tool chain. We believe the integration of both will result in optimized system. For example, the SystemJ compiler generates the CRCF and JCF code. The JCF methods are invoked by the CRCF by calculating the addresses of the methods. After JCF compilation, the method addresses produced by the JOP tool can be used by the SystemJ compiler, which can perform another pass and replace the methods identifiers with their addresses.

3. *C based execution platform:* Keeping in view that the embedded systems are restricted by resources and time constraints, another option is to use C as implementation language

instead of Java. Applictions described in SystemJ and compiled to Java can be translated to C by using Java to C converters. This can be executed on any general purpose processor and performance comparison should be made with existing platfroms.

4. *Interrupt based TP-JOP:* In the current implementation of TP-JOP, JOP polls the CP all the time and cannot perform any useful task when idle in the absence of a data call. The utilization of JOP can be improved by replacing the polling mechanism with interrupt. The CP will interrupt the JOP when any data computation is required by placing a data call. The JOP will respond to this data call by performing the required data computation and starts executing the assigned task again. This will increase the utilization of JOP.

5. *Extension of WCET analysis for JOP-Plus:* As CRCF execution times can be exactly calculated and JOP worst case execution times are also predictable, we plan to extend this work to the analysis of the worst case response times and optimization for use in real-time systems. This works is in progress and results are expected very soon.

6. *Power consumption:* As power and energy consumptions are becoming decisive design criteria, we recommend that power consumption, as well as the energy efficiency of the processor must be analyzed by introducing power models. This can be used for power sensitive applications, to give feedback to designers about the peak or average power consumption of the designed system. It can also help in work load balancing in multiprocessor system.

7. *Reducing demand of shared memory bandwidth:* In the JOP-Plus architecture presented in Chapter 5, the CRCF code is stored in on-chip memory whereas JCF code is stored in the main memory which is shared among all the processing cores. The sharing of memory bandwidth results in the degradation of performance as the processing cores compete for the bandwidth to fetch the JCF code to method cache prior to its execution. Providing the local storage for JCF code will ease the pressure on the shared memory bandwidth.

8. *Hardware support for channels:* In GALS-CMP system, clock-domains execute concurrently and channel-based communication between clock-domains takes place through the shared memory. We plan to investigate hardware support for point-to-point channel communication among the processing cores.

9. *Lock-free synchronization:* One of the possible research areas may be to investigate different means of providing atomic access to shared objects implementing channels in the memory. The transactional memory can be an alternative to locks that provides lock-free synchronization and has become a popular research area. Transactional memory was originally proposed as a better solution than locks to shared-memory synchronization,

and avoids the problems of serialization and deadlocks involved with locks [124]. It was shown to also have a performance advantage over locks.

10. *Automatic allocation of clock-domain:* One of the research area can be development of methods for static compile time clock-domains allocation to a GALS-CMP architecture. The worst case execution times of the clock-domains can be reliably estimated at compile time, and clock-domain mapping can be made statically at the compile time. The designer will be freed from the responsibility of allocation and clock-domains will be allocated based on the performance parameter rather than random choice of the designer.

11. *Distributed multiprocessor system:* As mentioned in Chapter 6, a tightly coupled multiprocessor system consisting of multiple CPUs and a single global physical memory has a serious bottleneck: Main memory is accessed via a common bus, a serialization point that limits system size. Distributed-memory multiprocessors, however, do not suffer from this drawback. The system can contain many orders of magnitude more processors than a tightly coupled system. The communication required between the concurrent clock-domain can be implemented by a shared-memory abstraction on top of message-passing distributed-memory systems. Alternately, the point-to-point communication channels can be implemented which will keep the communication network simple and can be scaled to any required number of core.

12. *Extending reliability through clock-domain migration:* Failures are likely to be more frequent in systems with more processors. Therefore, schemes for dealing with faults become increasingly important. In future, we intend to incorporate fault tolerance solution for SystemJ based applications being executed on the chip multiprocessor system where clock-domains are run concurrently. The main idea here is to guarantee, once a failure occurs, that the executing tasks are migrated to other non-faulty core.

13. *Better machine interface development:* An efficient tool based on graphical user interface can be developed which is capable of generating and running the code on different target platforms with the press of a button. Since we have developed a number of platforms, the tool should compile the code for different target platforms and be able to load and execute the programs on these target platforms. The tool should also be able to simulate the codes generated for different target platforms. This tool will not only make user friendly but greatly reduce the development time and cost. This will also add to the commercial value of the System.

As indicated above, there are still lots of research topics for these architectures that can be explored in the near future. We also hope anyone who is interested in architecture feels free to discuss with us if they have any questions.

References

[1] M. Barr. Embedded systems glossary. [Online]. Available: http://www.netrino.com/Publications/Glossary/

[2] International Data Corporation. [Online]. Available: http://www.idc.com/

[3] G. E. Moore, "Cramming more components onto integrated circuits," in *Readings in computer architecture*, D. H. Mark, P. J. Norman, and S. S. Gurindar, Eds. Morgan Kaufmann Publishers Inc., 2000, pp. 56–59.

[4] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring, "Statemate: a working environment for the development of complex reactive systems," Singapore, pp. 396–406, 1988.

[5] S. Edwards, "The specification and execution of heterogeneous synchronous reactive systems," Ph.D. dissertation, University of California, Berkeley, 1997. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1997/3235.html

[6] J. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, pp. 10–19, October 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=50810.50811

[7] D. Chapiro, "Globally asynchronous locally synchronous systems," Ph.D, dissertation, Stanford University, , Palo Alto, CA., 1984.

[8] A. Malik, "Principia lingua SystemJ," Ph.D. dissertation, University of Auckland, Auckland, 2010.

[9] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," in *Readings in hardware/software co-design*, M. Giovanni De, E. Rolf, and W. Wayne, Eds. Kluwer Academic Publishers, 2002, pp. 86–107.

[10] D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[11] T. Grotker, *System Design with SystemC*. Norwell, MA, USA: Kulwer Academic Publishers, 2002.

[12] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modelling*. Secaucus, NJ, USA: Springer-Verlag, 2006.

[13] D. Gajski, A. Gerstlauer, R. Domer, and J. Peng, *System Design - A Practical Guide with SpecC*. Kulwer Academic, 2001.

[14] *Synchronous Programming of Reactive Systems - A Tutorial and Commented Bibliography*. Springer Verlag, 1998.

[15] G. Berry, "The Esterel v5 Language Primer," Tech. Rep., 1999.

[16] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE," vol. 18, no. 9, pp. 785–793, 1992.

[17] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with SIGNAL," vol. 79, no. 9, pp. 1321–1336, 1991.

[18] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[19] M. Browne and E. Clarke, *SML: A high-level language for the design and verification of finite state machines*. Grenoble, France: Carnegie-Mellon University, Department of Computer Science, 1985, 1986, vol. Volumes 85-179 of Research Paper, Carnegie-Mellon University Computer Science Dept.

[20] C. André, "Representation and analysis of reactive behavior: a synchronous approach," in *Computational Engineering in Systems Applications (CESA)*, 1996, pp. 19–29.

[21] F. Maraninchi, "Operational and compositional semantics of synchronous automaton compositions," in *Proceedings of the Third International Conference on Concurrency Theory*. Springer-Verlag, 1992, pp. 550–564.

[22] C. A. R. Hoare, *Communicating sequential processes*. Prentice-Hall, Inc., 1985.

[23] J. Wexler, *Concurrent programming in OCCAM 2*. Ellis Horwood, 1989. [Online]. Available: http://books.google.com/books?id=xRCzAAAAIAAJ

[24] J. Thornley, "The Programming Language Declarative Ada Reference Manual," CA, USA, Tech. Rep., 1993.

[25] F. Gruian, P. Roop, Z. Salcic, and I. Radojevic, "The SystemJ approach to system-level design," in *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, july 2006, pp. 149–158.

[26] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, "SystemJ: A GALS language for system level design," *Comput. Lang. Syst. Struct.*, vol. 36, no. 4, pp. 317–344, December 2010. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2010.01.001

[27] C. Passerone, C. Sansoe, L. Lavagno, R. McGeer, J. Martin, R. Passerone, and A. Sangiovanni-Vincentelli, "Modeling reactive systems in Java," in *Proceedings of the Sixth International Workshop on Hardware/Software Codesign*, ser. CODES/CASHE '98. Washington, DC, USA: IEEE Computer Society, mar 1998, pp. 15–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=278241.278244

[28] J. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A. Newton, "Design and specification of embedded systems in Java using successive, formal refinement," in *Design Automation Conference, 1998. Proceedings*, 1998, pp. 70–75.

[29] M. Antonotti, A. Ferrari, A. Flesca, and A. L. Sangiovanni-Vincentelli, "JESTER: An Esterel-based reactive Java extension for reactive embedded system development." in *Proceedings of the International Forum on Specification and Design Languages (FDL)*. Tübingen: Proc., 2000, Sept. 2000.

[30] L. Hazard, J. Susini, and F. Boussinot, "The junior reactive kernel," Tech. Rep., 1999-07. [Online]. Available: http://hal.inria.fr/inria-00072933/en/

[31] B. Plummer, M. Khajanchi, and S. Edwards, "An Esterel Virtual Machine for Embedded Systems," in *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Vienna, Austria, March 2006.

[32] Z. Salcic, P. Roop, M. Biglari-Abhari, and A. Bigdeli, "REFLIX: a processor core with native support for control-dominated embedded applications," *Elsevier Journal of Microprocessors and Microsystems*, vol. 28, pp. 13–25, 2004.

[33] Z. Salcic, D. Hui, P. Roop, and M. Biglari-Abhari, "REMIC: design of a reactive embedded microprocessor core," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '05. New York, NY, USA: ACM, 2005, pp. 977–981. [Online]. Available: http://doi.acm.org/10.1145/1120725.1120771

[34] X. Li, R. von Hanxleden, "The kiel esterel processor - a semi-custom, configurable reactive processor," in *in: Synchronous Programming - SYNCHRON' 04*. Schloss Dagstuhl, Germany,: in Dagstuhl Seminar Proceedings,, 2005, pp. –. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2005/159

[35] X. Li and R. von Hanxleden, “Multithreaded Reactive Programming - the Kiel Esterel Processor,” *Computers, IEEE Transactions on*, vol. 61, no. 3, pp. 337 – 349, Mrach 2012.

[36] S. Yuan, S. Andalam, L. Yoong, P. Roop, and Z. Salcic, “Starpro — a new multithreaded direct execution platform for esterel,” *Electron. Notes Theor. Comput. Sci.*, vol. 238, no. 1, pp. 37–55, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2008.01.005

[37] Z. Salcic, P. Roop, D. Hui, and I. Radojevic, “HiDRA: A new architecture for heterogeneous embedded systems,” in *ESA/VLSI*, 2004, pp. 164–170.

[38] A. Malik, Z. Salcic, and P. Roop, “SystemJ compilation using the tandem virtual machine approach,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 3, pp. 34:1–34:37, June 2009. [Online]. Available: http://doi.acm.org/10.1145/1529255.1529256

[39] A. Malik, Z. Salcic, A. Girault, A. Walker, and S. C. Lee, “A customizable multiprocessor for Globally Asynchronous Locally Synchronous execution,” in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES ’09. New York, NY, USA: ACM, 2009, pp. 120–129. [Online]. Available: http://doi.acm.org/10.1145/1620405.1620423

[40] G. Berry and L. Cosserat, “The ESTEREL Synchronous Programming Language and its Mathematical Semantics,” in *Seminar on Concurrency, Carnegie-Mellon University*. London, UK: Springer-Verlag, 1984, pp. 389–448. [Online]. Available: http://dl.acm.org/citation.cfm?id=646723.702721

[41] G. Berry, “The foundations of Esterel,” in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000, pp. 425–454.

[42] G. Berry, “The constructive semantics of pure esterel.” pp. –, July 1999. [Online]. Available: http://www-sop.inria.fr/meije/

[43] D. Potop-Butucaru, S. Edwards, and G. Berry, *Compiling Esterel*. Springer, 2007. [Online]. Available: http://hal.inria.fr/inria-00072257/en/

[44] L. Yoong, P. Roop, and Z. Salcic, “Compiling Esterel for distributed execution,” in *Proceedings of Synchronous Languages, Applications and Programming*, 2006.

[45] W. Rosenstiel, “Embedded Java,” in *Proceedings of the 13th international symposium on System synthesis*, ser. ISSS ’00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 172–172. [Online]. Available: http://dx.doi.org/10.1145/501790.501826

[46] NIOS II, "http://www.altera.com/roducts/ip/rocessors/nios2."

[47] Altera Corp., "Nios II Processor Reference Handbook," 2008.

[48] Altera Corporation, *Avalon Bus Specification Reference Manual*, San Jose, California, July 2003.

[49] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[50] M. Schoeberl, *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. CreateSpace, 2009, no. ISBN 978-1438239699, available at http://www.jopdesign.com/doc/handbook.pdf.

[51] P. Molnar, A. Krall, and F. Brandner, "Stack allocation of objects in the cacao virtual machine," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 153–161. [Online]. Available: http://doi.acm.org/10.1145/1596655.1596680

[52] D. Simon and C. Cifuentes, "The squawk virtual machine: Java on the bare metal," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 150–151. [Online]. Available: http://doi.acm.org/10.1145/1094855.1094908

[53] T. Wilkinson, "Kaffe, a virtual machine to run java code." [Online]. Available: www.kaffe.org

[54] A. Azevedo, A. Nicolau, and J. Hummel, "Java annotation-aware just-in-time (ajit) complilation system," in *Proceedings of the ACM 1999 conference on Java Grande*, ser. JAVA '99. New York, NY, USA: ACM, 1999, pp. 142–151. [Online]. Available: http://doi.acm.org/10.1145/304065.304115

[55] FAJITA, "FAJITA Compiler Project," 1998. [Online]. Available: http://www.ri.silicomp.fr/adv-dvt/java/fajita/index-b.htm.

[56] Pendragon Software Corporation, "Caffeine Mark 3.0." [Online]. Available: http://www.pendragon-software.com/pendragon/cm3/info.html

[57] W. Lima, R. S. Lobato, l. Manacero, and R. Ulson, "Towards a Java bytecodes compiler for Nios II soft-core processor," in *4th IEEE Symposium on Computers and Communications (ISCC 2009)*. Sousse Tunisia: IEEE, 2009, pp. 104–109.

[58] M. O'Connor and M. Tremblay, "picoJava-I: The Java Virtual Machine in Hardware," *IEEE Micro*, vol. 17, pp. 45–53, March 1997. [Online]. Available: http://dl.acm.org/citation.cfm?id=623274.624084

[59] Sun. [a], "picoJava-II Microarchitecture Guide. Sun Microsystems," pp. –, March 1999.

[60] Sun. [b], *picoJava-II Programmer's Reference Manual*, Sun Microsystems, March 1999.

[61] Sun Microsystems, "Microjava-701 processor." [Online]. Available: http://www.sun.com/microelectronics/microJava-701

[62] PTSC, "IGNITE Processor Brochure Rev 1.0." [Online]. Available: http://www.ptsc.com

[63] FORTH Inc. [Online]. Available: http://www.forth.com/forth/

[64] D. Hardin, M. Frerking, P. Wiley, and G. Bolella, "Getting down and dirty: device-level programming using the Real-Time Specification for Java," in *Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, 2002, pp. 457–464.

[65] R. Zulauf, "Entwurf eines java-mikrocontrollers und prototypische implementierung auf einem fpga," Master's Thesis, University of Karlsruhe, Karlsruhe, Germany, 2000.

[66] S. Uhrig and J. 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*, ser. JTRES '07. New York, NY, USA: ACM, 2007, pp. 230–237,. [Online]. Available: http://doi.acm.org/10.1145/1288940.1288974

[67] A. Ito, L. Carro, and R. Jacobi, "Making Java Work for Microcontroller Applications," *IEEE Des. Test*, vol. 18, pp. 100–110, September 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=622205.623096

[68] A. Beck, S. Carlos, and L. Carro, "A VLIW low power Java processor for embedded applications," Pernambuco, Brazil, pp. 157–162, 2004.

[69] M. Tremblay, J. Chan, S. Chaudhry, A. Conigliaro, and S. Tse, "The MAJC Architecture: A Synthesis of Parallelism and Scalability," *Micro, IEEE*, vol. 20, no. 6, pp. 12–25, nov/dec 2000.

[70] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh, "VIPER: A 25-MHz, 100-MIPS peak VLIW microprocessor," in *Custom Integrated Circuits Conference, 1993., Proceedings of the IEEE 1993*, May 1993, pp. 4.1.1–4.1.5.

[71] A. Suga and K. Matsunami, “Introducing the FR500 embedded microprocessor,” *IEEE Micro*, vol. 20, no. 4, pp. 21–27, July 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=623294.624373

[72] N. Seshan, “High VelociTI processing [Texas Instruments VLIW DSP architecture],” *Signal Processing Magazine, IEEE*, vol. 15, no. 2, pp. 86–101, mar 1998.

[73] T. Halfhill, “Imsys hedges bets on Java,” Microdesign Resources, Microprocessor Report, April 2000.

[74] Vulcan ASIC Ltd., “Moon v1.0,” data sheet, Sep 2000.

[75] Vulcan ASIC Ltd., “Moon2 - 32 Bit Native Java Technology-Based Processor,” Product folder, 2003.

[76] Digital Communication Technologies Ltd., “Lightfoot 32-bit Java Processor Core,” data sheet, Aug 2001.

[77] VELOCITY SEMICONDUCTOR. Vs2000 typhoon family microcontroller. [Online]. Available: http://www.velocitysemi.com/processors.htm

[78] B. Bose, M. Tuna, and J. Nagy, “Lavacore configurable Java processor core,” in *Aerospace Conference Proceedings, 2002. IEEE*, vol. 4, 2002, pp. 4–1953 – 4–1959.

[79] Y. Tan, C. Yau, K. Lo, W. Yu, P. Mok, and A. Fong, “Design and implementation of a Java processor,” *Computers and Digital Techniques, IEE Proceedings*, vol. 153, no. 1, pp. 20–30, Jan. 2006.

[80] Azul Systems, “Azul compute appliances,” product datasheet, 2009.

[81] M. Schoeberl, “JOP: A Java Optimized Processor,” in *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, vol. 2889. Springer, 2003, pp. 346–359. [Online]. Available: http://www.jopdesign.com/doc/jtres03.pdf

[82] M. Zabel, T. Preuber, P. Reichel, and R. Spallek, “Secure, Real-Time and Multi-Threaded General-Purpose Embedded Java Microarchitecture,” in *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 59–62.

[83] M. Schoeberl, “A Time Predictable Instruction Cache for a Java Processor,” in *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, vol. 3292. Springer, 2004, pp. 371–382.

[84] M. Schoeberl, “SimpCon - a Simple and Efficient SoC Interconnect,” in *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, 2007.

[85] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber, “Worst-case execution time analysis for a Java processor,” *Software: Practice and Experience*, vol. 40/6, pp. 507–542, 2010. [Online]. Available: http://www.jopdesign.com/doc/wcetana.pdf

[86] M. Schoeberl, “SimpCon - a Simple and Efficient SoC Interconnect,” in *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, 2007.

[87] M. Schoeberl, “Design and Implementation of an Efficient Stack Machine,” in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05) - Workshop 3 - Volume 04*, ser. IPDPS ’05. Washington, DC, USA: IEEE Computer Society, 2005, p. 159.2. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2005.161

[88] W. Peterson, “WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores,” Specifications, 2001. [Online]. Available: http://www.opencores.org

[89] S. Edwards, “Estbench Esterel Benchmark Suite.” [Online]. Available: http://www1.cs.columbia.edu/sedwards/software.html

[90] L. Lavagno and E. Sentovich, “ECL: A Specification Environment for System-Level Design,” in *Design Automation Conference, 1999. Proceedings. 36th*, 1999, pp. 511 – 516.

[91] M. Nadeem, M. Biglari-Abhari, and Z. Salcic, “RJOP: a customized Java processor for reactive embedded systems,” in *Proceedings of the 48th Design Automation Conference*, ser. DAC ’11. New York, NY, USA: ACM, 2011, pp. 1038–1043. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024952

[92] P. Roop, Z. Salcic, and S. Dayaratne, “Towards direct execution of esterel programs on reactive processors,” Pisa, Italy, pp. 240–248, 2004.

[93] M. Schoeberl, “A Java processor architecture for embedded real-time systems,” *Journal of Systems Architecture*, vol. 54/1-2, no. 1-2, pp. 265–286, 2008. [Online]. Available: http://www.jopdesign.com/doc/rtarch.pdf

[94] T. Henties, J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek, “Java for safety-critical applications,” in *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, 2009.

[95] M. Nadeem, M. Biglari-Abhari, and Z. Salcic, “GALS-JOP - A Java Embedded Processor for GALS Reactive Programs,” in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*. Sydney, New South Wales Australia: IEEE, December 2011, pp. 292–299. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/DASC.2011.67

[96] G. Plumbridge and N. Audsley, “Extending Java for heterogeneous embedded system description,” in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, june 2011, pp. 1 –6.

[97] M. Nadeem, M. Biglari-Abhari, and Z. Salcic, “JOP-plus - A processor for efficient execution of Java programs extended with GALS concurrency,” in *ASP-DAC*, 2012, pp. 17–22.

[98] T. Ungerer, B. Robič, and J. Šilc, “A survey of processors with explicit multithreading,” *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, Mar. 2003. [Online]. Available: http://doi.acm.org/10.1145/641865.641867

[99] P. Hung and M. Flynn, “Optimum instruction-level parallelism (ILP) for superscalar and vliw processors,” Stanford, CA, USA, Tech. Rep., 1999.

[100] C. Pitter, M. Schoeberl, “Towards a Java Multiprocessor,” in *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. ACM Press, 2007, pp. 144–151. [Online]. Available: http://www.jopdesign.com/doc/jopcmp.pdf

[101] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The case for a single-chip multiprocessor,” *SIGPLAN Not.*, vol. 31, pp. 2–11, September 1996. [Online]. Available: http://doi.acm.org/10.1145/248209.237140

[102] J. L. Hennessy and D. A. Patterson, *Computer architecture (4th ed.): a quantitative approach*, 4th ed. Morgan Kaufmann Publishers Inc., 2006.

[103] K. Olukotun, L. Hammond, and J. Laudon, *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. M. D. Hill, Ed. Morgan & Claypool, 2007. [Online]. Available: [Online]. Available: www.morganclaypool.com

[104] S. Moore, “Multicore is bad news for supercomputers,” in *IEEE Spectrum*, November 2008.

[105] W. Wolf, *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann, San Francisco, CA, 2006.

[106] A. Aartieri, V. Dalto, R. Chesson, M. Hopkins, M. Rossi, and W. Peterson, “Nomadik - open multimedia platform for next generation mobile devices,” Tech. rep. TA305, 2004. [Online]. Available: http://www.st.com

[107] S. Dutta, R. Jensen, and A. Rieckmann, “Viper: A Multiprocessor SoC for Advanced Set-Top Box and Digital TV Systems,” *IEEE Des. Test*, vol. 18, pp. 21–31, September 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=622205.623088

[108] G. Martin and H. Chang, *Winning the SoC Revolution.* Amsterdam: Kluwer Academic Press, 2003, vol. chapter 5.

[109] S. Moch, M. Bereković, H. J. Stolberg, L. Friebe, M. B. Kulaczewski, A. Dehnhardt, and P. Pirsch, “HIBRID-SoC: a multi-core architecture for image and video applications,” in *Proceedings of the 2003 workshop on Memory performance: Dealing with Applications , systems and architecture*, ser. MEDEA ’03. New York, NY, USA: ACM, 2003, pp. 55–61. [Online]. Available: http://doi.acm.org/10.1145/1152923.1024303

[110] ARM, “ARM 11, MPcore Processor, Technical Reference Manual,” 2006. [Online]. Available: http://www.arm.com.

[111] D. Cormie, “The ARM11 microarchitecture,” *ARM Ltd. White Paper*, 2002.

[112] J. GAISLER and E. CATOVIC, “Multi-core processor based on leon3-ft ip core (leon3-ftmp),” in *Data Syst. Aerospace. 630, ESA Special Publication*, 2006.

[113] C. SPARC International, Inc., *The SPARC architecture manual: version 8.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc, 1992.

[114] ARM, *AMBA specification (rev. 2.0).*, ARM Std., 1999.

[115] Xilinx, “MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 9.2i,” 2007. [Online]. Available: http://www.xilinx.com

[116] IBM, “On-chip peripheral bus architecture specifications, v2.1.” 2001.

[117] XILINX, “Opb arbiter product specification (v1.10c).” 2005.

[118] ALTERA, “Quartus II Handbook, vol. 4: SOPC Builder (ver. 7.2).” 2007.

[119] S. Richardson, “MPOC: A chip multiprocessor for embedded systems.” Hewlett Packard, Technical report, 2002.

[120] B. Ackland, A. Anesko, D. Brinthaupt, S. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. Nicol, J. O’Neill, J. Othmer, E. Sackinger, K. Singh, J. Sweet, C. Terman, and J. Williams, “A single-chip, 1.6-billion, 16-b mac/s multiprocessor dsp,” *Solid-State Circuits, IEEE Journal of*, vol. 35, no. 3, pp. 412 –424, Mar 2000.

[121] Texas Instruments, "TMS320C6474 multicore digital signal processor," Texas Instruments, Technical report, 2009.

[122] C. Pitter and M. Schoeberl, "A real-time Java chip-multiprocessor," vol. 10, no. 1, pp. 1–34, 2010.

[123] C. Pitter, "Time-predictable memory arbitration for a Java chip-multiprocessor," in *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, ser. JTRES '08. New York, NY, USA: ACM, 2008, pp. 115–122. [Online]. Available: http://doi.acm.org/10.1145/1434790.1434808

[124] M. Herlihy and J. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/165123.165164

A

CVM Instruction Set

Instruction	Description	Register Transfers	Addressing Modes			
			Inherent	Immediate	Direct	Indirect
AND Rz Rx Operand	The contents of Rx and Rz / Operand are ANDed and the result is stored in Rz	Rz ← Rx AND Operand		X		
		Rz ← Rz AND Rx				X
OR Rz Rx Operand	The contents of Rx and Rz / Operand are ORed and the result is stored in Rz	Rz ← Rx OR Operand		X		
		Rz ← Rz OR Rx				X
ADD Rz Rx Operand	The contents of Rx and Rz / Operand are added and the result is stored in Rz	Rz ← Rx + Operand		X		
		Rz ← Rz + Rx				X
SUBV Rz Rx Operand	The contents of Rx and Rz / Operand are subtracted and the result is stored in Rz	Rz ← Rx - Operand		X		
SUB Rz Operand	The contents of Rz and the Operand are subtracted but the result is not stored	Rz - Operand		X		
INIT Operand	Load HP and TP with the content of immediate value	HP ← Operand TP ← Operand		X		
LDR Rz Rx	Load Rz with the content of immediate value / memory location pointed to by Rx or Operand	Rz ← Operand		X		
		Rz ← M[Rx]				X
		Rz ← M[Operand]			X	
STR Rz Rx	Store the content of Rx / immediate value, into memory location pointed to by Rz / direct address	M[Rz] ← Operand		X		
		M[Rz] ← Rx				X
		M[Operand] ← Rx			X	
JMP Rx	Jump to address location	PC ← Operand		X		

	unconditionally	PC ← Rx				X
PRESENT Rz Operand	Jump to address location if the thread pointed to by Rz is not present else continue execution	if Rz(0)=1 then PC←Operand else NEXT		X		
SENDATA Rx	Store the content of Rx in queue	M[TP]←Rx				X
CHKEND Rz Rx	Compare 4 bit memory blocks in Rx with Rz(3-0) and store the largest value back in Rz	Rz ←MAX{Rx[15:12], Rx[11:8],Rx[7:4], Rx[3:0], Rz[3:0] }				X
SWITCH Rz Rx	Switch execution to memory location pointed by addition of content of Rx with memory location pointed to by Rx plus 1	Rz ← M[Rx] Rz ← Rz + Rx + 1 PC ← M[Rz]				X
SZ Operand	Jump to address location if Z=1 else continue execution	if Z=1 then PC ← Operand else NEXT		X		
CLFZ	Clear Zero flag	Z ← 0	X			
CER	Clear EReady bit	ER ← 0	X			
CEOT	Clear EOT bit	EOT ← 0	X			
SEOT	Set EOT bit	EOT ← 1	X			
LER Rz	The content of ER is stored in Rz	Rz ← ER				X
SSVOP Rx	Load SVOP with the content of Rx	SVOP ← Rx				X
LSIP Rz	Load Rz with the content of SIP	Rz ← SIP				X
SSOP Rx	Load SOP with the content of Rx	SOP ← Rx				X
NOOP	No operation		X			

B

JVM Bytecodes arranged according to the number

Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode
nop	0x00 (0)	fload_0	0x22 (34)	fstore_1	0x44 (68)
aconst_null	0x01 (1)	fload_1	0x23 (35)	fstore_2	0x45 (69)
iconst_m1	0x02 (2)	fload_2	0x24 (36)	fstore_3	0x46 (70)
iconst_0	0x03 (3)	fload_3	0x25 (37)	dstore_0	0x47 (71)
iconst_1	0x04 (4)	dload_0	0x26 (38)	dstore_1	0x48 (72)
iconst_2	0x05 (5)	dload_1	0x27 (39)	dstore_2	0x49 (73)
iconst_3	0x06 (6)	dload_2	0x28 (40)	dstore_3	0x4A (74)
iconst_4	0x07 (7)	dload_3	0x29 (41)	astore_0	0x4B (75)
iconst_5	0x08 (8)	aload_0	0x2A (42)	astore_1	0x4C (76)
lconst_0	0x09 (9)	aload_1	0x2B (43)	astore_2	0x4D (77)
lconst_1	0x10 (10)	aload_2	0x2C (44)	astore_3	0x4E (78)
fconst_0	0x0B (11)	aload_3	0x2D (45)	iastore	0x4F (79)
fconst_1	0x0C (12)	iaload	0x2E (46)	lastore	0x50 (80)
fconst_2	0x0D (13)	laload	0x2F (47)	fastore	0x51 (81)
dconst_0	0x0E (14)	faload	0x30 (48)	dastore	0x52 (82)
dconst_1	0x0F (15)	daload	0x31 (49)	aastore	0x53 (83)
bipush	0x10 (16)	aaload	0x32 (50)	bastore	0x54 (84)
sipush	0x11 (17)	baload	0x33 (51)	castore	0x55 (85)
ldc	0x12 (18)	caload	0x34 (52)	sastore	0x56 (86)
ldc_w	0x13 (19)	saload	0x35 (53)	pop	0x57 (87)
ldc2_w	0x14 (20)	istore	0x36 (54)	pop2	0x58 (88)
iload	0x15 (21)	lstore	0x37 (55)	dup	0x59 (89)
lload	0x16 (22)	fstore	0x38 (56)	dup_x1	0x5A (90)
fload	0x17 (23)	dstore	0x39 (57)	dup_x2	0x5B (91)
dload	0x18 (24)	astore	0x3A (58)	dup2	0x5C (92)
aload	0x19 (25)	istore_0	0x3B (59)	dup2_x1	0x5D (93)
iload_0	0x1A (26)	istore_1	0x3C (60)	dup2_x2	0x5E (94)
iload_1	0x1B (27)	istore_2	0x3D (61)	swap	0x5F (95)
iload_2	0x1C (28)	istore_3	0x3E (62)	iadd	0x60 (96)
iload_3	0x1D (29)	lstore_0	0x3F (63)	ladd	0x61 (97)
lload_0	0x1E (30)	lstore_1	0x40 (64)	fadd	0x62 (98)
lload_1	0x1F (31)	lstore_2	0x41 (65)	dadd	0x63 (99)
lload_2	0x20 (32)	lstore_3	0x42 (66)	isub	0x64 (100)
lload_3	0x21 (33)	fstore_0	0x43 (67)	lsub	0x65 (101)

Mnemonic	Opcode	Mnemonic	Opcode	Mnemonic	Opcode
fsub	0x66 (102)	l2f	0x88 (136)	tableswitch	0xAA (170)
dsub	0x67 (103)	l2f	0x89 (137)	lookupswitch	0xAB (171)
imul	0x68 (104)	l2d	0x8A (138)	ireturn	0xAC (172)
lmul	0x69 (105)	f2i	0x8B (139)	lreturn	0xAD (173)
fmul	0x6A (106)	f2l	0x8C (140)	freturn	0xAE (174)
dmul	0x6B (107)	f2d	0x8D (141)	dreturn	0xAF (175)
idiv	0x6C (108)	d2i	0x8E (142)	areturn	0xB0 (176)
ldiv	0x6D (109)	d2l	0x8F (143)	return	0xB1 (177)
fdiv	0x6E (110)	d2f	0x90 (144)	getstatic	0xB2 (178)
ddiv	0x6F (111)	i2b	0x91 (145)	putstatic	0xB3 (179)
irem	0x70 (112)	i2c	0x92 (146)	getfield	0xB4 (180)
lrem	0x71 (113)	i2s	0x93 (147)	putfield	0xB5 (181)
frem	0x72 (114)	lcmp	0x94 (148)	invokevirtual	0xB6 (182)
drem	0x73 (115)	fcmpl	0x95 (149)	invokespecial	0xB7 (183)
ineg	0x74 (116)	fcmpg	0x96 (150)	invokestatic	0xB8 (184)
lneg	0x75 (117)	dcmpl	0x97 (151)	invokeinterface	0xB9 (185)
fneg	0x76 (118)	dcmpg	0x98 (152)	unused	0xBA (186)
dneg	0x77 (119)	ifeq	0x99 (153)	new	0xBB (187)
ishl	0x78 (120)	ifne	0x9A (154)	newarray	0xBC (188)
lshl	0x79 (121)	iflt	0x9B (155)	anewarray	0xBD (189)
ishr	0x7A (122)	ifge	0x9C (156)	arraylength	0xBE (190)
lshr	0x7B (123)	ifgt	0x9D (157)	athrow	0xBF (191)
iushr	0x7C (124)	ifle	0x9E (158)	instanceof	0xC1 (193)
lushr	0x7B (125)	if_icmpeq	0x9F (159)	monitorenter	0xC2 (194)
iand	0x7E (126)	if_acmpne	0xA0 (160)	monitorexit	0xC3 (195)
land	0x7F (127)	if_icmplt	0xA1 (161)	wide	0xC4 (196)
ior	0x80 (128)	if_icmpge	0xA2 (162)	multianewarray	0xC5 (197)
lor	0x81 (129)	if_icmpgt	0xA3 (163)	ifnull	0xC6 (198)
ixor	0x82 (130)	if_icmple	0xA4 (164)	ifnonnull	0xC7 (199)
lxor	0x83 (131)	if_acmpeq	0xA5 (165)	goto_w	0xC8 (200)
iinc	0x84 (132)	if_acmpne	0xA6 (166)	jsr_w	0xC9 (201)
i2l	0x85 (133)	goto	0xA7 (167)	breakpoint	0xCA (202)
i2f	0x86 (134)	jsr	0xA8 (168)	(unused opcodes)	203 - 253
i2d	0x87 (135)	ret	0xA9 (169)	impdep1	0xFE (254)
				impdep2	0xFF (255)

JVM Bytecodes arranged according to the Function

Data Operations

The Stack

Pushing constants onto the stack

bipush
Push a signed byte.
sipush
Push a signed word.
ldc
Push a single word constant.
ldc_w
Push a single word constant. (16-bit ref in constant pool)
ldc2_w
Push a double word constant.
aconst_null
Push the null object.
iconst_m1
Push integer -1.
iconst_n
Integers n = 0..5.
lconst_v
Longs v = 0..1.
fconst_v
Floats v = 0.0..2.0.
dconst_v
Doubles v = 0.0..1.0.

Stack Manipulation

nop
Do nothing.
pop
Pop the top word.
pop2
Pop the top two words.
dup
Duplicate the top word to place 2.
dup2
Duplicate the top two words.
dup_x1
Duplicate the top word to place 3.
dup2_x1
Duplicate the top two words to places 4 and 5.
dup_x2
Duplicate the top word to place 4.
dup2_x2
Duplicate the top two words to places 5 and 6.

swap
Swap the top two words.

Local Variables

Push local

aload

Load object from local variable n
aload_n
Load object from local variable n : n = 0..3
dload
Load double from local variable n
dload_n
Load double from local variable n : n = 0..3
fload
Load float from local variable n
fload_n
Load float from local variable n : n = 0..3
iload
Load integer from local variable n
iload_n
Load integer from local variable n : n = 0..3
lload
Load long from local variable n
lload_n
Load long from local variable n : n = 0..3
Pop stack into local var
astore
store object in local variable n
astore_n
store object in local variable n : n = 0..3
dstore
store double in local variable n
dstore_n
store double in local variable n : n = 0..3
fstore

store float in local variable n
fstore_n
store float in local variable n : n = 0..3
istore
store integer in local variable n
istore_n
store integer in local variable n : n = 0..3
lstore
store long in local variable n
lstore_n
store long in local variable n : n = 0..3

Arrays

Creating arrays

newarray
New array of primitive type
anewarray
New array of objects
multianewarray
New multidimensional array
Pushing array values
aaload
Push object from array.
baload
Push byte or boolean from array.
caload
Push char from array.
daload
Push double from array.
faload
Push float from array.
iaload
Push integer from array.
laload
Push long from array.
saload
Push short from array.
Storing values in arrays
aastore
Store object in array.
bastore
Store byte or boolean in array.
castore
Store char in array.
dastore
Store double in array.
fastore
Store float in array.
iastore
Store integer in array.
lastore
Store long in array.
sastore
Store short in array.
Objects
arraylength
Get length of array.
new
Allocate mem for object.
putfield
Store an instance variable.
getfield
Push an instance variable.
putstatic
Store a static object's variable.
getstatic
Push a static object's variable.
checkcast
Checks object type of stack top.
instanceof
Checks object's class.

Transformations

Arithmetic

iinc
Increment local var.
dadd
Add two doubles.
fadd
Add two floats.
iadd
Add two integers.
ladd
Add two longs.
dsub
Subtract two doubles.
fsub
Subtract two floats.
isub
Subtract two integers.
lsub
Subtract two longs.
dmul
Multiply two doubles.
fmul
Multiply two floats.

imul
Multiply two integers.
lmul
Multiply two longs.
ddiv
Divide two doubles.
fdiv
Divide two floats.
idiv
Divide two integers.
ldiv
Divide two longs.
drem
Take remainder of two doubles.
frem
Take remainder of two floats.
irem
Take remainder of two integers.
lrem
Take remainder of two longs.
dneg
Negate a double.
fneg
Negate a float.
ineg
Negate a integer.
lneg
Negate a long.

Bit Operations

ishl
shift integer left
ishr
shift integer right
iushr
shift integer right without regard to sign
lshl
shift long left
lshr
shift long righ
lushr
shift long right without regard to sign
iand
and two integers
land
and two longs
ior
or two integers
lor
or two longs
ixor
exclusive or two integers
lxor
exclusive or two longs

Type Conversions

i2l
integer to long
i2f
integer to float
i2d
integer to double
i2b
integer to byte
i2s
integer to short
i2c
integer to char
l2i
long to integer
l2f
long to float
l2d
long to double
f2i
float to integer
f2l
float to long
f2d
float to double
d2i
double to integer
d2l
double to long
d2f
double to float

Process Control Transfer

Conditional Branching

ifeq
branch if equal
ifne
branch if not equal
iflt
branch if less than
ifle

branch if less than or equal
ifgt
branch if greater than
ifge
branch if greater than or equal
ifnull
branch if null
ifnonnull
branch if not null
if_icmpeq
branch if two ints are equal
if_icmpne
branch if two ints are not equal
if_icmplt
branch if int2 less than int1
if_icmple
branch if int2 less than or equal to int1
if_icmpgt
branch if int2 greater than int1
if_icmpge
branch if int2 greater than or equal to int1
if_acmpeq
branch if references are equal
if_acmpne
branch if references are uneqal

Comparisons

lcmp
compare two longs
fcmpl
compare two floats (-1 on NaN)
fcmpg
compare two floats (1 on NaN)
dcmpl
compare two doubles (-1 on NaN)
dcmpg
compare two doubles (1 on NaN)
Unconditional Branches
goto
go to label
goto_w
go to label (wide address)
jsr
jump to subroutine
jsr_w
jump to subroutine (wide address)
ret
return from subroutine

Tables

lookupswitch
case statement equivalent
tableswitch
branch by range of values
Methods
invokeinterface
call interface method
invokespecial
call method in a specific class
invokestatic
call a static method
invokevirtual
call any other method
ireturn
return from method with integer
lreturn
return from method with long
freturn
return from method with float
dreturn
return from method with double
areturn
return from method with object
return
return from method with nothing

Miscellany

athrow
throw exception
breakpoint
used for debugging
wide
used for 2-word address or value
monitorenter
begin sychronization
monitorexit
end sychronization

C

Reactive JOP Bytecode to Microcode Mapping

jopsys_emit:	loadopd	T1 ← A, T2 ← T1, A ← B, B ← stack[sp], sp ← sp-1
	readsf	T1 ← SF[A] T1, A ← B, B ← stack[sp], sp ← sp-1
	loadormask	T1 ← ormask
	reor	T1 ← T1 ‖ T2
	writesf	SF[A] ← T1, A ← B, B ← stack[sp], sp ← sp-1
	nop nxt	jpc ← jpc+1
jopsys_demit:	loadopd	T1 ← A, T2 ← T1, A ← B, B ← stack[sp], sp ← sp-1
	readsf	T1 ← SF[A] T1, A ← B, B ← stack[sp], sp ← sp-1
	loadandmask	T1 ← andmask
	reand	T1 ← T1 && T2
	writesf	SF[A] ← T1, A ← B, B ← stack[sp], sp ← sp-1
	nop nxt	jpc ← jpc+1
jopsys_present:	loadopd	T1 ← A, T2 ← T1, A ← B, B ← stack[sp], sp ← sp-1
	readsf	T1 ← SF[A] T1, A ← B, B ← stack[sp], sp ← sp-1
	loadormask	T1 ← ormask
	reand	T1 ← T1 && T2
	pushonstack	A ← T1, B← A, stack[sp+1] ← B, sp ← sp+1
	nop nxt	jpc ← jpc+1
jopsys_lsip:	memrda	memrda ← A, A ← B, B ← stack[sp], sp ← sp-1
	wait	
	wait	

```
                ldmrd           A ←memrdd , B ← A, stack[sp+1]  ← B, sp ← sp+1
                popfromstack    T1 ← A, A ← B, B ← stack[sp], sp ← sp-1
                writesf         SF[A] ← T1, A ← B, B ← stack[sp], sp ← sp-1
                Nop nxt         jpc ← jpc+1
jopsys_lsop:    stmwa           memwra ← A, A ← B, B ← stack[sp], sp ← sp-1
                rdsf            T1 ← SF[A] T1, A ← B, B ← stack[sp], sp ← sp-1
                pushonstack     A ← T1, B← A, stack[sp+1]  ← B, sp ← sp+1
                stmwd           memwrd ← A, A ← B, B ← stack[sp], sp ← sp-1
                wait
                wait
                nop nxt         jpc ← jpc+1
jopsys_cer:     stmwa           stmwra ← A, A ← B, B ← stack[sp], sp ← sp-1
                stmwd           stmwrd ← A, A ← B, B ← stack[sp], sp ← sp-1
                wait
                wait
                nop nxt         jpc ← jpc+1
jopsys_seot:    stmwa           memwra ← A, A ← B, B ← stack[sp], sp ← sp-1
                stmwd           memwrd ← A, A ← B, B ← stack[sp], sp ← sp-1
                wait
                wait
                nop nxt         jpc ← jpc+1
jopsys_ceot:    stmwa           memwra ← A, A ← B, B ← stack[sp], sp ← sp-1
                stmwd           memwrd ← A, A ← B, B ← stack[sp], sp ← sp-1
                wait
                wait
                nop nxt         jpc ← jpc+1
```

D

SystemJ Benchmark examples

```
//demoloop.Sysj
system{
      interface{
            output signal B,C,D,E;

      }
      {
            { // start clock-domain

                  signal A,R; // signals local to clock-domain
                  {
                        abort(R){
                              {
                                    while(true){
                                          await(A);
                                          emit B;
                                          present(C){
                                                emit D;
                                          }
                                          pause;
                                    }
                              }
                              ||
                              {
                                    while(true){
                                          present(B){
                                                emit C;
                                          }
                                          pause;
                                          present(B){
                                                emit E;
                                          }
                                    }
                              }
                        }
                  }
                  ||
                  {
                        //This is the test vector input
                        pause;
                        emit A;
                        System.out.println("A");
                        pause;
                        pause;
```

```
                pause;
                pause;
                emit R;
                System.out.println("R");
            }
        }
        ><
        {}
    }
}
```

```
/*
runner.sysj
*/
reaction CHECK_HEART(:signal SECOND, signal HEART_BEAT,signal
HEART_ATTACK){
      await(HEART_BEAT);
      await(SECOND);
      emit HEART_ATTACK;
      System.out.println();
}

system{
      interface{
            output signal WALK,RUN,JUMP,GO_TO_HOSPITAL;
      }
      {
            {
                  signal HEART_ATTACK,METER,SECOND,HEART_BEAT,STEP;
                  int meters=0,seconds=0,u=0;
                  {
                    trap(heart_attack){
                     while(true){
                        trap(t){
                          while(true){
                           pause;
                           present(METER){
                              if(meters == 100){
                                exit(t);
                           }
                            emit WALK;
                            System.out.println("WALK");
                            meters=meters+1;
                        }
                   }
            }
            {
              abort(HEART_ATTACK){
                  trap(t1){
                    while(true){
                        pause;
                        if(seconds == 15){
                        exit(t1);
                       }
                      await(STEP);
                      emit JUMP;
                      System.out.println("JUMP");
                      present(SECOND){
                        ++seconds;
                      }
                   }
                }
                 while(true){
                   emit RUN;
                   System.out.println("RUN");
                   pause;
                 }
              }
            exit(heart_attack);
        }
       ||
        CHECK_HEART(SECOND,HEART_BEAT,HEART_ATTACK)
       }
      }
       emit GO_TO_HOSPITAL;
       System.out.println("GO_TO_HOSPITAL");
      }
      ||
      {
    //This is the input vector sequence for this design to work
       trap(t3){
          while(true){
             pause;
             if(u == 150){
                exit(t3);
             }
             if(u < 101)
                emit METER;
                System.out.println("METER");
             if(u < 133)
                emit STEP;
                System.out.println("STEP");
                if(u < 140)
                   emit SECOND;
                   System.out.println("SECOND");
                   u=u+1;
                   System.out.println(u);
                if(u == 139){
                  emit HEART_BEAT;
                 System.out.println("HEART_BEAT");
                }
            }
         }
        }
    }
    ><
    {}
  }
}
```

```
/*
 Combination Lock
 cl.sysj
*/
reaction ONE_BUTTON(:signal LOCKED,signal UNLOCKED, signal
PRESELECTED,input signal BUTTON, input signal LOCK, input signal
UNLOCK, output signal BUTTON_PRESELECTED_ON, output signal
BUTTON_PRESELECTED_OFF, output signal BUTTON_LOCKED_ON, output
signal BUTTON_LOCKED_OFF)
{

      emit BUTTON_PRESELECTED_OFF;
      System.out.println("BUTTON_PRESELECTED_OFF");
      emit BUTTON_LOCKED_OFF;
      System.out.println("BUTTON_LOCKED_OFF");
      while(true){

            trap(BACK_TO_MAIN_LOOP){

            trap(PRESELECTED){
                  while(true){
                        abort(LOCKED){
                              while(true){
                                    await(BUTTON); exit(PRESELECTED);
                              }
                        }
                        await(UNLOCKED);
                  }
            }
            while(true){
                  emit PRESELECTED;
                  System.out.println("PRESELECTED");
                  emit BUTTON_PRESELECTED_ON;
                  System.out.println("BUTTON_PRESELECTED_ON");

                  abort(LOCK){
                        while(true){
                              //await(BUTTON||PRESELECTED);
                              abort(PRESELECTED){
                                    abort(BUTTON){
                                          while(true){
                                                pause;
                                          }
                                    }
                              }
                              emit BUTTON_PRESELECTED_OFF;

                  System.out.println("BUTTON_PRESELECTED_OFF");
                              exit(BACK_TO_MAIN_LOOP);
                        }
                  }

                  emit BUTTON_PRESELECTED_OFF;
                  System.out.println("BUTTON_PRESELECTED_OFF");
                  emit LOCKED;
                  System.out.println("LOCKED");
```

```
                emit BUTTON_LOCKED_ON;
                System.out.println("BUTTON_LOCKED_ON");
                await(UNLOCK);
                emit BUTTON_LOCKED_OFF;
                System.out.println("BUTTON_LOCKED_OFF");
                emit UNLOCKED;
                System.out.println("UNLOCKED");

            }
        }
    }
}

reaction abcd(: input signal A, input signal B, input signal C,
input signal D, input signal LOCK, output signal A_PRESELECTED_ON,
output signal B_PRESELECTED_ON, output signal C_PRESELECTED_ON,
output signal D_PRESELECTED_ON,output signal A_PRESELECTED_OFF,
output signal B_PRESELECTED_OFF,output signal C_PRESELECTED_OFF,
output signal D_PRESELECTED_OFF, output signal A_LOCKED_ON,output
signal B_LOCKED_ON,output signal C_LOCKED_ON,
output signal D_LOCKED_ON,output signal A_LOCKED_OFF,output signal
B_LOCKED_OFF,output signal C_LOCKED_OFF,output signal D_LOCKED_OFF)
{
  signal LOCKED,UNLOCKED,PRESELECTED;
ONE_BUTTON(LOCKED,UNLOCKED,PRESELECTED,A,LOCK,LOCK,A_PRESELECTED_ON,
A_PRESELECTED_OFF,A_LOCKED_ON,A_LOCKED_OFF)
||
ONE_BUTTON(LOCKED,UNLOCKED,PRESELECTED,B,LOCK,LOCK,B_PRESELECTED_ON,
B_PRESELECTED_OFF,B_LOCKED_ON,B_LOCKED_OFF)
||
ONE_BUTTON(LOCKED,UNLOCKED,PRESELECTED,C,LOCK,LOCK,C_PRESELECTED_ON,
C_PRESELECTED_OFF,C_LOCKED_ON,C_LOCKED_OFF)
||
ONE_BUTTON(LOCKED,UNLOCKED,PRESELECTED,D,LOCK,LOCK,D_PRESELECTED_ON,
D_PRESELECTED_OFF,D_LOCKED_ON,D_LOCKED_OFF)

}

system{

     interface{
     input signal A,B,C,D,LOCK;
     output signal A_PRESELECTED_ON, B_PRESELECTED_ON,\
                  C_PRESELECTED_ON, D_PRESELECTED_ON;
     output signal A_PRESELECTED_OFF, B_PRESELECTED_OFF,
                  C_PRESELECTED_OFF,D_PRESELECTED_OFF;
     output signal A_LOCKED_ON,B_LOCKED_ON,C_LOCKED_ON,D_LOCKED_ON;
     output signal A_LOCKED_OFF, B_LOCKED_OFF, C_LOCKED_OFF,
                  D_LOCKED_OFF;
     }
     {
                abcd(A,B,C,D,LOCK,A_PRESELECTED_ON,B_PRESELECTED_ON,
                     C_PRESELECTED_ON,D_PRESELECTED_ON,A_PRESELECTE
                     D_OFF,B_PRESELECTED_OFF,C_PRESELECTED_OFF,D_PR
                     ESELECTED_OFF, A_LOCKED_ON, B_LOCKED_ON,
                                        C_LOCKED_ON, D_LOCKED_ON, A_LOCKED_OFF,
                                        B_LOCKED_OFF,C_LOCKED_OFF,D_LOCKED_OFF)

                    ><
                    {
                    }
          }

}
```

```
/*
 * This is the asynch proto stack
  **/
import Asproto.*;

system{
     interface{
          input int channel reset;
          output int channel reset;
          output byte channel data;
          input byte channel data;
     }
     {

          //TestBench(reset,data)
          {
               send reset(1);
               pause;
               byte tosend[] = new byte[6];
               tosend[0] = 13; tosend[1] = 73; tosend[2]=127;
               tosend[3]=100; tosend[4]=55; tosend[5]=77;

               int y=0;
               trap(y1){
                    while(true){
                         int len = 0; len = tosend.length;
                         if(y == len){
                              exit(y1);
                              pause;
                         }
                         else{
                              if(y < 6){
                                   byte r = 0;
                                   r = tosend[y];
                                   System.out.println("bytesent
"+r);

                                   send data(r);
                                   y =y+1;
                              }
                              pause;
                              pause;
                         }
                    pause;
                    }
                    pause;
               }
               // more of this!
               pause;
               System.err.println("Packet sent.");
               pause;
               //System.exit(1);
          }
          ><
          //TheStack(reset,data)
          {
               signal packet,kill_check;
signal res1,res2,res3;
Integer signal crc_ok;
Asproto buffer = null;
{
      while(true){
            receive reset;
            int u= 0;
            u = #reset;
            if(u==1){
                  emit res1;
                  emit res2;
                  emit res3;
            }
            pause;
      }
}
||
//Assemble(reset,in_byte,packet)
{
      int cnt=0;
     //changing the shared var
      buffer = new Asproto();
      while(true){
            abort(res1){

                  int e =0;
                  trap(e){
                        int len2 = Asproto.PKTSIZE;
                        if(e == len2){
                              exit(e);
                        }
                        else{
                              receive data;
                              byte t = 0; t = #data;
                              e=e+1;
                              buffer.setRaw(e,t);
                        }
                  }
                  emit packet;
        //Thus now bufer has become available
            }
            pause;
      }
}
||
{
      int crc=0;
      while(true) {
            abort(res2){

                  await(packet);
                  crc = buffer.computeCRC();
                  int val = 0;
                  val = (crc==buffer.getCRC()) ? 1:0;
                  emit crc_ok(val);
            }//abort
```

```
                    pause;
               } // while*/
          }
          //Checkcrc(reset,packet,crc_ok)
          ||
          {

               int match_ok=0;

               while(true){
                    abort(res3){

                         await(packet);//now buffer is avail

                         {
                              abort(kill_check){
                         //Some length computation
                                   match_ok = 1;
                                   pause;
                                   pause;
                                   pause;
                                   pause;
                              }
                         }
                         ||
                         {
                              await(crc_ok);
                              int re = 0;
                              re= #crc_ok;
                              if(re==0){
                                   emit kill_check;
                              }
                         }
                         int there = 0;
                         there = #crc_ok;
                         if(there==1 && match_ok==1){
                           System.out.println("Address match!");
                         }

                    }
                    pause;
               }
          }
     }
  }
}
```

E

GALS-JOP Bytecodes

Bytecode	Microcode	Register Transfer Description
jopsys_strimm	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	dmaddr	CAB+T1 → T1
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmwa	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	Stmwd	A → stmwrd, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	nop nxt	jpc → jpc+1
jopsys_strind	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	dmaddr	CAB+T1 → T1=RZ
	rdrf	RF[X] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmwa	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	Stmwd	A → stmwrd, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	nop nxt	jpc → jpc+1
jopsys_strdir	popfromstack	A → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	dmaddr	CAB+T1 → T1
	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmwa	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	Stmwd	A → stmwrd, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	nop nxt	jpc → jpc+1
jopsys_ldrimm	wrrf	A → RF[B], B → A, stack[SP] →B, SP-1 → SP
	pop nxt	B → A, stack[SP] →B, SP-1 → SP, jpc → jpc+1

Bytecode	Microcode	Register Transfer Description
jopsys_ldrind	rdrf	RF[X] → T1, T 1→ T2, B →A, stack[SP] →B, SP-1 → SP
	dmaddr	CAB+T1→ T1
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmra	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	ldmrd	memrdd → A, A →B, B →stack[SP+1], SP+1 → SP
	wrrf	A → RF[B], B → A, stack[SP] →B, SP-1 → SP
	pop nxt	B → A, stack[SP] →B, SP-1 → SP, jpc → jpc+1
jopsys_ldrdir	popfromstack	A → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	dmaddr	CAB+T1 → T1
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmra	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	ldmrd	memrdd → A, A →B, B →stack[SP+1], SP+1 → SP
	wrrf	A → RF[B], B → A, stack[SP] →B, SP-1 → SP
	pop nxt	B → A, stack[SP] →B, SP-1 → SP, jpc → jpc+1
jopsys_aluimm	popfromstack	A → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	aluop	A aluop B → A, stack[SP] →B, SP-1 → SP
	wrrf	A → RF[B], B → A, stack[SP] →B, SP-1 → SP
	pop nxt	B → A, stack[SP] →B, SP-1 → SP, jpc → jpc+1
jopsys_aluind	rdrf	RF[x] → T1, T → T2, B →A, stack[SP] →B, SP-1 → SP
	dup	A →B, B →stack[SP+1], SP+1 → SP
	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	and	A ^ B → A, stack[SP] →B, SP-1 → SP
	wrrf	A → RF[B], B → A, stack[SP] →B, SP-1 → SP
	pop nxt	B → A, stack[SP] →B, SP-1 → SP, jpc → jpc+1
jopsys_subv	popfromstack	A → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	aluop	A aluop B → A, stack[SP] →B, SP-1 → SP
	stsf	1→ZFreg ifA=0
	nop nxt	jpc → jpc+1

Bytecode	Microcode	Register Transfer Description
jopsys_jumpimm	popfromstack	A → T1, T → T2, B →A, stack[SP] →B, SP-1 → SP
	pmaddr	TT[label] → T1
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	jbr	
	pop	B → A, stack[SP] →B, SP-1 → SP
	nop nxt	jpc → jpc+1
jopsys_jumpdir	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pm_addr	TT[label] → T1
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	jbr	
	pop	B → A, stack[SP] →B, SP-1 → SP
	nop nxt	jpc → jpc+1
jopsys_chkend	rdrf	RF[X] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	dup	A →B, B →stack[SP+1], SP+1 → SP
	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	findmax	Max_out → T1
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	wrrf	A → RF[B], B → A, stack[SP] →B, SP-1 → SP
	pop nxt	B → A, stack[SP] →B, SP-1 → SP, jpc → jpc+1
jopsys_label	nop nxt	jpc → jpc+1
jopsys_nop	nop nxt	jpc → jpc+1
jopsys_clfz	clfz nxt	0 → Zfreg, jpc → jpc+1
jopsys_strio	Stmra	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	rdrf	RF[X] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmwd	A → stmwrd, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	nop nxt	jpc → jpc+1
jopsys_ldio	Stmra	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	ldmrd	memrdd → A, A →B, B →stack[SP+1], SP+1 → SP
	wrrf	A → RF[B], B → A, stack[SP] →B, SP-1 → SP
	pop nxt	B → A, stack[SP] →B, SP-1 → SP, jpc → jpc+1
jopsys_sz	popfromstack	A → T=Operand, T → T2, B →A, stack[SP] →B, SP-1 → SP
	pm_addr	TT[label] → T
	pushonstack	T → A, A →B, B →stack[SP+1], SP+1 → SP
	jbr	T → jpc if ZFreg=1
	pop	B → A, stack[SP] →B, SP-1 → SP
	nop	

Bytecode	Microcode	Register Transfer Description
jopsys_present	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	popfromstack	A → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pm_addr	TT[label] → T1
	pushonstack	T → A, A →B, B →stack[SP+1], SP+1 → SP
	pushonstack	T → A, A →B, B →stack[SP+1], SP+1 → SP
	jbr	T → jpc if ZFreg=0
	pop	
	pop	
	nop nxt	jpc → jpc+1
jopsys_switchjump	rdrf	RF[X] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	dmaddr	CAB+T1 → T1
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmra	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	rdrf	RF[Z] → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	wait	
	ldmrd	memrdd → A, A →B, B →stack[SP+1], SP+1 → SP
	add	A + B → A, stack[SP] →B, SP-1 → SP (Rx+Rz)
	add	A + B → A, stack[SP] →B, SP-1 → SP (Rx+Rz+1)
	dup	A →B, B →stack[SP+1], SP+1 → SP
	popfromstack	A → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	wrrf	A → RF[B], B → A, stack[SP] →B, SP-1 → SP
	dmaddr	CAB+T → T1
	pushonstack	T1 → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmra	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	ldmrd	memrdd → A, A →B, B →stack[SP+1], SP+1 → SP
	popfromstack	A → T1, T1 → T2, B →A, stack[SP] →B, SP-1 → SP
	pmaddr	TT[label] → T
	pop	B → A, stack[SP] →B, SP-1 → SP
	jbr	
	nop	
	nop nxt	jpc → jpc+1

Bytecode	Microcode	Register Transfer Description
jopsys_cabaseaddress	ld1	stack[vp+n] → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmra	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	ldmrd	memrdd → A, A →B, B →stack[SP+1], SP+1 → SP
	Stmra	A → stmwra, B→A, stack[sp] → B, SP-1 → SP
	wait	
	wait	
	ldmrd	memrdd → A, A →B, B →stack[SP+1], SP+1 → SP
	stdmbaseaddress	A →dm_CAB, B →A, stack[SP] →B, SP-1 → SP
	nop nxt	jpc → jpc+1

F

JOP-Plus : CRCF to Microcode Mapping

AND_IMMEDIATE

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc]
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldcrcfopdt	T ← ctrlopdreg
ldt	A ← T , stack[sp+1] ← B, sp ← sp+1
and	A ← A && B, B ← stack[sp] , sp ← sp-1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1

AND_INDIRECT

Microcode	Register-Transfer
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldrzt	T ← RF[Rz]
ldt	A ← T , stack[sp+1] ← B, sp ← sp+1
and	A ← A && B, B ← stack[sp] , sp ← sp-1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1

OR_IMMEDIATE

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc]
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldcrcfopdt	T ← ctrlopdreg
ldt	A ← T , stack[sp+1] ← B, sp ← sp+1
or	A ← A || B, B ← stack[sp] , sp ← sp-1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1

OR_INDIRECT

Microcode	Register-Transfer

ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldrzt	T ← RF[Rz]
ldt	A ← T , stack[sp+1] ← B, sp ← sp+1
or	A ← A || B, B ← stack[sp] , sp ← sp-1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, , jpc ← jpc+1

ADD_IMMEDIATE

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc] , jpc ← jpc+1
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldcrcfopdt	T ← ctrlopdreg
ldt	A ← T , stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

ADD_INDIRECT

Microcode	Register-Transfer
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldrzt	T ← RF[Rz]
ldt	A ← T , stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, , jpc ← jpc+1

SUBV_IMMEDIATE

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc] , jpc ← jpc+1
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldcrcfopdt	T ← ctrlopdreg
ldt	A ← T , stack[sp+1] ← B, sp ← sp+1
sub	A ← A – B, B ← stack[sp] , sp ← sp-1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, , jpc ← jpc+1

SUB_IMMEDIATE

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc] , jpc ← jpc+1
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1

ldcrcfopdt	T ← ctrlopdreg
ldt	A ← T , stack[sp+1] ← B, sp ← sp+1
sub	A ← A – B, B ← stack[sp] , sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1

LDR_IMMEDIATE

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc] , jpc ← jpc+1
Wrimm nxt	RF[Rz] ← Crcfopdreg, , jpc ← jpc+1

LDR_INDIRECT

Microcode	Register-Transfer
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
lddmt	T ← crcfdm[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop	
pop	A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

LDR_DIRECT

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc], jpc ← jpc+1
ldcrcfopdt	T ← ctrlopdreg
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
lddmt	T ← crcfdm[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop	
pop	A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

STR_IMMEDIATE

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc], jpc ← jpc+1
ldcrcfopdt	T ← ctrlopdreg
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldrzt	T ← RF[Rz]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
wrdm	crcfdm[B] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop	
pop	A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

STR_INDIRECT

Microcode	**Register-Transfer**
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldrzt	T ← RF[Rz]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
wrdm	crcfdm[B] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop	
pop	A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

STR_DIRECT

Microcode	**Register-Transfer**
nop opd	Crcfopdreg ← crcfpm[jpc], jpc ← jpc+1
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldcrcfopdt	T ← ctrlopdreg
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
wrdm	crcfdm[B] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop	
pop	A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

JMP_IMMEDIATE

Microcode	**Register-Transfer**
nop opd	Crcfopdreg ← crcfpm[jpc], jpc ← jpc+1
ldcrcfopdt	T ← ctrlopdreg
nop	
jbr	jpc ← T
nop	
nop nxt	jpc ← jpc+1

JMP_INDIRECT

Microcode	**Register-Transfer**
ldrxt	T ← RF[Rx]
nop	
jbr	jpc ← T
nop	
nop nxt	jpc ← jpc+1

PRESENT_IMMEDIATE

Microcode	**Register-Transfer**
nop opd	Crcfopdreg ← crcfpm[jpc], jpc ← jpc+1
ldrzt	T ← RF[Rz]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldcrcfopdt	T ← ctrlopdreg
nop	

jbr	jpc ← T if A=0
nop	
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

SZ_IMMEDIATE

Microcode	Register-Transfer
nop opd	Crcfopdreg ← crcfpm[jpc], jpc ← jpc+1
ldcrcfopdt	T ← ctrlopdreg
nop	
jbr	jpc ← T if ZF=1
nop	
nop nxt	jpc ← jpc+1

CHKEND_INDIRECT

Microcode	Register-Transfer
ldmax	T ← MAX
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop	
pop	A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

SWITCH_INDIRECT

Microcode	Register-Transfer
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
lddmt	T ← crcfdm[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
ldi 1	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
lddmt	T ← crcfdm[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop	
jbr	jpc ← T
pop	A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

CLFZ_INHERENT

Microcode	Register-Transfer
clfz	ZFREG ← 0
nop nxt	jpc ← jpc+1

SENDATA_INDIRECT

Microcode	Register-Transfer
ldjpc	A ← jpc, B ← A, stack[sp+1] ← B, sp ← sp+1
stcrcfpc	crcfreg ← A, B ← stack[sp]
ldjcfbasereg	T ← jcfbasereg
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldrxt	T ← RF[Rx]
ldt	A ← T, B ← A, stack[sp+1] ← B, sp ← sp+1
ldi 8	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
ushr	A ← B >>> A, B ← stack[sp] , sp ← sp-1
dup	A ← A, B ← A,stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
nop	
switchmode	ctrl_mode flag ← ctrl_mode flag
nop	
ldvp	A ← vp, B ← A, stack[sp+1] ← B, sp ← sp+1
stm old_vp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
dup	A ← A, B ← A, stack[sp+1] ← B, sp ← sp+1
ldi 1	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
stmrac	memrda ← A, A ← B, B ← stack[sp], sp ← sp-1
ldm mp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
stm old_mp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
stm mp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
wait	
wait	
ldmrd	A ←memrdd , B ← A, stack[sp+1] ← B, sp ← sp+1
ldjpc	A ←jpc , B ← A, stack[sp+1] ← B, sp ← sp+1
ldbcstart	A ←bcstart , B ← A, stack[sp+1] ← B, sp ← sp+1
sub	A ← A – B, B ← stack[sp] , sp ← sp-1
stm old_jpc	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
ldm mp	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
stmrac	memrda ← A, A ← B, B ← stack[sp], sp ← sp-1
ldm cp	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
stm old_cp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
wait	
wait	
ldmrd	A ←memrdd , B ← A, stack[sp+1] ← B, sp ← sp+1
stbcrd	membcr ← A, A ← B, B ← stack[sp] , sp ← sp-1
dup	A ← A, B ← A,stack[sp+1] ← B, sp ← sp+1
ldi 31	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
and	A ← A && B, B ← stack[sp] , sp ← sp-1

pop	A ← B, B ← stack[sp], sp ← sp-1
ldi 2	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
stm args	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
ldi 5	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
ushr	A ← B >>> A, B ← stack[sp] , sp ← sp-1
dup	A ← A, B ← A,stack[sp+1] ← B, sp ← sp+1
ldi 31	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
and	A ← A && B, B ← stack[sp] , sp ← sp-1
stm varcnt	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
ldi 5	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
ushr	A ← B >>> A, B ← stack[sp] , sp ← sp-1
stm cp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
ldsp	A ←sp , B ← A, stack[sp+1] ← B, sp ← sp+1
ldi 1	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
dup	A ← A, B ← A,stack[sp+1] ← B, sp ← sp+1
ldm args	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
sub	A ← A – B, B ← stack[sp] , sp ← sp-1
stm old_sp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
ldm old_sp	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
ldi 1	A ← stack[n+32], B ← A, stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
stvp	vp ← A, A ← B, B ← stack[sp]
ldm varcnt	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
add	A ← A + B, B ← stack[sp] , sp ← sp-1
nop	
stsp	sp ← A, A ← B, B ← stack[sp]
pop	A ← B, B ← stack[sp], sp ← sp-1
pop	A ← B, B ← stack[sp], sp ← sp-1
ldm old_sp	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
ldm old_jpc	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
ldbcstart	
stjpc	jpc ← A, A ← B, B ← stack[sp]
ldm old_vp	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
ldm old_cp	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
ldm old_mp	A ←stack[n] , B ← A, stack[sp+1] ← B, sp ← sp+1
wait	
wait	
nop nxt	jpc ← jpc+1

SEOT_ INHERENT

Microcode	Register-Transfer
ldi io_eot	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
stmwa	memwra ← A, A ← B, B ← stack[sp], sp ← sp-1
ldi 1	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
stmwd	memwrd ← A, A ← B, B ← stack[sp], sp ← sp-1
wait	A ← B, B ← stack[sp], sp ← sp-1
wait	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

nop nxt	jpc ← jpc+1

CEOT_ INHERENT

Microcode	Register-Transfer
ldi io_eot	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
stmwa	memwra ← A, A ← B, B ← stack[sp], sp ← sp-1
ldi 0	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
stmwd	memwrd ← A, A ← B, B ← stack[sp], sp ← sp-1
wait	
wait	
nop nxt	jpc ← jpc+1

LER_INDIRECT

Microcode	Register-Transfer
ldi io_er	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
stmra	memrda ← A, A ← B, B ← stack[sp], sp ← sp-1
wait	
wait	
ldmrd	A ←memrdd , B ← A, stack[sp+1] ← B, sp ← sp+1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop nxt	jpc ← jpc+1

LSIP_INDIRECT

Microcode	Register-Transfer
ldi io_er	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
stmra	memrda ← A, A ← B, B ← stack[sp], sp ← sp-1
wait	
wait	
ldmrd	A ←memrdd , B ← A, stack[sp+1] ← B, sp ← sp+1
wrrf	RF[Rz] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop nxt	jpc ← jpc+1

SSOP_ INDIRECT

Microcode	Register-Transfer
ldi io_sop	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
stmwa	memwra ← A, A ← B, B ← stack[sp], sp ← sp-1
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
stmwd	memwrd ← A, A ← B, B ← stack[sp], sp ← sp-1
wait	
wait	
nop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

SSVOP_ INDIRECT

Microcode	Register-Transfer
ldi io_svop	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
stmwa	memwra ← A, A ← B, B ← stack[sp], sp ← sp-1

ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
stmwd	memwrd ← A, A ← B, B ← stack[sp], sp ← sp-1
wait	
wait	
nop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

NOOP_INHERENT

Microcode	Register-Transfer
nop	
nop nxt	jpc ← jpc+1

ESL_INHERENT

Microcode	Register-Transfer
nop	
nop nxt	jpc ← jpc+1

CINIT_INHERENT

Microcode	Register-Transfer
nop	
nop nxt	jpc ← jpc+1

JCF Bytecode Extension for JOP-Plus

jopsys_rtc

Microcode	Register-Transfer
ldrxt	T ← RF[Rx]
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
ldi 255	A ← stack[n+32] , stack[sp+1] ← B, sp ← sp+1
and	A ← A && B, B ← stack[sp] , sp ← sp-1
wrdm	dm[A] ← B
nop	
pop	A ← B, B ← stack[sp], sp ← sp-1
pop	A ← B, B ← stack[sp], sp ← sp-1
stm mp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
stm cp	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
stvp	vp ← A, A ← B, B ← stack[sp]
stm old_jpc	Stack[n] ← A, A ← B, B ← stack[sp] , sp ← sp-1
nop	
stsp	sp ← A, A ← B, B ← stack[sp]
ldcrcfpct	T ← crcfpcreg
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
stjpc	jpc ← A, A ← B, B ← stack[sp]
nop	
switchmode	ctrl_mode flag ← ctrl_mode flag
nop	
nop	
nop nxt	jpc ← jpc+1

jopsys_stcrcf

Microcode	Register-Transfer
stjcfbasereg	jcfbasereg← A, A ← B, B ← stack[sp], sp ← sp-1
nop nxt	jpc ← jpc+1

jopsys_wrcrcf

Microcode	Register-Transfer
wrctrl	crcfpm[B] ← A, A ← B, B ← stack[sp], sp ← sp-1
nop	
pop	A ← B, B ← stack[sp], sp ← sp-1
pop nxt	A ← B, B ← stack[sp], sp ← sp-1, jpc ← jpc+1

jopsys_switchmode

Microcode	Register-Transfer
ldjpc	A ← jpc, B ← A, stack[sp+1] ← B, sp ← sp+1
stcrcfpc	crcfreg ← A, B ← stack[sp], sp ← sp-1
ldctrlpct	T ← crcfpcreg
ldt	A ← T, stack[sp+1] ← B, sp ← sp+1
stjpc	jpc ← A, B ← stack[sp], sp ← sp-1
nop	
toggle	mode_ctrl ← ~ mode_ctrl
nop	
nop	
nop nxt	jpc ← jpc+1