Outline

1. Introduction
2. From Bytecodes to Hardware
3. Experimental Evaluation
4. Summary & Future Work
Motivation

Stack machines make for nice models but slow implementations, hence bytecode folding, JIT on 3-address machines

Unrolling the stack or part of it, allows for fast data access in hardware

Java processors could use a performance boost (e.g. hardware accelerators)

Bluespec SystemVerilog offers useful abstractions, good tool support, a few success stories
Motivation

**Stack machines** make for nice models but slow implementations, hence bytecode folding, JIT on 3-address machines

**Unrolling the stack** or part of it, allows for fast data access in hardware

**Java processors** could use a performance boost (e.g. hardware accelerators)

**Bluespec SystemVerilog** offers useful abstractions, good tool support, a few success stories

**Question**

Can we employ BSV and automation to generate accelerators for some of the existing Java processors?
From Java to Hardware, via BSV

- Application basic blocks
- BSV rules
- Hardware JVM rules
Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

- **typing** strong, static type-checking, polymorphism
Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

- **typing** strong, static type-checking, polymorphism
- **modules** as building blocks, encapsulating states and behavior, requiring and implementing interfaces
Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

- **typing** strong, static type-checking, polymorphism
- **modules** as building blocks, encapsulating states and behavior, requiring and implementing interfaces
- **interfaces** described as sets of methods
Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

- **typing** strong, static type-checking, polymorphism
- **modules** as building blocks, encapsulating states and behavior, requiring and implementing interfaces
- **interfaces** described as sets of methods
- **methods** atomic, guarded, callable behavior with/without side effects
Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

- **typing** strong, static type-checking, polymorphism
- **modules** as building blocks, encapsulating states and behavior, requiring and implementing interfaces
- **interfaces** described as sets of methods
- **methods** atomic, guarded, callable behavior with/without side effects
- **rules** atomic, guarded behavior snippets in modules, may trigger in every execution cycle (always in Verilog), and finish within the same cycle
Bluespec SystemVerilog

A hardware description language based on SystemVerilog:

- **typing** strong, static type-checking, polymorphism
- **modules** as building blocks, encapsulating states and behavior, requiring and implementing interfaces
- **interfaces** described as sets of methods
- **methods** atomic, guarded, callable behavior with/without side effects
- **rules** atomic, guarded behavior snippets in modules, may trigger in every execution cycle (always in Verilog), and finish within the same cycle
- **clock** is not explicitly visible (determined by the longest rule)

The compiler generates a conflict free schedule for rules/methods, and needed control logic.
Using BSV

BSV compiles to:

- **SystemC** for modeling alongside other SystemC modules
- **Verilog** for synthesis, easy to combine with other VHDL/Verilog
- **Bluesim** host executable, fast, cycle accurate simulator

A number of BSV designs have been published, including a Java processor (BlueJEP) with hardware memory management.

**Idea**

Can we transform sequences of assembly code (Java bytecodes) to hardware using BSV high-level of abstraction constructs?
A BSV **module**, providing a subset of bytecodes as **interface**. (123 bytecodes as methods and rules)
Bytecodes as Action Methods

Bytecodes transform a computing context into another context . . .

- in **one** clock cycle = one method (see Listing 1)
- over **several** cycles = start method + several rules (see Listing 2)
Bytecodes as Action Methods

Bytecodes transform a computing context into another context . . .
- in one clock cycle = one method (see Listing 1)
- over several cycles = start method + several rules (see Listing 2)

Contexts = operand stack, locals, constant pool address, Java pc
- implemented as lists of registered signals
- registered (saved) explicitly (method) or by certain bytecodes
- restored explicitly (method)
Bytecodes as Action Methods

**Bytecodes** transform a computing context into another context . . .

- in **one** clock cycle = one method (see Listing 1)
- over **several** cycles = start method + several rules (see Listing 2)

**Contexts** = operand stack, locals, constant pool address, Java pc

- implemented as lists of registered signals
- registered (saved) explicitly (method) or by certain bytecodes
- restored explicitly (method)

```verbatim
method ActionValue#(Context) isub(Context in);
  let r1 = in.stack[0];
  let r2 = in.stack[1];
  let r = r2 - r1;
  $display("isub␣\[..,%d,%d␣->␣..,%d\]",r1,r2,r);
  return Context {stack:cons(r, drop(2,in.stack)),
          locals:in.locals, cp:in.cp, jpc:in.jpc+1};
endmethod
```
Bytecodes to BSV Details

Sequences of bytecodes $\rightarrow$ basic-ish blocks $\rightarrow$ guarded rules:

- **guards** are specific method id, Java pc
- **start by** building (restoring) context from registers
- **end with** saving context explicitly, or multi-cycle bytecodes

```plaintext
rule methodA_lXtolY( !jvm.busy() &&
    jvm.getCurrentMethod() == IdA && jvm.getCurrentJPC() == X );
  let in <- jvm.restoreContext();
  let lX <- jvm.bytecode1(in, opd);
  // ... more bytecode method calls ...
  let out <- jvm.bytecodeN(in, opd1, opd2);
  jvm.saveContext(out); // or
  // invokevirtual, return, getstatic, ldc, ...
endrule
```

The choices: one rule per method (sometimes) $\leftrightarrow$ one rule per bytecode
Bytecodes to BSV Concept

- **Application Basic Blocks**
  - A
  - B
  - C
  - D
  - E
  - F

- **BSV Methods**
  - A1
  - A2
  - B
  - C1
  - C2
  - C3
  - D, E, F

- **Hardware JVM**
  - Methods
    - dup
    - iadd
    - ldc
    - iload
    - isub
    - store
    - invoke
    - saveContext
    - restoreContext
Advanced Features

**invokes & recursion** supported via a specified size context stack, limiting the depth of calls

**object access** using **JOP/BlueJEP** memory layout, through an **OPB** bus

**memory management** new bytecodes, garbage collection should be handled by the companion processor

**exceptions** have limited support, stack restore & jumps

**multi-threading** is limited, only as independent hardware JVMs.
**Tools and Setup**

**Synthesis** → device area, maximum clock frequency
- BSV compiler 2012.01.A, $BSV \rightarrow Verilog$
- Xilinx ISE 14.1, $Verilog \rightarrow FPGA$
- FPGA, Xilinx Spartan-6 (XC6SLX16)
Experimental Evaluation

Tools and Setup

Synthesis → device area, maximum clock frequency
- BSV compiler 2012.01.A, BSV → Verilog
- Xilinx ISE 14.1, Verilog → FPGA
- FPGA, Xilinx Spartan-6 (XC6SLX16)

Simulation → executed clock cycles
- Desktop, Linux
- BSV compiler 2012.01.A, BSV → Bluesim (executable)
- custom tools for parsing the output from instrumented code, as well as estimate JOP timing
Tools and Setup

Synthesis → device area, maximum clock frequency
- BSV compiler 2012.01.A, BSV → Verilog
- Xilinx ISE 14.1, Verilog → FPGA
- FPGA, Xilinx Spartan-6 (XC6SLX16)

Simulation → executed clock cycles
- Desktop, Linux
- BSV compiler 2012.01.A, BSV → Bluesim (executable)
- custom tools for parsing the output from instrumented code, as well as estimate JOP timing

Applications: ours (hand coded) vs. JOP software vs. Hanna2011 [17]
- GCD Euclid’s algorithm, (12365400, 906)
- Sieve2 Eratosthenes sieve, 100 primes
- Qsort recursive, 4000 values
## Performance

<table>
<thead>
<tr>
<th>Application</th>
<th>Method</th>
<th>Clock Cycles</th>
<th>Max. Clock (MHz)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCD</td>
<td>BSV (8 rules)</td>
<td>27,348</td>
<td>151</td>
<td>0.181</td>
</tr>
<tr>
<td></td>
<td>Hanna</td>
<td>54,652</td>
<td>200</td>
<td>0.273</td>
</tr>
<tr>
<td></td>
<td>JOP</td>
<td>218,790</td>
<td>93</td>
<td>2.353</td>
</tr>
<tr>
<td>Sieve2</td>
<td>BSV (12 rules)</td>
<td>32,475</td>
<td>152</td>
<td>0.214</td>
</tr>
<tr>
<td></td>
<td>Hanna</td>
<td>16,023</td>
<td>125</td>
<td>0.128</td>
</tr>
<tr>
<td></td>
<td>JOP</td>
<td>113,198</td>
<td>93</td>
<td>1.217</td>
</tr>
<tr>
<td>Qsort</td>
<td>BSV (30 rules)</td>
<td>1,669,820</td>
<td>117</td>
<td>14.272</td>
</tr>
<tr>
<td></td>
<td>Hanna (Iter.)</td>
<td>486,520</td>
<td>125</td>
<td>3.892</td>
</tr>
<tr>
<td></td>
<td>JOP</td>
<td>4,377,628</td>
<td>93</td>
<td>47.071</td>
</tr>
</tbody>
</table>

Device area: published data to compare to is lacking, see Table 2.
Finally...

**Summary**  A method for translating Java bytecode sequences to hardware, via Bluespec SystemVerilog,
- well suited for automation
- intended for acceleration (e.g. of JOP, BlueJEP)

**To Do**  
- complete the translator tool
- optimize the partitioning into rules
- add simple new bytecodes implementations
Thank you!
Thank you!

Questions?