Computer-Based Tool and Method for Designing an Electronic Circuit and Related System

Inventors: John Rapp, Manassas, VA (US); Scott Hellenbach, Amisville, VA (US); T. J. Kurian, Manassas, VA (US); D. James Schooley, Manassas, VA (US); Troy Cherasaro, Culpeper, VA (US)

Correspondence Address:
GRAYBEAL JACKSON HALEY LLP
Suite 350
155-108th Avenue N.E.
Bellevue, WA 98004-5973 (US)

Assignee: Lockheed Martin Corporation

Publication Classification

Int. Cl.
G06F 17/50 (2006.01)

U.S. Cl. 716/18

Abstract

A computer-based circuit-design tool includes a front end, an interpreter coupled to the front end, and a generator coupled the interpreter. The front end receives symbols that define an algorithm, and the interpreter parses the algorithm into respective algorithm portions. The generator identifies a corresponding circuit template for each of the algorithm portions, each template defining a circuit for executing the respective algorithm portion, and interconnects the identified templates such that the interconnected templates define a circuit that is operable to execute the algorithm. As compared to prior design tools, this tool may decrease the time and effort required to design a circuit for instantiation on a programmable logic integrated circuit (PLIC) or on an application-specific integrated circuit (ASIC) by allowing one to construct the circuit from previously written templates that define previously tested and debugged circuits.
FIG. 6B
\[ Y = X_4 \cdot \cos(Z) + z^4 \cdot \sin(X) \]
<table>
<thead>
<tr>
<th>Algorithm portion</th>
<th>Templates</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(x) (portion 190 of equation(2))</td>
<td>114₁, . . . Latency: 10 clock cycles</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>z³ (portion 186 of equation(2))</td>
<td>114₂, . . . Latency: 10 clock cycles</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Cos(z) (portion 184 of equation(2))</td>
<td>114₄, . . . Latency: 10 clock cycles</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>• (multiplication portions 182 And 186 of equation(2))</td>
<td>114₅, . . . Latency: 10 clock cycles</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+ (portion 176 of equation(2))</td>
<td>114₆, . . . Latency: 5 clock cycles</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>√ (portion 170 of equation(2))</td>
<td>114₇, . . . Latency: 10 clock cycles</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIG. 9
COMPUTER-BASED TOOL AND METHOD FOR DESIGNING AN ELECTRONIC CIRCUIT AND RELATED SYSTEM

CLAIM OF PRIORITY

[0001] This application claims priority to U.S. Provisional Application Ser. Nos. 60/615,192, 60/615,157, 60/615,170, 60/615,158, 60/615,193, and 60/615,050, filed on Oct. 1, 2004, which are incorporated by reference.

CROSS REFERENCE TO RELATED APPLICATIONS


BACKGROUND

[0003] Electronics engineers often instantiate circuits, such as logic circuits, on programmable logic integrated circuits (PLICs) such as field-programmable gate arrays (FPGAs), and on application-specific integrated circuits (ASICs). Because an engineer typically configures with firmware the circuit components and interconnections inside of a PLIC, he can modify a circuit instantiated on the PLIC merely by modifying and reloading the firmware. An example of a computer architecture that exploits the ability to configure and reconfigure circuitry within a PLIC with firmware is described in U.S. Patent Publication No. 2004/0133763, which is incorporated herein by reference.

[0004] But unfortunately, it is often difficult and time consuming to design a circuit for instantiation on a PLIC, and an increase in the level of design difficulty and the time required to complete the design often accompany the routing resources, component density, and component variety on a PLIC.

[0005] Comparatively, when a software programmer writes source code for a software application, he can often save time by incorporating into the application previously written and debugged software objects from a software-object library. Suppose the programmer wishes to write a software application that solves for y in the following equation:

\[ y = x^2 + z^2 \]  

Further suppose that a software-object library includes a first software object for squaring a value (here x), a second software object for cubing a value (here z), and a third software object for summing two values (here \( x^2 \) and \( z^2 \)). By incorporating pointers to these three objects in the source code, a compiler effectively merges these objects into the software application while compiling the source code. Therefore, the object library allows the programmer to write the software application in a shorter time and with less effort because the programmer does not have to “reinvent the wheel” by writing and debugging pieces of source code that respectively square x, cube z, and sum \( x^2 \) and \( z^2 \). Furthermore, if the programmer needs to modify the software application, he can do so without modifying and re-debugging the first, second, and third software objects.

[0006] In contrast, there are typically no time- or effort-saving equivalents of software objects available to a hardware engineer who wishes to design a circuit for instantiation on a PLIC; consequently, when a hardware engineer designs a circuit for instantiation on a PLIC, he typically must write the source code (e.g., Verilog Hardware Description Language (VHDL)) “from scratch.” Suppose that an engineer wishes to design a logic circuit that solves for y by equation (1). Because there are typically no hardware equivalents of the first, second, and third software objects described in the preceding paragraph, the engineer may write source code that describes first and second portions of a circuit for solving equation (1). The first circuit portion squares x, cubes z, and sums \( x^2 \) and \( z^2 \), and the second circuit portion interfaces the first circuit portion to the external pins of the PLIC. The engineer then compiles the source code with PLIC design tool (typically provided by the PLIC manufacturer), which synthesizes and routes the circuit and then generates the configuration firmware that, when loaded into the PLIC, instantiates the circuit. Next, the engineer loads the firmware into the PLIC and debugs the instantiated circuit. Unfortunately, the synthesizing and routing steps are often not trivial, and may take a number of hours or even days depending upon the size and complexity of the circuit. And even if the engineer makes only a minor modification to a small portion of the circuit, he typically must repeat the synthesizing, routing, and debugging steps for the entire circuit.

[0007] Another factor that may add to the time and effort that an engineer expends while designing a circuit for instantiation on a PLIC is that a PLIC design tool typically recognizes only hardware-specific source code. Suppose that a mathematician, who writes an equation using mathematical symbols (e.g., \( +, \times, \leq, \geq, \forall, \exists, =, \nexists, \oplus, \otimes, \vee, \wedge \)), wishes to instantiate on a PLIC a circuit that solves for a variable in a complex equation that includes, e.g., partial derivatives and integrations. Because a PLIC design tool typically recognizes few, if any, mathematical symbols, the mathematician often must explain the equation and the desired operating parameters (e.g., latency and precision) of the circuit to a hardware engineer, who then translates the equation and operating parameters into source code that the design tool recognizes. These explanation and translation steps are often time consuming and difficult for the engineer, particularly where the equation is mathematically complex or the circuit has stringent operating parameters (e.g., high speed, high precision).

[0008] Therefore, a need has arisen for a new methodology and for a new tool for designing a circuit for instantiation on a PLIC.

SUMMARY

[0009] According to an embodiment of the invention, a computer-based circuit design tool includes a front end, an interpreter coupled to the front end, and an integrator coupled to the interpreter. The front end receives symbols that define a logical expression, and the interpreter parses the expression into respective portions. The integrator identifies a corresponding circuit template for each of the expression portions, and logically interconnects the identified templates into a representation of an electronic circuit that is operable to execute the expression.

[0010] As compared to prior circuit design tools, such a tool may shorten the time and reduce the effort that an
engineer expends designing a circuit for instantiation on a PLIC by allowing the engineer to build the circuit from templates of previously designed and debugged circuits.

According to a related embodiment of the invention, the front end of the design tool recognizes mathematical symbols so that one can design a PLIC circuit for executing a mathematical expression with little or no assistance from a hardware engineer.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a peer-vector computing machine having a pipelined accelerator that one can design with a design tool according to an embodiment of the invention.

FIG. 2 is a block diagram of a pipeline unit that includes a PLIC and that can be included in the pipelined accelerator of FIG. 1 according to an embodiment of the invention.

FIG. 3 is a diagram of the circuit layers that compose the hardware interface layer within the PLIC of FIG. 2 according to an embodiment of the invention.

FIG. 4 is a block diagram of the circuitry that composes the interface adapter and framework services layers of FIG. 3 according to an embodiment of the invention.

FIG. 5 is a diagram of a hardware-description file for a circuit that one can instantiate on a PLIC according to an embodiment of the invention.

FIG. 6 is a block diagram of a PLIC circuit-template library according to an embodiment of the invention.

FIG. 7 is a block diagram of a circuit-design system that includes a computer-based tool for designing a circuit using templates from the library of FIG. 6 according to an embodiment of the invention.

FIG. 8 illustrates the parsing of a mathematical expression according to an embodiment of the invention.

FIG. 9 illustrates a table of hardwired-pipeline library templates corresponding to the hardwired-pipelines available for executing respective portions of the parsed mathematical expression of FIG. 8 according to an embodiment of the invention.

FIG. 10 is a block diagram of a circuit that the tool of FIG. 7 generates from circuit templates downloaded from the library of FIG. 6 according to an embodiment of the invention.

FIG. 11 is a block diagram of a circuit that the tool of FIG. 7 generates from circuit templates downloaded from the library of FIG. 6 according to another embodiment of the invention.

FIG. 12 is a block diagram of a circuit that the tool of FIG. 7 generates from circuit templates downloaded from the library of FIG. 6 according to yet another embodiment of the invention.

FIG. 13 is a block diagram of a circuit that the tool of FIG. 7 generates for implementing a function as a series expansion according to an embodiment of the invention.

FIG. 14 is a block diagram of a circuit that the tool of FIG. 7 generates for implementing the function of FIG. 13 as a series expansion according to another embodiment of the invention.

FIG. 15 is a block diagram of a power-of-x term generator that the tool of FIG. 7 generates as a replacement for the power-of-x multipliers of FIGS. 13 and 14 according to an embodiment of the invention.

FIG. 16 is a block diagram of a circuit that the tool of FIG. 7 generates for implementing another function as a series expansion according to an embodiment of the invention.

FIG. 17 is a block diagram of a sign determiner from FIG. 16 according to an embodiment of the invention.

INTRODUCTION

But first is presented in conjunction with FIGS. 1-6 an overview of concepts that are related to the design tool according to an embodiment of the invention. An understanding of these concepts should facilitate the reader's understanding of the design tool.

Overview of Concepts Related to Design Tool

FIG. 1 is a schematic block diagram of a computing machine 10, which has a peer-vector architecture according to an embodiment of the invention. In addition to a host processor 12, the peer-vector machine 10 includes a pipelined accelerator 14, which is operable to process at least a portion of the data processed by the machine 10. Therefore, the host-processor 12 and the accelerator 14 are “peers” that can transfer data messages back and forth. Because the accelerator 14 includes hardwired logic circuits instantiated on one or more PLICs, it executes few, if any, program instructions, and thus typically performs mathematically intensive operations on data significantly faster than a bank of computer processors can for a given clock frequency. Consequently, by combing the decision-making ability of the processor 12 and the number-crunching ability of the accelerator 14, the machine 10 has the same abilities as, but can often process data faster than, a conventional processor-based computing machine. Furthermore, as discussed below and in U.S. Patent Publication No. 2004/0136241, which is incorporated by reference, providing the accelerator 14 with a communication interface that is compatible with the interface of the host processor 12 facilitates the design and modification of the machine 10, particularly where the communication interface is an industry standard. And where the accelerator 14 includes multiple pipeline units (FIG. 2), providing each of these units with this compatible communication interface facilitates the design and modification of the accelerator, particularly where the communication interface is an industry standard. Moreover, the machine 10 may also provide other advantages as described in the following other patent publications, which are incorporated by reference: 2004/0133763; 2004/0181621; 2004/0170070; and, 2004/0130927.
[0032] Still referring to FIG. 1, in addition to the host processor 12 and the pipelined accelerator 14, the peer-vector computing machine 10 includes a processor memory 16, an interface memory 18, a bus 20, a firmware memory 22, an optional raw-data input port 24, an optional processed-data output port 26, and an optional router 31.

[0033] The host processor 12 includes a processing unit 32 and a message handler 34, and the processor memory 16 includes a processing-unit memory 36 and a handler memory 38, which respectively serve as both program and working memories for the processor unit and the message handler. The processor memory 36 also includes an accelerator-configuration registry 40 and a message-configuration registry 42, which store respective configuration data that allow the host processor 12 to configure the functioning of the accelerator 14 and the structure of the messages that the message handler 34 sends and receives.

[0034] The pipelined accelerator 14 includes at least one PLIC (FIG. 2) on which are disposed hardwired pipeline 44, 44', which process respective data while executing few, if any, program instructions. The firmware memory 22 stores the configuration firmware for the PLIC(s) of the accelerator 14. If the accelerator 14 is disposed on multiple PLICs, these PLICs and their respective firmware memories may be disposed on multiple circuit boards that are often called daughter cards or pipeline units (FIG. 2). The accelerator 14 and pipeline units are discussed further in previously incorporated U.S. Patent Publication Nos. 2004/0136241, 2004/0181621, and 2004/0130927. The pipeline units are also discussed below in conjunction with FIGS. 2-4.

[0035] Generally, in one mode of operation of the peer-vector computing machine 10, the pipelined accelerator 14 receives data from one or more software applications running on the host processor 12, processes this data in a pipelined fashion with one or more logic circuits that execute one or more mathematical algorithms, and then returns the resulting data to the application(s). As stated above, because the logic circuits execute few if any software instructions, they often process data one or more orders of magnitude faster than the host processor 12. Furthermore, because the logic circuits are instantiated on one or more PLICs, one can modify these circuits merely by modifying the firmware stored in the memory 52; that is, one need not modify the hardware components of the accelerator 14 or the interconnections between these components. The operation of the peer-vector machine 10 is further discussed in previously incorporated U.S. Patent Publication No. 2004/0133763, the functional topology and operation of the host processor 12 is further discussed in previously incorporated U.S. Patent Publication No. 2004/0181621, and the topology and operation of the accelerator 14 is further discussed in previously incorporated U.S. Patent Publication No. 2004/0136241.

[0036] FIG. 2 is a diagram of a pipeline unit 50 of the pipelined accelerator 14 of FIG. 1 according to an embodiment of the invention.

[0037] The unit 50 includes a circuit board 52 on which are disposed the firmware memory 22, a platform-identification memory 54, a bus connector 56, a data memory 58, and a PLIC 60.

[0038] As discussed above in conjunction with FIG. 1, the firmware memory 22 stores the configuration firmware that the PLIC 60 downloads to instantiate one or more logic circuits.

[0039] The platform memory 54 stores a value that identifies the one or more platforms with which the pipeline unit 50 is compatible. Generally, a platform specifies a unique set of physical attributes that a pipeline unit may possess. Examples of these attributes include the number of external pins (not shown) on the PLIC 60, the width of the bus connector 56, the size of the PLIC, and the size of the data memory. Consequently, a pipeline unit 50 is compatible with a platform if the unit possesses all of the attributes that the platform specifies. So a pipeline unit 50 having a bus connector 56 with thirty-two bits is incompatible with a platform that specifies a bus connector with sixty-four bits. Some platforms may be compatible with the peer vector machine 10 (FIG. 1), and others may be incompatible. Therefore, the platform identifier stored in the memory 54 may allow the host processor 12 (FIG. 1) to determine whether the pipeline unit 50 is compatible with the platforms supported by the machine 10. And where the pipeline unit 50 is so compatible, the platform identifier may also allow the host processor 12 to determine how to configure the PLIC 60 or other portions of the pipeline unit.

[0040] The bus connector 56 is a physical connector that interfaces the PLIC 60, and perhaps other components of the pipeline unit 50, with the pipeline bus 20 of FIG. 1.

[0041] The data memory 58 acts as a buffer for storing data that the pipeline unit 50 receives from the host processor 12 (FIG. 1) and for providing this data to the PLIC 60. The data memory 58 may also act as a buffer for storing data that the PLIC 60 generates for sending to the host processor 12, or as a working memory for the hardwired pipelines 44.

[0042] Instatiated on the PLIC 60 are logic circuits that compose the hardwired pipeline(s) 44 and a hardware interface layer 62, which interfaces the hardwired pipelines to the external pins (not shown) of the PLIC 60, and which thus interfaces the pipelines to the pipeline bus 20 (via the connector 56), the firmware and platform-identification memories 22 and 54, and the data memory 58. Because the topology of interface layer 62 is primarily dependent upon the attributes specified by the platform(s) with which the pipeline unit 50 is compatible, one can often modify the pipeline(s) 44 without modifying the interface layer. For example, if a platform with which the pipeline unit 50 is compatible specifies a thirty-two-bit bus, then the interface layer 62 provides a thirty-two-bit bus connection to the bus connector 60 regardless of the topology or other attributes of the pipeline(s) 44. Consequently, as discussed below in conjunction with FIGS. 7-10, an embodiment of the computer-based design tool allows one to design and debug the pipeline(s) 44 independently of the interface layer 62, and vice versa.

[0043] Still referring to FIG. 2, alternate embodiments of the pipeline unit 50 are contemplated. For example, the memory 54 may be omitted, and the platform identifier may be stored in the firmware memory 22, or by a jumper-configurable or hardwired circuit (not shown).

[0044] A pipeline unit similar to the unit 50 is discussed in previously incorporated U.S. Patent Publication No. 2004/0136241.
FIG. 3 is a diagram of the hardware layers that compose the hardware interface layer 62 within the PLIC 60 of FIG. 2 according to an embodiment of the invention. The hardware interface layer 62 includes three layers of circuitry that is instantiated on the PLIC 60: an interface-adapter layer 70, a framework-services layer 72, and a communication layer 74, which is hereinafter called a communication shell. The interface-adapter layer 70 includes circuitry, e.g., buffers and latches, that interfaces the framework-services layer 72 to the external pins (not shown) of the PLIC 60. The framework-services layer 72 provides a set of services to the hardwired pipeline(s) 44 via the communication shell 74. For example, the layer 72 may synchronize data transfer between the pipeline(s) 44, the pipeline bus 20 (FIG. 1), and the data memory 58 (FIG. 2), and may control the sequence(s) in which the pipeline(s) operate. The communication shell 74 includes circuitry, e.g., latches, that interface the framework-services layer 72 to the pipeline(s) 44.

Still referring to FIG. 3, alternate embodiments of the hardware-interface layer 62 are contemplated. For example, although the framework-services layer 72 is shown as isolating the interface-adapter layer 70 from the communication shell 74, the interface-adapter layer may, at least at some circuit nodes, be directly coupled to the communication shell. Furthermore, although the communication shell 74 is shown as isolating the interface-adapter layer 70 and the framework-services layer 72 from the hardwired pipeline(s) 44, the interface-adapter layer or the framework-services layer may, at least at some circuit nodes, be directly coupled to the pipeline(s).

FIG. 4 is a schematic block diagram of the circuitry that composes the interface-adapter layer 70 and the framework-services layer 72 of FIG. 3 according to an embodiment of the invention.

A communication interface 80 and an optional industry-standard bus interface 82 compose the interface-adapter layer 70, and a controller 84, exception manager 86, and configuration manager 88 compose the framework-services layer 72.

The communication interface 80 transfers data between a peer, such as the host processor 12 (FIG. 1) or another pipeline unit 50 (FIG. 2), and the firmware memory 22, the platform-identifier memory 54, the data memory 58, and the following components instantiated within the PLIC 60: the hardwired pipelines 44 (via the communication shell 74), the controller 86, the exception manager 88, and the configuration manager 90. If present, the optional industry-standard bus interface 82 couples the communication interface 80 to the bus connector 56. Alternatively, the interfaces 80 and 82 may be combined such that the functionality of the interface 82 is included within the communication interface 80.

The controller 84 synchronizes the hardwired pipelines 44, 44, and monitors and controls the sequence in which they perform the respective data operations in response to communications, i.e., “events,” from other peers. For example, a peer such as the host processor 12 may send an event to the pipeline unit 50 via the pipeline bus 20 to indicate that the peer has finished sending a block of data to the pipeline unit and to cause the hardwired pipelines 44, 44, to begin processing this data. An event that includes data is typically called a message, and an event that does not include data is typically called a “door bell.”

The exception manager 86 monitors the status of the hardwired pipelines 44, 44, the communication interface 80, the communication shell 74, the controller 84, and the bus interface 82 (if present), and reports exceptions to the host processor 12 (FIG. 1). For example, if a buffer (not shown) in the communication interface 80 overflows, then the exception manager 86 reports this to the host processor 12. The exception manager may also correct, or attempt to correct, the problem giving rise to the exception. For example, for an overflowing buffer, the exception manager 86 may increase the size of the buffer, either directly or via the configuration manager 88 as discussed below.

The configuration manager 88 sets the “soft” configuration of the hardwired pipelines 44, 44, the communication interface 80, the communication shell 74, the controller 84, the exception manager 86, and the interface 82 (if present) in response to soft-configuration data from the host processor 12 (FIG. 1). As discussed in previously incorporated U.S. Patent Publication No. 2004/0133763, the “hard” configuration of a component within the PLIC 60 denotes the actual instantiation, on the transistor and circuit-block level, of the component, and the soft configuration denotes the physical parameters (e.g., data width, table size) of the instantiated component. That is, soft-configuration data is similar to the data that one can load into a register of a processor (not shown in FIG. 4) to set the operating mode (e.g., burst-memory mode) of the processor. For example, the host processor 12 may send to the PLIC 60 soft-configuration data that causes the configuration manager 88 to set the number and respective priority levels of queues (not shown) within the communication interface 80. The exception manager 86 may also send soft-configuration data that causes the configuration manager 88 to, e.g., increase the size of an overflowing buffer in the communication interface 80.

The communication interface 80, optional industry-standard bus interface 82, controller 84, exception manager 86, and configuration manager 88 are further discussed in previously incorporated U.S. Patent Publication No. 2004/0136241.

Referring again to FIG. 2, although the pipeline unit 50 is disclosed as including only one PLIC 60, the pipeline unit may include multiple PLICS. For example, as discussed in previously incorporated U.S. Patent Publication No. 2004/0136241, the pipeline unit 50 may include two interconnected PLICS, where the circuitry that composes the interface-adapter layer 70 and framework-services layer 72 is instantiated on one of the PLICS, and the circuitry that composes the communication shell 74 and the hardwired pipelines 44 is instantiated on the other PLIC.

FIG. 5 is a diagram of a hardware-description file 100 from which a conventional PLIC, synthesizer and router tool (not shown) can generate the configuration firmware for the PLIC 60 of FIGS. 2-4 according to an embodiment of the invention. Typically, the hardware-description file 100 includes templates that are written in a conventional hardware description language (HDL) such as Verilog® HDL. The top-down structure of the file 100 resembles the top-down structure of software source code that incorporates software objects. Such a top-down structure for software source code provides at least two advantages. First, it allows a programmer to avoid writing and debugging source code
for a function when a software object that performs the function has already been written and debugged. Second, it allows the programmer to change or add a function by modifying an existing object or writing a new object with little or no rewriting and debugging of the source code that incorporates the object. As discussed below, the top-down structure of the file includes similar advantages. For example, it allows one to incorporate in the file existing templates that define an already-debugged hardware-interface layer, the framework-services layer, and the communication shell (collectively the hardware-interface layer) of the PLIC 60 (FIGS. 2-4). The template also defines the connections between the external pins (not shown) of the PLIC 60 and the interface-adapter layer 70 and also defines the connections between the framework-services layer (and in some cases the interface-adapter layer) and the communication shell 74.

The top-level definition 102 of the interface-adapter layer 70 (FIGS. 3-4) incorporates an interface-adapter-layer template 108, which further defines the portions of the interface-adapter layer defined by the top-level definition 102. For example, suppose that the top-level definition 102 defines a data-input buffer (not shown) in terms of its input and output nodes. Then, the top-level definition 102 defines the data-input buffer as a functional block having defined input and output nodes. The template 108 defines the circuitry that composes this functional block, and defines the connections between this circuitry and the buffer input nodes and output nodes received in the input buffer layer 70 in the top-level definition 102. Furthermore, the template 108 may incorporate one or more lower-level templates 109 that further define the data buffer or other components of the interface-adapter layer 70 received in the template 108. Moreover, these one or more lower-level templates 109 may each incorporate one or more even lower-level layers (not shown), and so on, until all portions of the interface-adapter layer 70 are defined in terms of circuit components (e.g., flip-flops, logic gates) that the PLIC synthesizing and routing tool recognizes.

Similarly, the top-level definition 104 of the framework-services layer 72 (FIGS. 3-4) incorporates a framework-services-layer template 110, which further defines the portions of the framework-services layer defined by the framework-services layer. For example, suppose the top-level definition 104 defines a counter (not shown) in terms of its input and output nodes. The template 110 defines the circuitry that composes this counter, and defines the connections between this circuitry and the counter input and output nodes received by the top-level definition 104. Furthermore, the template 110 may incorporate a hierarchy of one or more lower-level templates 111 and even lower-level templates (not shown), and so on, such that all portions of the framework-services layer 72 are at some level of the hierarchy, defined in terms of circuit components (e.g., flip-flops, logic gates) that the PLIC synthesizing and routing tool recognizes. For example, suppose the template 110 defines the counter as including a count-up/down-selector circuit having input and output nodes. The template 110 may incorporate a lower-level template 111 that defines the circuitry within the selector circuit and defines the connections between this circuitry and the selector circuit’s input and output nodes defined by the template 110.

Likewise, the top-level definition 106 of the communication shell 74 (FIGS. 3-4) incorporates a communication-shell template 112, which further defines the portions of the communication shell defined by the definition 106 and which also includes a top-level definition 113 of the hardwired pipeline(s) 44 disposed within the communication shell. For example, the definition 113 defines the connections between the communication shell 74 and the hardwired pipeline(s) 44.

The top-level definition 113 of the hardwired pipeline(s) 44 (FIGS. 3-4) incorporates one or more hardwired-pipeline templates 114, which further define the portions of the hardwired pipeline(s) 44 defined by the definition 113. The template or templates 114 may each incorporate a hierarchy of one or more lower-level templates 115 and even lower-level templates (not shown) such that all portions of the respective pipeline(s) 44 are, at some level of the hierarchy, defined in terms of circuit components (e.g., flip-flops, logic gates) that the PLIC synthesizing and routing tool recognizes.

Moreover, the communication-shell template 112 may incorporate a hierarchy of one or more lower-level templates 116 and even lower-level templates (not shown) such that all portions of the communication shell 74 other than the hardwired pipeline(s) 44 are, at some level of the hierarchy, defined in terms of circuit components (e.g., flip-flops, logic gates) that the PLIC synthesizing and routing tool recognizes.

Still referring to FIG. 5, a configuration template 118 provides definitions for one or more parameters having values that one can set to configure the circuitry that the templates 101, 108, 110, 112, 114 and lower-level templates 109, 111, 115, and 116 define. For example, suppose that the bus interface 82 of the interface-adapter layer 70 (FIG. 4) is configurable to have either a thirty-two-bit or a sixty-four-bit interface with the bus connector 56. The configuration template 118 defines a template BUS-WIDTH, the value of which determines the width of the interface between the bus interface 82 and the connector 56. For example, BUS-WIDTH=0 configures the interface 82 to have a thirty-two-bit interface, and BUS-WIDTH=1 configures the interface 82 to have a sixty-four-bit interface. Examples of other parameters that may be configurable include the depth of a first-in-first-out (FIFO) data buffer (not shown) disposed within the framework-services layer 72 (FIGS. 2-4), the lengths of messages received and transmitted by the interface-adapter layer 70, and the precision and data structure (e.g., integer, floating-point) of the hardwired pipeline(s) 44.

One or more of the templates 101, 108, 110, 112, 114 and the lower-level templates (not shown) incorporate the parameters defined in the configuration template 118. The PLIC synthesizer and router tool (not shown) configures the interface-adapter layer 70, the framework-services layer 72, the communication shell 74, and the hardwired pipeline(s) 44 (FIGS. 3-4) according to the values in the
template 118 during the synthesis of this circuitry. Consequently, to reconfigure the circuit parameters represented by the parameters in the configuration template 118, one need only modify the values of these parameters in the template 118, and then rerun the synthesizer and router tool on the file 100. Alternatively, if one or more of the parameters in the configuration template 118 can be sent to the PLIC as soft-configuration data after instantiation of the circuit, then one can modify the corresponding circuit parameters by merely modifying the soft-configuration data. Therefore, according to this alternative, may avoid rerunning the synthesizer and router tool on the file 100. Moreover, templates (e.g., 101, 108, 109, 110, 111, 112, 114, 115, and 116) that do not incorporate settable parameters such as those provided by the configuration template 118 are sometimes called modules or entities, and are typically lower-level templates that include Boolean expressions that a synthesizer and router tool (not shown) converts into circuitry for implementing the expressions.

[0064] Alternate embodiments of the hardware-description file 100 are contemplated. For example, although described as defining circuitry for instantiation on a PLIC, the file 100 may define circuitry for instantiation on an ASIC.

[0065] FIG. 6 is a block diagram of a library 120 that stores PLIC circuit templates, such as the templates 101, 108, 110, 112, and 114 (and any existing lower-level templates) of FIG. 5, according to an embodiment of the invention.

[0066] The library 120 has m-1 sections: m sections 122-122_m, for the respective m platforms that the library supports, and a section 124 for the hardwired-pipelines 44 (FIGS. 2-4) that the library supports.

[0067] For example purposes, the library section 122, is discussed in detail, it being understood that the other library sections 122-122_m are similar.

[0068] The library section 122, includes a top-level template 101, which is similar in structure to the template 101 of FIG. 5, and which thus includes top-level definitions 102, 104, and 106, of versions of the interface-adapter layer 70, the framework-services layer 72, and the communication shell 74 that are compatible with the platform m=1.

[0069] In this embodiment, we assume that there is only one version of the interface-adapter layer 70 and one version of the framework-services layer 72 available for each platform m, and, therefore, that the library section 122, includes only one interface-adapter-layer template 108, and only one framework-services-layer template 110. But in an embodiment that includes multiple versions of the interface-adapter layer 70 and multiple versions of the framework-services layer 72 for each platform m, the library section 122, would include multiple interface-adapter- and framework-services-layer templates 108 and 110.

[0070] The library section 122, also includes n communication-shell templates 112_1-112_n, which respectively correspond to the hardwired-pipeline templates 114_1-114_n, in the library section 124. As stated above in conjunction with FIG. 3, the communication shell 74 interfaces a hardwired pipeline or hardwired-pipelines 44 to the framework-services layer 72. Because each hardwired pipeline 44 is different and typically has different interface specifications, the communication shell 74 is typically adapted for each hardwired pipeline. Consequently, in this embodiment, one provides design adjustments to create a unique version of the communication shell 74 for each hardwired pipeline 44. The designer provides these design adjustments by writing a unique communication-shell template 112 for each hardwired pipeline. Of course the group of communication-shell templates 112_1-112_n corresponds only to the version of the framework-services layer 72 that is defined by the template 110; consequently, if there are multiple versions of the framework-services layer 72 that are compatible with the platform m=1, then the library section 122, includes a respective group of n communication-shell templates 112 for each version of the framework-services layer.

[0071] In addition, the library section 122, includes a configuration template 118, which defines configuration constants having designer-selectable values as discussed above in conjunction with the configuration template 118 of FIG. 5.

[0072] Furthermore, each template within the library section 122, includes, or is associated with, a respective description 126-134. The descriptions 126-132 describe the operational and other parameters of the circuitry that the respective templates 101, 108, 110, and 112-112_n define. Similarly, the description 134, describes the settable parameters in the configuration template 118, the values that these parameters can have, and the meanings of these values. The design tool discussed below in conjunction with FIGS. 7-11 uses the descriptions 126-134, to design and simulate a circuit that includes a combination of the hardwired pipelines 44, 44, which are respectively defined by the templates 114_1-114_n. Examples of parameters that the descriptions 126-132, may describe include the width of the data bus and the depths of buffers that the circuit defined by the corresponding template includes, the latency of the circuit, and the precision of the values received and generated by the circuit. Furthermore, an example of a settable parameter and the associated selectable values that the description 134, may describe is BUS-WIDTH, which represents the width of the interface between the communication interface 80 and the bus connector 56 (FIG. 4), and BUS_WIDTH=0 sets the bus width to thirty-two bits and BUS_WIDTH=1 sets the width to sixty-four bits.

[0073] Each of the descriptions 126-134, may be embedded within the respective template 101, 108, 110, 112-112_n, and 118, to which it corresponds. For example, the description 128, may be embedded within the template 108, as extensible markup language (XML) tags or comments that are readable by both a human and the tool discussed below in conjunction with FIGS. 7-11.

[0074] Alternatively, each description 126-134, may be disposed in a separate file that is linked to the template to which the description corresponds, and this file may be written in a language other than XML. For example, the description 126, may be disposed in a file that is linked to the top-level template 101.

[0075] The section 122, of the library 120 also includes a description 136, which describes the parameters of the platform m=1. The design tool discussed below in conjunction with FIGS. 7-11 may use the description 136, to determine which platforms the library 120 supports. Examples of parameters that the description 136, may
describe include 1 for each interface, the message specification, which lists the transmitted variables and the constraints for those variables, and 2) a behavior specification and any behavior constraints. Messages that the host processor 12 (FIG. 1) sends to the pipeline units 50 (FIG. 2) and that the pipeline units send among themselves are further discussed in previously incorporated U.S. Patent Publication No. 2004/0181621. Examples of other parameters that the description 136 may describe include the size and resources (e.g., the number of multipliers and the amount of available memory) of the PLIC 60 (FIGS. 2-4).

Furthermore, the platform description 136, may be written in XML or in another language.

[0076] Still referring to FIG. 6, the section 124 of the library 120 includes n hardwired-pipeline templates 114,-114n, which each define a respective hardwired pipeline 44,-44n (FIGS. 2-4). As discussed above in conjunction with FIG. 5, because the templates 114,-114n are platform independent (the corresponding communication-shell templates 112,-112n define the specified interface to the interface-adapter and framework-services layers 70 and 72 of FIGS. 3-4), the library 120 stores only one template 114 for each hardwired pipeline 44 (FIGS. 2-4). That is, each hardwired pipeline 44 does not require a separate template 114 for each platform that the library 120 supports. As discussed above, an advantage of this top-down design is that one need only create a single template 114 to define a hardwired pipeline 44, not n templates.

[0077] Furthermore, each hardwired-pipeline template 114 includes, or is associated with, a respective description 138,-138n, which describes the parameters of the hardwired-pipeline 44 that the template defines. Like the descriptions 126,-134, discussed above, the design tool discussed below in conjunction with FIGS. 7-11 uses the descriptions 138 to design and simulate a circuit that includes a combination of the hardwired pipelines 44,-44n, which are respectively defined by the templates 114,-114n. Examples of parameters that the descriptions 138,-138n may describe include the type (e.g., floating point or integer) and precision of the data that the corresponding hardwired pipeline 44 can receive and generate, and the latency of the pipeline. Also like the descriptions 126,-134, each of the descriptions 138,-138n may be embedded within the respective template 114,-114n, to which the description corresponds as, e.g., XML tags, or may be disposed in a separate file that is linked to the template to which the description corresponds.

[0078] Referring again to the library section 122,-122n, this section also includes a description 140 of the one or more available pipeline accelerators 14 (FIG. 1) that support the platform m=1. More specifically, the description 140 describes the resources that each of the pipeline accelerators 14 includes. For example, the description 140 may indicate that one available accelerator 14 includes only one pipeline unit 50 (FIG. 2), while another available accelerator includes five pipeline units. The description 140 may be written in XML or in another language.

[0079] Still referring to FIG. 6, alternate embodiments of the library 120 are contemplated. For example, instead of each template within each library section 122,-122n, being associated with a respective description 126-134, each library section 122,-122n may include a single description that describes all of the templates within that library section. For example, this single description may be embedded within or linked to the library section 120 or the configuration template 118. Furthermore, although each library section 122,-122n is described as including a respective communication-shell template 112 for each hardwired-pipeline template 114 in the library section 124, each section 122 may include fewer communication-shell templates, at least some of which are compatible with, and thus correspond to, more than one pipeline template 114. In an extreme, each library section 122,-122n, may include only a single communication-shell template 112, which is compatible with all of the hardwired-pipeline templates 114 in the library section 124. In addition, the library section 124 may include respective versions of each pipeline template 114 for each communication-shell template 112 in the library sections 122,-122n.

[0080] FIG. 7 is a block diagram of a circuit-design system 150, which includes a computer-based software tool 152 for designing a circuit using templates from the library 120 of FIG. 6 according to an embodiment of the invention. By using library templates, the tool 152 allows one to design a circuit that includes a combination of one or more previously designed and debugged hardware-interface layers 62 (FIG. 2) and hardwired pipelines 44 (FIGS. 2-4). Because another has already tested and debugged the one or more layers 62 and pipelines 44, the tool 152 may significantly decrease the time required for one to design such a combination circuit as compared to a conventional design progression. Furthermore, where one wants to design a circuit for executing an algorithm, the tool 152 allows him to define the circuit with an expression of conventional mathematical symbols, where the expression defines the algorithm; consequently, one having little or no experience in circuit design can use the tool to design a circuit for executing an algorithm.

[0081] The system 150 includes a processor (not shown) for executing the software code that composes the tool 152. Consequently, in response to the code, the processor performs the functions that are attributed to the tool 152 in the discussion below. For clarity of explanation, the tool 152, not the processor, is described as performing the actions.

[0082] In addition to the processor, the system 150 includes an input device 154, a display device 155, and the library 120 of FIG. 6. The input device 154, which may include a keyboard and a mouse, allows one to provide to the tool 152 information that describes an algorithm and that describes a circuit for executing the algorithm. Such information may include an expression of mathematical symbols, circuit parameters (e.g., buffer width, latency), operation exceptions (e.g., a divide by zero), and the platform on which one wishes to instantiate the circuit. And as described below, the device 155 displays the input information and other information, and the library 120 includes the templates that the tool 152 uses to build the circuit and to generate a file that defines the circuit.

[0083] The tool 152 includes a symbolic-math front end 156, an interpreter 158, a generator 160 for generating a file 162 that defines a circuit, and a simulator 164.

[0084] The front end 156 receives from the input device 154 the mathematical expression that defines the algorithm that the circuit is to execute and other design information,
and converts this information into a form that is readable by the interpreter 158. To allow one to define a circuit in terms of the mathematical expression that defines the algorithm that the circuit is to execute, in one embodiment the front end 156 includes a web browser that accepts XML with a schema for Math Markup Language (MathML). MathML is software standard that allows one to enter expressions using conventional mathematical symbols. The schema of MathML is a conventional plug in that imports to a web browser this same ability, i.e., the ability to enter expressions using mathematical symbols. Alternatively, the front end 156 may utilize another technique for allowing one to define a circuit using a mathematical expression. Examples of such another technique include the technique used by the conventional software mathematical-expression solver MathCAD. Furthermore, as discussed below, one may enter the identity of a platform or pipeline accelerator 14 (FIG. 1) on which he wants the circuit instantiated, and may enter test data with which the simulator 164 will simulate the operation of the circuit. Moreover, one may enter valid-range constraints for any variables within the entered mathematical expression and constraints on execution of the expression, and may specify the action(s) to be taken if the constraints are violated. For example, because $-1 \leq \sin(x) \leq 1$ for all values of $x$, for an expression that includes $\sin(x)$, one may enter this constraint, and specify that any data generated from a value of $\sin(x)$ outside of this range is to be disregarded. Or, because division by zero of any $x$ yields infinity, one may specify that data generated in response to a division by zero is to be disregarded. The front end 156 then converts all of the entered information into a format, such as HDL, that is compatible with the interpreter 158. Moreover, as discussed above, the front end 156 may cause the device 155 to display the input information and other related information. For example, the front end 156 may cause the device 155 to display the mathematical expression that the designer enters to define the algorithm to be executed by the circuit.

[0085] The interpreter 158 parses the information from the front end 156 and determines: 1) whether the library 120 includes templates 114 (FIG. 6) defining hardwired pipelines 44 (FIGS. 2-4) that, when combined, can execute the algorithm entered by the designer, and 2), if the answer to (1) is "yes," which, if any, available pipeline accelerators 14 (FIG. 1) described by the description 140 in the library 120 has sufficient resources to instantiate a circuit that can execute the algorithm. For example, suppose the algorithm includes the mathematical operation $\forall P$. If the library 120 does not include a template 114 (FIG. 6) defining a hardwired pipeline 44 (FIGS. 2-4) that calculates the square root of a value, then the interpreter 158 determines that the tool 152 cannot generate a file 162 that defines a circuit for executing the algorithm. Furthermore, suppose that the circuit for executing the algorithm requires the resources of at least five PLICs 60 (FIGS. 2-4). If the description 140 indicates that the available accelerators 14 each have only three pipeline units 50 (FIG. 2), and thus each have only three PLICs, 60, then the interpreter 158 determines that even though the tool 152 may be able to generate a file 162 that defines a circuit for executing the algorithm, one cannot implement this circuit on an available accelerator. The interpreter 158 makes a similar determination if the designer indicates that he wants the algorithm executed by a circuit having a sixty-four-bit bus width, but the available platforms support only a thirty-two-bit bus width. In situations where the interpreter 158 determines that the tool 152 cannot generate a circuit for executing the desired algorithm or that one cannot implement the circuit on an existing platform and/or accelerator 14, the interpreter 158 causes the device 155 to display an appropriate error message (e.g., “no library template for instantiating ‘V’; insufficient PLIC resources, ‘bus-width not supported’). Furthermore, where the designer identifies a platform or accelerator 14 on which he desires to instantiate the resulting circuit, the interpreter 158 determines whether the circuit can be instantiated on the identified platform or accelerator. But if the circuit cannot be so instantiated, the interpreter 158 may determine that the circuit can be instantiated on another platform or accelerator, and thus may so inform the designer with an appropriate message via the display device 155. This allows the designer the choice of instantiating the circuit on another platform or accelerator 14.

[0086] If the interpreter 158 determines that the library 120 includes a sufficient number of hardwired-pipeline templates 114 (FIG. 6) to define a circuit that can execute the desired algorithm, and also determines that the circuit can be instantiated on an available platform and accelerator 14 (FIG. 1), then the interpreter provides to the file generator 160 the identities of the hardwired-pipeline templates 114 that correspond to portions of the algorithm.

[0087] The file generator 160 combines the hardwired pipelines 44 (FIGS. 2-4) defined by the identified hardwired-pipeline templates 114 such that the combination forms a circuit that can execute the algorithm.

[0088] The generator 160 then generates the file 162, which defines the circuit for executing the algorithm in terms of the hardwired pipelines 44 (FIGS. 2-4) and the hardware-interface layers 62 (FIG. 2) that compose the circuit, the PLIC(s) 60 (FIGS. 2-3) on which the pipelines are disposed, and the interconnections between the pipelines (if multiple pipelines on a PLIC) and/or between the PLICs (if the pipelines are disposed on more than one PLIC).

[0089] Next, the host processor 12 (FIG. 1) can use the file 162 to instantiate on the pipeline accelerator 14 (FIG. 1) the defined circuit as discussed in previously incorporated U.S. patent application Ser. No. (Attorney Docket No. 1934-25-3). Alternatively, also as discussed in U.S. patent application Ser. No. (Attorney Docket No. 1934-25-3), the host processor 12 may instantiate some or all portions of the defined circuit in software executed by the processing unit 32. Or, one can instantiate the circuit defined by the file 162 in another manner.

[0090] The simulator 164 receives the file 162 from the generator 160 and receives from the front end 154 designer-entered test data, such as a test vector, designer-entered constraint data, and a designer-entered exception-handling protocol, and then simulates operation of the circuit defined by the file 162. The simulator 164 also gathers parameter information (e.g., precision, latency) from the description files 138 (FIG. 6) that correspond to the hardwired-pipeline templates 114 that define the pipelines 44 that compose the circuit. The simulator 164 may retrieve this parameter information directly from the library 120, or the generator 160 may include this parameter information in the file 162.

[0091] FIG. 8 illustrates the parsing of a symbolic mathematical expression by the interpreter 158 according to an
embodiment of the invention. In other words, the syntax of the design language is the same as that used by mathematicians for writing algebraic equations. The explanations that follow show how a symbolic mathematical expression is a sufficient syntax for defining the hardwired pipelines 44 from a simple set of circuit primitives.

[0092] FIG. 9 illustrates a table of hardwired-pipeline templates 114, which correspond to the hardwired pipelines 44 (FIGS. 2-4) that the interpreter 158 (FIG. 7) identifies for executing portions of the parsed algorithm (FIG. 8) according to an embodiment of the invention.

[0093] Referring to FIGS. 5-9, the operation of the tool 152 is discussed according to an embodiment of the invention.

[0094] Suppose that one wishes to design a circuit that solves for a value y, which equals a mathematical expression according to the following equation:

\[ y = x^2 \cos(z) + z^3 \sin(x) \]  

(2)

Also suppose that x, y, and z are thirty-two-bit floating-point values.

[0095] Using the input device 154, the designer enters equation (2) into the front end 156 of the tool 152 by entering the following sequence of mathematical symbols: “\( y \), \( x^2 \), \( \cos(z) \), \( \cdot \), \( z^3 \), \( \cdot \), \( \sin(x) \)”. The designer also enters information specifying the input and output message specifications, for example indicating that x, y, and z are thirty-two-bit floating-point values. The designer may also enter information indicating desired operating parameters, such as the desired latency, in clock cycles, from inputs x and z to output y, and the desired types and precision of any intermediate values, such as cos(z) and sin(x), generated during the calculation of y. Furthermore, the designer may enter information that identifies a desired platform or pipeline accelerator 14 (FIG. 1) on which he wants the circuit instantiated. Moreover, the designer may specify the accuracy of any mathematical approximations that the tool 152 may make. For example, if the tool 152 approximates cos(z) using a Taylor series expansion, then by specifying the accuracy of this approximation, the designer effectively specifies the number of terms needed in the expansion. Alternatively, the designer may directly specify the number of terms in the expansion. The implementation of a function as a Taylor series expansion is further described below in conjunction with FIGS. 13-17.

[0096] The front end 156 converts these mathematical symbols and the other information into a format compatible with the interpreter 158 if this information is not already in a compatible format.

[0097] Next, the interpreter 158 determines whether any of the hardwired-pipeline templates 114 in the library 120 defines a hardwired-pipeline 44 that can solve for y in equation (2) within the specified behavior and operating parameters and that can be instantiated within the desired platform and on the desired pipeline accelerator 14 (FIG. 1).

[0098] If the library 120 does include such a template 114, then the interpreter 158 informs the designer, via the display device 155, that a conventional FPGA synthesizing and routing tool can generate firmware for instantiating this hardwired pipeline 44 from the identified template 114, the corresponding communication-shell template 112, and the corresponding top-level template 101.

[0099] If, however, the library 120 includes no template 114 that defines a hardwired pipeline 44 that can solve for y in equation (2), then the interpreter 158 parses the equation (2) into portions, and determines whether the library includes templates 114 that define hardwired pipelines 44 for executing these portions within the specified behavior, operating parameters, and platform and on the specified pipeline accelerator 14 (FIG. 1).

[0100] To identify a circuit that can solve for y in equation (2) but that includes the fewest number of hardwired pipelines 44, the interpreter 158 parses the equation (2) according to a top-down parsing sequence as discussed below. Typically, this top-down parsing sequence corresponds to the known algebraic laws for the order of operations.

[0101] First, the interpreter 158 parses the equation (2) into the following two portions: “y”, which is portion 170 in FIG. 8, and “\( x^2 \cos(z) + z^3 \sin(x) \)”, which is portion 172.

[0102] If the interpreter 158 determines that the library 120 includes at least two hardwired-pipeline templates 114 that define hardwired pipelines 44 for respectively executing the portions 170 and 172 of equation (2), then the interpreter passes the identity of these templates to the file generator 160.

[0103] In this example, however, the interpreter 158 determines that although the library 120 includes a hardwired-pipeline template 114 that defines a pipeline 44 for executing the square-root operation 170 of equation (2), the library includes no hardwired-pipeline template that defines a pipeline for executing the portion 172.

[0104] Next, the interpreter 158 parses the portion 172 of equation (2). Specifically, the interpreter 158 parses the portion 172 into the following three respective portions 174, 176, and 178: “\( x^2 \cos(z) \)”, “\( + \)”, and “\( z^3 \sin(x) \)”.

[0105] If the interpreter 158 determines that the library 120 includes at least three hardwired-pipeline templates 114 that define hardwired pipelines 44 for respectively executing the portions 174, 176, and 178 of equation (2), then the interpreter passes the identity of these templates to the file generator 160.

[0106] In this example, however, the interpreter 158 determines that although the library 120 includes a hardwired-pipeline template 114 that defines a hardwired pipeline 44 for executing the summing operation 176 of equation (2), the library includes no templates 114 that define hardwired pipelines for executing the portions 174 or 178.

[0107] Next, the interpreter 158 parses the portions 174 and 178 of equation (2). Specifically, the interpreter 158 parses the portion 174 into three portions 180 (“\( x^2 \)"), 182 (“\( \cdot \)”), and 184 (“\( \cos(z) \)”), and parses the portion 178 into three portions 186 (“\( z^3 \)”), 188 (“\( \cdot \)”), and 190 (“\( \sin(x) \)”).

[0108] If the interpreter 158 determines that the library 120 does not include hardwired-pipeline templates 114 that define hardwired pipelines 44 for respectively executing each of the portions 180, 182, 184, 186, 188, and 190, then the interpreter displays via the device 155 an error message indicating that the library does not support a circuit that can solve for y in equation (2). In one embodiment of the
invention, however, the library 120 includes hardwired-pipeline templates 114 that provide the primitive operations for multiplication and for raising variables to a power (e.g., cubing a value by using two multipliers in sequence) for single- or double-precision floating-point data types, and for data-type conversion. Also in this embodiment, the tool 152 recognizes common factors, for example that x is a factor of x^2 if \( \sin(x^2) \) was needed instead of \( \sin(x) \), and generates circuitry to provide these common factors from chained multipliers.

[0109] In this example, however, the interpreter 158 determines that the library 120 includes hardwired-pipeline templates 114 that define hardwired pipelines 44 for respectively executing each portion 180, 182, 184, 186, 188, and 190 of equation (2).

[0110] Then, the interpreter 158 provides to the file generator 160 the identities of all the hardwired-pipeline templates 114 that define the hardwired-pipelines 44 for executing the following eight portions of equation (1): 170 ("y"), 176 ("x"), 180 ("x"), 182 ("z"), 184 ("cos(z)"), 186 ("z"), 188 ("y"), and 190 ("sin(x)").

[0111] Referring to FIGS. 6-10, the file generator 160 generates a table 192 (FIG. 9) of the hardwired-pipeline templates 114 identified by the interpreter 158, and displays this table via the device 155. In a first column 194, the table 192 lists the portions 170 ("y"), 176 ("x"), 180 ("x"), 182 ("z"), 184 ("cos(z)"), 186 ("z"), 188 ("y"), and 190 ("sin(x)"). In a second column 196, the table 192 lists the hardwired-pipeline or templates 114 that define a hardwired pipeline 44 for executing the respective portion of equation (2). And in a third column 198, the table 192 lists parameters, such as the latency (in units of cycles of the signal that clocks the defined pipeline 44) and the input and output precision, of the hardwired pipeline(s) 44 defined by the templates 114 in the second column 196. As shown in the table 192, in this example the seven hardwired-pipeline templates 114, 116, 118, 119 define hardwired pipelines 44, 44, for respectively executing the corresponding portions of equation (2) in column 194. There are only seven pipeline templates 114, 116, 118, 119, for the eight portions of equation (2) because the template 114, defines a multiplier pipeline 44, that can execute both 37.5 portions 182 and 188. Furthermore, although we have labeled the pipeline templates as 114, 116, it is not required that these templates be sequentially ordered within the library 120. Moreover, the library 120, and thus the table 192, may include multiple templates 114 that define respective pipelines for executing each of the eight portions 170, 176, 180, 182, 184, 186, 188, and 190 of equation (2).

[0112] Next, using the table 192, the file generator 160 selects the pipelines 44 from which to build a circuit that solves for y in equation (2). The generator 160 selects these pipelines 44 based on the behavior(s), operating parameter(s), platform(s), and pipeline accelerator(s) 14 (FIG. 1) that the designer specified. For example, if the designer specified that x, y, and z are thirty-two-bit floating-point quantities, then the generator 160 selects pipelines 44 that operate on thirty-two-bit floating-point numbers. If the available pipelines 44 for a particular portion of the equation (2) do not meet all of the designer's specifications, then the generator 160 may use a default set of rules to select the best pipeline. For example, the rules may indicate that if there is no available pipeline 44 that meets the specified latency and precision requirements, then, with the designer's authorization, the generator 160 defaults to the pipeline having the specified precision and the latency closest to the specified latency. Otherwise a new pipeline 44 with the specified latency is placed in the library, or the designer can select another pipeline from the table 192. As an example of satisfying the latency requirements, two versions of an x^4 circuit may be represented by respective hardwired-pipeline templates 114 in the library 120: a pipelined version using two fully registered multipliers in a cascade, or an in-place version using a single, fully registered multiplier, a one-bit counter, and a multiplexer. The pipelined version consumes roughly twice the circuit resources but accepts one input value every clock cycle. In contrast, the in-place version consumes fewer circuit resources but accepts a new input value only every other clock cycle.

[0113] Then, the file generator 160 interconnects the selected hardwired pipelines 44 to form a circuit 200 (FIG. 10) that can solve for y in equation (2). The generator 160 also generates a schematic diagram of the circuit 200 for display via the device 155.

[0114] To form the circuit 200, the file generator 160 first determines how the selected hardwired pipelines 44, 44, can "fit" into the resources of a specified accelerator 14 (FIG. 1) (or a default accelerator if the designer does not specify one). For example, the file generator 160 calculates the number of PLICs 60 (FIG. 3) needed to contain the eight instances of the pipelines 44, 44, (this includes two instances of the pipeline 445).

[0115] In this example, the generator 160 determines that each PLIC 60 (FIG. 3) can hold only a respective one of the pipelines 44, 44; consequently, the generator 160 determines that eight pipeline units 50, 50 are needed to instantiate the circuit 200.

[0116] Next, based on the platform that the designer specifies, the generator 160 "inserts" into each of the PLICs 60, 60, of the pipeline units 50, 50, a respective hardware-interface layer 62, 62. Assuming that the designer specifies platform m=1, the generator 160 generates the layers 62, 62, from the following templates in section 122, of the library 120: the interface-adapter-layer template 108, the framework-services-layer template 110, and the communication-shell templates 112, 112, which respectively correspond to the pipeline templates 114, 114, and thus to the pipelines 44, 44. More specifically, the generator 160 generates the hardware-interface layer 62, from the interface-adapter-layer template 108, the framework-services-layer template 110, and the communication-shell template 112. Similarly, the generator 160 generates the hardware-interface layer 62, from the templates 108, 110, and 112, so on. Furthermore, because the PLICs 60 and 60, both will include the multiplier pipeline 44, the generator 160 generates both of the hardware-interface layers 62, 62, from the interface-adapter and framework-services templates 108, 110, from the communication-shell template 112, 112; consequently, the hardware-interface layers 62, 62, are identical but are instantiated on respective PLICs 60, and 60. Moreover, the generator 160 generates the hardware-interface layer 62, from the templates 108, 110, and 112, and the hardware-interface layer 62, from the templates 108, 110, and 112.
Then, the generator 160 inserts into each hardware-interface layer $62_s, 62_x$ a respective hardwired pipeline $44_s, 44_x$, (the generator 160 inserts the pipeline $44_s$ into both of the hardware-interface layers $62_s$, and $62_x$; the pipeline $44_s$ into the hardware-interface layer $62_s$, and the pipeline $44_x$ into the hardware-interface layer $62_x$). More specifically, the generator 160 inserts the pipelines $44_s, 44_x$ into the hardware-interface layers $62_s, 62_x$, by respectively inserting the hardwired-pipeline templates $114_s, 114_x$, into the communication-shell templates $112_s, 112_x$.

Next, the generator 160 interconnects the pipeline units $50_s, 50_x$, to form the circuit 200, which generates the value $y$ from equation (2) at its output (i.e., the output of the pipeline unit $50_s$).

Referring to FIG. 10, the circuit 200 includes an input stage 206, first and second intermediate stages 208 and 210, and an output stage 212, and operates as follows. The input stage 206 includes the hardwired pipelines $44_s, 44_x$, and operates as follows. The pipeline $44_s$, receives a stream of values $x$ via an input port of the hardware-interface layer $62_s$, and generates, in a pipelined fashion, a corresponding stream of values $z$ via an output port of the layer $62_s$. Likewise, the pipeline $40_x$, receives a stream of values $z$ via an input port of the hardware-interface layer $62_x$, and generates, in a pipelined fashion, a corresponding stream of values $z^2$ via an output port of the layer $62_x$. The pipeline $44_s$, receives the stream of values $x$ via an input port of the hardware-interface layer $62_s$, and generates, in a pipelined fashion, a corresponding stream of values $x^2$ via an output port of the layer $62_s$, and the pipeline $44_x$, receives the stream of values $z$ via an input port of the hardware-interface layer $62_x$, and generates, in a pipelined fashion, a corresponding stream of values $cos(z)$ via an output port of the layer $62_x$.

The first intermediate stage 208 of the circuit 200 includes two instantiations of the pipelines $44_s, 44_x$, and operates as follows. The pipeline $44_s$, in the PLC 60_s, receives the streams of values $sin(x)$ and $z$ from the input stage 206 via an input port of the hardware-interface layer $62_s$, and generates, in a pipelined fashion, a corresponding stream of values $z^2$ via an output port of the layer $62_s$. Similarly, the pipeline $44_x$, in the PLC 60_x, receives the streams of values $x$ and $cos(z)$ from the input stage 206 via an input port of the hardware-interface layer $62_x$, and generates, in a pipelined fashion, a corresponding stream of values $x^2 cos(z)$ via an output port of the layer $62_x$.

The second intermediate stage 210 of the circuit 200 includes the hardwired pipeline $44_s, 44_x$, which receives the streams of values $z^2 sin(x)$ and $x^2 cos(z)$ from the first intermediate stage 208 via an input port of the hardware-interface layer $62_s$, and generates, in a pipelined fashion, a corresponding stream of values $z^2 sin(x)+x^2 cos(z)$ via an output port of the layer $62_s$.

And the output stage 212 of the circuit 200 includes the hardwired pipeline $44_s, 44_x$, which receives the stream of values $z^2 sin(x)+x^2 cos(z)$ from the second intermediate stage 210 via an input port of the hardware-interface layer $62_s$, and generates, in a pipelined fashion, a corresponding stream of values $y=\sqrt{z^2 sin(x)+x^2 cos(z)}$ via an output port of the layer $62_s$. Referring to FIGS. 7, 9, and 10, the designer may choose to alter the circuit 200 via the input device 154.

For example, the designer may swap out one or more of the pipelines $44_s, 44_x$, with one or more other pipelines from the table 192. Suppose the square-root pipeline $44_s$ has a high precision but a relatively long latency per the default rules that the generator 160 follows as discussed above. If the table 192 includes another square-root pipeline having a shorter latency, then the designer may replace the pipeline $44_x$, with the other square-root pipeline, for example by using the input device 154 to "drag" the other pipeline from the table into the schematic representation of the PLIC 60_x.

In addition, the designer may swap out one or more of the hardwired pipelines $44_s, 44_x$, with a symbolically defined polynomial series (i.e., a Taylor Series equivalent) that approximates one of the pipelined operations. Suppose the available square-root pipeline $44_s$ has insufficient mathematical accuracy per the designers' specification and the default rules that the generator 160 follows as discussed above. If the designer then specifies a new square-root function as a series summation of related monomials, then the front end 156, interpreter 158, and file generator 160 concatenate a series of parameterized monomial circuit templates into a circuit that solves for square roots. In this way the designer replaces the default pipeline $44_s$, with the higher-precision square-root circuit using symbolic design. This example illustrates the symbolic use of polynomials to define new mathematical functions as established by Taylor's Theorem. A more detailed example is discussed below in conjunction with FIGS. 13-17.

The designer may also change the topology of the circuit 200. Suppose that according to the default rules discussed above, the generator 160 places each instantiation of the hardwired pipelines $44_s, 44_x$, into a separate PLIC 60. But also suppose that each PLIC 60 has sufficient resources to hold multiple pipelines $44_s$. Consequently, to reduce the number of pipeline units 50 that the circuit 200 occupies, the designer may, using the input device 154, move some of the pipelines $44_s$ into the same PLIC. For example, the designer may move both instantiations of the multiplier pipeline $44_s$ out of the PLICs 60_s, and 60_x, and into the PLIC 60_x, with the adder pipeline $44_x$, thus reducing by two the number of PLICs that the circuit 200 occupies. The designer then manually interconnects the two instantiations of the pipeline $44_s$, to the pipeline $44_x$, within the PLIC 60_x, or may instruct the generator 160 to perform this interconnection. Although the library 120 may not include a communication-shell template 112 that defines a communication shell 74 for this combination of multiple pipelines $44_s$, and $44_x$, the designer or another may write such a template and debug the communication shell that the template defines without having to rewrite the interface-adapter-layer and framework-services templates 108, and 110, and, therefore, without having to re-debug the layers that these templates define. This rearranging of pipelines $44_s$ within the PLICs 60 is also called "refactoring" the circuit 200.

Moreover, the designer may decide to breakdown one or more of the pipelines $44_s, 44_x$, into multiple, less complex pipelines $44_s$. For example, to equalize the latencies in the stage 206 of the circuit 200, the designer may decide to breakdown the $x^2$ pipeline $44_s$, into two $x^2$ pipelines (not shown) and a multiplier pipeline $44_x$. Or, the designer may decide to replace the $sin(x)$ pipeline $44_s$, with a combination...
of pipelines (not shown) that represents \(\sin(x)\) in a series-expansion form (e.g. Taylor series, MacLaurin series).

[0128] Referring to FIGS. 7 and 10, after the designer has made any desired changes to the circuit 200, the generator 160 generates the file 162, which describes the circuit in terms of the pipeline units 50, the PLICs 60, the library templates that compose the circuit, and the interconnections between the pipeline units. Specifically, assuming that the designer has not modified the circuit 200 from the layout shown in FIG. 10, the file 162 indicates that the circuit is designed for instantiation on eight pipeline units 50, 50, of a platform accelerator 14 (FIG. 1) that is compatible with platform m=1. The file 162 also identifies the eight PLICs 60, 60, on the eight pipeline units 50, 50, and for each PLIC, identifies the templates in the library 120 that define the circuitry to be instantiated on the PLIC. For example, referring to FIGS. 6 and 10, the file 162 indicates that the combination of the following templates in the library 120 defines the circuitry to be instantiated on the PLIC 60: 101, 108, 110, 111, 114, and 116. Furthermore, the file 162 includes the values of all constants defined in the configuration template 118. The file 162 may also include one or more of the descriptions 128-134 and 138 corresponding to these templates, or portions of these descriptions. Moreover, the file 162 defines the interconnections between the PLICs 60, 60, and the message specifications for these interconnections. The file 162 also defines any designer-specified range constraints for generated values, exceptions, and exception-handle routines. The generator 160 may write the file 162 in XML or in another language with XML tags so that both humans and other tools/machines can read the file. Alternatively, the generator 160 may write the file 162 in a language other than XML and without XML tags.

[0129] Referring to FIGS. 6, 7, 9, and 10, the designer may instruct the simulator 164, via the input device 154, to simulate the circuit 200 using a conventional simulation algorithm. The simulator 164 uses the information in the file 162 and the test vectors provided by the designer to simulate the operation of the circuit 200. The simulator 164 first determines the operating parameters of the hardware-interface layers 62, 62, and of the hardwired pipelines 44, 44, from the file 162, or by extracting this information directly from the description files 128, 130, 131, 131, 132, 132, and 138, in the library 120. As described above, these parameters include, e.g., circuit latencies, and the precision (e.g., thirty-two-bit integer, sixty-four-bit floating point) of the values that the pipelines 44, 44, receive and generate. For example, from the description files 128, 130, 131, 132, and 138, the simulator 164 determines the latency of the PLIC 60, from the time a value x enters the hardware-interface layer 62, until the time that the layer 62, provides \(\sin(x)\) on an external pin (not shown) of the PLIC 60. The latency information in these description files may be estimated information, or may be actual information derived from an analysis of an instantiation of the pipeline 44, and the hardware-interface layer 62, on the PLIC 60. The simulator 164 then estimates the latencies and other operating parameters of the PLICs 60, 60, and simulates the operation of the circuit 200 to generate an output test stream of values y in response to input test streams of values x and z.

[0130] FIG. 11 is a schematic diagram of the circuit 200 of FIG. 10 disposed on a single pipeline unit 50, and in a single PLIC 60 according to an embodiment of the invention.

[0131] Referring to FIGS. 6, 7, 9, and 11, the operation of the tool 152 is discussed according to another embodiment of the invention.

[0132] Following the same steps described above in conjunction with the formation of the circuit 200 of FIG. 10, the generator 160 determines that all of the hardwired pipelines 44, 44, (the multiplier pipeline 44, is instantiated twice) can fit within a single PLIC 60 with the same topology shown in FIG. 10.

[0133] Although the library 120 includes no communication-shell templates 112 for this combination of the hardwired pipelines 44, 44, for simulation purposes the tool 152 derives the operational parameters and message specifications of the hardware-interface layer 62 from the description files 128, 130, 131, 131, 132, 132, and 132, 132. Because the PLIC 60 incorporates the interface-adapter layer 70 and framework-services layer 72 defined by the templates 108, and 110, the tool 152 estimates the input and output operational parameters, e.g., input and output latencies, and the message specifications of the layers 70 and 72 directly from the description files 128, and 130. Then, referring to FIGS. 10-11, because the values x and z are input in parallel to the pipelines 44, 44, the tool 152 derives the input operating parameters of the communication shell 74 of FIG. 11 from the description files 132, 132, which describe the communications shells for the pipelines 44, 44. For example, if the operational parameters of these communication shells are similar, then the tool 152 may merely estimate that the input-side operational parameters for the shell 74 are the same as the parameters from one of the description files 132, 132, Alternatively, the tool 152 may estimate that an intermediate data-type translation is needed for the input-side operational parameters of the communication shell 74, or that an averaging operation is needed for the input-side operational parameters of the communication shell, if the respective input-side parameters in the description files 132, 132, do not match. Similarly, because the values y are output from the pipeline 44, the tool 152 derives the output operating parameters for the communication shell 74 from the description file 132, which describes the communication shell for the pipeline 44. For example, the tool 152 may estimate that the output-side operational parameters for the shell 74 are the same as the output-side parameters from the description file 132.

[0134] Next, the generator 160 generates the file 162, which defines the circuit 200 of FIG. 11, and the simulator 164 simulates the circuit using the operational parameters calculated for the hardware-interface layer 62 by the generator 160.

[0135] FIG. 12 is a block diagram of a circuit 220, for which the tool 152 of FIG. 7 generates a file 162 according to an embodiment of the invention where the circuit solves for a variable in an equation that includes constant coefficients. The circuit 220 is similar to the circuit 200 except that the hardwired pipelines 44, and 44, respectively generate ax and bx2 instead of x2 and z2, where a and b are constant coefficients.
In this embodiment, the designer wants to design a circuit to solve for \( y \) in the following equation:

\[
y = \sqrt{ax^2 \cos(z) + bz^2 \sin(x)}
\]  

(3)

The only differences between equation (3) and equation (2) is the presence of the constant coefficients \( a \) and \( b \).

Referring to FIG. 10, one way for the tool 152 to generate such a circuit is to modify the circuit 200 to is to parse equation (3) into portions including "\( a x^2 \)" and "\( b z^2 \)", and to add two corresponding PLICs (not shown) on which are instantiated the multiplication pipeline 44, one such multiplier PLIC between the PLICs 60 and 60, and receiving as inputs \( x^2 \) and \( a \), and the other such multiplier PLIC between the PLICs 60 and 60, and receiving as inputs \( z^2 \) and \( b \).

Although such a modified circuit 200 is contemplated to accommodate the constant coefficients \( a \) and \( b \), this circuit would require two additional pipeline units 50.

Referring to FIGS. 7, 10, and 12, in this embodiment, however, the tool 152 generates the circuit 220 by replacing the pipelines 44, and 44, in the circuit 200 with pipelines 44, and 44, which respectively perform the operations \( b z^2 \) and \( a x^2 \). Of course this assumes that the section 124 of the library 120 (FIG. 6) includes corresponding hard-wired-pipeline templates 114, and 114.

Referring to FIGS. 7 and 12, to set the values of the coefficients \( a \) and \( b \), the designer may enter the values as part of equation (3), or may enter the values separately. Assume that the designer wants \( a = 2.0 \) and \( b = 3.5 \). According to the former entry method, he enters equation (3) as: "\( y = \sqrt{2x^4 \cos(z) + 3.5z^2 \sin(x)} \)". And according to the latter entry method, he enters equation (3) as \( y = \frac{ax^2 \cos(z) + bz^2 \sin(x)}{ax^2 + bz^2} \) and then enters "\( a = 2.0, b = 3.5 \)."

The generator 160 then generates the file 162 to include the entered values for the coefficients \( a \) and \( b \). These values may be contained within one or more XML tags or be present in some other form.

In another variation, the values of \( a \) and \( b \) may be provided to the configuration managers 88 (FIG. 3) of the PLICs 60 and 60, as soft-configuration data. More specifically, a configuration manager (not shown and different from the configuration managers 88), which is described in previously incorporated U.S. patent application Ser. No. (Attorney Docket No. 1934-25-3, 1934-26-3, and 1934-36-3) and which is executed by the host processor 12 (FIG. 1), initializes the values of \( a \) and \( b \) by sending configuration messages for \( a \) and \( b \) to the pipeline units 50, and 50. The accelerator-configuration registry 40 (FIG. 1) may store a and \( b \) as XML files to initialize the configuration messages created and sent by the configuration manager executed by the host processor 12.

Still referring to FIGS. 7 and 12, the tool 152 can use similar techniques to set the values of constant coefficients for other types of circuit portions such as filters, Fast Fourier Transformers (FFTs), and Inverse Fast Fourier Transformers (IFFTs).

Referring to FIGS. 7-12, other embodiments of the tool 152 and its operation are contemplated.

For example, one or more of the functions of the tool 152 may be performed by a functional block (e.g., front end 156, interpreter 158) other than the block to which the function is attributed in the above discussion.

Furthermore, the tool 152 may be described using more or fewer functional blocks. In addition, although the tool 152 is described as either fitting the eight instantiations of the hardwired pipelines 44, 44, in eight PLICs 60, 60, (FIGS. 10 and 12) or in a single PLIC 60 (FIG. 11), the tool 152 may fit these pipelines in more than one but fewer than eight PLICs, depending on the resources available on each PLIC.

Moreover, although described as allowing a designer to define a circuit using conventional mathematical symbols, alternate embodiments of the front end 156 of the tool 152 may lack this ability, or may allow one to define a circuit using other formats or languages such as C++ or VHDL.

Furthermore, although the tool 152 is described as allowing one to design a circuit for instantiation on a PLIC, the tool 152 may also allow one to design a circuit for instantiation on an ASIC.

In addition, although the tool 152 is described as generating a file 162 that defines an algorithm-implementing circuit, such as the circuit 200 (FIG. 11), for instantiation on a specific pipeline accelerator 14 (FIG. 14) or on a pipeline accelerator that is compatible with a specific platform, the tool may generate, in addition to or instead of the file 162, a file (not shown) that more generally defines the algorithm. Such a file may include algorithm-definition data that is sometimes called "meta-data," and may allow the host processor 12 (FIG. 1) to implement the algorithm in any manner (e.g., hardwired pipeline(s), software, a combination of both pipeline(s) and software) supported by the peer vector machine 10 (FIG. 1). Typically, meta-data describes something, such as an algorithm or another file, but is not executable. For example, the information in the description files 126-134 (FIG. 6) may include meta-data. But a processor, such as the host processor 12, may be able to generate executable code from meta-data. Consequently, a meta-data file that defines an algorithm may allow the host processor 12 to configure the peer vector machine 10 for implementing the algorithm even where the machine does not support the implementation(s) specified by the file 162. Such configuring of the peer vector machine 10 is described in U.S. patent application Ser. No. (Attorney Docket Nos. 1934-25-3, 1934-26-3, and 1934-36-3), which were previously incorporated by reference.

Moreover, the tool 152 may generate, and the library 120 (FIG. 6) may store, one or more meta-data files (not shown) for describing the messages that carry data to/from the PLICs 60 (or software equivalents) of a circuit, such as the circuit 200 (FIG. 10). For example, if the data generated by the PLICs 60 is floating-point data, then a meta-data file specifies this. The file 162 (FIG. 7) incorporates or points to these meta-data files so that the host processor 12 (FIG. 1) can instantiate the message objects that generate such messages as discussed in previously incorporated U.S. patent application Ser. Nos. (Attorney Docket Nos. 1934-25-3, 1934-26-3, and 1934-36-3).

Furthermore, the tool 152 may generate, and the library 120 (FIG. 6) may store, one or more meta-data files (not shown) for describing the exceptions that the PLICs 60
(or software equivalents) of a circuit, such as the circuit 200 (FIG. 10), generate. For example, if a PLIC 60 implements a divide-by-zero exception, then a meta-data file specifies this. The file 162 (FIG. 7) incorporates or points to these meta-data files so that the host processor 12 (FIG. 1) can instantiate corresponding exception handlers as discussed in previously incorporated U.S. patent application Ser. Nos. (Attorney Docket Nos. 1934-25-3, 1934-26-3, and 1934-36-3).

[0152] In addition, the tool 152 may generate, and the library 120 (FIG. 6) may store, one or more meta-data files (not shown) for describing the PLIC’s 60 (or software equivalents) of a circuit, such as the circuit 200 (FIG. 10). For example, such a meta-data file may describe the mathematical operation performed by, and the input and output specifications of, circuitry to be instantiated on a corresponding PLIC (or a software equivalent of the circuitry). The file 162 (FIG. 7) incorporates or points to these meta-data files so that the host processor 12 (FIG. 1) can 1) determine which firmware files (or software equivalents) stored in the library 120 or in another library will respectively cause the PLICs (or the host processor 12) to instantiate the desired circuitry, or 2) generate one or more of these firmware files (or software equivalents) that are otherwise available, as described in previously incorporated U.S. patent application Ser. Nos. (Attorney Docket Nos. 1934-25-3, 1934-26-3, and 1934-36-3).

[0153] Moreover, the library 120 (FIG. 6) may store one or more of the files 162 (FIG. 7) that the tool 152 generates, so that a designer can incorporate previously designed circuits, such as the circuit 200 (FIG. 10), into a new larger and more complex circuit. The tool 152 may then generate a new file 162 that defines this new circuit.

[0154] Referring to FIGS. 13-17, according to another embodiment of the invention, the tool 152 (FIG. 7) allows one to design a circuit for implementing virtually any complex function f(x) by expanding the function into an equivalent infinite series. Many functions, such as f(x)=cos(x) and f(x)=e^x, can be expanded into an infinite series, such as the Taylor series or the following MacLaurin series, which is a special case (a=0) of the Taylor series:

\[ f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots + \frac{f^{(n)}(0)}{n!}x^n \]  
\[ f(x) = e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \]

Consequently, a combination of summing and multiplying hardwired pipelines 44 interconnected to generate ax+bx^2+cx^3+\ldots+vx^n can implement any function f(x) that one can expand into a MacLaurin series, where the only differences in this combination of pipelines from function to function are the values of the constant coefficients a, b, c, \ldots, v. Therefore, if the tool 152 is programmed with, or otherwise has access to, the coefficients for a number of functions f(x), then the tool can implement any of these functions as a series expansion. Furthermore, because the accuracy of the implementation of a function f(x) is proportional to the number of expansion terms calculated and summed together, the tool 152 may set the number of expansion terms that the interconnected pipelines 44 generate based on the level of accuracy for f(x) that the circuit designer (not shown) enters into the tool. Alternatively, a designer may directly enter a function f(x) into the front end 156 (FIG. 7) of the tool 152 in series-expansion form.

[0155] FIG. 13 is a block diagram of a circuit 240 that the tool 152 (FIG. 7) defines for implementing f(x)=cos(x) as a MacLaurin series according to an embodiment of the invention. For clarity, FIG. 13 shows only the adders, multipliers, and delay blocks that compose the circuit 240, it being understood that the tool 152 may define the circuit for instantiation on one or more PLICs 60 using one or more hardwired pipelines 44 and one or more hardware-interface layers 62 (e.g., FIGS. 10 and 12) per one of the techniques described above in conjunction with FIGS. 7-12. Furthermore, the circuit 240 may be part of a larger circuit (not shown) for implementing an algorithm having cos(x) as one of its portions.

[0156] f(x)=cos(x) is represented by the following MacLaurin series:

\[ \cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots \]  
\[ \cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots \]  
\[ \cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots \]  
\[ \cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots \]

The circuit 240 includes a term-generating section 242 and a term-summing section 244. For clarity, only the parts of these sections that respectively generate and sum the first four power-of-x terms of the cos(x) series expansion are shown, it being understood that any remaining portions of these sections for respectively generating and summing the fifth and higher power-of-x terms are similar.

[0157] The term-generating section 242 includes a chain of multipliers 246, 246, (only multipliers 246, 246, are shown) and delay blocks 244, 244, (only delay blocks 244, 244, are shown) that generate the power-of-x terms of the cos(x) series expansion. The delay blocks 248 ensure that the multipliers 246 multiply powers of x from the same sample time.

[0158] The term-summing section 244 includes two summing paths: a path 250 for positive numbers, and a path 252 for negative numbers. The path 250 includes a chain of adders 254, 254, (only adders 254, 254, are shown) and delay blocks 256, 256, (only blocks 256, and 256, are shown). Similarly, the path 252 includes a chain of adders 258, 258, (only adder 258, is shown) and delay blocks 260, 260, (only blocks 260, and 260, are shown). A final adder 262 sums the cumulative positive and negative sums from the paths 250 and 252 to provide the value for cos(x). Although the adder 262 is shown as summing the first five terms of the expansion (1 and the first four power-of-x terms), it is understood that the final adder 262 may be disposed further down the paths 250 and 252 if the circuit 240 generates additional terms of the cos(x) expansion. Where numbers being summed are floating-point numbers, exceptions, such as a mantissa-register underflow, may occur when a positive number is summed with a negative number that is almost equal to the positive number. But by providing separate summing paths 250 and 252 for positive and negative numbers, respectively, the circuit 240 limits the number of possible locations where such exceptions can occur to a single adder 262. Consequently, providing the separate paths 250 and 252 may significantly reduce the
frequency of such floating-point exceptions, and thus may reduce the time that the peer-vector machine 10 (FIG. 1) consumes handling such exceptions and the size and complexity of the exception manager 86 (FIG. 4).

[0159] Still referring to FIG. 13, the operation of the circuit 240 is discussed according to an embodiment of the invention. For purposes of explanation, it is assumed that each of the multipliers 246, adders 254 and 258, has a latency (i.e., delay) D of one clock cycle. For example, prior to a first clock edge, a value x is present at the inputs of the multiplier 246, and after the first clock edge, the value x2 is present at the output of the multiplier 246. It is understood, however, that the multipliers 246 and adders 254 and 258 may have different latencies and latencies other than one, and that the delays provided by the blocks 248, 256, and 260 may be adjusted accordingly.

[0160] At a start time, a value x1 is present at the input of the multiplier 246, where the subscript “1” denotes the time or position of x1 relative to the other values of x.

[0161] In response to a first clock edge, a value x2 is present at the input of the multiplier 246, and x2 is present at the output of this multiplier. For brevity, this example follows only the propagation of x1, it being understood that the propagation of x1 and subsequent values of x is similar but delayed relative to the propagation of x1. Furthermore, for clarity, x1 is hereinafter referred to “x” in this example.

[0162] In response to a second clock edge, −x2 is present at the output of the multiplier 246, and x2 is available at the output of the block 248.

[0163] In response to a third clock edge, “1” is present at the output of the block 256, x4/4! is present at the output of the multiplier 246, x6 is present at the output of the multiplier 246, and x2 is available at the output of the block 248.

[0164] In response to a fourth clock edge, −x6 is present at the output of the multiplier 246. x4/4! is present at the output of the multiplier 246. x6 is available at the output of the block 248, and “1+x4/4!” is available at the output of the summer 254.

[0165] In response to a fifth clock edge, x8/8! is present at the output of the multiplier 246. “1” is present at the output of the block 256, and “−x2/2!−x6/6!” is available at the output of the block 260.

[0166] In response to a sixth clock edge, “1+x4/4!+x8/8!” is available at the output of the adder 254, and “−x2/2!−x6/6!” is available at the output of the block 260.

[0167] And in response to a seventh clock edge, “cos(x)=1−x2/2!+x4/4!−x6/6!+x8/8!” (cos(x) approximated to the first four power-of-x terms of the MacLaurin series expansion) is available at the output of the adder 262. Therefore, in this example the latency of the circuit 240 (i.e., the number of clock cycles from when x is available at the inputs of the multiplier 246, to when cos(x) is available at the output of the adder 262) is seven clock cycles. Furthermore, if the adder 262 summing a positive number and a negative floating-point number generates an exception, the exception manager 86 (FIG. 4) or the host processor 12 (FIG. 1) may handle this exception using a conventional floating-point-exception routine.

[0168] Alternatively, if the circuit 240 calculates one or more higher power-of-x terms, then the adder 262 is located after (to the right in FIG. 13) the adder that sums the highest generated term to a preceding term, and the operation continues as above.

[0169] Still referring to FIG. 13, alternate embodiments of the circuit 240 are contemplated. For example, the circuit 240 may include multipliers and adders to generate and sum the odd power-of-x terms (e.g., x, x3, x5) with the coefficients of these terms set to zero. Such an alternate circuit 240 is more flexible because it allows one to implement function expansions that include odd powers of x, but in this case would have a greater latency than seven clock cycles.

[0170] FIG. 14 is a block diagram of a circuit 270 that the tool 152 (FIG. 7) defines for implementing f(x)=cos(x) as a MacLaurin series according to another embodiment of the invention. The circuit 270 has a topology that reduces the number of delay blocks and the latency as compared to the circuit 240 of FIG. 13. Furthermore, like FIG. 13, FIG. 14 shows only the adders, multipliers, and delay blocks that compose the circuit 270, it being understood that the tool 152 may define the circuit for instantiation on one or more PLICs 60 using one or more hardware pipelines 44 and one or more hardware-interface layers 62 (e.g., FIGS. 10 and 12) per one of the techniques described above in conjunction with FIGS. 7-12. Furthermore, like the circuit 240, the circuit 270 may be part of a larger circuit (not shown) for implementing an algorithm having cos(x) as one of its portions.

[0171] The circuit 270 includes a term-generating section 272 and a term-summing section 274. For clarity, only the parts of these sections that respectively generate and sum the first four power-of-x terms of the cos(x) series expansion are shown, it being understood that any remaining portions of these sections for respectively generating and summing the fifth and higher power-of-x terms are similar.

[0172] The term-generating section 272 includes a hierarchy of multipliers 276, 276, (only multipliers 276, 276, are shown) and delay blocks 278, 278, (only delay blocks 278, 278, are shown) that generate the power-of-x terms of the cos(x) series expansion. The delay blocks 278 ensure that the multipliers 276 only multiply powers of x from the same sample time.

[0173] The term-summing section 274 includes two summing paths: a path 280 for positive numbers, and a path 282 for negative numbers. The path 280 includes a chain of adders 284, 284, (only adders 284, 284, are shown) and delay blocks 286, 286, (only block 286, is shown). Similarly, the path 282 includes a chain of adders 288, 288, (only adder 288, is shown) and delay blocks 290, 290, (only block 290, is shown). A final adder 292 sums the cumulative positive and negative sums from the paths 280 and 282 to provide the value for cos(x). Although the adder 292 is shown as summing the first five terms of the expansion (and the first four power-of-x terms), it is understood that the final adder 292 may be disposed further down the paths 280 and 282 if the circuit 270 generates additional terms of the cos(x) expansion.

[0174] Still referring to FIG. 14, the operation of the circuit 240 is discussed according to an embodiment of the invention. For purposes of explanation, it is assumed that
each of the multipliers 276, adders 284 and 288, has a latency (i.e., delay) $D$ of one clock cycle. It is understood, however, that the multipliers 276 and adders 284 and 288 may have different latencies and latencies other than one, and that the delays provided by the blocks 278 and 288 may be adjusted accordingly.

At a start time, a value $x$ is present at the input of the multiplier 276.

In response to a first clock edge, $x^2$ is present at the output of the multiplier 276.

In response to a second clock edge, $x^4$ is present at the output of the multiplier 276, and $x^2$ is available at the output of the block 278.

In response to a third clock edge, “1” is present at the output of the block 286, $x^4/4!$ is present at the output of the multiplier 276, $x^6$ is present at the output of the multiplier 276, $-x^2/2!$ is available at the output of the multiplier 276, and $x^2$ is available at the output of the multiplier 276.

In response to a fourth clock edge, $-x^6/6!$ is present at the output of the multiplier 276, $x^8/4!$ is present at the output of the multiplier 276, $-x^2/2!$ is available at the output of the block 290, and “1+x^4/4!” is available at the output of the sum of the two.

In response to a fifth clock edge, “1+x^4/4!+x^8/8!” is available at the output of the adder 284, and “-x^2/2!+x^6/6!+x^8/8!” is available at the output of the adder 288.

And in response to a sixth clock edge, “cos(x)=1-x^2/2!+x^4/4!-x^6/6!+x^8/8!” (cos(x) approximated to the first four power-of-x terms of the Maclaurin series expansion) is available at the output of the adder 292. Therefore, in this example the latency of the circuit 270 is six clock cycles, which is one fewer clock cycle than the latency of the circuit 240 of FIG. 13. But as the number of the power-of-x terms increases beyond four, the gap between the latencies of the circuits 270 and 240 increases such that the circuit 270 provides an even greater improvement in the latency.

Alternatively, if the circuit 270 calculates one or more higher power-of-x terms, then the adder 292 is located after (to the right in FIG. 14) the adder that sums the highest generated term to a preceding term, and the operation continues as above.

Still referring to FIG. 14, alternate embodiments of the circuit 270 are contemplated. For example, the circuit 270 may include multipliers and adders to generate and sum the odd power-of-x terms (e.g., $x$, $x^3$, $x^5$) with the coefficients of these terms set to zero. Such an alternate circuit 270 may be more flexible because it allows one to implement function expansions that include odd powers of $x$ without increasing the circuit’s latency for a given highest power of $x$. That is, where the highest power of $x$ generated by the circuit 270 is $x^6$, adding multipliers and adders to generate $x^2$, $x^4$, and $x^6$ would not increase the latency of the circuit 270 beyond six clock cycles. This is because the circuit 270 would generate the power-of-x terms in parallel, not serially like the circuit 240 of FIG. 13.

FIG. 15 is a block diagram of a power-of-x term generator 300 that to the tool 152 (FIG. 7) defines to replace the power-of-x-term odd multipliers 246, 248, 246, . . . of the term-generating section 242 of FIG. 13 and the power-of-x-term multipliers 276, 278, 278, . . . of FIG. 14 according to an embodiment of the invention. Generally, the generator 300 includes fewer multipliers (here one) than the term-generating sections 242 and 273 (which each include eight multipliers), but may have a higher latency for a given number of generated power-of-x terms. Furthermore, like FIGS. 13-14, FIG. 15 shows only the multipliers and other components that compose the term generator 300, if it being understood that the tool 152 may define a circuit that includes the term generator for instantiation on one or more PLICs 60 using one or more hardwired pipelines 44 and one or more hardware-interface layers 62 (e.g., FIGS. 10 and 12) per one of the techniques described above in conjunction with FIGS. 7-12.

The term generator 300 includes a register 302 for storing $x$, a multiplier 304, a multiplexer 306, and term-storage registers 308, 308, (only registers 308, 308 are shown). For clarity, only the parts of the generator 300 that constitutes the first four power-of-x terms of the cos(x) series expansion are shown, it being understood that any remaining portions of the generator for generating the fifth and higher power-of-x terms are similar.

Still referring to FIG. 15, the operation of the circuit 300 is discussed according to an embodiment of the invention. For purposes of explanation, it is assumed that each of the register 302, multiplier 304, and registers 308, 308 has a respective latency (i.e., delay) of one clock cycle, and that the multiplexer 306 is not clocked, i.e., is asynchronous. It is understood, however, that the register 302, multiplier 304, and registers 308 may have different latencies and latencies other than one, that the multiplexer 306 may be clocked and have a latency of one or more clock cycles, and that the term-summing sections 244 and 274 of FIGS. 13 and 14, respectively, may be adjusted accordingly.

At a start time, a value $x$ is present at the input of the register 302.

In response to a first clock edge, the current value of $x$ is loaded into, and thus is present at the output of, the register 302, and is present at the output of the multiplexer 306, which couples its input 312 to its output. The register 302 is then disabled. Alternatively, the register 302 is not disabled but the value of $x$ at the input of this register does not change.

In response to a second clock edge, $x^2$ is present at the output of the multiplexer 304, and the multiplexer changes state and couples its input 314 to its output such that $x^2$ is also present at the output of the multiplexer 306.

In response to a third clock edge, $x^2$ is loaded into, and thus is available at the output of, the register 3101, and $x^2$ is available at the output of the multiplexer 304 and at the output of the multiplexer 306.

In response to a fourth clock edge, $x^4$ is available at the output of the multiplexer 304 and at the output of the multiplexer 306.

In response to a fifth clock edge, $x^4$ is loaded into, and thus is available at the output of, the register 3102, and $x^4$ is available at the output of the multiplexer 304 and at the output of the multiplexer 306.
In response to a sixth clock edge, $x^6$ is available at the output of the multiplier 304 and at the output of the multiplexer 306.

In response to a seventh clock edge, $x^6$ is loaded into, and thus is available at the output of, the register 310, and $x^6$ is available at the output of the multiplier 304 and at the output of the multiplexer 306.

In response to an eighth clock edge, $x^8$ is available at the output of the multiplier 304 and at the output of the multiplexer 306.

And in response to a ninth clock edge, $x^8$ is loaded into, and thus is available at the output of, the register 310, the next value of $x$ is loaded into the register 302. But if the generator 300 generates powers of $x$ higher than $x^8$, the generator continues operating in the described manner before loading the next value of $x$ into the register 302.

After the generator 300 generates all of the specified powers of the current value of $x$, the register 302, multiplier 304, multiplexer 306, and registers 310 repeat the above procedure for each subsequent value of $x$.

Alternative embodiments of the generator 300 are contemplated. For example, to generate the odd powers of $x$ for a function other than cos($x$), one can merely add additional registers 310 to store these values, because the multiplier 304 inherently generates these odd powers. Alternatively, the generator 300 may be modified to load $x^8$ into the register 302 so that the multiplier 304 thereafter generates only even powers of $x$. Moreover, one or more of the registers 308 may be eliminated, and the multiplexer 306 may feed the respective powers of $x$ directly to the term multipliers, e.g., the term multipliers 246, 246, 246, 246, 246, ... of FIG. 13 and the term multipliers 276, 276, 276, 276, 276, ... of FIG. 14.

FIG. 16 is a block diagram of a circuit 320 that defines $f(x)=e^x$ as a MacLaurin series according to an embodiment of the invention. The circuit 320 is similar to the circuit 240 of FIG. 13, but because the odd power-of-$x$ terms for the $e^x$ expansion may be positive or negative, the circuit 320 also includes sign determiners (described below and in conjunction with FIG. 17) that respectively provide these odd-power-of-$x$ terms to the proper path (positive or negative) of the term-summing section. For clarity, FIG. 16 shows only the adders, multipliers, delay blocks, and sign determiners that compose the circuit 320, it being understood that the tool 152 may define the circuit for instantiation on one or more PLICs 60 using one or more hardwired pipelines 44 and one or more hardware-interface layers 62 (e.g., FIGS. 10 and 12) per one of the techniques described above in conjunction with FIGS. 7-12. Furthermore, the circuit 320 may be part of a larger circuit (not shown) for implementing an algorithm having $e^x$ as one of its portions.

The circuit 320 includes a term-generating section 322 and a term-summing section 324, which includes positive- and negative-value summing paths 326 and 328. For clarity, only the parts of these sections that respectively generate and sum the first five power-of-$x$ terms of the $e^x$ series expansion are shown, it being understood that any remaining portions of these sections for respectively generating and summing the sixth and higher power-of-$x$ terms are similar.

The term-generating section 322 includes a chain of multipliers 330, 330 (only multipliers 330, 330, 330, are shown) and delay blocks 332, 332 (only delay blocks 332, 332, are shown) that generate the power-of-$x$ terms of the $e^x$ series expansion. The section 322 also includes, for each odd-power-of-$x$ term (e.g., $x$, $x^3$, $x^5$, ...), a respective sign determiner 334, 334, (only determiners 334, 334, are shown) that directs positive values of the odd-power-of-$x$ term to the positive summing path 326 of the term-summing section 324, and that directs negative values of the odd-power-of-$x$ term to the negative summing path 328.

The positive-value path 326 of the term-summing section 324 includes a chain of adders 336, 336, (only adders 336, 336, are shown) and delay blocks 338, 338, (only blocks 338, 338, are shown). Similarly, the negative-value path 328 includes a chain of adders 340, 340, (only adders 340, 340, are shown) and delay blocks 342, 342, (only blocks 342, 342, are shown). A final adder 344 sums the cumulative positive and negative sums from the paths 326 and 328 to provide the value for $e^x$. Although the final adder 344 is shown as summing the first six terms of the $e^x$ expansion ("1" and the first five power-of-$x$ terms), it is understood that the final adder may be disposed further down the paths 326 and 328 if the circuit 320 generates additional terms of the expansion.

Still referring to FIG. 16, the operation of the circuit 320 is discussed according to an embodiment of the invention. For purposes of explanation, it is assumed that each of the multipliers 330, sign determiners 334, and adders 336 and 340 has a latency (i.e., delay) 1 of one clock cycle. It is understood, however, that the multipliers 330, sign determiners 334, and adders 334 and 336 may have different latencies and latencies other than one, and that the delays provided by the blocks 332, 338, and 342 may be adjusted accordingly.

At a start time, a value $x$ is present at both inputs of the multiplier 330, at the input of the delay block 332, and at the input of the sign determiner 334.

In response to a first clock edge, $x^2$ is available at the output of the multiplier 330, $x$ is available at the output of the delay block 332, and "$x^1$" is available at the output of the delay block 338. Furthermore, if $x$ is positive, $x$ and logic "0" are respectively available at the (+) and (−) outputs of the sign determiner 334; conversely, if $x$ is negative, logic "0" and $x$ are respectively available at the (+) and (−) outputs of the determiner 334.

In response to a second clock edge, $x^2/2!$ is available at the output of the multiplier 330, $x^1$ is present at the output of the multiplier 330, and $x$ is available at the output of the delay block 332. Furthermore, if $x$ is positive, "$1+x^1$" is available at the output of the adder 336; conversely, if $x$ is negative, "1+0=1" is present at the output of the adder 336.
In response to a third clock edge, \( x^3/x! \) is available at the output of the multiplier \( 330_i \), \( x^4 \) is available at the output of the delay block \( 332_i \), and \( 1+i+x^2/2! \) (x positive) or \( 1+i+x^2/2! \) (x negative) is available at the output of the delay block \( 338_i \). Furthermore, if \( x^7/3! \), and thus x, is positive, \( x^3/3! \) and logic \( 0 \) are respectively present at the \((+)\) and \((-)\) outputs of the sign determiner \( 334_i \). Conversely, if \( x^7/3! \), and thus x, is negative, logic \( 0 \) and \( x^3/3! \) are respectively present at the \((+)\) and \((-)\) outputs of the sign determiner \( 334_i \). Moreover, if x is negative, then \( x \) is available at the output of the delay block \( 342_i \), conversely, if x is positive, then logic \( 0 \) is available at the output of the delay block \( 342_i \).

In response to a fourth clock edge, \( x^4/4! \) is present at the output of the multiplier \( 330_i \), \( x^5 \) is available at the output of the delay block \( 332_i \), and \( 1+i+x^2/2! \) (x positive) or \( 1+i+x^2/2! \) (x negative) is available at the output of the delay block \( 338_i \). Furthermore, if \( x^7/3! \), and thus x, is positive, \( x^3/3! \) and logic \( 0 \) are respectively present at the \((+)\) and \((-)\) outputs of the sign determiner \( 334_i \), conversely, if \( x^7/3! \), and thus x, is negative, logic \( 0 \) and \( x^3/3! \) are respectively present at the \((+)\) and \((-)\) outputs of the sign determiner \( 334_i \). Moreover, if x is negative, then \( x \) is available at the output of the delay block \( 342_i \), conversely, if x is positive, then logic \( 0 \) is available at the output of the delay block \( 342_i \).

In response to a fifth clock edge, \( x^5/5! \) is available at the output of the multiplier \( 330_i \), \( 1+i+x^2/2!+x^3/3! \) (x positive) or \( 1+i+x^2/2! \) (x negative) is available at the output of the delay block \( 338_i \), \( x^4/4! \) is available at the output of the delay block \( 332_i \), and \( 0 \) (x positive) or \( -x^7/3! \) (x negative) is available at the output of the adder \( 340_i \).

In response to a sixth clock edge, \( 1+i+x^2/2!+x^3/3!+x^4/4!+x^5/5! \) (x positive) or \( 1+i+x^2/2!+x^3/3! \) (x negative) is available at the output of the adder \( 336_i \), and \( 0 \) (x positive) or \( -x^7/3!-x^4/4! \) (x negative) is available at the output of the adder \( 340_i \).

And in response to a seventh clock edge, \( c^e=-1+x+x^2/2+i+x^2/2-i+x^2/2+i \) (x positive) or \( c^e=1-x+x^2/2-i-x^2/2+i \) (x negative) is available at the output of the adder \( 344 \).

Therefore, in this example, the latency of the circuit \( 320 \) is eight. Furthermore, if the adder \( 344 \), while summing a positive number and a negative floating-point number, generates an exception, the exception manager \( 86 \) (FIG. 4) or the host processor \( 12 \) (FIG. 1) may handle this except with using a conventional floating-point exception routine.

Alternatively, if the circuit \( 320 \) calculates one or more power-of-x terms higher than the fifth power, then the adder \( 344 \) is located after (to the right in FIG. 16) the adder \( 336 \) or \( 340 \) that sums the highest generated term to a preceding term, and the operation continues as above.

Still referring to FIG. 16, alternate embodiments of the circuit \( 320 \) are contemplated. For example, one may replace the term-generating section \( 322 \) with a section similar to the term-generating section \( 272 \) of FIG. 14, or may replace the chain of multipliers \( 330 \) with a power-of-x generator similar to the generator \( 300 \) of FIG. 15.

FIG. 17 is a block diagram of the sign determiner \( 334 \) of FIG. 16 according to an embodiment of the invention, it being understood that the sign determiners \( 334_i \), \( 334_i \), are similar.

The sign determiner \( 334 \) includes an input node \( 350 \), a (−) output node \( 352 \), a (+) output node \( 354 \), a register \( 356 \) that stores a logic “0”, and demultiplexers \( 358 \) and \( 360 \).

The demultiplexer \( 358 \) includes a control node \( 362 \) coupled to receive a sign bit of the value at the input node \( 350 \), a (−) input node \( 364 \) coupled to the input node \( 350 \), a (+) input node \( 366 \) coupled to the register \( 356 \), and an output node \( 368 \) coupled to the (−) output node \( 352 \).

Similarly, the demultiplexer \( 360 \) includes a control node \( 370 \) coupled to receive the sign bit of the value at the input node \( 350 \), a (−) input node \( 372 \) coupled to the register \( 356 \), a (+) input node \( 374 \) coupled to the input node \( 350 \), and an output node \( 376 \) coupled to the (−) output node \( 354 \).

Still referring to FIG. 17, two operating modes of the sign determiner \( 334 \) are described according to an embodiment of the invention.

In one operating mode, the sign determiner \( 334 \) receives at its input node \( 350 \) a positive (+) value \( v \), which, therefore, includes a positive sign bit. This sign bit is typically the most-significant bit of \( v \), although the sign bit may be any other bit of \( v \). In response to the positive sign bit, the demultiplexer \( 360 \) couples \( v \) (including the sign bit) from its (+) input node \( 374 \) to its output node \( 376 \), and the (+) output node \( 354 \) of the sign determiner \( 334_i \). Furthermore, the demultiplexer \( 358 \) couples the logic “0” stored in the register \( 356 \) from the (+) input node \( 366 \) to the output node \( 368 \), and thus to the (−) output node \( 352 \) of the sign determiner \( 334_i \).

In the other operating mode, the sign determiner \( 334 \) receives at its input node \( 350 \) a negative (−) value \( v \), which, therefore, includes a negative sign bit. In response to the negative sign bit, the demultiplexer \( 358 \) couples \( v \) (including the sign bit) from its (−) input node \( 364 \) to its output node \( 368 \), and thus to the (−) output node \( 352 \) of the sign determiner \( 334_i \). Furthermore, the demultiplexer \( 360 \) couples the logic “0” stored in the register \( 356 \) from the (−) input node \( 372 \) to the output node \( 376 \), and thus to the (−) output node \( 354 \) of the sign determiner \( 334_i \).

Still referring to FIG. 17, alternative embodiments of the sign determiner \( 334 \) are contemplated. For example, one may replace the logic “0” register with a component, such as pull-down resistor, coupled to a logic “0” voltage level, such as ground.

Referring to FIGS. 1-17, alternate embodiments of the peer vector machine \( 10 \) are contemplated. For example, some or all of the components of the peer vector machine \( 10 \), such as the host processor \( 12 \) (FIG. 1) and the pipeline units \( 50 \) (FIG. 3) of the pipeline accelerator \( 14 \) (FIG. 1), may be disposed on a single integrated circuit.

The preceding discussion is presented to enable a person skilled in the art to make and use the invention. Various modifications to the embodiments will be readily apparent to those skilled in the art, and the generic principles
herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

What is claimed is:

1. A computer-based design tool, comprising:
   a front end operable to receive symbols that define an algorithm;
   an interpreter coupled to the front end and operable to parse the algorithm into respective algorithm portions; and
   a generator coupled to the interpreter and operable to identify a corresponding circuit template for each of the algorithm portions, each template defining a circuit for executing the respective algorithm portion, and interconnecting the identified templates such that the interconnected templates define a circuit that is operable to execute the algorithm.

2. The design tool of claim 1 wherein the generator is operable to generate a file comprising:
   a respective pointer to each of the identified templates within a template library; and
   a list of the interconnections between the identified templates.

3. The design tool of claim 1 wherein the symbols comprise mathematical symbols.

4. The design tool of claim 1 wherein the interpreter is operable to parse the algorithm into respective algorithm portions that each correspond to a template in a library.

5. The design tool of claim 1 wherein the generator is operable to identify the corresponding templates by accessing a library that includes the identified templates.

6. The design tool of claim 1, further comprising a library coupled to the generator and operable to store the identified templates.

7. The design tool of claim 1, further comprising a simulator coupled to the generator and operable to simulate operation of the circuit by determining a transfer function of the circuit defined by the interconnected templates.

8. The design tool of claim 1 wherein:
   the front end is further operable to receive a desired operational characteristic of the circuit; and
   the generator is further operable to identify the corresponding template for each of the algorithm portions such that the interconnection of identified templates defines the circuit having the desired operational characteristic.

9. The design tool of claim 8 wherein the desired operational characteristic comprises a latency of the circuit.

10. The design tool of claim 1 wherein:
   the front end is further operable to receive an identity of a platform; and
   the generator is further operable to identify a hardware-abstraction-layer template corresponding to the platform, and
   interconnect the identified circuit templates to the identified layer template such that the interconnection of the templates defines the electronic circuit.

11. The tool of claim 1 wherein:
   the front end is further operable to receive an identity of a platform; and
   the generator is further operable to determine whether the circuit defined by the interconnection of the identified templates can be instantiated on the identified platform.

12. A computer-based design tool, comprising:
   a front end operable to receive symbols that define an algorithm;
   a generator coupled to the front end and operable to:
       identify a template that defines a first electronic circuit that is operable to execute the algorithm,
       identify a template that defines a hardware interface that is compatible with the first electronic circuit, and
       interconnect the identified templates to define a resulting electronic circuit that includes the first circuit interconnected to the hardware interface.

13. A method, comprising:
   parsing an algorithm into a combination of respective smaller algorithms;
   identifying a corresponding template for each of the smaller algorithms, each template defining a respective circuit that is operable to execute the respective smaller algorithm; and
   interconnecting the identified templates such that the interconnected templates define an electronic circuit that is operable to execute the algorithm.

14. The method of claim 13, further comprising:
   generating a respective pointer to each of the identified templates within a library; and
   generating a list of the interconnections between the identified templates.

15. The method of claim 13, further comprising:
   receiving an expression of mathematical symbols that defines the algorithm; and
   wherein parsing the algorithm comprises parsing the expression into groups of symbols that respectively define the smaller algorithms.

16. The method of claim 13 wherein identifying the corresponding templates comprises searching a library that includes the corresponding templates.

17. The method of claim 13, further comprising simulating the electronic circuit by:
   determining a transfer function of the circuit from characteristics of the interconnected templates; and
   determining a signal output from the circuit in response to a signal input to the circuit.

18. The method of claim 13 wherein identifying the corresponding templates comprises identifying the templates such that the interconnection of the templates represents the electronic circuit having a predetermined operational characteristic.
19. The method of claim 13, further comprising:
identifying an interface template that defines a hardware
interface that is compatible with a predetermined plat-
form on which the electronic circuit can be instantiated;
and
wherein interconnecting the templates comprises inter-
connecting the circuit templates to the interface tem-
plate such that the interconnection of the circuit and
interface templates defines the electronic circuit.

20. The method of claim 13, further comprising deter-
mining whether the electronic circuit can be instantiated on
a predetermined platform.

21. A method, comprising:
identifying a circuit template that defines a first electronic
circuit that is operable to execute an algorithm;
identifying an interface template that defines a hardware
interface that is compatible with the first electronic
circuit; and
interconnecting the identified circuit and interface tem-
plates to generate a definition of a resulting electronic
circuit that includes the first circuit interconnected to
the hardware interface.

22. The method of claim 21, further comprising using the
definition to instantiate the resulting electronic circuit on a
programmable logic circuit.

23. The method of claim 21, further comprising using the
definition to instantiate the resulting electronic circuit on a
programmable logic circuit having signal pins such that the
hardware interface is disposed between the pins and the first
circuit.

24. The method of claim 21, further comprising simulat-
ing operation of the resulting electronic circuit based on
information included in the circuit and interface templates.

25. The method of claim 21, further comprising simulat-
ing operation of the resulting electronic circuit based on
information included in a description file that corresponds to
the circuit and interface templates.

26. A computer-readable medium, that when executed by
a processor, causes the processor to:
parse an algorithm into a combination of respective
smaller algorithms;
identify a corresponding template for each of the smaller
algorithms, each template defining a respective circuit
that is operable to execute the respective smaller algo-

* * * * *