Using a Java Optimized Processor in a Real World Application

Martin Schoeberl

JOP.design,
Strausseng. 2-10/2/55, A-1050 Vienna, Austria
martin@jopdesign.com

Abstract *— Java, a popular programming language on desktop systems, is rarely used in embedded systems. Some features of Java, like thread support in the language, could greatly simplify development of embedded systems, but the common implementations of the JVM (Java Virtual Machine), as interpreter or just-in-time compiler, are not practical. This paper describes an alternative approach: JOP (a Java Optimized Processor) is a hardware implementation of the JVM with short and predictable execution time of most bytecodes. JOP is implemented as a configurable soft core in an FPGA. The experiences of the first application of JOP and the benefits from using an FPGA in an embedded distributed control system are described in the second part of this paper.*

1 Introduction

JOP, a Java Optimized Processor, is an implementation of the JVM targeted for small embedded systems. It shall help to increase the acceptance of Java for these systems.

During the research for my thesis on JOP I got the chance to use JOP in a real world application. The first use of a new architecture in a commercial project is risky, but it is the best test case to prove the feasibility of the processor.

Balfour Beatty Austria has developed a *Kippfahrleitung* to speed up loading and unloading of goods wagons. JOP is used to control up to 15 asynchronous motors.

The system consists of one processor board per motor and one master station. The sub systems communicate over a RS485 bus. The main challenge was to react to the sensors in real time and control the bus access of this distributed system.

2 Motivation for a Java Processor

Current software design practice for embedded systems is still archaic compared to software development for desktop systems. C and even Assembler is used on top of a small RTOS. The variety of embedded operating systems is large and this fragmentation of the market leads to high cost. Java [1] can be a way out of this dilemma.

2.1 Java for embedded systems

Java possesses language features not found in C: object-oriented, memory management with a garbage collector, implicit memory protection and threads. Memory management and threads are (besides device drivers) the main components of embedded operating systems. Finding these features in the language embedded systems can be programmed in Java without the need of an operating system.

Java on desktop systems comes with a large library. However, if Java is stripped down to the core components it has a very small memory footprint. And with careful programming (which is also necessary in C coded embedded systems) the garbage collector can be avoided. Java can be used even in hard real-time systems.

2.2 Implementation of Java

The definition of the language includes also the definition of the binary and the virtual machine [2] to execute these programs. The Java Virtual Machine (JVM) is a stack machine. The drawback of stack machines is that every operand must be explicitly loaded. Java can be implemented in several ways:

Interpreter: A simple solution with low memory requirements, but lacks in performance.

Just-in-time compilation: Very popular on desktop systems, but has two main disadvantages in embedded systems: A compiler is necessary on the target and due to compilation during runtime execution times are not predictable.

Batch compilation: Java can be compiled in advance to the native instruction set of the target. However, dynamic loading of classes is no longer possible (not a main concern in embedded systems).

Hardware implementation: A Java Processor with JVM bytecodes as native instruction set.

Implementing the JVM in hardware is challenging but the most attractive way for embedded systems.

3 JOP – a Java Optimized Processor

The goal of JOP is a simple and small processor optimized to execute Java bytecode. The processor core has to be small enough to fit in a mainstream FPGA such that it is possible to use Java for embedded applications. Knowing the processor core it should be easy to implement a real-time enabled JVM.

It's not the goal to execute every bytecode directly in hardware. This would lead to a very complex design. JOP is a mixed hardware/software implementation of the JVM:

- Simple instructions are executed directly (in one cycle)
- Medium complex instructions are executed in micro code
- Particular JVM instructions can also be coded in Java (e.g. floating point instructions)

The micro code instructions must be general enough to implement the sometimes very complex JVM instructions.

Since threads are an integrated part of the Java programming language we will see more applications using them. When the processor core is small it will make sense to integrate multiple JOPs in a single chip to enhance performance of multithreaded programs.

3.1 FPGA Implementation

An FPGA has two basic building blocks: logic elements and memory. A logic element (LE) consists of a 4-bit LUT (Look Up Table) and a flip-flop. Memory blocks (ESB) are usually small (e.g. 0.5 KB) with independent read and write ports of configurable size. With these constraints a stack machine is an attractive architecture in an FPGA:

- The stack can be implemented in internal memory, making it an implicit data cache.
- A register file in a RISC CPU needs two read ports and one write port for single cycle instructions. A stack needs only one read and one write port (common in current FPGAs).
- Instruction set is simpler and can be reduced to 8 bits.
- No data forwarding is necessary.

JOP is implemented in the low cost ACEX FPGA [3] from Altera. Memory blocks are used to implement the stack, store micro code and for caching of Java bytecode.

3.2 Micro Code

Java bytecodes, the instructions of the JVM, have a great variation in complexity. There are simple instructions like arithmetic and logic operations on the stack. But the semantics of instructions like *new* or *invokestatic* can result in class loading and verification. As a consequence not every JVM instruction can be implemented in hardware. One solution, used in Suns picoJava-II [4], is to execute a subset of the bytecode native and trap on the more complex ones. This solution has a constant overhead for the software trap.

The approach to this problem in JOP is different. JOP has its own instruction set (the so called micro code). Some bytecodes have a 1 to 1 mapping to JOP instructions, for the more complex a sequence of JOP instructions is necessary.

Every bytecode is translated to an address in the micro code that implements the JVM. If the bytecode has an equivalent JOP instruction it is executed in one cycle and the next bytecode is translated. For complex bytecodes JOP just continues to execute micro code in the following cycles. The end of this sequence is coded in the instruction. This translation needs an extra pipeline stage but has zero overheads for complex JVM instructions.

3.3 Pipeline Overview

The stack architecture allows a short pipeline resulting in short branch delays. Figure 1 shows an overview of the pipeline. Three stages form the core of JOP, executing JOP instructions. An additional stage in the front of the core pipeline translates bytecodes to addresses in micro code.

Every JOP instruction (except wait) takes one cycle. Conditional branches have an implicit delay of two cycles. This branch delay can be filled with instructions or *nop*.

3.4 Java Bytecode Fetch

The first pipeline stage can be seen in Figure 2. All bytecodes are fetched from internal memory (*jbc ram*). This memory (instruction cache) is filled on function call and return. Every byte is translated through a table (*jtbl*) to an address for the micro code ROM. It is also stored in a register for later use as operand. *jinstr* is used to decode the type of a branch and *jpcbr* to calculate the target address.

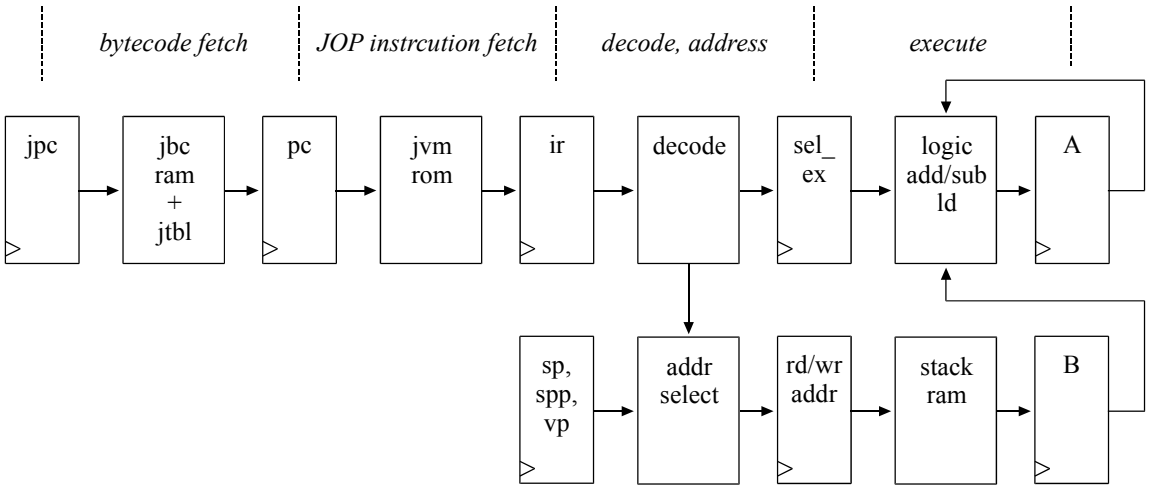

Figure 1: Pipeline of JOP

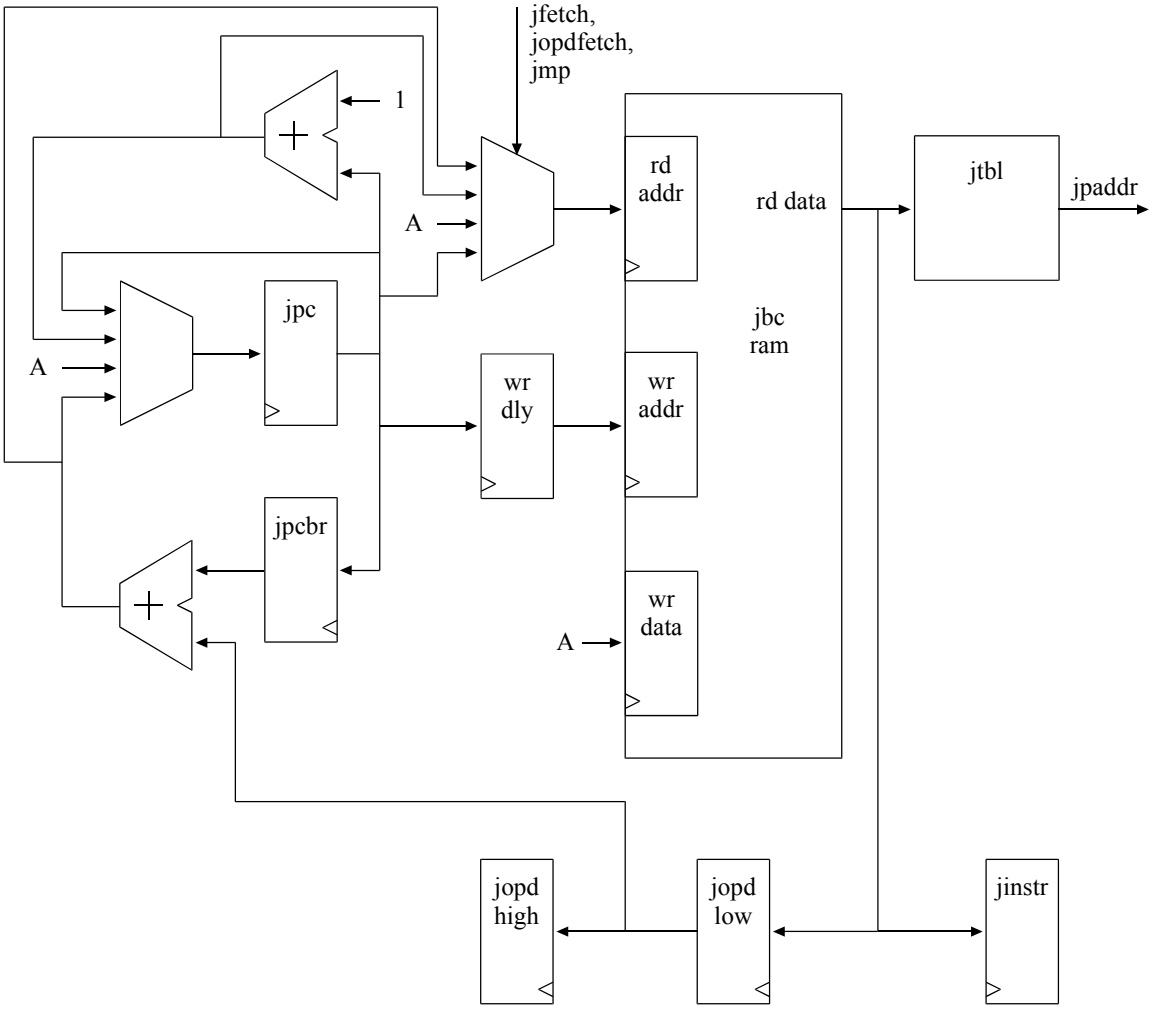

Figure 2: Java bytecode fetch

3.5 JOP Instruction Fetch

Figure 3 shows the second pipeline stage. JOP micro code that implements the JVM is stored in the memory labeled *jvm rom*. The program counter *pc* is incremented during normal execution. If the instruction is labeled to be the last one for the bytecode *pc* is loaded with *jpaddr* the starting address of the implementation of the next bytecode to be executed.

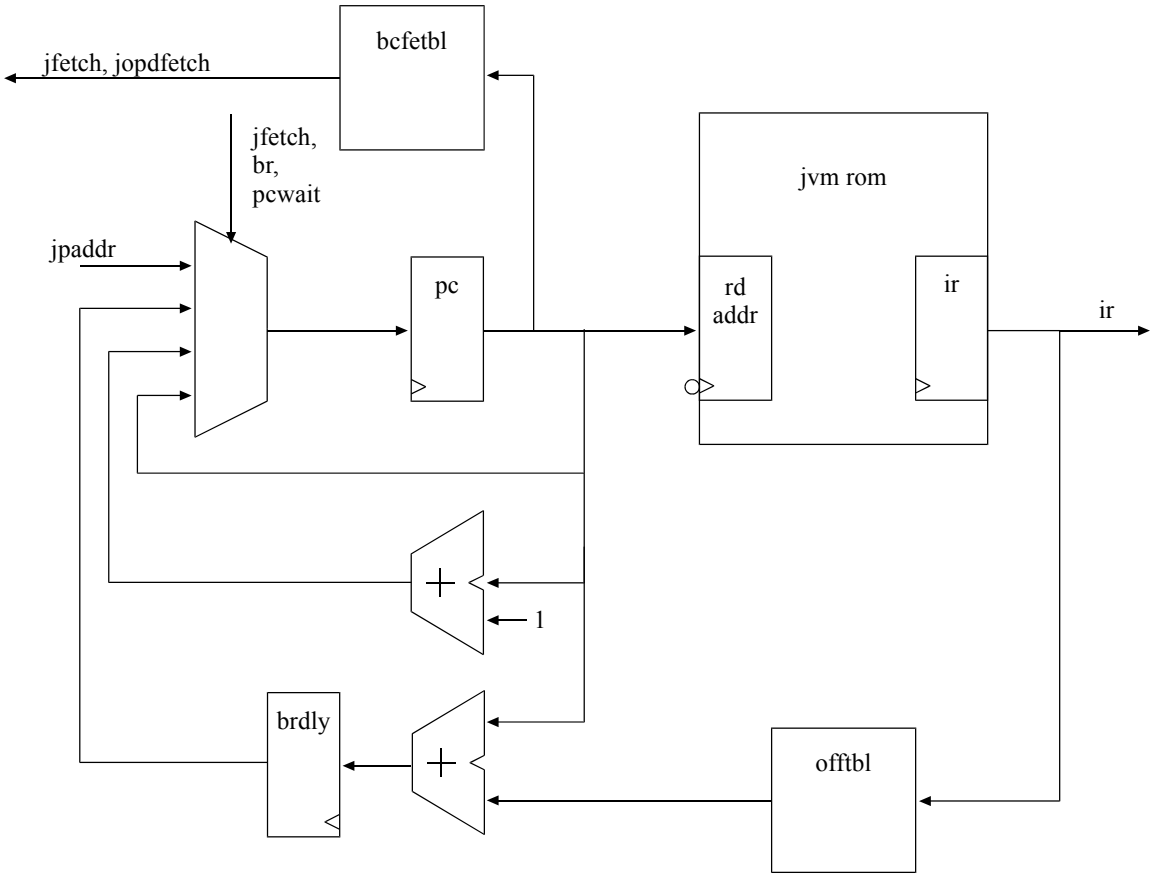

Figure 3: JOP instruction fetch

brdly holds the target for a conditional branch. A lot of branch destinations share the same offset. A table (*offtbl*) is used to store these offsets. This indirection makes it possible to use only 5 bits in the instruction coding for branch targets and still allow larger offsets. The three tables *bcfetbl*, *offtbl* and *jtbl* (from the bytecode fetch stage) are generated during assembly of the JVM code. The outputs are VHDL files. For an implementation in an FPGA it is no problem to recompile the design after changing the JVM implementation. For an ASIC with loadable JVM a different solution is necessary.

3.6 Decode and Address Generation

The third pipeline stage shown in Figure 4 is more complex than the first two. JOP instructions are decoded for the execution stage and addresses for read and write accesses of the stack RAM are generated.

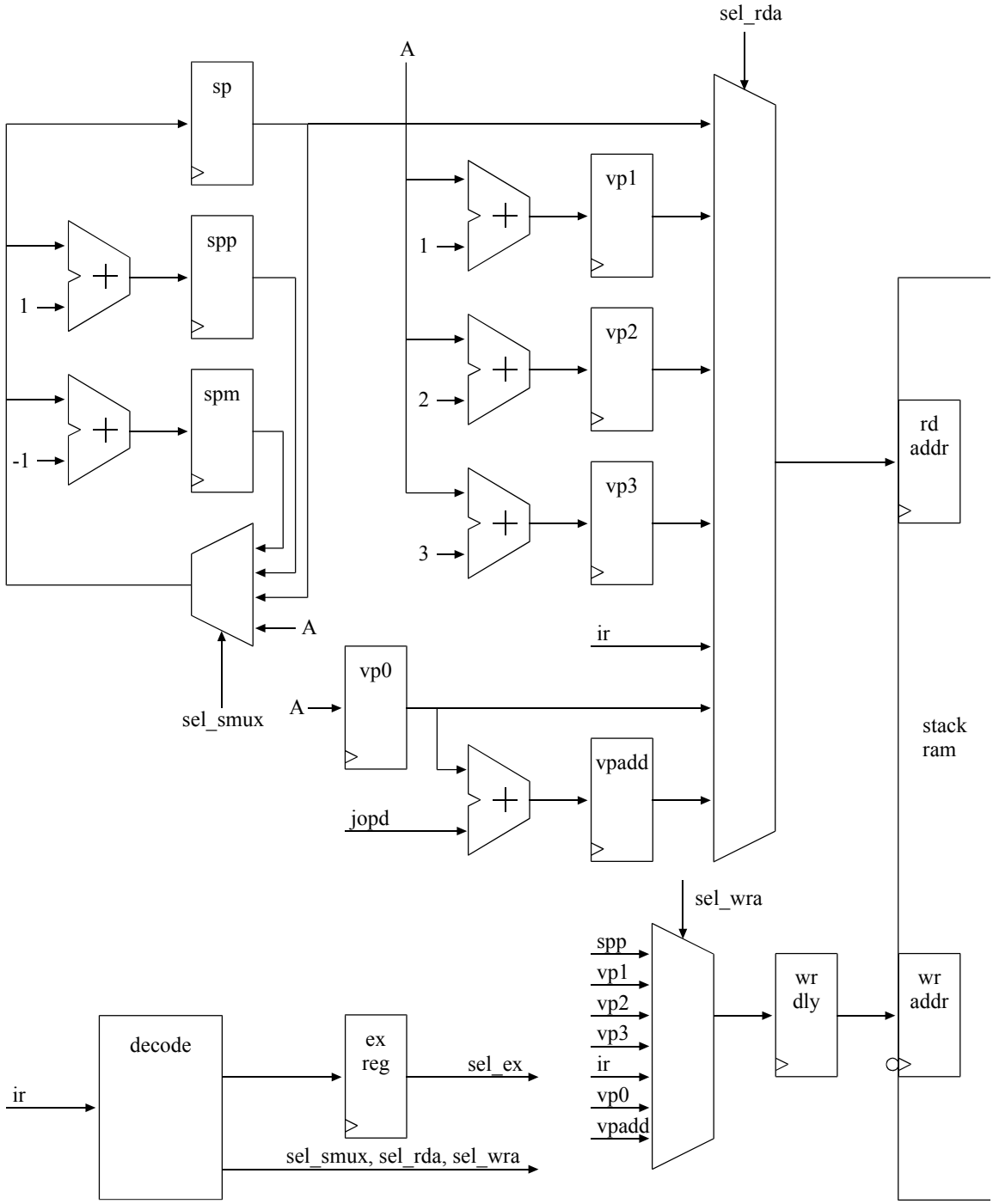

Figure 4: Decode and address generation

Instructions of a stack machine can be categorized with respect to stack manipulation in pop or push:

Pop instructions reduce the stack. Register *B* (TOS-1) from the execution stage is filled with a new word from stack RAM. The stack pointer is decremented. In short:

`A op B → A, stack[sp] → B, sp-1 → sp`

Push instructions generate a new element on the stack. Register *B* is spilled to stack RAM and the stack pointer is incremented:

```
data → A, A → B, B → stack[sp+1], sp+1 → sp
```

An instruction needs either read or write access to the stack RAM. Access to local variables, also residing in the stack, need simultaneous read and write access. As an example *ld0* loads the memory word pointed by *vp* on TOS:

```
stack[vp+0] → A, A → B, B → stack[sp+1], sp+1 → sp
```

There are two types of local variables. The locals of the JVM addressed with *vp* and internal scratch variables for the implementation of the JVM addressed with *ir*.

3.7 Execute

As can be seen in Figure 5 TOS and TOS-1 are implemented as register *A* and *B*. Every arithmetic/logical operation is performed with *A* and *B* as source and *A* as destination.

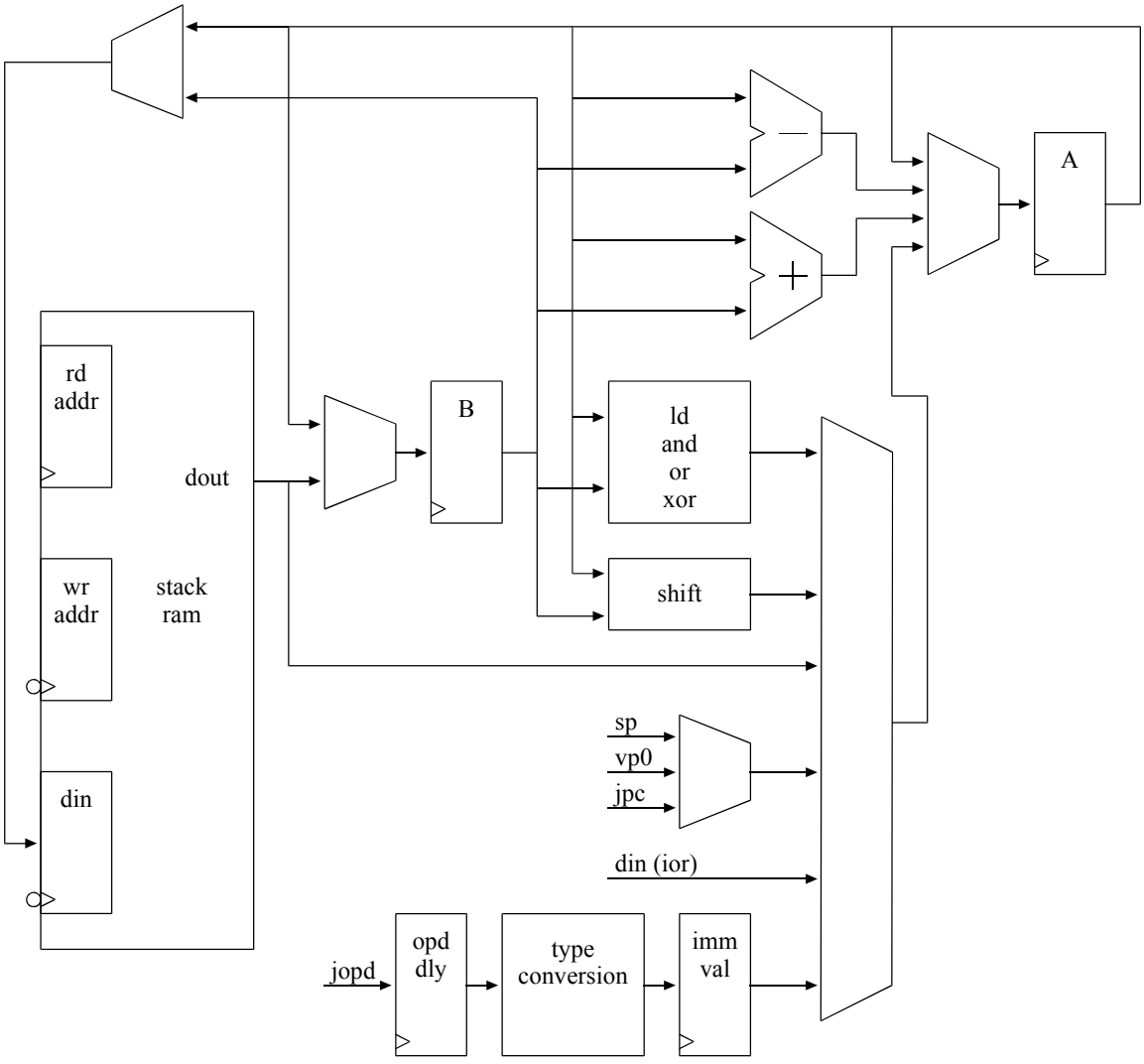

Figure 5: Execute

All load operations (local variables, internal register, external memory and periphery) result in the value loaded in *A*. As a consequence no write back pipeline stage is necessary. *A* is also the source for store operations.

Register *B* is never accessed directly. It is both read as implicit operand or for stack spill on push instructions and written during stack spill and fill.

3.8 Different Soft Core CPUs in FPGA

Table 1 compares the resource usage of different soft-core processors. Estimating equivalent gate counts for designs in an FPGA is problematic. It is better to compare the two basic structures LE (logic element) and ESB (embedded system block).

Nios: [5] Altera's configurable load/store RISC processor.

- Data path of 16 or 32 bits
- 16-bit instructions set
- 5-stage pipeline architecture
- Single clock shifts configurable

SPEAR: [6] Scalable Processor for Embedded Applications in Real-time Environments.

- 16-bit instructions set with deterministic execution times
- 3-stage pipeline architecture
- Predicated instructions to support single-path programming

Lightfoot: [7] Java Processor core from Xilinx.

- 3-stage pipeline architecture
- Stack-based design

JOP: Configuration for JOP:

- Hardware multiplier
- Single cycle barrel shifter
- External memory interface (8 Bit)

Processor	LEs	ESB	Data Path	Configuration
Nios	1700	2.5 KB	32-bit	UART, timer
SPEAR	1700	8 KB	16-bit	basic processor
Lightfoot	3400	1 KB	32-bit	basic processor
JOP	2100	3 KB	32-bit	UART, timer

Table 1: FPGA Soft Core Processors

4 The Project

In rail cargo a large amount of time is spent on loading and unloading of goods wagons. The contact wire above the wagons is the main obstacle. Balfour Beatty Austria devel-

oped and patented a technical solution to tilt the contact wire up. This is done on a line up to one kilometer. An asynchrony motor on each mast is used for this tilting. But it has to be done synchronously on the whole line.

Every motor is controlled by an embedded system. This system also measures the position and communicates with a base station. The base station has to control the deviation of individual positions during the tilt. It also includes the user interface for the operator.

Technically this is a distributed embedded real time control system communicating over a RS485 network.

4.1 Real Hardware

Although this system is not a mass product there were cost constraints. Even a small FPGA is more expensive than a general purpose CPU. To compensate for this, additional chips for memory and FPGA configuration were optimized for cost. One standard 128 KB Flash was used to hold FPGA configuration data, the Java program and a logbook. External main memory was reduced to 128 KB with 8-bit data bus.

To reduce external components the boot process is a little bit tricky: A watchdog circuit delivers a reset signal to a 32 macro-cell PLD. This PLD loads configuration data into the FPGA. When the FPGA starts it disables the PLD and loads the Java program from the Flash in the external RAM. After initialisation of the JVM the program starts at main().

The motor is controlled with five silicon switches connected to the FPGA with opto couplers. Position is measured with two end sensors and a revolving sensor. The processor supervises voltage and current. A display and a keyboard are attached on the base station for the user interface. The communication bus (up to one kilometer) is attached with an isolated RS485 data interface.

4.2 Synthesized Hardware

The following I/O modules were added to the JOP core in the FPGA:

- Timer
- UART for debugging
- UART with FIFO for the RS485 line
- Four sigma delta ADCs
- I/O ports

Five switches in the power line had to be controlled from the program. A wrong setting of the switches due to a software error could result in a short circuit. Ensuring that this could not happen was an easy task at the VHDL level.

4.3 Software Architecture

The main task of the program was to measure the position with the revolving sensor and communicate with the base station. This has to be done under real time constraints.

This is not a very complicated task. But at the time of development a lot of features from a full-blown JVM implementation, like threads or objects, were missing in JOP.The resulting Java was more like a ‘tiny Java’. It had to be kept in mind which Java constructs were supported by JOP. Due to the missing multi threading capability and for simplicity a

simple infinite loop with constant time intervals was used. After initialization and memory allocation this loop was entered and never exited.

```
public static void main(String[] args) {

      init();
      Timer.start();
      forever();
      // NEVER getting to this point
}

private static void forever() {

      for (;;) {
            Msg.loop();
            Triac.loop();
            if (Msg.available) {
                  handleMsg();
            } else {
                  chkMsgTimeout();
            }
            handleWatchDog();
            Timer.waitForNextInterval();
      }
}
```

Figure 6: Simplified Program Structure

4.4 Communication

The communication is based on a client server structure. Only the base station is allowed to send a request to one mast station. This station has to reply. The maximum reply time is bound by two time intervals. The base station handles timeout and retry. If an unrecoverable error happens, the base station switches off power for the mast stations including power supply for the motor. This is the fail safe state of the whole system.

On the other side every mast station supervises the base station. The base station has to send requests on a regular basis. If this is violated the mast station switches off the motor.

The data is exchanged in small packets of 4 bytes including one byte CRC. For development commands to program the Flash in the mast stations and to force a reset were included. So it is possible to update the program or even change the FPGA configuration over the network.

5 Benefits from using an FPGA

The flexibility of FPGAs made it possible to postbone some design decisions after production of the PCB. Since the production of the PBC was on the critical time line this helped to finish the complete project in time.

During development there have been situations where problems showed up that have not been foreseen. Two examples are given where it was possible to find simple solutions:

The routing of the PCB was almost finished. A question about cooling the switches (TRIAC) has arisen. The electronic development insisted on a temperature sensor. The requirements in resolution and conversion time were low. A sigma delta ADC with only

two external passive components (a NTC thermistor and a capacitor) was implemented in the FPGA.

The AC current of the motor had to be monitored. The solution for this was a shunt resistor in every power line and an opto coupler for isolation. But it turned out that the shunt resistors got too hot and delivered to little voltage for the opto couplers to work reliable. Having seen that it is possible to build an ADC in the FPGA a new idea was born. For EMC reasons there is an inductor in every AC line. With a few windings of wire a simple transformer can be built. The resulting voltage was amplified, rectified and used for current measurement. An additional comparator was used for an exacter threshold than the input buffer of the FPGA. This solution kept the board cool and added extra functionality. It is now possible to define two thresholds for too little and too much current.

5.1 A Sigma-Delta ADC

When the sample rate for an ADC is low compared to the clock frequency of the digital system it is possible to transfer the AD conversion problem from the analogue to the time domain. In Figure 7 the principle of a Sigma-Delta ADC is shown.

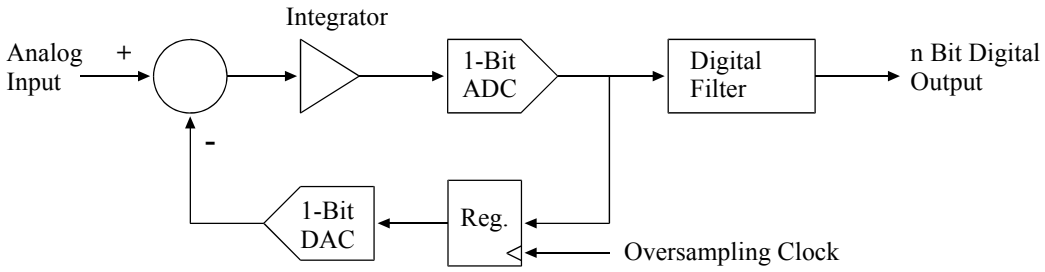

Figure 7: Sigma-Delta ADC

Only the adder and the integrator have to be analogue components. A single bit DAC is just the FPGA digital output driving the signal between GND and VCCIO. The single bit ADC can be built with a comparator. For low resolution the threshold of the FPGA input is practical. The simplest form of an ADC built with an FPGA is shown in Figure 8. The filter averages 2^n samples and can be implemented as a counter.

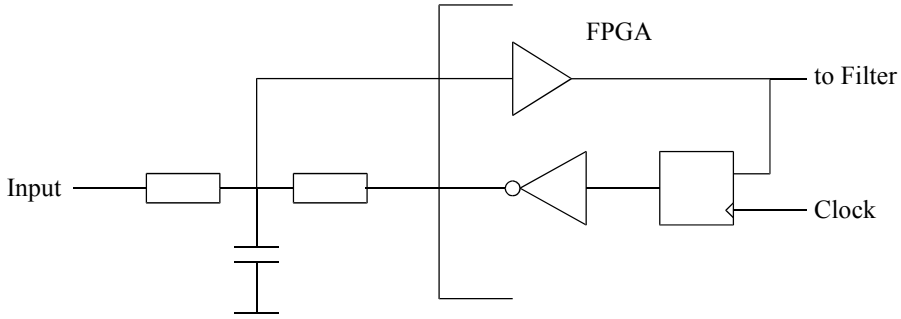

Figure 8: Minimal ADC

6 Tools

Tools have a major impact on productivity and on development cost. It was possible to use only free tools for software and FPGA development.

The Java programs were compiled with Sun's free Java compiler. The resulting class files had to be 'romized' with Suns JavaCodeCompact. JCC original produced C structures from the class files to be compiled and linked with a JVM written in C. JCC was adapted to generate binary class structures suitable for the JVM in JOP.

JOP is written entirely in VHDL. These VHDL sources were synthesized with an Altera OEM version of Leonardo Spectrum. The generated EDIF file was placed and routed in Alteras free version of MAX+Plus II. MAX+Plus II was also used for post routing timing simulation. No simulation on the VHDL level was necessary.

Additional tools were written in Java: Jopa an assembler for JOP micro instructions and JopSim a simulator of JOP at the JVM level (a JVM written in Java).

7 Conclusion

This paper showed one way to use Java in embedded applications. JOP, as hardware implementation of the JVM, makes it possible to use Java even in a hard real-time system. Due to the small size of the processor it can be implemented in a low cost FPGA. For low volume systems the flexibility of an FPGA can be of more importance than the slightly higher cost compared to conventional processors.

Using an FPGA as basis for the processor in the project added flexibility, which is not possible with conventional processors. A real running system proved the feasibility of JOP. But the constrains from the project directed some design decisions. Further research will focus on real-time threads in JOP.

More Information and all VHDL and Java sources for JOP can be found at http://www.jopdesign.com.

Acknowledgments

I would like to thank Walter Wilhelm from EEG for making it possible to use JOP in this project.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*, Addison Wesley, 2nd edition, 1997.

[2] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Addison Wesley, 2nd edition, 1999.

[3] Altera Corporation. *ACEX Programmable Logic Family*, Data Sheet, ver. 1.01, April 2000.

[4] Sun microsystems. *picoJava-II Processor Core*, Data Sheet, April 1999.

[5] Altera Corporation. *Nios Soft Core Embedded Processor*, Data Sheet, ver. 1, June 2000.

[6] M. Delvai, W. Huber, P. Puschner and A. Steininger. Processor Support for Temporal Predictability - The SPEAR Design Example. To appear at *Euromicro Conference on Real-Time Systems (ECRTS03)*, Porto, Portugal, July 2003.

[7] Xilinx Corporation. *Lightfoot 32-bit Java Processor Core*, Data Sheet, September 2001.