Microprocessors and Microsystems 38 (2014) 76–81

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A SystemC library for specifying pipeline abstractions ☆

Ed Harcourt [a,*], James Perconti [b]

[a] Department of Computer Science,[1] St. Lawrence University, Canton, NY, United States
[b] Department of Computer Science, Northeastern University, Boston, MA, United States

ARTICLE INFO

Article history:
Available online 25 November 2013

Keywords:
Pipeline
SystemC
Domain specific languages
System modeling

ABSTRACT

We describe a SystemC library for specifying, modeling, and simulating hardware pipelines. The library includes a set of overloaded operators defining a *pipeline expression language* that allows the user to quickly specify the architecture of the pipeline. The pipeline expression is used to derive the connectivity of the SystemC modules that define the stages of the pipeline and to automatically insert latches and control modules between the stages to handle the proper routing of transactions through pipeline. Using the SystemC simulator the pipeline can then be simulated and evaluated. The pipeline expression language sits on top of SystemC, exposes all of the features of C++ and SystemC enabling the user to specify, evaluate, and analyze pipeline architectures.

1. Introduction

Pipelining is an important and ubiquitous hardware implementation technique for increasing the performance of hardware. Pipelining works by taking a function and partitioning it into *stages* that operate concurrently thereby allowing multiple uses of the function to execute in parallel. Stages can be configured in a variety of ways from simple linear pipelines to pipelines that have stages that fork, merge, or contain feedback loops. Being able to quickly model and evaluate various pipeline configurations in an existing modeling framework such as SystemC is beneficial.

We describe `sc_pipes`, a SystemC library that defines a set of operators useful for quickly specifying and modeling hardware pipelines in SystemC. Our library also instantiates latches between stages to properly handle the routing of data through the pipeline. Using this library the modeler develops an expression on how those stages interact.

We use SystemC [1,2] as our simulation framework because of its support for system level modeling and simulation and because it is embedded in C++, a general purpose programming language with support for generic, polymorphic, object oriented programming. Furthermore C++ is suitable for constructing domain specific languages (DSLs) [3].

1.1. Outline

After reviewing related work in the next section we will introduce the pipeline expression language in Section 3 along with examples that use each of the operators. In Section 4 we describe how a pipeline expression is elaborated into SystemC modules. We close the paper by giving a realistic example of a pipelined floating-point adder in Section 5 and conclude in Section 6.

2. Related work

Excellent overviews of pipelining, hardware implementation techniques, and taxonomies are described in [4,5]. The compiler research community has developed high-level notations for pipelines to generate instruction schedulers [6,7]. Our notation is inspired by that of [7]. The work in [8–10] describes notations for specifying pipelines for downstream tools. Mishra and Dutt [11] describe how to validate a pipeline specification written in the architectural description language *expression* [12]. Petri Nets [13], Process Algebras [14,15], and Higher-Order Logic [16,17] have been used to formally model and simulate pipelines with all of the benefits that formal notations bring such as verification and proving properties about the models. The work in [20] automatically generates a pipelined datapath, but the specification language is not rooted in an existing general purpose framework such as SystemC. An earlier version of this paper where the pipeline expression language and many of the operators were not yet fully developed appeared in [18].

Our research builds on the above work in two fundamentally different ways. The first is the set of pipeline operators we define to construct our pipeline expression language. In [18] the operators for forking were not fully developed and it used a static typing scheme on ports as opposed to the more flexible dynamic typing

☆ This material is based upon work supported by the National Science Foundation under Grant No. 0959713.
[1] Department of Mathematics, Computer Science, and Statistics.
* Corresponding author.
E-mail addresses: edharcourt@stlawu.edu (E. Harcourt), jtpercon@ccs.neu.edu (J. Perconti).

0141-9331/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2013.11.010

on ports in sc_pipes. The second is how the overloaded operators define a notation that is itself embedded in the system simulation language SystemC, which is itself embedded in the general purpose programming language C++. As [19] so aptly points out external domain specific languages (DSLs) that are not embedded in a general purpose language "tend to have short life-spans due to limited support and portability, suffer from a lack of tools (particularly debuggers), and it is usually impossible to use two DSLs in the same source file."

3. Pipeline operators

Pipelines specified in sc_pipes can be quite complex including pipelines with feedback loops, and stages that fork or merge. Beginning with simple linear pipelines we'll examine how pipelines are specified and modeled using sc_pipes. The pipeline expression language is implemented by overloading the C++ operators +, *, /, %, |, and >>. We explore each of these in the following sections.

3.1. Linear pipelines

In a linear pipeline stages are connected sequentially such that stage n receives its input from stage $n-1$ and sends its output to stage $n+1$ with the input for the first stage and the output of the final stage managed by the testbench. For example, consider a pipeline to compute the (somewhat arbitrary) function $f(x) = 2x^2 + 4x - 7$. This could be implemented by any number of architectures. Fig. 1 shows three possibilities; there are more. Only the top two pipelines are linear. The third pipeline could compute the more general function of $nx^2 + 4x - 7$ where n is the number of times the first stage would need to be repeated.

In sc_pipes we specify the functionality of each stage as a SystemC module using the special purpose sc_pipes port types scp_in and scp_out (as opposed to the SystemC port type sc_in and sc_out). Fig. 2 shows a generic SystemC module for a stage for the pipelines in Fig. 1. The constructor takes an object that implements a C++ functor interface Function that overloads the function call operator (). This nicely encapsulates and abstracts the function being called on line 8. Lines 2-3 in Fig. 2 declare an input port and an output port for the stage. In sc_pipes ports are untyped and generic but reads and writes to the ports are typed (lines 7 and 9). The port types scp_in and scp_out are template specializations for the standard SystemC ports sc_in and sc_out. Lines 12-17 are boilerplate, declaring the module constructor and that the stage is sensitive to changes on the input port in.

To specify the first pipeline from Fig. 1 the user declares the stages and ties them together with a *pipeline expression*.

```
stage s1(f1), s2(f2), s3(f3);
scp_pipeline p("simple", s1 >> s2 >> s3);
```

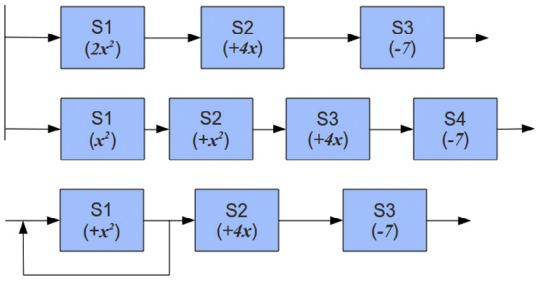

Fig. 1. Three different pipeline architectures for computing $2x^2+4x-7$.

```
1  struct stage : sc_module {
2    scp_in in;   // input
3    scp_out out; // output
4    Function f;  // function computed
5
6    // executed when input changes
7    void run() {
8      int x = in.typed_read<int>();
9      x = f(x);
10     out.typed_write<int>(x);
11   }
12
13   SC_HAS_PROCESS(stage);
14     stage(sc_module_name name, Function f) :
15     sc_module(name), f(f), in(), out() {
16     SC_METHOD(run);
17     sensitive << in;
18   }
19 };
```

Fig. 2. Implementation of a stage for the pipelines in Fig. 1.

The >> operator is an overloaded C++ operator and specifies that two pipeline stages are connected sequentially. A pipeline expression really specifies the path (or *route*) that transactions take through the pipeline with connectivity being automatically derived from the expression. By default every pipeline stage completes its operation in one clock cycle — though this can be changed with the delay operator |. The expression (s1 | 3) specifies that stage s1 takes three cycles to compute its result.

For stages with multiple inputs and outputs, ports are connected between stages positionally. If stage A has output ports out1, out2, etc. and stage B has input ports in1, in2, etc. then out1 is connected to in1 and so on. For any two adjacent stages connected sequentially the number of output ports on the first stage must equal the number of input ports on the second stage.

The sc_pipes library inserts all of the connectivity code to associate pipeline stages with SystemC modules connected through SystemC signals. The library also inserts latches to store intermediate results between stages. Latch insertion is complex and will be discussed in Section 4. It is straightforward to construct a SystemC testbench that instantiates the pipeline and drives it with sample data to see that the first result is delivered after three cycles and delivers a new result every clock cycle thereafter. Fig. 3 shows the output of the simulation as a space-time diagram for the three stage linear pipeline from Fig. 1.

To model the second linear pipeline in Fig. 1 where stages one and two are identical we need to ensure that SystemC instantiates two separate modules, one for each stage.

```
stage s1(f1), s2(f1), s3(f2), s4(f3);
scp_pipeline p("simple2", s1 >> s2 >> s3 >> s4);
```

Notice that stages s1 and s2 are declared to be distinct hardware stages that compute the same function f1 $(x) = x^2$.

3.2. Pipelines with feedback

The bottom pipeline in Fig. 1 contains a feedback loop on stage 1 that should be repeated twice to compute f. To model stage one