(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
8 August 2002 (08.08.2002)

PCT

(10) International Publication Number
WO 02/061633 A2

(51) International Patent Classification⁷: G06F 17/50

(21) International Application Number: PCT/GB02/00384

(22) International Filing Date: 29 January 2002 (29.01.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/772,616 29 January 2001 (29.01.2001) US

(71) Applicant *(for all designated States except US)*: **CELOXICA LIMITED** [GB/GB]; 7 Milton Park, Abingdon, Oxfordshire OX14 4RT (GB).

(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: **BOWEN, Matt** [GB/GB]; 47 Surman House, Mandelbrote Drive, Littlemore, Oxford OX4 4XG (GB).

(74) Agents: **NICHOLLS, Michael, John** et al.; J.A. Kemp & Co., 14 South Square, Gray's Inn, London WC1R 5JJ (GB).

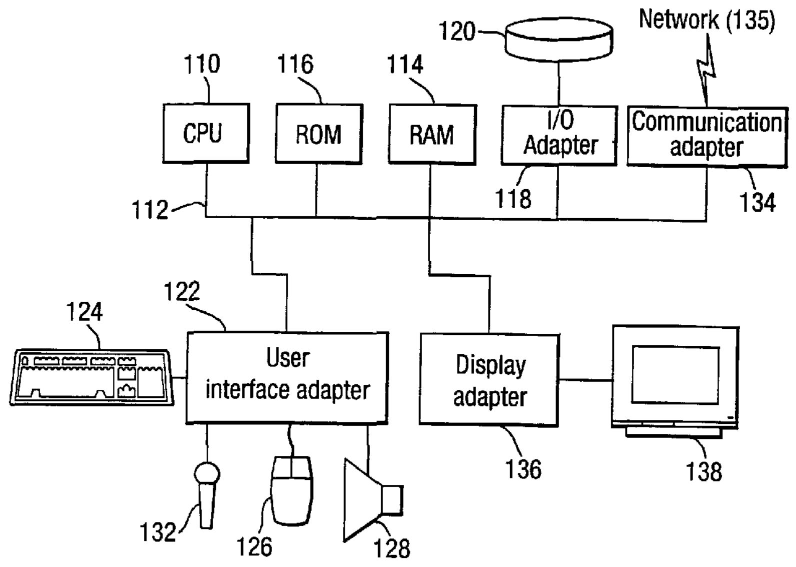
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR SIGNAL CONSTRUCTS IN A PROGRAMMING LANGUAGE CAPABLE OF PROGRAMMING HARDWARE ARCHITECTURES



(57) **Abstract:** A system, method and article of manufacture are provided for using a dynamic object in a programming language. In general, an object is defined with an associated first value and second value. The first value is used in association with the object during a predetermined clock cycle. The second value is used in association with the object before or after the predetermined clock cycle.

WO 02/061633 A2

SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR SIGNAL CONSTRUCTS IN A PROGRAMMING LANGUAGE CAPABLE OF PROGRAMMING HARDWARE ARCHITECTURES

FIELD OF THE INVENTION

The present invention relates to programmable hardware architectures and more particularly to programming field programmable gate arrays (FPGA's).

BACKGROUND OF THE INVENTION

It is well known that software-controlled machines provide great flexibility in that they can be adapted to many different desired purposes by the use of suitable software. As well as being used in the familiar general purpose computers, software-controlled processors are now used in many products such as cars, telephones and other domestic products, where they are known as embedded systems.

However, for a given function, a software-controlled processor is usually slower than hardware dedicated to that function. A way of overcoming this problem is to use a special software-controlled processor such as a RISC processor which can be made to function more quickly for limited purposes by having its parameters (for instance size, instruction set etc.) tailored to the desired functionality.

Where hardware is used, though, although it increases the speed of operation, it lacks flexibility and, for instance, although it may be suitable for the task for which it was designed it may not be suitable for a modified version of that task which is desired later. It is now possible to form the hardware on reconfigurable logic circuits, such as Field Programmable Gate Arrays (FPGA's) which are logic circuits which can be repeatedly reconfigured in different ways. Thus they provide the speed advantages of dedicated hardware, with some degree of flexibility for later updating or multiple functionality.

In general, though, it can be seen that designers face a problem in finding the right balance between speed and generality. They can build versatile chips which will be software controlled and thus perform many different functions relatively slowly, or they can devise application-specific chips that do only a limited set of tasks but do them much more quickly.

SUMMARY OF THE INVENTION

A system, method and article of manufacture are provided for using a dynamic object in a programming language. In general, an object is defined with an associated first value and second value. The first value is used in association with the object during a predetermined clock cycle. The second value is used in association with the object before or after the predetermined clock cycle.

In an aspect of the present invention, the object may be used to split up an expression into sub-expressions. As an option, the sub-expressions may be reused. In another aspect, the first value may be assigned to and read from the object during the predetermined clock cycle. In a further aspect, the programming language may be adapted for programming a gate array. As an option, the programming language may include Handel-C.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be better understood when consideration is given to the following detailed description thereof. Such description makes reference to the annexed drawings wherein:

Figure **1** is a schematic diagram of a hardware implementation of one embodiment of the present invention;

Figure **2** illustrates a design flow overview, in accordance with one embodiment of the present invention;

Figures **3A** and **3B** illustrate a table showing various differences between Handel-C and the conventional C programming language, in accordance with one embodiment of the present invention;

Figure **4** illustrates the manner in which branches that complete early are forced to wait for the slowest branch before continuing;

Figure **5** illustrates the link between parallel branches, in accordance with one embodiment of the present invention;

Figure **6** illustrates the scope of variables, in accordance with one embodiment of the present invention

Figure **7** illustrates a method for using a dynamic object in a programming language.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

A preferred embodiment of a system in accordance with the present invention is preferably practiced in the context of a personal computer such as an IBM compatible personal computer, Apple Macintosh computer or UNIX based workstation. A representative hardware environment is depicted in Figure **1**, which illustrates a typical hardware configuration of a workstation in accordance with a preferred embodiment having a central processing unit **110**, such as a microprocessor, and a number of other units interconnected via a system bus **112**.

The workstation shown in Figure **1** includes a Random Access Memory (RAM) **114**, Read Only Memory (ROM) **116**, an I/O adapter **118** for connecting peripheral devices such as disk storage units **120** to the bus **112**, a user interface adapter **122** for connecting a keyboard **124**, a mouse **126**, a speaker **128**, a microphone **132**, and/or other user interface devices such as a touch screen (not shown) to the bus **112**, communication adapter **134** for connecting the workstation to a communication network (e.g., a data processing network) and a display adapter **136** for connecting the bus **112** to a display device **138**.

The workstation typically has resident thereon an operating system such as the Microsoft Windows NT or Windows/95 Operating System (OS), the IBM OS/2 operating system, the MAC OS, or UNIX operating system. Those skilled in the art will appreciate that the present invention may also be implemented on platforms and operating systems other than those mentioned.

In one embodiment, the hardware environment of Figure **1** may include, at least in part, a field programmable gate array (FPGA) device. For example, the central processing unit **110** may be replaced or supplemented with an FPGA. Use of such device provides flexibility in functionality, while maintaining high processing speeds.

Examples of such FPGA devices include the XC2000™ and XC3000™ families of FPGA devices introduced by Xilinx, Inc. of San Jose, Calif. The architectures of these devices are exemplified in U.S. Pat. Nos. 4,642,487; 4,706,216; 4,713,557; and 4,758,985; each of which is originally assigned to Xilinx, Inc. and which are herein incorporated by reference for all purposes. It should be noted, however, that FPGA's of any type may be employed in the context of the present invention.

A preferred embodiment is written using Handel-C. Handel-C is a programming language marketed by Celoxica Limited. Handel-C is a programming language that enables a software or hardware engineer to target directly FPGAs (Field Programmable Gate Arrays) in a similar fashion to classical microprocessor cross-compiler development tools, without recourse to a Hardware Description Language. This allows the designer to directly realize the raw real-time computing capability of the FPGA.

Handel-C allows one to use a high-level language to program FPGAs. It makes it as easy to implement complex algorithms by using a software-based language rather than a hardware architecture-based language. One can use all the power of reconfigurable computing in FPGAs without needing to know the details of the FPGAs themselves. A program may be written in Handel-C to generate all required state machines, while one can specify storage requirements down to the bit level. A clock and clock speed may be assigned for working with the simple but explicit model of one clock cycle per assignment. A Handel-C macro library may be used for bit manipulation and arithmetic operations. The program may be compiled and then simulated and debugged on a PC similar to that in Figure 1. This may be done while stepping through single or multiple clock cycles.

When one has designed their chip, the code can be compiled directly to a netlist, ready to be used by manufacturers' place and route tools for a variety of different chips.

As such, one can design hardware quickly because he or she can write high-level code instead of using a hardware description language. Handel-C optimizes code, and uses efficient algorithms to generate the logic hardware from the program. Because of the speed of development and the ease of maintaining well-commented high-level code, it allows one to use reconfigurable computing easily and efficiently.

Handel-C has the tight relationship between code and hardware generation required by hardware engineers, with the advantages of high-level language abstraction. Further features include:

- C-like language allows one to program quickly
- Architecture specifiers allow one to define RAMs, ROMs, buses and interfaces.
- Parallelism allows one to optimize use of the FPGA
- Close correspondence between the program and the hardware
- Easy to understand timing model
- Full simulation of owner hardware on the PC
- Display the contents of registers every clock cycle during debug
- Rapid prototyping
- Convert existing C programs to hardware
- Works with manufacturers' existing tools
- Rapid reconfiguration
- Logic estimation tool highlights code inefficiencies in colored Web pages
- Device-independent programs
- Generates EDIFand XNF formats (and XBLOX macros)

Handel-C is thus designed to enable the compilation of programs into synchronous hardware; it is aimed at compiling high level algorithms directly into gate level hardware. The Handel-C syntax is based on that of conventional C so programmers familiar with conventional C will recognize almost all the constructs in the Handel-C language. Sequential programs can be written in Handel-C just as in conventional C but to gain the most benefit in performance from the target hardware its inherent parallelism must be exploited. Handel-C includes parallel constructs that provide the means for the programmer to exploit this benefit in his applications. The compiler compiles and optimizes Handel-C source code into a file suitable for simulation or a net list which can be placed and routed on a real FPGA.

For more information regarding the Handel-C programming language, reference may be made to “EMBEDDED SOLUTIONS Handel-C Language Reference Manual: Version 3,” “EMBEDDED SOLUTIONS Handel-C User Manual: Version 3.0,” “EMBEDDED SOLUTIONS Handel-C Interfacing to other language code blocks: Version 3.0,” and “EMBEDDED SOLUTIONS Handel-C Preprocessor Reference Manual: Version 2.1,” each authored by Rachel Ganz, and published by Embedded Solutions Limited, and which are each incorporated herein by reference in their entirety.

Conventions

A number of conventions are used throughout this document. These conventions are detailed below. Hexadecimal numbers appear throughout this document. The convention used is that of prefixing the number with '0x' in common with standard C syntax.
Sections of code or commands that one must type are given in typewriter font as follows:

“void main();”

Information about a type of object one must specify is given in italics as follows:

"**copy** *SourceFileName DestinationFileName*"

Menu items appear in narrow bold text as follows:

"insert Project into Workspace"

Elements within a menu are separated from the menu name by a > so **Edit>Find** means the **Find** item in the **Edit** menu.

Introduction

Handel-C is a programming language designed to enable the compilation of programs into synchronous hardware. The Handel-C compiler and simulator will now be described. The Handel-C language will be described hereinafter in greater detail.

Overview

Design flow overview

Figure **2** illustrates a design flow overview **200**, in accordance with one embodiment of the present invention. The dotted lines **202** show the extra steps **204** required if one wishes to integrate Handel-C with VHDL.

HARDWARE EMBODIMENTS

If one is approaching Handel-C from a hardware background, one should be aware of these points:

- Handel-C is halfway between RTL and a behavioral HDL. It is a high-level language that requires one to think in algorithms rather than circuits.
- Handel-C uses a zero-delay model and a synchronous design style.
- Handel-C is implicitly sequential. Parallel processes must be specified.
- All code in Handel-C (apart from the simulator **chanin** and **chanout** commands) can be synthesized. so one must ensure that he or she disables debug code when he or she compiles to target real hardware.
- Signals in Handel-C are different from signals in VHDL; they are assigned to immediately, and only hold their value for one clock cycle.
- Handel-C has abstract high-level concepts such as pointers.

Points of difference

Figures **3A** and **3B** illustrate a table showing various differences **3100** between Handel-C and the conventional C programming language, in accordance with one embodiment of the present invention.

1 LANGUAGE REFERENCE

This section deals with some of the basics behind the Handel-C language. Handel-C uses the syntax of conventional C with the addition of inherent parallelism. One can write sequential programs in Handel-C, but to gain maximum benefit in performance from the target hardware one must use its parallel constructs. These may be new to some users.

If one is familiar with conventional C he or she may recognize nearly all the other features. Handel-C is designed to allow one to express the algorithm without worrying about how the underlying computation engine works. This philosophy makes Handel-C a programming language rather than a hardware description language. In some senses, Handel-C is to hardware what a conventional high-level language is to microprocessor assembly language.

It is important to note that the hardware design that Handel-C produces is generated directly from the source program. There is no intermediate 'interpreting' layer as exists in assembly language when targeting general purpose microprocessors. The logic gates that make up the final Handel-C circuit are the assembly instructions of the Handel-C system.

Handel-C programs

Since Handel-C is based on the syntax of conventional C, programs written in Handel-C are implicitly sequential. Writing one command after another indicates that those instructions should be executed in that exact order.

Just like any other conventional language, Handel-C provides constructs to control the flow of a program. For example, code can be executed conditionally depending on the value of some expression, or a block of code can be repeated a number of times using a loop construct.

Parallel programs

Because the target of the Handel-C compiler is low-level hardware, massive performance benefits are made possible by the use of parallelism. It is possible (and indeed essential for writing efficient programs) to instruct the compiler to build hardware to execute statements in parallel. Handel-C parallelism is true parallelism - it is not the time-sliced parallelism familiar from general purpose computers.

When instructed to execute two instructions in parallel, those two instructions may be executed at exactly the same instant in time by two separate pieces of hardware.

When a parallel block is encountered, execution flow splits at the start of the parallel block and each branch of the block executes simultaneously. Execution flow then re-joins at the end of the block when all branches have completed. Figure **4** illustrates the manner **4900** in which branches that complete early are forced to wait for the slowest branch before continuing.

Figure **4** illustrates the branching and re-joining of the execution flow. The left hand branch **4902** and middle branch **4904** must wait to ensure that all branches have completed before the instruction following the parallel construct can be executed.

Channel communications

Figure **5** illustrates the link **5000** between parallel branches, in accordance with one embodiment of the present invention. Channels **5001** provide a link between parallel branches. One parallel branch **5002** outputs data onto the channel and the other branch **5004** reads data from the channel. Channels also provide synchronization between parallel branches because the data transfer can only complete when both parties are ready for it. If the transmitter is not ready for the communication then the receiver must wait for it to become ready and vice versa.

Here, the channel is shown transferring data from the left branch to the right branch. If the left branch reaches point **a** before the right branch reaches point b, the left branch waits at point **a** until the right branch reaches point **b**.

Scope and variable sharing

Figure **6** illustrates the scope **6100** of variables, in accordance with one embodiment of the present invention. The scope of declarations is, as in conventional C, based around code blocks. A code block is denoted with {...} brackets. This means that:

- Global variables must be declared outside all code blocks.

- An identifier is in scope within a code block and any sub-blocks of that block.

Since parallel constructs are simply code blocks, variables can be in scope in two parallel branches of code. This can lead to resource conflicts if the variable is written to simultaneously by more than one of the branches. Handel-C syntax states that a single variable must not be written to by more than one parallel branch but may be read from by several parallel branches. This provides some powerful operations to be described later.

If one wishes to write to the same variable from several processes, the correct way to do so is by using channels which are read from in a single process. This process can use a **prialt** statement to select which channel is ready to be read from first, and that channel is the only one which may be allowed to write to the variable

```
while(1)
prialt
{
case chan1 ? y:
break;
case chan2 ? y:
break;
case chan3 ? y:
break;
}
```

In this case, three separate processes can attempt to change the value of **y** by sending data down the channels, **chan1**, **chan2** and **chan3**. **y** may be changed by whichever process sends the data first. ***A single variable should not be written to by more than one***

parallel branch..

1.1 Alternate Embodiments

Introduction

This section summarizes some new features in Handel-C version 3 for those familiar with previous versions. It also details incompatibilities between the current version and Handel-C version 2.1.

The following constructs have been added or changed. Terms specific to Handel-C have been given in bold. All other terms are fully compatible with ISO-C (ISO/IEC 9899:1999) unless otherwise stated. (ISO-C was previously known as ANSI-C.)

Architecture

There is a new type to represent signals. One can have multi-dimensional arrays of RAMs and dual-ported RAMs. Interfaces have been extended to allow one to connect to undefined input or output ports. One can also define the sorts of interface and use them to link to blocks of external code (currently VHDL or EDIF). Interfaces declarations have changed, and the previous style is deprecated.

2 LANGUAGE BASICS

Introduction

This section of the present description deals with the basics of producing Handel-C programs

Program structure

Sequential structure

As in a conventional C program, a Handel-C program consists of a series of statements which execute sequentially. These statements are contained within a

main() function that tells the compiler where the program begins. The body of the **main** function may be split into a number of blocks using {...} brackets to break the program into readable chunks and restrict the scope of variables and identifiers.

Handel-C also has functions, variables and expressions similar to conventional C. There are restrictions where operations are not appropriate to hardware implementation and extensions where hardware implementation allows additional functionality.

Parallel structure

Unlike conventional C, Handel-C programs can also have statements or functions that execute in parallel. This feature is crucial when targeting hardware because parallelism is the main way to increase performance by using hardware. Parallel processes can communicate using channels. A channel is a one-way point-to-point link between two processes.

Overall structure

The overall program structure consists of one or more **main** functions, each associated with a clock. One would only use more than one main function if he or she needed parts of the program to run at different speeds (and so use different clocks).. A **main** function is defined as follows:

```
Global Declarations
Clock Definition
void main(void)
{
Local Declarations
Body Code
}
```

The **main()** function takes no arguments and returns no value. This is in line with a hardware implementation where there are no command line arguments and no environment to return values to. The *argc*, *argv* and *envp* parameters and the return value familiar from conventional C can be replaced with explicit communications with an external system (e.g. a host microprocessor) within the body of the program.

Using the preprocessor

As with conventional C, the Handel-C source code is passed through a C preprocessor before compilation. Therefore, the usual **#include** and **#define** constructs may be used to perform textual manipulation on the source code before compilation.

Handel-C also supports macros that are more powerful than those handled by the preprocessor.

Comments

Handel-C uses the standard /* ... */ delimiters for comments. These comments may not be nested. For example:

```
/* Valid comment */
/* This is /* NOT */ valid */
```

Handel-C also provides the C++ style // comment marker which tells the compiler to ignore everything up to the next newline. For example:

```
x = x + 1; // This is a comment
```

Comments are handled by the preprocessor.

Types

Handel-C uses two kinds of objects: logic types and architecture types. The logic types specify variables. The architecture types specify variables that require a particular sort of hardware architecture (e.g., ROMs, RAMs and channels). Both kinds are specified by their scope (**static** or **extern**), their size and their type. Architectural types are also specified by the logic type that uses them.

Both types can be used in derived types (such as structures, arrays or functions) but there may be some restrictions on the use of architectural types.

Specifiers

The type specifiers **signed**, **unsigned** and **undefined** define whether the variable is signed and whether it takes a default defined width. One can use the storage class specifiers **extern** and **static** to define the scope of any variable.

Functions can have the storage class **inline** to show that they are expanded in line, rather than being shared.

Type qualifiers

Handel-C supports the type qualifiers **const** and **volatile** to increase compatibility with ISO-C. These can be used to further qualify logic types.

Disambiguator

Handel-C supports the extension <>. This can be used to clarify complex declarations of architectural types.

Logic types

The basic logic type is an **int**. It may be qualified as **signed** or **unsigned**. Integers can be manually assigned a width by the programmer or the compiler may attempt to infer a width from use. Enumeration types (**enums**) allow one to define a specified set of values that a variable of this type may hold. There are derived types (types that are derived from the basic types). These are arrays, pointers, **structs** bit fields, and functions. The non-type **void** enables one to declare empty parameter lists or functions that do not return a value. The **typeof** type operator allows one to reference the type of a variable.

Architectural types

The architectural types are channels (used to communicate between parallel processes), **interfaces** (used to connect to pins or provide signals to communicate with external code), memories (**rom** , **ram** , **wom** and **mpram**) and **signal** (declares a wire). The disambiguator < > has been provided to help clarify the definitions of memories, channels and signals.

Channels

Handel-C provides channels for communicating between parallel branches of code. One branch writes to a channel and a second branch reads from it. The communication only occurs when both tasks are ready for the transfer at which point one item of data is transferred between the two branches. Channels are declared with the **chan** keyword. For example:

```
chan int 7 link;
```

As with variables, the Handel-C compiler can infer the width of a channel from its usage if it is declared with the **undefined** keyword. Channels can also be declared with no explicit type. The compiler infers the type and width of the channel from its usage.

Arrays of channels

Handel-C allows arrays of channels to be declared. For example:

```
chan unsigned int 5 x[6];
```

This is equivalent to declaring 6 channels each of which is 5 bits wide. It is also possible to declare multi-dimensional arrays of channels. For example:.

```
chan unsigned int 6 x[4][5][6];
```

This declares 4 * 5 * 6 = 120 channels each of which is 6 bits wide

Interfaces

One may use an **interface** to communicate with an external device or component. An interface consists of data *port*s, together with information about each port. A port definition consists of the data type that uses it (either defined or inferred from its first use), an optional name and the specification for that port (e.g., input pins for a bus) if needed.

Internal RAMs and ROMs

RAMs and ROMs may be built from the logic provided in the FPGA using the **ram** and **rom** keywords. For example:

```
ram int 6 a[43];
rom int 16 b[4]; = { 23, 46, 69, 92 };
```

This example constructs a RAM consisting of 43 entries each of which is 6 bits wide and a ROM consisting of 4 entries each of which is 16 bits wide.

Multidimensional arrays

It is possible to create simple multi-dimensional arrays of memory using the **ram**, **rom** and **wom** keywords. The definitions can be made clearer by using the optional disambiguator <>.

signal

Figure 7 illustrates a method **7040** for using a dynamic object, i.e. signal, in a programming language. In general, in operation **7042**, an object is defined with an associated first value and second value. The first value is then used in association with the object during a predetermined clock cycle. See operation **7044**. The second value is used in association with the object before or after the predetermined clock cycle, as indicated in operation **7046**.

In an aspect of the present invention, the object may be used to split up an expression into sub-expressions. As an option, the sub-expressions may be reused. In another aspect, the first value may be assigned to and read from the object during the predetermined clock cycle. In a further aspect, the programming language may be adapted for programming a gate array. As an option, the programming language may include Handel-C.

More information regarding the above concept will now be set forth in greater detail.

A signal is an object that takes on the value assigned to it but only for that clock cycle. The value assigned to it can be read back during the same clock cycle. At all other times it takes on its initialisation value. The default initialisation value is 0. The optional disambiguator <> can be used to clarify complex signal definitions.

Syntax

signal [<*type data-width***>]** ***signal_Name*****;**

Example

```
int 15 a, b;
signal <int> sig;
a = 7;
par
{
sig = a;
b = sig;
}
```

sig is assigned to and read from in the same clock cycle, so **b** is assigned the value of a. Since the signal only holds the value assigned to it for a single clock cycle, if it is read from just before or just after it is assigned to, one gets its initial value. For example:

```
int 15 a, b;
static signal <int> sig = 690;
a = 7;
par
{
sig = a;
b = sig;
}
a = sig;
```

Here, **b** is assigned the value of **a** through the signal, as before. Since there is a clock tick before the last line, **a** is finally assigned the signal's initial value of 690.

Using signals to split up complex expressions

One can split up complex expressions. E.g., **b = (((a * 2) - 55) << 2) + 100;** could also be written

```
int 17 a, b;
signal s1, s2, s3, s4;
par
{
s1 = a;
s2 = s1 * 2;
s3 = s2 - 55;
s4 = s3 << 2;
b = s4 + 100;
}
```

Breaking up expressions also enables one to re-use sub-expressions:

```
unsigned 15 a, b;
signal sig1;
par
{
sig1 = x + 2;
a = sig1 * 3;
b = sig1 / 2;
```

Type qualifiers

Handel-C supports the type-qualifiers **const** and **volatile** to increase compatibility with ISO-C. These can be used to further qualify logic types.

CLAIMS

What is claimed is:

1. A method for using a dynamic object in a programming language, comprising the steps of:
(a) defining an object with an associated first value and second value;
(b) using the first value in association with the object during a predetermined clock cycle; and
(c) using the second value in association with the object before or after the predetermined clock cycle.

2. A method as recited in claim 1, wherein the object is used to split up an expression into sub-expressions.

3. A method as recited in claim 2, wherein the sub-expressions are reused.

4. A method as recited in claim 1, wherein the first value is assigned to and read from the object during the predetermined clock cycle.

5. A method as recited in claim 1, wherein the programming language is adapted for programming a gate array.

6. A method as recited in claim 1, wherein the programming language includes Handel-C.

7. A computer program product for using a dynamic object in a programming language, comprising:
(a) computer code for defining an object with an associated first value and second value;
(b) computer code for using the first value in association with the object during a predetermined clock cycle; and

(c) computer code for using the second value in association with the object before or after the predetermined clock cycle.

8. A computer program product as recited in claim 7, wherein the object is used to split up an expression into sub-expressions.

9. A computer program product as recited in claim 8, wherein the sub-expressions are reused.

10. A computer program product as recited in claim 7, wherein the first value is assigned to and read from the object during the predetermined clock cycle.

11. A computer program product as recited in claim 7, wherein the programming language is adapted for programming a gate array.

12. A computer program product as recited in claim 7, wherein the programming language includes Handel-C.

13. A system for using a dynamic object in a programming language, comprising:

(a) logic for defining an object with an associated first value and second value;

(b) logic for using the first value in association with the object during a predetermined clock cycle; and

(c) logic for using the second value in association with the object before or after the predetermined clock cycle.

14. A system as recited in claim 13, wherein the object is used to split up an expression into sub-expressions.

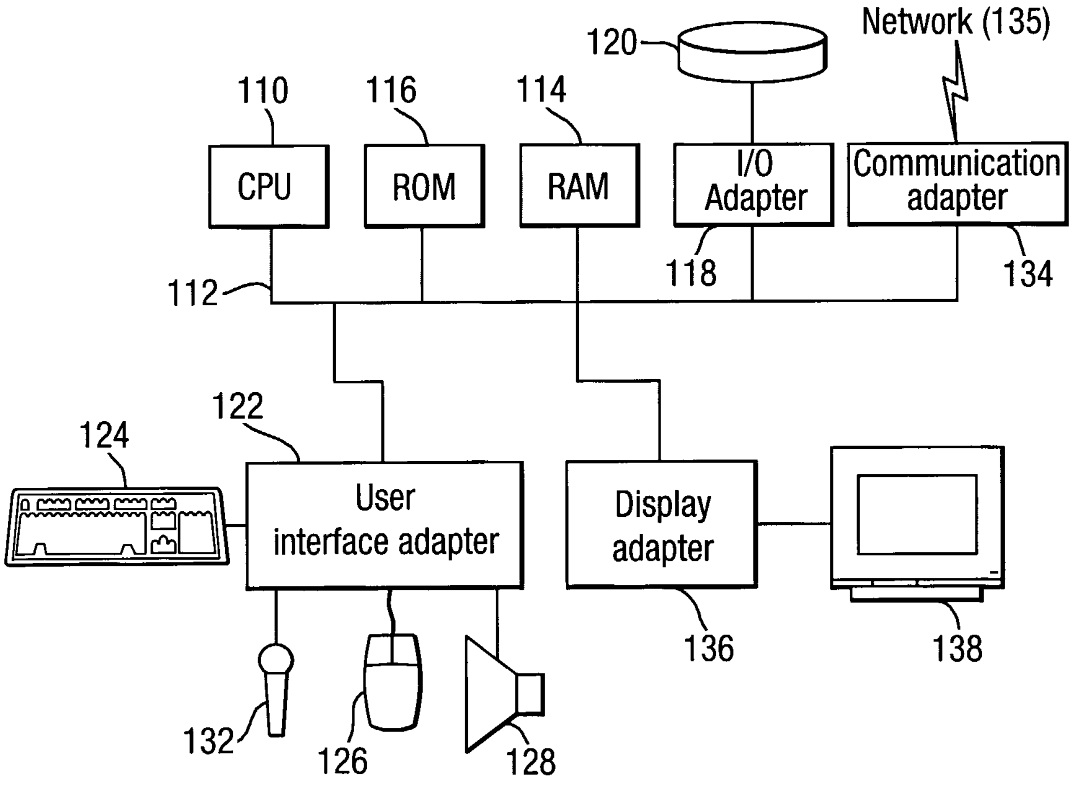
15. A system as recited in claim 14, wherein the sub-expressions are reused.

16. A system as recited in claim 13, wherein the first value is assigned to and read from the object during the predetermined clock cycle.

17. A system as recited in claim 13, wherein the programming language is adapted for programming a gate array.

18. A system as recited in claim 13, wherein the programming language includes Handel-C.

Fig.1.



Network (135)
120
110
116
114
CPU
ROM
RAM
I/O Adapter
Communication adapter
118
134
112
122
124
User interface adapter
Display adapter
136
138
132
126
128

Fig.2.

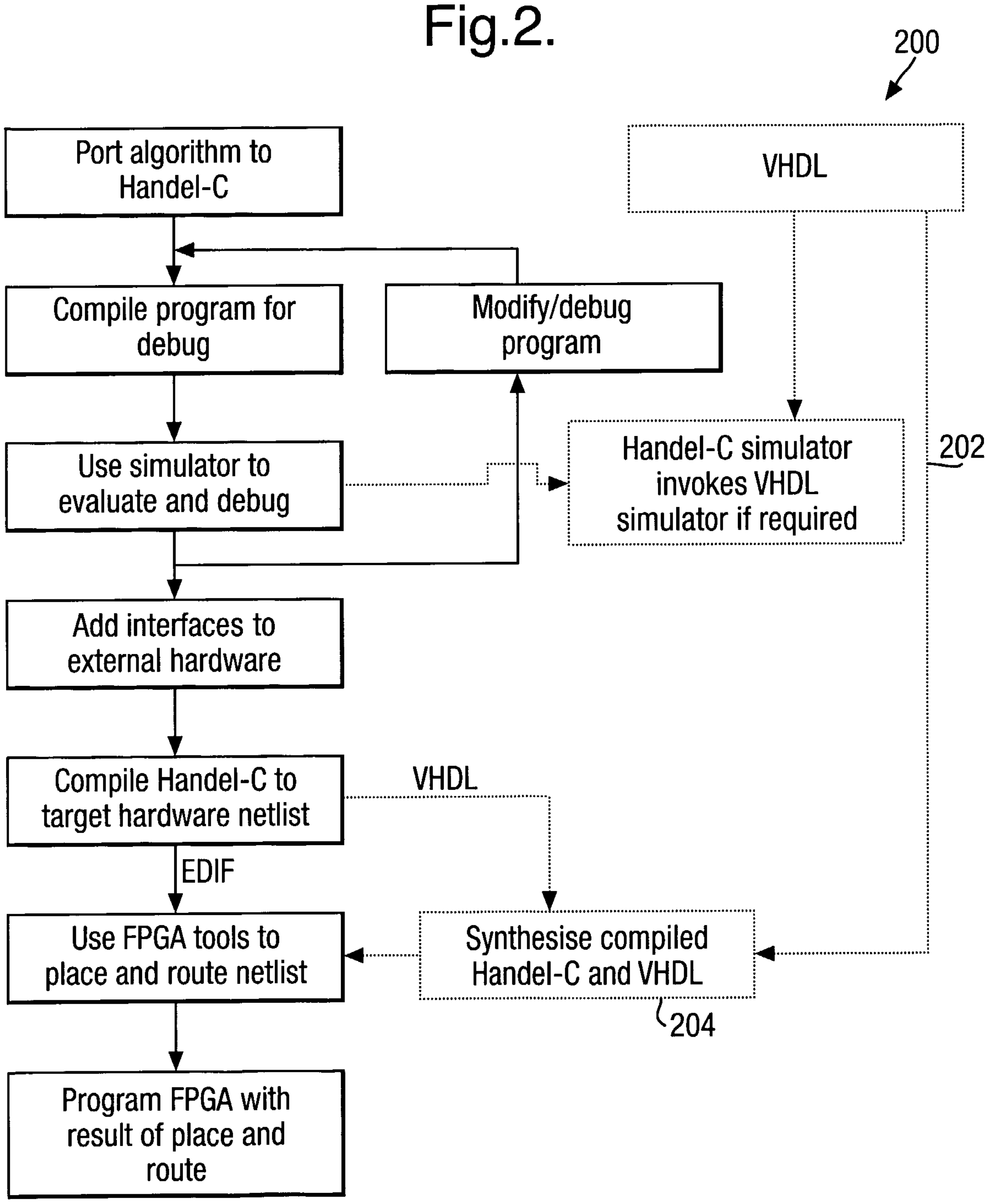

200
Port algorithm to Handel-C
VHDL
Compile program for debug
Modify/debug program
Use simulator to evaluate and debug
Handel-C simulator invokes VHDL simulator if required
202
Add interfaces to external hardware
Compile Handel-C to target hardware netlist
VHDL
EDIF
Use FPGA tools to place and route netlist
Synthesise compiled Handel-C and VHDL
204
Program FPGA with result of place and route

Fig.3A.

3100

Strong typing	Handel-C has variables which can be defined to be of any width.
	Casting can't change width.
	There are no automatic conversions between signed and unsigned values. Instead, values must be 'cast' between types to ensure that the programmer is aware that a conversion is occurring that may alter the meaning of a value.
	Pointers can only be cast to void and back, between signed and unsigned and between similar structs. You cannot cast pointers to any other type.
True parallelism	You can have multiple main functions in a project. Each Handel-C main function must be associated with a clock.
	Although implicity sequential, Handel-C has parallel constructs which allow you to speed up your code.
Width of variables	Handel-C has variables which can be defined to be of any width.
	In ISO-C, bit fields are made up of words, and only the specified bits are accessed, the rest are padded. Since there are no words in Handel-C, no form of packing can be assumed.
	If you have an array[4] and you use its index as a counter, the index width will be assumed by the Handel-C compiler to be two bits wide (to hold the values 0-3). It will not be able to hold the value 4.
No side-effects allowed	Instead of writing complex single statements, it is more efficient in Handel-C to write multiple single statements and run them in parallel.
	You cannot perform two assignments in one statement.
	Auto variables cannot be initialised, as that means that hidden clock cycles are required, Instead, they must be explicity assigned to in a separate statement.

Fig.3B.

3100

	You cannot have empty loops in Handel-C.
Constrained functions	Functions may not be recursive.
	Variable length parameter lists are not supported.
	Old-style function declarations are not supported.

Fig.4.

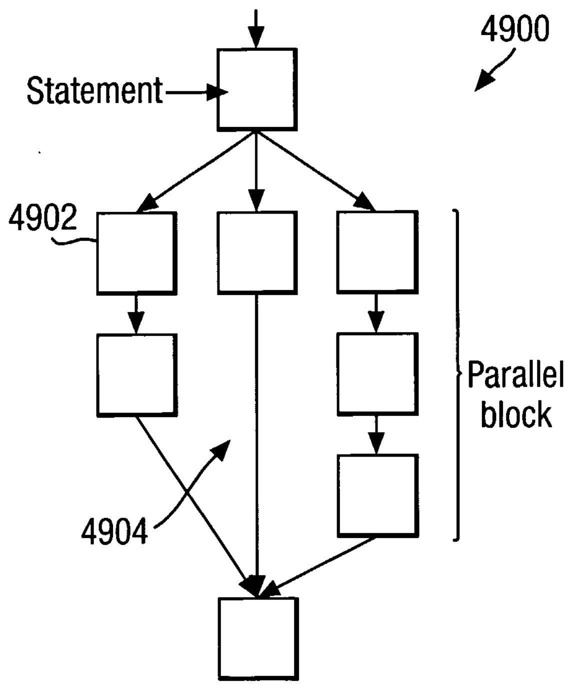


Fig.5.

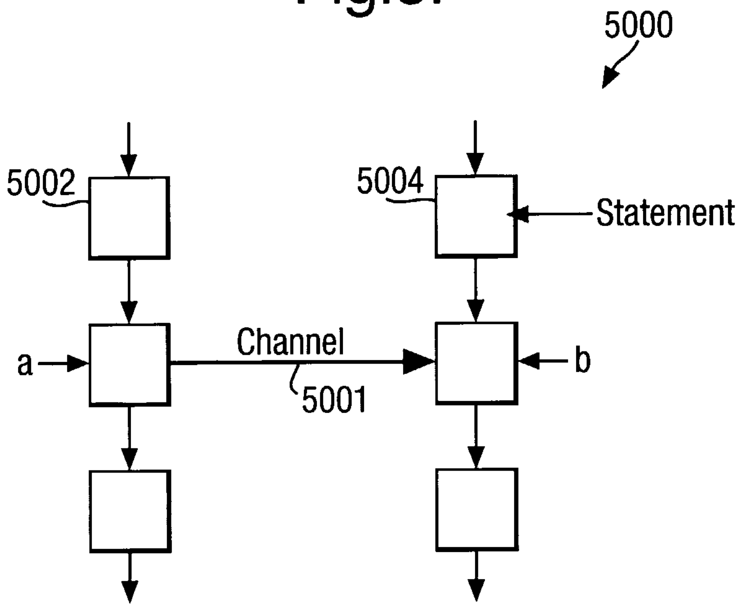
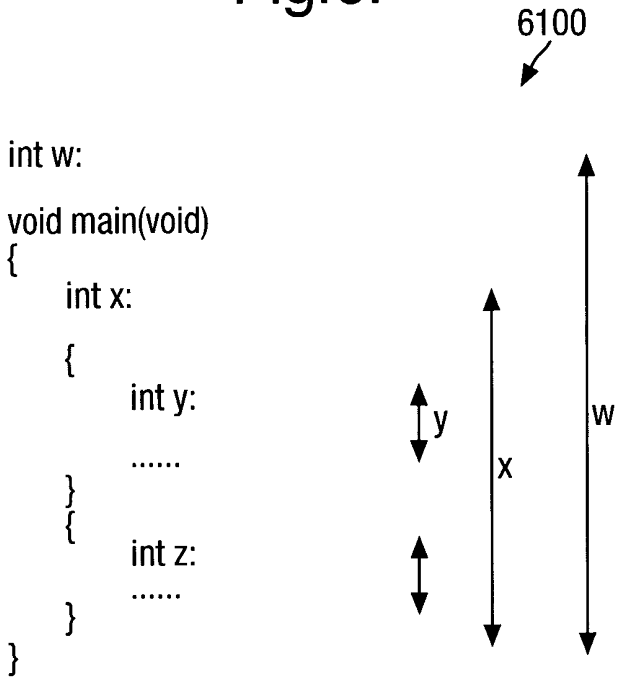


Fig.6.

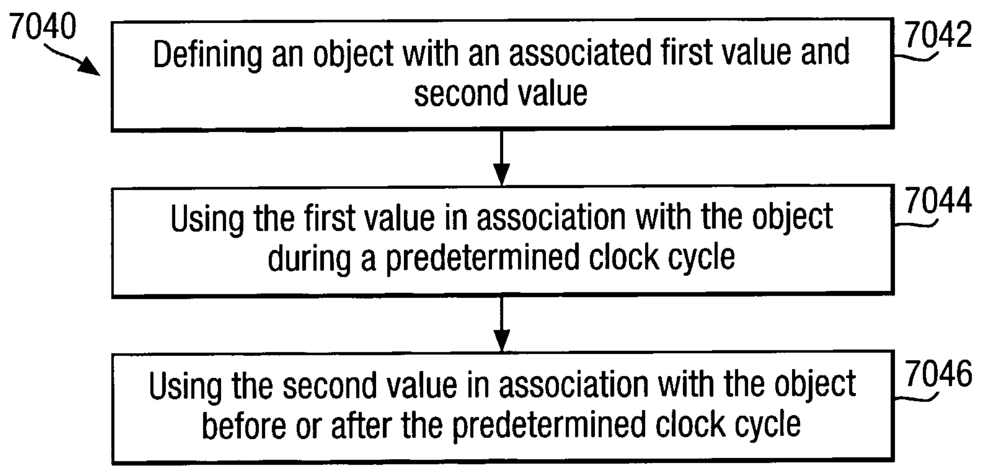


```
int w:

void main(void)
{
    int x:

    {
        int y:
        ......
    }
    {
        int z:
        ......
    }
}
```

Fig.7.


7040
Defining an object with an associated first value and second value
7042
Using the first value in association with the object during a predetermined clock cycle
7044
Using the second value in association with the object before or after the predetermined clock cycle
7046