Making Java Work for Microcontroller Applications

Sérgio Akira Ito and Luigi Carro
Universidade Federal do Rio Grande do Sul, Brazil

Ricardo Pezzuol Jacobi
Universidade de Brasilia, Brazil

The authors investigate complete system development using a Java machine aimed at FPGA devices. A new design strategy targets a single FPGA chip, within which the dedicated Java microcontroller—FemtoJava—is synthesized.

Embedded applications development is part of a new, promising computer systems market. However, applications based on devices embedded in consumer electronics have different design constraints than those of stand-alone systems. In embedded applications, for example, low power consumption, high code density, and the ability to integrate peripheral devices into the same circuit can be more important than performance requirements.

Time-to-market pressures and the proliferation of incompatible devices make software design a difficult task for consumer device developers. Embedded-system developers have embraced Java over the past few years because the language is abstracted from the underlying hardware, enhancing portability.[1] Java is a natural choice for embedded system development because of its ability to overcome some C and C++ problems.

New application requirements (such as multimedia processing) drive the embedded processor market to powerful 32-bit devices with software environments that can easily support the Java runtime environment.[2] In fact, we have seen that solutions now focus on embedded systems with enough resources to incorporate a real-time operating system (RTOS), a specific implementation of the Java virtual machine (JVM), multithreading support, garbage collection mechanisms, and so on.[3,4]

On the other hand, traditional eight-bit microcontrollers are still advancing into new products—boosted by low cost and new capabilities—increasing their estimated shipments. This change promotes interest in using Java for small microcontrollers. When considering a Java machine as the target architecture, there is one primary issue: the Java platform's suitability for implementing embedded applications in devices with eight to 16-bit CPUs and limited memory. For very simple embedded applications, however, the mechanisms just discussed (the JVM, RTOS, and so on) can be too costly for the runtime environment to support.

A processor like PicoJava was designed to obtain performance and could not address resource-constrained applications (such as garage door openers, embedded controls in portable devices, and identification and security systems). Java microcontrollers can be the best choice in these applications.[5]

The simplest way to implement an execution engine for Java in hardware is through a stack machine compatible with the JVM specification. Compiling Java to native code or designing a microprocessor with another behavior loses the valuable software compatibility feature.

0740-7475/01/$10.00 © 2001 IEEE

Related work

The Java card platform[1] is targeted for developing applications that run in environments as small as those with 512 bytes of RAM, 16 Kbytes of ROM, and an eight-bit CPU. This platform supports dynamic object creation and has a two-part JVM and reduced application programming interface (API). It is exclusively targeted to devices like smart cards, because it depends on card acceptance devices to run the applications.

Some lightweight JVMs are designed to run dynamically loaded applications in embedded systems, such as the Hewlett-Packard JVM, Kaffe, and Spotless System.[3,6,7] However, Sun Labs reports that the Spotless project's goal—designing a Java platform for devices with a few kilobytes of RAM for both the runtime environment and applications—may be an arduous task. Sun also claims that the PicoJava microprocessor can be configured for an embedded market. However, since PicoJava's microarchitecture incorporates sophisticated mechanisms for performance gain, it seems impossible for it to fit in a smaller space than a classic microcontroller.[5]

Compiling Java source or bytecodes to native code could overcome poor performance and help maintain a smaller runtime environment. However, this approach requires a compiler (or compiler back end) for each new device, as required in the microcontroller market, making software portability more difficult to attain.

The JASIP (Java application-specific integrated processor) architecture is an interesting solution to native multithreaded Java application execution.[4] This architecture requires a processing element for each thread, early class hierarchy resolution, and object allocation support at global-memory and thread scheduling. However, implementing its prototype required an FPGA (field-programmable gate array) board with 100,000 gates and a memory module. These requirements don't fit the concept of a single chip, low-power application.

Java has also been studied as a specification language for embedded systems and hardware-software systems. In this research area, Young presented an approach to synthesize hardware from system specifications using Java and some restrictions in the application modeling.[8] However, this codesign methodology is targeted to generate hardware pieces that accelerate Java applications, making no assumption about resource constraints. The Sashimi (system as software and hardware in microcontrollers) approach shares some concepts with these works, such as system specification using a subset of Java, CPU customizing, and early reference resolution. However, we provide a general methodology to support the development of embedded applications, based on a single-language and single-chip approach to reducing costs.

Embedded-processor requirements

Microprocessors designed for the embedded market have different constraints than desktop microprocessors. Embedded applications include video game consoles, modems, set-top boxes, digital cameras, cellular phones, printers, and so on.[2] Resource-constrained applications like building-access controllers, watches, pens, smart cards, and smart rings are also considered embedded applications. For the last set of applications, restrictions include

- *Power consumption.* In addition to reducing power supply voltage, the processor standby mode reduces power consumption.
- *Program size.* Small program code size allows application execution in limited-memory devices. The stack machines allow more compact code than complex-instruction-set computing (CISC) and reduced-instruction-set computing (RISC) processors.
- *Microarchitecture optimizations.* Depending on the application, an application-specific instruction set processor (ASIP) can be a good solution. Also, address and data widths and register file size can be adjusted to fulfill application needs.
- *Higher levels of integration.* Integrating memory and communication interfaces in the same die can save power and simplify system design. A serial communication interface also reduces embedded processor

costs for specific applications.

- *Design reuse.* Considering costs and time to market, design reuse can be a very effective approach to reducing problems in the embedded market.
- *CAD support.* Having adequate CAD support is key to achieving higher productivity and reliability, high-quality products, and effective design reuse.

The cost of building a new compiler should be considered when ASIPs are going into a design. In addition, for most embedded applications, compilers cannot produce code that is as compact and efficient as programs hand-coded using assembly language. However, assembly-level programming has problems such as software compatibility, reuse, and maintenance cost.

We have omitted multimedia acceleration and special application software acceleration because our focus is on applications without mass-processing requirements.[2] Our target applications can be implemented with low-cost microcontrollers. Because of this focus, we also consider generating an ASIP to reduce processor size rather than gain performance.

In fact, we explore Java code compactness to overcome the code size problem and generate an ASIP to deal with the hardware cost issue. We also provide an appropriate CAD framework to make new compilers unnecessary and take advantage of Java's software compatibility through the FemtoJava microcontroller.[9] Moreover, we synthesize the system using FPGAs because of their flexibility and low cost.

FemtoJava microcontroller

There are several alternatives for running Java programs: native and just-in-time (JIT) compilers, interpreters, or a Java processor. Native compilers do not consider the software compatibility issue because they compile Java into native code. JIT compilers and interpreters maintain this Java feature but incur some overhead cost, because they require more memory to run the software, which results in poorer performance. Java processors can execute JVM bytecode natively, exhibiting a specific architecture organization (most are stack machines) to run Java programs efficiently. This is why Java processors can concurrently address the software compatibility and performance issues of Java programs. Although Java interpreters running on embedded processors would also provide such compatibility, the performance penalty is evident.

Our target domain application requires a simple microcontroller, but must still execute a Java program with the desired throughput. In addition, we defined the microarchitecture for our FemtoJava microcontroller based on studies of the JVM architecture and information we gathered about existing Java processors.

The JVM is based on a stack architecture and is an abstract machine with Java bytecode execution capability.[10] In general, the JVM has three major components: the class loader, class verifier, and execution engine. In fact, the class loader and verifier act at runtime and are only necessary if you want a multiapplication platform and have to download code over a network. We are using a compiler that obeys the JVM specification and will synthesize an ASIP version of FemtoJava. Only the execution core and some tools to extract the software at design time are really necessary.

Most Java processors support stack operations through stack emulation on their register files.[4,5] This approach reduces the memory access bottleneck of the stack machine, improving performance. While several proposed mechanisms (such as instruction folding, stack caching, and pipelining) address the performance issue in desktop systems, it is difficult to find work about supporting Java in small devices.

The JVM instruction set is large and complex. There are 226 instructions with varying formats, and many instructions correspond to awkward functions. Despite these characteristics, the JVM instruction set remains incomplete. To maintain software portability, Java's designers removed instructions that make assumptions about the hardware. For instance, the JVM does not include specific I/O instructions or addressing modes that involve registers. These omissions are why you must add instructions to the JVM instruction set to run the Java code directly on hardware.

Table 1. FemtoJava instruction set.

Instruction type	Mnemonics
Arithmetic and logic	iadd, isub, imul, ineg, ishr, ishl, iushr, iand, ior, and ixor
Control flow	goto, ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, return, ireturn, and invokestatic
Stack	iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, bipush, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, and swap
Load/store	iload, iload_0, iload_1, iload_2, iload_3, istore, istore_0, istore_1, istore_2, and istore_3
Array	iaload, baload, caload, saload, iastore, bastore, castore, sastore, and arraylength
Extended	load_idx, store_idx, and sleep
Others	nop, iinc, getstatic, putstatic

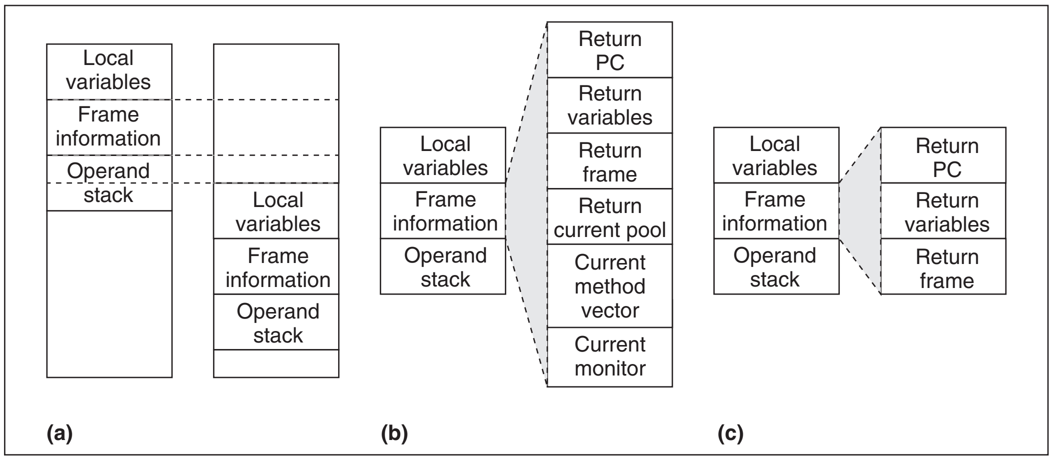


Figure 1. Frame allocation onto the stack: JVM (a), PicoJava (b), and FemtoJava (c) models.

After studying previous Java processors, we realized that supporting the full JVM instruction set in hardware was impractical for our project. Then, we started by compiling pieces of Java code and checked the resulting executable program. We observed that only a few instructions were really necessary to implement typical embedded applications.

This observation let us define a basic subset—just 68 instructions—for the FemtoJava microcontroller to support.[9] This subset includes instructions necessary to perform basic integer and stack operations, array manipulation, conditional and unconditional jumps, execution of the Java static methods, and class field access.

Table 1 presents the FemtoJava instruction set using the mnemonic convention of the JVM specification. The extended bytecodes are necessary to perform I/O operations, for interrupt programming (discussed next), and also to put the microcontroller in suspend mode. The FemtoJava microcontroller can run only class code because its instruction set includes only invokestatic, return, and ireturn as method instructions. However, this is not a serious limitation because it is not necessary for most embedded software to allocate complex objects at runtime.

Memory organization

Frame allocation plays an important role for the Java program's execution compatibility, since several instructions (such as load and store) use the current frame as a base to calculate the correct target addresses. In such cases, FemtoJava implements a Java-compatible frame allocation scheme, as shown in Figure 1.

Compared to that of processors such as PicoJava, the FemtoJava frame allocation is much simpler. In Figure 1, the JVM model only specifies that stack operands must be kept on top of the frame information allocated over local variables' space. Because we designed FemtoJava for single-threaded applications and static linked code (with merged method vectors and constant pools), we don't need to store as many fields on frame information as PicoJava does. We simply remove the information on monitors, method vectors, and constant pool.

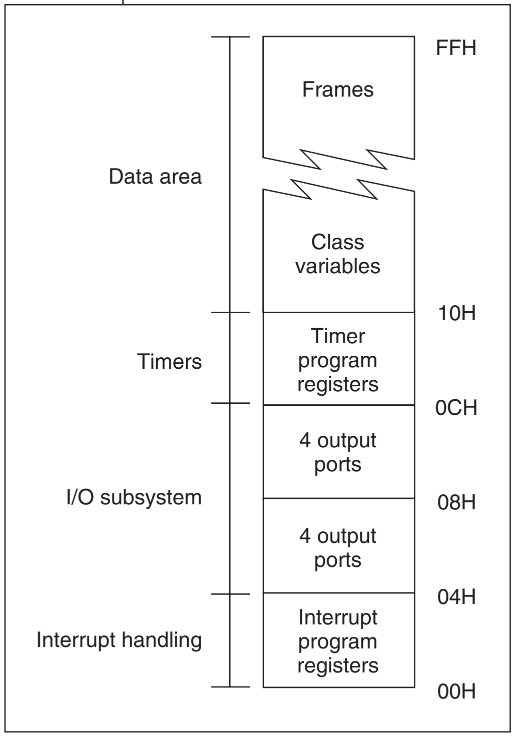


Figure 2. FemtoJava data memory organization.

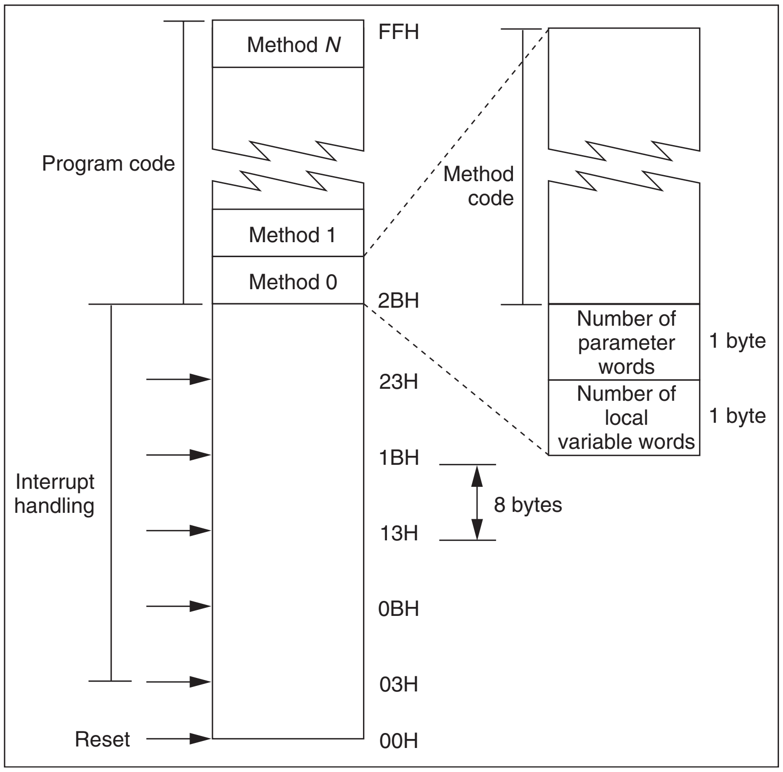


Figure 3. FemtoJava program memory organization.

The JVM or Java processors with enough resources for an RTOS can organize the program and its data in memory by using well-known object-oriented techniques. However, the microcontroller environment cannot always support dynamic Java features. For this reason, we defined specific schemes to organize both the program and data memories to obtain a simple hardware implementation. This implementation provides capabilities for I/O port mapping and the static linking of application code.

Data memory is organized as shown in Figure 2; its initial address space, from position 00H to 10H, contains some memory-mapped registers. These registers are intended for interrupt and timer programming, and also for I/O operations. The remaining memory is for storing class fields (variables and constants) and allocating frames for Java methods.

The techniques used to map the code and class information into program memory must be carefully considered to produce simple hardware structures. For FemtoJava, we used the mapping illustrated in Figure 3. In the memory's first part, we store the code necessary to call methods for handling interrupt requests. For faster method calls (and frame allocation), we store some information on the method heading. These fields let stack-pointer calculations skip over the stack and correctly restore the frames when return and ireturn instructions execute.

FemtoJava is a Java microcontroller with a reduced-instruction-set Harvard architecture. The FemtoJava microarchitecture, shown in Figure 4, uses simple building blocks. It contains some multiplexers, a few registers, memories, and a unique arithmetic logic unit (ALU). Inside the ALU we include a barrel shifter, a Booth parallel multiplier, a ripple carry adder/subtractor, and a Boolean logic unit. The component architecture was selected for hardware cost reasons, but can be replaced by others, if necessary, because we use FPGAs as synthesis target devices.

In our work, we exploit the reconfigurability of FPGAs to synthesize customized versions of FemtoJava for each application. In this case, the target FPGA characteristics had some influence on design. In fact, in the implemented microarchitecture, we changed buses to multiplexers

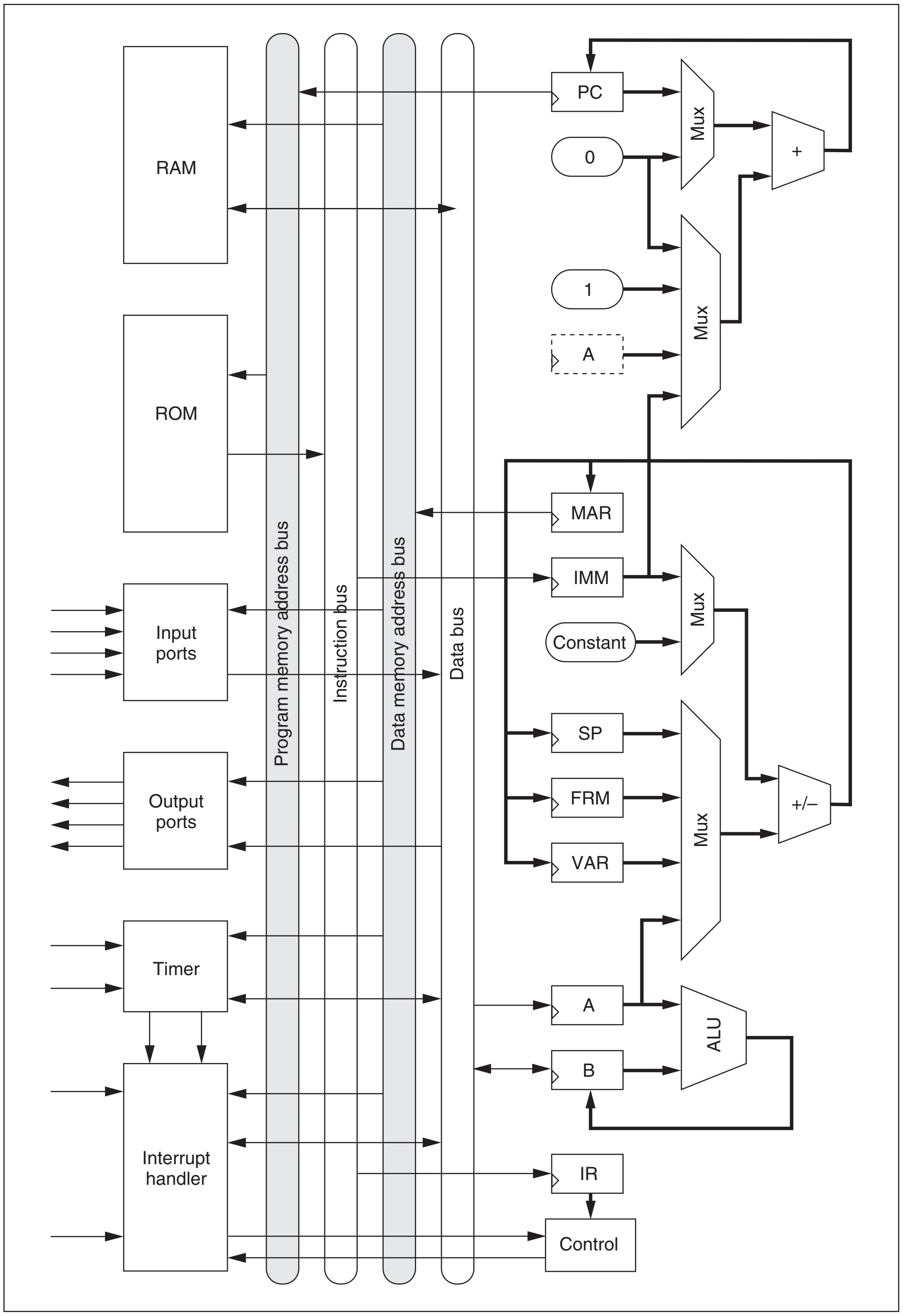


Figure 4. FemtoJava microarchitecture.

because of difficulties in using buses inside the FPGAs of Altera's Flex 10K series. Also, sharing the same ALU to execute the instructions and to operate over stack pointer (SP) and program counter (PC) registers makes no sense, because the FPGA architecture makes an adder's size

Table 2. Characteristics of synthesized microcontrollers. Microcontrollers are synthesized to the Altera Flex EPF10K30RC240-4 unless otherwise noted.

Microcontroller	Version	Application	No. of logic cells	FPGA device usage (percentage)	Frequency (MHz)	No. of distinct instructions required
FemtoJava	8 bits	None	1,481	85	4.85	68
		Biquad	991	57	7.97	22
	16 bits	Synthesized	1,979	85	3.93	69
		ECS	1,556	90	5.65	31
		Podos	1,465	89	5.55	29
		Translator	1,253	72	5.13	32
8051 ASIP	8 bits	Synthesized	659	38	3.07	14
Risco*	32 bits	Synthesized	1,271	25	4.03	35

* Synthesized to the Altera Flex EPF10K100GC503-3

similar to that of a multiplexer. Moreover, this design strategy lets some instructions take fewer clock cycles to execute.

The FemtoJava core can execute all supported instructions in, at most, 14 cycles. Some instructions are memory bound and others are much simpler, like *iadd*. An important observation is that decoding Java instructions needs complex hardware structures, and reducing the number of control-machine states can help reduce circuit area. Therefore, to save hardware, we grouped instructions according to the number of cycles they take to execute. FemtoJava has four categories of instructions: those that execute in three, four, seven, or 14 cycles.

We built the implementation using VHDL (VHSIC hardware description language); we performed synthesis and analysis in the Maxplus-II environment from Altera. FemtoJava's VHDL code uses a style that makes changing data path widths easy, and adapting its instruction set is straightforward for each target application. In fact, the Sashimi tools can automatically adapt to the FemtoJava architecture.

For embedded systems applications, it is very important to define structures like timers, I/O ports, and interrupts, despite their absence in the JVM specification. We have included the corresponding VHDL models for these components in our microcontroller.

Table 2 presents the synthesis results for the FemtoJava and two other microprocessor cores. The 8051 ASIP core starts from a reduced-instruction-set version of the well-known Intel 8051 microcontroller. The Risco is a 32-bit RISC-like microprocessor with a three-stage pipeline. The applications we used as benchmarks are

- the classic biquadratic filter,
- a simple elevator control system (ECS),
- an algorithm used on a portable device to measure the distance that a person walks or runs (Podos), and
- a hash-based searching algorithm used to translate words.

In fact, we can see that FemtoJava can be used in applications requiring compact hardware. Note that 8051 ASIP and Risco cores do not include interrupt-handling mechanisms and timers. In terms of program size, Table 3 shows that FemtoJava can implement the application software more compactly. In this case, we stripped out the code for programming interrupts and timers for comparison with other microprocessors. Table 3 (next page) also shows that the available instructions on the 8051 ASIP are insufficient to implement those applications without modifying the code.

CAD framework

We developed the Sashimi environment to support automatic adaptation of software, and the ASIP and application-specific integrated circuit (ASIC) generation. Using Sashimi, the

Table 3. Software generation. Microcontrollers are synthesized to the Altera Flex EPF10K30RC240-4.

Microcontroller	Version	Application	Program size (bytes)	Data memory (bytes)	No. of distinct instructions required
FemtoJava	8 bits	Biquad	49	30	22
	16 bits	ECS	612	74	31
		Podos	246	90	29
		Translator	280	118	32
8051	8 bits	Biquad	62	3	20
		ECS	602	15	37
		Podos	549	31	67
		Translator	358	42	63
RISCO	32 bits	Biquad	332	80	10
		ECS	2,224	120	17
		Podos	1,064	160	14
		Translator	884	164	11

designer can model, simulate, and build the system implementation directly in Java. We provide libraries that improve simulation accuracy and allow direct mapping of classes used by simulation to actual code in the final implementation. These predefined classes also cover all the details required to interface the microcontroller with the real world (interrupt mechanism programming, communication with LCD displays, and keyboards).

The automated tasks performed by the development environment are code analysis, performance estimation, and critical-routine identification. A set of tools can help the designer predict the final system performance and costs.

Design process

The Sashimi design environment uses freely available tools, like the Java compiler and the JVM included in the Java development kit (JDK). In addition, we also provided tools specifically designed for Sashimi. Figure 5 (next page) illustrates the entire design flow from the Java source code to the synthesized microcontroller chip. In the Sashimi environment, the user starts with Java files representing the application source code. In this phase, the development process follows the traditional edit-compile-run cycle in a desktop computer with a standard JDK. In this scenario, running the application is equivalent to simulating it in still-unavailable hardware, which, in this case, is emulated by the Java interpreter. During simulation, the designer can use predefined classes (where threads are allowed) to model the behavior of necessary peripherals. Later, in the synthesis step, the system will replace these classes for code providing the interface with real components.

When the designer considers the application ready, the user-provided input vectors and the simulation-generated output vectors are saved for later use by the bytecode validation phase. The code analyzer tool will take the executable code (class files) to estimate quantities like performance and hardware size. The analyzer tool estimates performance using method-call information, number of instructions per method, cycles per instruction, and probable frequency to be reached after synthesis. FPGA area and frequency are estimated by using previous results obtained in synthesizing FemtoJava with different numbers of instructions. Since FemtoJava does not support the full JVM instruction set, the code analysis phase must provide information for ASIP generation and code adaptation.

Adapting bytecodes involves some transformations in class files, while semantic modifications are not allowed. This process transforms complex instructions (such as tableswitch and lookupswitch) in a sequence

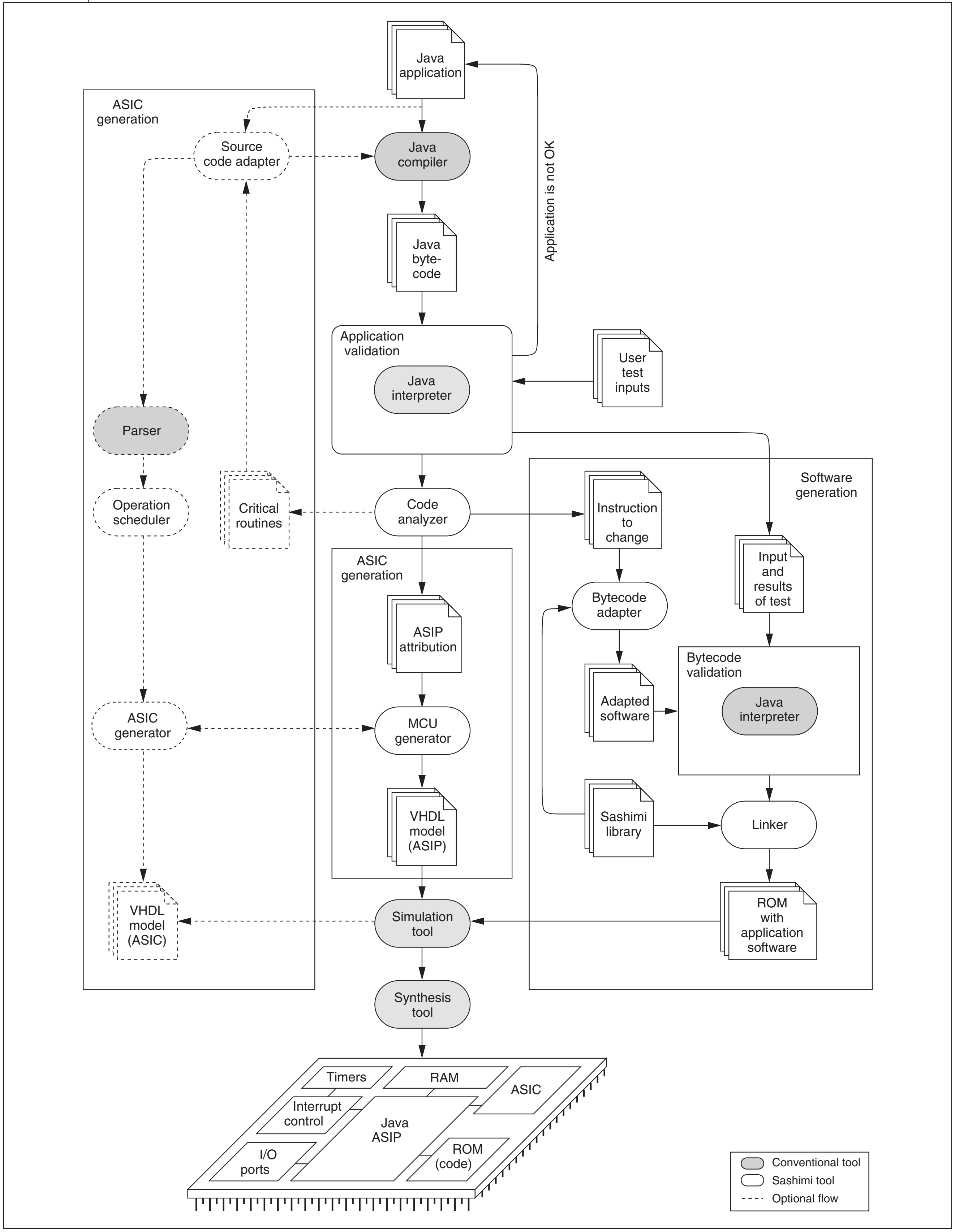


Figure 5. Sashimi design flow.

of simpler instructions. Class file structures keep the maximum stack size for each method in a class, and this information must be recalculated to allow correct adapted-code execution. After this step, the tools perform a new simulation to validate the bytecode transformation. This process uses the previously stored output test vectors and can be automatic.

The criteria that drive the bytecode substitution process are the instructions supported by the target microcontroller; size and speed requirements; each instruction's use rate; and the memory size available to store the application code. At this point in the design flow, a set of class files is adapted to be compatible with the specific instruction set that will be implemented in the Java ASIP. The next step removes unnecessary information (such as line number structures) and resolves class hierarchy. It also links and converts the application code and the necessary libraries in a unique program memory image. In addition, useless bytecodes for embedded software generation (such as calls for the System.out.println method used during simulation) are discarded, based on information generated during the previous phases.

When performance estimates don't match application requirements, the code analyzer tool can identify the critical code, and the designer can choose whether to discard some instructions. Alternatively, the designer can provide an ASIC specification for integration into the hardware to improve performance. Communication between the ASIC and the Java ASIP depends on the ASIC's behavior, but can be through either the stack or dedicated addresses of main memory. In the future, Sashimi will provide direct synthesis of Java methods to VHDL and automatic interface generation between FemtoJava and the generated ASICs.

System model

FemtoJava must be able to run the code stored in ROM. From attributes (such as instructions to implement and RAM size) extracted by the code analyzer, an optimized microcontroller VHDL model is generated. Finally, this model is simulated using the target application and synthesized in FPGA using any synthesis tool that accepts a VHDL input. The code is stored in nonvolatile memory, and only necessary classes are linked. The application is updated by replacement of this memory module or reprogramming (if using erasable memories).

The microcontroller can also be updated because it is synthesized in an FPGA. Adding new processor functions involves creating new Java instructions required by some specific application (by changing the specification file with available instructions) or synthesizing specific hardware functions, like timers, watchdogs, or even digital filters.

Modeling constraints

Target applications induce some constraints in coding style. These restrictions are similar to those that make the synthesizable version of VHDL smaller than the full language, which was originally oriented toward simulation. Therefore, a Java application must conform to the following conditions to be synthesizable in Sashimi:

- the new operator is not allowed, since it would require virtual memory management by the hardware machine;
- only static methods and variables are supported, for the same reason;
- no recursive methods are allowed, since they'd require dynamic memory management;
- interfaces are not supported, because dynamic binding represents additional cost at runtime;
- floating-point arithmetic is not allowed, although it could be enabled at the price of a larger FPGA; and
- multiple threads are not supported, because most microcontroller applications can be described in a single thread, lowering design and hardware costs.

We apply these rules to ensure that the generated software is fully implemented by the FemtoJava instruction set. Providing some new instructions or informing Sashimi to implement some parts through ASICs are alternatives that would let designers relax these rules.

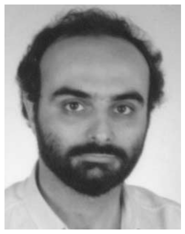
It is possible to synthesize small Java microcontrollers in a single FPGA chip. The microcontroller executes Java bytecodes natively, with no new compiler or JVM implementation required. Because they are reconfigurable, the FPGA devices provide opportunities to update microcontroller capabilities. The design environment fully supports generating an optimized microcontroller and the adapted code.

We plan several improvements for Sashimi and FemtoJava. We are implementing new versions of the microcontroller that include testability features, a pipeline, and a smaller core. Some tools for Java to VHDL synthesis are already implemented and tested to determine how to perform the final system integration through interface generation. The availability of wrappers to perform cosimulation with existing design environments, like those from Synopsys and Matlab, will be key for Sashimi's dealings with legacy design parts. ■

■ References

1. D. Mulchandani, "Java for Embedded Systems," *IEEE Internet Computing*, vol. 2, no. 3, May/June 1998, pp. 30-39.
2. M. Schlett, "Trends in Embedded-Microprocessor Design," *Computer*, vol. 31, no. 8, Aug. 1998, pp. 44-49.
3. M. Barr, "A Free Java Virtual Machine for Embedded Systems," *Proc. Embedded Systems Conf.,* Miller Freeman, San Francisco, 1998, pp. 277-288.
4. M. Mrva, K. Buchenrieder, and R. Kress, "A Scalable Architecture for Multi-threaded Java Applications," *Proc. 1998 Design Automation and Test in Europe* (DATE 98), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 868-874.
5. H. McGhan and M. O'Connor, "PicoJava: A Direct Execution Engine for Java Bytecode," *Computer*, vol. 31, no. 10, Oct. 1998, pp. 22-30.
6. D. Clark, "HP Enters the Java Fray," *Computer*, vol. 31, no. 6, June 1998, p. 19.
7. A. Taivalsaari, B. Bush, and D. Simon*, The Spotless System: Implementing a Java System for the Palm Connected Organizer,* tech. report SMLI TR-99-77, Sun Microsystems, Palo Alto, Calif., 1999.
8. J.S. Young et al., "Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement," *Proc. Design Automation Conf.*, ACM Press, New York, 1998, pp. 70-75.
9. S.A. Ito, L. Carro, and R.P. Jacobi, "Designing a Java Microcontroller to Specific Application," *XII Brazilian Symp. Integrated Circuit Design* (SBCCI 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 12-15.
10. B. Venners, *Inside the Java Virtual Machine,* McGraw-Hill, New York, 1998.

Sérgio Akira Ito is a research assistant in the Electrical Engineering Department, Microelectronics Group of Universidade Federal do Rio Grande do Sul (UFRGS), Brazil. His reseach interests include computer architecture, EDA, embedded systems, and core-based design. Ito has a BS in computer science from the Universidade Estadual de Maringa (UEM), Brazil, and an MS in computer science from UFRGS, Brazil.


Luigi Carro is a professor in the Electrical Engineering Department of the UFRGS. His research interests include mixed-signal design, digital signal processing, and rapid system prototyping. Carro has a BS in electrical engineering and a PhD in computer science from UFRGS.

Ricardo Pezzuol Jacobi is a professor in the Computer Science Department, Universidade de Brasilia, Brazil. His research interests include embedded system design, CAD, and reconfigurable architectures. Jacobi has an MsC in electrical engineering from the UFRGS, Brazil, and a PhD in applied sciences from Katholieke Universiteit Leuven, Belgium.

■ Direct questions and comments about this article to Luigi Carro, Departamento de Engenharia Eletrica, Universidade Federal do Rio Grande do Sul, Av. Osvaldo Aranha 103, CEP 90035-190, Porto Alegre, RGS, Brasil; carro@iee.ufrgs.br.