JIM: (CLAIM) IS USED TO INDICATE WHAT I THINK IS A CLAIM

DO I NEED TO INCLUDE LANGUAGE SYNTAX EXAMPLES?

COMPUTER CHIP DESIGN

1. CONFIGURABLE/PARAMETERIZABLE SPECIFICATION OF HARDWARE COMPONENTS AND THE RELATIONSHIP BETWEEN THOSE COMPONENTS AND THE CONFIGURABLE/PARAMETERIZABLE CROSS FUNCTIONAL RELATIONSHIPS IN CHIP DESIGN ARCHITECTURAL, DESIGN AND VERIFICATION COMPONENTS WHICH IS COMPILED, CHECKED AND CONVERTED INTO C++/RTL CODE AND DOCUMENTATION
2. COMPILER-OBJECT MODEL TRANSFORMATION-HIGH LEVEL LANGUAGE TO RTL/C++/DOCUMENTATION
3. GENERATE SKELETON CODE FOR CYCLE BASED C++ SIMULATOR HIERARCHY/INTERFACES, VECTOR/STATE DATA GENERATOR, MEMORIES, MEMORY MAP CONSTANTS, ISA CONSTANTS FOR TRANSACTION LEVEL MODELING
4. INTERCONNECT/DESIGN HIERARCHY AUTO ROUTER
5. MEMORY MAP/AUTO MAPPER
6. PIPELINES
7. INSTRUCTION SET ARCHITECTURE
8. VERIFICATION COMPONENTS/TEST BENCHES
9. REGISTER FILES, FIFOS, MEMORIES
10.

## BACKGROUND

The present disclosure relates to computer chip design technologies.

In the past computer chip design was done exclusively with schematics of the gates and their connections. Designing computer circuits using gates was a slow process. Hardware description languages and tools were developed which required fewer details to describe circuits. A compiler is used to synthesize/transform the HDL files into circuits, thus abstracting the design description. The software describes the functional behavior or structural boolean equations but does not describe the physical implementation of the actual circuits. Instead a synthesis compiler determines the physical timing and hence the gate sizes and topology pf the circuits. Currently a designer of a computer chip creates a software description of a computer chip design by typing in RTL code or drawing schematics. This process can be quite time consuming for the designer because all implementation details are described in software and gates. Designs and verification environments needs to be interfaced with one another in order to verify the design with known good values. Connecting the verification environment to the design manually is labor intensive. Creating the

verification environment automatically and connecting the stimulus and expected vectors to the design has been typically done manually.

5

SUMMARY

In a general aspect, the present invention relates to a method, including a high level language that is used to specify and is compiled to create parts of computer chip designs, verification frameworks, and simulator skeletons. The missing parts are in turn designed by software programmers and designers and are then in turn "plugged" into the frameworks and skeletons to form a working design, verification environment, and simulator. While many of the elements described below are available in point tools we are not aware of a single solution in the industry that offers the combined features and hence efficiency of the solution described in this patent.

In another general aspect, the present invention relates to a method, including which uses a higher level of design abstraction and design capture mechanism to capture the cross functional relationships between functional units on a chip which is not possible with an RTL language or schematics. The higher level language must describe the architectural details at a higher level. The higher level language must be a language that can describe the architecture of the chip, the implementation "skeleton", and the verification framework of the chip. The high level language is then compiled and it generates RTL and C++ code and documentation.

In yet another general aspect, the present invention relates to a method, including A shared specification that has one source to avoid confusion and misunderstandings, eliminate duplication of the source in documents and source code, and to facilitate rapid changes to the specification which can be immediately propagated throughout the design team. A single programmatic source for the specification achieves all of the above goals. Implementations of the language may include one or more of the following aset of language constructs that describe the various aspects of chip architecture specifications, chip verification specifications, and chip design implementations and a compiler which can check the different parts of the specification against one another. This is not available in other tools that we know of.

The method can further include constructing the object models comprising a plurality of categories each defined by a category name, wherein each of plurality of categories is associated with a plurality of language syntax elements in the language, wherein each language syntax element is associated with other language syntax elements.

The method can further include constructing the verification environment automatically and connecting the stimulus and expected vectors to the design using the single interconnect and verification component specification.

## BRIEF DESCRIPTION OF THE DRAWINGS

The following drawings, which are incorporated in and form a part of the specification, illustrate embodiments of the present invention and, together with the description, serve to explain the principles of the invention.

Figure 1 is a schematic diagram for the compiler components and the processes in accordance with one implementation of the present specification.

Figure 2 is a schematic diagram for the

and the processes in accordance with another implementation of the present specification.

Figure 3 illustrates the operation of the auto router in accordance with the present specification.

Figure 4 illustrates the flow diagram showing the steps to create and compile CSL code and the generated output and how the code is used.

Figure 5 illustrates the CSL micro engine.

Figure 6 illustrates the CSL system on chip (SOC).

Figure 7 illustrates the CSL network of processors on chip.

Figure 8 illustrates the C++ simulator plus testbench

Figure 9 illustrates the Example memory map and SOC bus drawing with address detectors ( a copy is in Alina's notebook)

Figure 10 illustrates the ISA tree of isa elements and fields

Figure 11 illustrates the hierarchical fields

Figure 12 illustrates the signal group

Figure 13 illustrates the auto router figure with all connection types shown

Figure 14 illustrates the pipeline

Figure 15 illustrates the micro engine as an example of what can be built using CSL

Figure 16 illustrates the IDE layout

## DETAILED DESCRIPTION

The present specification addresses the drawbacks of current chip design languages and design and verification methodologies and flows. Examples of current chip design languages are Verilog and VHDL which are used to describe circuits using behavioral and structural descriptions.

CSL is a high level language used to capture a chip design's architecture, design implementation, and verification infrastructure details. Register transfer level language (RTL) describes computer designs at a logic level. CSL describes cross functional areas such as the chip's memory map, the chip's instruction set architecture, the chip's hierarchy and interconnect, the aggregate structures such as special registers, FIFO's and register files, connections between simulation models and the test benches used to verify designs and the designs. Moreover, the

CSL language is used to implement more complex aggregate structures such as networks on a chip, SOC fabrics, hierarchical MBIST

CSL is an object oriented language that uses many of the same syntax elements as C++ (classes, constructors, inheritance) and additional syntactic sugar borrowed from RTL languages (formal to actual mappings, bit ranges).

The CSL compiler (CSLC) parses CSL files, creates an object model in memory, elaborates the object model using algorithms to transform the higher level description into a lower level detailed model which is the implementation model, checks the implementation model for correctness, and finally generates RTL, C++, and documentation.

One of the main features of CSL and cslc is the generation of the same information in multiple output languages. This reduces and/or eliminates the “book keeping” required to ensure that the different language files for a particular aspect of the design contain the same information. Different teams on the project need to use the same (including but not limited to) numbers, symbol names, and protocols but need to code in different languages. Use of a specification with these numbers, symbol names, and protocols which is then compiled into different languages ensures that all teams use the same specification concurrently. A the single CSL source for the specification can be checked by a compiler for specification correctness.

A cross functional area is defined as a set of related functions which are defined across many functional units. Memory maps affect many functional units on a chip. The hierarchy and interconnect on a chip affect many units. The instruction set architecture (ISA) affects many units and defines to a certain extent the pipeline(s) on chip. These cross functional areas need to be specified in an architecture, implemented by designers, tested by the verification team, used by the software team, and documented by the documentation team. The cross functional areas are therefore a shared specification used across the chip and by many different teams.

In addition cross functional areas are related to each other. This will be described later in this document. Relationships between cross functional areas in the past have been checked by generating code, checking for compilation errors and simulating and checking for simulation errors. This process is expensive in terms of engineering time, computing time and resources, and EDA licenses for RTL simulation tools. Finding errors in the specification prior to simulation is facilitated by using a specification language (CSL), which is checked by a specification compiler (CSLC).

**Generate equivalent constants and functions in different languages.**

(CLAIM) A high level language can be used to generate equivalent constants and functions in different languages. The equivalent code in different languages eliminates parallel code streams and guarantees that the different teams are using the same numbers and methods. A high level language specification is compiled and the compiler generates RTL and C++ code and documentation in different languages and forms from the same original specification.

**CSL Interconnect Specification**

CSL facilitates the architecture, design, implementation, simulation, generation of test, generation of software tools and documentation of complex chips. A chip is a set of functional units. Units have interfaces (ports), which are connected to other ports or to the pads of the chip (the system interface). The functional unit hierarchy and interconnections are described using CSL.

**CSL Memory Map Specification**

The functional units communicate with one another and with the system and software running on other chips via the interfaces. The functional units have address maps which are written to or read from by other functional units or by software. An address map associates an address or address range with a state element or aggregate state element respectively. Address buses in conjunction with operation (read/write), control and data lines write or read data to or from the state elements in the address map. Address maps for chips can be either flat (linear

address range for entire chip), hierarchical (a unit address to select a functional unit and a subsequent local address to select a local state element), or virtual where the unit address and the local address are combined in a single address bus operation. Address buses may be implemented using different topologies such as a bus, token ring, tree, or mesh.

A major problem arises when the functional and alternate models are written in different languages (unconnected tools). The use of different languages creates a need to assure that address map constants and functions are the same in each language. (CLAIM) A only a specification language such as CSL can automatically generate *consistent* address map constants and functions.

**CSL enumerated Type Specification**

Enumerated types which are identical to C++ enumerated types may be specified. (CLAIM) Enumerated types may be associated with signals and fields.

Other high level and RTL languages do not generate decoders from enumerated types associated with hardware elements and connectivity objects (e.g. wires). (CLAIM) CSL generates decoders from enumerated types associated with fields in hardware elements and connectivity objects (e.g. wires). (CLAIM) The decoders are constructed correctly by the CSL compiler. (CLAIM) Constants in functional languages such as C++ are generated correctly by the CSL compiler and can be used to construct software which aligns information (values) in the correct position associated with the field in the hardware model or actual hardware that is connected to the decoder.

**CSL Field Specification**

Fields are bit ranges that are named objects. Fields have an upper and a lower bit range index. The field has a name. Decoders can be automatically generated from fields associated with connectivity objects and register or other state elements . Decoders can be automatically generated from enums associated with fields in connectivity objects and register or other state elements . Other languages do not

generate decoders from fields. The decoders are constructed correctly. (CLAIM) A high level language field may be joined with other fields using a linked list like set_next or set_previous operation and that the linked list may be added to connectivity objects and register or other state elements. Note that there are other applications of the linked list of fields and this claim should not be limited to just associating linked lists of fields with connectivity objects and register or other state elements

(CLAIM) A high level language field can be used to construct a hierarchical tree of fields and that the hierarchical tree of fields can be implemented using different programming methods but not limited to trees or inheritance. Note that there are other applications of the hierarchical tree of fields and this claim should not be limited to just associating linked lists of fields with connectivity objects and register or other state elements. (CLAIM) A high level language hierarchical field specification are used in the design of instruction sets. (CLAIM) A that high level language enumerated type can be associated with a high level language field(s). (CLAIM) A that high level language field specification associated with high level language decoder generation method and that decoders can be automatically generated from high level language field specification.

**CSL instruction set architecture Specification**

Functional units and alternate models (e.g. a simulation model) may execute programs which are built from an instruction set architecture (ISA) specification written in CSL. The instruction set architecture is described using CSL. The functional units and the external model (e.g. a simulation model outside of the functional unit) that implement the instruction set architecture, and execute programs written in the instruction set architecture assembly language, need to use the same set of instruction formats, instruction field definitions and enumerated type values.

A major problem arises when the functional and alternate models are written in different languages. The use of the shared constants in the different languages by the different teams require that the same values be generated in the different languages. The constants describe the width and position of each field in each

instruction format and instruction and associated functions. Additionally functions are auto generated which insert and extract values to and from the instruction words. The constants are automatically generated from a specification language such as CSL.

(CLAIM) A high level specification language can be used specify constant values in functional areas and that those constants in those functional areas are related to other constants in other high level specification language functional areas and that the combination of the areas and constants in the high level specification can be compiled and the compiler can generate constants in different languages and in different files for use by different teams on a chip design project in order so that the teams have the same constant values represented in different languages. (CLAIM) A high level language description of an instruction set architecture in a tree or inheritance hierarchy or any other hierarchical data structure can be compiled and the compiler can automatically generate RTL and alternate model languages which eliminates inconsistencies between the functional unit and alternate models.

(CLAIM) A instruction formats and instructions can be built using hierarchies of fields. (CLAIM) A enumerated types are created to specify the opcodes and other sets of select values for fields in an instruction set. The enumerated types are associated with a field in an instruction format or instruction. Instruction formats may be inherited by other instruction formats or instructions or may be connected in a tree. (CLAIM) A ISA's specified by a high level language description can be associated with register specified by a high level language description. (CLAIM) A decoders can be generated for fields and enums associated with registers specified by a high level language description which have an associated ISA's specified by a high level language description.

The ISA has several types of information that are used in the processor data path. 1. The encoded information such as opcodes. 2. The selectors such as register file address fields. The encoded information is decoded by decoders automatically generated by CSL. The “selectors” are fields. The fields are connected to signals. The signals are created by using a hierarchical identifier to refer to the signal name

and then the hierarchical path to the field in the instruction. The hierarchical path includes the instruction name and the field name.

Example

```
csl_signal [31:0] ir;
ir.associate_isa(isa);

csl_signal alu_opcode(ir.isa.alu_instruction.alu_opcode.get_width()) =
ir.isa.alu_instruction.alu_opcode;
```

selects the alu_opcode field from the signal.

ISA's contain two types of elements instruction formats and instructions which use the instruction formats. ISA elements are derived from a common base format which specifies the width of the ISA instruction formats and the name of the ISA. Sub ISA instruction formats can be inherited from parent ISA formats. ISA instructions inherit an ISA instruction format. Instruction formats and instructions contain ISA fields. An ISA field is a named bit range with a width. Fields are placed into instruction formats and fields in either absolute positions or relative to other fields or the left or right edge of the instruction. Since there is a hierarchical relationship between the ISA instructions formats and the ISA instructions and there is one root format a tree of ISA instructions formats and the ISA instructions may be constructed to represent the ISA. The ISA tree may then be analyzed to form instruction decoders. A hierarchical identifier is a set of names in scope holders separated by a period. An hid forms a hierarchical path to an element in the ISA. Hierarchical identifiers (hid) can be used to access any instruction format, instruction, field, or enum in the ISA. The hid is a reference to an object in the ISA. Operations such as decoder or signal generation can be performed using the hid information. ISA fields have a type. The type information is used to control the generation of hardware in the control and data paths. the type information can also be used to check the usage of the hid's in the designer's custom code. For example, the type information may indicate that the field is a selector which means it may be used as a mux select. Or the field may be a branch address so it must be used as an

input to a branch address mux, or the field is an opcode so it must be decoded and used to enable or select objects.

(CLAIM) A high level language compiler can check the ISA specification for the following types of errors: pipeline for data and control hazards, register leaks, data duplication problems related to bad stall logic

(CLAIM) The high level language ISA specification can tie the hardware registers which hold ISA instructions to the high level language interconnect specification.

**CSL Register Specifications**

(CLAIM) A high level language can be used to specify registers. (CLAIM) The registers can be added to the high level language memory map specification. The registers can be addressed for read and/or write operations. The CSLC generates hardware which selects the registers for read and/or write operations. instruction set architecture's can be associated with registers. Fields in registers can be defined (CLAIM) A high level language register specification can be associated with other high level specifications such as but not limited to connections, memory map locations, decoders, enumerated types. (CLAIM) A high level language register specification can be used as a building block to construct aggregate structures described in the hig level language.

**CSL Decoder Specifications**

(CLAIM) A CSL can be used to specify that decoders be generated from signals, registers, fields, and instruction set architecture's. The fields can reside in signals or registers. Decoders can be generated from registers with instruction set architecture's associated with them. The decoders can have enable signals which qualify the decoders outputs. Decoders can be generated to decode the memory addresses using the CSL memory map specification.

**CSL Pipeline Specifications**

Digital pipelines can be specified using CSL, checked for correctness by a compiler, and the compiler can then generate code in different languages for use by members of a hardware and software team. The functional units contain logic which implements an algorithm or  instruction set architecture as a digital pipeline.

Pipelines contain pipe stages. The pipe stages in a pipeline share signals such as clock, reset, stalls, v**alid, data, and control signals**. Moreover, there are signals common to the connection between each pipe stage such as data, valid, and control. (CLAIM) A digital pipelines can be specified using a high level language, the high level language language specification can be checked for correctness by a compiler, and the compiler can then generate code in different languages for use by members of a hardware and software team.

In fact sets of data, valid, and control signals can be modeled as three separate pipelines whose latencies, clock, reset, and stalls are connected identically in each pipeline. Branches and merges of the three different types of parallel pipelines requires careful design and any changes to one pipeline must be mirrored in the two other pipelines. **The user must use the CSL pipeline mirroring commands to associate pipelines.** Register leaks (data loss due to a pipeline continuing to execute when a downstream pipe stage is stalled) or register data duplication (as a result of failing to qualify a valid bit with a down stream stall signal) are all problems faced by designers of digital pipelines. Pipeline design and subsequent changes can be simplified by specifying the digital pipeline in a CSL specification. The specification can be checked by cslc for the above mentioned problems and the designer can make the necessary changes. CSL pipelines can be combined with interconnect aggregate structures (signal groups) to create maintainable pipelines where the definition of the signals in the pipeline is done once in a signal group and the signals are then added to the pipeline from a starting pipe stage to an ending pipe stage. The pipeline specification can be checked for correctness by CSLC, the CSL compiler.

Valid bit qualification on stall and valid bit cancellation on pipe stage branch are just some of the pipeline properties checked in the CSL compiler. (CLAIM) A digital pipeline can be specified using CSL, checked for correctness by a compiler, and the compiler can then generate code in different languages for use by members of a hardware and software team. (CLAIM) A digital pipeline can be specified using a high level language can specify that stall signals be correctly connected to all state elements automatically. (CLAIM) The stall signal digital pipeline specification using

a high level language can be checked for register leaks and duplicate data in the pipeline. (CLAIM) The valid bit signal digital pipeline specification using a high level language can be checked for correct qualification in branch and merge points and when the pipeline is connected to stall signal(s). (CLAIM) The high level language pipeline specification with branch points can be compiled and that the down stream merge point can have a checker to verify that only one branch has a valid bit at any point in time at the merge point . (CLAIM) The digital pipeline specification using a high level language can be used to create copies of the pipeline for use by not but not limited to the generation of parallel control, data, and valid pipelines. (CLAIM) Signals which contain many fields or group of signals which are pipelined can be connected to each input and output of the pipe stages in the pipeline automatically and that the signals are automatically named. (CLAIM) A that hierarchical identifiers can be used to access parts of signals in pipelines which contain many fields at any point in the pipeline. (CLAIM) A high level specification language can be used to specify the prefixes and and suffixes for automatically generated pipe stage signals. One example of automatic suffix generation from a high level language specification is the creation of suffixes which have a pipe stage number for the automatically generated signals in each pipe stage. (CLAIM) The high level specification language pipeline specification can be checked for correctness and can be connected to other high level specification language elements such as but not limited to registers, register files, interconnect, and FIFO's. (CLAIM) A high level language pipeline specification have an option to generate clock gated micro pipelines which turn on the down stream pipe stages one cycle in advance to avoid races between clock and data and that there is a two/three cycle “window” of pipe stages in the pipeline and that the micro clock gating reduces the power consumption on the chip. Stages which do not have valid data as indicated by a a valid bit associated with data in each pipe stage are shut off. Stages downstream from the current valid pipe stage which do are off are turned on one cycle in advance in order to avoid the race between clock and data.

**CSL FIFO Specification**

(CLAIM) A CSL FIFO is used to specify FIFO's. CSL FIFO specifications are converted to RTL and alternate modeling languages such as C/C++. (CLAIM) A high

level language specification for FIFOs' can include the following features which are compiled and added to the generated models: bypass generation, fifo added to memory map, synchronous design, asynchronous design, low/high water marks, full/empty pointers, push and pop signals, and stall calculation based on time to stop FIFO.

**CSL Register File Specification**

(CLAIM) A CSL Register File is used to specify Register File 's. CSL Register File specifications are converted to RTL and alternate modeling languages such as C/C++. The Register File specification is used to specify the typical register file configuration options such as the number of write and read ports, create connections to registers external to the Register File by in the Register File address space, connect individual registers to the Register File inputs/outputs, and create bypass signals. (CLAIM) A high level language specification for register files can include the following features which are compiled and added to the generated models: bypass generation, external registers, connect individual registers to signals or ports, and rf added to memory map.

CSL Registers

(CLAIM) A high level language spec for registers can include the following features and the feature is added to the generated models: registers with certain features list a few but limited to the list, spec for certain types of registers, fields can be associated with registers, and registers can be connected to connectivity objects.

CSL Clock Tree

All state elements can be connected to a clock in the clock domain without specifying the connection automatically if there is only one clock in the functional unit. Clock domains can be set for certain hierarchies. Gated clocks can be specified Different types of clock gates can be specified. Clock trees can be built and simulated in RRTL and C++. Clock trees and user/generated logic can be checked for races. Delay lines and wave shapers can be specified.

**CSLC object model generators**

(CLAIM) A CSLC contains object model generators which are used to convert concise language specifications into fully elaborated models. The generators include the auto router and the auto mapper. (CLAIM) The high level language can specify a point to point connection between two or more end points in two or more units separated by one or more units and that an auto router can automatically create the connections between the two or more end points in the intermediate units. The auto router “routes” point to point connections in different hierarchies. Intermediate ports and formal to actual mappings between the connection end points are created. The user can mark state elements such as registers, register file, and memories in units to be included in the memory map. The auto mapper elaborates the memory map specification by visiting each unit in the hierarchy and adding elements in the units which have been marked as belonging to the memory map. The type of the memory map (flat, hierarchical, or virtual) determines the addressing scheme that the auto mapper uses to create the memory map. (CLAIM) A high level language can specify that there are memory elements in a unit such as but not limited to registers, register files, memories, and FIFOs and that each memory element that is to be included in the memory map is marked and optionally has associated memory map attributes added to the memory element and that the auto mapper walks through the memory map specification which can be in text or converted to symbolic form and that auto mapper generates the memory map in the object model of the compiler. (CLAIM) An auto mapper can automatically create the memory map from a high level language specification.

**Cross functional relationships**

CSL types have relationships that create a system for specifying chip designs that has not been previously available in one language and compiler. A high level language can specify the relationships between different design components. (CLAIM) The cross functional relationships can be checked by the high level language compiler. The following is a list of some of the cross functional relationships in chip designs:

signal and field

field and enumerated type

field and decoder generation

interconnect_connectivity_objects and vector specification
fifo and state data
memory and state data
register_file and state data
vector and test bench
state data and test bench
memory map and interconnect
memory map and registers
registers and field
registers and instruction set architecture
registers and interconnect
instruction set architecture and decoder generation
pipeline and interconnect
pipeline and register file
pipeline and fifo

**CSLC Specification Checkers**

(CLAIM) The CSL compiler includes specification checkers which verify that the CSL specification are correct. It is not enough to check individual cross functional specifications since the individual cross functional specifications affect each other. Checks between the individual cross functional specifications must be made.

- interconnect-checks for correct connections between modules based on type, direction, and width
- pipeline-checks for data leaks, stalls, and resource deadlocks
- memory map-checks for duplicate addresses,...

**CSL Cycle Based Functional Simulator specification**

(CLAIM) A that compiling a high level language specification and generating the skeleton for a C++ cycle based simulator in conjunction with a set of C++ cycle based library files and a set of user generated functional logic comprise a maintainable golden simulator that generates vectors and state data to be used in the verification of designs under test in automatically generated test benches.

Furthermore, the C++ simulator can be used to verify diagnostic tests (write, read, compare) which then can be run using a test bench in a stand alone mode to verify the RTL design under test. (CLAIM) The generated C++ simulation skeleton contains a method for including user generated code including but not limited to an include file mechanism where inline include statements include users defined code in the included files. (CLAIM) The generated C++ simulation skeleton contains an API which the user can “write to” to connect the user generated functions with the compiler generated functions and library functions. (CLAIM) The generated C++ simulation skeleton instantiate memories specified with the high level language and that there are generated methods for writing and reading the memories in a single transaction mode and a “side door” method for writing and reading parts of or the entire memory in a parallel mode.

(CLAIM) A high level language specification can describe vectors that are created by the C++ simulator and the compiler can generate the C++ vector writers to generate the transaction level vectors which are then used to stimulate the design under test and also to compare against the design under test outputs. (CLAIM) The vector writers can generate either cycle accurate vectors or transaction accurate vectors or pipes tage and transaction (models the latency through the pipeline but the actual logic is not in the exact same pipe stage in the C++ model as the RTL design under test) accurate level vectors. (CLAIM) The high level language specification can describe vectors that are read by the C++ simulator and applied to chip or unit level interfaces in the C++ simulator and that the compiler can generate the C++ vector readers to generate the transaction level vectors which are then used to stimulate the C++ simulator and also to compare against the C++ simulator chip and unit level outputs. (CLAIM) The vector readers can read either cycle accurate vectors or transaction accurate vectors or pipe stage and transaction (models the latency through the pipeline but the actual logic is not in the exact same pipe stage in the C++ model as the RTL design under test) which are then used to stimulate the C++ simulator and also to compare against the C++ simulator chip and unit level outputs. (CLAIM) A high level language specification can describe state data in the the C++ simulator and the compiler can generate the C++ state data readers to load the state data with initial state and generate the test bench

support code to load the RTL design under test memories. (CLAIM) A high level language specification can describe instruction memories and other storage units in the chip which contain executable instructions in the the C++ simulator and RTL design under test and the compiler can generate the C++ state data readers to load the state data with instructions in numerical format including but not limited to binary and generate the test bench support code to load the RTL design under test memories.

(CLAIM) The generated C++ code can be compiled by a standard C++ compiler to create an executable which which when executed will simulate the chip using cycle based simulation. CLAIM) The cycle based simulation will generate vectors which can be used to stimulate and test the RTL design under test. (CLAIM) A C++ cycle based simulator can use but is not limited to a single threaded mode of execution or a parallel multi threaded mode of execution or a or a parallel processor mode of execution.

(CLAIM) The high level language interconnect, state data, and test bench specifications can be compiled and the compiler can generate the C++ code for vector and state data writers which use a remote method including but not limited to a socket or other remote procedure call to connect directly to the test bench containing the design under test between threads on in a system or over a network or other communication mechanism. (CLAIM) The high level language interconnect specification for the chip design can be compiled and the compiler can generate C++ code which uses a remote method including but not limited to a socket or other remote procedure call to connect one part of the simulator to another part of the C++ simulator between threads on in a system or over a network or other communication mechanism.

**Memory map**

A memory map can use one of the following address schemes: flat, hierarchical, or virtual. (CLAIM) Memory map test generators can be built automatically to test the entire address range using write, read, and compare tests. (CLAIM) The same memory map test(s) can be loaded using a side door method into the C++ simulator

or the design under test. (CLAIM) The same memory map test(s) can be loaded using vectors written into interface of the C++ simulator or the design under test. Andrei add the rest of the memory map specification methods here to this section.

## CSL Verification Specifications

Functional units may be verified by modeling the external drivers and receivers of the functional units interface (input and output ports) using alternate models built in different languages. The stimulus and expect vectors, and state data generated by the alternate models are used to verify the functional units. (CLAIM) The software which performs the following tasks is generated by the CSLC from a CSL specification

- Vectors and state data are captured in the alternate model and sent to a file or connected to the functional unit test bench via inter-process communications.
- The file format is understood by the compiler via the vector and state data specification. A file reader is created in the test bench and the file is read into a memory in the test bench, which is the correct size to hold the vector
- A set of signals to control reading the vector/state data memory are created
- A set of signals to connect the stimulus memory to the functional unit under test are inserted into the test bench.
- A set of signals to connect the expect memory to the output comparators are inserted into the test bench.

The **(**automatic**)** connection of the vectors/state data generated by the alternate model to the functional unit in the test bench is fully automated. The CSL interconnect, vector, state data, and test bench specification are all defined in one language. The inconsistencies introduced by specifying these different components in different languages and using different tools to generate the RTL and alternate model code are avoided.

Moreover, the CSLC can check the CSL interconnect, vector, state data, and test bench specification in one tool and find inconsistencies between the various specifications. Moreover, since the CSL interconnect, vector, state data, and test

bench specifications have been read into one object model in a single software program, the CSL syntax supports reading objects such as the entire set of output signals from a functional unit using one statement. Using an aggregate connectivity structure which refers to a group of objects facilitates changes in the sub objects (name and width changes, additions and deletions) without having to change the upper level interface name which is used to specify that the output signals constitute a vector. The aggregate connectivity structure is routed from one unit to another unit. The underlying signals can be modified without affecting the routing since the aggregate connectivity structure is auto routed from end point to end point in the object model.

Verification of functional units on chips and of chips requires the development of vector and state data (two dimensional vectors from two dimensional memory elements) from a golden reference. Both stimulus vectors and expected vectors/state data are required in order to test the the design under test. The order of the signal values and the widths of the signals, the size of the memories need to be the same in the golden reference model (the functional simulator), the communication mechanism (file/socket), the mechanism in the test bench which applies the vectors to the DUT, loads the binary instruction files into the DUT instruction memories if present, and compares the outputs of the DUT with the expected vectors.

(CLAIM) A high level language can be used to specify the order of the signals and any part selects for individual signals, the event capture mechanism for the vector (cycle accurate or transaction accurate), the vector ID, the name of the unit(s) that the signals are driver by, and the event which indicates that a new vector value should be captured from the group of signals in the vector. Note that since the interconnect specification is included in the same specification the widths of the signals in the vector are known to the compiler,

**CSL Memory**

(CLAIM) A high level specification language can specify the dimensions of memories that are used in the chip design. (CLAIM) A high level specification language can specify the relationship between the width of the ISA instruction word and the

memory and then specify the width of the memory. (CLAIM) A high level specification language can specify the relationship between a memory and a state data element and that the relationship between the state data element and the memory can be used to connect the state data writer to the memory. (CLAIM) A high level specification language can specify the relationship between binary files and the memory and that the memory load routine code can be automatically generated in different languages. (CLAIM) A high level specification language can specify the FPGA, memory compiler, or hand generated memories to use in the implementation model and that the equivalent memory will be generated in the function and structural models.

**CSL producer consumer**

Multi media chips move large amounts of pixels in matrices around chps and process the pixels. A producer unit will write the results of its operations to a memory and the consumer will read the memory. The data access patterns of the producer and consumer are fixed for a periodic operation. The producer write pattern may be different from the consumer read pattern. In other words the write and read orders for the producer and consumer respectively may be different. Programmable engines such as but not limited to counters or micro controllers can be used to write and read the memory. (CLAIM) A high level specification language can specify the write and read operation sequences for the memory and that the cslc compiler can generate the programmable or hard wired controllers which will generate the write and read address sequences to access the memory.

**CSL Buses**

Buses have a protocol and buses have a packet format. (CLAIM) A high level language can specify the packet format and that the packet format is made up of fields and which cycle the parts of the packet are assembled and the signals that make up the bus which carry the bits of the packet and that logic which implements the packet framing and de-framing logic can be automatically generated. (CLAIM) A high level language can specify the packet format of the bus and the logic which implements the packet framing and de-framing logic can be automatically generated. (CLAIM) A high level language can specify the protocol of the bus and

that the protocol checkers can be automatically generated.(CLAIM) A high level language can specify the protocol of the bus and that the protocol hand shaking logic in the sender and the receiver can be automatically generated. (CLAIM) A high level language can specify the bus connections between endpoints using point to point connections to and from bus transmit and receive modules. An on chip bus uses a specific protocol to facilitate communication between different units on a chip. The units on a chip may be functional units which can be reused on different designs with different buses. Instead of redesigning the bus interface for each unit on the chip a standard bus interface is designed and each unit writes and reads from that bus interface in response to commands issued by the bus masters or if the unit is a bus master the unit issues bus writes or reads through the standard bus interface. The standard bus interface connects to a bus specific interface which in turn implements a specific bus protocol. A functional unit block (FUB) is connected to a bus

FUB<-> standard bus interface <-> bus specific interface <-> bus
FUB<-> standard bus interface <-> bus specific interface <-> bus

Changes are not required to the standard bus interface which implements bi directional flow control and input and output buffers. The standard bus interface has a bus master and a bus slave implementation. The standard bus interface allows the reuse of the functional units without redesigning the functional unit's interface. The design team then can concentrate on implementing new bus SOC designs such as OCP, AMBA,... and custom bus designs to connect the implemented IP blocks.

(CLAIM) A high level language can specify the connections between an industry standard SOC bus or any other on chip bus and a bus protocol translation unit with input and output queues and protocol conversion logic to convert to/from the chip bus protocol and the hardware abstraction layer connected to the local unit. Moreover, the units bus interface hardware abstraction layer is designed to connect without modification to bus HAL units that connect to specific bus implementations without modification to the .

**Automatic interconnect generation via high level design language description and an auto router**

Modern designs have connections between connectivity objects (signals, port, signal groups, and interfaces) and containers (FIFO, Register Files, memories) and registers and functional logic. Connections between these objects can be made manually or using point to point connections and an auto router. A point to point connection is a set of end points in functional unit (ports, pins on combinational and state elements,...) that are connected to other end points. The rules for making the connections are defined in academic books. Driver contention is to be avoided. For example, a simple wire must have a one driver and one or more receivers. The connections between the end point can traverse through the design hierarchy (e.g. in Verilog RTL the design hierarchy is created with modules, module interfaces which contain ports, and instantiations of the module instances with formal to actual mappings connecting upper level signals and ports to the module instance's ports). Intermediate signals in modules may only be connected to other ports and signals and not to logic. The end points are connected to logic elements and state elements (flip flips, registers, memories,...) The connections between the end points can vary depending on the architecture and design/implementation choices made by the architecture and design teams. Limitations in in the number of metal layers between block boundaries (referred to as “metal limited” in designer jargon) some times is discovered late in the design process and requires changes in the RTL design hierarchy and RTL module interfaces. Changes in the RTL design hierarchy ripple through to the test benches and simulator interfaces and vectors produced by the simulators for verification of the design under test.

Making hierarchy and interface changes late in the design cycle (project) can be very expensive and delay the development of the project. Automating the design hierarchy and interconnect generation process reduces the impact of design changes by facilitating rapid changes in the design hierarchy and interconnect without affecting the underlying functional logic of the design. Automation of the design hierarchy and interconnect generation process can be achieved by using a high level language to describe the point to point connections on the chip and the design hierarchy. The point to point connections on the chip are routed from leaf

module to leaf module through the design hierarchy using a automatic routing tool to create the intermediate connections in the modules between the two endpoints.

(CLAIM) A high level design language can specify a point to point connect between two or more objects and that a compiler can use the width detection via get_width() functions to determine the width of one or more connectivity object(s) in the point to point connect and the compiler can use the width information from the connectivity object(s) to set the widths of the connectivity objects in the point to point connect that have unknown widths, check the widths of all known widths, and to set the widths of the intermediate signals which are created automatically between the end points of the point to point connection. (CLAIM) A high level design language compiler can check for width mismatches between signals and other objects such as registers. (CLAIM) A high level design language compiler can check field linked lists and trees for overlaps between adjacent fields in connectivity objects such as but not limited to signals using linked lists of fields and relative positions. (CLAIM) A high level design language compiler can detect width mismatches between a group of fields and connectivity objects that the fields are associated with. (CLAIM) A high level design language compiler can detect width mismatches between a field and the fields associated enumerated type (an enumerated type has a specified number of elements and the number of elements can be represented by a binary number which must be equal to or less than the field width).

**CSL and assertions**

Assertions can be specified in a high level language specification and generated in RTL and C++ code to detect assertion failures and assertion coverage during simulation.

**CSL and user generated code.**

(CLAIM) CSL generated code and user generated code can be integrated to form one executable via include statements and other methods and that the two sets of code can be compiled by but not limited to a standard compiler such as gcc or an RTL compiler such as Cadence ncverilog.

**CSL and documentation generation**

(CLAIM) A high level language specification can be annotated with different types of descriptions and abreviations for each object type describe by the high level language and that the different text annotations can be added to the generated documentation that describes the design. Cslc generates assembly language documentation from the high level language ISA specification. Cslc generates interconnect interface documentation on a module and per instance basis from the high level language interconnect specification. Cslc generates memory map documentation the high level language memory map specification. Cslc can generate documentation from any of the CSL specification types.

**CSL Network of processors**

(CLAIM) A high level language plus additional libraries can be used to specify a network topology which is used to send messages between processors using a network such as but not limited to a token ring, bus, mesh, tree, and star. (CLAIM) A high level language can specify the custom instructions used to send messages between processors using a network such as but not limited to a token ring, bus, mesh, tree, and star and that the user can add the additional logic to implement the functionality of the custom send and receive instructions. (CLAIM) A high level language specification for custom instructions used to send messages between processors using a network such as but not limited to a token ring, bus, mesh, tree, and star can be used to create software tools which will compile the specification and then create a software tool to be used to compile the instructions into binaries that are then loaded into each processors instruction memory. The software tool will check the communication between the instructions and generate warnings and errors about communication problems in the user written instruction programs. (CLAIM) A high level language can specify the buffers used by each processor to store incoming and outgoing messages sent between processors using a network such as but not limited to a token ring, bus, mesh, tree, and star. (CLAIM) A high level language memory map specification can be used to create the address map used by each processor address the individual processors using a network such as but not limited to a token ring, bus, mesh, tree, and star. (CLAIM) A high level language memory map specification processor address map can be combined with other high level methods to create hardware to detect writes, reads and other

operations sent to the individual processors using a network such as but not limited to a token ring, bus, mesh, tree, and star and that the hardware can write or read the memory map elements in the processor. (CLAIM) A high level language specification of the network such applies to networks on chip, networks between chips and other networks that can be specified using the high level language.

**CSL Network of functional units**

(CLAIM) A high level language plus additional libraries can be used to specify a network topology which is used to send messages between functional units using a network such as but not limited to a token ring, bus, mesh, tree, and star. (CLAIM) A high level language can specify the custom instructions used to send messages between functional units using a network such as but not limited to a token ring, bus, mesh, tree, and star and that the user can add the additional logic to implement the functionality of the custom send and receive instructions. (CLAIM) A high level language can specify the buffers used by each functional unit to store incoming and outgoing messages sent between functional units using a network such as but not limited to a token ring, bus, mesh, tree, and star. (CLAIM) A high level language memory map specification can be used to create the address map used by each functional unit address the individual functional units using a network such as but not limited to a token ring, bus, mesh, tree, and star. (CLAIM) A high level language memory map specification functional unit address map can be combined with other high level methods to create hardware to detect writes, reads and other operations sent to the individual functional units using a network such as but not limited to a token ring, bus, mesh, tree, and star and that the hardware can write or read the memory map elements in the functional unit. (CLAIM) A high level language specification of the network such applies to networks on chip, networks between chips and other networks that can be specified using the high level language.

**CSL Micro Engine Example**

We will now describe an example design using CSL. Note that this is just an example of how CSL can be used to design a chip. A network of application specific processors (micro engines) on a chip is a challenging design to implement. The

application specific processors have general purpose instruction load/store/branch/logical/arithmetic instructions and additional specialized instructions that use an optimized data path to increase the effective computation power of the processor per compute cycle. A network of application specific processors combined with firmware can replace custom ASIC and VLSI pipelines. Applications which are not latency sensitive are ideal candidates for networks of application specific processors. Less engineering time and resources are required to build one application specific processor design, then instantiate it many times and write the associated firmware/microcode than it is to build a custom hardware pipeline. CSL can be used to implement the application specific processors pipeline, ISA, register file(s), interconnect, and memory map. The result is a skeleton of the design which is missing the functional unit logic. The generated skeleton includes the pipeline inference hardware, the connections between the pipelines, the instruction decoder, the extracted bit ranges from the pipelined instruction register, The designer then implements the functional unit logic. Functional unit logic include arithmetic, logical, load/store, and branch units. Software tools such as assemblers and compilers are automatically generated by the high level language compiler from the the ISA specification .

It is understood that the disclosed systems and methods are compatible with other configurations of ...

and structures without deviating from the spirit of the present specification.

What is claimed is:

**Prior art**

Tensilica/ARC patents on their extensible processors (the extend the functionality of their base processor design)

Denali RDL

Vast Systems

Arteris

Sonics Inc.

Coware

Bluespec