A RISC Hardware Platform for Low Power Java

Paul Capewell and Ian Watson
School of Computer Science, The University of Manchester
Oxford Road, Manchester, M13 9PL, UK
{paul.capewell, iwatson}@cs.man.ac.uk

Abstract

Java is increasingly being used as a language and binary format for low power, embedded systems. Current software only approaches to Java execution do not always suit the type of resources available in many embedded systems. Hardware support for Java is a potential solution, reducing memory and power requirements while increasing execution speed. This paper presents a prototype architecture for hardware Java support within a RISC processor core, along with a synthesised asynchronous implementation. A breakdown of gate and silicon level simulation results quantifies where performance increases are achieved, providing a template for future work.

1. Introduction

Java [10] is a general purpose, object orientated programming language introduced by Sun Microsystems. Java programs are usually compiled down to 8 bit instructions (byte-codes), targeting a standard stack based virtual machine. Executables therefore have a very high code density [8, 3]. Resulting binary files can be interpreted on any platform which has a Java Virtual Machine (JVM) implementation. In terms of embedded systems, Java provides an ideal common binary distribution mechanism between a rapidly evolving set of target devices. Modern mobile phones are a successful example of the application of embedded Java.

The high memory requirements of a Java virtual machine, and associated performance penalty forms a stumbling block for embedded devices. Performance of JVM's also tends to get worse with lower memory implementations. Cut down embedded JIT (Just In Time) compilers exist, but still require more memory than is suitable for many devices. Simplified interpreter only virtual machines are far too slow when running on embedded architectures, such as ARM [4] or MIPS [6]. A hardware solution could provide the answer, and many have been designed. ARM's Jazelle [1] is an example of an efficient design. Jazelle integrates tightly with the RISC processor's execution pipeline and is transparent to the surrounding hardware in a device. External Java co-processors or even independent processors exist such as PicoJava [8, 11] but these are not necessarily compatible with the low-power and low component count requirements essential in the mobile/wireless appliance market.

The Java solution presented here has a similar structure to the ARM Jazelle architecture, in that the JVM is assisted by a Java byte-code to RISC instruction translator module. The module exists as a stage in an asynchronous processor pipeline. When in Java mode it interprets simple byte-codes directly in hardware with negligible interpretation time penalty, at the cost of extra logic gates. The implementation was designed to support the development of more advanced on-chip binary translation mechanisms, as well as exploring effective elastic pipelines.

Balsa [2], an asynchronous synthesis system, developed at The University of Manchester, has been employed to provide a route to silicon via handshake circuits [12]. An elastic asynchronous pipeline allows for common byte-code execution to be optimised, while allowing longer decoding cycles in more complex or infrequent cases.

2. A Java Aware Processor

Key aims in designing the Java processor architecture were: low power consumption, low gate count, efficiency and implementability. All but the last of these features are key to making such hardware both economical and suitable for use in embedded Java systems. Implementability is purely to allow progression of research into more novel translation schemes and future architectural improvement. Power is saved through the reduction of high power CPU cycles by generating efficient translated RISC code and reducing memory accesses, with minimal extra decoding logic.

Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05)
1063-9667/05 $20.00 © 2005 **IEEE**

2.1. Java Decoder Design

The Java hardware presented here fits into an ARM compatible processor and translates a subset of the Java binary instruction set (byte-codes) into native RISC code for execution further down the pipeline. Simple arithmetic instructions, operand stack and address calculations are translated entirely in hardware, while remaining byte-codes are handled by software handlers. The operand stack is cached in registers to allow execution on a load/store RISC architecture. The main Java interpretation loop, operates exclusively in hardware, reducing further the overhead of decoding and processing even un-handled byte-codes, compared to a software interpreter loop.

To make hardware for Java acceleration efficient and small it was decided to embed a Java unit into an existing asynchronous ARM compatible RISC processor, under development within the APT group in Manchester [9]. The unit acts as an extra pipeline stage in the processors operation when in Java mode, otherwise its existence is transparent, adding no latching or processing latency. The necessary instructions for jumping to a section of Java code and changing mode are defined by ARM for the Jazelle [1] family of cores.

2.2. Architecture

The Java decoder architecture is shown in Figure 1. This unit fits into the processor pipeline between the fetch and decode stages. The decoder takes single words fetched from memory as input and outputs RISC instructions for execution by the existing pipeline.

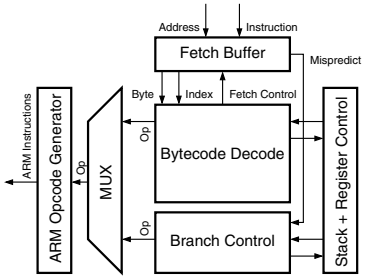

Figure 1. The Java Decoder Block.

Architecturally the Java unit inherits an asynchronous distributed control model, suiting simple integration with different asynchronous host cores, assuming that necessary information can be sourced. The unit needs a fetched memory address, to allow relative branch generation, along with the associated Java byte-code(s).

Once the processor has switched to Java mode (after a `bxj` instruction). Individual bytes are dispatched from a word buffer, filled by the processor's fetch unit, to the Java decode block. Often byte-codes require further arguments, the decode block requests bytes as needed. Each byte interpreted is treated as part of a Java binary stream. As byte-codes are recognised, appropriate RISC instructions are generated to perform the desired function. ARM instruction opcodes are then generated at the opcode generator, feeding the main processors execute unit. The component parts of the architecture are described in detail below:

2.2.1. Fetch Buffer The fetch buffer is present to latch a word from the fetch unit when in Java mode, and then dispatch individual byte-codes to the decode unit. The fetch buffer hides the complexities of instruction fetch from the rest of the Java decoder, such as the fact that multi-byte Java byte-codes may run over word boundaries. The other main job this unit does is to keep track of where in a word byte-codes were initially fetched from, allowing for branches and software calls. A software byte-code handler must be passed a valid return address. As byte-codes are (pre) fetched word at a time, more efficient use is made of the 32 bit memory bus.

2.2.2. Stack and Register Control Unit To implement efficient allocation of operand stack data to registers, a circular buffer strategy is used. This will be referred to as the stack cache. The state of the stack cache is maintained in this unit and is supplied to the instruction issuing units to provide correct register allocation. State is updated accordingly when a stack operation is requested by the decode or branch unit.

2.2.3. Byte-code Decoder This unit performs the task of an interpreter loop. It takes bytes in a Java instruction stream from the fetch buffer, and identifies instruction groups. If an instruction is not handled by the hardware, then a branch is taken to an appropriate software handler, otherwise an instruction sequence is generated to for execution by the host processor.

Once the byte has been classified the appropriate instruction sequence is generated, communication with the stack and register control unit allows for correct register indexes to be issued, correlating with the current stack cache state. Stack cache spill or fill operations may be required at this time, invoking appropriate memory load or store instructions.

2.2.4. Branch Control Unit The branch control unit is required to deal with the effects of pipelining around branch instructions. If a conditional branch is issued by the decoder unit then, depending on result, subsequent byte-codes may be executed in error. In the Java decoder, each (pre-fetched)

word can contain up to 4 byte-codes. Currently the stack cache is flushed when a conditional branch is issued, therefore when a branch is detected the stack cache is flushed correcting the state of the unit. Incorrectly issued instructions will be ignored by the execute unit. In the current architecture this is achieved through instruction colouring [5].

2.2.5. ARM Opcode Generator This unit translates from the internal opcode format to ARM instruction opcodes. When the instructions are dispatched, necessary extra data fields such as instruction colour are added and sent to the RISC processor units for decode and execution. Other RISC instruction sets could be targeted by redesigning this unit.

2.3. Register Allocation

Register(s)	Usage
R0–R3	Stack Cache
R4	Local Variable 0
R5	Points to handler routines
R6	Points to Java Stack
R7	Points to Java variables
R8	Points to Java constants
R9 – R11	For software JVM use
R12	For hardware JVM use
R13	Points to ARM stack
R14	Java link register
R15	Java program counter

Figure 2. ARM Register Allocation.

RISC instructions output by the decoder assume the ARM user mode register space and register allocation is based on the technique specified in the Jazelle white paper [1]. A allocation table is shown in Figure 2. The most important feature of this mapping is the four entry stack cache (R0 to R3). This system allows for single register to register RISC operations to replace single data processing Java byte-codes when when the operand(s) are cached (usually just one or two stack items are needed).

2.4. Integration into SPA

Figure 3 shows the Java aware processor architecture, known as JASPA (Java Aware Synthesisable Portable AMULET). SPA is a low speed synthesised asynchronous ARM core, developed in Manchester [9] for *secure* smart-card applications. The Java decoder module is shown taking the output of the fetch stage in the pipeline. The result gathered here is either forwarded directly to the RISC decode stage or latched for processing by the Java decoder, if the processor is in Java mode. Direct forwarding of instructions reduces branch latency (through pre-fetch depth) when executing RISC code, this is significant as the SPA does not currently perform any branch prediction.

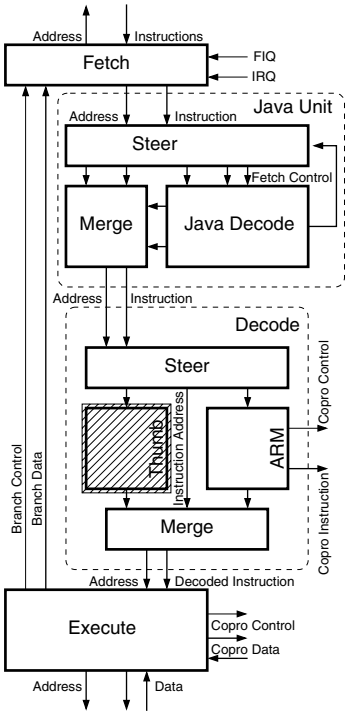

Figure 3. JASPA System Level Diagram.

The decoder is placed as an extra decode stage in the pipeline in order to leave a clean interface at the input and output. If the decoder was placed in parallel with the ARM and THUMB decoders shown, there would be less latency but the decoder would have then been sensitive to changes made within the SPA, which was in development concurrently with this design. This approach is also portable, as it has an easily customisable instruction interface at it's output. Asynchronous design with distributed control and uniform signalling protocols makes the integration task trivial. A synchronous version would have inherited timing constraints from other parts of the design, principally the maximum cycle time.

3. Prototype Implementation

Implementability was a key aim in the planning and design of the architecture. The realisation to silicon level for detailed simulation was the ultimate goal, fabrication would be too expensive for a proof of concept design. The use of Balsa enabled a rapid development cycle, helped by the SPA core being developed within the same design flow. Although the design is a simple prototype, with naive byte-code translation, it paves the way for future work and is open to expansion with new translation techniques.

The Balsa implementation of the Java decoder architecture was mainly a problem of organising communica-

tion and storage within the unit's different components. Asynchronous communication between blocks of hardware working in parallel must be explicitly synchronised through handshaking channels. In a clocked design all synchronisation takes place on the clock edge. Arbitration of shared resources must therefore be implemented carefully in a self-timed design as there is no globally shared timing reference. The only place in the JASPA design where this occurs is in the SPA fetch unit, when deciding if a branch/interrupt has taken place. A mutual exclusion element must be used at the arbitration point, and has several unfortunate timing properties [7]. This will not pose a problem in our core, due to our mutex design and the probabilities involved.

The design was partitioned as in the architectural description (Figure 1). Balsa allows, and almost forces one to think in a very modular manner facilitating decomposition of a design through communicating concurrent procedures. Re-use and future modification is natural, supported by well defined interfaces and abstract data types. Clarity and functional testability also result from this approach. Although channels can have abstract types in a Balsa description, underlying circuit styles and protocols can be changed at the synthesis stage.

3.1. Simulation and Functional Verification

To test the Java decoder implementation, functional level simulation was used. Balsa is extensible in that it allows for many circuit styles to be generated as output. All circuit types are derived from a graph of *handshake components*. This is a control and data-flow graph describing the high level functionality of a design, independent of how values are represented in a circuit. This behavioural level of representation is also ideal for rapid high-level simulation and functional verification. Issues such as deadlock and problems with concurrency were tackled using the simulator which is now part of the Balsa tool-set. Simulation at this level is around 6-10 times faster than un-extracted gate level verilog simulation.

JASPA was synthesised as a secure dual-rail circuit, as this was the technology used for the SPA chip. Dual-rail circuits are resistant to changes in wire delays, as each data bit signals its own arrival, hence automatic layout can be used. Balsa generates a gate-level verilog netlist for simulation, this was successfully utilised to verify the design. Although gate-level simulations do not give an accurate picture of silicon performance, they can be used to find problem areas in the design such as long critical paths of logic. Common verilog simulators provide a satisfactory level of performance, even when simulating benchmarks on the whole processor.

As far as circuit area is concerned the current design synthesises to 45,000 transistors in single-rail technology, and around 90,000 using a dual-rail technology mapping. These results are with a cell library developed within the APT group in Manchester targeting a ST Microelectronics 0.18 micron process technology.

3.2. Generating Silicon

In order to generate silicon, or a silicon level layout suitable for accurate power and timing simulation as well as hand off to a fabrication facility, a cell library and technology design kit are needed. Along with a front to back layout tool such as Cadence it is possible to achieve the necessary standard cell layout, placement, routing and extraction in order to generate and verify the design implementation. The only problems we encountered were with obtaining cell library details from the foundry, needed for accurate silicon layout level simulation.

The first issue to arise, when generating layout, is the efficiency of the generated circuits. Silicon area costs and speed of operation need to be determined. Typical elements used in the construction of most asynchronous circuits are mutexes and Muller-C elements, these are not present in commercial standard cell libraries. Although these elements can be constructed from standard gates, it is much better to add extra custom cells to increase performance, and reduce silicon area. The problem we faced was a limited cell library, without cell internals - needed to extract resistance and capacitance of circuit nodes. Good RC extraction is essential for realistic asynchronous circuit simulations in packages such as Nanosim or Spice. This level of simulation is needed for accurate power and speed estimations, this is *not* normally required for synchronous ASIC design as static timing analysis is usually sufficient. A full cell library was designed, within the APT group, for use with the 0.18 micron ST process.

4. Results

The following section presents a breakdown of where performance increases are achieved by the hardware, in comparison to a minimal threaded software interpreter. Such an embedded interpreter has similar memory requirements to the hardware solution, but with additional handler code for each byte-code handled in hardware by JASPA. This analysis is broken down into two sections: one dealing with a reduction in RISC execution cycles, and the second looking at benefits gained through the use of an elastic, self-timed, pipeline.

4.1. Behavioural Level

The following results, show the benefit of handling simple byte-codes entirely in hardware. Gains over software are

made in dispatching the correct handler routine for a byte-code and in stack/register management.

4.1.1. Handler Dispatch When interpreting Java, handler routines must be dispatched. In software, a lookup table would be used to find the appropriate function for a given byte-code. In hardware the same action occurs, but *only when the byte-code is not handled by the hardware decoder*.

Interpreter Code	JASPA Code
1 load bytecode [mem] update bytecode address	1 store stack cache state to register
	2 copy bytecode address to register
2 calculate handler lookup table index load handler address from table [mem]	3 calculate handler lookup table index load handler address from table [mem]
3 branch to handler routine	4 branch to handler routine

Figure 4. Handler Dispatch Comparison.

Figure 4 shows that when unhandled bytes are decoded in hardware, a similar penalty is payed in comparison to a software interpreter. The numbers at the left show how many RISC instruction cycles are needed in each case. In hardware, one memory access is saved at the expense of a constant store to a register, transferring stack cache state to the handler. Code sent from the hardware Java decoder bypasses instruction fetch, reducing congestion on the memory bus for the table lookup. Power will also be saved through this reduced activity in the self-timed design.

4.1.2. RISC Code Generation When executing byte-codes in hardware stack management is handled internally. A software interpreter must either manage such a cache with extra state maintaining code, or use a stack stored in memory. Further to this, RISC code dispatched by the Java decoder does not have to be fetched from memory, hence will not pollute the cache and reduces memory traffic.

Figure 5, shows comparative performance in terms of the number of RISC executions needed for a selection of byte-codes, ranging from the most efficient, to the most problematic (goto). The software routines use main memory for the operand stack, removing problems with state management. Instruction counts shown in black are for best case timings, while the grey bars indicate the worst case timing. The hardware timings often have poor worst case timings as there is the possibility of stack cache spill and fill. The only case where this is worse than software is for goto, when the stack cache must be flushed, this involves four memory stores.

In reality, worst case timings are very rarely incurred. Importantly, the Sun Java compiler tends to minimise operand stack depth for a given expression. In common examples this will fit in the stack cache of four registers, or will require few extra memory operations. As far as the goto byte-code is concerned, using Sun's compiler we have never experienced the need for a stack cache flush as it has always been emptied by preceding code.

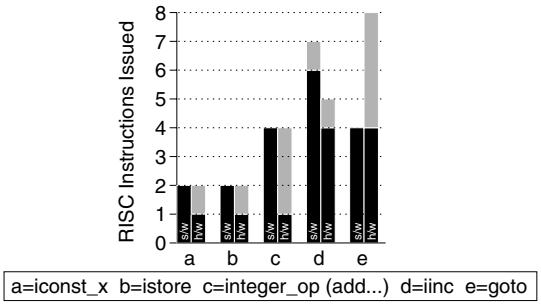

Figure 5. Code Generation Comparison.

A summary of overall performance, for the previous byte-codes, including the handler dispatch overhead is shown in Figure 6. In practise instruction sequences generated while executing simple arithmetic benchmarks resulted in typically a factor of 4 speed increase over interpretation. Including the effects of memory accesses a factor 7 improvement has been observed with some Java code.

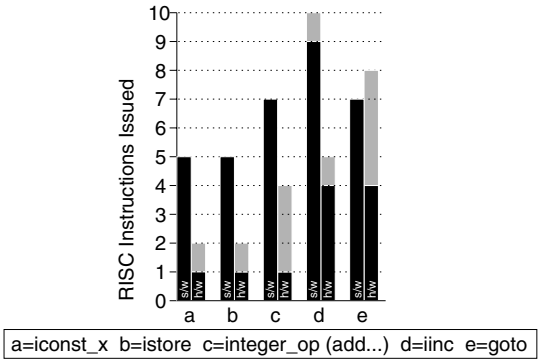

Figure 6. Cumulative Difference.

4.2. Asynchronous Pipeline

The elastic pipeline latency inherent in our self-timed design allows simple operations to complete faster than more complex ones. When little processing is required to translate a byte-code then the time taken to produce the associated RISC instructions will be reduced. The main example observed during simulation was the difference between operations requiring many operand stack operations

and those which do not. A difference is also experienced between when word buffer fetches occur, and when bytes are already available. Average case performance is achieved overall, as the best possible timing for each byte-code is achieved. Even with this simple scheme there is a big difference between fast and slow translations, this would make a synchronous design either more complicated or globally slower.

Simulation results for the self-timed Java decoder were run using Nanosim, on a Spice netlist extracted from layout data. The Java decoder was simulated in isolation to remove bottlenecks present in SPA and discover the absolute performance of the unit.

Averaged timings over 100 byte-code sequences showed a variation in latency between 30ns and 99ns per issued RISC instruction. Timing is apparently mainly dependant on the stack management requirements of a byte-code. Fetch latency seemingly has little impact, but was hard to test in isolation, and may have more effect when simulated with the SPA design. When RISC output was part of a sequence, latency hit the lower bound of 30ns, along with byte-codes such as `iconst_x` (34ns). Surprisingly, simple byte-codes such as `iadd` took up to 99ns as usually a single RISC operation was issued, but after 3 internal stack cache checks.

Unfortunately the Java decoder unit's absolute performance was very slow, considering the 0.18 micron process used. However, it is faster than the SPA execute unit, and would suit usage in a secure smart card environment. Part of the performance problem is related to the balanced secure circuit style used, and mostly down to the wholly non-hierarchical one pass place and route flow. No time was afforded for gate/layout level timing optimisation. Balsa also has much scope for circuit synthesis improvements. Importantly it was shown that with the self-timed Java decoder, simple byte-codes could produce RISC output 3 times faster than in more demanding cases.

5. Conclusion

This paper has presented a framework for a self-timed Java co-processor, designed to work with a RISC based host processor. An implementation has been described in terms of its main functional blocks and communications mechanisms, all the way down implementation in silicon using the freely available Balsa tool and commercial layout and simulation software. The conceptual simplicity of the structure has been highlighted as a key feature, and allows for future extensions (an optimising translation system) in the main byte-code decoder engine without requiring changes to the framework. The simplicity implies efficiency in terms of hardware and is complemented by flexibility of implementation style, as different self-timed circuit styles can be generated at synthesis time.

Performance has been shown to be around four times that of a software interpreter, with relatively small hardware costs. Problems with the implementation have been highlighted in terms of absolute speed, although important features of the architecture and design style have been demonstrated. Power benefits have been shown in terms of execution cycle reduction, although increased redundancy of the fetch unit and cache would contribute further to this saving. Recent work has since focused on the development of more efficient translation algorithms, simulated at the architectural level providing a further reduction in RISC execution cycles, through byte-code folding and more intelligent register allocation. It is expected that the elastic properties of the asynchronous decoder pipeline will yield further improvements when using more complex approaches.

Acknowledgements

The work presented here was funded through an EPSRC Ph.D studentship. This support is gratefully appreciated.

References

[1] ARM ltd. *Accelerating to Meet The Challenges of Embedded Java*, 2002.

[2] A. Bardsley. Balsa: An asynchronous circuit synthesis system. Master's thesis, Department of Computer Science, The University of Manchester, 1999.

[3] A. El-Mahdy, I. Watson, and G. Wright. Java virtual machine and integrated circuit architecture (JAMAICA) - choosing the instruction set. In V. Narayanan and M. L. Wolczko, editors, *Java Microarchitectures*. Kluwer, 2002.

[4] S. B. Furber. *ARM System-on-Chip Architecture*. Addison Wesley Longman, 2000.

[5] J. Gurd, C. Kirkham, and I. Watson. The manchester prototype dataflow computing system. In *Communications of the ACM*, volume 28, pages 34–52, Janurary 1985.

[6] G. Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., 1988.

[7] D. Kinniment and J. Woods. Synchronisation and arbitration circuits in digital systems. *Proc. IEE*, 123(10):961–966, October 1976.

[8] H. McGhan and M. O'Connor. PicoJava: A direct execution engine for Java bytecode. *IEEE Computer*, 31(10):22–30, Oct. 1998.

[9] L. A. Plana, P. A. Riocreux, W. J. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple. Spa-a synthesisable Amulet core for smartcard applications. In *Eighth International Symposium on Asynchronous Circuits and Systems*, pages 201–210, 2002.

[10] Sun Microsystems Computer Corporation. *The Java Language Specification*, 1995.

[11] A. Systems. Ajile web site. http://www.ajile.com/, 2002.

[12] K. van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Systems*. Cambridge University Press, 1994.