Comparison of Object-oriented and Functional Programming for Code Generation

Eric Allen

April 21st, 2010

Abstract

Creating software systems begins with creating abstractions that model the problem and/or solution. The most popular paradigm for creating these abstractions is Object-Oriented Programming (OOP), but Functional Programming (FP) is gaining traction. Functional programs are reputedly easier to reason about, simpler to understand, and friendlier to concurrency. Computer.Build is a code generation tool for designing simple microprocessors in Ruby and Clojure (a Lisp derivative) that generates VHDL for compilation. This tool was written in both Clojure and Ruby to compare the relative merits of each language for the particular application of code generation. While the two implementations have much in common, the functional implementation in Clojure is clearer, simpler, and more consistent. However, the Ruby implementation is easier to read, easier to modify, and easier to debug. These results suggest that functional programming could be better than object-oriented programming for building software systems in some situations, but when using domain-specific languages for code generation, object-orientation makes a software developer's life much easier.

1 Introduction

Building software is about creating abstractions to model the problem and solution, then applying those abstractions to the task at hand. Programming languages have developed out of the common need for a basic set of abstractions, such as subroutines and data types. Most languages are designed for serial execution on a standard von Neumann computer and share many abstractions for developing solutions targeted at these systems. In the same way, VHDL and Verilog have developed as languages targeting Field Programmable Gate Arrays (FPGAs) and other hardware synthesis applications. While VHDL and Verilog may look like conventional programming languages at the surface, their fundamental abstractions are quite different from mainstream toolchains such as C or Java. They compile to a different set of primitive operations, such as look-up tables and combinational digital logic, and parallelism is inherent in the hardware synthesis process.

All computer languages have limitations on their expressiveness, many of which can be overcome with design patterns. In some cases, however, the languages are simply not powerful enough to clearly express the abstractions a software developer uses to build software. When this is the case, the best response is to switch to a more expressive language, but this is not always possible for a myriad of reasons, including legacy code or a specialized platform. When it is necessary to express concepts at a higher level than the target language can support, one option is source code generation, (also known as automatic programming): using a higher-level language to generate source code in a target language. Source code generation is quite common. For example, the Ruby on Rails web framework provides a "generate" utility to automatically generate default scaffolding for common class types.

Most modern software is built on object-oriented design, but it is not always the best tool. Functional, data-oriented programming fits some tasks more naturally. Ruby, with roots in Smalltalk, is one of the purest object-oriented languages in use. Clojure is a recent addition to the Lisp family that includes more basic data types and complete Java integration. VHDL, a popular hardware description language, lacks facilities for metaprogramming and higher-level abstraction. Computer.Build is an attempt to build a domain-specific language (DSL) for creating simple computer architectures, implemented in both Ruby and Clojure. In the process of developing Computer.Build, I have analyzed the differences between implementing source code generation and abstract syntax tree (AST) manipulation in a functional style in Clojure and an object-oriented style in Ruby. In the remainder of this paper, I will make qualitative comparisons between these two implementations of VHDL code generation.

2 Background

Source code generation has become ubiquitous in the software world, despite significant controversy about its efficacy. Some software developers argue that source code generation is a poor workaround for bad programming or limited languages. Others use it on a daily basis to accelerate their development process. Regardless, both Clojure and Ruby are used for source code generation in popular applications.

CodeWorker [2] is a general-purpose framework for both source code generation and parsing of arbitrary grammars. Provided with an "extended-BNF" grammar description, CodeWorker can parse a piece of code, then generate different code based on the abstract syntax tree. Unfortunately, CodeWorker is not designed for AST transformations, sacrificing flexibility for simplicity. As such, it would be difficult to apply CodeWorker to the problem of transforming an instruction set into a working VHDL description of a computer that implements the instruction set.

The Ruby on Rails framework for web application development has sparked significant controversy over its system of generating default code for basic database manipulation. While this is by far the most common use of source code generation in Ruby, other Ruby tools also make use of source code generation. The RGen framework [3] provides a rich framework for generating code based on a domain model, including automatic creation of a Ruby-based DSL. CGen [4] is a more pragmatic tool for generating C source code from Ruby, designed for writing C extensions to the Ruby language in Ruby itself, letting the tool generate C source code from Ruby source code. Given the wealth of Ruby tools for source code generation, Ruby is clearly a popular language for implementing source code generation.

Due to its homoiconic nature, Lisp is well-suited to generating Lisp code. It is a popular choice for metaprogramming, and its macro facility is essentially very powerful source code generation. Clojure, being a Lisp derivative, has similar attractive properties for code generation. However, generating non-Lisp code from Lisp is not as common. Tools such as CLiCC [5] exist for transforming Lisp code to other languages, such as C, but the aim is generally to run real Lisp code. Computer.Build, on the other hand, is a DSL on top of Lisp (or Ruby), and as such bears little resemblance to Lisp. It does, however, conform to Lisp syntax, allowing for the use of the built-in Lisp reader to parse the code.

For VHDL, source code generation is commonplace. Major FPGA design tools, such as Altera Quartus, generate source code in VHDL and Verilog for various components based on a "wizard" dialog that allows the user to set parameters. While software developers are often uncomfortable with source code generation, hardware developers depend on it constantly. The ubiquity of VHDL source code generation makes it a good target for the high-level to low-level transformation provided by Computer.Build.

3 Implementation

In order to compare OOP and FP, two implementations of the same program were developed. The primary implementation of Computer.Build was developed in Clojure, using simple, composable data structures to represent the Abstract Syntax Tree (AST) and leverage the general-purpose functions available in Clojure's standard library. The secondary implementation of Computer.Build was developed in Ruby using object-oriented principles, including inheritance and polymorphism. The two implementations leveraged their respective languages' strengths.

3.1 Functional (Clojure)

Due to Clojure's functional nature, the Computer.Build implementation (Appendix B) in Clojure was designed in a functional style, with simple, immutable data structures not bound to specific functionality. Generic functions, like the `make-states-for-instructions` (Listing 1) then transform these data structures into others, eventually producing VHDL source code.

Listing 1: make-states-for-instructions function

```clojure
(defn make-states-for-instruction [[_ instruction-name & RTLs]]
  (let [microcode (flatten-1 (map rtl-to-microcode RTLs))
        last-index (- (count microcode) 1)]
    (apply merge (map (partial link-state instruction-name last-index)
                      microcode (iterate inc 0)))))
```

Some modularity is achieved by dividing up functions into separate namespaces, but the vast majority of the code is in a single file. Approximately half of that code is static VHDL-in-Clojure code that wraps around the dynamically generated segments, so effectively the entire interesting part of the program is 100 lines. Another 180 lines are dedicated to taking VHDL-in-Clojure code and transforming it into real VHDL, but the entire codebase clocks in at only a few hundred lines of code. Almost all functions are under twenty lines, making for clear separation of different pieces.

Computer architectures are defined using a simple DSL (Listing 2) that is transformed into a literal Clojure data structure by a macro. This overall architectural specification is then transformed into another stucture representing the states of the state machine to be generated. A separate state machine generator is then invoked, which creates VHDL-in-Clojure code that is fed through a simple recursive-descent compiler that generates the final VHDL for the control subsystem. The top-level design is literal VHDL-in-Clojure code with a small amount of dynamic code for interfacing with the dynamically-generated control subsystem.

VHDL-in-Clojure is compiled to Clojure using a single multimethod, called `to-vhdl`, that dynamically dispatches based on the first symbol of the passed-in data structure. The `to-vhdl` method calls itself recursively to generate sub-nodes, allowing for almost complete orthogonality of statements. The output of `to-vhdl` is a tree of strings, which is then concatenated with appropriate linebreaks. Indentation is handled automatically by the natural nesting of this tree.

3.2 Object-Oriented (Ruby)

Ruby, being a Smalltalk descendant, is a deeply object-oriented language. Therefore, Computer.Build's Ruby implementation (Appendix A) is object-oriented, using Ruby's block syntax to create a DSL for VHDL-in-Ruby (Listing 4). The Ruby code is roughly broken up into modules along the same lines as the Clojure modules, but the actual process by which Computer.Build in Ruby generates VHDL is quite different. While the Clojure implementation simply uses a literal data structure to represent the instruction set, the Ruby version creates a DSL for writing instructions.

The Ruby implementation defines a DSL using Ruby blocks that allows for specification of a computer's microcode, much like the Clojure version. The user-defined architecture written in the Computer.Build DSL and evaluted at runtime to create an object representing the computer's definition. The `generate` method is then called on this object, which in turn creates a state machine object (using the state machine DSL), generates it, then creates a top-level entity. The process of transforming VHDL-in-Ruby behaves somewhat like the Clojure process, but instead of a simple data structure, each node is represented by a particular class that implements the `generate` method, which may call the `generate` method of children to compile a complete VHDL design.

3.3 Results

Both implementations turned out to take approximately the same amount of code. While the Clojure implementation contains fewer absolute lines of code, the discrepancy is mostly due to due to the language's syntax and tendency to chain function calls on one line. The Clojure implementation was able to use macros (Listing 2) to simplify the VHDL compiler, while the Ruby implementation achieved a similar simplification using inheritance. The Ruby compiler, however, is still approximately 3x the size of the Clojure compiler as measured by the number of lines of code.

Listing 2: Macros to help define VHDL generation

```clojure
; Multi-line statement that causes an extraneous level of nesting in AST
(defmacro def-vhdl-multiline [kword bindings & body]
  `(defmethod to-vhdl ~kword [~(vec (concat '[_] bindings))]
     (not-indented (list ~@body))))

; Inline or single-line statements that don't produce a list of lines
(defmacro def-vhdl-inline [kword bindings & body]
  `(defmethod to-vhdl ~kword [~(vec (concat '[_] bindings))]
     (str ~@body)))
```

After the initial writing of both versions, many modifications were made to improve the output and increase functionality. Modifying the Ruby implementation was quite easy, while modifying the Clojure turned out to be considerably more challenging. The time required just to understand code written several weeks prior in Clojure was on the order of 30 minutes to an hour, with productive development happening only after the existing code was understood. Due to its simpler, more imperative style, the Ruby code was much easier to understand. While the Clojure architecture was cleaner, and arguably more pure, it was restrictive to the point of significantly impeding progress. Adding conditional branching, for example, required a re-think of the Clojure architecture, whereas the Ruby implementation was flexible enough to be bent to the new approach. Long-term, the purity of the Clojure architecture could be an advantage, but in the early stages of this project, it was a hindrance.

3.4 Example processor definitions

Listing 3: Clojure example input

```clojure
(require 'computer-build)
```

```
; RTL-level description
(computer-build/build "mccalla2"
  ; options
  {:address-width 4}
  (instruction "cla"
               (A <- 0))
  (instruction "cmp"
               (A <- (complement A)))
  (instruction "inc"
               (A <- (+ A 1)))
  (instruction "neg"
               (A <- (complement A))
               (A <- (+ A 1)))
  (instruction "adddir"
               (MA <- (and IR 0x0F))
               (A <- (+ A MD)))
  (instruction "subdir"
               (MA <- (and IR 0x0F))
               (A <- (- A MD)))
  (instruction "addind"
               (MA <- (and IR 0x0F))
               (MA <- (+ MD 0))
               (A <- (+ A MD)))
  (instruction "subind"
               (MA <- (and IR 0x0F))
               (MA <- (+ MD 0))
               (A <- (- A MD)))
  (instruction "lda"
               (MA <- (and IR 0x0F))
               (A <- MD))
  (instruction "sta"
               (MA <- (and IR 0x0F))
               (MD <- A))
  (instruction "jmp"
               (PC <- (and IR 0x0F)))
  (instruction "bra0"
               (if (= A 0)
                 (PC <- (and IR 0x0F)))))
```

Listing 4: Ruby example input

```
require 'Computer.Build'

# RTL-level description
Computer.Build "mccalla" do |computer|
  computer.address_width = 4

  computer.instruction "cla" do |i|
    i.move :A, 0
  end
  computer.instruction "inc" do |i|
    i.move :A, add(:A, 1)
  end
  computer.instruction "neg" do |i|
    i.move :A, complement(:A)
    i.move :A, add(:A, 1)
  end
  computer.instruction "add" do |i|
    i.move :MA, bitwise_and(:IR, 0x0F)
    i.move :A, add(:A, :MD)
  end
  computer.instruction "lda" do |i|
    i.move :MA, bitwise_and(:IR, 0x0F)
    i.move :A, :MD
  end
  computer.instruction "sta" do |i|
    i.move :MA, bitwise_and(:IR, 0x0F)
    i.move :MD, :A
  end
  computer.instruction "jmp" do |i|
    i.move :pc, bitwise_and(:IR, 0x0F)
  end
  computer.instruction "bra0" do |i|
    i.if equal(:A, 0) do |thn|
      thn.move :pc, bitwise_and(:IR, 0x0F)
    end
  end
end
```

4 Conclusions

Implementing code generation in a functional style using Clojure yields a smaller, more consistent, but harder to understand program compared to an object-oriented design using Ruby. Both languages are powerful and have their place, but for this particular task, Ruby appears to be a better choice.

4.1 Syntactic

Members of the Lisp family tend to have extremely minimal syntax. This pushes much more of the meaning of the program up to the lexical level, which can vary between developers, and even between programs written by the same developer. While this is considered an advantage by many Lisp advocates, it can also be a hindrance: every Lisp (or Clojure) program must be understood on its own, without many commonalities with other programs. Ruby, on the other hand, has a much richer syntax. While it is still flexible enough to support easy creation of Domain-Specific Languages, it gives software developers much more common structure between programs. Common structure and concepts make switching between projects easier for developers, improving efficiency and reducing overhead. For example, the custom code injection for the `store_instruction` state is easier to read in Ruby (Listing 5) than in Clojure (Listing 6). Multiple developers working on the same project face the same commonality dilemma with a language like Clojure: without considerable documentation, developers may not have consistent mental models of the program, leading to very different implementations in their respective components.

Listing 5: Ruby state machine generation

```
          if name == 'store_instruction'
            s.if event(:clock), VHDL::Equal.new(:clock,"0") do |thn|
              thn.assign :opcode, "#{opcode_length-1} downto 0",
                :system_bus, "7 downto #{7-opcode_length+1}"
            end
          end
```

Listing 6: Clojure state machine generation

```
                    :store_instruction
                      {:next :decode,
                       :control-signals '(:rd_MD, :wr_IR, :inc_pc),
                       :code
                       [`(if (and (event :clock) (= :clock 0))
                           (<= :opcode ~(- opcode-width 1) 0
                               :system_bus 7 ~(- 8 opcode-width)))]}
```

4.2 Architectural

The dichotomy between functional and object-oriented programming becomes most relevant during the architectural design phase of a software project. Even using a "general-purpose" tool, such as the Unified Modeling Language, can nudge a software developer into the OOP paradigm, so care must be taken to avoid deciding prematurely on an architectural paradigm. when first designing a system. When the problem at hand involves manipulating virtual doppelgangers of real objects, an OOP design is likely the best choice. However, when the design focuses on operations on data, rather than the representation of the data, a functional approach can be more appropriate. When operating on homogeneous data structures, like trees or arrays, functional languages have a wealth of tools built-in, and more can be created easily. Functional designs also support parallelism better, because they emphasize pure functions, which always return the same result given the same inputs. Clojure especially takes advantage of this aspect of functional programming to create powerful abstractions for concurrency and parallelism at the language level. Fundamentally, the optimal language choice for architectural fit depends on the developer's mindset. A software developer who is comfortable with object-oriented architecture will be better off building OO software, whereas a developer more experienced in functional design will likely be more productive using FP.

5 Recommendations

Software development is as much about reading code as it is about writing code. Donald Knuth's literate programming [1] is an extreme example, but almost every program written must be read at some point in its lifetime. In a team environment, readability becomes even more crucial, as other members of the team must read code that each member writes. Therefore, an important language criterion is readability, and Ruby is a clear winner here over Clojure. While Clojure is considerably more terse, the simple concepts of OOP make understanding the Ruby code easier, and Lisp's extremely simple syntax can be challenging for a human to parse. Ruby's executable class bodies and block syntax allow for quite extensive metaprogramming in a much easier way than Clojure's macro system, again favoring Ruby for readability.

In addition to poor readability, Clojure code is considerably harder to debug. Simple, straightforward code will yield useful stack traces, but complicated code paths with macros and first-class functions can often generate cryptic stack traces. Clojure's lazy evaluation also causes trouble, as errors can surface in places where they were not expected. Debugging also takes up a significant portion of software development time, so debuggability is a crucial attribute on which to evaluate languages. Ruby's metaprogramming facilities can also produce confusing stack traces (as often occurs in Ruby on Rails), but an object-oriented design tends to produce simpler stack traces than a functional design, and Ruby is generally used for object-oriented programming.

Ruby can be used to some degree for functional programming, and it supports object-oriented programming very well. Its syntax is one of the most readable of any popular language, yet it is flexible enough to support some metaprogramming. For programs of the scale and scope of Computer.Build, Clojure's advantages are barely relevant. When building a small, or medium-sized piece of software, Ruby is a great choice, and generally preferable to Clojure. However, when building complex systems that include significant concurrency, Clojure may be the better choice.

6 Future Work

6.1 Computer generation

Computer.Build is capable of generating working computers, but they are slow and inefficient. Now that the basic framework is in place, potential exists for many optimizations and increased flexibility.

The data path generated by the current Computer.Build is completely bus-oriented, forcing instructions to take many clock cycles to complete their microcode as data is transferred across the single bus. The compiler knows all possible transfers from the microcode specification, so it is quite possible to generate a directed graph of register transfers, and from that generate an optimized data path using multiplexers. This would make the system faster and allow microcoded states to be collapsed in some situations.

Computer.Build was originally intended to be completely architecture-agnostic, but it has developed a number of assumptions about the target architecture that need to be eliminated. Currently the bus width is fixed at 8 bits instead of being read from the architecture definition. The set of possible ALU operations is fixed by the ALU definition, but this could potentially be generated (or at least optimized) based on the user's definitions. In addition, conditionals are not currently particularly flexible, and the possibility exists for a dynamically generated, dedicated design entity for handling conditional logic.

6.2 Language comparison

To better compare Ruby and Clojure, it would be useful to implement Computer.Build in a functional style in Ruby and in an object-oriented style in Clojure. Both languages are flexible enough to support both paradigms, and comparing the two languages' implementations of the exact same constructs would yield a more direct comparison. However, using a language designed for one paradigm in a way that runs counter to its design is counter-productive, and a developer is likely better off switching to a language more suited to the particular application.

Ruby and Clojure are both dynamically-typed languages, allowing for great flexibility and minimal boilerplate code. Many software developers prefer statically-typed languages, and Computer.Build could certainly be implemented in one. The project is intended as a non-trivial codebase for comparing higher-level language constructs, and it could be used for comparing statically-typed and dynamically-typed languages. Haskell, for example, would make an interesting target language for a third Computer.Build port.

7 Appendix A: Ruby implementation source code

The source code to Computer.Build is presented here for easy access, but the most recent version can always be found at `http://github.com/epall/Computer.Build` and is licensed under the MIT License.

Listing 7: Computer.Build main module

```
require 'computer_build/vhdl'
require 'computer_build/state_machine'

class Computer
  include VHDL::Helpers

  def self.Build(name)
    instance = Computer.new(name)
    yield(instance)
    instance.generate
  end

  attr_writer :address_width

  def initialize(name)
    @name = name
    @instructions = []
  end

  # DSL method
  def instruction(name)
    inst = Instruction.new(name)
    yield(inst)
    @instructions << inst
  end

  def generate
    states = make_states(@instructions).merge(static_states)
    opcodes = make_opcodes(@instructions)
    opcode_length = opcodes.values.first.length
    control_signals = states.values.map(&:control_signals).flatten.uniq

    control = state_machine("control_unit") do |m|
      m.input :reset, VHDL::STD_LOGIC
      m.input :condition, VHDL::STD_LOGIC
      m.inout :system_bus, VHDL::STD_LOGIC_VECTOR(7..0)
      m.output :alu_operation, VHDL::STD_LOGIC_VECTOR(2..0)
      control_signals.each do |sig|
        m.output sig, VHDL::STD_LOGIC
      end

      m.signal :opcode,
        VHDL::STD_LOGIC_VECTOR((opcode_length -1)..0)

      m.reset do |r|
        r.goto :fetch
        control_signals.each do |sig|
          r.low sig.to_sym
        end
        r.assign :alu_operation, "000"
        r.assign :system_bus, "ZZZZZZZZ"
      end

      states.each do |name, state|
    m.state(name) do |s|
      control_signals.each do |sig|
        s.assign sig, state.control_signals.include?(sig) ? '1' : '0'
      end
      if state.alu_op
        s.assign :alu_operation, state.alu_op.opcode
      else
        s.assign :alu_operation, "000"

      if state.constant_value
        s.assign :system_bus, state.constant_value.to_logic(8)
      else
        s.assign :system_bus, "ZZZZZZZZ"
      end

      if name == 'store_instruction'
        s.if event(:clock), VHDL::Equal.new(:clock,"0") do |thn|
          thn.assign :opcode, "#{opcode_length-1} downto 0",
            :system_bus, "7 downto #{7-opcode_length+1}"
        end
      end
    end

    if state.condition
      m.transition :from => name, :to => state.next,
        :on => VHDL::Equal.new(:condition, "0")
      m.transition :from => name, :to => name+"_0",
        :on => VHDL::Equal.new(:condition, "1")
    else
      m.transition :from => name, :to => state.next if state.next
    end
  end

  # instruction decode
  opcodes.each do |instruction, opcode|
    m.transition :from => :decode, :to => instruction.name+"_0",
      :on => equal(:opcode, opcode)
  end
end

main = entity("main") do |e|
  e.port :clock ,:in, VHDL::STD_LOGIC
  e.port :reset, :in, VHDL::STD_LOGIC
  e.port :bus_inspection, :out, VHDL::STD_LOGIC_VECTOR(7..0)

  e.signal :system_bus, VHDL::STD_LOGIC_VECTOR(7..0)
  e.signal :alu_operation, VHDL::STD_LOGIC_VECTOR(2..0)
  e.signal :alu_condition, VHDL::STD_LOGIC

  control_signals.each do |sig|
    e.signal sig, VHDL::STD_LOGIC
  end

  e.component :reg do |c|
    c.in :clock, VHDL::STD_LOGIC
    c.in :data_in, VHDL::STD_LOGIC_VECTOR(7..0)
    c.out :data_out, VHDL::STD_LOGIC_VECTOR(7..0)
    c.in :wr, VHDL::STD_LOGIC
    c.in :rd, VHDL::STD_LOGIC
  end

  e.component :program_counter do |c|
    c.in :clock, VHDL::STD_LOGIC
    c.in :data_in, VHDL::STD_LOGIC_VECTOR(7..0)
    c.out :data_out, VHDL::STD_LOGIC_VECTOR(7..0)
    c.in :wr, VHDL::STD_LOGIC
    c.in :rd, VHDL::STD_LOGIC
    c.in :inc, VHDL::STD_LOGIC
  end

  e.component :ram do |c|
    c.in :clock, VHDL::STD_LOGIC
    c.in :data_in, VHDL::STD_LOGIC_VECTOR(7..0)
    c.out :data_out, VHDL::STD_LOGIC_VECTOR(7..0)
    c.in :address, VHDL::STD_LOGIC_VECTOR(4..0)
    c.in :wr_data, VHDL::STD_LOGIC
    c.in :wr_addr, VHDL::STD_LOGIC
    c.in :rd, VHDL::STD_LOGIC
  end

  e.component :alu do |c|
    c.in :clock, VHDL::STD_LOGIC
    c.in :data_in, VHDL::STD_LOGIC_VECTOR(7..0)
    c.out :data_out, VHDL::STD_LOGIC_VECTOR(7..0)
    c.in :op, VHDL::STD_LOGIC_VECTOR(2..0)
    c.in :wr_a, VHDL::STD_LOGIC
    c.in :wr_b, VHDL::STD_LOGIC
    c.in :rd, VHDL::STD_LOGIC
    c.out :condition, VHDL::STD_LOGIC
  end

  e.component :control_unit do |c|
    c.in :clock, VHDL::STD_LOGIC
    c.in :reset, VHDL::STD_LOGIC
    c.in :condition, VHDL::STD_LOGIC
    c.out :alu_operation, VHDL::STD_LOGIC_VECTOR(2..0)
    control_signals.each do |sig|
      c.out sig, VHDL::STD_LOGIC
    end
    c.inout :system_bus, VHDL::STD_LOGIC_VECTOR(7..0)
  end

  e.behavior do |b|
    b.instance :program_counter, "pc", :clock, :system_bus, :system_bus,
      :wr_pc, :rd_pc, :inc_pc
    b.instance :reg, "ir", :clock, :system_bus, :system_bus, :wr_IR, :rd_IR
    b.instance :reg, "A", :clock, :system_bus, :system_bus, :wr_A, :rd_A
    b.instance :ram, "main_memory", :clock, :system_bus, :system_bus,
      subbits(:system_bus, 4..0), :wr_MD, :wr_MA, :rd_MD
    b.instance :alu, "alu0", :clock, :system_bus, :system_bus,
        :alu_operation, :wr_alu_a, :wr_alu_b, :rd_alu, :alu_condition
      b.instance :control_unit, "control0", [:clock, :reset, :alu_condition,
          :alu_operation] + control_signals + [:system_bus]
      b.assign :bus_inspection, :system_bus
    end
  end

  Dir.mkdir @name rescue nil # ignore error if dir already exists
  File.open(File.join(@name, 'control.vhdl'), 'w') do |f|
    generate_vhdl(control, f)
  end

  File.open(File.join(@name, 'main.vhdl'), 'w') do |f|
    generate_vhdl(main, f)
  end
end

# inner classes

class Instruction
  attr_reader :name, :steps

  def initialize(name)
    @name = name
    @steps = []
  end

  def move(target, source)
    @steps << RTL.new(target, source)
  end

  def microcode
    steps.map(&:to_microcode).flatten
  end

  def if(condition, &body)
    @steps << Conditional.new(condition, &body)
  end
end

class ALUOperation
  attr_reader :op, :operands

  def initialize(op, *operands)
    @op = op
    @operands = operands
  end

  def opcode
    return {
      :and        => "001",
      :or         => "010",
      :complement => "011",
      :add        => "100",
      :subtract   => "101",
        :equal       => "110",
        :lessthan    => "111"}[@op]
    end
  end

  private

  class Conditional
    def initialize(condition)
      @condition = condition # instance of ALUOperation
      @steps = []
      @true_body = []
      yield self
    end

    # DSL method
    def move(target, source)
      @steps << RTL.new(target, source)
    end

    def to_microcode
      steps = []
      steps << MicrocodeState.new do |state|
        state.control_signals = ["rd_#{@condition.operands.first}", "wr_alu_a"]
        state.alu_op = @condition
      end
      conditional = MicrocodeState.new do |state|
        op = @condition.operands.last
        state.control_signals = ["wr_alu_b"]

        if op.is_a? Fixnum
          state.constant_value = op
        else
          state.control_signals += "rd_#{op}"
        end

        state.alu_op = @condition
        state.condition = @condition
      end
      steps << conditional

      body = @steps.map(&:to_microcode).flatten
      conditional.body_size = body.length

      body.each {|state| state.conditional = conditional}

      return steps + body
    end
  end

  class RTL
    def initialize(target, source)
      @target = target
      @source = source
    end
    def to_microcode
      if @source.is_a? Fixnum
        return MicrocodeState.new do |state|
          state.control_signals = "wr_#{@target}"
          state.constant_value = @source
        end
      elsif @source.is_a? Symbol
        return MicrocodeState.new do |state|
          state.control_signals = ["wr_#{@target}", "rd_#{@source}"]
        end
      elsif @source.is_a? ALUOperation
        steps = []
        steps << MicrocodeState.new do |state|
          state.control_signals = ["rd_#{@source.operands.first}", "wr_alu_a"]
          state.alu_op = @source
        end

        if @source.operands.length == 2
          steps << MicrocodeState.new do |state|
            state.control_signals = ["wr_alu_b"]
            if @source.operands.last.is_a? Fixnum
              state.constant_value = @source.operands.last
            else
              state.control_signals << "rd_#{@source.operands.last}"
            end
          end
        end

        steps << MicrocodeState.new do |state|
          state.control_signals = ["rd_alu", "wr_#{@target}"]
          state.alu_op = @source
        end
        return steps
      end
    end
  end

  class MicrocodeState
    attr_accessor :control_signals, :alu_op, :constant_value, :next,
      :condition, :conditional, :body_size, :index

    def initialize(&blk)
      yield(self) if blk
    end
  end

  def make_states(instructions)
    states = {}
    instructions.each do |instr|
      steps = instr.microcode
      indexes = {nil => 0}
      steps.each do |step|
        indexes[step.conditional] ||= 0
        index = indexes[step.conditional]
```

```
        if step.conditional
          if index < step.conditional.body_size - 1
            step.next = instr.name+"_"+(step.conditional.index.to_s)+"_"+(index+1).to_s
          else
            step.next = step.conditional.next
          end
          states[instr.name+"_"+(step.conditional.index.to_s)+"_"+index.to_s] = step
        else
          if index < steps.reject(&:conditional).length - 1
            step.next = instr.name+"_"+(index+1).to_s
          else
            step.next = :fetch
          end
          states[instr.name+"_"+index.to_s] = step
        end
        step.index = index
        indexes[step.conditional] += 1
      end
    end

    return states
  end

  def make_opcodes(instructions)
    bits = (Math.log(instructions.length)/Math.log(2)).ceil
    opcodes = {}
    instructions.each_with_index do |instruction, idx|
      bin_string = idx.to_s(2)
      bin_string = ("0" * (bits - bin_string.length)) + bin_string
      opcodes[instruction] = bin_string
    end
    return opcodes
  end

  def static_states
    states = {}
    states['fetch'] = MicrocodeState.new do |s|
      s.control_signals = ['rd_pc', 'wr_MA']
      s.next = 'store_instruction'
    end

    states['store_instruction'] = MicrocodeState.new do |s|
      s.control_signals = ['rd_MD', 'wr_IR', 'inc_pc']
      s.next = 'decode'
    end

    states['decode'] = MicrocodeState.new do |s|
      s.control_signals = []
    end

    return states
  end
end
```

```
def complement(value)
  Computer::ALUOperation.new(:complement, value)
end

def add(operand1, operand2)
  Computer::ALUOperation.new(:add, operand1, operand2)
end

def bitwise_and(operand1, operand2)
  Computer::ALUOperation.new(:and, operand1, operand2)
end

def subtract(operand1, operand2)
  Computer::ALUOperation.new(:subtract, operand1, operand2)
end

def equal(operand1, operand2)
  Computer::ALUOperation.new(:equal, operand1, operand2)
end
```

Listing 8: State machine generator

```
require 'computer_build/vhdl'

module ComputerBuild
  class State
    include VHDL::StatementBlock

    attr_reader :name

    def initialize(name, body)
      @name = name
      @statements = []
      body[self]
    end

    def self.full_name(shortname)
      ("state_"+shortname.to_s).to_sym
    end
  end

  class Transition
    attr_reader :from, :to, :condition
    def initialize(options)
      @from = options[:from]
      @to = options[:to]
      @condition = options[:condition] || options[:on]
    end
  end

  class StateMachine
    include VHDL::Helpers
    def initialize(name, body)
      @name = name
      @inputs = []
      @outputs = []
    @inouts = []
    @signals = []
    @states = []
    @transitions = []
    body[self]
  end

  def inputs(*rest)
    @inputs = rest
  end

  def input(name, type)
    @inputs << [name, type]
  end

  def outputs(*rest)
    @outputs = rest
  end

  def output(name, type)
    @outputs << [name, type]
  end

  def inout(name, type)
    @inouts << [name, type]
  end

  def signal(*rest)
    @signals << rest
  end

  def state(name, &body)
    @states << State.new(name, body)
  end

  def reset(&body)
    @reset = body
  end

  def transition(options)
    @transitions << Transition.new(options)
  end

  def generate(out)
    representation = entity(@name) do |e|
      e.port "clock", :in, VHDL::STD_LOGIC

      @inputs.each do |pair|
        name, type = pair
        e.port name, :in, type
      end

      @outputs.each do |pair|
        name, type = pair
        e.port name, :out, type
      end

      @inouts.each do |pair|
        name, type = pair
        e.port name, :inout, type
      end

      e.type "STATE_TYPE", @states.map {|s| "state_"+s.name.to_s}
      e.signal "state", "STATE_TYPE"
      @signals.each do |args|
        e.signal(*args)
      end

      e.behavior do |b|
        b.process [:clock, :reset, :state] do |p|
          if @reset
            ifelse = p.if(equal(:reset, '1')) do |b|
              def b.goto(state)
                self.assign(:state, ("state_"+state.to_s).to_sym)
              end
              @reset.call(b)
            end

            ifelse.else do |b|
              b.case :state do |c|
                @states.each do |state|
                  c["state_" + state.name.to_s] = state
                end
              end

              b.if event(:clock), equal(:clock, "1") do |b|
                @transitions.each do |transition|
                  conditions = [equal(:state, State.full_name(transition.from))]
                  conditions << transition.condition unless transition.condition.nil?
                  b.if(*conditions) do |c|
                    c.assign(:state, State.full_name(transition.to))
                  end
                end
              end
            end
          else
            p.case :state do |c|
              @states.each do |state|
                c["state_" + state.name.to_s] = state
              end
            end

            p.if event(:clock), equal(:clock, "1") do |b|
              @transitions.each do |transition|
                conditions = [equal(:state, State.full_name(transition.from))]
                conditions << transition.condition unless transition.condition.nil?
                b.if(*conditions) do |c|
                  c.assign(:state, State.full_name(transition.to))
                end
              end
            end
          end
        end # process
      end
    end

    representation.generate(out)
  end
end
end

def state_machine(name, &body)
ComputerBuild::StateMachine.new(name, body)
end
```

Listing 9: VHDL-in-Ruby to VHDL compiler

```
module VHDL
  STD_LOGIC = "STD_LOGIC"

  def self.STD_LOGIC_VECTOR(range)
    if range.first > range.last
      return "STD_LOGIC_VECTOR(#{range.first} downto #{range.last})"
    else
      return "STD_LOGIC_VECTOR(#{range.first} upto #{range.last})"
    end
  end

  module StatementBlock
    def case(input, &body)
      @statements << Case.new(input, body)
    end

    def if(*conditions, &body)
      ifthenelse = If.new(conditions, body)
      @statements << ifthenelse
      return ifthenelse
    end

    def assign(*args)
      @statements << Assignment.new(*args)
    end

    def high(target)
      assign(target, '1')
    end

    def low(target)
      assign(target, '0')
    end

    # Default generate, generally overridden
    def generate(out, indent)
      @statements.each {|s| s.generate(out, indent + 1)}
    end
  end
```

```
class Statement
  protected

  def quoted(expression)
    if expression.is_a? String
      if expression.length == 1
        return "'#{expression}'"
      else
        return "\"#{expression}\""
      end
    elsif expression.is_a? Fixnum
      return "std_logic_vector(#{expression})"
    else
      return expression
    end
  end
end

class SingleLineStatement < Statement
  def generate(out, indent)
    out.print "  " * indent
    out.print self.line()
    out.print "\n"
  end
end

class MultiLineStatement < Statement
end

class InlineStatement < Statement
end

class Entity
  attr_reader :name
  def initialize(name, body)
    @name = name
    @ports = []
    @signals = []
    @types = []
    @components = []
    body[self]
  end

  def port(*args)
    @ports << Port.new(*args)
  end

  def signal(*args)
    @signals << Signal.new(*args)
  end


  def behavior(&body)
    @behavior = Behavior.new(body)
  end

  def type(*args)
    @types << Type.new(*args)
  end

  def component(*args, &body)
    @components << Component.new(*args, &body)
  end

  def generate(out=$stdout)
    out.puts "ENTITY #{@name} IS"
    out.puts "PORT("
    @ports.each_with_index do |port, index|
      port.generate(out, 1, (index == @ports.length-1))
    end
    out.puts ");"
    out.puts "END #{@name};"
    out.puts "ARCHITECTURE arch_#{@name} OF #{@name} IS"
    @types.each {|t| t.generate(out, 1)}
    @signals.each {|t| t.generate(out, 1)}
    @components.each {|c| c.generate(out, 1)}
    out.puts "BEGIN"
    @behavior.generate(out, 1)
    out.puts "END arch_#{@name};"
  end
end

class Component < MultiLineStatement
  def initialize(name)
    @name = name
    @ports = []
    yield(self)
  end

  def in(name, type)
    @ports << Port.new(name, :in, type)
  end

  def out(name, type)
    @ports << Port.new(name, :out, type)
  end

  def inout(name, type)
    @ports << Port.new(name, :inout, type)
  end

  def generate(out, indent)
    prefix = "  " * indent
    out.puts prefix + "COMPONENT #{@name}"
    out.puts prefix + "PORT("
    @ports.each_with_index do |port, index|
      port.generate(out, indent+1, (index == @ports.length-1))
    end
    out.puts prefix + ");"
    out.puts prefix + "END COMPONENT;"
  end
end

class Type < SingleLineStatement
  def initialize(name, values)
    @name = name
    @values = values
  end

  def line
    "TYPE #{@name} IS ( #{@values.join(", ")} );"
  end
end

class Port < SingleLineStatement
  def initialize(id, direction, description)
    @id = id
    @direction = direction
    @description = description
  end

  def generate(out, indent, last)
    out.print "  " * indent
    out.print "#{@id}: #{@direction} #{@description}"
    out.puts last ? '' : ';'
  end
end

class Signal < SingleLineStatement
  def initialize(id, type)
    @id = id
    @type = type
  end

  def line
    "SIGNAL #{@id} : #{@type};"
  end
end

class Behavior
  include StatementBlock

  def initialize(body)
    @statements = []
    body.call(self)
  end

  def process(inputs, &body)
    @statements << VHDL::Process.new(inputs, body)
  end

  def instance(*args)
    @statements << Instance.new(*args)
  end
```

```
end

class Instance < SingleLineStatement
  def initialize(component, name, *ports)
    @component = component
    @name = name
    @ports = ports
  end

  def line
    "#{@name}: #{@component} PORT MAP(#{@ports.join(', ')});"
  end
end

class Process
  include StatementBlock
  def initialize(inputs, body)
    @inputs = inputs
    @statements = []
    body[self]
  end

  def generate(out, indent)
    prefix = "  " * indent
    args = @inputs.map(&:to_s).join(',')
    out.puts prefix + "PROCESS(#{args})"
    out.puts prefix + "BEGIN"
    @statements.each {|s| s.generate(out, indent + 1)}
    out.puts prefix + "END PROCESS;"
  end
end

class Case
  def initialize(input, body)
    @input = input
    @conditions = {}
    body.call(@conditions)
  end

  def generate(out, indent)
    prefix = "  " * indent
    out.puts prefix+"CASE #{@input} IS"
    @conditions.each do |pair|
      condition, expression = pair
      out.print prefix+"  WHEN "
      if condition =~ /^\d$/
        out.print "'#{condition}'"
      elsif condition =~ /^\d+$/
        out.print "\"#{condition}\""
      else
        out.print condition
      end
      out.print " =>"
      if expression.is_a? InlineStatement
        out.puts expression.generate
      else
        out.puts
        expression.generate(out, indent+1)
      end
    end
    out.puts prefix+"END CASE;"
  end
end

class If < MultiLineStatement
  include StatementBlock

  def initialize(conditions, body)
    @conditions = conditions
    @compound = false
    @statements = []
    body[self]
  end

  def elsif(*conditions, &body)
    unless @compound
      @clauses = [@statements]
      @conditions = [@conditions]
    end
    @compound = true

    @statements = []
    body.call(self)
    @clauses << @statements
    @conditions << conditions
  end

  def else(*conditions, &body)
    @whentrue = @statements
    @statements = []
    body.call(self)
  end

  def generate(out, indent)
    prefix = "  " * indent
    if @compound
      conditions = @conditions.first.map(&:generate).join(' and ')
      out.puts(prefix+"IF #{conditions} THEN")
      @clauses.first.each {|s| s.generate(out, indent+1)}
      @clauses[1..100].zip(@conditions[1..100]).each do |statements, conditions|
        conditions = conditions.map(&:generate).join(' and ')
        out.puts(prefix+"ELSIF #{conditions} THEN")
        statements.each {|s| s.generate(out, indent+1)}
      end
      out.puts(prefix+"END IF;")
    elsif @whentrue
      conditions = @conditions.map(&:generate).join(' and ')
      out.puts(prefix+"IF #{conditions} THEN")
      @whentrue.each {|s| s.generate(out, indent+1)}
      out.puts(prefix+"ELSE")
      @statements.each {|s| s.generate(out, indent+1)}
      out.puts(prefix+"END IF;")
    else
      conditions = @conditions.map(&:generate).join(' and ')
      out.puts(prefix+"IF #{conditions} THEN")
      @statements.each {|s| s.generate(out, indent+1)}
      out.puts(prefix+"END IF;")
    end
  end
end

class Assignment < SingleLineStatement
  def initialize(*args)
    @assign = Assign.new(*args)
  end

  def line
    @assign.generate + ";"
  end
end

class Assign < InlineStatement
  def initialize(*args)
    if args.length == 2
      @target = args[0]
      @expression = args[1]
    else
      @target = args[0].to_s + "(#{args[1]})"
      @expression = args[2].to_s + "(#{args[3]})"
      @expression = @expression.to_sym
    end
  end

  def generate
    "#{@target} <= #{quoted(@expression)}"
  end
end

class Equal < InlineStatement
  def initialize(target, expression)
    @target = target
    @expression = expression
  end

  def generate
    "#{@target} = #{quoted(@expression)}"
  end
end

class Event < InlineStatement
  def initialize(target)
    @target = target
  end

  def generate
```

```
      "#{@target.to_s}'EVENT"
    end
  end

  class Invert < InlineStatement
    def initialize(body)
      @body = body
    end

    def generate
      "NOT (#{@body})"
    end
  end

  class Block < MultiLineStatement
    include StatementBlock

    def initialize(body)
      @statements = []
      body.call(self)
    end
  end

# Global scope methods for creating stuff

  module Helpers
    def entity(name, &body)
      VHDL::Entity.new(name, body)
    end

    def assign(target, expression)
      VHDL::Assign.new(target, expression)
    end

    def high(target)
      assign(target, '1')
    end

    def low(target)
      assign(target, '0')
    end

    def equal(target, expression)
      VHDL::Equal.new(target, expression)
    end

    def event(target)
      VHDL::Event.new(target)
    end

    def block(&body)
      VHDL::Block.new(body)
    end

    def subbits(sym, range)
```

```
      "#{sym}(#{range.first} downto #{range.last})".to_sym
    end

    def invert(body)
      VHDL::Invert.new(body)
    end
  end
end

# Monkeypatching
class Symbol
  def <=(other)
    return assign(self, other)
  end
end

class Fixnum
  def to_logic(width)
    str = self.to_s(2)
    return "0"*(width-str.length) + str
  end
end


def generate_vhdl(entity, out=$stdout)
  out.puts "LIBRARY ieee;"
  out.puts "USE ieee.std_logic_1164.all;"
  out.puts "USE ieee.numeric_std.all;"
  out.puts
  entity.generate(out)
end
```

8 Appendix B: Clojure implementation source code

The source code to Computer.Build is presented here for easy access, but the most recent version can always be found at `http://github.com/epall/Computer.Build` and is licensed under the MIT License.

Listing 10: Computer.Build main module

```
(ns computer-build
  (:use computer-build.vhdl
        computer-build.state-machine
        clojure.set
        clojure.contrib.pprint))

(defmacro build [cpuname options & instructions]
  `(build* ~cpuname ~options (quote ~instructions)))

(defn rd [port]
  (keyword (str "rd_" (name port))))

(defn wr [port]
  (keyword (str "wr_" (name port))))
```

```
(defn binary* [accumulator num]
  (let [last-bit (if (even? num) "0" "1")]
    (cond
      (= 0 num) (str "0" accumulator)
      (= 1 num) (str "1" accumulator)
      true (recur (str last-bit accumulator) (int (/ num 2))))))

(defn binary [width num]
  "Convert number to binary literal format expected in VHDL"
  (let [value (binary* "" num)
        length (count value)]
    (str (apply str (repeat (- width length) "0")) value)))

(defn alu-op-to-opcode [op]
  (if op
    (cond
      (= (name op) "and") "001"
      (= (name op) "complement") "011"
      (= (name op) "+") "100"
      (= (name op) "-") "101"
      (= (name op) "=") "110")
    "000"))

(defn flatten-1 [things]
  (if (empty? things)
    '()
    (if (list? (first things))
      (concat (first things) (flatten-1 (rest things)))
      (cons (first things) (flatten-1 (rest things))))))

(defn mapmap [f m]
  "Replace all values in map m with the result of calling
  f with the value"
  (zipmap (keys m) (map f (vals m))))

(defn rtl-to-microcode [[target _ source & conditional-body]]
  (cond
    (number? source) ; constant-to-register
    {:control-signals (list (wr target)),
     :constant-value source}

    (symbol? source) ; register-to-register
    {:control-signals (list (rd source) (wr target))}

    (= "if" (name target)) ; conditional
    (let [[condition target expectation] _
          body (flatten-1 (map rtl-to-microcode (cons source conditional-body)))]
      (list
        ; load target
        {:control-signals (list (rd target) (wr :alu_a)) }
        ; load expectation and compare
        (if (number? expectation)
          {:control-signals (list (wr :alu_b))
           :constant-value expectation
           :alu_op condition
           :conditional true
           :body body}
          {:control-signals (list (rd expectation) (wr :alu_b))
           :alu_op condition
           :conditional true
           :body body})
        ))

    (and (seq? source) (symbol? (first source))) ; ALU-to-register
    (let [[alu_op operand_a operand_b] source]
      (list
        {:control-signals (list (rd operand_a) (wr :alu_a)) :alu_op alu_op}
        (if (= 3 (count source))
          (if (number? operand_b)
            {:control-signals (list (wr :alu_b)) :alu_op alu_op
              :constant-value operand_b}
            {:control-signals (list (rd operand_b) (wr :alu_b)) :alu_op alu_op}))
        {:control-signals (list :rd_alu (wr target)) :alu_op alu_op}))))

(defn name-for-state [instruction-name index]
  (keyword (str instruction-name "_" index)))

(defn link-state [instruction-name last-index body index]
  (let [next-state
          (if (= index last-index)
            :fetch
            (name-for-state instruction-name (+ index 1)))
        connected-body
          {(name-for-state instruction-name index)
            (assoc body :next next-state)}]
  (if-let [conditional-body (:body body)]
    ; conditional
    (let [instruction-name (str instruction-name "_" index)
          last-index (dec (count conditional-body))]
      (merge connected-body
             (apply merge (map (partial link-state instruction-name last-index)
                               conditional-body (iterate inc 0)))))
    ; not conditional
    connected-body)))

(defn make-states-for-instruction [[_ instruction-name & RTLs]]
  (let [microcode (flatten-1 (map rtl-to-microcode RTLs))
        last-index (- (count microcode) 1)]
    (apply merge (map (partial link-state instruction-name last-index)
                      microcode (iterate inc 0)))))

(defn make-states [instructions]
  "Given a set of instructions, create the set of states
  necessary to execute their microcode"
  (apply merge (map make-states-for-instruction instructions)))

(defn make-opcodes [instructions]
  "Given a set of instructions, assign opcodes to each one in
  a map with keys being instruction names"
  (let [names (map #(nth % 1) instructions)
        width (Math/ceil (/ (Math/log (count names)) (Math/log 2)))
        to-binary (fn [n] (let [raw (Integer/toString n 2)
                               padlength (- width (count raw))]
                           (str (apply str (repeat padlength "0")) raw)))
    op-values (map to-binary (range (count names)))]
    (zipmap names op-values)))

(defn realize-state [control-signals state]
  (let [highs (:control-signals state)
        assertions (map #(list 'high %) highs)
        clears (map #(list 'low %) (difference control-signals highs))]
    (vec (concat
           (:code state)
           assertions
           clears
           (if-let [const (:constant-value state)]
             `((<= :system_bus ~(binary 8 const)))
             `((<= :system_bus ~(apply str (repeat 8 "Z")))))
           `((<= :alu_operation ~(alu-op-to-opcode (:alu_op state))))))))

(defn control-unit [instructions]
  "Given a set of states, make a control unit that will
  execute them"
  (let [opcodes (make-opcodes instructions)
        opcode-width (count (second (first opcodes)))
        static-states {:fetch
                         {:control-signals '(:rd_pc, :wr_MA), :next :store_instruction}
                       :store_instruction
                         {:next :decode,
                          :control-signals '(:rd_MD, :wr_IR, :inc_pc),
                          :code
                          [`(if (and (event :clock) (= :clock 0))
                              (<= :opcode ~(- opcode-width 1) 0
                                  :system_bus 7 ~(- 8 opcode-width)))]}
                       :decode {:control-signals '()}}
        states (merge (make-states instructions) static-states)
        conditional-states
          (select-keys states (for [[k v] states :when (:conditional v)] k))
        unconditional-states
          (select-keys states (for [[k v] states :when (not (:conditional v))] k))
        control-signals (set (apply concat (map
                                            (fn [[_ body]] (:control-signals body))
                                            states)))
        inputs {:reset std-logic, :condition std-logic}
        outputs (assoc
                  (zipmap control-signals (repeat (count control-signals) std-logic))
                  :alu_operation (std-logic-vector 2 0))]

    (list (state-machine "control_unit"
              ; inputs
              inputs
              ; outputs
              outputs
              ; input/outputs
              {:system_bus (std-logic-vector 7 0)}
              ; signals
              { :opcode (std-logic-vector (- opcode-width 1) 0) }
              ; reset
              (list* '(<= :alu_operation "000")
                     '(goto :fetch)
                     '(<= :system_bus "ZZZZZZZZ")
                     (map #(list 'low %) control-signals))
              ; states
              (mapmap (partial realize-state control-signals) states)
              ; transitions
              (concat
               ; states
               (map
                (fn [[k v]] (list k (:next v)))
                (dissoc unconditional-states :decode))
               ; conditional false
               (map
                (fn [[k v]] (list k '(= :condition 0) (:next v)))
                conditional-states)
               ; conditional true
               (map
                (fn [[k v]] (list k '(= :condition 1) (name-for-state (name k) 0)))
                conditional-states)

               ; decode
               (map #(list
                         ; from
                         :decode
                         ; condition
                         `(= :opcode ~(second %))
                         ; to
                         (keyword (str (first %) "_0")))
                       ; for each opcode
                       opcodes)))
          (assoc inputs :clock std-logic) outputs)))

(defn build* [cpuname options instructions]
  (.mkdir (java.io.File. cpuname))
  (let [[control-unit control-in control-out] (control-unit instructions)
        control-signals (concat (dissoc control-in :clock :reset) control-out)
        dynamic-signals (map (fn [[k v]] (list 'signal k v)) control-signals)]
    (with-open [main-vhdl (java.io.FileWriter. (str cpuname "/main.vhdl"))
                       control-vhdl (java.io.FileWriter. (str cpuname "/control.vhdl"))]
          (pprint dynamic-signals)
          (binding [*out* control-vhdl]
        (generate-vhdl control-unit))
          (binding [*out* main-vhdl]
        (generate-vhdl `(entity "main"
          ; ports
          [(:clock :in ~std-logic)
           (:reset :in ~std-logic)
           (:bus_inspection :out ~(std-logic-vector 7 0))]
          ; defs
          [~@dynamic-signals
           (signal :system_bus ~(std-logic-vector 7 0))
```

```
(component :reg
  (:clock :in ~std-logic)
  (:data_in :in ~(std-logic-vector 7 0))
  (:data_out :out ~(std-logic-vector 7 0))
  (:wr :in ~std-logic)
  (:rd :in ~std-logic))

(component :program_counter
  (:clock :in ~std-logic)
  (:data_in :in ~(std-logic-vector 7 0))
  (:data_out :out ~(std-logic-vector 7 0))
  (:wr :in ~std-logic)
  (:rd :in ~std-logic)
  (:inc :in ~std-logic))

(component :ram
  (:clock :in ~std-logic)
  (:data_in :in ~(std-logic-vector 7 0))
  (:data_out :out ~(std-logic-vector 7 0))
  (:address :in ~(std-logic-vector 4 0))
  (:wr_data :in ~std-logic)
  (:wr_addr :in ~std-logic)
  (:rd :in ~std-logic))

(component :alu
  (:clock :in ~std-logic)
  (:data_in :in ~(std-logic-vector 7 0))
  (:data_out :out ~(std-logic-vector 7 0))
  (:op :in ~(std-logic-vector 2 0))
  (:wr_a :in ~std-logic)
  (:wr_b :in ~std-logic)
  (:rd :in ~std-logic)
  (:condition :out ~std-logic))

(component :control_unit
  ~@(concat (map input control-in)
            (map output control-out)
            `((:system_bus :inout ~(std-logic-vector 7 0)))))]
; architecture
[
(instance :program_counter "pc" :clock :system_bus :system_bus
          :wr_pc :rd_pc, :inc_pc)
; instruction register
(instance :reg "ir" :clock :system_bus :system_bus :wr_IR :rd_IR)
; accumulator
(instance :reg "A" :clock :system_bus :system_bus :wr_A :rd_A)
(instance :ram "main_memory" :clock :system_bus :system_bus
          ~(subbits :system_bus 4 0) :wr_MD :wr_MA :rd_MD)
(instance :alu "alu0" :clock :system_bus :system_bus :alu_operation
          :wr_alu_a :wr_alu_b :rd_alu, :condition)
(instance :control_unit "control0"
          ; same ports as the control signals we got
          ~@(map first (concat control-in control-out)) :system_bus)
(<= :bus_inspection :system_bus)
```

```
          ])))))))
```

Listing 11: State machine generator

```
(ns computer-build.state-machine
  (:use computer-build.vhdl)
  (:require [clojure.zip :as zip])
  (:refer-clojure :rename {:name :keyword-to-str}))

(defn reformat-ports [ports inout]
  (if (map? ports)
    (map #(list (first %) inout (second %)) ports)
    (map #(list % inout "std_logic") ports)))

(defn state-from-name [statename]
  (keyword (str "state_" (keyword-to-str statename))))

(defn rewrite-gotos [state-variable block]
  "Given a set of statements, replace all gotos with assignments
  to the appropriate state variable"
  (vec (map #(if (= 'goto (first %))
                 (list '<= state-variable (state-from-name (second %)))
                 %)
            block)))

(defn flatten-states [m]
  (if (empty? m) '()
    (let [[state body] (first m)]
      (list* (state-from-name state) body (flatten-states (dissoc m state))))))

(defn translate-transition [state-variable transition]
  (if (= (count transition) 2)
  `(if (= ~state-variable ~(state-from-name (first transition)))
     [(<= ~state-variable ~(state-from-name (last transition)))])
  `(if (and (= ~state-variable ~(state-from-name (first transition)))
            ~(second transition))
     [(<= ~state-variable ~(state-from-name (last transition)))])))

(defn state-machine [name inputs outputs inouts signals reset states transitions]
        "Create a state machine that operates from the given inputs, triggering
        the specified transitions to the listed states that generate outputs on
        the signals in the outputs array"
  (let [inports (reformat-ports inputs :in)
        outports (reformat-ports outputs :out)
        inoutports (reformat-ports inouts :inout)]
        `(entity ~name
          ; Ports
          ~(concat ['(:clock :in "std_logic")] inports outports inoutports)
          ; Definitions
          ((deftype "STATE_TYPE"
                    ~(map #(str "state_" (keyword-to-str %)) (keys states)))
          (signal :state "STATE_TYPE")
          ~@(map #(cons 'signal %) signals))
          ; Behavior
          (process (:clock :state :reset)
                   [(if-else (= :reset "1")
                          ; true body
                          ~(rewrite-gotos :state reset)
                          ; false body
                          [
                           (case :state ~@(flatten-states states))
                           (if (and (event :clock) (= :clock 1))
                           ~(vec (map (partial translate-transition :state) transitions)))
                          ])]))))
```

Listing 12: VHDL-in-Clojure to VHDL compiler

```
(ns computer-build.vhdl
  (:use clojure.contrib.str-utils))

(def std-logic "STD_LOGIC")

(defn std-logic-vector [start end]
  (str "STD_LOGIC_VECTOR(" start " downto " end ")"))

(defn subbits [k start end]
  (keyword (str (name k) "(" start " downto " end ")")))

(defn input [[id kind]]
  (list id :in kind))

(defn output [[id kind]]
  (list id :out kind))

(defn indented-lines [strings] (map (partial str "  ") strings))

(defn indent-lines [[line & lines]]
  (if (empty? line) '()
    (if (string? line)
      (cons line (indent-lines lines))
      (concat
          (if (:noindent (meta line))
              (indent-lines line)
              (indented-lines (indent-lines line)))
          (indent-lines lines)))))

(defn flat-list [head & tail]
  (let [tail (if (empty? tail) '() (apply flat-list tail))]
    (if (seq? head)
      (concat head tail)
      (cons head tail))))

(defn spaces [& strings] (str-join " " strings))

(defn commaify [lines]
    (map #(apply str %)
         (partition 2 (concat (interpose ";" lines) [""]))))

(defn keyword-to-str [sym]
  (cond
    (keyword? sym) (name sym)
    (number? sym) (str \' sym \')
    (string? sym) (if (= (count sym) 1)
      (str \' sym \')
      (str \" sym \"))))

(defmulti to-vhdl (fn [block]
  (if (symbol? block) (throw (Error. (str "Invalid VHDL: " block))))
  (if (vector? block) :block
    (-> block first name keyword))))

(defn not-indented [body]
  (with-meta body {:noindent true}))

; Multi-line statement that causes an extraneous level of nesting in AST
(defmacro def-vhdl-multiline [kword bindings & body]
  `(defmethod to-vhdl ~kword [~(vec (concat '[_] bindings))]
     (not-indented (list ~@body))))

; Inline or single-line statements that don't produce a list of lines
(defmacro def-vhdl-inline [kword bindings & body]
  `(defmethod to-vhdl ~kword [~(vec (concat '[_] bindings))]
     (str ~@body)))

(defmethod to-vhdl :default [arg] (str "###UNIMPLEMENTED: " (first arg) "###"))

(defmethod to-vhdl :block [block]
  (map to-vhdl block))

(defmethod to-vhdl :entity [[type name ports defs & architecture]]
  (apply str (interpose "\n" (indent-lines
  (list
    (spaces "ENTITY" name "is")
    "PORT("
    (commaify (map to-vhdl (map (partial cons :port) ports)))
    ");"
    (str "END " name ";")

    (str "ARCHITECTURE arch_" name " OF " name " IS")
    (map to-vhdl defs)
    "BEGIN"
    (map to-vhdl architecture)
    (str "END arch_" name ";"))))))

(def-vhdl-multiline :process [ports definition]
    (str "PROCESS(" (str-join "," (map keyword-to-str ports)) ")")
    "BEGIN"
    (to-vhdl definition)
    "END PROCESS;")

(def-vhdl-multiline :case [target & cases]
    (spaces "CASE" (keyword-to-str target) "IS")
    (map #(not-indented (list
      (spaces "WHEN" (keyword-to-str (first %)) "=>")
      (to-vhdl (second %)))) (partition 2 cases))
    "END CASE;")
```

```
(def-vhdl-multiline :if [condition body]
    (spaces "IF" (to-vhdl condition) "THEN")
    (to-vhdl body)
    "END IF;")

(def-vhdl-multiline :if-else [condition truebody falsebody]
    (spaces "IF" (to-vhdl condition) "THEN")
    (to-vhdl truebody)
    "ELSE"
    (to-vhdl falsebody)
    "END IF;")

(def-vhdl-multiline :if-elsif [condition body & clauses]
    (spaces "IF" (to-vhdl condition) "THEN")
    (to-vhdl body)
    (not-indented
      (map #(not-indented (list
        (spaces "ELSIF" (to-vhdl (first %)) "THEN")
        (to-vhdl (second %)))) (partition 2 clauses)))
    "END IF;")

(def-vhdl-multiline :component [name & ports]
  (str "COMPONENT " (keyword-to-str name))
  "PORT("
  (commaify (map to-vhdl (map (partial cons :port) ports)))
  ");"
  (str "END COMPONENT;"))

; inline / single-line statements

(defmethod to-vhdl :<= [[type & args]]
  (let [target-str (keyword-to-str (first args))]
    (cond
      (= (count args) 2)
        (let [[target expression] args]
          (str target-str " <= " (keyword-to-str expression) \;))
      (= (count args) 4)
        (let [[target target-index source source-index] args]
          (str target-str "(" target-index ") <= "
               (keyword-to-str source) "(" source-index ");"))
      (= (count args) 6)
        (let [[target target-start target-end source source-start source-end] args]
          (str target-str "(" target-start " downto " target-end ") <= "
               (keyword-to-str source) "(" source-start " downto " source-end ");")))))

(def-vhdl-inline :port [id direction kind]
  (keyword-to-str id) ": " (keyword-to-str direction) " " kind)

(def-vhdl-inline :instance [component name & mappings]
  name ": " (keyword-to-str component) " PORT MAP("
                 (str-join ", " (map keyword-to-str mappings)) ");")

(def-vhdl-inline :low [target]
  (to-vhdl `(<= ~target "0")))
```

```
(def-vhdl-inline :high [target]
  (to-vhdl `(<= ~target "1")))

(def-vhdl-inline :signal [sig kind]
  "SIGNAL " (name sig) " : " kind ";")

(def-vhdl-inline :deftype [name values]
  "TYPE " name " IS (" (str-join ", " values) ");")

(def-vhdl-inline :and [& conditions]
  (str-join " AND " (map to-vhdl conditions)))

(def-vhdl-inline :event [target]
  (keyword-to-str target) "'EVENT")

(def-vhdl-inline := [condA condB]
  (name condA) " = " (keyword-to-str condB))

;;;;;;;;;;;;;;; Final output generation ;;;;;;;;;;;;;;

(defn generate-vhdl [& entities]
  (do
    (println "LIBRARY ieee;")
    (println "USE ieee.std_logic_1164.all;")
    (println (to-vhdl (first entities)))))
```

9 Appendix C: Program output

While the goal of this paper and project was to compare Ruby and Clojure for a non-trivial programming task, the actual Computer.Build program successfully generates complex microprocessors from instruction set definitions. The generated designs were compiled for an Altera FPGA using Altera Quartus, and their behavior was simulated. The generated hardware design and test waveforms for one of the test processors are included in this Appendix to demonstrate the successful operation of Computer.Build.

9.1 Synthesized RTL

Computer.Build produces VHDL source code as its output. Being an intermediate step, this source code is excessively verbose and not fit for inclusion in this paper. However, the source code is compiled into a schematic design by Quartus, and the relevant schematics are presented in this section.

Figure 1: Top-level CPU design

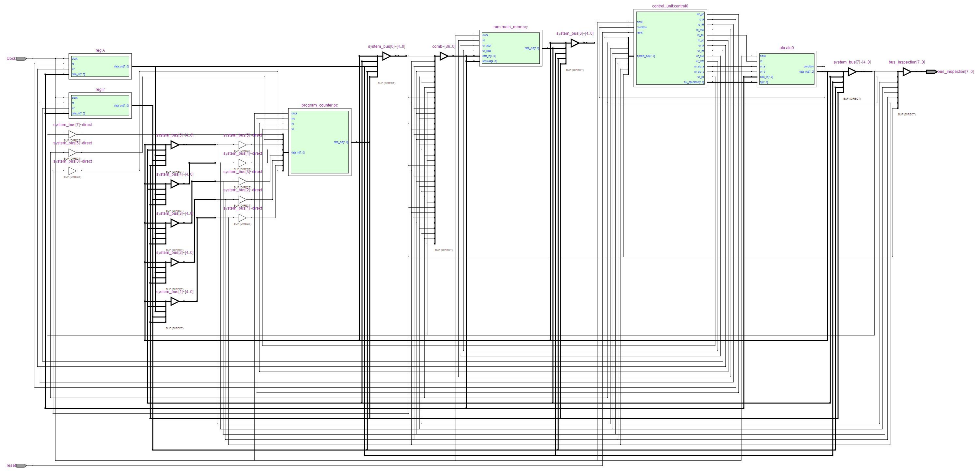
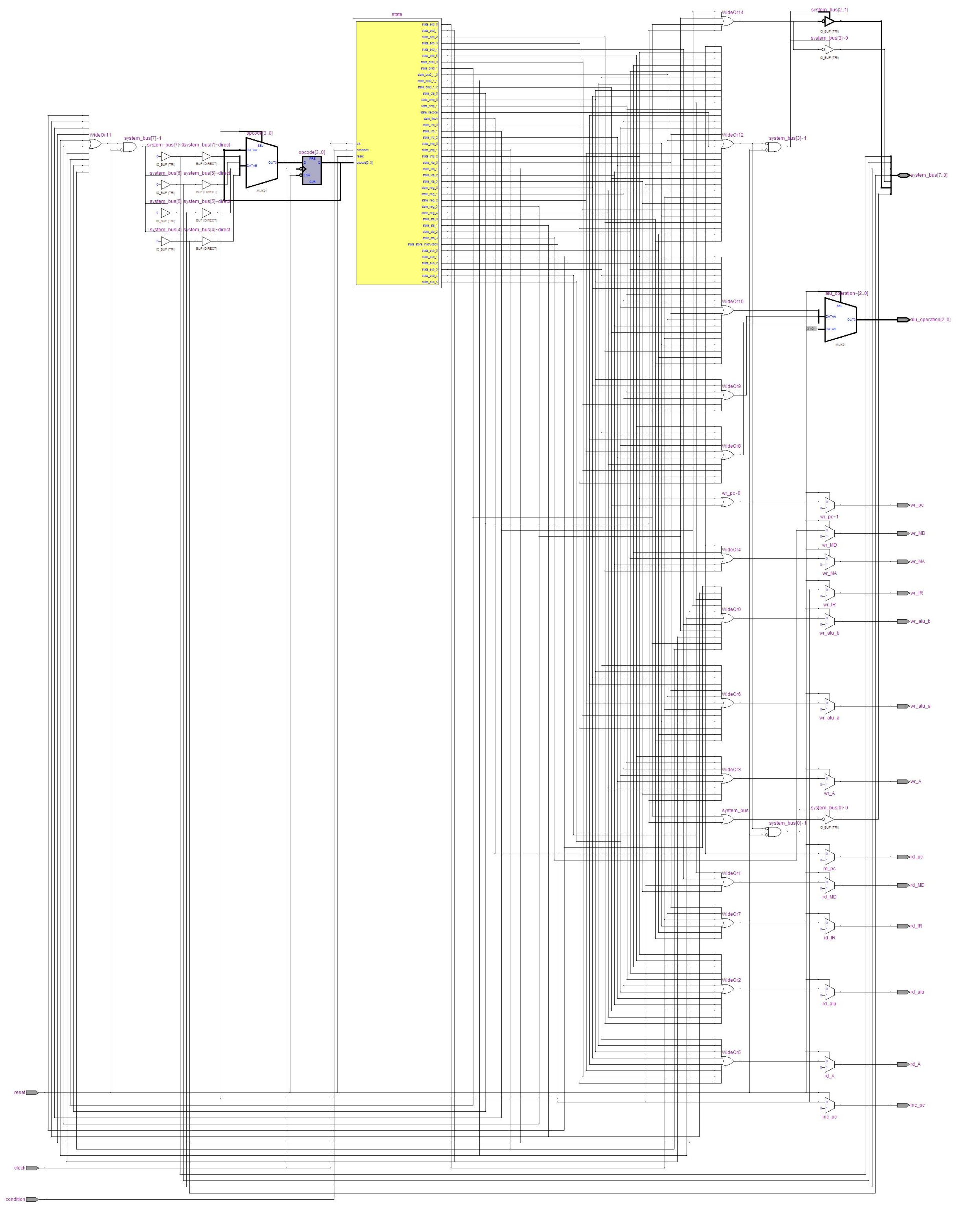


Figure 2: Control unit



9.2 Simulation waveforms

After compilation, the processor was tested using a simple program that exercised the critical instructions. The program loads the value 0xFB into the A register, then increments it until it reaches 0. When it reaches 0, the program branches back to the initialization and re-loads 0xFB. This demonstrates the arithmetic and conditional branching capabilities of one of the synthesized processors.

Figure 3: High-level waveform

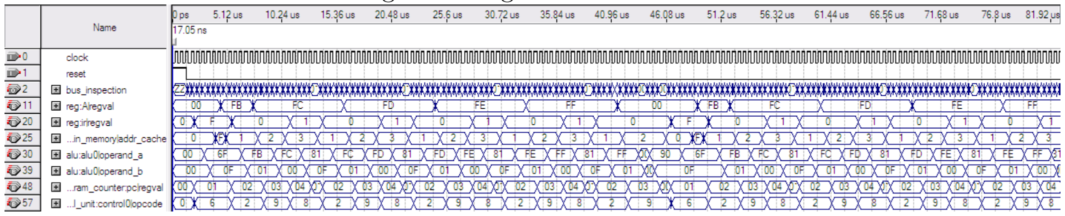
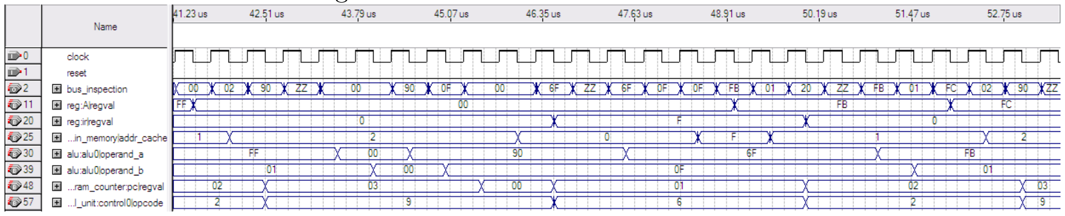


Figure 4: Focus on conditional branch



References

[1] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[2] Cedric Lemaire. Codeworker: A universal parsing tool & a source code generator, 2008. `http://codeworker.free.fr/index.html`.

[3] Martin Thiede. Rgen: Ruby modelling and code generation framework. *InfoQ*, February 2009. `http://www.infoq.com/articles/thiede-ruby-modelling`.

[4] Joel VanderWerf. Cgenerator: A framework for generating c extensions from ruby, 2001. `http://cgen.rubyforge.org/`.

[5] Heinz Knutze Wolfgang Goerigk, Ulrich Hoffmann. Clicc - the common lisp to c compiler, 1996. `http://www.informatik.uni-kiel.de/ wg/clicc.html`.