Building hardware from C# models

Kenneth Skovhede and Brian Vinter, Niels Bohr Institute, University of Copenhagen, Denmark

Abstract

This paper introduces a method for modeling hardware in the C# language, using an abstraction called Synchronous Message Exchange. We describe how Synchronous Message Exchange helps model hardware from a high-level language, and describe the process used in automatically *transpiling* a subset of C# models into standard VHDL. We evaluate the approach with a set of examples, comprising a memory component, a financial trading algorithm, and AES encryption.

1 Introduction

General-purpose CPUs excel in many areas, such as price/performance ratio and versatility, and are supported by an unparalleled amount of software, such as, compilers, software, libraries etc. For some applications, such as power-aware devices for the Internet-of-Things and high speed data capture for sensors, general-purpose CPUs are unattractive because their versatility also includes an overhead [1]. This overhead is present because the CPU follows the Von-Neumann architecture and thus processes one item at a time, and communicates with external devices, such as memory storage. With an FPGA, the overhead can be reduced because it is possible to forgo the Von-Neumann architecture and perform many tasks in parallel, and also because the need for communication with external memory can be reduced [2].

Unfortunately, writing applications for FPGAs is significantly more complicated than writing applications for a CPU. One of the obstacles is the need to utilize parallelism, which is known to be more difficult for programmers than sequential programmers [3]. Another obstacle is the fact that the de-facto programming languages, VHDL and Verilog, are largely unchanged from their initial design in the 1980s, thus missing many productivity enhancements seen in the last 30 years.

From a software developers perspective, writing HDL seems arcane and ancient in many ways. The languages themselves lack most of the productivity features and syntactic sugar seen in modern languages. On the structural level there is a lot of redundancy and overly verbose constructs required, and on the functional level, the need to work with timed logic is very different from sequential code.

With all code development, FPGA as well as software, a large part of the development time is spent debugging and verifying. In HDL, the test benches are also written in the same language as the application itself. This has the benefit that types and other constructs are equivalent. However, the HDL languages are ill suited for many common operations performed in testing, such as reading and parsing files.

Modern programming languages, on the other hand, have so many free third-party libraries that they have one or more package managers, that allows easy installation of commonly used functionality. With FPGAs, some functionality can be found on OpenCores, but most functionality is vendor specific and comes with a price tag.

These issues combined, makes it difficult and cumbersome for software developers to venture into FPGA development, thus inhibiting many projects that could benefit from FPGA solutions.

1.1 Contribution

In this paper, we introduce a new programming model, a simulation library and transpiler that allows software developers to write FPGA applications entirely within the .Net framework. The implementation does not attempt to automatically parallelize code, but rather allows the user to write the processes, similar to how MyHDL and SystemC work.

As the library is implemented in .Net, it can be used with any .Net supported languages, such as C#, Visual Basic, F# and others, and runs on Windows, Linux and OSX. In this article, we focus on the C# language and show how we can use many of the program features, such as type inference, type consistency, and operator overloads to allow the user to rapidly prototype and test the program.

Once a program is sufficiently tested, it can be converted automatically to VHDL, which can be loaded in regular vendor tools for simulation and synthesis. With the generated VHDL, a test bench and trace files are also generated, that these can also be loaded with vendor tools to verify a clock-level equivalence between the .Net implementation and the VHDL version.

Should parts of the application benefit from VHDL specific implementation, it is possible to suppress VHDL generation on process or function level, and then supply the missing content directly in the generated VHDL. If the VHDL is regenerated, it will retain the changes, such that it is possible to have a development flow where the model can be

changed without breaking the implementation. With the generated test bench, it is possible to continuously verify the changes, and quickly pinpoint divergent implementations.

2 Related work

Recently, a number of alternative approaches to writing applications in a HDL have emerged, such as MyHDL [4], CλaSH [5], Xilinx Vivado HLS [6] and Altera OpenCL [7]. An interesting observation is that these approaches have little traction, and there is a general skepticism for non-HDL approaches as abstractions get in the way of full control. From a software developers view, using HDL seems similar to insisting on writing programs in assembly code.

The Altera OpenCL approach taps into the existing programmers that are familiar with the OpenCL approach used for GPGPUs. The abstractions provided by OpenCL allow the programmer to map highly parallel programs to FPGA, but at the same time has the downside that the OpenCL approach requires a host system with a management runtime. The host system is both a performance bottleneck, as it is for GPGPUs, but also limits the application of OpenCL to setups where a host system is available. If an existing codebase uses OpenCL already, the Altera approach offers an attractive choice.

The Vivado HLS approach uses a subset of the C programming language, and translates the sequential code into a finite state machine, with parallelism where the compiler can guarantee that there are no dependencies. This approach requires that the compiler can extract sufficient parallelism from the sequential code, otherwise the implementation will not be able to efficiently utilize the hardware. In Vivado HLS, the parallelism is largely extracted from loops, which the user needs to ensure are free of dependencies, and of statically assigned size. Once the loops are written such that iterations are dependency free, the user can annotate the loops in various ways to map it for efficient implementation. This approach can yield a faster prototyping phase, but applying annotations requires a deep understanding of the underlying hardware as well as the mapping from source code to FPGA design. The HLS approach is a variant of the holy grail of high performance computing, an auto-parallelizing compiler, as it attempts to extract parallelism from otherwise sequential code. From a substantial body of research conducted over the last 40 years, the limitations of this approach are well known, and unlikely to yield new groundbreaking results [8, 9]. If there is an existing C codebase, possibly with OpenMP parallelized loops, the HLS approach appears as the obvious choice, especially when paired with a full-system environment such as SDSoC [10] or QuickPlay [11].

The CλaSH approach is essentially a *transpiler*, in that it translates a program from Haskell into VHDL. A key feature of the functional programming paradigmes is that functions are *side-effect free*, and as such are easily parallelizable. CλaSH exploits this parallelism to generate VHDL code from a strongly typed program, which is plain Haskell but with only fixed length lists. The Haskell nature makes it simple to test and develop the algorithm, but requires that the problem can be expressed with Haskell syntax in a way that can be mapped efficiently to hardware.

MyHDL is closer to approaches, such as SystemC [12] and Cx [13], in that it models the hardware structures, such as signals and ports, and thus enables the programmer to explicitly design the application parallelism. Such approaches are usually faster in simulation, as they do not need to implement RTL simulation, but can execute in the host language. One advantage of MyHDL being implemented in Python is that it is very simple to implement test benches and integrate with existing unit test and continuous integration tools. A downside of using Python is the dynamically typed approach, which requires that programs contain some degree of annotated type information not typically found in Python programs.

3 Synchronous Message Exchange

To enable experimentation and rapid prototype development for clocked circuits, we needed an modeling tool that would allow us to to quickly test out ideas to see if they were worth pursuing, and came up with Synchronous Message Exchange, SME [14]. SME is based on the theory for Communicating Sequential Processes [15], CSP, and as such any SME implementation has an equivalent CSP model that can be verified to be free of race conditions and deadlocks. In CSP, processes communicate over channels using a strictly synchronous rendezvous model, similar to how a hardware component communicates by sending signals over wires. Our initial experiments with CSP for hardware design revealed that while the model works, it is cumbersome because wires can have multiple recipients, and thus needs to be implemented as *broadcast* operations. Another observation was that the CSP model is mostly useful due to the notion of external choice, which we found mapped poorly to FPGA usage scenarios.

With the SME model, we implemented only a single communication medium, which has broadcast semantics, and have no support for external choice. Even though this mapping result in a reduced kind of CSP, it is trivial to model a SME network as a CSP network, thus retaining the benefits from CSP with improved speed.

3.1 Communication

For processes to communicate we use a mechanism that is similar to `Ports` in VHDL, but instead of treating each signal independently we introduce a collection of signals in a *Bus*. This reduces the amount of typing required as it enables grouping of shared items, such as the enable, address and data signals for a memory component.

To emulate the behavior of timed logic, the signals on a bus are initially undefined, and reading them causes an error. When the signals are written, the value is held back until the next clock cycle, providing a latching mechanism.

This is similar to how a signal is treated inside a process in VHDL, and does not cause additional storage nor delays.
Multiple writes are allowed in the same clock cycle, provided that they happen in the same process, as that simplifies initializing values and then updating, as is commonly done in HDLs. In the C# implementation, the signals are stored in a three-layer buffer, to ensure all writes and reads adheres to these rules.
A bus is described in the program by an `interface` that inherits from the `IBus` interface and defines the names and types of contained signals as properties on the interface. When the interface is used, the library will automatically create an instance that implements the timed logic rules, and expose it through the interface, such that the user never writes the implementation of a bus, only the definition. An example of a bus definition is shown in listing 1, which could be used as the write interface for an external memory component. In this example, the `IntialValue` attribute is used to simplify the logic, by having a value assigned to the `WriteEnable` signal on startup and reset.

```
interface IMemory : IBus
{
    [InitialValue(false)]
    bool WriteEnable { get; set; }
    ushort Address { get; set; }
    uint Value { get; set; }
}
```

Listing 1 Example of a bus definition

3.2 Processes

A process in SME is implemented as a class that derives from the `Process` class, and implements a `Run` method. The implementation is based on the `async/await` feature in C#, as we have experimented with this feature and found that it scales to a very high number of processes. Whenever the process has completed all work for the current clock cycle, it can `await` the `ClockAsync()` method. This allows a more sequential process design than the normal *finite state machine* approach normally used in HDLs.
However, since it is very common to perform the same operation in each clock cycle, there is also a `SimpleProcess` method with an `OnTick` method that wraps the logic, and guarantees that the code is called exactly once per cycle.
A process can access a bus by simply declaring a field or property with the desired interface type. An example of a process that writes the bus from listing 1 is shown in listing 2. Note that the instance field `memory` is never assigned; it is automatically set during load, such that all processes using the same bus are connected.
Another detail is that there is no special `.read()` or `.next` method required for accessing the bus value as in Cx and MyHDL respectively.

```
[ClockedProcess]
class Memory : SimpleProcess
{
    readonly uint[] storage = new uint
        [1024*16];
    IMemory input;

    override void OnTick()
    {
        if (input.WriteEnable)
          storage[input.Address] = input.
              Value;
    }
}
```

Listing 2 Example of a simple clocked process

3.3 Clock semantics

Even though we are only interested in modeling clocked networks, we need to distinguish between clocked and un-clocked processes. Clocked processes are activated on the rising clock edge, right after all signals are propagated on all busses. If all processes were clocked, two modules communicating would need to delay their work by a clock cycle, as output from one would not be available in the same cycle. In turn, this would require the programmer to build large complicated processes, instead of composing the design of small testable and reusable modules.
For this reason we also support un-clocked processes, which will have its `Run` method executed once all processes that write its inputs have completed. This ensures that there will be no races or intermediate results propagated to the process, honoring the CSP model. For an FPGA design, clocked processes will be similar to a pipeline stage, and un-clocked processes similar to combinatorial logic.
It is also possible to declare a `Bus` clocked, which is semantically equivalent to having a clocked process reading and writing the signals on the clock edge.

3.4 Timed logic in processes

The program running inside the SME process is not restricted and will naturally run in a sequential fashion, just as any other C# program would. However, it might be beneficial to also use timed logic inside the process, for instance, to increment a local counter value in each clock cycle. This can be achieved by creating a clocked `Bus` inside the process, and then only using it from within the process. As this is a common scenario, it is also possible to put an `InternalBus` attribute the local field, which ensures that the bus only propagates on the rising clock edge as well as disabling the automatic pairing with other processes that use the same interface.

3.5 Loading the network

The intended way to use SME is to create a normal .Net library that contains only code that is used for the SME model. The user can then call a loader function, and it will automatically find all classes that implement the `IProcess interface` in the library. For each such class it discovers, it will examine all fields and construct a *dynamic proxy* instance for each field that is derived from `IBus`. Fields that are of the same type in different processes will be automatically paired such that they both point to the same *dynamic proxy instance*.

Once the network is fully loaded, the SME library will create a dependency graph, using a simple algorithm. Initially, all clocked processes are considered to have all input dependencies met and are added to the graph.

1. All busses where all writers are in the graph are marked as ready.
2. All processes where all inputs are marked ready are added to the graph.
3. Repeat until no processes are added to the graph.

This simple algorithm ensures that there are no cycles, or mutual dependencies in the network.

3.6 Simulation

Once the execution graph has been built, the simulation merely walks the graph in breadth-first order. After each process has been executed, the simulation checks that the process has not written any of its input busses, and signals the output busses, such that conflicting writes are detected. After all items in the graph are executed, all busses are instructed to propagate their values and clear the conflicting write states.

3.7 Software programmer perspective

Since the SME library is based on C# and running on the .Net runtime, we are tapping into an existing eco-system. This eco-system includes, developers, compilers, integrated development environments, debuggers, support libraries, etc. The SME library itself is designed for modeling hardware processes, but it is not limited to FPGA designs or constructs. The processes are suitable for any system that uses synchronicity, and can use any functionality from C#, including graphics output, dynamic lists, files, network communication, database lookups, etc. As an example, we implemented the classic Frogger game in an earlier version of SME [16], and found it to be a good fit due to the clock-based logic found in games.

This versatility can be very useful, particularly in the early design stages, where it might be easier to us a dictionary, dynamic list or test database, when examining the viability of a possible solution model.

For a software programmer, this approach is much less automatic than high-level synthesis, in that it requires the programmer to explicitly declare parallelism and dependencies. The motivation for this design choice is that it more closely follows that of physical hardware, and thus enables a more accurate description of the design.

The most alien item is likely the timed logic enforced by the busses. It can be explained as a synchronization construct, that allows race-condition free communication between processes without requiring semaphores or similar low level constructs. It can also be explained as a group of flip-flops, or simply a time-based propagation mechanism, but it requires that the user is able to understand the program as a set of parallel processes.

Either way, since it is a regular construct in the runtime, the debugger can assist with variable inspection, and help the programmer understand why the program behaves as it does. We have also added a number of checks to ensure that the user does not violate the communication rules in a way that would be inconsistent with an FPGA implementation.

Compared to HLS approaches, the SME method only requires knowledge about designing process based programs, which we assume is familiar, as it is similar to how threads are implemented, for example in the Java programming language. As we discuss in the next section, the need for understanding the low-level hardware details cannot be fully abstracted when targeting FPGAs.

The SME library is published under the open source LGPL license on GitHub [17]. The part of the SME library that emits VHDL from the model is not currently published.

4 Generating VHDL

We initially developed and used SME as a tool for modeling circuits, while re-implementing the designs in VHDL by hand. We quickly noticed that this was a tedious process as we often tested an implementation in VHDL and then went back to SME and changed something, and then needed to update the VHDL to match. This workflow quickly became a source of errors and it became difficult to synchronize the implementation and the model. We then decided to build a translation process, which allowed us to build the VHDL implementation directly from the SME model. This translation process has a number of limitations, similar to how VHDL has a synthesizable subset, and reflects both limitations in our tool as well as limitations in the FPGA.

4.1 Structure

Each process in SME maps largely to an `Entity` in VHDL, so we started out by making a simple translation that set up each process as an `Entity`. Since the grouping performed by busses does not have an equivalent in VHDL, we simply made a flat list of all signals as input and output ports in the VHDL file.

The input and output attributes on the busses are verified during the simulation, so we extracted those, and used them to build the sensitivity lists for the processes. For clocked busses we used the standard VHDL template with a single `process` handling the clock and reset signals, and then leaving another `process` empty for the actual contents.

To make sure everything is connected the same way as in the SME model, we then produce a *top-level* file, which maps all signals as expected. With this, we were able to generate the structure of the SME network in VHDL files.

4.2 Type system

When building FPGA designs from a high-level language, there are always some problematic issues surrounding types. General-purpose CPUs are optimized for handling values in the common 8, 16, 32, and 64 bit sizes, and are

so efficient at this that there is rarely any reason to choose another size. For FPGA designs this is not the case, as a smaller bit width can lead to a significantly more efficient implementation.

The Cx language has the type system as one of their key features, stating that using a general-purpose language is poorer at utilizing the types [13]. While that is correct, we argue that this is less of a problem, and is outweighed by the benefits from having a general-purpose language for writing test suites and everything else.

We expect that most software developers would start out using the common sizes for their variables, and then later fine-tune the implementation to use the smallest fitting size. For this we have built-in support for the common sizes, in both signed and unsigned variants, such that the VHDL is generated with types matching the expected .Net values. The data types in VHDL map to a `std_logic_vector` of the appropriate size, but uses an alias so it is clear from reading the VHDL what the user specified.

For sizes that are different from the common ones, we also supply helpful implementations that work the same in .Net and VHDL, with 2 to 64 bits, both signed and unsigned. There is currently no support for accessing individual bits in the values, but that can be achieved through bitwise operators.

The type system also allows the user to specify a custom VHDL type, using attributes. The user needs to supply a type that works the same in VHDL and .Net, and can then simply supply the VHDL name in the .Net attribute.

The C# type `struct` is also supported, implemented with a `record` in VHDL.

4.3 Functionality

For the code itself, i.e. the content of the `Run` method, we use the underlying intermediate language, IL. Since C# is a compiled language, we load the method at runtime and examine only the compiled IL. From the IL we use the NRefactory library to build an abstract syntax tree, an AST, which we transform to undo some of the compiler transformations, so we end up with a higher level C# like representation, containing loops, if statements, and similar items in a tree-like structure.

While it seems backwards to compile and then de-compile, it helps with supporting multiple languages. At the same time it makes programs more homogenous as the compiler will erase some of the coding style, leaving a representation that is easier to analyze.

We support most of the IL constructs, such as if statements, assignments, logical operators, arithmetic operators, bitwise operators, assignments, case statements, etc. Loops are only supported in the most primitive cases, where the number of iterations can be determined statically.

During the code generation, the type system keeps track of the data types, such that it can correctly pair signed and unsigned operations, sign extend values, etc.

Function calls are also supported, provided that they are implemented in the same class as the method calling them and declared `static`. If the input arguments are arrays, the user can supply the array sizes through attributes, or rely on the VHDL type system to automatically resolve the sizes from the arguments.

4.4 Custom code

Even though the system allows low-level design of the FPGA, there are often cases where a tool with a higher abstraction hinders the optimal solution. For such cases, we allow the user to suppress generation of VHDL code, either for the entire process, for a specific method, or for the entire `Run` method. It is also possible to augment the generated VHDL code, thus allowing minor overrides to the generated VHDL. If the transformation to VHDL fails for some reason, usually because an unsupported feature is used, the resulting VHDL will appear as if the user has suppressed code generation.

When code generation is not complete, the resulting VHDL program will not compile, and the user must remedy this by supplying the missing code by hand. To make this as seamless as possible we support editing the generated VHDL files directly, as opposed to having code elsewhere and then merging it in. This is implemented by having strategically placed comment regions in the VHDL code, that are extracted prior to overwriting a file. With this approach, it is much simpler to see what the code performs in the context of the related program. It also makes it less likely that the user changes some code in the VHDL file, only to have it overwritten by the next rebuild.

4.5 Test benches

During the simulation phase, it is possible to get a trace file written, that records each signal on each bus for each clock cycle. With this information, it is possible to compare the simulation with the FPGA implementation, and quickly pinpoint which value differs when. To further automate this process, the SME library also generates a VHDL test bench that reads such a trace file, and for each clock cycle compares the signal values with those found in the trace file.

This makes it simple to verify that the model and implementation are equivalent, and makes it simple to utilize the flexibility in the C# language to write a large number of tests, e.g. parameter sweeps, and then confirm these in the VHDL simulator.

If the user decides to hand-tune parts of the FPGA design, the test bench can be used to verify that the new implementation is functionally equivalent to the previous. This makes it easy to have continuous integration testing of the design.

4.6 Arrays

Naturally, only fixed size arrays are supported in FPGA logic, but C# does not support fixed size arrays. To work around this implementation, we support only statically allocated arrays, created with the `readonly` keyword in C#. With arrays created statically, we can observe the size and produce the correct statically sized arrays as required for

VHDL.

4.7 IP Integration

SME is a new library, so naturally no third-party components exist, but since a large collection of components exist for FPGA designs, we would like to support these as well. As the implementation may already exist in HDL, we have added support for writing a functionally equivalent component in C#, and then simply mapping to this component when generating VHDL. We are using this mechanism to support the configurable ram blocks found on Xilinx devices, as well as partial support for the `DSP48E1`, also found on Xilinx devices. This shows that it is indeed possible to support third-party IP, but it may be a non-trivial task to write a functional equivalent process in C#, even though it is possible to use all features in the C# language for the implementation.

4.8 Limitations

Unlike the simulation part described in section 3, the VHDL generation is not able to utilize the full range of the C# library and CIL runtime. As described in section 4.6, only static arrays are supported, which means that most existing libraries are not supported for VHDL generation. When translating to VHDL, any code outside the loaded assembly will be ignored, leaving the VHDL with calls to methods that does not exist. This is deliberate in that it allows the developer to fix it by either providing proper synthesizable methods C#, or to simply supply those in VHDL, for example through an existing IP block.

For code lines, loops or variable types that cannot be translated into VHDL, the generator will output the problematic lines as comments into the VHDL, such that the programmer can either change the C# source, or provide an equivalent VHDL implementation directly in the generated file.

Recursive methods are supported, in the sense that they are simply reproduced *as-is* in the VHDL output.

If we wanted to enable all the entire C# language and all parts of the runtime, we would need to implement the Virtual Execution System, with all the dynamic allocations, garbage collection etc. This approach has been investigated with a co-execution setup where parts of the code runs on a host and parts on an FPGA device [18]. While such an approach has some obvious benefits, we consider it to yield sub-optimal FPGA designs, as the code would likely be sequential, and thus only parallelism that can be automatically infered can be used.

The generated VHDL can be compiled into a bitstream and transferred onto an actual FPGA device. However, the curent implementation requires that the programmer manually chooses system properties inside the vendor tools, such as which FPGA to target, what clock frequency to use, layout strategies, port/pin mappings etc.

4.9 Software programmers perspective

From a software programmers perspective, generating VHDL from the C# description can provide a significant productivity boost, when compared to developing directly in a HDL. However, there are currently many limitations to the generated output, that may cause some frustation for a software programmer.

We expect a software programmer to understand the process oriented nature of SME fairly quickly, but understanding the limitations of FPGA design are expected to be more challenging.

4.10 Comparison to other approaches

The SME approach is many ways comparable to MyHDL, Cx and SystemC, in that neither performs automatic parallelization of sequential code, but instead relies on the programmer to divide the program into individual communicating components.

Where SystemC, Cx, and the related ESys.Net [19] approaches are very verbose in terms of details such as signal sensitivity and bit-width, the SME approach is more software-developer-oriented, where direct signal dependencies are infered, and bit-width specifications are optionally specified. This is a trade-off, where engineers will likely prefer the explict nature with direct control, and we consider that software developers will prefer the implicit defaults.

Where SystemC defines a sort of *domain specific language* within C++ through macros and template constructs, Cx takes it one step further and defines a complete new language. Both these approaches allow a great deal of control, but at the expense of requiring the programmer to learn a new language. The MyHDL and SME approaches leverage an existing language, such that a programmer familiar with the languages can focus only on the logic. For SystemC, MyHDL, and SME, they are based on existing languages, and as such not all constructs allowed in the language are valid in the context of FPGA designs.

The Kiwi system [20] is similar to SME, as it is attempts to convert C# code to Verilog. Kiwi leverages common software parallelism constructs, such as threads and semaphores, to model the parallelism, which has the benefit that the programs contain some degree of parallelism and can be used on general purpose hardware simultaneously with FPGAs [21].

Where the other systems are based on the concept of a signal, usually modelled through a variable or channel-like instance, the SME approach has groups of signals in a bus. Also, unlike the other approaches, there is no explicit *read* or *write* calls in SME.

The SME system currently only considers a single clock, and only the raising edge of this, where the other approaches can interact with the clock in more detailed ways.

5 Examples

Performing a quantitative measurement of SME is difficult, because the output is tightly follows the input, thus the users design choices in the model reflect how efficient the VHDL output is. Comparing SME to HLS is also non-

trivial as the efficiency is highly dependent on the HLS programmer being able to choose the correct annotations. Instead, we focus on how closely we are able to follow the SME design choices in the VHDL result.
The full source code for all the examples can be found in the Examples folder on the SME website [17].

5.1 Memory component

The memory component is an example that is chosen for its simplicity, rather than being a useful item. To simulate a memory component in software, we can simply use a fixed size array, and then read and write from the array.
In the example shown in listing 3 we set up a dual port memory interface, such that the component can be read a written.

```
namespace Tester
{
  interface IMemoryInterface : IBus
  {
    [InitialValue(false)]
    bool WriteEnabled { get; set; }
    [InitialValue(false)]
    bool ReadEnabled { get; set; }

    uint ReadAddr { get; set; }
    uint WriteAddr { get; set; }

    ulong WriteValue { get; set; }
    ulong ReadValue { get; set; }
  }

  class SimpleMockMemory : SimpleProcess
  {
    [InputBus, OutputBus]
    IMemoryInterface Interface;

    readonly ulong[] m_data = new ulong
        [1024];

    override void OnTick()
    {
      DebugOutput = true;

      if (Interface.ReadEnabled)
        Interface.ReadValue = m_data[
            Interface.ReadAddr];

      if (Interface.WriteEnabled)
        m_data[Interface.WriteAddr] =
            Interface.WriteValue;
    }
  }
}
```

Listing 3 Memory interface

To test the component, we write a simple test bench that writes a value to the memory and read it back, as shown in listing 4. Since we do not intent to translate this component to VHDL, we could easily read files with test patterns, add loops, or use random numbers.

```
[ClockedProcess]
class MemoryTester : Process
{
  [InputBus, OutputBus]
  IMemoryInterface Interface;

  async override Task Run()
  {
    await ClockAsync();
    Interface.WriteAddr = 22;
    Interface.WriteValue = 32;
    Interface.WriteEnabled = true;

    await ClockAsync();
    Interface.ReadAddr = 22;
    Interface.WriteEnabled = false;
    Interface.ReadEnabled = true;

    await ClockAsync();
    Assert(Interface.ReadValue == 32);
  }
}
```

Listing 4 Memory interface test bench

The generated VHDL code is shown in listing 5, and illustrates how the names and types are transferred from the C# model to the VHDL implementation. The shown result has been edited to better fit the page layout, and we have also removed the regions where user code can be inserted.

```
entity Tester_SimpleMockMemory is
  port(
    IMemoryInterface_WriteEnabled: in
        T_SYSTEM_BOOL;
    IMemoryInterface_ReadEnabled: in
        T_SYSTEM_BOOL;
    IMemoryInterface_ReadAddr: in
        T_SYSTEM_UINT32;
    IMemoryInterface_WriteAddr: in
        T_SYSTEM_UINT32;
    IMemoryInterface_WriteValue: in
        T_SYSTEM_UINT64;
    IMemoryInterface_ReadValue: out
        T_SYSTEM_UINT64;
    RST : Std_logic;
    CLK : Std_logic
  );
end Tester_SimpleMockMemory;

architecture RTL of Tester_SimpleMockMemory
    is
  subtype m_data_type is
      T_SYSTEM_UINT64_ARRAY(0 to 1024 - 1);
begin
  process(CLK, RST)
  begin
    if RST = '1' then
    elsif rising_edge(CLK) then
    end if;
  end process;


  process(RST)
    variable m_data : m_data_type;
  begin
    if RST = '1' then
      IMemoryInterface_ReadValue <=
          STD_LOGIC_VECTOR(TO_UNSIGNED(0,
          T_SYSTEM_UINT64'length));
      m_data := (others => (others => '0'));
    else
      if IMemoryInterface_ReadEnabled = '1'
          then
        IMemoryInterface_ReadValue <= m_data(
            TO_INTEGER(UNSIGNED(
            IMemoryInterface_ReadAddr)));
```

```
      end if;
      if IMemoryInterface_WriteEnabled = '1'
          then
        m_data(TO_INTEGER(UNSIGNED(
            IMemoryInterface_WriteAddr))) :=
            IMemoryInterface_WriteValue;
      end if;
    end if;
  end process;
end RTL;
```

Listing 5 Memory component VHDL output

5.2 Financial trading algorithm

In high frequency trading, a simple approach is to compute two averages for prices, one with a short interval and one with a long, then detect when the two averages cross [22]. The averages can be implemented in many ways, where a FIR filter would be an obvious choice. In our implementation, shown in reduced form in listing 6, we chose to implement the two averages as exponential weighted moving averages, as that is more optimal in terms of required storage. To avoid floating point logic, we chose the decay values to be four and eight respectively.

```
var newShortValue = (Input.Value >> 2) +
  (Internal.ShortValue >> 2) * (4 - 1);

var newLongValue = (Input.Value >> 3) +
  (Internal.LongValue >> 3) * (8 - 1);

if (Internal.StartCounter <
    STARTUP_VALUE_COUNT)
{ Internal.StartCounter++; }
else
{
  Output.GoingDown = newLongValue >
      newShortValue && Internal.LongValue <=
      Internal.ShortValue;
  Output.GoingUp = newLongValue <
      newShortValue && Internal.LongValue >=
      Internal.ShortValue;
  Output.Valid = true;
}

Internal.ShortValue = newShortValue;
Internal.LongValue = newLongValue;
```

Listing 6 Exponential Weighted Moving Average trader

The generated VHDL is displayed in listing 7.

```
num := STD_LOGIC_VECTOR((shift_right(UNSIGNED
    (ITraderInput_Value), 2)) + (resize((
    shift_right(UNSIGNED(IInternal_ShortValue
    ), 2)) * TO_UNSIGNED(3, 32), 32)));
num2 := STD_LOGIC_VECTOR((shift_right(
    UNSIGNED(ITraderInput_Value), 3)) + (
    resize((shift_right(UNSIGNED(
    IInternal_LongValue), 3)) * TO_UNSIGNED
    (7, 32), 32)));
if UNSIGNED(IInternal_StartCounter) <
    TO_UNSIGNED(10, 32) then
  IInternal_StartCounter <= STD_LOGIC_VECTOR(
      UNSIGNED(IInternal_StartCounter) +
      TO_UNSIGNED(1, 32));
else
  if (UNSIGNED(num2) > UNSIGNED(num)) and (
      UNSIGNED(IInternal_LongValue) <=
      UNSIGNED(IInternal_ShortValue)) then
    ITraderOutput_GoingDown <= '1';
  else
    ITraderOutput_GoingDown <= '0';
  end if;
  if (UNSIGNED(num2) < UNSIGNED(num)) and (
      UNSIGNED(IInternal_LongValue) >=
      UNSIGNED(IInternal_ShortValue)) then
    ITraderOutput_GoingUp <= '1';
  else
    ITraderOutput_GoingUp <= '0';
  end if;
  ITraderOutput_Valid <= '1';
end if;
IInternal_ShortValue <= num;
IInternal_LongValue <= num2;
```

Listing 7 Trader core method in VHDL

5.3 AES encryption

For a more complicated example, we have implemented AES256 encryption in CBC mode. Rather than writing our own AES function, we have simply used an open source C# version, written by the Mono project [23].

With the core AES implementation in C#, we wrote a small interface that controls input, such that keys and the IV can be loaded, and data be encrypted. For the CBC operation, it is then simply a matter of applying the XOR operation on the output and storing it as the next IV.

The C# source code for the implementation is shown in listing 8, not including the implementation of the AES functions that were copied verbatim from the Mono project. Parts of the VHDL is shown in listing 9, to illustrate the mapping between the C# source and the VHDL.

```
if (Input.LoadKey)
{
  // Removed for read-ability
}
else if (Input.DataReady)
{
  UnpackLongToArray(m_input,0, Input.Data0);
  UnpackLongToArray(m_input,8, Input.Data1);

  for(var i = 0; i < 16; i++)
    m_input[i]= (byte)(m_input[i] ^ m_iv[i]);

  Encrypt128(m_input,m_output,m_expandedKey);

  for(var i = 0; i < 16; i++)
    m_iv[i]= m_output[i];

  Output.Data0= PackArrayToLong(m_output, 0);
  Output.Data1= PackArrayToLong(m_output, 8);
  Output.DataReady = true;
}
```

Listing 8 AES CBC implementation in C# SME

```
if IInput_LoadKey = '1' then
  -- Removed for read-ability
else
  if IInput_DataReady = '1' then
    UnpackLongToArray(m_input,
        STD_LOGIC_VECTOR(TO_UNSIGNED(0,
        T_SYSTEM_UINT8'length)), IInput_Data0
        );
```

```
    UnpackLongToArray(m_input,
        STD_LOGIC_VECTOR(TO_UNSIGNED(8,
        T_SYSTEM_UINT8'length)), IInput_Data1
        );
    for i in 0 to 15 loop
      m_input(i) := m_input(i) xor m_iv(i);
    end loop;
    Encrypt128(m_input, m_output,
        m_expandedKey);
    for j in 0 to 15 loop
      m_iv(j) := m_output(j);
    end loop;
    IOutput_Data0 <= PackArrayToLong(m_output
        , STD_LOGIC_VECTOR(TO_UNSIGNED(0,
        T_SYSTEM_UINT8'length)));
    IOutput_Data1 <= PackArrayToLong(m_output
        , STD_LOGIC_VECTOR(TO_UNSIGNED(8,
        T_SYSTEM_UINT8'length)));
    IOutput_DataReady <= '1';
  end if;
end if;
```

Listing 9 AES CBC core method in VHDL

6 Conclusion

We have presented the design of a library using Synchronous Message Exchange as a modeling tool for hardware designs. For SME models that use only features that are statically allocated, it is also possible to automatically generate VHDL implementations. Built-in device components, custom types and third-party IP is supported, by supplying a C# implementation of the functionality to be used in the simulation. By providing an automated test bench, it is possible to continuously verify that the generated code is functionally equivalent to the simulated code. Combining the test bench with a mechanism for augmenting the generated code with hand-written VHDL, enables gradual refinement from the simulation while retaining continuous verification of the correctness.

7 Literature

[1] Martin C Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. Achieving high performance with fpga-based computing. *Computer*, 40(3):50, 2007.

[2] Peter Bishop. A tradeoff between microcontroller, dsp, fpga and asic technologies. *EE Times design*, 2009.

[3] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 35–35. IEEE, 2005.

[4] Jan Decaluwe. Myhdl: a python-based hardware description language. *Linux journal*, 2004(127):5, 2004.

[5] Rinse Wester. A transformation-based approach to hardware design using higher-order functions. 2015.

[6] Xilinx Inc. Vivado hls overview. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`. [Online; accessed June 2016].

[7] Altera corporation. Altera sdk for opencl. `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.highResolutionDisplay.html`. [Online; accessed June 2016].

[8] Utpal Banerjee. *Data dependence in ordinary programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1976.

[9] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *ACM SIGPLAN Notices*, volume 35, pages 321–333. ACM, 2000.

[10] Xilinx Inc. Xilinx SDSoC. `http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html`. [Online; accessed July 2016].

[11] PLDA Group. Quickplay. `https://www.quickplay.io`. [Online; accessed July 2016].

[12] Accellera. SystemC. `http://accellera.org/downloads/standards/systemc`. [Online; accessed July 2016].

[13] Cx language. `http://cx-lang.org`. [Online; accessed July 2016].

[14] Brian Vinter and Kenneth Skovhede. Bus centric synchronous message exchange for hardware designs. *Proceedings of Communicating Process Architectures 2015*, 2015.

[15] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[16] Brian Vinter and Kenneth Skovhede. Synchronous message exchange for hardware designs. *Proceedings of Communicating Process Architectures 2014*, 2014.

[17] K. Skovhede. Sme source code. `https://github.com/kenkendk/sme`. [Online; accessed July 2016].

[18] S Srinath. *IMPLEMENTATION OF THE .NET CLR ON FPGAs*. PhD thesis, ANNA UNIVERSITY CHENNAI, 2006.

[19] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, François R Boyer, JP David, and Guy Bois. Esys. net: a new solution for embedded systems modeling and simulation. In *ACM SIGPLAN Notices*, volume 39, pages 107–114. ACM, 2004.

[20] Satnam Singh and David J Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, pages 3–12. IEEE, 2008.

[21] David Greaves and Satnam Singh. Distributing c# methods and threads over ethernet-connected fpgas using kiwi. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 1–9. IEEE, 2011.

[22] Abdalla Kablan and Joseph Falzon. The use of dynamically optimised high frequency moving average strategies for intraday trading. *Word Academy of Science, Engineering and Technology*, 2012.

[23] Mono Project Sebastien Pouliot et al. Open source AES encryption in C#. `https://github.com/mono/mono/blob/ef407901f8fdd9ed8c377dbec8123b5afb932ebb/mcs/class/System.Core/System.Security.Cryptography/AesTransform.cs`. [Online; accessed July 2016].