The J Computer

Advay Mengle

6.111 Spring 2007 Final Project
T.A: Amir Hirsch
May 17, 2007

ABSTRACT

This paper presents the J Computer, a processor that natively implements a subset of the Java Virtual Machine specification for Java ME CLDC v1.0a in hardware. The computer contains a 32-bit microprocessor, method code cache, and a number of hardware device modules, such as a tone generator, a PS/2 keyboard, and an alpha-blending VGA graphics renderer into which Java code running on the main processor can make remote procedure calls. The user may download arbitrary class files to the J Computer via an RS-232 connection. The design was verified in simulation, and synthesized on a Xilinx Virtex2 FPGA. Successful execution of Java implementations of a prime number discovery algorithm and a simple pong game with audio demonstrated the computer's functionality.

Table of Contents

I Introduction

We give below a detailed problem statement, background and motivation, an overview of how a user interacts with the J Computer, and the high-level system architecture and specification.

1 Problem Statement

The J Computer will natively execute a subset of the Java ME CLDC v1.0a bytecode in hardware. The main processor shall convert Java bytecode into a custom microcode, called simple operations (SOPs), developed for this specific purpose and thus optimized for the nature of Java instructions, and execute this microcode. The processor shall support a kernel (written in Java bytecode), the ability to execute multiple static methods, the ability to manipulate static fields, and perform 32-bit computation. The J Computer must be able to download Java class files (that have been preprocessed for formatting reasons, if necessary) via an RS-232 connection with a PC, and execute the downloaded code on the fly. The main processor must be able to connect to other modules in such a way that permits Java code to transparently perform remote procedure calls (RPC[1]) into these other hardware modules without knowledge of the internals of these modules. All functionality shall

[1] The phrase "remote procedure call" is used here as an appeal to familiarity, but does not resemble true RPC as in full Java.

be demonstrated by developing a prime number-finding program, and a simple pong game in which Java code controls the image displayed on the VGA. The design must be realized on a 6-million gate Xilinx Virtex2 XC2V6000 FPGA with limited BRAM.

2 Background and Motivation

Java is a popular high-level, object-oriented language developed by Sun Microsystems, Inc., specified in [2], often used for application and web programming. One of the strengths of Java is its attempt to be platform-independent, with the ability to write Java source code, compile to bytecode in class files, and then run those class files on any platform supported by Java. Originally designed to be executed by software virtual machines (VMs), specified in [3], that effectively acted as interpreters, converting bytecode to a processor's native instruction set during run-time, Java execution is now usually supplemented by just-in-time (JIT) compilation which can convert methods into native code once when first executed and then use the native versions for future invocations. The Connected, Limited Device Configuration (CLDC) of Java Micro Edition (ME), defined in [1], is (mostly) a subset of the desktop Java Standard Edition, and designed for use on portable devices with "limited" capabilities, with the most notable reduced functionality being floating-point computation and small minimum memory requirements (on the order of dozens of kilobytes instead of megabytes).

The motivation for running Java on an FPGA is two-fold. First, we wish to be able to quickly program the FPGA for general computation without having to perform the time-consuming process of FPGA synthesis, translation, mapping, placing and routing. The existing toolsets for the Java ecosystem far exceed the development environment available for traditional FPGA programming in terms of both user efficiency and friendliness, and working in Java instead of Verilog or VHDL is often more natural for software programmers. However, we recognize that the reason for developing in HDL is often to make use of the much faster and more parallel architecture of hardware; therefore our second motivation is to develop a seamless interface between Java code and other hardware modules, such that a programmer can make method calls into hardware almost as if the other modules were classes with static methods written in Java, balancing the need for speed with ease of programming. The possibility of creating a processor that natively runs Java has been

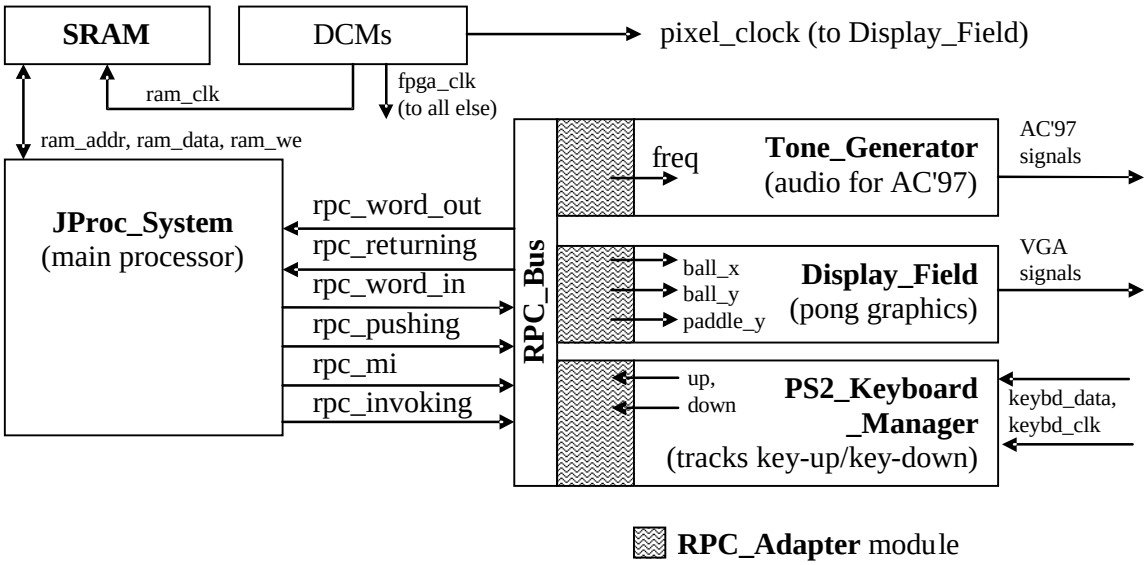

Figure 1 - System-level block diagram. The main processor connects to other modules via the RPC bus.

examined both by Sun (ex. [6]) and others (ex. [7]). Although acceleration of Java execution when using a microcode developed specifically and solely for running Java is theoretically possible, since the clock frequencies attainable on an FPGA implementation of the system are between one and two orders of magnitude slower than that of a modern PC and, in the general case, lack sufficiently parallelizable computations, running equivalent bytecode faster on the J Computer than on a software VM on a PC is beyond the scope of this project.

3 User Interaction

Users may interact directly with the J Computer and its hardware devices. To start execution of a class file, one must first boot up the J Computer. This is performed by flipping the reset switch (rightmost switch on the FPGA) from resetting (up) to running (down), pressing and releasing the second rightmost blue button (init button), and then downloading the appropriate JCA (J Computer archive) file via RS-232 at 115200bps, 1 stop bit, even parity, and no flow control. The method at index 0 (usually the main method of the first class) in the JCA will begin execution. At any time after downloading, the init button may be pressed again to restart method 0. The interaction with the hardware devices is of course device dependent, but for pong the up and down keys control the paddle and audio is played through the headphone jack when collisions or "game over" occurs.

4 System Architecture

The J Computer consists of two distinct parts, as in Figure 1, the main processor that performs all bytecode manipulation and processing, and the hardware modules that interact with the user; these two distinct parts are connected via the RPC method bus. The main processor depicted in Figure 2 consists of a class memory manager, a metadata manager, a bytecode reader with method cache, a bytecode-to-SOP translator, a SOP processor (with 32-bit ALU, register file, and stack manager), and RS-232 UART. Details of each module are given in Section II.

Two versions of the J Computer were realized – a "medium memory" configuration, which has a 64KByte total stack, 64KByte method cache, 2KByte kernel ROM, 32KByte metadata table, and a "small memory" configuration where the memories was reduced down to 1 to 2KByte each; both configurations have 2MByte of program code RAM. Other than the difference in memory sizes, the configurations are identical.

II Implementation

The most important modules are described below. Verilog source code for each of the modules can be found in Appendix D.

1 Class Memory Manager

The Class Memory Manager (CMM) is an FSM responsible for downloading the JCA file (described in Appendix C), and routing the various parts of the file to the appropriate modules. It is connected to the RS-232 UART via a FIFO Reader, a minor FSM which handles blocking calls to read bytes from any FIFO (in this case, the RS-232 receiving buffer). When the user presses the init button, the CMM idles, waiting for bytes to appear on the FIFO reader. The first 4 bytes of the file indicate the size of the remaining part of the file. The next section contains the metadata (i.e. constants, static fields, and method addresses); this part of the file is transparently handed-off to the Metadata Manager (MDM). Moreover, the CMM is not aware of the contents or length of this section and simply transmits the bytes from RS-232 directly to the MDM until the MDM indicates that there is no more metadata.

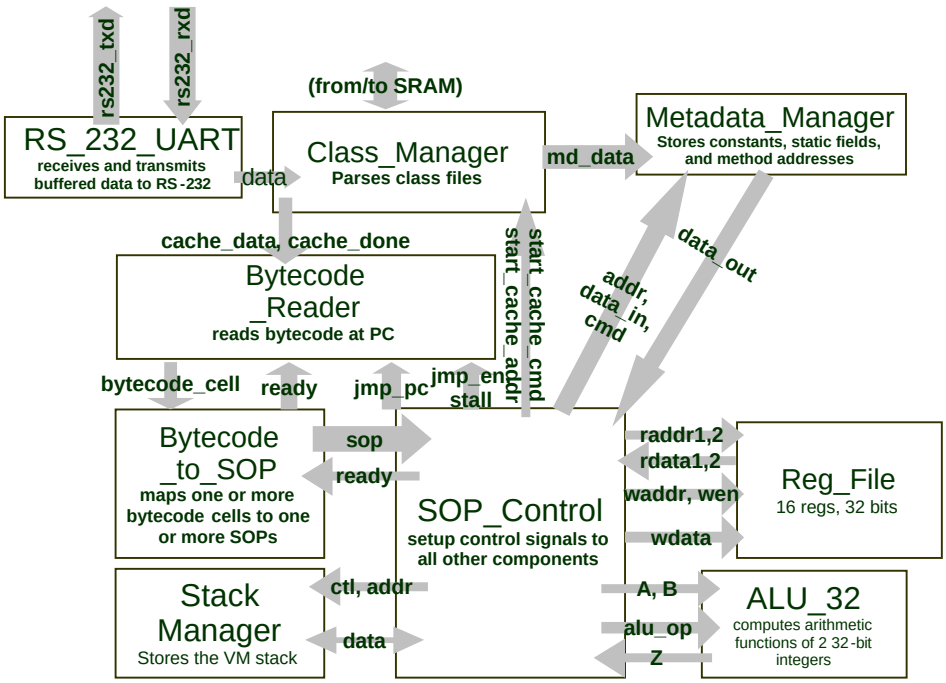

Figure 2 - Main processor block diagram (simplified). For ease of viewing, some less important wires are not displayed.

The remainder of the file contains the bodies of all the methods. Once again, this part of the CMM is unaware of the contents of this part of the file, but simply stores it into the ZBT SRAM, beginning with address 0, storing 4 bytes into each 36-bit word in SRAM. When the CMM is finished downloading the JCA, it flags init_done, indicating to the Bytecode Reader (BR) that it may initiate the trap that will eventually start the main method. Whenever the SOP processor asserts the "start caching" command, the CMM will start reading from the SRAM at the specified address and transmit the contents to the BR; since the first four bytes of every method indicate its length, the CMM is able to read only enough to send the method and not require a full 64 KB caching every time.

2 RS-232 UART

The RS-232 UART performs buffered transmission and reception over the RS-232 serial cable. The UART has two 512 byte FIFOs, one for each buffer. Since RS-232 transmission rates (115200 bps, or approx 9600 bytes per second after counting overhead) are much slower than the processor, the transmission buffer is likely to fill up; when this occurs the FIFO discards new bytes until a slot in the memory opens up due to the transmission of a byte. The analogous problem for receiving bytes should never occur since the rest of the system will process incoming bytes faster than the UART will receive them; thus our JCA files will retain their integrity through download.

3 Metadata Manager

The Metadata Manager stores the constants, static fields, and method addresses in block RAM. On JCA download, the bytes are stored in successive locations in the RAM. The RAM has 16-bit and 32-bit ports so downloaded bytes are shifted in as needed. The first 8 bytes give 16-bit counts of the quantity of constants, static fields, and methods (and one extra metadata type for future use, i.e.

classes). Each constant and static field is stored as a 32-bit word; each method address is stored as a 16-bit address (the CMM's SRAM has 19-bit addressing with 4 bytes per location, so methods must begin on 32-byte boundaries). The SOP processor can command the MDM to read a constant, read or write a static field, or read the address for a method. Because the number of each type of metadata is known ahead of time, we can read or write an item given a type and index in constant time (i.e. one clock cycle).

4 Bytecode Reader and Method Cache

The bytecode reader is responsible for serving the Bytecode-to-SOP translator with a stream of bytecodes from the currently executing method. These bytecodes are read from a 64KB method cache that always contains the full contents of the currently executing method. The BR keeps track of its own program counter (PC) which is an offset from the beginning of the currently executing method. Whenever the CMM indicates that a method is about to be cached, the PC is reset to 0, and the BR writes the method body data from the CMM into the RAM.

The bottom 2KB of the method cache are reserved for the kernel. The kernel is written in Java bytecode and stored in a ROM. Whenever the PC is greater than 62 KB, the bytecodes sent to the translator are actually being read from the kernel ROM. This allows a very easy implementation of software traps. Whenever an exceptional condition (not to be confused with Java exceptions, which, since they are full-fledged objects, are not supported yet) occurs, the bytecode reader jumps to a trap address. There are traps for the initial download of a JCA file, stack over/underflow, and illegal bytecodes (so some bytecodes could be implemented in kernel software in the future), and others.

5 SOP Processor

The SOP processor contain the stack manager, the control signal generator, the register file, and the 32-bit ALU. The register file has two read ports and one write port and stores its values in immediate registers, instead of a BRAM to facilitate single clock cycle ALU operations. The ALU contains separate hardware for every arithmetic, comparison and binary operation defined by Java on 32-bit integers, including a Coregen divider with remainder. All ALU operations take "less than" one clock cycle (they are computed asynchronously), except division which takes a number of cycles proportional to the number of bits being divided.

a Instruction Set Architecture

Because many bytecodes are difficult to execute in a single clock cycle, we developed a custom microcode called Simple Operations (SOPs) which are actually used to perform computation. Each simple operation instruction is 40-bits long, carrying an 8-bit opcode and either a 32-bit signed two's-complement literal, or 4 8-bit register identifiers. The 8-bit opcode is further divided into a 2-bit type-of-operation (TOp) and a 6-bit specific operation (SpecOp). The goal was to create an ISA that allowed certain operations that constitute the vast majority of bytecode execution to be performed in a single clock cycle, instead of the multiple clock cycles those operations might take in a another microcode (such as the Beta).

b Stack Manager

The stack manager maintains a large two-port BRAM that allow the pushing and popping single values to and from the Java VM stack; one port always reads from the current top of the stack, while the other is always writing to the next available location in the stack. Because almost every bytecode

performs at least one push or pop regardless of what other computation they perform, fast pushes and pops are essential to maintaining performance. The downside of this is that uncommon operations, like the manipulation of the stack pointer in order to read/write local variables or invoke/return from methods take multiple (three) clock cycles. However since even those operations also require standard pushes and pops, it is still guaranteed that this decision minimizes the total number of clock cycles used.

c Control Signals

In most cases the TOp determines which modules attached to the SOP processors are activated. The ALU TOp activates[2] the storage of the ALU computation result in a given register. The stack TOp sends a control command to the Stack Manager instructing it to perform an operation other than simply reading from the current stack pointer (ex. pop, push, temporarily move SP, restore SP, etc.) When the SOPs were first designed, we made the decision to let RS-232 operations have their own TOp; with only 4 possible TOps this turned out to be, in hindsight, a poor decision since it required putting both metadata operations and jump operations under a "general" TOp, which increased the number of levels of logic by one (to test whether a SOP with general TOp was a jump or a metadata op) than would otherwise have been required.

Most control signals generated by the SOP Processor are asynchronous and are thus generated immediately (with a propagation delay) after the Bytecode-to-SOP translator generates a new SOP on the positive edge of the clock. For those signals that are synchronous, they are generated on the negative edge of the clock in order to not waste an entire clock cycle. For example, the cache start and RPC invocation signals need to be available for the CMM and the RPC bus by the positive edge of the clock.

6 Bytecode-to-SOP Translator

The Bytecode-to-SOP translator is essentially a stream processor with the input stream being bytecodes in a method from the BR, and the output stream being SOPs sent to the SOP processor for execution. The SOP generation code is completely "macroized" (i.e. uses Verilog `define macros) so if we wish to change the design of the 40-bit SOPs (for example if we discover that certain encodings of opcodes allow greater logic optimization), we only have to modify the macros (and the SOP processor to use the new encodings).

The state transition diagram for the translator is given in Figure 3. In addition to the explicit state as seen in the diagram, the number of SOPs in the queue (sopsRem) and the array of array of SOPs themselves (sops) also contribute to the FSM's full state. The most common state is the DispatchSOP state in which a queue of SOPs are being delivered to the SOP processor. In this state, the translator asserts ready to the BR (indicating acceptance of more bytecodes) only when sopsRem is at 1, meaning that at the beginning of this cycle there is one SOP remaining in the queue and that at the end of the cycle there will be no SOPs in the queue – this scheme is used to avoid wasting one clock cycle at the end of the series of SOPs generated for a single bytecode. Certain bytecodes in Java takes parameters within the bytecode stream itself. For example, the bytecode to goto a particular location in the bytecode stream takes a signed 16-bit number (i.e. two 8-bit

[2] The phrase "activates" as it applies to the ALU may be misleading because the ALU is continually performing asynchronous computations on the A and B inputs at all times, even if they are garbage. "Activates" here means that the output of the ALU is actually used. It may be prudent to actually disable the ALU when not in use in order to lower power consumption during useless logic transitions.

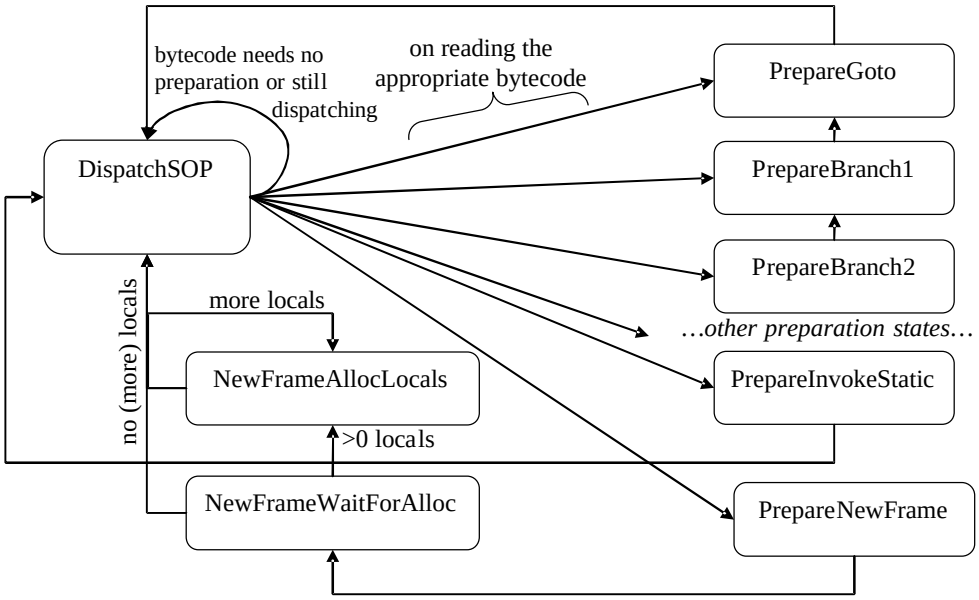

Figure 3 - State transition diagram for Bytecode-to-SOP translation FSM.

bytecodes) that identify the address to jump to relative to the address of the goto instruction – such bytecodes require preparation, i.e. additional bytecodes must be read from the stream in order to fully determine the sequence of SOPs that the bytecodes must be translated into. When a bytecode is initially read in the DispatchSOP, a large case statement determines whether or not the bytecode requires preparation, if so the FSM transitions into the appropriate . Most preparation states store the incoming parameters from the bytecode as literals in various SOP instructions. For example, the preparation for the getstatic instruction will place the immediate index into a save literal to register SOP. Finally, one special case is the X_NEWFRAME run on method invocation; Java requires all local variables to be initialized to 0, so a "for" loop is built into the FSM to push enough 0s on to the stack.

The translator receives stall and ready signals from the SOP processor. A stall instructs the translator to blank (i.e. replace with NOPs) SOPs that it sends until stall is lowered. A ready signal instructs the translator to send the next SOP in its queue, if ready is not asserted the queue is paused. Stalls are used for jumps and branches, where the SOPs that are in the pipeline need to be discarded, while bringing ready low is used to temporarily halt the stream when a SOP takes more than one clock cycle.

7 RPC Bus and Other Hardware Modules

The RPC bus incorporates multiple very simple few-state FSMs that continually sample the RPC method id to determine whether or not a given FSM is responsible for the current RPC call. If so, the appropriate RPC adapter transitions into a state in which it will read the parameters, and then perform the required operation and/or return a value to the processor. If an operation is expected to occur in parallel with Java execution on the processor, the method is simply declared to return void, and the processor will not wait for a return value. If a return value is specified, the processor stalls until a value is provided by the RPC adapter.

The VGA controller, alpha-blending overlay manager, PS/2 keyboard reader, and tone generator were developed and tested by the author in [4] and [5] and are not further discussed in this report. In our case, each RPC adapter

III Testing and Debugging

Initially, the SOP processor, ALU, register file, stack manager, and RS-232 were developed in a "waterfall" development process with their requirements being analyzed. The bytecode reader, class memory manager, metadata manager and all other components were developed incrementally adding support for a few bytecodes at a time. The testing process is described below. Sample test code is given in Appendix E.

1 Unit Testing

Until the entire system was nearly complete, running a partially finished J Computer on an FPGA would have been a poor way of performing testing because the only output we could have relied upon were the RS-232 (which transmits at a far slower rate than the processor) and the on-labkit LEDs. This demanded the extensive use of behavioral (for ensuring logical correctness) and post-place-and-route (for ensuring timing integrity) simulations. Simulation waveforms for a sample of the tests performed can be found in Appendix A. Any features which used the ZBT SRAM, however, were not tested in simulation (except to ensure that the modules' transmissions to the SRAM had proper timing) due to the lack of an accurate Verilog model of the SRAM.

2 System Testing and Demonstration

When unit testing had failed to determine a bug seen in on-FPGA testing, the logic analyzer proved invaluable. Often bugs which, when tackled by looking only at Verilog source, were very difficult and time-consuming to track down could be immediately identified by examining the analyzer waveforms.

Final system testing was performed through the use of two demonstration programs. The first is a prime finding algorithm – a Java implementation of the naïve $O(N^{0.5})$ algorithm of testing all odd numbers for factors up to $N^{0.5}$ that also detects twin primes. This program was used to test all computational, local variable, RS-232, and static method invocation features of the main processor. The pong game tested the RPC features of the processor (with output to the AC'97 audio codec and the overlay manager for pong). Both demonstrations of functionality were successful.

3 Resolved and Remaining Issues

There are no known remaining bugs. During the course of the implementation of the project, the most critical issues were those involving timing (including bugs where signals were off by entire clock cycles, and when hold/setup times were violated). Adding a single timing constraint to the UCF file not only fixed all the hold/setup times, but also considerably reduced place and route times (down from approximately 26 minutes to 7 minutes).

IV Conclusions

We present the results and potential opportunities for future work below. Overall, we were very satisfied with the completion of the main processor and RPC architecture.

1 Results

At this point, the J Computer supports a subset of the Java ME CLDC v1.0a functionality, with the tested and verified ability to execute 84 bytecodes out of the 144 bytecodes supported by CLDC. Supported features include all 32-bit arithmetic computation, local variable and parameter

manipulation, primitive casting to/from 32-bit values, stack manipulation, static field manipulation, loading constants, and static method invocation/return, amongst others. Appendix B lists the specific bytecodes supported by the J Computer.

After the completion of the main processor, only a little over a day remained to create all of the RPC components. It is a testament to both the versatility of the RPC architecture and the design of [4] and [5] that the components could be very quickly integrated and work "out of the box." The entire RPC architecture is inferred by a few dozen lines of Verilog. The last clock rate we successfully ran on the FPGA was 24MHz. However, synthesis tools indicated that the processor could be potentially clocked at 36MHz without any additional pipelining.

2 Future Work

We did not intend to fully support all Java ME bytecodes in this iteration of the J Computer due to the time constraints of this project. However, the logical next steps would be to add object/array instantiation and virtual method calling. Instantiation of objects (and arrays, since they, too, are first-class objects) on the heap using the current architecture should be fairly straight-forward. More difficult would be implementing a garbage collector (which, while not required by the Java VM specification, is a practical necessity) in hardware, but due to the flexible nature of the kernel, could be performed in software if the performance hit was not too large. Virtual method calling requires more complicated algorithms that, too, would be hard to implement in hardware but could be done in the kernel.

More ambitious would be supporting multiple main processor cores and performing thread scheduling on a single core. We believe that RPC between multiple cores should provide no further difficulties since the RPC protocol is already well-defined and tested. However given the block RAM memory constraints of an FPGA, we would have to develop a way to share memory between the cores. Multi-threading in and of itself is trivial given multiple instantiations of the Stack Manager (one for each thread), however locking and synchronization would have to be added to the supported bytecodes in order to make multi-threading useful.

3 Acknowledgements

The author thanks all of the MIT 6.111 Spring 2007 course staff, especially Professor Anantha Chandrakasan and the T.A. for this project, Amir Hirsch, for providing cogent advice and being eager to assist when needed. The author also thanks the suppliers of all the various labkit components for providing the opportunity to implement the design on a state-of-the-art FPGA system.

V References

1. "Connected, Limited Device Configuration. Specification Version 1.0a. Java 2 Platform Micro Edition." Sun Microsystems, Inc. 19 May 2000.
2. Gosling, James, Guy L Steele Jr., and Gilad Bracha. The Java Language Specification. 3rd ed. Sun Microsystems, Inc., 2005.
3. Lindholm, Tim, and Frank Yellin. The Java Virtual Machine Specification. 2nd ed. Sun Microsystems, Inc., 1999.
4. Mengle, Advay. "Implementation of 'MIT Pong' on an FPGA." 09 Apr 2007. Unpublished report.
5. Mengle, Advay. "Traffic Light Controller for a Main and Side Street Controller." 07 Mar 2007. Unpublished report
6. "picoJava." Sun Microsystems, Inc. 16 May 2007. <http://www.sun.com/software/communitysource/processors/picojava.xml>.
7. Schoeberl, Martin. JOP: A Java Optimized Processor for Embedded Real-Time Systems, PhD thesis, Vienna University of Technology, 2005.