SHIM: A Language for Hardware/Software Integration

Stephen A. Edwards∗
Department of Computer Science, Columbia University
1214 Amsterdam Avenue, New York, New York, 10027
sedwards@cs.columbia.edu

Abstract

Virtually every system designed today is an amalgam of hardware and software. Unfortunately, software and circuits that communicate across the hardware/software boundary are tedious and error-prone to create. This suggests a more automatic way to synthesize them.

This paper presents the SHIM language, which combines imperative C-like semantics for software and RTL-like semantics for hardware to allow a unified description of hardware/software systems. Hardware processes and software functions communicate through shared variables, hardware for which is automatically synthesized by the SHIM compiler, which generates C and synthesizable VHDL.

I demonstrate the effectiveness of the language by re-implementing an F2C bus controller. The SHIM source is half the size of an equivalent manual implementation, slightly faster, and has a smaller memory footprint. Partial and complete hardware implementations in SHIM are also presented, showing that SHIM is succinct and effective.

1 Introduction

As integrated circuit technology advances relentlessly, the size and complexity of a typical design continues to spiral upward. As always, managing complexity is the designer’s greatest challenge. The design must be right the first time and be completed faster than before. While validation methods such as simulation and formal verification work well to discover mistakes, moving to higher levels of abstraction, such as from the gate level to the register transfer level, is more effective because it helps designers to avoid mistakes in the first place. In this paper, I describe a language that raises the abstraction level for hardware/software systems.

My intention with SHIM (Software/Hardware Integration Medium), the language I propose here, is to provide seamless communication between hardware and software modules. It arose in part from observing beginning design students tackle combined hardware/software systems: they understood C well, and could learn VHDL, but had difficulty making the two worlds communicate.

Rather than propose a completely new semantics for SHIM, I chose to integrate two well-known, well-established semantics: C-like imperative semantics for the software portion of the design and register-transfer level semantics for the hardware. The syntax of SHIM most closely mimics C, which I chose both for its familiarity (C, C++, Java, and C# programmers all know it well) and its succinctness.

The SHIM compiler takes a single specification—a collection of software functions and hardware processes—and generates both C and synthesizable VHDL source. No attempt is made to do automatic partitioning between the two domains: it is the user’s responsibility to mark everything as hardware, software, or shared. This is partially to simplify the compiler, but the more serious issue is the very different semantics of the two domains. When I began this project, I wanted to make switching functionality between the two domains as easy as marking a function differently, but the semantics of RTL and software are just too different: how many clock cycles should a piece of software code take, and how is concurrency implemented in software? Hardware-like software and software-like hardware languages (e.g., Esterel [2] and Handel-C [4]) have been proposed, but all deviate significantly from software and RTL semantics and none are widely accepted.

Aside from minor improvements in RTL syntax (SHIM’s RTL can be half as verbose as the equivalent VHDL), SHIM’s major contribution is the automatic synthesis of communication between the software and hardware components based on a shared memory model. While algorithmically simple, such synthesis simplifies the designer’s task, avoids errors, and improves code portability.

Figure 1a shows a simple SHIM program that represents a hardware timer. It manages the shared variable counter, whose value is stored in the hardware but can also be read and written by the two software functions reset_timer and get_time. From this description, the compiler produces a C header file (Figure 1b) that describes the external software interface for the module, a C source file containing code implementing reset_timer and get_time (Figure 1c), and VHDL for a peripheral that attaches to a proces-

∗Edwards is supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and by New York State’s NYSTAR program.
The SHIM compiler generates twenty lines of VHDL from it, code a designer would have otherwise had to write manually. The fractional improvement is high for such a small example because most of the code is related to the bus interface, but large examples remain smaller and easier to code.

As its name suggests, SHIM is designed for creating “glue” that connects hardware and software. It specifically does not try to be a general-purpose software language or a full hardware description language. Instead, it provides facilities for connecting subsystems written in hardware and software and external interfaces: externally-visible functions and variables in software, ports in hardware.

SHIM can also be thought of as a language for simultaneously writing peripherals and their device drivers. As such, it is applicable to systems where hardware can be customized, such as ASICS with processor cores; field-programmable gate arrays, especially those with processors such as the Virtex II Pro; and more programmable SoCs that consist of multiple processor cores, hard peripherals, and a substantial amount of programmable logic. This latter class of chip, a platform with a mix of programmable hardware and software, seems a likely architecture of the future since the cost of designing complete chips is skyrocketing; it seems more likely that standard programmable platforms will grow more popular. When the hardware is provided and immutable, the NDL language [8] would be better-suited.

SHIM currently generates code (hardware and software) for the Xilinx Microblaze soft processor core driving IBM’s CoreConnect On-Chip Peripheral Bus (OPB). I chose this configuration because we have target boards from XESS and a development environment from Xilinx, but there is nothing OPB-specific about the SHIM language. The synthesis code is about 1/6th of the compiler and could be easily re-targeted to a different processor and bus.

2 Related Work

Unlike other hardware/software codesign systems, SHIM uses imperative C semantics for software and RTL for hardware. Polis [1] describes its systems with communicating extended finite-state machines. COSMOS (Jerraya et al. [11]), COSYMA (Ernst et al. [9]), and CoWare (Bolsens et al. [3]) all use concurrently-running processes communicating through remote procedure calls (RPC). FIFO-based communication among concurrently-running processes is another choice (see, e.g., Gupta and De Micheli [10]).

Although such higher-level semantics enable automatic hardware/software partitioning (a focus of earlier work), they raise efficiency issues. While RPC is natural in software, it seems overly sequential for hardware. Furthermore, some techniques employ high-level hardware synthe-
sis, which industry has largely rejected due to efficiency concerns. COSYMA describes a hardware/software system using a C-like imperative language, partitions it, and passes certain processes to high-level synthesis.

Most other work proposes synthesizing communication mechanisms far more complicated than shared memory. Jerraya et al. [5] propose synthesizing wrappers. CoWare layers protocols. It is unclear whether these approaches are general enough and produce efficient hardware.

SHIM synthesizes bus-based communication, but is not wedded to it. Chou et al. [6, 7] target microcontrollers communicating through a limited number of I/O pins.

Other languages, such as Mérillon et al.’s Devil [13] (generates C macros for communicating with peripherals), Thibault et al.’s GAL [14] (generates graphics adapter drivers), and Conway et al.’s NDL [8] (synthesizes Unix device drivers), focus only on synthesizing software and assume the hardware is given.

Lavagno and Sentovich’s ECL [12], which combines Esterel-like synchrony [2] with imperative C, partially inspired SHIM. Like SHIM, their compiler uses simple rules to split a program into the two domains, but they interleave Esterel and C at the statement (instead of function) level and they do not target hardware/software systems.

3 The SHIM Language

SHIM was designed to be simple. A SHIM program is a module containing global variables and functions. Each variable is either unmarked, marked hw, or marked shared indicating it is to be visible in software only, hardware only, or both. The translation of hardware- and software-only variables is straightforward; the state of a shared variable is held in hardware and the compiler synthesizes a circuitry that allows it to be read and written from software. Each variable can also be marked in or out, indicating its value comes from outside the module or is visible outside (variables are only module-visible by default).

Functions may either be unmarked (indicating software) or marked hw. The translation of a software function is nearly one-to-one. A hardware function becomes a concurrently-running synchronous hardware process clocked by the bus to which the synthesized peripheral is attached. Currently, all hardware functions take no arguments, do not return a value, and may not be explicitly called (they are implicitly called once per clock cycle).

SHIM supports bit vectors, arrays of bit vectors, and string literals. Bit vectors can be manipulated as integers and are either signed (int) or unsigned (uint). The width of each bit vector may be specified or may be omitted (it defaults to a standard value). Array dimensions must be given.

The body of a SHIM function contains the familiar set of C statements, if, switch, for, while, and return; expressions; and local variable declarations. Expressions follow the usual C syntax with a few extensions: applying the array index operator to a bit vector returns a bit, e.g., a[1] is the second least-significant bit of vector a. Looping statements are not allowed in hardware processes.

Software functions have the usual sequential semantics; there is a single program counter; bit vectors are call-by-value, and arrays are call-by-reference.

Hardware processes have RTL semantics. Every hardware process is effectively invoked once per clock cycle. Each shared or hardware-only variable may be written by a single process only, a syntactic constraint common in RTL, but may be read by multiple processes. Outputs are latched, meaning that if a process writes a variable, the new value can be seen by other processes only in the next cycle, but the same process can see the new value in the same cycle.

When software-only variables have an initial expression, the value is assigned once when the variable enters scope (e.g., when the program starts for global variables, and when control enters the enclosing block for local variables) and stored. Software-only variables marked const must be given an initial value that can be determined at compile-time and may not be written.

Hardware-only variables may also be assigned to an expression, but their semantics differ: hardware variables assigned to an expression always take the value of that expression (i.e., are effectively continuous assignments) and may not be assigned in other processes. Such variables generate combinational logic and conceptually execute after all the processes have executed for the cycle and updated their outputs. That is, they can be used to communicate among processes, but only across clock-cycle boundaries.

4 Compiling SHIM

The SHIM compiler comprises some some 3000 lines of OCAML code. The largest single module (about 500 lines) performs the interface synthesis. Much of it comes from explicit construction of SHIM code for the bus controller.

The compiler’s structure is typical: an automatically-generated scanner and parser generates an abstract syntax tree, which is sent through a static semantic analyzer that resolves names and types and dismantles certain constructs to produce an AST-like intermediate representation.

The interface generator takes the interface-agnostic IR from the dismantler, enumerates the shared variables (a simple walk through the symbol table), and assigns each an I/O address. It transforms the body of each software function by changing reads and writes of shared variables to the appropriate I/O function call.

Interface synthesis for hardware is more complicated. At the end of each process, the compiler adds code that reads from the bus each shared variable written in that process. This generates at most one read of each shared variable since each such variable may be written by at most one
hardware process (software functions have no such restriction). Furthermore, placing the read at the end of the process means a write from the software domain takes precedence over a write from hardware. I chose these semantics because hardware writes every cycle are a common idiom (e.g., the count variable in Figure 1). Users who dislike software taking precedence over hardware can use separate shared variables for each communication direction.

Two additional hardware processes are synthesized: one reads all the unwritten shared variables from the bus (e.g., those shared variables that are only written from software, never from hardware). The other copies each shared variable to the bus when requested (i.e., when software generates a read cycle to an I/O address).

Finally, the interface synthesis procedure adds interface ports for all the bus signals and processes that decode the address bus and speak the bus protocol. After this, the IR contains all the necessary interface code.

Once the interface synthesis procedure runs, C source, C header, or VHDL source is generated. Syntax-directed translators transform the IR into simple ASTs for C or VHDL, which are then pretty-printed. Using another intermediate representation guarantees syntactically-correct output.

5 An Example: An I\(^2\)C Bus Controller

To test SHIM, I re-implemented a simple hardware/software interface with it. In an embedded systems class, we use XSB–300E boards from XESS Corporation that contain, among other things, a Xilinx Spartan IIE FPGA and a Philips SAA7114H video decoder. An I\(^2\)C bus connecting the two can write to the 7114’s many configuration registers. The I\(^2\)C bus is a low-speed two-wire clocked bus designed for exactly such an application, and is very lenient about timing. The bus has no maximum delays between transitions so it can be controlled completely from software.

One of my students (Marcio Buss) implemented a simple all-software I\(^2\)C bus controller to program the 7114. Its only task is to send a series of commands to write over eighty configuration registers. Marcio spent a few days writing its 259 lines of C and 133 lines of VHDL. An I\(^2\)C bus controller can be faster than its software (bytes are still sequenced in software).

The first two lines of this table are most telling: Marcio wrote 259 + 133 = 392 lines of C and VHDL to implement this controller, while I was able to achieve the same functionality in 171 lines of SHIM, a reduction of over 55%.

Figure 2 shows a fragment of the all-software SHIM implementation of the I\(^2\)C controller. Controls for the two I\(^2\)C wires, SDA and SCL, are shared one-bit variables, and the send function simply toggles them appropriately.

I then implemented more of this controller in hardware using SHIM, the sizes of which I report in the next three lines of the table. By design, changing the implementation of a function from software to hardware is not as simple as marking it as hw (SHIM hardware uses RTL semantics, which are very different from C’s). The main difference is that sequencing must be coded as a state machine using, say, a switch statement (see Figure 3). This is the main source of the additional 152 lines of SHIM.

Figure 3 is a hardware fragment similar to the software of Figure 2. It toggles the clock and data lines to send a single byte of data. Most of the code is now devoted to sequencing since traditional RTL, and by design SHIM, requires all state machines’ next-state functions to be explicit. Additional complexity comes from synchronizing with software, done here with a four-phase handshake. The controller raises ready in the IDLE state to indicate it is ready to accept a command. The software then writes a command such as SEND into the command variable, which steps the state machine through a state sequence ending with IDLE0. In this state, the controller waits until it receives IDLE from the software, which sends it back to the IDLE state. The send function at the bottom of Figure 3 performs the software half of this handshake.

The SHIM version of a controller can be fasters than its handwritten equivalent. In the handwritten version of the

<table>
<thead>
<tr>
<th>Example</th>
<th>SHIM</th>
<th>C</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>I(^2)C Software (by hand)</td>
<td>259</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>I(^2)C Software</td>
<td>171</td>
<td>175</td>
<td>136</td>
</tr>
<tr>
<td>I(^2)C Bit-level</td>
<td>283</td>
<td>173</td>
<td>337</td>
</tr>
<tr>
<td>I(^2)C Byte-level Receive</td>
<td>299</td>
<td>163</td>
<td>358</td>
</tr>
<tr>
<td>I(^2)C Byte-level Send/Receive</td>
<td>323</td>
<td>116</td>
<td>344</td>
</tr>
<tr>
<td>Timer</td>
<td>15</td>
<td>20</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 1: Lines of code for various examples. The first line is a reference implementation of an I\(^2\)C bus controller written without the aid of SHIM. The next four lines list SHIM implementations with the same functionality with an increasing fraction of the system in hardware. The final line lists statistics for the simple example in Figure 1. For all but the first line, the SHIM column represents the number of lines written by the designer; the C and VHDL columns are lines of code generated by the SHIM compiler.
void send(uint:8 byte) {
    SDA_oe = 0; delay();
    for (int i = 7; i >= 0; i = i - 1) {
        SDA = (byte & 0x80) >> 7; delay();
        SCL = 1; byte = byte << 1; delay();
        SCL = 0; delay();
    }
    SCL = 1; delay();
    bool acknowledge_received = SDA_data;
    if (!acknowledge_received)
        xio.print("Acknowledge not received\r\n");
    delay();
}
I believe SHIM is successful in its aim to fuse two computational models but it raises the question of whether these two models are the best choices. A particularly glaring issue is that SHIM models are not easy to simulate. This is due to the models themselves: the two domains run asynchronously and while the hardware is timed, the software effectively is not, meaning that the behavior of the system may be nondeterministic or at least very difficult to predict without careful modeling of software timing, such as by using an instruction-set simulator. The timer example in Figure 1 is perhaps the simplest example illustrating this problem: the hardware effectively counts the number of clock cycles between calls of reset_timer and get_time, which is a complicated function of the processor and C compiler used to implement the system. Of course, such a performance timer is often desired for analyzing software, but more frequently the behavior of hardware/software systems is meant to be timing independent.

My feeling is that the shared-variable model of hardware/software communication, while the de facto standard, is too flexible and more complicated, but timing-independent, protocols are almost always implemented on top of it. For example, a peripheral usually provides status registers and interrupts that inform the software when it is ready for the next command, and expects that software will obey its protocol. In implementing the I²C bus controller described in Section 5, I implemented a four-phase handshake protocol to insure that the software waited for the hardware to complete its task before starting the next. Although this works and is robust in practice, it can be error-prone and is certainly more verbose than it needs to be.

Part of the problem is that the hardware/software boundary represents true parallelism and concurrency. Peripherals run truly independently from their processors (in most cases, that is the point of a peripheral), and so are truly concurrent systems posing all the classical problems such as races and deadlocks. However, since hardware/software systems are neither completely hardware nor software, using classical software techniques such as semaphores and critical regions or the standard hardware technique of a global clock seems inappropriate.

So I leave this question for future work: what is an appropriate general, timing-independent model of computation for hardware/software systems? Single-threaded imperative software and RTL hardware with shared variables, such as in SHIM, is reasonably general, but is too low-level, easily nondeterministic, and error-prone. There must be something higher-level that avoids tedious manual protocol implementation and ensures correctness.

References