Domain-Specific Language for HW/SW Co-Design for FPGAs

Jason Agron

Dept. of Computer Science and Computer Engineering
University of Arkansas
504 J.B. Hunt Building, Fayetteville, AR
{jagron}@uark.edu

Abstract. This article describes FSMLanguage, a domain-specific language for HW/SW co-design targeting platform FPGAs. Modern platform FPGAs provide a wealth of configurable logic in addition to embedded processors, distributed RAM blocks, and DSP slices in order to help facilitate building HW/SW co-designed systems. A technical challenge in building such systems is that the practice of designing software and hardware requires different areas of expertise *and* different description domains, i.e. languages and vocabulary. FSMLanguage attempts to unify these domains by defining a way to describe HW/SW co-designed systems in terms of sets of finite-state machines – a concept that is reasonably familiar to both software programmers and hardware designers. FSMLanguage is a domain-specific language for describing the functionality of a finite-state machine in such a way that its implementation can be re-targeted to software or hardware in an efficient manner. The efficiency is achieved by exploiting the resources found within modern platform FPGAs – namely the distributed RAM blocks, soft-core processors, and the ability to construct dedicated communication channels between FSMs in the reconfigurable fabric. The language and its compiler promote uniformity in the description of a HW/SW co-designed system, which allows a system designer to make partitioning and implementation strategy decisions later in the design cycle.

1 Introduction

Modern platform FPGAs provide both a "sea" of logic gates dedicated towards implementation of custom hardware, as well as typical general purpose processors that can be programmed using traditional software techniques [**?**]. Typically the hardware portion of such a system is described in an hardware design language (HDL) and the software portion is described in a traditional software language, often C or C++. The dichotomy of these two paradigms makes it difficult to program, reason about, and debug systems built on platform FPGAs especially when the pieces of a system are expressed in fundamentally different terms. Unifying these descriptions of software and hardware gives promise to be able to more effectively build and reason about these hardware/-software co-designed systems. Domain-specific languages can be used to construct a uniform level on which to express such systems. FSMLanguage is a domain-specific language for describing finite-state machines that can be efficiently compiled to either

software- or hardware-implementations. The language includes abstractions for memories and communication channels that allow a designer to programmatically describe a system's communication infrastructure while making efficient use of the resources on the platform FPGA.

1.1 Motivation

In general, finite-state machines (FSMs) provide a concise vocabulary to describe a set of behaviors using the notion of states, transitions, and actions [**?**,**?**,**?**]. The concept of what an FSM is and does is familiar to both software programmers and hardware designers, thus making it a realistic and well-suited abstraction for describing HW/SW co-designed systems.

The main purpose of FSMLanguage is to provide a common form, or single source language, to describe a HW/SW co-designed system. Programs written in FSMLanguage can be compiled to different implementation targets, currently VHDL and C, and the different targets are able to communicate and synchronize with one another using abstractions built into the language itself. The available abstractions include shared memories, local memories, and FIFO channels all of which are readily available on modern FPGA architectures. A programmer is able to make use of these structures in FSMLanguage without having to understand the low-level protocols and timing requirements normally associated with HW/SW co-design. This has two important, positive effects: (1) it unifies system descriptions - allowing them to be looked at as a cohesive whole, and (2) it eliminates the need for a programmer or designer to manually integrate system components [**?**].

Currently, many other projects use C-like languages for HW/SW co-design [**?**,**?**,**?**,**?**]. However, these languages differ widely in how they support generation of hardware implementations. Some languages are merely HDLs that offer a C-like syntax, while others provide true support for mapping software constructs to hardware implementations. In general, all of these languages target HW/SW co-design, but each focuses on different sub-domains within the area.

The "look" of C combined with variance in the amount and function of existing C operators has a tendency to make these languages confusing for both software and hardware programmers. The main problem is that these languages advertise themselves as being a traditional software-like language, however the actual semantics, best practices, and restrictions are in truth very different [**?**,**?**].

The C language, while widely considered a high-level general-purpose language, is inherently a domain-specific language for sequential processors; and transforming it into a hardware/software co-design language has proven to be a great challenge [**?**]. Another approach would be to find a middle-ground between software and hardware development practices. FSM-based design is common in both software and hardware design, which makes it a suitable candidate for HW/SW co-design environments. This is the primary motivation for creating an FSM-based hardware/software co-design environment.

1.2 Goal

When compared to general-purpose languages, domain-specific languages (DSLs) are often more narrowly focused and provide a way to concisely describe a limited problem type [**?**]. The use of a smaller, concise language can lead to a much more clear description of a program or system; leading to a more re-targetable representation, as more high-level information about the "intent" of the program exists. This has 2 major positive effects: (1) it makes it easier for a compiler writer to create multiple, compatible implementations of programs written in a DSL, (2) while also making it easier for a programmer to describe their problem within the context of the DSL [**?**].

FSMLanguage was developed to provide a common form for describing components in HW/SW co-designed systems. Thus it provides a way for software programmers and hardware designers to program in the same language, while allowing implementations of such programs to target either side of the hardware/software boundary. This eliminates the need for programmers and system designers to create custom middleware to link hardware and software, which is a time-consuming and error-prone task [**?**]. Additionally, FSMLanguage does not have to be solely used as a tool for programmers. It can be used as a compiler target, or intermediate format, itself. Allowing higher-level imperative and functional languages to become re-targetable to both hardware and software-based implementations.

2 Background

An FSM-based description was chosen as it is quite easy for software programmers and hardware designers to comprehend. Additionally it provides a way to write programs that have both sequential (series of states) and parallel (within one state) sections of code. FSMs are also easily visualizable [**?**] in terms of control-flow graphs (CFGs) and fit very easily into static analysis frameworks (control flow, reachability, liveness, etc.). Many other specification tools have used an FSM-based approach, as it is a natural way of describing systems [**?**].

DOT2VHDL [**?**] is a tool that translates graphical representations of FSMs in DOT to VHDL via the KISS2 intermediate format. The DOT descriptions supported by this tool do not support higher-level arithmetic (only pure Boolean assignment) which compared to a modern HLL or HDL, is extremely limiting. FPGA-vendors have developed their own graphical FSM entry tools, such as Xilinx's StateCAD or Altera's Max+II Graphical Editor, however these tools only target hardware implementations of specified FSMs (in the form of VHDL, Verilog, and ABEL). Additionally, none of these tools provide programmer abstractions for using the available embedded IP cores and memory blocks present in modern FPGAs.

StateWorks [**?**] uses the concept of VFSMs [**?**,**?**], or Virtual Finite State-Machines, to model software systems. The StateWorks development environment provides programmers with a way to specify a system composed of sets of VFSMs. The environment also includes a run-time system for executing system models, however StateWorks does not provide a way to generate executable code for use outside of the development environment. Deployment of a StateWorks model requires the use of the StateWorks

run-time system along with additional programmer-specified libraries for I/O and user-interface routines.

AsmL, or the Abstract State-Machine Language, was developed by Microsoft Research [?] for use in system and components modeling as well as executable specifications. The asmL toolset is mainly focused on robust modeling, visualization, and analysis tools, but an increasing number of features geared towards executable specifications are appearing [?]. While asmL programs can now be executed via interpretation in Haskell [?], no public tools exist that produce re-targetable implementations of asmL programs.

These tools are all useful for modeling portions of systems, but modeling does not lead directly to system implementations. On the other hand, FSMLanguage is capable of system modeling as well as targeting *embedded* platforms, namely modern FPGA offerings from Xilinx. This allows programmers and system designers to use FSMLanguage as an implementation tool that has the potential to replace traditional tools that are less flexible. The focus of the aforementioned toolsets encompasses a large area, however this area does not focus on re-targetable compilation, nor does it include the breadth of operators and abstractions that are currently available in platform FPGAs.

3 Design

3.1 Simplification Through Mechanization

Hardware Description Languages, or HDLs, are one of the most prevalent languages used to describe FSMs destined for hardware implementation. Languages such as VHDL, Verilog, and ABEL are capable of behaviorally describing both the interface and function of a FSM. HDL descriptions of FSMs are executable in that they can be run in a simulation environment, or synthesized into an executable component for use in an FPGA or ASIC implementation. HDLs, in general, are designed to allow programmers to describe arbitrary types of hardware components. Therefore most HDLs contain many more features than those required to describe FSMs. Although FSMs are straight-forward to describe in most HDLs, the complexity and verbosity of HDLs can make them cumbersome to use. For instance, adding a single new state variable to an FSM in VHDL requires the following modifications (assumes a 2-process FSM [?]):

- Definition of a new signal (or set of signals – current and next.)
- Modifying the sensitivity list of the synchronous transition process (STP).
- Modifying the sensitivity list of the asynchronous logic process (ALP).
- Adding a default value assignment to the ALP.
- Adding reset behavior and transition behavior for the signal in the STP.

Unfortunately, any errors or omissions made when performing these modifications do not necessarily result in compilation/synthesis errors. However, mistakes can result in incorrect simulation results, improper reset behavior, and improper logic inference (latches vs. registers). Even worse, HDL tool vendors have different requirements for describing FSMs [?,?]. The verbosity of HDLs paired with the chances to make unnoticed coding mistakes makes FSM description a prime candidate for a domain-specific

language (DSL). The DSL can improve programmer efficiency by reducing code size, clutter, and verbosity. Additionally the language can support ”correct-by-construction” FSM design, thereby making it impossible to make many of the mistakes that can be made when coding FSMs in HDLs.

3.2 The FSM Domain-Specific Language

FSMLanguage is a domain-specific language (DSL) for describing finite-state machines (FSMs). The language, and its associated compiler, targets the configurable logic, embedded memories, and soft-core processors that can be found in modern platform FPGAs [**?**]. FSMLanguage eliminates the need for a programmer to manually control sensitivity lists, state enumerations, FSM reset behavior, and FSM default output behavior. The language also guarantees that all FSM state variables are correctly inferred as flip-flops which greatly improves the resource utilization and timing characteristics of an FSM. The resulting FSM description is much smaller, and less cluttered, than equivalent code written in an HDL. Additionally, the FSMLanguage compiler is re-targetable – currently capable of producing FSM implementations for software (in C) and hardware (in VHDL). The hardware and software implementations generated by the FSMLanguage compiler are compatible with one another in that the language's built-in abstractions for communication are able to operate across the hardware/software boundary.

The program constructs that allow this form of transparent communication include primitives for declaring and accessing channels and memories. Channels are bi-directional communication channels that can be used to connect FSMs together in a CSP-style framework [**?**]. Memories are dual-ported array-like structures that can be internal or external to an FSM. An internal memory can only be used by a single FSM, as the FSM has control over both of the memory's ports, while an external memory can be shared among FSMs (each FSM has access to a single memory port). The memory primitives are mapped directly onto the embedded Block RAM (BRAM) components found within platform FPGAs, whereas the channel primitives are implemented using independent FIFO-based Fast-Simplex Links (FSLs) [**?**]. FSLs and BRAMs are often used within an HDL environment, however in this case, a programmer is responsible for correctly implementing the access protocol required by such components. FSMLanguage provides a clear and easy-to-use syntax for accessing these components, so the programmer need not worry about the access protocols of such components. For instance when interacting with RAMs, one must account for the memory access latency (which may be technology specific) after performing a read operation in order to get a valid result. Also, when reading from a FIFO one must ensure that data exists before continuing on with the rest of the program. These problems can be avoided through mechanization, as FSMLanguage contains special syntactic constructs that can be elaborated by the compiler into the correct access protocols.

3.3 Example - Memory Access

FSMLanguage programs specify memory reads/writes using an array-like syntax familiar to both software and hardware designers. This syntax makes the intent of a memory read/write more clear by making the operation wholly visible in a single line of code

within the FSM, as opposed to several lines and states within a traditional VHDL-based FSM as shown in Figure 1. The FSMLanguage syntax is not only more simple, but it also prevents the programmer from making timing and synchronization mistakes. These mistakes are common in VHDL as the programmer is responsible for defining idle states for handling BRAM latencies. It is important to note that the VHDL snippet shown does not include the extra code needed to enumerate extra idle states, nor the default signal assignments involved with de-asserting BRAM control signals.

```
state1 -> state2 where
{
    a'   <= my_mem[x];
}
```

(a) FSMLanguage Memory Read Syntax

```
when state1 =>
  my_mem_addr          <= x;
  my_mem_read_enable   <= '1';
  next_state           <= state1_int;

when state1_int =>
  next_state <= state1_final;

when state1_final =>
  a   <= my_mem_data_out;
  next_state <= state2;
```

(b) VHDL Memory Read Syntax

Fig. 1. Memory Read Syntax Comparison

3.4 FSMLanguage Programs

The structure and syntax of an FSMLanguage program is shown in Figure 2. FSMLanguage programs consist of the following 10 sections:

- State Names - internal names for FSM state variables
- Generics - compile time variables
- Ports - inputs/outputs from/to the outside world
- Connections - permanent connections of output ports to FSM signals
- Memories - internal/external memory blocks
- Channels - FIFO ports to the outside world
- Signals - internal FSM state
- Initial - initial state definition for the FSM
- Transitions - logic/behavior of an FSM
- VHDL - optional section for linking in libraries of native VHDL constructs

The body, or logic, of an FSM is fully described within the *Transitions* section, while the remainder of the sections are used for declarations for the FSM program itself, such as for memories, ports, and local FSM signals. The statements contained inside of a transition must be assignment statements, in which the left-hand side (LHS) is either a signal, a memory, or a channel, and the right-hand side (RHS) is an expression. Expressions can be composed of data accesses: signals, input ports, memories, channels;

```
-- **** Internal state signal names ****
CS: <current_state_signal_name>;
NS: <next_state_signal_name>;

-- **** Generics (compile-time vars) ****
GENERICS:
  (<genName>, <type>, <static_value>;)*

-- **** Input/Output ports ****
PORTS:
  (<portName>, <in|out>, <type>;)*
CONNECTIONS:
  (<outputPortName>   <=  <rhs>;)*

-- **** Definitions of memories ****
MEMS:
 (<mName>,
 <dataWidth>, <addrWidth> [,EXTERNAL];)*

-- **** Definitions of FIFO channels ****
CHANNELS:
  (<channelName>, <dataWidth>;)*

-- **** Internal FSM signals ****
SIGS:
  (<sigName>, <type>;)*

-- **** Internal State Definition  ****
INITIAL: <stateName>;

-- *** Definition of logic/transitions ****
TRANS:
(<curr_st> [|<bool_guard>] -> <next_st>
[where
{
    (<lhs>  <= <rhs>;)*}
])*

-- **** Native VHDL Defs. ****
VHDL: <un-parsed VHDL code>
```

Fig. 2. FSMLanguage Program Structure and Syntax

arithmetic operators: addition, subtraction, multiplication, division-by-a-constant, bit-concatenation (&), and bit-slicing; boolean operators: and, or, not, xor; as well as function calls. Function calls enable FSMLanguage programs to access external libraries of code, whether in C or VHDL. This allows programmers to extend the abilities of the language, by encapsulating new operations within VHDL and C functions.

FSMLanguage uses a Mealy machine model in which FSM outputs depend on both the current FSM state as well as the FSM inputs. State transitions are defined as atomic guarded transitions; where the entire body of a transition is executed atomically when the guard statement found to be true. The syntax for defining a state transition can be seen in Figure 3. Multiple guarded transitions can be defined for a single start state so that a single given state can have multiple behaviors based off of inputs or internal FSM state. The transitions are prioritized by the compiler according to their order of appearance in an FSMLanguage program, allowing a programmer to tune the precedence of the transitions. Guard expressions must be boolean, and are not allowed to contain memory/channel accesses, however, the result of such an access can be used within a guard expression.

```
<cur_st> [|<bool_guard>] -> <next_st>
[where
{
    (<lhs>  <= <rhs>;)*
}]
```

Fig. 3. FSMLanguage Guarded Transition Syntax

The *Generics* section allows a programmer to define a set of generics, or compile-time variables, that can be used as static constants throughout an FSMLanguage program. These generics are identical to the generics found in VHDL, which can be used to change the width and size of internal variables and ports at compile time.

The *Ports* and *Connections* sections allow a programmer to define dedicated input and output ports to the outside world. This section is commonly used to provide a way to connect external inputs and outputs, such as control signals, to an FSM. *Ports* provide external I/O connections to the outside world, while *connections* provide the ability to tie internal FSM signals to output ports. The connections are very similar to concurrent assignment in VHDL in that they allow an output port to be constantly driven by the internals of the FSM. Output ports are not able to be driven directly from within the body of the FSM. Instead, output ports are driven indireclty by a signal that is tied to a given port via a connection.

```
x'   <= m[a] + 1; // Memory read
m[a] <= x + 2;    // Memory write
```

Fig. 4. FSMLanguage Memory Reads and Writes

```
x' <= #c;     // Channel Read
#c <= x + 2; // Channel write
```

Fig. 5. FSMLanguage Channel Reads and Writes

The *Memories* and *Channels* sections allow a programmer to declare individual memories and channels to be used within the body of their FSM. Memories can be declared with a special *EXTERNAL* keyword to indicate that the memory is shared with another object, therefore only one of the ports of a dual-ported Block RAM will be used by this FSM. The syntax for accessing a location in a memory is identical to the syntax for accessing arrays in C by pairing a memory name with a square-bracketed index. Memory reads and writes are differentiated by the location of the access itself. If the access is on the left-hand side of an assignment statement, then it is a write, while a read would have a memory access on the right-hand side of an assignment statement as shown in Figure 4. Channels also support read/write operations by using a special syntax. Channel accesses are similar to memory accesses, in that the type of access is determined by the location of the access in an assignment statement as shown in Figure 5. Channels allow FSMs to communicate with one another via message-passing, and the channel-specific syntax in FSMLanguage is used to highlight states in which inter-FSM communication occurs. The channel abstraction allows FSMs to be composed in a CSP-style framework [**?**].

The *Signals* section of an FSMLanguage program is used to define the internal state variables of the finite-state machine. Definitions for a signal include the signal's name and data type as shown in Figure 2. A programmer defines the initial state of the FSM in this *Initial* section. The state name defined as *initial* is used to provide well-defined FSM behavior during reset and start up.

The *VHDL* section is primarily targeted to users wanting to develop hardware-only implementations of FSMLanguage programs. This section is dedicated for hand-written VHDL functions, procedures, and signal definitions. This allows programmers to take advantage of some of the more advanced features of VHDL in a library-like fashion. FSMLanguage supports a function call syntax that allows the body of an FSMLanguage program to call VHDL library functions (such as conv_std_logic_vector) defined in the *VHDL* section, VHDL standard libraries, or even in a user-defined library.

A concise state machine definition is formed by combining all of the individual FSMLanguage sections. The FSMLanguage compiler can use this description as input to generate hardware and software implementations of the program, suited for execution on Platform FPGAs.

4 Implementation

The FSMLanguage compiler is implemented in Haskell using the Parsec monadic parser library [**?**,**?**]. Haskell lends itself very well to language and compiler development as Haskell data structures are able to easily and directly represent BNF-style grammars. The compiler is composed of a parser, a VHDL back-end, a C back-end, and a CFG back-end for producing DOT visualizations of FSMs [**?**]. Currently. the entire FSMLanguage compiler implementation has been performed in less than 5,000 lines of Haskell code.

4.1 VHDL Compiler Back-End

The VHDL back-end for FSMLanguage builds a synchronous FSM in VHDL using a basic 2-process model. The generated code handles reset logic as well as transition logic for the programmer. Additionally, the structure of the generated code ensures that all FSM signals are inferred as registers and not latches which greatly improves the timing of an FSM during synthesis. The memory constructs of FSMLanguage are directly elaborated into the correct read and write access protocols to the Block RAMs (BRAMs) internal to Xilinx Platform FPGAs. The channel constructs of FSMLanguage are transformed into FIFO interfaces that are compatible with the Fast-Simplex Link (FSL) standard from Xilinx [**?**].

The memory constructs of FSMLanguage are directly elaborated into read/write accesses to the Block RAMs (BRAMs) internal to Xilinx Platform FPGAs. The generated VHDL abides by the BRAM access protocol and is capable of using both ports of dual-ported BRAMs simultaneously allowing parallel reads/writes from the same memory. Additionally, every memory in an FSMLanguage program can be accessed in parallel as these memories are implemented with physically different BRAMs within the FPGA.

The BRAMs embedded in Xilinx FPGAs have 2 clock-cycle read latency, and 1 clock-cycle write latency, which allows a programmer to access data with very low overhead, and no jitter.

The channel constructs of FSMLanguage are transformed into FIFO interfaces that are compatible with the Fast-Simplex Link (FSL) standard from Xilinx [**?**]. These interfaces provide FIFO status signals which aid in the construction of both blocking and non-blocking channel accesses. By default, the FSMLanguage compiler produces blocking reads and writes, however the language does allow a programmer to query a channel interface to see if the channel has data (data exists) or if the channel is full. The ability to query a channel allows a programmer to construct non-blocking read and write operations in an application-specific way.

All VHDL produced by the compiler uses inference templates for instantiated objects such as FSMs, BRAMs, and FSLs. The templates are architecture-independent, behavioral VHDL code that can be altered individually. This allows the compiler to be tuned for different FPGA architectures, chip families, synthesis tools, as well as changing the layout of generated code.

4.2 C Compiler Back-End

The C back-end for FSMLanguage builds a ”giant” switch [**?**] style of FSM. This allows for the C-implementation to accurately reflect the characteristics of an FSM. Namely, that all actions, or assignment statements, within a given state transition appear to happen atomically as they do in the VHDL models of FSMLanguage-based FSMs. Additionally, the ”giant” switch allows for an accurate estimation of execution time (measured in clock cycles) of an FSM when implemented in VHDL. While the VHDL models of an FSM are deterministic, the execution of a C model may not be due to a myriad of factors in software-based systems. The code is ascribed with a built-in counter to estimate execution times that can be compared to results from hardware simulation models. The ability to estimate execution times allows programmers who have no knowledge of hardware simulation tools to get an idea of how efficient their program implementation will be when implemented in hardware.

The C back-end is intended to be used with Xilinx’s MicroBlaze soft-core processor [**?**]. This processor contains built-in FSL ports that are accessible via put/get instructions in the MicroBlaze ISA [**?**]. Compilation directly translates FSMLanguage channel constructs into C-macros that make use of the MicroBlaze’s FSL ports. This architecture allows software-based FSMs to interact directly with other FSMs that use the channel abstraction available in FSMLanguage; thus enabling transparent communication across the hardware/software boundary.

The type system of FSMLanguage is currently based directly on an HDL-like type system: composed solely of individual bits (std_logic) and arrays of bits (std_logic_vectors). The flexibility of this type system along with the ability to express arbitrary bit-arithmetic, bit-slicing, and bit-concatenation can result in extremely complex, inefficient, and hard-to-read C code. This complexity is the direct result of trying to map a flexible, arbitrary bit-width set of operations, into a less flexible, fixed bit-width language. The resulting code must make use of objects (structs) and high-level functions instead of native data types and operations, thus introducing a significant performance overhead.

A less flexible, software-oriented type system would allow for simpler representations of expressions in both C and VHDL, but at the cost of reduced specialization. Future versions of FSMLanguage may use a more software-like type system in order to allow for more efficient software code generation. Currently the C back-end uses native C data types, and does not support arbitrary bit-width operations. A prototype back-end that does support arbitrary bit-width operations through the use of structured data types is currently being developed, and will most likely be transformed into a C++ back-end to make use of operator overloading. Figure 6 demonstrates how arbitrary bit-width operations are handled in both FSMLanguage, VHDL, and C. Note that in the C version, a total of 8 function calls are required in order to pack (box) and unpack (unbox) the objects used to represent arbitrary bit-width values. This style of C code, even with function inlining, is not as efficient as using native C data types. This overhead can be avoided by using the native C compiler and encapsulating all arbitrary bit-width operations inside of hand-written functions that can be linked against at compile-time.

```
my_mem[a(16 to 31)] <= my_mem[a(15 to 30)] + 10;
```

(a) FSMLanguage Representation

```
when state0 =>
     my_mem_addr0 <= a(16 to 31);
     my_mem_rENA0 <= '1';
     next_state <= state1;
when state1 =>
     next_state <= state2;
when state2 =>
     my_mem_addr0 <= a(15 to 30);
     my_mem_dIN0 <= my_mem_dOUT0 + 10;
     my_mem_wENA0 <= '1';
     my_mem_rENA0 <= '1';
     next_state <= state3;
```

(b) VHDL Representation

```
bit_vec_t addr0 = get_slice(
                   a,
                   bit_vec_CREATE(16, 32),
                   bit_vec_CREATE(31, 32));
bit_vec_t addr1 = get_slice(
                   a,
                   bit_vec_CREATE(15, 32),
                   bit_vec_CREATE(30, 32));

my_mem[ addr0.val ] = bit_vec_ADD(
                        my_mem[ addr1.val ],
                        bit_vec_CREATE(10, 32));
```

(c) C Representation

Fig. 6. Examples of Arbitrary Bit-Width Representation

5 Experimental Results

The following sections describe some of the experimental results of using FSMLanguage to develop software- and hardware-implementations of finite-state machines. The FSM implementations were all synthesized and tested on a Xilinx ML507 Development Board containing a Virtex-5 FXT-70 FPGA. Each test is evaluated in terms of performance, circuit size (chip area), and code size. While performance and circuit size are the dominant factors in HW/SW co-design, programmer and designer productivity is also important. Productivity can be related to code size in an abstract way; and in these tests, is measured in terms of lines-of-code (LoC). The LoC metric is included in this paper to illustrate how a single language can replace the use of multiple, distinct languages while simultaneously reducing the total amount of code required to describe a system.

5.1 Producer/Consumer Example

A simple producer/consumer example, previously illustrated in KIWI [**?**] and similar to a test in PRET [**?**], has been constructed to illustrate the use of the communication abstractions available in FSMLanguage. The implementation of each FSM can be targeted to software or hardware while still remaining compatible with other types of FSMs, regardless of their implementation type.

The producer, in this example, generates a stream of integers, starting at zero, that are monotonically increasing until a pre-defined limit is reached, at which time the producer stops executing. The producer's data stream is sent over an FSMLanguage channel so that it can be accessed by the consumer. The job of the consumer is to consistently take data from the producer, multiply this data by 2, and produce an output value on another outgoing channel. If the consumer receives an input from the producer, then it will generate a new output. The consumer will then ”block” until another input from the producer is received.

The FSMLanguage programs for the producer and consumer can be seen in Figure 7 and Figure 8 respectively. The channel interface used between the two FSMLanguage programs is an abstraction that is compatible with both the MicroBlaze soft-core processor as well as with custom logic, thus allowing each of the FSM programs to be executed either in hardware or software. The producer/consumer example was tested for correctness in several different configurations listed in Table 1. All configurations correctly execute on a Xilinx ML507 development board. It is important to note that the software-based implementations of the producer and consumer also require extra Block RAM for storing instructions and data that the hardware-based implementations do not require at all.

This benchmark highlights the ability to encapsulate hardware/software interfaces within FSMLanguage. None of configurations require the programmer to make changes to the application code. This allows a programmer to very easily perform design space exploration without requiring reimplementation of the application. FSMLanguage also eliminates the need for a programmer to write driver routines to interact with architecture-specific features such as dedicated memories or communication channels. Instead, a programmer describes such interaction with a special syntax that is then elaborated by

```
-- ******************************
-- Producer Example
-- ******************************
-- Generates a set of outputs
-- through a channel
-- ******************************

CS: current_state;
NS: next_state;

GENERICS:
DWIDTH, integer, 32;    -- Data width

PORTS:

CONNECTIONS:

MEMS:

CHANNELS:
chan1, DWIDTH;

SIGS:
counter, std_logic_vector(0 to DWIDTH-1);

INITIAL: reset;

TRANS:

reset -> initialize

-- Initialize counter to 0
initialize -> genOutput where
{
    -- Initialize counter
   counter' <= ALL_ZEROS;
}

-- Generate 10 outputs
genOutput | (counter < 10) -> genOutput where
{
    -- Increment the counter
    counter' <= counter + 1;
    -- Output the current value
    #chan1   <= counter;
}
genOutput -> halt

-- Halt (remain in halt state forever)
halt -> halt

VHDL:
```

Fig. 7. Producer FSMLanguage Program

```
-- ******************************
-- Consumer Example
-- ******************************
-- Infinitely consumes inputs
-- and generates outputs
-- ******************************
CS: current_state;
NS: next_state;

GENERICS:
DWIDTH, integer, 32;    -- Data Width

PORTS:

CONNECTIONS:

MEMS:

CHANNELS:
chan1, DWIDTH;
chan2, DWIDTH;

SIGS:
counter, std_logic_vector(0 to DWIDTH-1);

INITIAL: reset;

TRANS:

reset -> grabInput

-- Grab a value from the input channel
grabInput -> genOutput where
{
    counter'   <= #chan1;
}
-- Generate a new value on the ouptut channel
-- and repeat (loop back)
genOutput -> grabInput where
{
        #chan2  <= counter + counter;
}

VHDL:
```

Fig. 8. Consumer FSMLanguage Program

the FSMLanguage compiler; thus eliminating this error-prone task from the design and implementation cycle [**?**].

The producer/consumer example was also evaluated using a lines-of-code metric (LoC) as shown in Table 2. The inversion in both the VHDL and C LoC metric for the producer and consumer relates to the fact that the consumer uses an additional channel interface. The extra channel interface, which requires only 1 LoC in FSMLanguage, requires a multitude of changes to a VHDL program (10 lines of code). These changes affect a VHDL program's port and signal declarations, sensitivity lists, and synchronous state transition process. These changes are similar to the changes required in a C program generated by the FSMLanguage compiler except for those involved in sensitivity lists.

Table 1. Producer/Consumer Testing Configurations

Producer	**Consumer**	**System Size**	**Correct?**
SW	SW	2,608 LUTs	Yes
SW	HW	1,460 LUTs	Yes
HW	SW	1,508 LUTs	Yes
HW	HW	360 LUTs	Yes

Table 2. Lines of Code (LoC) for Producer/Consumer

	Language		
	FSMLang.	**VHDL**	**C**
Producer	53	273	156
Expansion Factor		5.1x	2.9x
Consumer	43	277	159
Expansion Factor		6.4x	3.6x

5.2 Sorting Benchmark

A benchmark is constructed using a combination of bubblesort and mergesort to sort a large array of integers using the hthreads platform, an operating system designed for hybrid CPU/FPGA environments [**?**,**?**]. The purpose of this benchmark is to show how FSMLanguage can be used to develop custom hardware components to replace software components that are performance critical. In this benchmark the generation of data as well as the merging of sorted data is always performed in software, while the sorting of each "section" of data can be performed in either software or hardware, and with varying numbers of threads.

The sorting application generates a set of random data to be sorted, in this case 1 MB of 32-bit integer data to be sorted in a divide-and-conquer manner. After generating

the data, the main thread then spawns a number of sorting threads, either in software or hardware, and feeds data to each thread in an on-demand manner through software-based mailboxes in chunks of 2,048 words of data. Each hardware-based sorting thread has a thin software-based wrapper that performs the mailbox operations and marshals data into and out of the FSMLanguage-based sorting core, as programs written in FSM-Language are not currently able to use the hthreads OS API calls at this time. The data marshaling done by the wrapper thread involves copying the data to be sorted to a memory-mapped dual-ported BRAM that is connected directly to the FSMLanguage-based bubblesort core, as well as monitoring the go/done control signals emanating from the hardware core. The main thread continues to feed data to the sort threads until all data chunks have been sorted. Next, merging is done on all of the chunks of data and a correctness check is done to make sure that the data has been correctly sorted and merged.

The performance results of the sorting threads in both software and hardware are shown in Table 3. In all tests the instruction and data caches of the CPU, a PowerPC440, were enabled and the compiler optimization level was set to -O2. The body of the FSM-Language implementation of bubblesort can be see in Figure 9.

Table 3. Sorting Results (1 MB of Data)

	Number of Threads			
	1	**2**	**3**	**4**
SW Exec. Time	18.4 s	18.4 s	18.4 s	18.4 s
HW Exec. Time	9.50 s	4.70 s	3.10 s	2.30 s
Speedup	1.93	3.91	5.93	8.00

Table 4. Lines of Code (LoC) for BubbleSort (C** is a handwritten BubbleSort function in C)

	Language			
	FSMLang.	**VHDL**	**C**	**C****
Sort	96	365	203	23
Expansion Factor		3.8x	2.1x	0.23x

The speedup achieved by using multiple hardware threads comes from the fact that the hardware threads can truly execute in parallel, while in a single-CPU system, software threads merely execute pseudo-concurrently in time. Therefore no additional speedup is achieved when using multiple software threads, as they were all being time-multiplexed on a single CPU. The speedup achieved by using a single hardware thread can be understood by comparing the steps taken during bubblesort on the PowerPC440 to the steps taken by the hardware core generated from FSMLanguage.

The control-flow graphs (CFGs) for each application have been generated from their respective executable formats: PowerPC440 assembly language and VHDL. The CFGs

```
TRANS:
reset -> idle
idle | (go = ’0’) -> idle where
{
    stopped’    <= ’1’;
}
idle | (go = ’1’) -> begin_sort where
{
  stopped’ <= ’0’;
  n’       <= sort_length;
  n_new’   <= sort_length;
  swapped’ <= ’1’;
}

begin_sort | (swapped = ’0’) -> halt
begin_sort | (swapped = ’1’) -> for_loop where
{
  -- Initialize variables before FOR loop begins
  swapped’ <= ’0’;
  i’       <= ALL_ZEROS;

  -- Prefetch the "1st" data1 before the for loop begins
  data1’  <= array[ALL_ZEROS];
}

  for_loop | (i >= n) -> begin_sort where
  {
    n’  <= n_new;
  }
  for_loop | (i < n) -> cond_check where
  {
    -- Fetch data2, while data1 has already
    -- been calculated during the last iteration
    data2’  <= array[i+1];
  }

    cond_check | (data1 <= data2) -> for_loop where
    {
      -- Move value in data2 to data1 for
      -- next pass (now only need to fetch the new data2)
      data1’  <= data2;
      i’      <= i + 1;
    }
    cond_check | (data1 > data2) -> cond_body where
    {
      -- 1/2 update (only one write per state)
      array[i]  <= data2;
    }

      cond_body -> for_loop where
      {
        -- the other 1/2 of the update (only one write per state)
        -- Note, this is the "next" data1 value so keep it around
        array[i+1]  <= data1;
        n_new’      <= i;
        swapped’    <= ’1’;
        i’          <= i + 1;
      }

halt -> idle where
{
  stopped’ <= ’1’;
}
```

Fig. 9. BubbleSort FSMLanguage Program

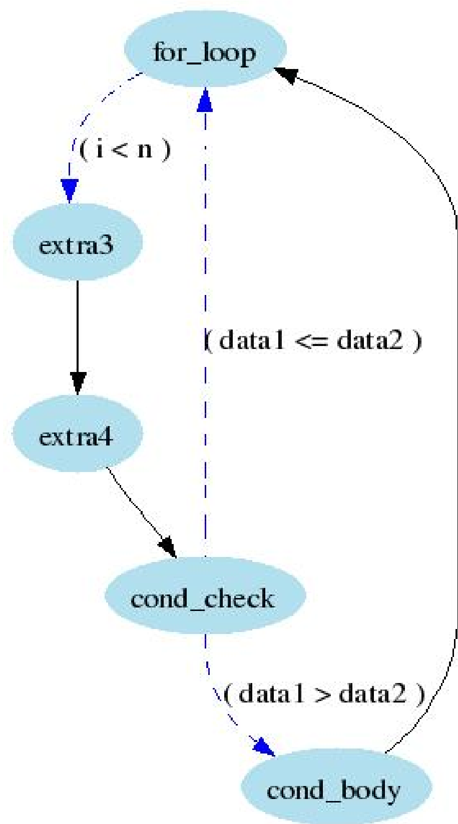

Fig. 10. BubbleSort CFG - VHDL from FSMLanguage

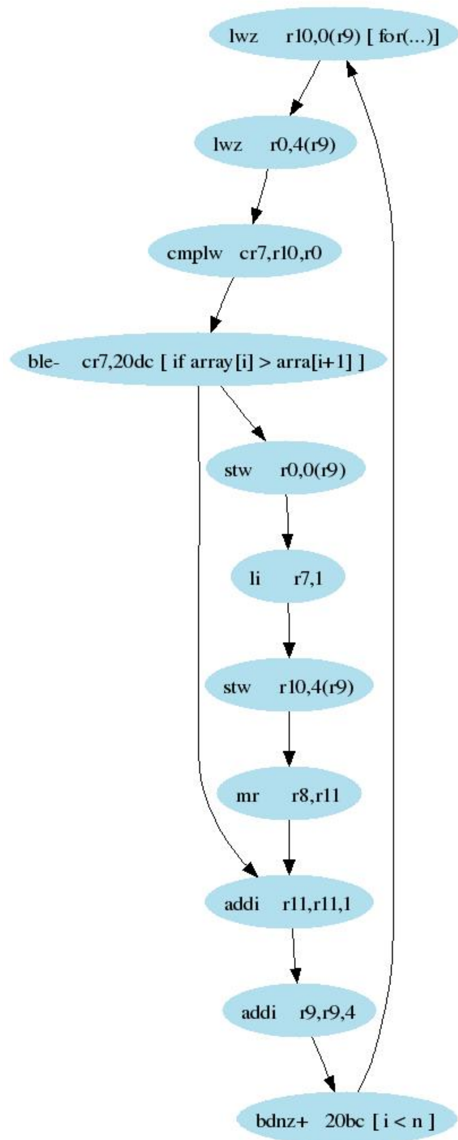

Fig. 11. BubbleSort CFG - PPC440 Assembler

shown only represent the ”kernel” of the bubblesort algorithm, which is the inner for-loop which iterates over the array while swapping items. Figure 11 represents the CFG of the PowerPC440 assembly implementing the bubblesort routine. When executing on the PowerPC440, this for loop contains 11 instructions when a swap occurs, and 7 instructions when no swap occurs. Given that the PowerPC440 has a 7-stage pipeline this means that a best-case estimate of the execution time of this loop is 14 cycles assuming no pipeline stalls, no cache misses, and no branch mispredictions.

The CFG of the VHDL generated from the FSMLanguage implementation of bubblesort can be seen in Figure 10. The for loop kernel in this CFG contains 5 state transitions when a swap occurs, and 4 cycles for when no swap occurs. Given that the circuit generated from FSMLanguage is completely deterministic, this means that the worst-case execution time of this kernel is 5 clock cycles, a 2.8x improvement over the PowerPC. The speedup of 1.93 achieved when using a single hardware thread does not meet the ideal figure due to overhead incurred by copying data into and out of the local BRAM attached to each hardware thread.

Additional speedup can be achieved by increasing the number of hardware threads used for sorting, as each of the hardware threads can run concurrently. Each hardware thread operates out of their own local BRAM, which prevents unwanted system bus contention, and leads to an almost linear speedup. The logic overhead incurred from using the hardware-based sorting core is also very minimal as each sorting core only requires 453 slice registers and 582 slice LUTs (look-up tables) on the Virtex-5 FXT chip. This space usage represents only 1% of the capacity available on the FPGA. Additionally, each sorting core requires 8 Block RAMs, which represents only 5% of the Block RAM resources of the Virtex-5 FXT 70 chip. This resource usage is quite small when considering that a MicroBlaze soft-core processor, in synthesized form, requires more than twice as much logic resources to implement on the same FPGA architecture (approx. 1300 slices including memory buses and controllers).

A lines-of-code (LoC) comparison of the sorting benchmark can be seen in Table 4. While the FSMLanguage bubblesort program is approximately four times longer than an equivalent handwritten function in C, it is still 3.8 times the size of a VHDL implementation of the algorithm. The C code generated by the FSMLanguage compiler uses a ”giant switch” style of FSM coding that is not as compact as handwritten C code.

5.3 Interpretation System

Interpreters are easily thought of and implemented as ”giant” case/switch statements [**?**], and as such are easily converted to an FSM description. The purpose of this benchmark is to show how easily an interpreter can be re-targeted when written in FSMLanguage. In this test a small interpreter is written in FSMLanguage that implements the instruction set listed in Table 5. The interpreter is designed to have 3 separate interfaces: (1) a control interface, (2) a memory interface for instructions and data, and (3) a ”state” interface used to perform context switching. The internal state of the interpreter is composed of a 256-entry 32-bit memory used as a register file, as well as a 10 other registers used for bookkeeping (instruction decode, intermediate results, program counter, etc.). A snippet of the interpreter’s FSMLanguage program is shown in Figure

12. This code snippet highlights the fetch/decode/execute cycle of the interpreter for arithmetic instructions.

```
-- **** Fetch Next Instruction ****
fetch | (go = '1' and mode = CMD_INTERPRET) -> decode where
{
    instr' <= prog_mem[pc];
}
--- **** Decode Instruction ****
decode | (opcode_type = TYPE_ARITHMETIC) -> do_arithmetic where
{
    -- Fetch arguments
    a'  <= regfile[r_arg_a];    -- Contents of registerA
    b'  <= regfile[r_arg_b];    -- Contents of registerB
}
-- **** Execute Stage (ARITHMETIC) ****
do_arithmetic | (opcode = OPCODE_ADD) -> writeback where
{
    regfile[r_dest] <= a + b;
}
do_arithmetic | (opcode = OPCODE_SUB) -> writeback where
{
    regfile[r_dest] <= a - b;
}
    .
    .
    .
-- **** Writeback Stage ****
writeback -> fetch where
{
    -- Increment PC to go to the next instruction
    pc' <= pc + pcInc;
}
```

Fig. 12. Code Snippet - Interpreter FSMLanguage Program

The FSMLanguage description of the interpreter is compiled to both software and hardware implementations and tested for correctness using a a set of recursive programs (Fibonacci, factorial, and McCarthy91). These programs exercise a majority of the interpreter's control and data path through data/stack manipulation, control flow, and arithmetic operations. The code structure of the interpreter uses HDL-specific bit manipulation routines that are not available in C, so a small library of C routines was created by hand to duplicate this functionality. The language-specific bit manipulation routines can be encapsulated in functions in both C and VHDL so that an FSMLanguage program can make use of the functions in an implementation-independent way. This problem will be solved in the future by outfitting the FSMLanguage compiler with additional support for implementing arbitrary bit-manipulation routines in the generated C code.

The hardware implementation of the interpreter requires a total of 730 slice LUTs, 306 slice registers, and 1 Block RAM for implementing the register file, which is approximately half of the size of the MicroBlaze processor implemented in the same FPGA technology. Another advantage of the hardware implementation is that it operates in a completely deterministic manner. Instruction fetches always take 3 clock

Table 5. Interpreter Instruction Set

Name	Format	Description
Add	ADD Rd Ra Rb	Rd = Ra + Rb
Subtract	SUB Rd Ra Rb	Rd = Ra - Rb
Multiply	MULT Rd Ra Rb	Rd = Ra * Rb
Logical And	AND Rd Ra Rb	Rd = Ra ’and’ Rb
Logical Or	OR Rd Ra Rb	Rd = Ra ’or’ Rb
Exclusive Or	XOR Rd Ra Rb	Rd = Ra ’xor’ Rb
Shift-Right Arithmetic	SHRA Rd Ra	Rd = Ra ’shr’ 1, (MSB’ = LSB)
Shift-Right Logical	SHRL Rd Ra	Rd = Ra ’shr’ 1, (MSB’ = 0)
Shift-Left	SHL Rd Ra	Rd = Ra ’shl’ 1, (LSB’ = 0)
Load Low	LLOW Rd Imm	Rd = (Rd ’and’ 0xffff0000) ’or’ (Imm ’and’ 0x0000ffff)
Load High	LHI Rd Imm	Rd = (Rd ’and’ 0x0000ffff) ’or’ (Imm ’and’ 0xffff0000)
Jump Equal To Zero	JEZ Rc Ra	If (Rc == 0) Then PC = Ra
Load	LOAD Rd Ra Roff	Rd = MEM[Ra + Roff]
Store	STORE Rd Ra Roff	MEM[Ra + Roff] = Rd

cycles (1 cycle read setup, and 2 cycle read latency for BRAM), decode requires an additional 3 cycles as the register file is also implemented using BRAM. Instruction execute and write-back latency is deterministic, but varies for each instruction type. Execution requires 2 cycles for arithmetic operations, 3 cycles for multiplies, 5 cycles for load immediate instructions, 5 cycles for stores, and 7 cycles for loads. This results in a worst-case execution time for an interpreter instruction of 13 clock cycles. The fetch/decode cycle of a software-based interpreter requires multiple accesses to memory and often involves bus operations that require an order of magnitude more time (100s to 1000s of cycles) to complete. The fetch/decode process in the hardware implementation is able to make use of a dual-ported BRAM as the interpreter’s register file. This architecture allows simultaneous access to the dual-ports of the register file, as found in the decode stage shown in Figure 12. Additionally, the BRAM-based register file and memories makes the hardware implementation fully deterministic, whereas the software implementation has non-determinacy introduced by cache misses and branch mispredictions. Overall, the results of the interpreter benchmark show that a simple FSMLanguage description of an interpreter can be translated into an efficient hardware implementation without requiring a programmer to have detailed knowledge of hardware design techniques.

6 Conclusion

FSMLanguage provides programmers with the ability to concisely describe Mealy finite-state machines in a form that allows the code to be targeted to efficient software and hardware implementations. The language and compiler are designed to take advantage of all the resources provided by modern Platform FPGAs; namely custom logic, distributed Block RAMs, soft-core processors, and FIFO channel connections. Individual implementation strategies can be changed for each FSM without affecting overall sys-

tem operation, as FSMLanguage communication abstractions are able to transparently cross HW/SW boundaries. This makes it much easier for designers to explore their system's design space by eliminating the need to manually re-implement higher-level descriptions of system components.

The three micro-benchmarks demonstrate that FSMLanguage can reduce code size and verbosity, while also providing choices in terms of implementation style, speed, and resource usage. The producer/consumer benchmarks highlights the ability of FSMLanguage programs to cross the HW/SW boundary as well as the ability to make changes to the HW/SW partitioning of a system after initial design and implementation has already occurred. The sorting benchmark highlights the ability to produce efficient hardware implementations of FSMLanguage programs that can be used as application accelerators or co-processors. Finally, the interpreter use-case highlights the ease of re-targeting FSMLanguage programs, and the different features of the software and hardware implementation options.

Overall, the purpose of FSMLanguage is to demonstrate that simple domain-specific languages can be effective for hardware/software co-design for FPGAs. A re-targetable language, such as FSMLanguage, allows a single program specification to be implemented in a variety of ways without forcing a programmer to undergo re-implementation. FSMLanguage permits program specifications to be coded, compiled, debugged, and tested in both a hardware- and software-environment. The hardware and software implementations of FSMLanguage programs remain compatible with one another, allowing the hardware/software partitioning of a system to be altered post-design time.