Design Decisions for a Java Processor

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

Abstract

This paper describes design decisions for JOP, a Java Optimized Processor, implemented in an FPGA. FPGA density-price relationship makes it now possible to consider them not only for prototyping of processor designs but also as final implementation technology. However, using an FPGA as target platform for a processor different constraints influence the CPU architecture. Digital building blocks that map well in an ASIC can result in poor resource usage in an FPGA. Considering these constraints in the architecture can result in a tiny soft-core processor.

1 Introduction

Common design praxis for embedded systems is using an of the shelf microcontroller and programming in C on top of a small RTOS (real-time operating system). The approach of JOP, a Java Optimized Processor, differs in two ways:

- The processor is a soft core for an FPGA. FPGAs are still expensive compared to microcontroller. However, if the processor core is small enough to fit in a low-cost FPGA and leaving enough resources free for periphery components, chip count can be reduced and overall cost will be lower. Combining processor and periphery in one FPGA also adds flexibility, which is not possible with conventional processors (for an example see [1]). This can result in shorter development time further reducing cost for low to medium volume systems.
- Java is still seldom used as programming language for embedded systems although it offers features in the language (like object-oriented, threads and implicit memory protection) that ease application development. Threads and synchronization, as part of the language, can even result in systems without an underlying RTOS. The main disadvantage of Java on small processors is that the JVM (Java Virtual Machine) [2] has to be implemented as interpreter resulting in low performance. JOP implements the instruction set of the JVM (the so called bytecodes) in hardware, minimizing the performance gap between C and Java.

The paper is organized as follows: Section 2 gives an overview of the architecture of JOP. In section 3 some design decisions for the implementation in an FPGA are described. One example of the flexibility of an FPGA based architecture is given in section 4.

2 Overview of JOP

Figure 1 shows the datapath of JOP. In the first pipeline stage Java bytecodes, the instructions of the JVM, are fetched. These bytecodes are translated to addresses in the micro code. Bytecode branches are also decoded and executed in this stage. A fetched bytecode results in an absolute jump in the micro code (the second stage). If the bytecode has a 1 to 1 mapping with a JOP instruction, a new bytecode is fetched, resulting in a jump in the micro code in the next cycle. If the bytecode is a complex one JOP continues to execute micro code. At the end of this instruction sequence the next bytecode is requested.

The second pipeline stage fetches JOP instructions form the internal micro code memory and executes micro code 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 ether a *pop* or *push* characteristics it is possible to generate the address for fill or spill for the *following* instruction in this stage.

In the execution stage operations are performed with two discrete registers: TOS and TOS-1. Data between stack ram and TOS-1 is also moved (fill or spill) in this stage. A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill needs neither an extra write back stage nor any data forwarding.

Through full pipelining every JOP instruction takes one cycle. Two branch delay slots are available after a conditional branch.

3 Design Decisions

Every design is influenced by the available tools. The same is true for CPU architecture. The first and primary implementation of JOP is in an FPGA.

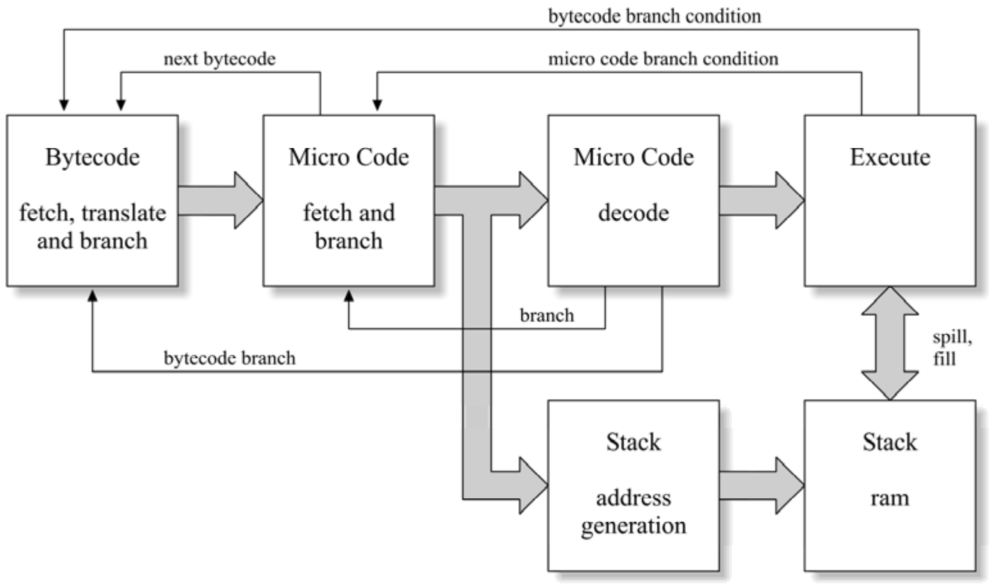

Figure 1: Datapath of JOP

An FPGA consists of 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. These two basic elements influence resource usage and speed of different design variants.

3.1 Memory

Current FPGAs (e.g. from Altera, Xilinx and Actel) provide memory blocks with two ports and usually synchronous access. These fast memory blocks are an ideal candidate for the register file of a processor. However, a pipelined RISC CPU with two operands and a different destination register needs two read and one write port. This can only be solved by doubling the memory for the second read port. A stack, on the other hand, needs only one read and one write port. Using a stack architecture for JOP has some additional benefits:

- The JVM is a stack machine resulting in a better mapping between JOP instructions and bytecodes.
- Instruction set is simpler and can be reduced to eight bits. This reduces the number of memory blocks necessary to store micro code
- No data forwarding is necessary.

The main disadvantage is that all operands have to be explicit loaded on the stack.

3.2 Micro Code

There is a great variation of Java bytecodes. Simple instructions like arithmetic and logic operations on the stack are easy to implement in hardware. However, the semantics of bytecodes like *new* or *invokestatic* can result in class loading (even over a network) and verification. These bytecodes have to be implemented in some kind of subroutine. Suns picoJava-II [3] solves this problem by implementing only a subset of the bytecodes and generating a software trap on the more complex. This solution results in a constant execution overhead for the trap.

A different approach is used in JOP. JOP has its own instruction set (the so called micro code). 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 more complex JOP just continues to execute micro code in the following cycles. The end of this sequence is coded in the instruction (as the *nxt* bit). This translation needs an extra pipeline stage but has zero overheads for complex JVM instructions.

The example in Figure 2 shows the implementation of a single cycle bytecode and a bytecode as a sequence of JOP instructions. In this example, *ineg* takes 4 cycles to execute and after the last `add` the first instruction for the next bytecode is executed.

```
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 2: Implementation of iadd, isub and ineg

The micro code is translated with an assembler to a memory initialization file, which is downloaded during configuration. No further hardware is needed to implement loadable micro code.

3.3 JOP Instruction Fetch

Figure 3 shows the second pipeline stage of JOP. Micro code that implements the JVM is stored in the eight bit wide memory labeled *jvm rom*. *jpaddr* is the starting address for the implementation of the bytecode to be executed. The table *bcfetbl* stores the micro code addresses where a new bytecode or operand has to be fetched in the first stage.

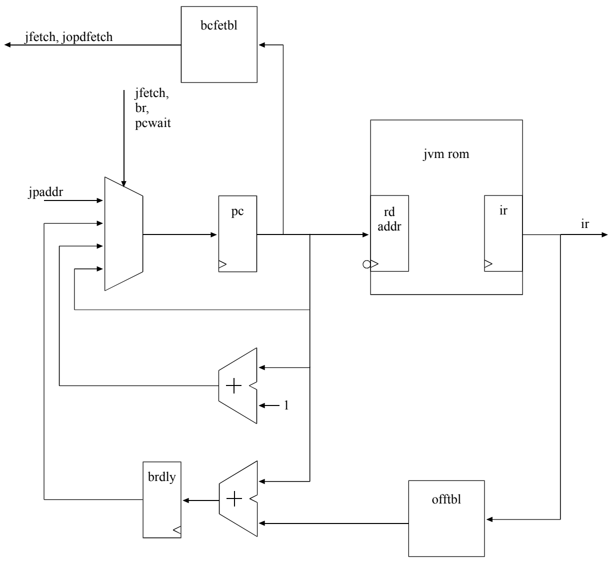

Figure 3: JOP Instruction Fetch

Many branch destinations share the same offset. A table (*offtbl*) is used to store these offsets. This indirection makes it possible to use only five bits in the instruction coding for branch targets and allow larger offsets.

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 a loadable JVM implementation a more complex solution would be necessary.

Current FPGAs don't allow asynchronous memory access. They force us to use the registers in the memory blocks. However, the output of these registers is not accessible. To avoid an additional pipeline stage just for a register-register move the read address register is clocked on the negative edge.

A different solution for this problem would be to use the output of the multiplexer for *pc* and the read address register. This solution results in a longer critical path since the multiplexer cannot longer be combined with the flip-flops that form the *pc* in the same LEs. This is another example how implementation technology (FPGA) influences architecture.

4 Utilizing FPGA Flexibility

Using a hardware description language and loading the design in an FPGA the former 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 easy between hardware and software resulting in a high 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 relative 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 32 bit single cycle multiplications are far beyond the possibilities of current FPGAs the first solution is a sequential multiplier.

4.1 Sequential Booth Multiplier in VHDL

Figure 4 shows the VHDL code of the multiplier.

```
process(clk, wr_a, wr_b)

    variable count  : integer range 0 to width;
    variable pa     : signed(64) downto 0);
    variable a_1    : std_logic;
    alias p         : signed(32 downto 0) is
                      pa(64 downto 32);
begin
    if rising_edge(clk) then
        if wr_a='1' then
            p := (others => '0');
            pa(width-1 downto 0) :=
signed(din);

        elsif wr_b='1' then
            b <= din;
            a_1 := '0';
            count := width;
        else
            if count > 0 then
                case std_ulogic_vector'(pa(0),
a_1) is
                    when "01" =>
                        p := p + signed(b);
                    when "10" =>
                        p := p - signed(b);
                    when others =>
                        null;
                end case;
                a_1 := pa(0);
                pa := shift_right(pa, 1);
                count := count - 1;
            end if;
        end if;
    end if;
    dout <= std_logic_vector(pa(31 downto 0));
end process;
```

Figure 4: Booth Multiplier

Three JOP instructions are used to access this function: *stopa* stores the first operand and *stpob* stores the second operand and starts the sequential multiplier. After 33 cycles, the result is loaded with *ldmul*. Figure 5 shows the micro code for *imul*.

```
imul:
      stopa   // first operand
      stopb   // second and start mul

      ldi 5   // 6*5+3 cycles wait
imul_loop:
      dup
      nop
      bnz imul_loop
      ldi -1   // decrement in branch slot
      add

      pop    // remove counter
      nop    // wait
      nop    // wait

      ldmul nxt
```

Figure 5: Micro Code to Access the Multiplier

4.2 Multiplication in Micro Code

If we run out of resources in the FPGA, we can move the function to micro code. The implementation of *imul* is almost identical with the Java code in Figure 6 and needs 73 JOP instructions.

4.3 Bytecode imul in Java

JOP micro code 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 micro code result in a static Java 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. Figure 6 shows the Java method for *imul*.

```
public static int imul(int a, int b) {

    int c, i;
    boolean neg = false;
    if (a<0) {
        neg = true;
        a = -a;
    }
    if (b<0) {
        neg = !neg;
        b = -b;
    }
    c = 0;
    for (i=0; i<32; ++i) {
        c <<= 1;
        if ((a & 0x80000000)!=0) c += b;
        a <<= 1;
    }
    if (neg) c = -c;
    return c;
}
```

Figure 6: *imul* in Java

The additional overhead for this implementation is a call and return with cache refills.

4.4 Implementations Compared

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. The implementation in Java loads bytecodes from a slow memory interface (8 bit, 3 cycle per byte) into the bytecode cache.

	Hardware [LE]	Micro code [Byte]	Time [Cycle]
VHDL	300	12	37
Micro code	0	73	750
Java	0	0	~2300

Table 1: Different implementations of *imul*

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.

5 Conclusion

This paper presented an overview of the architecture of a Java Optimized Processor. JOP is implemented in the low-cost ACEX FPGA [4] from Altera. Memory blocks are used to implement the stack, store micro code and for caching of Java bytecode. The processor core needs less than 2000 LEs and 6 memory blocks. It fits in an EP1K50 leaving enough LEs for IO devices.

Taking the architecture of the FPGA into account during VHDL coding, the resulting design can be optimized for an FPGA but can result in poor resource usage and speed on a different target platform. Dismissing the possibility to transfer the design to silicon leads to new options: e.g. generating VHDL code from assembler code for an embedded processor. Treating VHDL like a software language allows easy movement of function blocks between hardware and software. Further information, VHDL and Java sources can be found in [5].

References

[1] M. Schoeberl, Using a Java Optimized Processor in a Real World Application, In *Proc. Workshop on Intelligent Solutions in Embedded Systems*, Vienna, Austria, June 2003.

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

[3] Sun Microsystems. *picoJava-II Processor Core*. Data Sheet, April 1999.

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

[5] Martin Schoeberl. JOP - a Java Optimized Processor, available at: http://www.jopdesign.com/