picoJava-II in an FPGA

Wolfgang Puffitsch
Institute of Computer Engineering
Vienna University of Technology, Austria
e0125944@student.tuwien.ac.at

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

ABSTRACT

picoJava is a Java microprocessor developed by Sun to speedup execution of Java in embedded systems and an often-cited reference design for other Java processors. Information about implementations of picoJava is rare however. In contrast to a number of new Java processors which are targeted at FPGAs, picoJava was designed for ASICs, and no implementation in an FPGA is known up to date. In this paper we show the implementation and evaluation of Sun's picoJava-II microprocessor in an FPGA.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Run-time environments, Java*

Keywords

Java processor, FPGA

1. INTRODUCTION

Java is a promising language for embedded systems, due to its object-oriented paradigm, its portability, robustness and security. The latter three are achieved by compiling the source code into a platform independent representation. Usually, this representation is interpreted or executed via just-in-time compilation; both ways are not feasible in embedded systems for reasons of performance and worst-case execution time (WCET) predictability, however. To increase performance and in some cases help WCET analysis, several Java processors have been developed, most prominently picoJava, which was developed by Sun Microsystems and first released in 1997.

Although picoJava is often referenced in research papers about other Java processors, information about implementations is rare. The processor was never released commercially, and only one research paper related to the implementation of picoJava in an ASIC could be found ([10]). This paper describes the implementation of the picoJava processor in a FPGA and compares it to other processors.

JTRES '07 September 26-28, 2007 Vienna, Austria

The paper is structured as follows: The rest of this Section gives an introduction into the Java Virtual Machine (JVM). Related work on Java processors is described in Section 2. In Section 3 an overview of the picoJava architecture is provided. Section 4 describes the steps which had to be taken to convert the provided sources into a running system, which is evaluated in Section 5. Finally, a conclusion is drawn in Section 6.

1.1 The Java Virtual Machine

The Java virtual machine [16] is an abstract computing machine designed to support the Java programming language [6]. In order to enhance portability and to achieve a high code density, it is a stack machine with variable-length instructions (called *bytecodes*) [17]. Furthermore, its instruction set is left intentionally incomplete, because one design goal was to provide a high level of security. This includes that memory is treated as black box so a malicious program cannot exploit a certain memory layout. As a consequence, the JVM must rely on an underlying operating system, or at least rudiments thereof.

The 201 bytecodes defined by the JVM specification span a wide range of complexity: from very simple operations (like `iadd`) through floating point operations (like `dmul`) to highly sophisticated operations like `anewarray` which resolves a symbolic reference to a class and allocates an array of that type on the heap.

A fully compliant implementation of the JVM must be able to parse the *class* file format, dynamically load new classes and to verify loaded classes. As the Java programming language relies on garbage collection for memory management, this also has to be implemented. These constraints entail that a fully compliant implementation of the JVM cannot consist of pure hardware, but also has to include some software as well.

2. JAVA PROCESSORS

Since the appearance of Java in 1995 many projects, both in industry and academia, have been devoted to speedup Java bytecode execution through hardware implementation. Despite promising first results Java processors are still a niche product. Many industrial projects disappeared after a few Years. The fact can be explained by the advances in just-in-time (JIT) compilers. JIT compilation is the standard execution mode of the JVM in desktop and server environments. In the embedded domain, where memory resources are scarce, a Java processor is still an option. Especially in real-time systems JIT compilation is not feasible. The compilation during runtime is hard to predict and introduces a high variability in the execution time. In the following section we give an overview of Java processors.

2.1 ASIC Designs

Sun introduced the first version of picoJava [19] in 1997. Sun's picoJava is the Java processor most often cited in research papers. It is used as a reference for new Java processors and as the basis for research into improving various aspects of a Java processor. Ironically, this processor was never released as a product by Sun. A redesign followed in 1999, known as picoJava-II that is now freely available with a rich set of documentation [23, 24]. The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions [19] and is the most complex Java processor available. According to [10] the processor can be implemented in about 440 K gates.

Simple Java bytecodes are directly implemented in hardware and some performance critical instructions are implemented in microcode. picoJava traps on the remaining complex instructions and emulates this instruction. A trap is rather expensive and has a minimum overhead of 16 clock cycles. The worst-case interrupt latency is 926 clock cycles [24]. This great variation in execution times for a trap hampers tight WCET estimates.

aJile's JEMCore is a direct-execution Java processor that is available as both an IP core and a stand alone processor [1, 12]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. Two silicon versions of JEM exist today: the aJ-80 and the aJ-100. Both versions comprise a JEM2 core, 48 KB zero wait state RAM, and peripheral components. 16 KB of the RAM is used for the writable control store. The remaining 32 KB is used for storage of the processor stack. The aJile processor is intended for real-time systems with an on-chip real-time thread manager.

The Cjip processor [11, 14] supports multiple instruction sets, allowing Java, C, C++ and assembler to coexist. Internally, the Cjip uses 72 bit wide microcode instructions, to support the different instruction sets. At its core, Cjip is a 16-bit CISC architecture with on-chip 36 KB ROM and 18 KB RAM for fixed and loadable microcode. Another 1 KB RAM is used for eight independent register banks, string buffer and two stack caches. Cjip is implemented in 0.35-micron technology and can be clocked up to 80 MHz. The JVM is implemented largely in microcode (about 88% of the Java bytecodes). Java thread scheduling and garbage collection are implemented as processes in microcode. Microcode instructions execute in two or three cycles. A JVM bytecode requires several microcode instructions. The Cjip Java instruction set and the extensions are described in detail in [13]. For example: a bytecode `nop` executes in 6 cycles while an `iadd` takes 12 cycles. Conditional bytecode branches are executed in 33 to 36 cycles. Object oriented instructions such `getfield`, `putfield` or `invokevirtual` are not part of the instruction set.

Besides the *real* Java processors a FORTH chip (PSC1000 [20]) is marketed as a Java processor. Java coprocessors (e.g. JSTAR [18]) provide Java execution speedup for general-purpose processors. Jazelle [5] is an extension of the ARM 32-bit RISC processor. It introduces a third instruction set (bytecode), besides the Thumb instruction set (a 16-bit mode for reduced memory consumption), to the processor. The Jazelle coprocessor is integrated into the same chip as the ARM processor.

2.2 FPGA Designs

Hardware resources in FPGA are usually measured in number of logic cells (*LCs*) and on-chip memory. Converting the number of LCs to equivalent gate counts is problematic, but a factor of 5.5 to 7.4 is suggested in [21] for rough estimates.

Vulcan ASIC's Moon processor is an implementation of the JVM to run in an FPGA. The execution model is the often-used mix of direct, microcode and trapped execution. As described in [26], a simple stack folding is implemented in order to reduce five memory cycles to three for instruction sequences like *push-push-add*. The first version of Moon uses 3840 LCs and 10 embedded memory blocks in an Altera FPGA. The Moon2 processor [27] is available as an encrypted HDL source for Altera FPGAs (22% of an APEX 20K400E equates to 3660 LCs) or as VHDL or Verilog source code. The minimum silicon cost is given as 27 K gates plus 3 KB ROM and 1 KB single port RAM. The single port RAM is used to implement 256 entries of the stack.

The Moon project is a typical example for the lifetime of a Java processor company. When Vulcan ASIC started to develop Moon, a lot of information was available. This information was usually more of a presentation of the concept. Nevertheless it gave some insights into how they approached the different design problems. However, at the point at which the projects reached production quality, this information quietly disappeared from their web site. It was replaced with colorful marketing prospectuses about the wonderful world of the new Java-enabled mobile phones. Later the company and the web site faded into history.

The Lightfoot 32-bit core [9] is a hybrid 8/32-bit processor based on the Harvard architecture. Program memory is 8 bits wide and data memory is 32 bits wide. The core contains a 3-stage pipeline with an integer ALU, a barrel shifter and a 2-bit multiply step unit. According to DCT, the performance is typically 8 times better than RISC interpreters running at the same clock speed. The core is provided as an EDIF netlist for dedicated Xilinx devices. Lightfood consumes 3400 LCs and can be clocked at 40 MHz in a Xilinx Virtex-II FPGA.

LavaCORE [8] is another Java processor targeted at Xilinx FPGA architectures. It implements a set of instructions in hardware and firmware. Floating-point operations are not implemented. A 32×32-bit dual-ported RAM implements a register-file. For specialized embedded applications, a tool is provided to analyze which subset of the JVM instructions is used. The unused instructions can be omitted from the design. The core can be implemented in 1926 CLBs (= 3800 LCs) in a Virtex-II FPGA and runs at 20 MHz.

Komodo [15] is a multithreaded Java processor with a four-stage pipeline. It is intended as a basis for research on real-time scheduling on a multithreaded microcontroller. The unique feature of Komodo is the instruction fetch unit with four independent program counters and status flags for four threads. A priority manager is responsible for hardware real-time scheduling and can select a new thread after each bytecode instruction. Komodo's multi-threading is similar to hyper-threading in modern processors that are trying to hide latencies in instruction fetching. The fact that the pipeline clock is only a quarter of the system clock wastes a considerable amount of potential performance.

FemtoJava [7] is a research project to build an application specific Java processor. The bytecode usage of the embedded application is analyzed and a customized version of FemtoJava is generated (similar to LavaCORE) in order to minimize the resource usage. The resource usage is very high (about 2000 LCs), compared to the minimal Java subset implemented and the low performance of the processor.

JOP [21, 22] is a Java processor designed especially for embedded real-time systems. The main design goal was a time predictable processor. All hard to analyze processor features, such as prefetching or automatic stack dribbling as found in picoJava, have been avoided. To still provide acceptable performance a special stack cache and a WCET analyzable method cache have been developed. JOP consumes about 2000 LCs (depending on the configuration) and can be clocked at 100 MHz in an Altera Cyclone FPGA.

The jHISC project [25] proposes a high-level instruction set ar-

chitecture for Java. This project is closely related to picoJava. The processor consumes 15500 LCs in an FPGA and the maximum frequency in a Xilinx Virtex FPGA is 30 MHz. According to [25] the prototype can only run simple programs and the performance is estimated with a simulation. In [28] the clocks per instruction (CPI) values for jHISC are compared against picoJava and JOP. However, it is not explained with which application the CPI values are collected. We assume that the CPI values for picoJava and JOP are derived from the manual and do not include any effects of pipeline stalls or cache misses.

3. PICOJAVA ARCHITECTURE

Sun introduced the first version of picoJava [19] in 1997, targeted at the embedded systems market as a pure Java processor with restricted support of C. A redesign followed in 1999, known as picoJava-II; this is the version described below. After Sun decided to not produce picoJava in silicon, Sun licensed picoJava to Fujitsu, IBM, LG Semicon and NEC. However, these companies also did not produce a chip and Sun finally provided the full Verilog code under an open-source license.

Java bytecodes with low complexity are directly implemented in hardware, most of them execute in one to three cycles. Performance critical instructions with higher complexity, for instance invoking a method, are implemented in microcode. picoJava traps on the remaining complex instructions, and emulates this instruction. To access memory, internal registers and for cache management picoJava implements 115 extended instructions with 2-byte opcodes. These instructions are necessary to write system-level code to support the JVM.

Traps are generated on interrupts, exceptions and for instruction emulation. A trap is rather expensive and has a minimum overhead of 16 clock cycles:

```
6 clocks trap execution
n clocks trap code
2 clocks set VARS register
8 clocks return from trap
```

This minimum value can only be achieved if the trap table entry is in the data cache and the first instruction of the trap routine is in the instruction cache. The worst-case interrupt latency is 926 clock cycles [24].

Figure 1 shows the major function units of picoJava. The integer unit decodes and executes picoJava instructions. The instruction cache is direct-mapped, while the data cache is two-way set-associative, both with a line size of 16 bytes. The caches can be configured between 0 and 16 KB. An instruction buffer decouples the instruction cache from the decode unit. The FPU is organized as a microcode engine with a 32-bit data path supporting single- and double-precision operations. Most single-precision operations require four cycles. Double-precision operations require four times the number of cycles as single-precision operations. For low-cost designs, the FPU can be removed and the core traps on floating-point instructions to a software routine to emulate these instructions. picoJava provides a 64-entry stack cache as a register file. The core manages this register file as a circular buffer, with a pointer to the top of stack. The stack management unit automatically performs spill to and fill from the data cache to avoid overflow and underflow of the stack buffer. To provide this functionality the register file contains five memory ports. Computation needs two read ports and one write port, the concurrent spill and fill operations the two additional read and write ports. The processor core consists of following six pipeline stages:

Fetch: Fetch 8 bytes from the instruction cache or 4 bytes from the bus interface to the 16-byte-deep prefetch buffer.

Decode: Group and precode instructions (up to 7 bytes) from the prefetch buffer. Instruction folding is performed on up to four bytecodes.

Register: Read up to two operands from the register file (stack cache).

Execute: Execute simple instructions in one cycle or microcode for multi-cycle instructions.

Cache: Access the data cache.

Writeback: Write the result back into the register file.

The integer unit together with the stack unit provides a mechanism, called instruction folding, to speed up common code patterns found in stack architectures, as shown in Figure 2. The first step in instruction folding is the classification of operations in the instruction buffer into six groups. One group corresponds to a local variable load or a constant push (*LV*), three groups to different types of operations (*OP*, *BG1* and *BG2*), one group to a local variable store (*MEM*) and one group to an instruction that cannot be folded (*NF*).

A Java instruction

```
c = a + b;
```

translates to the following bytecodes:

```
iload_1
iload_2
iadd
istore_3
```

Figure 2: A common folding pattern that is executed in a single cycle

In patterns like *LV LV OP MEM* (the instructions in Figure 2 form such a pattern), *LV* and *MEM* resemble access to registers in a RISC architecture – the idea behind instruction folding is to detect such patterns and access the stack cache like a register file when possible. This eliminates the need for stack manipulation and allows up to four instructions to by executed in one RISC-style single cycle operation. Figure 3 shows how the instructions from Figure 2 are executed depending on whether folding is enabled or not.

picoJava contains a simple mechanism to speed-up the common case for monitor enter and exit. The lowest order bit of an object header is used to indicate the lock holding or a request to a lock held by another thread. This bit is examined by `monitorenter` and `monitorexit`. Hardware registers cache up to two locks held by a single thread.

To efficiently implement a generational or an incremental garbage collector picoJava offers hardware support for write barriers through memory segments. The hardware checks all stores of an object reference if this reference points to a different segment (compared to the store address). In this case, a trap is generated and the garbage collector can take the appropriate action. Additional two reserved bits in the object reference can be used for a write barrier trap.

The layout of class and method structures imposed by the hardware leaves much space for software dependent uses. In the two

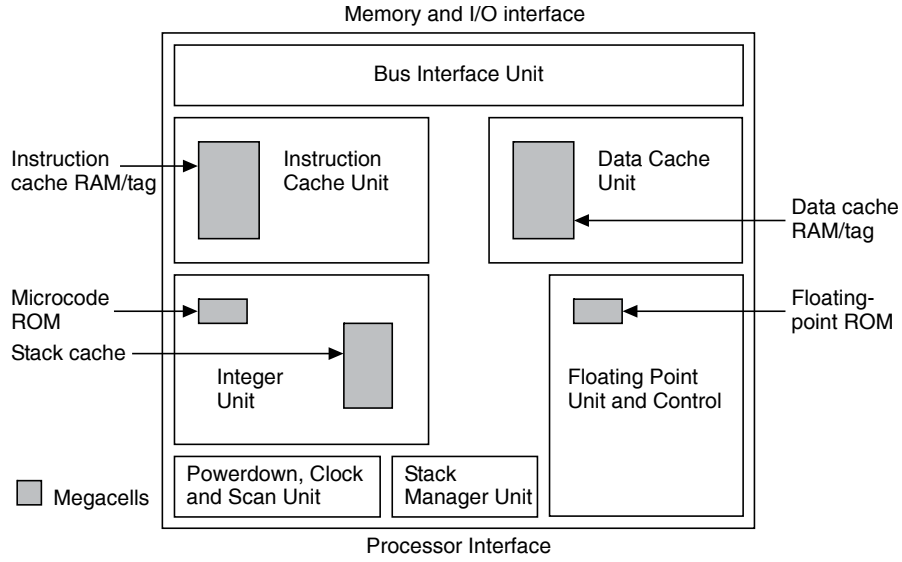

Figure 1: Block diagram of picoJava-II (from [23])

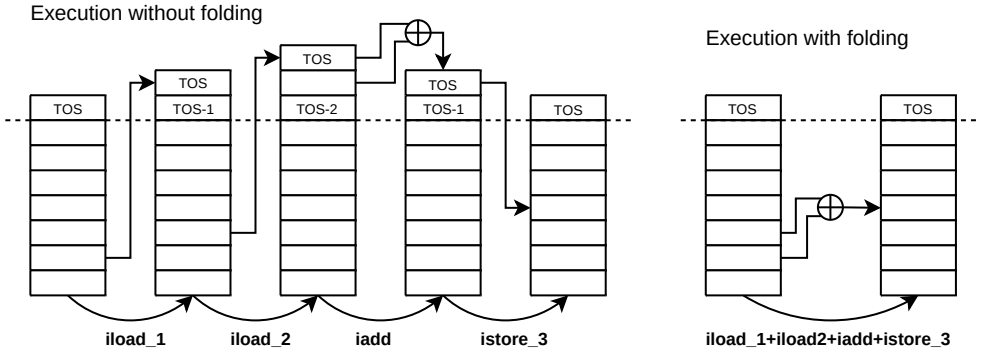

Figure 3: Execution of a common folding pattern

types of structures used for representing classes, 13 out of 18 words are unused by the hardware; a method uses a structure consisting of 9 words, of which 4 are unused. Although this is an advantage in terms of flexibility, it can add a significant overhead to the memory consumption of programs. Compared to layouts used by other processors, picoJava does not only consume a lot of memory, it also uses more indirections. This is compensated by the use of "quick" instructions however, which cut short many indirections.

The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions [17] and is the most complex Java processor available. The processor can be implemented [10] in about 440 K gates (128 K for the logic and 314 K for the memory components: 284×80 bits microcode ROM, $2\times192\times64$ bits FPU ROM and 2×16 KB caches).

4. FPGA IMPLEMENTATION

The distribution of picoJava comprises the source code for the actual hardware, an instruction accurate simulator and extensive documentation. For the hardware to work properly, memory and I/O modules have to be implemented. The simulator is written in C and comes along with support software, such as a loader, code that implements the software traps and an assembler. The latter (written in Java) was ready to use, while the loader and the traps had to be modified in the course of the implementation (see Sections 4.4 and 4.5, respectively).

Other parts, such as a garbage collector and the standard class library, are not part of the distribution and have to be written in order to accomplish a compliant implementation of the JVM. The current implementation does not include a garbage collector and the class library was only implemented as far as it was necessary for conveying benchmarks.

Apart from the sources provided by Sun, JOP was a vital resource for the implementation. The floating point unit was not included, because it is not featured in all processors intended for comparison by default and similar resource consumption and performance gains from such a unit are expected for all processors.

4.1 Target Platform

The target board for the implementation was the DE2 board from Altera [3]. This features a Cyclone II (speed grade 6) FPGA with 33,216 LCs and 483,840 bits on-chip memory. Of the available resources on the board, the SRAM (512 KB, 10 ns access time), the UART and the LEDs were used.

Quartus II (Web Edition, version 7.1) [2], Altera's tool for hardware design was used for synthesizing and downloading the hardware. Mentorgraphic's ModelSim is available for download along with Quartus II and was used for simulation.

4.2 Hardware

As the first step towards the implementation of the picoJava processor, a project in Quartus II was started and all relevant files were added. The dependencies between these files could be easily concluded from the scripts for synthesizing the processor with Synopsys, which are part of the distribution. The caches and the FPU were defined to be non-existent in the according configuration file, according to the instructions provided in this file. Only one minor change was necessary (certain bits were assigned a value twice) to make the core compile.

4.2.1 Stack Cache

As the stack cache is an essential part of the processor, this unit was to be designed before all other hardware. The stack cache has two write ports and three read ports to support both instruction folding and dribbling at the same time. While no memory with this setup is readily available, the situation is worsened by the fact that the stack cache is specified to be implemented as asynchronous memory, which is not available in modern FPGAs. This means that writes – reads are not critical in this context – have to be triggered by the write enable signal themselves. A race condition is the inevitable consequence of this behavior, because the write enable signals both have to be available as edge (to avoid latches) and as level (to fulfill their purpose as enable signals) to the same flip-flop. Quartus-II did not recognize the appropriate timing relations in the first attempts to design the stack cache, and therefore did not even mention possible timing violations. Several possible implementations were tried (a separate clock for every memory word vs. one common clock, e.g.), until finally a design was found which fulfills all specified properties and is handled correctly by the design tool.

A drawback of the current implementation of the stack cache is that it is implemented with flip-flops instead of on-chip memory. This uses 6 K LCs and thus enlarges the design significantly. It also might slow down the whole processor due to the more complex placement and routing. Implementing the stack cache with time-multiplexed access to on-chip memory by using a multiple of the system clock has been suggested but not evaluated yet.

4.2.2 Memory and I/O

A processor cannot be reasonably evaluated without an interface to its environment, so the next step was to implement an interface for memory and I/O. The SimpCon interface which is used with JOP had the advantage that several modules were already available for use with the DE2 board from Altera and thus was chosen. The SimpCon interface itself was extended with a signal *sel_bytes* for byte- and halfword-wise access – this signal had already been considered, but did not make it to the specification of the interface, because JOP only features word-wise access. A signal *error* was also added to the specification to signal invalid use of a module to the processor.

The translation of the signals generated by the picoJava processor to the SimpCon interface was relatively simple. 16 byte burst accesses which occur on transactions that involve the caches are serialized; other accesses only have to map the two least significant bits of an address to an according mask for *sel_bytes* and set *rd* or *wr* to high. The mapping of modules to addresses is slightly more complex, because *rd_data*, *rdy_cnt* and *error* from the module least recently accessed have to be routed back to the processor. To ease the task of mapping addresses to the according modules, a simple xml schema was defined from which Verilog code for such an address mapping is generated automatically. While there was some overhead to create a tool to do so, this already proved to be useful when changing the configuration of modules, and it also might be useful for other processors using the SimpCon interface.

As a basic set of modules the following configuration was chosen:

- 8 KB on-chip memory as boot ROM for initializing the processor and loading a program to the main memory.
- An interface to the SRAM on the DE2 board, which was derived from the according module for JOP. The module was enhanced to handle 8 and 16 bit accesses.
- A UART for downloading programs and diagnosis. This module was already available from JOP.
- An interface to access the on-board LEDs for diagnosis.
- A Timer module for measuring execution times.

4.2.3 Instruction and Data Caches

To enhance the performance of the processor, an instruction and a data cache had to be implemented. While the implementation of the instruction cache along the specification in [24] worked from the beginning, the data cache was a little more complex due to the interleaved usage of the memory banks by the two cache sets. It also triggered a bug in the translation of memory transactions to the SimpCon interface, because the cache fill ordering had been disregarded in the initial version.

4.3 Booting

Figure 4 shows how booting and invoking the main method is done. The processor starts execution at address 0, where the boot ROM is located. The trap base register is set to the suiting value at the very start, and then the boot method is invoked. When this method returns, the code jumps to the main memory, where the trap base might be set up again and the main method is called. Execution ends in an endless loop to prevent random code from being executed. The grey-shaded areas in Figure 4 are generated by the loader, which is described in Section 4.4

The presented procedure might look overly complex, but it is well motivated. Invoking a method instead of jumping to some piece of code has the advantage that internal registers are set to reasonable values in a fast and convenient way. It would also not have been feasible to invoke the main method from the boot ROM: this would have meant to require either the method structure to be located at some fixed location (which limits flexibility) or that this information is passed separately. The latter would probably have caused more trouble than the additional jump which is executed now. The current practice is also convenient because the boot ROM and programs have an almost identical format.

The boot method itself enables the stack cache and initializes and enables the instruction and data caches. The code for doing this is available in [24]. It then enables instruction folding and reads a program from the UART and writes it to the main memory.

4.4 Loader

To both make the implementation simpler and speed up execution, it was decided to load all classes statically. A loader to do this is provided, but it is written in C and relies on being executed on a big-endian 32 bit processor and has only been tested on SPARC. On the other side JOP has a similar loader written in Java, which has also a less restrictive license. Having this in mind, it seemed more profitable in terms of portability and reusability to port the latter.

The loader first looks for all classes which can be reached from the main class and parses them. After that, instructions which have

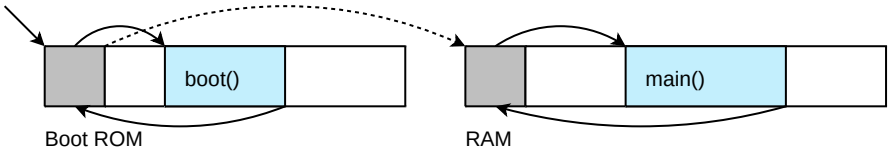

Figure 4: Schematic of bootstrap and execution

a "quick" counterpart are substituted and the symbolic references in the constant pool are resolved. Finally, the code to achieve the behavior described in Section 4.3 is emitted and the class information is dumped in a format that suits the representation picoJava stipulates.

In the course of the implementation of the loader, the BCEL [4] package which is used for parsing the class files had to be modified. picoJava does support extended bytecodes and these codes might appear in assembly code or code generated by the loader. BCEL must know about these instructions in order to handle them correctly.

4.5 Traps

Traps are an integral part of picoJava, because they implement parts of its instruction set. Object-oriented bytecodes, which have to resolve symbolic references from the constant pool cannot be implemented reasonably in hardware or microcode. While much overhead can be eliminated by replacing opcodes with their "quick" counterparts, some instructions are implemented as traps even in their "quick" version (`new_quick` is an example of such an instruction).

Implementations for the traps are provided by Sun, but these have to be modified. On the one hand there are some parts that are specific to the simulation environment, on the other hand the characteristics of the target system have to be taken into account (e.g. size of the available memory). Still, even with the rewritten loader, the original code was a good source for how the respective traps work.

The trap table is organized as 256 8-byte entries; the 4 bytes at the lower address provide the address of the trap code, the following 4 bytes can be used freely in principle, but are most useful for storing the address of the constant pool for the trap method. The table has to be aligned to a 2 KB boundary which can make a larger memory necessary than its actual size of 2 KB itself. Instructions that can cause a trap have their bytecode as index to the respective trap table entry. Entries of other instructions take care of illegal instructions, unaligned memory access and similar conditions.

The loader described in Section 4.4 provides means for registering methods as traps and takes care that the trap table is aligned correctly.

The only trap with a working implementation is at the moment `newarray`, which implies that only static programs (i.e. programs that do not create any objects) are supported. Future work will include writing the missing traps in order to provide full support for the Java programming language.

4.6 Timing Issues

The integer multiplication / division / remainder unit contains a combinatorial loop that has to be cut in order to get proper results from the timing analysis (there is no de facto data flow through the loop). The design constraints provided by Sun cut this loop and can be translated easily to constraints understood by the Time Quest timing analyzer that comes along with Quartus II.

Apart from the mentioned issues with the stack cache and the timing loop described above, no problems concerning the timing of the core were encountered. Problems with the adaptation of timing constraints mentioned in [10] did not arise.

5. EVALUATION

5.1 Logic Resource Usage

The implementation of picoJava in the final configuration (16 KB both data and instruction cache, no FPU) uses 27.5 K LCs. Of the units which had to be written, the stack cache is the biggest consumer, using more than 6 K LCs. The memory and I/O modules use 1 K LCs, including the bridge between picoJava and the SimpCon interface. The parts of the instruction and data caches to be written use less than 300 LCs together. For detailed numbers see Table 1. These numbers suggest that efforts for optimization would be spent best in finding a better solution for the stack cache than the current one.

Unit	Logic Cells
Data Cache Unit	1241
Data Cache RAM	192
Data Cache Tags	80
Instruction Cache Unit	3983
Instruction Cache RAM	0
Instruction Cache Tags	16
Execution Unit	6900
Hold Logic	8
Folding Unit	1036
Microcode Unit	2733
Pipeline Control	535
Register Control Unit (without Stack Cache)	3340
Stack Cache	6242
Trap Unit	117
Stack Management Unit	591
Powerdown, Clock and Scan Unit	0
Bus Interface Unit	28
Memory Map	216
Boot ROM	0
LEDs	45
SRAM Interface	164
Timer	138
UART	128
Interface picoJava/SimpCon	353
Total	27543

Table 1: LC usage of individual components

picoJava consumes between seven and thirteen times the number of LCs the designs presented Section 2.2 use. The only exception is jHISC, but even compared to this processor, picoJava uses 76% more LCs.

When comparing the number of LCs to the gate count reported in [10], 128 K gates for the logic, this yields a factor of 4.7. This

factor seems to be rather low but still plausible, taken into account that some logic functions are realized through memory blocks and thus do not add to the number of LCs.

5.2 Memory Consumption

47.6 KB of on-chip memory are used: 37 KB for the caches (actual cache and tag memory), 8 KB for the boot ROM and the remaining 2.6 KB for the implementation of special logic functions. Detailed numbers are provided in Table 2. There does not seem to be much potential for optimizations in this area, except for resizing the boot ROM (which at least has to be 4 KB in size to hold the trap table) or the caches, possibly affecting performance.

Unit	Memory Blocks	Bits
Data Cache	32	2×65536
Data Cache Tags	6	2×9728
Data Cache Status	5	2560
Instruction Cache	32	2×65536
Instruction Cache Tags	5	19456
Boot Memory	16	65536
Folding Unit	8	18432
Microcode Unit	3	624
Others	2	2048
Total	109	390256

Table 2: Memory usage of individual components

5.3 Speed

Although for comparisons, a frequency of 100 MHz is sometimes assumed for picoJava [19], this could not be met. Even highest optimization efforts could not push beyond around 43 MHz. For easy deduction from the provided clock frequency on the DE2 board with a PLL, 40 MHz were chosen as frequency for operation. The limiting factor in this case is the integer multiplication/division/remainder unit; the worst case timing path is located within this unit and is dominated by the interconnect delay (69% of the total delay). A different organization of the stack cache could as a side effect help in terms of speed by allowing for more efficient placement and routing – the effect of this is hardly predictable, but only minor enhancements are expected.

5.4 Performance

Two benchmarks from the benchmark suite used in [21] could be run on the picoJava processor at the current state of implementation: *Sieve*, a synthetic benchmark, and *Kfl*, an adaption of a real-time application. The benchmarks return how often per second the program can be executed; consequently, a higher number corresponds to better performance.

Figure 5 shows benchmark results (geometric mean of *Sieve* and *Kfl*, scaled such that the fastest configuration corresponds to 100) for different configurations of picoJava. In absolute numbers, the fastest configuration of picoJava returned 7797 iterations per second for *Sieve* and 23290 for *Kfl*. Both caches were configured to be 16 KB in size when present. The figure clearly shows that instruction folding is futile without the instruction cache. If the instruction cache is present, folding improves performance by 15 to 29%. Furthermore, the instruction cache has a bigger effect on the overall performance than the data cache – while the instruction cache speeds up execution by almost 90%, the data cache yields a speedup of 32% only compared to the version without caches.

Figure 6 compares benchmark results for different Java processors, scaled such that picoJava's performance corresponds to 100.

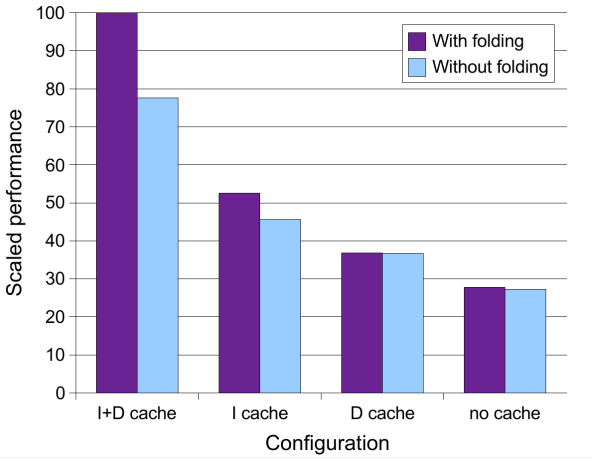

Figure 5: Bechmarks for different configurations of picoJava

*JOP** refers to the most recent version of JOP, while *JOP* refers to the version used in [21]. Apart from minor changes that do not concern performance, JOP* has more instructions implemented directly in hardware instead of microcode. The original results for Komodo, JStamp and SaJe are also found in [21].

The platform on which JOP was bench-marked is slightly different from the DE2 board; most importantly, a 32 bit SRAM with 15 ns access time was used instead of a 16 bit SRAM with 10 ns access time. Fortunately, these discrepancies equal out and both processors need two cycles for 32 bit memory transactions. The FPGAs show only minor differences, with the one picoJava runs on being faster by approximately 10%. This makes the results usable for comparisons of the processors themselves rather than comparisons of the overall systems, as it is the case with the other platforms.

The performance of JOP was measured at 100 MHz for both versions. SaJe features an aJ-100 processor running at 100 MHz, JStamp is a development platform containing an aJ-80 processor clocked at 74 MHz. The results for the Komodo processor were derived from cycle-accurate simulations using 33 MHz as clock frequency. picoJava runs, as already mentioned, at 40 MHz.

As shown by Figure 6, picoJava outperforms the processors it is compared against: it is more than ten times faster than Komodo and JStamp and about than 1.7 times faster than SaJe and the original version of JOP. Even the closest follower, JOP*, is almost 20% slower.

6. CONCLUSION AND OUTLOOK

In this paper we presented the implementation of the first industrial Java processor, Sun's picoJava, in an FPGA. To best of our knowledge this is the first successful attempt to implement the originally ASIC targeted design in an FPGA.

During the port to an FPGA we have encountered several solvable issues in the course of implementing the internal memories for the various caches. Results show that picoJava outperforms other Java processors, but that it is considerably larger too.

Future work will include completing the implementation of all trap functions and a bigger subset of the Java class library. With these prerequisites, more complex benchmarks will be possible and a thus better understanding of the architecture.

We provide a package named *Harvey* to adapt the original sources to the Altera DE2 board (see Appendix for download location and build instructions). We hope that the availability of a pico-

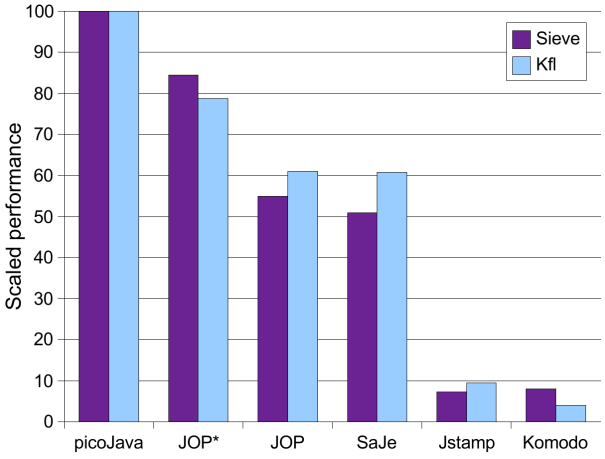

Figure 6: Bechmarks for different processors

Java implementation in an FPGA on a low-cost prototyping board is a basis where other researches can evaluate their ideas within an industrial strength Java processor. With a real implementation enhancements can be easily verified using a quantitative approach.

7. REFERENCES

[1] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[2] Altera. Quartus II Web Edition Software. Available at http://www.altera.com/support/software/sof-quartus.html.

[3] Altera. *DE2 Development and Education Board User Manual*, 2006.

[4] Apache Software Foundation. Byte Code Engineering Library. Available at http://jakarta.apache.org/bcel/.

[5] ARM. Jazelle technology: ARM acceleration technology for the Java platform. white paper, 2004.

[6] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[7] A. C. Beck and L. Carro. Low power java processor for embedded applications. In *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, December 2003.

[8] B. Bose, M. Tuna, and J. Nagy. LavaCORE™ configurable Java™ processor core. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 4, pages 4–1953–4–1959 vol.4, 2002.

[9] DCT. Lightfoot 32-bit Java processor core. data sheet, September 2001.

[10] S. Dey, D. Panigrahi, L. Chen, C. Taylor, K. Sekar, and P. Sanchez. Using a soft core in a SoC design: experiences with picoJava. *Design & Test of Computers, IEEE*, 17(3):60–71, July-Sept. 2000.

[11] T. R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.

[12] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.

[13] Imsys. ISAJ reference 2.0, January 2001.

[14] Imsys. Im1101c (the cjip) technical reference manual / v0.25, 2004.

[15] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[16] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[17] H. McGhan and M. O'Connor. Picojava: a direct execution engine for java bytecode. *Computer*, 31(10):22–30, Oct. 1998.

[18] Nazomi. JA 108 product brief. Available at http://www.nazomi.com.

[19] J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[20] PTSC. Ignite processor brochure, rev 1.0. Available at http://www.ptsc.com.

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

[22] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

[23] Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.

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

[25] 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-*, 153:20–30, 2006.

[26] Vulcan. Moon v1.0. data sheet, January 2000.

[27] Vulcan. Moon2 - 32 bit native Java technology-based processor. product folder, 2003.

[28] T. Yiyu, R. L. Lo Wan Yiu, Yau Chi Hang, and A. S. Fong. A java processor with hardware-support object-oriented instructions. *Microprocessors and Microsystems*, 30(8):469–479, 2006.

APPENDIX

A. BUILD INSTRUCTIONS

The original sources for picoJava are available at `www.sun.com/software/communitysource/processors/picojava.xml`; the package to adapt it to the DE2 board is available at `www.soc.tuwien.a.at/files/harvey/`.

Please note that the instructions given here might change in the future; latest build instructions will be provided in the file `README-HARVEY`.

Unpack the sources for picoJava, `pj2_rtl_2.0.tar.gz` and `pj2_sw_2.0.tar.gz`, to some directory of your choice. Unpack the archive for Harvey (named `harvey-de2-`*`date`*`.tar.gz`) to the same directory. Change to the directory `picoJava-II` and call `patch.sh` to apply the patches. The source code is now prepared for all other operations.

The easiest way to build picoJava is to simply type `make` in the directory `picoJava-II`. This creates the memory map (`make mmap`), creates the boot ROM (`make bootrom`), synthesizes the hardware (`make hardware`) and builds the software (`make software`). Please note that synthesizing the hardware might require some time; in the archive for Harvey there is a precompiled file that can be downloaded to the board (`quartus/picojava.sof`) which you may want to use.

For downloading the processor to the board, connect the appropriate USB port to your computer and call `make download`.

If only the boot ROM has changed, call `make bootrom update` to rebuild it and update the hardware (note that the target `update` does not download the hardware to the board).

To download software to the board, connect it to a serial port of your computer and configure this port with 57600 baud, 8 bit, 1 stop bit, no parity, no flow control. Pick a *.bin* file from the directory `sw` and send it to the serial port. You should see the LEDs of the DE2 board flash while downloading, and - if you use the files `Sieve.bin` or `Kfl.bin` without changing their sources - you should receive data via the serial port. The last 8 characters of output from these two programs represent the cycles used for conveying the respective benchmark.

It it also possible to use the graphical interface of Quartus II to build and download the hardware. The according project file is `quartus/picojava.qpf`. Simulations with ModelSim can be started from here by using the menu entry *Tools→EDA Simulation Tool→Run EDA Gate Level Simulation*.