Tero Säntti | Juha Plosila

Internal Structure of an Enhanced Java Execution Engine

Turku Centre *for* Computer Science

TUCS Technical Report
No 665, February 2005

Internal Structure of an Enhanced Java Execution Engine

Tero Säntti
University of Turku, Department of Information Technology
Lemminkäisenkatu 14 A, 20520 Turku, Finland
`teansa@utu.fi`
Juha Plosila
University of Turku, Department of Information Technology
Lemminkäisenkatu 14 A, 20520 Turku, Finland
`juplos@utu.fi`

TUCS Technical Report
No 665, February 2005

Abstract

This report describes pipeline structure for a Java co-processor (from now on JPU). The pipeline structure is tailored with the peculiarities of Java bytecode streams in mind. Also the instruction set of bytecode is taken into account at the pipeline structure analysis. The JPU can be used in a single CPU and single co-processor environment or in a network of multiple CPUs and co-processors. The co-processor does not need to know what kind of environment it is placed in, as all communication goes through an interface unit designed especially for that environment. This modularity of the design makes the co-processor more reusable and allows system level scalability. This work is a part of a project focusing on design of an advanced Java co-processor for Java intensive SoC applications.

Keywords: Java, co-processor, pipeline, asynchronous

TUCS Laboratory
Communication Systems Laboratory

1 Introduction

Java is very popular and portable, as it is a write-once run-any-where language. This enables coders to develop portable software for any platform. Java code is first compiled into bytecode, which is then run on a Java Virtual Machine (hereafter JVM). The JVM acts as an interpreter from bytecode to native microcode, or more recently uses just in time compilation (JIT) to affect the same result a bit faster at the cost of memory. This software only approach is quite inefficient in terms of power consumption and execution time. These problems rise from the fact that executing one Java instruction requires several native instructions. Another source for inefficiency is the cache usage. As the JVM is the only part of software running natively, it occupies the instruction cache, whereas the Java bytecode is treated as data for the JVM, hence being located in the data cache. Also the actual data processed by the Java code is assigned to the data cache. This clearly causes more memory accesses missing the cache. When the execution of the bytecode is performed on a hardware co-processor this is avoided and the overall amount of memory accesses is reduced.

This work is a part of the REALJava [2] project, which aims to design a Java co-processor that is easily implemented to various systems. We have chosen to use asynchronous techniques in this project because then we can achieve good performance with reasonable power consumption and vary easy integration with existing systems, since no clock limitations need to be considered. Asynchronous self-timed circuit technology [5], where timing is based on local handshakes between circuit blocks instead of a global clock signal, provides a promising platform for obtaining a highly modular low-power and low-noise Java accelerator implementation.

Overview of the paper We proceed as follows. In Section 2 we shortly describe the structure of any JVM, and show how the proposed co-processor fits into the specifications. Section 3 describes the pipeline structure. In Section 4 the connections between pipeline stages are described, with the main functionality of the stages. Finally in Section 5 we draw some conclusions and describe the future efforts related to the REALJava co-processor.

2 Generic Java Virtual Machine Structure

In the Java Virtual Machine Specification [4], Second Edition the structure and behavior of all JVM's is specified at a quite abstract level. This specification can be met using several techniques. The usual solutions are software only, including some performance enhancing features, such as JIT (Just In Time Compilation). We have chosen to use a HW/SW combination [2] in order to maximize the hardware support and minimize the power consumption.

2.1 Partitioning

The HW portion (highlighted in Figure 1) handles most of the actual Java bytecode execution, whereas the SW portion takes care of memory management, class loading and native method calling. This partitioning gives the possibility to use the co-processor with any type of host CPU(s) and operating systems, as all of the platform dependent properties are implemented in software and (most of) the common bytecode execution is done in hardware.

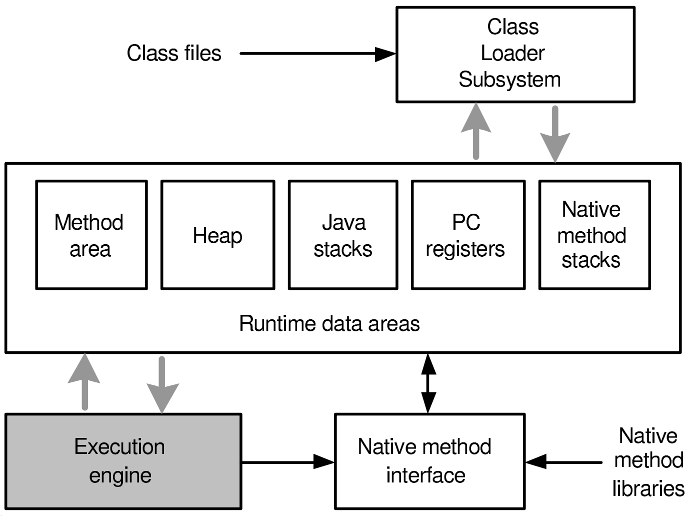

Figure 1: Internal architecture of the JVM

Because Java supports multithreading at language level, it makes sense to integrate several co-processors as a SoC. This gives an ideal solution for complex systems running several Java threads and possibly some native code at the same time. This approach brings forth true multithreading and thus improves performance. Also large systems possibly contain several software subsystems, such as internet protocols, user interface controllers and so on, these can easily be coded in Java, and since they all are executed in parallel the user experience is enhanced. The multithreading also improves the predictability of the real time performance, as the threads do not get any wait states and the caches and stacks for each thread are kept intact inside their respective JPUs.

The system architecture can be chosen to be a network of any kind or bus based, as suitable for other components in the system. The structure of the underlying network or bus is rather irrelevant, as long as the lower level provides two properties: 1) the datagrams must arrive in their destination in the same order that they were sent, and 2) the datagrams arriving from two different sources to a same destination must be identifiable. The first property can be be achieved with a lower level network protocol, like ATM adaptation layer (AAL) for internet, or by the physical structure of bus. The example we use here is a pipelined bus structure which guarantees the order of the datagrams by structure. The second property seems quite natural, and should be present in all solutions.

2.2 Bus selection for SoC environment

We chose to use the pipelined bus [3], since it provides a good platform for multiple processing units accessing the bus simultaneously. The bus provides high throughput at the expense of increased latency in comparison to a conventional bus. These properties rise from the structure of bus. Figure 2 shows the internal 3-level pipelines in each transfer stage. Our example system has a bus with 32-bit data word and 4 control bits per datagram as a payload. The bus itself contains more information about destination and the sender. The sender's id is also passed on to the interface unit. A simple yet efficient protocol for this case is given in [6].

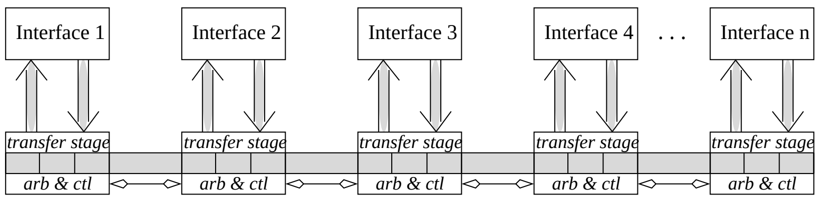

Figure 2: Detailed view of the pipelined bus with the interface units.

3 Pipeline Structure

The pipeline structure of the co-processor differs from the structure normally used for processors. This is due to the fact, that normally the instruction base of a processor is engineered with hardware implementation in mind, but this is not the case for Java. The Java bytecode is designed to be executed in software, resulting in several significant differences. The bytecode instructions are also based on (one) stack, instead of the normal processor approach of using several registers. This also calls for optimizations not seen in conventional processor design.

3.1 Conventional Processor Pipeline Architecture

The normal strategy for pipelining a general purpose processor involves 5 stages, namely:

1. instruction fetch
2. instruction decode / register access
3. execute / ALU
4. memory access
5. write back

This approach has been used in several processors and is also presented in textbooks, such as the DLX presented in [1]. This strategy is based on the assumption that the processor has internal registers for temporary or working data storage. Usually these registers can be accessed in parallel, and there usually are several registers available. As an example the DLX has 32 32-bit general-purpose registers Some processors also include separate registers for storing floating point numbers. The DLX provides 32 32-bit floating point registers, which can be used as even-odd pairs to hold 16 64-bit double-precision values.

3.2 The Modified Architecture for the Java Co-Processor

The Java Virtual Machine Specification states that the JVM has no internal registers, instead the temporary and working data is stored in a stack. Normally the coder can improve performance by ordering the accesses to the registers to keep the pipeline flowing, but in Java this is not possible, since all instructions that manipulate data are based on the stack. This situation is comparable with a normal processor architecture with only one register available to the programmer. This would keep the pipeline stalled for a large portion of the time, because of data dependency issues. To keep our pipeline in effective use, we have modified the normal strategy to better suit the stack based operation.

We also begin our structure with instruction fetching, we just use a fifo inside this unit to provide the folding unit with fast access to the instruction stream. The instruction decoder is the next unit. Since we are using instruction folding to minimize unnecessary stack access, the folding is also included in this stage. After that we have a intermediate buffer level to store the folded instructions before execution. This buffer also performs minor operations, such as extending data items to 32 bits, and generation addresses for local variable access.

The next stage performs data fetching, if necessary. Then comes the ALU, which contains the write back stage. The write back stage is included to the ALU because the bytecode instructions are based on the stack. One might wonder what this has to do with selecting the pipeline stages, but the answer is rather simple. In Java bytecode the instructions take the operands from the stack and write the result back to the stack. This causes the "normal" pipeline structure to generate huge amounts of wait-states to move the data to and from the stack. Thus the execution in the ALU will be often halted while the data is moved back and forth. Actually we will describe also more advanced methods to alleviate this problem, but those will be presented later in Chapter 4.2.

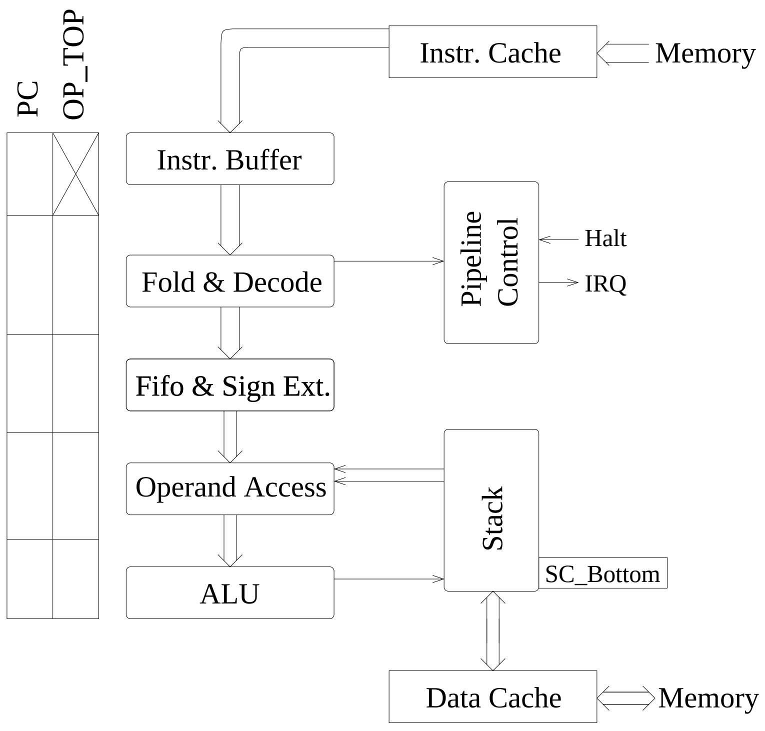

Figure 3: A Simplified view of the pipeline.

In the Figure 3 a simplified view of the pipeline structure is shown. The PC and OP_TOP labels stand for program counter and stack top, respectively. The boxes below those labels show how they are moved along the pipeline, to keep the values correct with the instructions related to them. The PC value is required for flow control commands (for example the conditional jump commands) and the OP_TOP is used for finding the correct addresses for data items. The pipeline control unit sends a halt command to all pipeline stages upon receiving an external halt command or a halt request from the fold and decode unit. The fold and decode unit is required to have halt access to facilitate pipeline halting when a software handled instruction is encountered. After the whole pipeline is idle, the pipeline control sends an IRQ to the host processor.

3.3 Shared Resources

Several pipeline stages need to access certain shared resources. These include the stack, the control registers and the program counter. The access to these resources is controlled by similar handshakes as the data flow through the pipeline. The main difference is, that since several units need to access these resources, we must provide some mechanisms to prevent simultaneous accesses and to guarantee the correct ordering of events.

The pipeline control unit can also be seen as a shared resource, connected to most of the pipeline stages. Note that the control needs not to be connected to all of the pipeline stages, since for instance the instruction fifo is passive on both direction, and thus remains idle if the previous and the next stage remain idle. This helps us in many ways, such as a slightly simpler controller, and a faster fifo structure, due to the fact that the fifo is “free-floating”.

4 Stages in More Detail

This chapter gives a more detailed view of the pipeline stages and the communication between them.

4.1 Instruction Fetching, Decoding and Buffering

This section starts after the instruction cache. The cache handles all memory accessing, so the instruction buffer needs only to access the cache. The address is generated at the instruction buffer. The buffer is active in communication towards the cache and passive towards the folding and decoding unit. The folding unit is active in both directions, towards the buffer and towards the decoded instruction fifo. The fifo performs sign extension on the data, if required. The fifo is passive in all directions, towards the folding unit and towards the register access unit.

The pipeline control unit is connected to the folding and decoding unit with two way communication. The folding unit needs to request a halt when it encounters an instruction to be handled in software. Of course the pipeline control unit must be able to stop the processing in this segment, so there needs to be bidirectional channel. The control unit also connects to all other pipeline stages, with a halt signal. The CPU can also request a halt, for thread switching or setting new values to internal registers.

The folding and decode unit has two communication channels to the instruction buffer. This is required because instructions may be followed by data, such

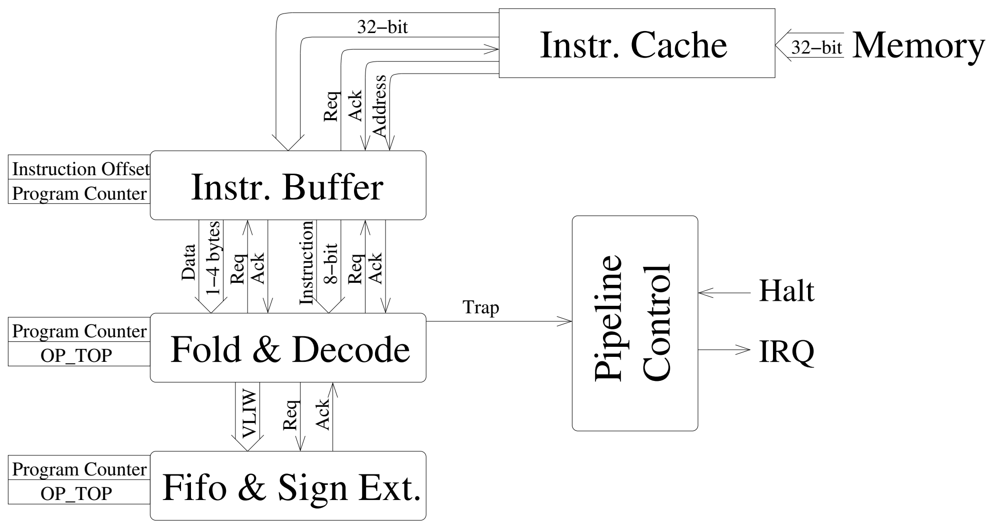

Figure 4: The instruction folding and decoding pipeline.

as literal operand or an address. The amount of data can be found out only by decoding the instruction first. After the decoding is completed, the correct amount of data bytes is read in parallel. The amount of data is between 0 and 4 bytes. If it is 0 bytes, no request is send to the data read port.

After the instruction has been decoded and the data related to that instruction is read in, the next instruction is checked to see if it can be folded with the previous one. If it can be, then the procedure is repeated to see if the third instruction can be folded. If at any point the instructions can not be folded together, the previous instructions are sent out, and the procedure starts over with the current instruction as a base for new foldings.

The fifo is only a few levels long, and both provides time marginal for folding and performs sign extension. Because the ALU is 32 bits wide, the sign extension is required for 16-bit and 8-bit literal data values. The time marginal for folding is increased because with asynchronous techniques all units exhibit average case performance. This means that the ALU may complete some instructions (bit-vice OR, etc.) in very short time, whereas some instructions (32-bit multiplication) take a lot more time. Since folding may produce new VLIW (Very Long Instruction Word) instructions at the rate of 1/1 to 1/4 in comparison to the original bytecode stream, the fifo balances the effects of both folding and the average case performance of the ALU. The fifo also generates actual addresses for local variable area accesses.

4.2 Operand access, ALU and Result Storing

The operand access unit takes care of providing the ALU with the actual operands, which may come from the local variable area, the stack or as a literal data from the bytecode stream. The operand access has two read channels to the top of the stack, one read channel to the local variable area and one by-pass channel to the end of the ALU. This by-pass channel reduces unnecessary traffic to and from the stack. This can be demonstrated with an example of an addition followed by a multiplication. In straight forward method the operations would be carried out as follows. First the addition is performed and the result stored to the stack, then the stack is read out to perform the multiplication. The result of the addition is consumed and does not remain in use. The improved method removes the consecutive write and read functions and replaces them with a straight connection from the result of the ALU to the operand access unit. This solution provides better performance in terms of execution time and power usage.

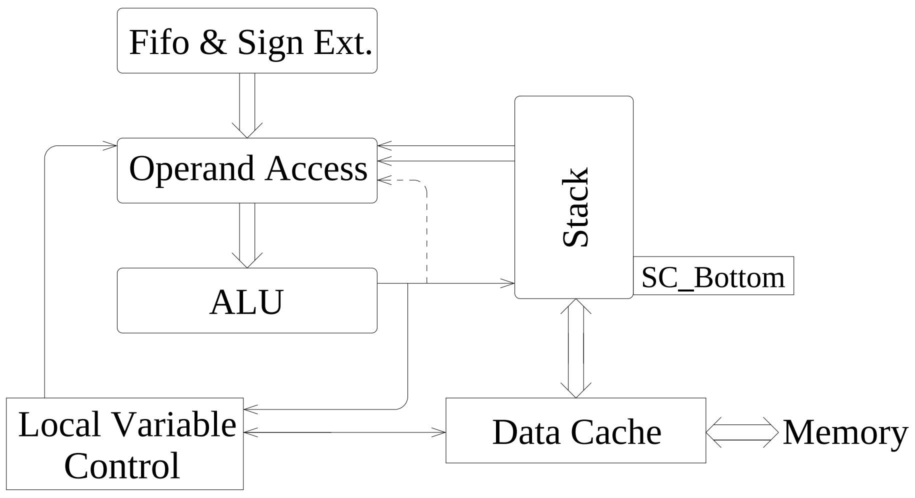

Figure 5: The execution pipeline and data transportation.

The Figure 5 shows the data connections in the execution part of the pipeline. The request and acknowledge signals are not shown, in order to keep the figure readable. As stated before, the fifo is passive in both directions. The operand access is active towards the fifo and also towards the data sources (namely the local variable controller, both of the stack read ports and the result of the ALU. The operand access is also active towards the ALU. The ALU is active only towards the stack write port.

4.3 Caches, Stack and Registers

The JPU contains two caches, namely the data cache and the instruction cache. The instruction cache is (quite naturally) read-only, whereas the data cache can be written and read. The instruction cache is less complex also because it is connected to only one unit, namely the instruction buffer. The data cache, on the other hand, is connected to the stack and to the local variable control. Both caches are also giving state information to the pipeline control unit, to notify the controller when the current operations are finished.

The stack is implemented as a 64 words long ring buffer. The buffer holds the top of the stack. When the buffer is close to full, the bottom of the ring is rolled to the memory via data cache. Naturally if the buffer is close to being empty more data is retrieved from the memory. The stack performs these transactions automatically, and no direct commands are required. However a command for flushing the cache to the memory is required, since jumping to a method causes a new stack-frame to be initialized, with its own local variables etc.

The internal registers of the JPU are all addressable from the CPU. This is required in order to be able to configure the JPU in the beginning of the execution as well as during thread switching. Two of the registers are also copied along the pipeline, namely the program counter and the OP_TOP. The rest of the registers remain in a normal register bank. The registers are passive in all their communication, with the exception of the two registers traveling with the pipeline. Those registers copy the communication scheme from the pipeline to their own communication to keep the register versions and the related instructions coherent.

5 Conclusions and Future Work

A novel pipeline structure for executing Java threads natively was presented. The structure takes into account the peculiarities of the Java bytecode streams, and provides reasonable performance with low power usage. The proposed structure also includes some "tricks" to enhance the execution of bytecode, such as instruction folding and stack by-passing.

The approach chosen here is energy aware, even in comparison to running compiled C code on the CPU. This is achieved by using asynchronous methods. Asynchronous methods excel especially in situations where the workload of the processing unit is not constant. Synchronous systems waste a lot of power by clocking internal latches even when no processing is done. This type of energy wasting is not present in asynchronous system. Also the current consumption of asynchronous systems (usually) is more stable, resulting in lesser noise.

We are currently investigating the potential benefits of integrating hardware timers and expanding the JVM with own functions designed to make use of the timers. These functions would make it possible for users to execute pieces of code based on timer information. We are considering a set-up with 4 internal timers and 4 external timing connections. All of these would be configurable and they could be masked out, when not needed. This would bring the real time performance of Java applications to a new level.

We plan to continue with designing the REALJava co-processor. The co-processor concept and hardware-software co-operation will be verified by building a FPGA demonstrator. After the FPGA phase we will continue manufacturing the co-processor as a separate ASIC. Later a larger NoC system with several CPUs and JPUs will be designed to implement a real-life application.

References

[1] J. Hennessy and D. Patterson. “Computer Architecture: a Quantitative Approach”, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[2] Z. Liang, J. Plosila, and K. Sere. “Asynchronous Java Accelerator for Embedded Java Virtual Machine”, *In Proc. of IEEE CAS Symposium on Emerging Technologies, Frontiers of Mobile and Wireless Communication*, Shanghai, China, June 2004.

[3] P. Liljeberg, J. Plosila, and J. Isoaho. “Self-Timed Communication Platform for Implementing High-Performance Systems-on-Chip”, *the VLSI Integration* Journal 38, Elsevier, 2004.

[4] T. Lindholm and F. Yellin. “The Java Virtual Machine Specification”, Second Edition, Addison-Wesley, 1997.

[5] J. Sparso and S. Furber. ”Principles of Asynchronous Circuit Design - A System Perspective”, Kluwer Academic Publishers, 2001.

[6] T. Säntti and J. Plosila. “Communication Scheme for an Advanced Java Co-Processor”, *In Proc. Norchip 2004*, Oslo, Norway, November 2004.

ISBN 952-12-1505-4
ISSN 1239-1891