A COMPILER FOR ELABORATING HARDWARE DESCRIPTION LANGUAGE SPECIFICATIONS OF ASYNCHRONOUS DIGITAL-LOGIC SYSTEMS

1. Introduction

1.1. Field of the Invention

[0001] This invention relates to the computer-aided design of asynchronous digital-logic systems.

1.2. Background of the Invention

[0002] Today's electronic chips are largely implemented as synchronous digital-logic systems. A synchronous digital-logic system relies on a global clock that synchronizes the transfer of data between data registers. An asynchronous digital-logic system, on the other hand, does not rely on a global clock for data transfer. Instead, data transfers are governed by handshake circuits that communicate locally with each other through a handshake protocol. Data transfers are carried out using channels, as depicted in FIG. 1. A channel consists of handshake signals (*Request* and *Acknowledge*) between handshake circuits and datapaths between the data registers. *Request* is output by the sender to the receiver to indicate the availability of new data, whereas *Acknowledge* is output by the receiver to the sender to indicate that current data has been consumed and new data may be released.

[0003] A typical design flow for digital-logic systems consists of four major steps: (i) specification of design in a high-level language; (ii) verification of design through simulations; (iii) compilation of design into an intermediate model (for example, a gate-level netlist); and (iv) placement and routing of design. The present invention concerns the third step – design compilation, i.e., the elaboration of a high-level design specification into an intermediate model.

[0004] Many design compilers for asynchronous digital-logic systems have been reported. These design compilers can be broadly classified based on the type of architecture that they support into two categories – controller- or modular-based.

[0005] Controller-based design compilers, with a few exceptions, focus mainly on the design of asynchronous controllers and do not encompass the design of datapaths. Several reported controller-based design compilers are graph-based: US Patent 6,594,815 Lee et al.; US Patent 5,748,487 Sawasaki et al.; US Patent 6,314,553 Stevens et al.; Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications," PhD dissertation, Massachusetts Inst. Technol., Cambridge, 1987 (Chu 1987); Meng et al., "Automatic synthesis of asynchronous circuits from high-level specifications," *IEEE Trans. Computer-Aided Design*, vol. 8, no. 11, pp. 1185-1205, Nov. 1989 (Meng 1989); Cortadella et al., "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. Inform. Syst.*, vol. E80-D, no. 3, pp. 315-325, 1997 (Cortadella 1997); and Sokolov et al., "STG optimization in the direct mapping of asynchronous circuits," in *Proc. Eur. Design, Automation and Test*, pp. 932-937, Mar. 2003 (Sokolov 2003). These design compilers accept graph-based design specifications, usually in the form of Signal Transition Graphs (STG's), and synthesize them into hazard-free asynchronous circuit implementations.

[0006] Another group of controller-based design compilers generate Asynchronous Finite State Machines (AFSM's). AFSM's are similar to their synchronous counterparts but do not rely on global clocks for state transition. Instead, state transition in an AFSM is triggered by changes in its input signals. Reported design compilers under this group that allow multiple-input changes and generate hazard-free logic implementations include: Norwick, "Automatic synthesis of burst-mode asynchronous controllers," PhD dissertation, Stanford Univ., Dept. Comput. Sci., Stanford, CA, 1993 (Norwick 1993); and Yun, "Automatic synthesis of extended burst-mode circuits: part I," *IEEE Trans. Computer-Aided Design*, vol. 18, no. 2, pp. 101-117, 1999 (Yun 1999).

[0007] The controller-based design compilers mentioned thus far accept design specifications in the form of graphs or state diagrams. For large circuits, this design style usually becomes error prone and unnecessarily complex. Yet another group of controller-based design compilers attempts to simplify the design specification of asynchronous controllers through the use of high-level Hardware Description Languages (HDL's). Controller-based design compilers that support either Verilog HDL or VHDL have been reported by Blunno and

Lavagno, “Automated synthesis of micropipelines from behavioral Verilog HDL,” in *Proc. Advanced Research in Asynchronous Circuits and Systems*, pp. 84-92, Apr. 2000 (Blunno 2000); Jacobson et al., “High-level asynchronous system design using the ACK framework,” in *Proc. Advanced Research in Asynchronous Circuits and Systems*, pp. 93-103, Apr. 2000 (Jacobson 2000); and Morizawa and Nanya, “Specification style of four-phase handshaking asynchronous controllers and its return-to-zero phase optimization,” *IEICE Trans. Fundamentals Electron., Commun. and Comput. Sci.*, vol. E83-A, no. 12, pp. 2446-2455, Dec. 2000 (Morizawa 2000); and Vanbekbergen et al., “A design and validation system for asynchronous circuits,” in *Proc. Design Automation Conf.*, pp. 725-730, June 1995 (Vanbekbergen 1995).

[0008] There are several problems associated with controller-based design compilers, particularly for designing at the system-level. First, it is well recognized that the compilation process is inherently of high-complexity and has an execution time that tends to increase exponentially with circuit size and complexity, a phenomenon usually referred to as the state-explosion problem. For graph-based design compilations, a fundamental challenge involves uniquely encoding every reachable state of the controller and there usually arises a need to insert additional internal signals in order to resolve state-encoding contentions, which in turn leads to a more complex circuit. For system-level designs, the execution time can therefore become prohibitively long. This is a major disadvantage in a typical commercial design environment where time-to-market of product is often a top-priority goal. Second, for AFSM’s, the removal of gate-level hazards limits the logic style to a two-level logic implementation which restricts the space for logic optimization. For large controllers with complex logic, this restriction may not be acceptable.

[0009] Modular-based design compilers implement asynchronous designs as interconnections of discrete handshake, logical and arithmetic modules that communicate with each other using channels. Typically, the designer uses a high-level language that is based on the concept of Communicating Sequential Processes (CSP), whose key characteristics include the support of concurrent processes, sequential and concurrent composition of statements within a process, and synchronized message passing over channels. See Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985. Syntax-directed translation

is utilized to convert the program into hardware, with each type of language construct replaced by a corresponding handshake module.

[0010] CSP and syntax-directed translation have been the basis for several modular-based design compilers: US Patent 5,005,136 Van Berkel et al.; US Patent 6,021,266 Kay; Brunvand, "Designing self-timed systems using concurrent programs," *J. VLSI Signal Processing*, vol. 7, no. 1-2, pp. 47-59, 1994 (Brunvand 1994); Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," *Developments in Concurrency and Communications*, C. A. R. Hoare, Ed. UT Year of Programming Series, Addison-Wesley, pp. 1-64, 1990 (Martin 1990); Akella and Gopalakrishnan, "SHILPA: a high-level synthesis system for self-timed circuits," in *Proc. Int. Conf. Computer-Aided Design*, pp. 587-591, Nov. 1992 (Akella 1992); Bardsley, "Implementing Balsa Handshake Circuits," PhD dissertation, Univ. Manchester, 2000 (Bardsley 2000); and Barringer et al., "Rainbow: development, simulation and analysis of asynchronous micropipeline hardware design," *Comput. J.*, vol. 45, no. 1, pp. 2-11, 2002 (Barringer 2002).

[0011] These modular-based design compilers all support design specifications written in high-level programming languages. However, it is well recognized that the generality of a high-level programming approach to hardware design inherently implies that the translated design is likely to be highly un-optimized due to the significant amount of redundant functionalities (in the form of channels and handshake circuits) built into the circuit. This problem is especially serious for designs that are control-dominated. The redundancies often stem in part from the program 'idioms' at the source code level. Large area overheads due to the handshake circuits can therefore be expected. Moreover, the switching of signals on the large number of handshake lines would undermine the typical power advantages made possible by the asynchronous design style.

[0012] In summary, it is apparent that the reported design compilers for asynchronous digital-logic systems are unsatisfactory due to either prohibitive execution time, restrictive logic implementation style or significant redundancies in circuit implementations.

1.3. Summary of the Invention

[0013] The present invention provides a compiler for elaborating HDL specifications of asynchronous digital-logic systems into equivalent modular Register-Transfer-Level (RTL) models that are suitable for HDL simulations and logic synthesis.

[0014] The compiler operates in three steps: analysis of specification, construction of internal representations and generation of RTL model.

[0015] The first step of the compilation involves reading and analyzing specifications that are written in standard Verilog HDL (see IEEE Standard 1364-2001, IEEE Standard Verilog® Hardware Description Language, IEEE Computer Society, IEEE) and are augmented by a set of pragmas (i.e., compiler directives) provided for by the present invention.

[0016] The second step of the compilation involves constructing internal representations of the specifications using directed graphs by extracting the causal relations between the entities declared in the specifications, establishing the corresponding communication channels and determining the nature (such as push or pull, control or data, etc.) of the established channels.

[0017] The third step of the compilation involves generating equivalent modular RTL models of the specifications and comprises the following sub-steps: (i) inferring handshake modules from the internal representations; (ii) expanding the established channels into explicit handshake signals and bundled data; (iii) instantiating the inferred handshake modules; (iv) interconnecting the instantiated handshake modules based on the established channels; and (v) establishing the control of the instantiated handshake modules over their respective datapath components.

1.4. Brief Description of the Drawings

[0018] FIG. 1 depicts an asynchronous communication channel.

[0019] FIG. 2 depicts a handshake protocol.

[0020] FIG. 3 is the symbol of a `Sync` module.

[0021] FIG. 4 is the symbol of a `SyncPassiveOut` module.

[0022] FIG. 5 is the symbol of a `SyncActiveIn` module.

[0023] FIG. 6 is the symbol of a `SyncGuard` module.

[0024] FIG. 7 is the symbol of a `SyncGuardActiveIn` module.

[0025] FIG. 8 is the symbol of a `ChannelMux` module.

[0026] FIG. 9 is the symbol of a `ChannelMuxArb` module.

[0027] FIG. 10 depicts the implementation of behavioral loops using a `Loop` module.

[0028] FIG. 11 depicts the causal relations extracted from the continuous assignment `assign a=b+c`.

[0029] FIG. 12 depicts the causal relations extracted from Example 4.

[0030] FIG. 13 depicts the causal relations extracted from Example 5.

[0031] FIG. 14 depicts the communication channels extracted from Example 5.

2. Detailed Description of the Invention

2.1. Inferring Asynchronous Communication Channels

[0032] Communications between two parties in asynchronous digital-logic systems take place over channels. A channel comprises a request line, an acknowledge line and the (optional) corresponding data. Data transfers over a channel are governed by handshake protocols. As depicted in FIG. 2, a handshake protocol consists of, in essence, a request that is forwarded by the sender to the receiver to indicate the availability of a new piece of data, followed by an acknowledgement from the receiver to the sender indicating the consumption of that piece of data.

[0033] These protocols, and the channels over which they occur, are such fundamental features in any asynchronous digital-logic system that they must be represented by any high-level languages that are developed for, or adapted to, asynchronous design. These features are of course available in CSP-based programming languages but are understandably lacking in HDL's. Although it is certainly possible to describe channel activities through explicit handshake signals in HDL's, such a low-level modeling technique is unlikely to be

acceptable to designers since it would represent a tremendous overhead in design time and effort when compared to the design style of synchronous circuits. Solutions suggested in the literature all involve the use of specially-developed packages and subroutines that provide abstract models for channels and communication actions (such as subroutines for sending and receiving data), either in the same or in a different language, such as C, through some form of language interface. Although these abstraction models ease to a certain extent the difficulties in describing channel activities using HDL's, the resulting design style still leaves much to be desired since the subroutines must be called in the code whenever and wherever channel communication is required.

[0034] In the present invention, communication channels and actions are totally transparent to the designer, i.e. they do not have to be explicitly described in the specification of a design, either through low-level handshake signals or abstract channel models. This frees the designer to concentrate fully on modeling the behavior of the design at hand. In the present invention, various types of communication channels and actions are implicitly inferred by HDL constructs, augmented by a set of pragmas. For example, a one-place buffer can be specified as shown in Example 1, in which no explicit interaction of handshake signals or abstract models for channel communications are present. Based on the notion that data transfer can only take place over channels, the specification implicitly infers two channels – one from `in` to `buffer`, inferred by the non-blocking procedural assignment "`buffer<=in`" at line 2; the other from `buffer` to `out`, inferred by the continuous assignment "`assign out=buffer`" at line 3. During design compilation, these implicit channels are elaborated into explicit handshake signals which, together with the inferred handshake modules (in this example only one is inferred for `buffer`), allow the specification to be simulated and synthesized.

Example 1:

```
1     always @(in)
2        buffer<=in;
3     assign out=buffer;
```

It is important to recognize the fundamental difference between our interpretation of the codes in Example 1 and its original semantics. The latter simply specifies that `in` be assigned to `buffer` whenever there is a change in the value of `in`. To support channel-based communications, the assignment "`buffer<=in`" has been overloaded with additional conditions such that it only executes when (i) `in` has made a request to transfer its data to `buffer` and (ii) the data held by `buffer`, if any, has been consumed by its direct successors. In other words, operations are activated by specific events, i.e. they are event-driven. Other (more sophisticated) inference styles are described later.

2.2. Inferring Handshake Modules

[0035] Handshake modules are inferred from design specifications to provide communication and synchronization both within the design, and between the design and its environment. They provide control over almost all sequential logic elements and, to a smaller extent, combinational logic elements in the design. Handshake modules are inferred based on specific HDL constructs. Three classes of handshake modules are defined in the present invention to support the embodied design style: *synchronization*, *channel multiplexing* and *looping*.

2.2.1. Synchronization Class

[0036] All handshake modules in this class provide synchronizations within the design but differ in the specific channel types that they serve. They are: `Sync`, `SyncPassiveOut`, `SyncActiveIn` and `SyncGuard{,ActiveIn}`.

[0037] `Sync` performs fundamental handshake communications within an asynchronous system. Specifically, `Sync` joins *passive-in* channels, forks to *active-out* channels, and controls one or more storage device. Passive-in and active-out channels are input and output channels, respectively, that are activated by their respective sources, i.e. they are *push* channels. FIG. 3 depicts the symbol of a one-input, one-output `Sync`. The passive-in handshake ports consist of the input request *ri* and acknowledge *ai*. The active-out handshake

ports consist of the output request *ro* and acknowledge *ao*. The latch-enable output *lb* controls the slave storage devices. A `Sync` with multiple input channels joins the multiple input requests from its predecessors into a single signal and forks its input acknowledge to its predecessors. Likewise, a `Sync` with multiple output channels forks its output request to its successors and joins the output acknowledges from its successors into a single signal. A `Sync` is inferred for a variable that is assigned a value in a non-blocking procedural assignment if it has passive-in and active-out channels, or a named event that is triggered by passive-in channels. In Example 1, the variable `buffer` is assigned a value in a non-blocking assignment (line 2) and has a passive-in channel `in` and an active-out channel `out`. As a result, a `Sync` is inferred for `buffer`.

[0038] `SyncPassiveOut` is a variant of `Sync` that has *passive-out* channels, i.e. the output channels are *pull* channels on which a data transfer is initiated by the channel destination. FIG. 4 depicts the symbol of a `SyncPassiveOut` that has one passive-in and one passive-out channel. `SyncPassiveOut` can be considered as a memory-like element, and its input and output channels as write and read ports, respectively. Since the events on these ports are not necessarily mutually-exclusive, `SyncPassiveOut` contains an arbiter to arbitrate between concurrent requests on them. `SyncPassiveOut` is inferred for a variable that is assigned a value in a non-blocking procedural assignment if it has passive-in and passive-out channels. A variable with passive-out channels is specified by attaching to it the `passive_out` attribute during its declaration.

[0039] `SyncActiveIn` is a variant of `Sync` that has *active-in* channels. FIG. 5 depicts the symbol of a `SyncActiveIn` that has one passive-in, one active-in and one active-out channel. `SyncActiveIn` initiates a data transfer on its active-in channels *after* it has received requests on its passive-in channels. As such, variables with active-in channels must have at least one passive-in channel. `SyncActiveIn` is inferred for a variable that is assigned a value in a non-blocking procedural assignment if it has at least one active-in channel. An input channel of a variable is inferred to be active if it is the destination of a pull channel, i.e. the variable has as one of its sources a variable with passive-out channels or an input port declared to be active-in.

[0040] `SyncGuard` is a variant of `Sync` that has *guarded* output channels. FIG. 6 depicts the symbol of a one passive-in, one active-out `SyncGuard`. Based on the outcome of the evaluation of its *guard expression*, which is sampled by the guard input *g*, `SyncGuard` either carries on or terminates (to avoid unnecessary computations) the data-flow along a pipeline fragment. More specifically, if *g* is '1', i.e. the guard expression is evaluated to be true, `SyncGuard` accepts the new token, enables its storage device to register the new data, completes its input handshake and activates its output channels. Otherwise, it merely completes its input handshake, with the new data not registered and the output channels not activated. `SyncGuard` is inferred for a variable or named event if in the HDL model the variable's assignment to a new value or the triggering of the named event is specified within a conditional construct that has a `guard` attribute attached. The corresponding conditional expression is taken as the guard expression. In Example 2, the variable `y` is assigned to variable `z` in an `if` statement that has the `guard` attribute attached. As a result, a `SyncGuard` with the guard expression "`x==1`" is inferred for `z`.

Example 2:

```
1     always @(x or y)
2          (*async,guard*) if (x==1)
3               z<=y;
```

[0041] `SyncGuardActiveIn` is a variant of `Sync` that combines the features of `SyncGuard` and `SyncActiveIn`. Like the former, it has guarded output channels, and like the latter, it has active-in channels. FIG. 7 depicts the symbol of a `SyncGuardActiveIn` that has one passive-in, one active-in and one active-out channel. The guard expression is evaluated only after the data on the active-in channel is valid and stable.

2.2.2. Channel-Multiplexing Class

[0042] `ChannelMux` merges two *mutually-exclusive* channels into a single channel by passing on events on its passive-in channels to its only active-out channel. As depicted in FIG. 8, *ri0* and *ri1* are the request signals of the first and second input channels, respectively,

and *ai0* and *ai1* are the corresponding acknowledge signals. It also outputs a *data-select* signal *s* which is used as the select input to a data multiplexer to perform datapath multiplexing, selecting the input data whose channel is active. A `ChannelMux` is inferred to merge the input channels of a variable if it is assigned values in non-blocking procedural assignments in *two* `always` blocks. Channels inferred from the first `always` block form the first set of input channels, whereas those inferred from the second block form the second set. Example 3 shows the inference of a `ChannelMux` for the variable `z`, which has two input channels, `x` and `y`, inferred from two `always` blocks.

Example 3:

```
1     always @(x)
2          z<=x;
3     always @(y)
4          z<=y;
```

[0043] `ChannelMuxArb` is a variant of `ChannelMux` that contains an arbiter to arbitrate between its two input channels that are *not* mutually-exclusive. As depicted in FIG. 9, the internalized input channels are mutually-exclusive and can therefore be handled by a `ChannelMux` core.

[0044] `ChannelMuxComb` is a variant of `ChannelMux` but does *not* control any storage device. It is smaller and consumes less power than `ChannelMux` and is therefore preferred to the latter when the multiplexed data in the datapath need not be stored. `ChannelMuxComb` is suitable for merging the input channels of (i) variables that are to be implemented as *nets*, not storage devices, and (ii) named events. To ensure proper dataflow and to avoid loss of data, `ChannelMuxComb` input channel handshaking always encloses output channel handshaking. A `ChannelMuxComb` is inferred for a variable if it has *blocking* assignments '=' in *two* `always` blocks, and for a named event if it is triggered in *two* `always` blocks.

[0045] `ChannelMuxCombArb` is a variant of `ChannelMuxComb` that contains an arbiter to arbitrate between its two input channels that are *not* mutually-exclusive.

2.2.3. Looping Class

[0046] Three types of behavioral looping constructs are supported by the present invention: `for`, `repeat` and `while`. The handshake module `Loop` implements all three types of loops and consists of two sub-blocks: `LoopFront` and `LoopEnd`. As depicted in FIG. 10, loop operation is governed by three channels: *initiation*, *evaluation* and *iteration*. The initiation channel initiates loop operation by passing a new token (either in the form of new data or a triggering event) to `LoopFront`, which then passes the token on to `LoopEnd` via the evaluation channel. The looping condition is evaluated in the loop body to determine whether iteration is required. If so, `LoopEnd` activates the iteration channel and passes the token back to `LoopFront`. Otherwise, it outputs the computed result to the loop's successors and informs `LoopFront` to complete the handshake on the initiation channel. A loop handles only one set of data at any one time and does not accept new data when in operation. This is ensured by enclosing the handshakes on the iteration channel within the initiation channel's handshake. Two sets of registers are inferred for loop variables (defined as those assigned values in a loop). The first set, which can be regarded as *buffers* for the loop variables, is controlled by `LoopFront` and receives as input either the initial values of the loop variables (during initiation), or their iterated values (during iteration). Buffers serve as inputs to the computations in the loop body. The second set, which implements the loop variables themselves, is controlled by `LoopEnd` and receives as input either the results of the computations in the loop body (during iteration) or the values of the buffers (when terminating).

2.4. Pragmas

[0047] A set of *pragmas* (or compiler directives) is defined and incorporated into the present invention to facilitate the inference or specification of certain handshake modules, asynchronous-related features and behaviors, to remove design ambiguities, and to clarify design intentions. In compliance with IEEE Standard 1364.1-2002, IEEE Standard for Verilog Register Transfer Level Synthesis, pragmas are specified through *attributes* and their instances. An attribute instance that specifies one of the pragmas in the present invention is

headed by the keyword `async` and has as its body one or more asynchronous-related attributes, some of which have values assigned to them. The attributes are `reset`, `passive_out`, `active_in`, `bundle`, `guard`, `control`, `sychronous`, `mutually_exclusive`, `channel`, `no_handshake_module` and `no_handshake`.

[0048] The `reset` attribute is attached to an input port to specify that it is an asynchronous reset input for handshake modules.

[0049] The `passive_out` attribute is attached to a `reg` variable to specify that it has passive-out channels.

[0050] The `active_in` attribute is attached to an input port to specify that it is an active-in channel.

[0051] The `bundle` attribute is attached to a module and the value assigned to it is a set of input ports or `reg` variables that are to be bundled together. Bundled entities share common channels and, for storage devices, are controlled by the same handshake module.

[0052] The `guard` attribute is attached to an `if` statement to specify that the variables assigned values within it have guarded output channels.

[0053] The `control` attribute is attached to an input or output port to specify that it is a control channel, i.e., it has no bundled data.

[0054] The `synchronous` attribute is attached to a `reg` variable to specify that it is to be implemented using edge-sensitive storage devices (i.e., flip-flops), instead of level-sensitive ones (i.e., latches).

[0055] The `mutually_excluisve` attribute is attached to a module and the value assigned to it is a set of inputs, outputs or `reg` variables whose corresponding channels are mutually-exclusive to each other, i.e. only one channel is active at any point in time.

[0056] The `channel` attribute is attached to a named event to associate it to an input port or a `reg` variable. This means that the named event represents channel events on the input port or on the output channels of the `reg` variable.

[0057] The `no_handshake_module` is attached to a `reg` variable to specify that no handshake module is to be inferred for it. This implies that the `reg` variable is to be controlled directly by a datapath component.

[0058] The `no_handshake` attribute is attached to a causal relation to specify that it shall not lead to the establishment of a corresponding channel.

2.5. Internal Representations Using Directed Graphs

[0059] Communications within a modular-based asynchronous system are carried out by a network of handshake modules interconnected by channels. This network is represented in the present invention using directed graphs (or digraphs). The digraphs are constructed in two steps. First, a digraph D_1 is constructed that captures all causal relations between the entities declared in the HDL specification. In D_1, the vertices $V(D_1)$ represent all entities declared in the HDL specification, including ports, nets, variables and named events, and the arcs $A(D_1)$ represent the causal relations between the entities. Second, a digraph D_2 that represents all communication channels in the HDL specification is derived from D_1. In D_2, the arcs $A(D_2)$ represent the communication channels between the design's various handshake modules, and vertices $V(D_2)$ represent the module and instance ports, registers, named events and channel multiplexers.

2.5.1. Extracting Causal Relations

[0060] A digraph D_1 is constructed such that the vertices $V(D_1)$ represent the ports, nets, variables and named events declared in the HDL model, and the arcs $A(D_1)$ represent the causal relations between them, i.e. D_1 captures all causal relations between the entities

declared in the HDL specification. A causal relation (*a*,*b*) exists, where *a* and *b* represent either physical data or events, if *b* is *dependent* on *a*. Two types of dependences are defined: (i) *b* is *data*-dependent on *a* if the value of *b* depends on that of *a* either combinationally (i.e., *a* and *b* are the input and output of a gate, respectively) or sequentially (i.e., *a* and *b* are the input and output of a register, respectively), and (ii) *b* is *event*-dependent on *a* if *b* or/and *a* is an event, and the arrival of data *a* (triggering of event *a*) causes the triggering of event *b* (assignment of a new value to *b*). To extract the causal relations, the *assignments* specified in the HDL model, as well as the behavioral constructs (such as conditional and `case` statements) within which the assignments are specified, are analyzed.

[0061] Continuous assignments. Causal relations are established *from* the operands in the expression on the right-hand side of the assignment statement *to* the nets on the left-hand side. For example, the continuous assignment

```
assign a=b+c
```

establishes arc `ba` from vertex `b` to vertex `a`, and arc `ca` from vertex `c` to vertex `a`. The corresponding digraph fragment is shown in FIG. 11.

[0062] Procedural assignments *not* specified within any behavioral construct. Causal relations are identified in exactly the same way as those for continuous assignments.

[0063] Procedural assignments specified within conditional (`if` and `else-if`) statements. Causal relations are established *from* (i) the operands in the expressions on the right-hand side of the assignment statements and (ii) the operands in the conditional expressions, *to* the variables on the left-hand side of the assignment statements. The extraction of the causal relations takes into account the scope of the operands. Example 4 shows an `if-else-if` statement with nested `if` statements. The causal relations are extracted as follows. The scope of `a` covers the entire `if-else-if` statement, thus creating arcs `ax`, `ay` and `az` that are incident from `a` and incident to all variables that are assigned values in the statement (`x`, `y` and `z`). The scope of `b` and `e` each covers only a single assignment (lines 4 and 9, respectively) and therefore only one arc is established from each of these vertices (`bx` and `ey`). The scope of `d` covers the `else-if` and `else` blocks (lines 6-11), giving rise to arcs `dy`

and `dz`. Arcs directly inferred from the procedural assignments are `cx`, `fy` and `gz` (lines 4, 9 and 11, respectively). FIG. 12 depicts the extracted digraph.

Example 4:

```
1     if (a==1)
2          begin
3          if (b==2)
4               x<=c;
5          end
6     else if (d==3)
7          begin
8          if (e==4)
9               y<=f;
10         end
11    else z<=g;
```

[0064] Procedural assignments specified within `case` statements. Causal relations are established *from* the operands in (i) the `case` expression, (ii) the `case` item expressions and (iii) the expressions on the right-hand side of the assignment statements, *to* the variables on the left-hand side of the assignment statements. The scope of an operand in (i) covers the entire `case` statement, that in (ii) is limited to the statements belonging to the same `case` item and those in the following `case` items, and that in (iii) is limited to the statements belonging the to same `case` item.

[0065] Event controls and triggers. Causal relations are established (i) *from* the operands in the conditional expression of a conditional statement *to* the named event triggered within the conditional statement, and (ii) *from* the named event associated *to* an event-control *to* the variables on the left-hand side of the assignment statements controlled by the same event-control.

[0066] The HDL model of a two-stage addition of four numbers is shown in Example 5. To construct the corresponding digraph D_1, the vertex set $V(D_1)$ is populated by extracting from the model all ports, nets and variables. The arc set $A(D_1)$ is determined by parsing the

assignment statements. The continuous assignment "`assign add0=in0+in1`" at line 11 leads to {(`in0`,`add0`),(`in1`,`add0`)}, and "`assign add1=in2+in3`" at line 12 leads to {(`in2`,`add1`),(`in3`,`add1`)}. The procedural assignment "`add0_reg<=add0`" at line 16 leads to (`add0`,`add0_reg`), "`add1_reg<=add1`" at line 17 leads to (`add1`,`add1_reg`), and "`out<=add0_reg+add1_reg`" at line 18 leads to arcs {(`add0_reg`,out),(`add1_reg`,out)}. In summary, $V(D_1)$ = {`in0`,`in1`,`in2`,`in3`,`add0`,`add1`,`add0_reg`,`add1_reg`,`out`} and $A(D_1)$ = {(`in0`,`add0`),(`in1`,`add0`),(`in2`,`add1`),(`in3`,`add1`),(`add0`,`add0_reg`),(`add1`,`add1_reg`),(`add0_reg`,`out`),(`add1_reg`,`out`)}. FIG. 13 depicts the digraph D_1.

Example 5:

```
1     module addition (reset,in0,in1,in2,in3,out);
2
3     (*async,reset*) input reset;
4     input [3:0] in0,in1,in2,in3;
5     output [5:0] out;
6
7     wire [4:0] add0,add1;
8     reg [4:0] add0_reg,add1_reg;
9     reg [5:0] out;
10
11    assign add0=in0+in1;
12    assign add1=in2+in3;
13
14    always @(add0 or add1)
15          begin
16          add0_reg<=add0;
17          add1_reg<=add1;
18          out<=add0_reg+add1_reg;
19          end
20
21    endmodule
```

2.5.2. Establishing Communication Channels

[0067] Together with other information extracted from the original HDL specification, a channel-based digraph D_2 is derived from D_1 such that the arcs $A(D_2)$ represent the communication channels between the design's various components, and vertices $V(D_2)$ represent the module and instance ports, registers, named events and channel multiplexers. Since $V(D_2)$ is a subset of $V(D_1)$ and can be extracted directly from the HDL specification, the construction of D_2 essentially means determining $A(D_2)$ from D_1. The steps for doing so are as follows:

a) $V(D_2)$ is decomposed into two intersecting subsets V_S and V_D, where V_S is the set of all channel sources and V_D the set of all channel destinations. $V_{SD}=V_S \cap V_D$ consists of vertices that are to be implemented as, or controlled by, handshake modules, i.e. registers, named events and data multiplexers. $V_S'=V_S \setminus V_D$ consists of module inputs and instances' outputs, whereas $V_D'=V_D \setminus V_S$ consists of module outputs and instances' inputs.

b) For each channel source $v_s \in V_S$, the corresponding channel destinations $V_{D,vs}$ are determined by traversing all paths leading away from v_s. When a path reaches a vertex $v_d \in V_D$, a channel (v_s,v_d) is formed. To extract all channels, the following algorithm recursively is executed recursively:

```
find_channel_destination(v) {
find from D1 the arcs A1 that are incident from v;
for each arc a1 ∈ A1 {
        find from D1 the vertex v1 to which a1 is incident;
        if v1 ∈ VD
                put v1 into VD,vs;
        else
                find_channel_destination(v1);
        }
}
```

c) For each $v_s \in V_S$, arcs are created that are incident from v_s and incident to each $v_d \in V_{D,vs}$. This completes the generation of D_2.

[0068] Consider Example 5 again. The vertex set of the first digraph D_1 is first decomposed into two subsets V_S and V_D consisting of {in0,in1,in2,in3,add0_reg,add1_reg} and {add0_reg,add1_reg,out}, respectively. The channel destinations for each of the channel source in V_S are then determined by executing the algorithm find_channel_destination(v). For example, consider the case for in0. The algorithm finds from D_1 that an arc joins in0 to add0. However, add0 is *not* in V_D, therefore the algorithm repeats itself on add0 and finds that an arc joins add0 to add0_reg. Since add0_reg is in V_D, it is put into the set $V_{D,in0}$ and the operation on in0 terminates. The resulting channel destination set for in0 and in1 is {add0_reg}, that for in2 and in3 is {add1_reg}, that for add0_reg and add1_reg is {out}. FIG. 14 depicts the digraph D_2.

2.6. Generation of Modular RTL Model

[0069] The generation of modular RTL models equivalent in behavior to the specifications consists of several operations. As an illustration, the generated RTL model that corresponds to the specification in Example 5 is shown below.

```
1     module addition (rst,in0_req,in0_ack,in0,in1_req,in1_ack,in1,
2     in2_req,in2_ack,in2,in3_req,in3_ack,in3,out_req,out_ack,out);
3
4     input rst;
5     input in0_req;
6     output in0_ack;
7     input [3:0] in0;
8     input in1_req;
9     output in1_ack;
10    input [3:0] in1;
11    input in2_req;
12    output in2_ack;
13    input [3:0] in2;
14    input in3_req;
15    output in3_ack;
```

```
16    input [3:0] in3;
17    output out_req;
18    input out_ack;
19    output [5:0] out;
20
21    reg [4:0] add0_reg;
22    reg [4:0] add1_reg;
23    reg [5:0] out;
24    wire [4:0] add0;
25    wire [4:0] add1;
26
27    sync2i1o sync2i1o_add0_reg
28    (.rst(rst),
29    .ri0(in0_req),.ai0(in0_ack),
30    .ri1(in1_req),.ai1(in1_ack),
31    .ro(add0_reg_req),.ao(add0_reg_ack),
32    .lb(add0_reg_lb));
33
34    sync2i1o sync2i1o_add1_reg
35    (.rst(rst),
36    .ri0(in2_req),.ai0(in2_ack),
37    .ri1(in3_req),.ai1(in3_ack),
38    .ro(add1_reg_req),.ao(add1_reg_ack),
39    .lb(add1_reg_lb));
40
41    sync2i1o sync2i1o_out
43    (.rst(rst),
44    .ri0(add0_reg_req),.ai0(add0_reg_ack),
45    .ri1(add1_reg_req),.ai1(add1_reg_ack),
46    .ro(out_req),.ao(out_ack),
47    .lb(out_lb));
48
49    assign add0=in0+in1;
50    assign add1=in2+in3;
51
```

```
52    always @(add0 or add0_reg_lb or add1 or add1_reg_lb or out_lb)
53          begin
54          #1;
55          if (~add0_reg_lb)
56                add0_reg<=add0;
57          if (~add1_reg_lb)
58                add1_reg<=add1;
59          if (~out_lb)
60                out<=add0_reg+add1_reg;
61          end
62
63    endmodule
```

[0070] First, the established channels are expanded into explicit handshake signals and bundled data. For example, the input channel (`in0`,`add0_reg`) is expanded into the request signal `in0_req`, the acknowledge signal `in0_ack` and the bundled data `in0[3:0]`. These signals are declared as ports at lines 5 to 7. Likewise, the channel (`add0_reg`,`out`) is expanded into the request signal `add0_reg_req`, the acknowledge signal `add0_reg_ack` and the bundled data `add0_reg[4:0]`.

[0071] Second, the inferred handshake modules are instantiated. In the model, three `Sync2i1o` (`Sync` with two input channels and one output channel) modules are instantiated at lines 27 to 47. The first module `Sync2i1o_add0_reg` controls the variable `add0_reg`, the second module `Sync2i1o_add0_reg` controls the variable `add0_reg`, and the third module `Sync2i1o_out` controls the variable `out`.

[0072] Third, the instantiated modules are interconnected based on the established channels. For example, the module `Sync2i1o_add0_reg` is connected to the module `Sync2i1o_out` via the request signal `add0_reg_req` and the acknowledge signal `add0_reg_ack`.

[0073] Fourth, the control of the instantiated handshake modules over their respective datapath components is established. In the always block at lines 52 to 61, the latch-enable

signal `add0_reg_lb` from the module `Sync2i1o_add0_reg` controls the variable `add0_reg`. Likewise, the latch-enable signals add1_reg_lb and out_lb from the modules `Sync2i1o_add0_reg` and `Sync2i1o_add0_reg`, respectively, control the variables `add1_reg` and `out`.

Claims

We claim:

1. A compiler for elaborating Hardware Description Language (HDL) specifications of asynchronous digital-logic circuits into equivalent modular Register-Transfer-Level (RTL) models that are suitable for HDL simulations and logic synthesis, comprising the steps of:

a. reading and analyzing said specifications that are written in standard Verilog HDL and augmented by a set of compiler directives;

b. constructing the internal representations of said specifications using directed graphs; and

c. generating equivalent modular RTL models of said specifications.

2. The method of claim 1 wherein said compiler directives comprise:

a. an attribute that is attached to an input port in said specification to specify that said input port is an asynchronous reset input for all handshake modules;

b. an attribute that is attached to a variable in said specification to specify that said variable has passive output channels;

c. an attribute that is attached to an input port in said specification to specify that said input port is to be implemented as an active input channel;

d. an attribute that is attached to a module in said specification wherein the value assigned to said attribute is a set of input ports or variables that share common communication channels and are controlled by the same handshake module;

e. an attribute that is attached to a conditional statement in said specification to specify that the variables assigned values within said conditional statement have output channels on which data transfers are initiated only when the expression in said conditional statement is evaluated to be true;

f. an attribute that is attached to an input or output port in said specification to specify that the channel that corresponds to said input or output port is a control channel, wherein a control channel is one that contains only handshake signals and no bundled data;

g. an attribute that is attached to a variable in said specification to specify that said variable is to be implemented using edge-sensitive storage devices;

h. an attribute that is attached to a module in said specification wherein the value assigned to said attribute is a set of input ports, output ports or variables such that no more than one of the corresponding channels is active at any point in time;
i. an attribute that is attached to a named event in said specification to specify an association between the said named event and an input port or a variable, wherein said association means that said named event represents events on the channel that corresponds to said input port or the output channel that corresponds to said variable;
j. an attribute that is attached to a variable in said specification to specify that no handshake module is to be inferred for said variable; and
k. an attribute that is attached to a causal relation in said specification to specify that no channel is to be established based on said causal relation.

3. The method of claim 1 wherein said step of constructing the internal representations of said specifications comprises the steps of:
a. extracting the causal relations between the entities declared in said specifications; and
b. establishing the communication channels that correspond to said causal relations.

4. The method of claim 3 wherein said causal relations between entities declared in said specifications comprise:
a. data-dependence wherein the value of an entity is dependent on the value of another entity either combinationally or sequentially; and
b. event-dependence wherein the assignment of a new value to an entity triggers another entity that is a named event, or the triggering of an entity that is a named event assigns a new value to another entity.

5. The method of claim 3 wherein said step of extracting the causal relations between the entities declared in said specifications leads to a directed graph wherein the vertices are said entities and the arcs represent said causal relations between said entities.

6. The method of claim 3 wherein said step of extracting the causal relations between the entities declared in said specifications comprises the step of analyzing the following statements in said specifications:

a. continuous assignments wherein causal relations are established from the operands in the expression on the right-hand side of the assignment to the nets on the left-hand side of the assignment;
b. procedural assignment not specified within any behavioral construct wherein causal relations are established from the operands in the expression on the right-hand side of the assignment to the variables on the left-hand side of the assignment;
c. procedural assignments specified within conditional statements wherein causal relations are established from the operands in the expressions on the right-hand side of the assignments and the operands in the conditional expressions to the variables on the left-hand side of the assignments;
d. procedural assignments specified within case statements wherein causal relations are established from the operands in the case expression, the operands in the case item expressions and the operands in the expressions on the right-hand side of the assignments to the variables on the left-hand side of the assignments; and
e. event controls and triggers wherein causal relations are established from the operands in the conditional expression of a conditional statement to the named event triggered within the conditional statement, and from the named event associated to an event-control to the variables on the left-hand side of the assignments controlled by the same event-control.

7. The method of claim 3 wherein said step of establishing the communication channels that correspond to said causal relations in said specifications leads to a directed graph wherein the vertices are the module and instance ports, registers, named events and channel multiplexers in said specifications, and the arcs represent the communication channels in said specifications.

8. The method of claim 3 wherein said step of establishing the communication channels that correspond to said causal relations in said specifications comprises the steps of:
a. decomposing said vertices of said directed graph of claim 7 into two intersecting subsets of all channel sources and all channel destinations;
b. traversing all paths in said directed graph of claim 5 leading away from each channel source and terminating a path when it reaches a vertex that is a channel destination; and
c. forming a channel from said channel source to said channel destination.

9. The method of claim 1 wherein said step of generating equivalent modular RTL models of said specifications comprises the steps of:

a. inferring handshake modules from said specifications and said internal representations of said specifications;
b. expanding said channels into explicit handshake signals and bundled data;
c. instantiating said handshake modules;
d. interconnecting said handshake modules based on said channels; and
e. establishing the control of said handshake modules over their respective variables.

10. The method of claim 9 wherein said step of inferring handshake modules from said specifications and said internal representations of said specifications comprises the step of analyzing each vertex of said directed graph of claim 7 and the characteristics of said vertex's incoming and outgoing arcs of said directed graph of claim 7 to infer one of the following handshake modules:

a. a synchronizer that performs basic handshake communications with its predecessors and successors, has one or more passive input channels and one or more active output channels, and controls one or more storage devices;
b. a variant of said synchronizer that has one or more passive output channels and contains one or more arbiters to arbitrate between concurrent events on its input and output channels;
c. a variant of said synchronizer that has at least one passive input channels and one or more active input channels wherein data transfers on said active input channels are initiated by requests received on said passive input channels;
d. a variant of said synchronizer that has a one-bit input that determines whether data transfers are initiated on the active output channels after request are received on the passive input channels;
e. a variant of said synchronizer that has one or more active input channels and a one-bit input that determines whether data transfers are initiated on the active output channels after request are received on the passive input channels;

f. a channel multiplexer that merges two passive input channels into one active output channel, and controls a data multiplexer and one or more storage devices, wherein data transfers occur on only one of said passive input channels at any point in time;
g. a variant of said channel multiplexer that contains an arbiter to arbitrate between concurrent events on its input channels;
h. a variant of said channel multiplexer that does not control any storage device;
i. a variant of said channel multiplexer that contains an arbiter to arbitrate between concurrent events on its input channels and does not control any storage device; and
j. a loop implementer that implements behavioral loops, comprising two sub-modules, an initiation channel, an evaluation channel and an iteration channel, wherein the first said sub-module upon receiving a request on said initiation channel activates said evaluation channel, and the second said sub-module upon receiving a request on said evaluation channel either activates said iteration channel if iteration is required or activates its output channel if iteration is not required.

11. The method of claim 9 wherein said step of expanding said channels into explicit handshake signals and bundled data comprises the step of replacing a channel with a request signal, an acknowledge signal and the corresponding data, if any.

12. The method of claim 9 wherein said step of establishing the control of said handshake modules over their respective variables comprises the step of writing conditional statements wherein said variables are assigned values.