Java Technology in an FPGA

Martin Schoeberl

JOP.design, Vienna, Austria
martin@jopdesign.com

Abstract. The application of Field Programmable Gate Arrays (FPGA) has moved from simple glue logic to complete systems. The potential for FPGA use in embedded systems is steadily increasing continuously opening up new application areas. Low cost FPGA devices are available in logic densities where the CPU with necessary peripheral device can be integrated in a single device. Java, with its pragmatic approach to object orientation and enhancements over C, got very popular for desktop and server application development. Some features of Java, such as thread support in the language, could greatly simplify development of embedded systems. However, due to resource constraints in embedded systems, the common implementations of the Java Virtual Machine (JVM), 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. With JOP it is possible to develop applications in pure Java on resource constraint devices.

1 Architecture

JOP is the implementation of the Virtual Machine (JVM) [3] in hardware. JOP is intended for applications in embedded real-time systems and the primary implementation technology is in an FPGA, which results in the following design constraints:

- Every aspect of the architecture has to be time predictable
- Low worst-case execution time is favored over average execution speed
- The processor has to be small enough to fit in a low cost FPGA device

JOP is a full-pipelined architecture with single cycle execution of microinstructions and a novel approach to map Java bytecode to these microinstructions. Figure 1 shows the datapath of JOP. Three stages form the core of JOP, executing JOP microcode. An additional stage in the front of the core pipeline fetches Java bytecodes, the instructions of the JVM, and translates these bytecodes to addresses in microcode. The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches. The third pipeline stage performs, besides the usual decode function, address generation for the stack ram. Since every instruction of a stack machine has either *pop* or *push* characteristics, it is possible to generate the address for fill or spill for the *following* instruction in this stage. The last pipeline stage performs ALU operations, load, store and stack spill or fill.

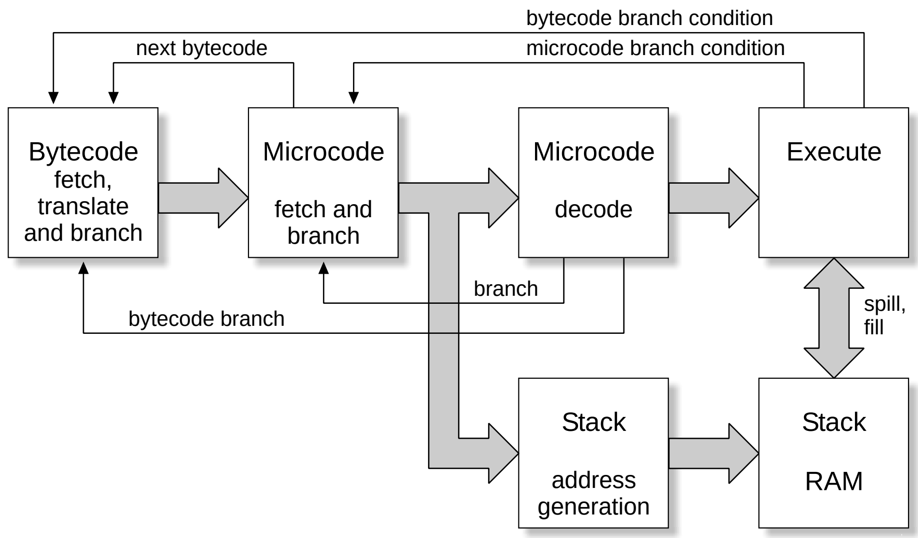
bytecode branch condition
next bytecode
microcode branch condition
Bytecode fetch, translate and branch
Microcode fetch and branch
Microcode decode
Execute
branch
bytecode branch
spill, fill
Stack address generation
Stack RAM

Figure 1. Datapath of JOP

Memory blocks in an FPGA are usually small (e.g. 0.5 KB) with two independent read/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
- 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
- Instruction set is simpler and instruction coding can be reduced to 8 bit
- No data forwarding is necessary

The basic stack is implemented in a FPGA memory block. The two top elements of the stack are implemented as register *A* and *B*. Every arithmetic/logical operation is performed with *A* and *B* as source and *A* as destination. All load operations (local variables, internal register and memory) result in the value loaded in *A*. Therefore no write back pipeline stage is necessary. *A* is also the source for store operations. Register *B* is never accessed directly. It is read as implicit operand or for stack spill on push instructions and written during stack spill and fill. 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:

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

2 Microcode

There is a great variation in complexity of Java bytecodes, the instructions of the JVM. There are simple instructions like arithmetic and logic operations on the stack. However, the semantic of bytecodes like *new* or *invoke* are too complex for hardware implementation. These bytecodes have to be implemented in a subroutine. One common solution, used in Suns picoJava-II [5], is to execute a subset of the bytecode native and trap on the more complex ones. This solution has an overhead (a minimum of 16 clock cycles in picoJava) for the software trap.

A different approach is used in JOP. JOP has its own instruction set (the so called microcode). Every bytecode is translated to an address in the microcode that implements the JVM. If the bytecode has a 1 to 1 mapping with a JOP instruction, it is executed in one cycle and the next bytecode is fetched and translated. For more complex bytecodes, JOP just continues to execute microcode in the following cycles. At the end of this instruction sequence the next bytecode is requested. This translation needs an extra pipeline stage but has zero overheads for complex JVM instructions. Figure 2 shows an example of this indirection. The fetched bytecode is used as an index into the jump table. The jump table contains the start addresses of the JVM implementation in microcode. This address is loaded into the JOP program counter for every executed bytecode.

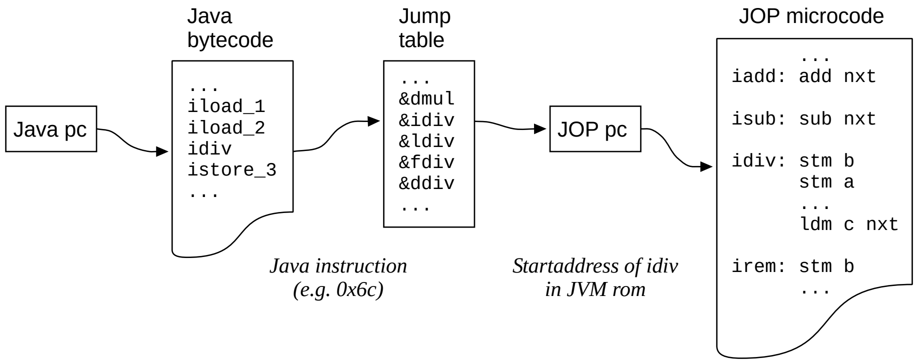

Figure 2. Data flow for a bytecode instruction

The example in Figure 3 shows the implementation of single cycle bytecodes and a bytecode as a sequence of JOP instructions. In this example, *ineg* takes 4 cycles to execute and after the last instruction (add) for *ineg*, the first instruction for the next bytecode is executed. The microcode is translated with an assembler to a memory initialization file, which is downloaded during FPGA configuration.

```
iadd:    add nxt    // 1 to 1 mapping
isub:    sub nxt
ineg:    ldi -1     // there is no -val
         xor        // function in the
         ldi 1      // ALU
         add nxt    // fetch next bc
```

Figure 3. Implementation of iadd, isub and ineg

3 HW/SW Co-Design

Using a hardware description language and loading the design in an FPGA, the traditional strict border between hardware and software gets blurred. Is configuring an FPGA not more like loading a program for execution?

This looser distinction makes it possible to move functions easily between hardware and software resulting in a highly configurable design. If speed is an issue, more functions are realized in hardware. If cost is the primary concern these functions are moved to software and a smaller FPGA can be used. Let us examine these possibilities on a relatively expensive function: multiplication. In Java bytecode *imul* performs a 32 bit signed multiplication with a 32 bit result. There are no exceptions on overflow.

Since single cycle multiplications for 32 bits are far beyond the possibilities of current FPGAs, we can implement *imul* with a sequential booth multiplier in VHDL. Three JOP instructions are used to access this function. If we run out of resources in the FPGA, we can move the function to microcode. The implementation of *imul* needs 73 JOP instructions and has an almost constant execution time. JOP microcode is stored in an embedded memory block of the FPGA. This is also a resource of the FPGA. We can move the code to external memory by implementing *imul* in Java bytecode. Bytecodes not implemented in microcode result in a static method call from a special class (com.jopdesign.sys.JVM). The class has prototypes for every bytecode ordered by the bytecode value. This allows us to find the right method by indexing the method table with the value of the bytecode. The additional overhead for this implementation is a call and return with the cache refills.

Table 1 lists the resource usage and execution time for the three implementations. Executions time is measured with both operands negative, the worst-case execution time for the software implementations. Only a few lines of code have to be changed to select one of the three implementations. The showed principle can also be applied to other expensive bytecodes like: *idiv*, *ishr*, *iushr* and *ishl*. As a result, the resource usage of JOP is highly configurable and can be selected for every application.

Table 1. Different implementations of imul

	Hardware [LC]	*Microcode [Byte]*	*Time [Cycle]*
VHDL	300	12	37
Microcode	0	73	750
Java	0	0	~2300

4 Results

Table 2 shows resource usage for different soft-core processors and different configurations of JOP implemented in an EP1C6 FPGA from Altera [1]. All configurations of JOP contain a memory interface to 32-bit static RAM and an 8-bit FLASH for the Java program and configuration data. The minimum configuration implements multiplication and the shift operations in microcode. In the core configuration, these operations are implemented as sequential Booth multiplier and a single-cycle barrel shifter. The typical configuration contains some useful I/O devices such as an UART and a timer with interrupt logic for multi threading. Lightfood [6] is a Java processor targeted at Xilinx FPGA architectures. As a reference NIOS [2], the RISC soft-core from Altera, is also included in the list. Version A is a minimum configuration. Version B adds an external memory interface, multiplication support and a timer.

Table 2. Different FPGA soft cores

Processor	*Resource [LC]*	*Memory [KB]*	*fmax [MHz]*
JOP Minimal	1238	3.25	101
JOP Core	1670	3.25	101
JOP Typical	2036	3.25	100
Lightfoot	3400	1	40
NIOS A	1828	6.2	120
NIOS B	2923	5.5	119

5 Conclusion

Java possesses language features as safety and object orientation that can greatly improve development of embedded systems. However, implementation as interpreter with a JIT-compiler are usually not practicable in resource constraint embedded systems. This paper presented the architecture of a hardware implementation of the JVM. The flexibility of FPGAs and HW/SW co-design makes it possible to adapt the resource usage of the processor for different applications. Predictable execution time of bytecodes enables usage of Java in real-time applications. JOP has been used in three real-world applications showing that it can compete with standard microcontrollers. JOP encourages usage of Java in embedded systems. Full source (VHDL and Java) of JOP can be found at [4]. The main features of JOP are summarized below:

- Small core that fits in a low cost FPGA
- Configurable resource usage through HW/SW co-design
- Predictable execution time of Java bytecodes
- Fast execution of Java bytecodes without JIT-Compiler
- Flexibility for embedded systems through FPGA implementation

References

[1] Altera Corporation. *Cyclone FPGA Family*, Data Sheet, ver. 1.2, April 2003.

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

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

[4] M. Schoeberl. *JOP - a Java Optimized Processor*, http://www.jopdesign.com.

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

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