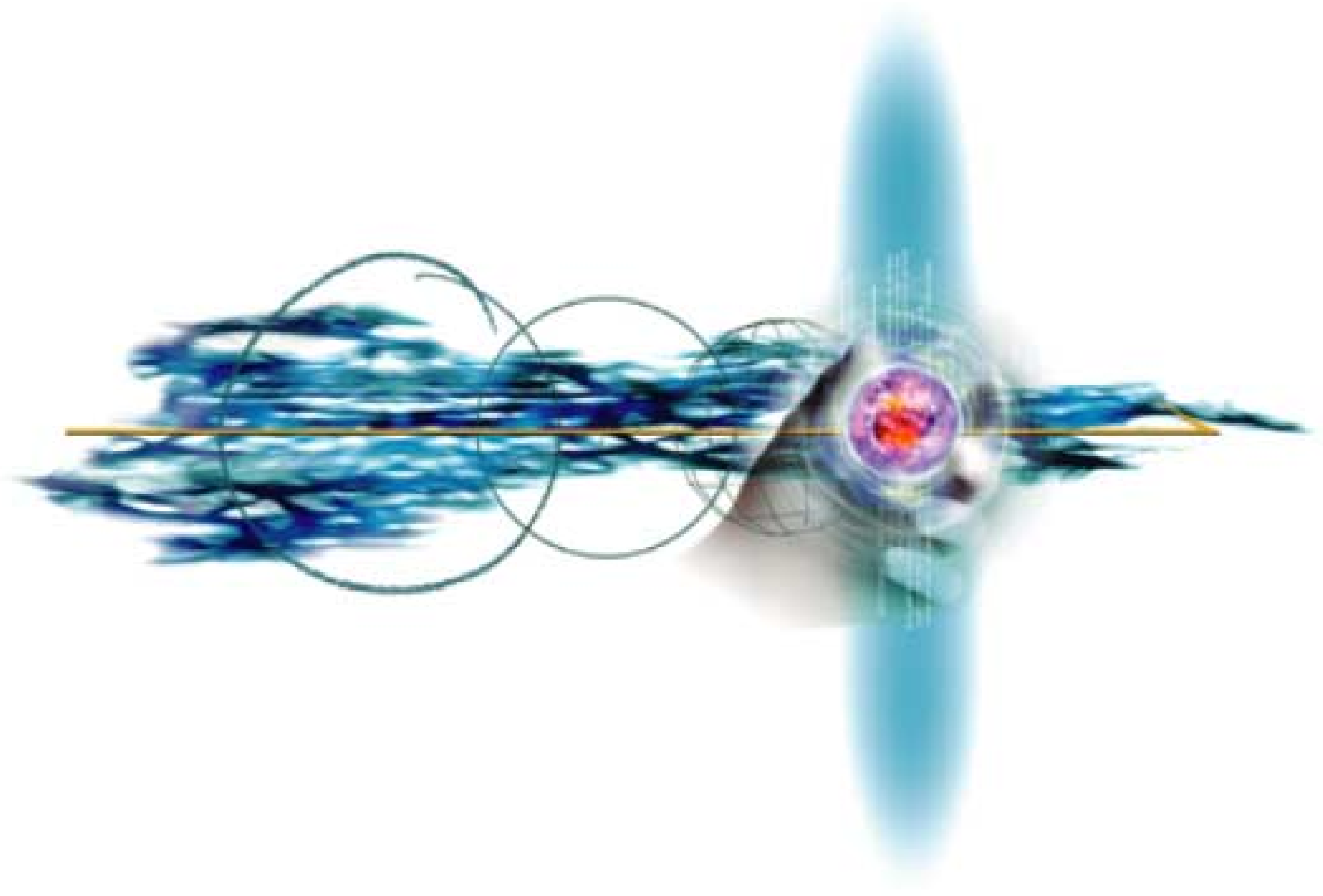


Handel-C Language Reference Manual

Version 3.1

Celoxica

Authors: RG, SB

Document number: RM-1003-3.0

>: Table of contents

Table of contents

>: Table of contents

>: Table of contents

>: Table of contents

>: Table of contents

>: Table of contents

>: Table of contents

>: Conventions

Conventions

A number of conventions are used in this document. These conventions are detailed below.

Warning Message. These messages warn you that actions may damage your hardware.

Handy Note. These messages draw your attention to crucial pieces of information.

Hexadecimal numbers will 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 you must type are given in typewriter font like this:

```
void main();
```

Information about a type of object you must specify is given in italics like this:

`copy` *SourceFileName DestinationFileName*

Optional elements are enclosed in square brackets like this:

`struct` [*type_Name*]

Curly brackets around an element show that it is optional but it may be repeated any number of times.

string ::= **"**{*character*}**"**

>: Assumptions

Assumptions

This manual assumes that you:

- are familiar with common programming terms (e.g., functions)
- are familiar with MS Windows

Omissions

This manual does not include:

- instruction in VHDL
- instruction in the use of place and route tools
- tutorial example programs. These are provided in the *Handel-C User Manual*

>: 1. Introduction

>: 1 Introduction

1.1 References

- The C Programming Language 2nd Edition
  Kernighan, B. and Ritchie, D.
  Prentice-Hall, 1988
- Xilinx Data Book
  Xilinx 2000
- Altera Databook
  Altera 2001
  www.altera.com/literature/lit-index.html
- VHDL for logic synthesis
  Author: Andrew Rushton
  Publisher: John Wiley and Sons
  ISBN: 0-471-98325-X
  Published: May 1998
- IEEE standard 1364 -1995
  IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language.
  http://standards.ieee.org/

>: 2 Getting started with Handel-C

2.1 Language changes in DK1.1

- You must now initialize your variables to zero explictly.
- You can now perform a global reset on your design, using `set reset`.
- You can make direct calls to C or C++ functions using `extern " language”`.
- The trysema() expression has been corrected so that you can no longer take the semaphore twice without releasing it.

New specifications

- `fastclock`: allows you to specify that an external clock should use a fast clock buffer
- properties: allows you to specify parameterize interfaces when compiling to an EDIF netlist
- std_logic_vector: allows you to specify that a port is a VHDL `std_logic_vector` port

Extensions to existing specifications

- busformat: now allows you to specify a single port for the whole bus
- clockport: can now be applied to external clocks
- dci: can now have split termination or single termination

2.2 Notes for C programmers

If you are an experienced C user, you may be caught unawares by some of the differences between C and Handel-C.

`auto` variables cannot be initialized, as that means that hidden clock cycles are required. Instead, they must be explicitly assigned to in a separate statement.

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.

>: 2. Getting started with Handel-C

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 implicitly 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.

You may not change the width of a variable by casting.

In ANSI-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

You cannot perform two assignments in one statement.

This means:

- Shortcut assignments (e.g. +=) must appear as standalone statements.
- The initialization and iteration phases of `for` loops must be statements, not expressions
- You cannot have empty loops in Handel-C.

Instead of writing complex single statements, it is more efficient in Handel-C to write multiple single statements and run them in parallel.

Constrained functions

Functions may not be called recursively.

Variable length parameter lists are not supported.

Old-style function declarations are not supported.

2.2.1 How Handel-C differs from ANSI-C

Handel-C differs from ANSI-C in the following ways:

>: 2. Getting started with Handel-C

- Functions may not be called recursively.
- Old-style function declarations are not supported.
- Variable length parameter lists are not supported.
- You may not change the width of a variable by casting.
- You cannot convert pointer types except to and from `void`, between `signed` and `unsigned` and between similar `structs`.
- Floating point is not supported.
- Expressions in Handel-C may not cause side-effects. This has the following consequences:
  - The initialization and iteration phases of `for` loops must be statements, not expressions.
  - Shortcut assignments (e.g. +=) must appear as standalone statements.

2.2.2 Statements in C and Handel-C

In both	Handel-C only
`{;}`	`par`
`switch`	`delay`
`do ... while`	`?`
`while`	`!`
`if ... else`	`prialt`
`for (;;)`	`seq`
`break`	`ifselect`
`continue`	
`return`	
`goto`	
`assert`	`assert` is an expression in Handel-C and not the same as in ISO-C

>: 2. Getting started with Handel-C

2.2.3 C and Handel-C types, type operators and objects

In both	Conventional C only	Handel-C only
`int`	`double`	`chan`
`unsigned`	`float`	`ram`
`char`	`union`	`rom`
`long`		`wom`
`short`		`mpram`
`enum`		`signal`
`register`		`chanin`
`static`		`chanout`
`extern`		`undefined`
`struct`		`interface`
`volatile`		**<>**
`void`		`inline`
`const`		`typeof`
`auto`		
`signed`		
`typedef`		

>: 2. Getting started with Handel-C

2.3 Expressions in C and Handel-C statements

In both	Conventional C only	Handel-C only
`*` (pointer indirection)	`sizeof`	`select(…)`
`&` (address of)		`width(…)`
`-`		`@`
`+`		`\\`
`*` (multiplication)		`<-`
`/`		`[:]`
		`let…in`
`%`		
`<<`		
`>>`		
`>`		
`<`		
`>=`		
`<=`		
`==`		
`!=`		
`&` (bitwise and)		
`^`		
`|`		
`? :`		
`[]`		
`!`		
`&&`		
`~`		
`||`		
`->`		

The following constructs are available as expressions in conventional C and as statements in Handel-C. This means that in Handel-C, they must appear as standalone statements and not in the middle of more complex expressions.

>: 2. Getting started with Handel-C

```
=       +=      -=      *=       /=      %=
<<=     >>=     &=      |=       ^=
++      --
```

2.4 Notes for hardware engineers

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

- Handel-C is halfway between RTL and a behavioural HDL. It is a high-level language that requires you 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. Therefore you must ensure that you disable debug code when you compile 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.

2.5 Basic concepts

Handel-C uses much of the syntax of conventional C with the addition of inherent parallelism. You can write sequential programs in Handel-C, but to gain maximum benefit in performance from the target hardware you must use its parallel constructs. These may be new to some users. If you are familiar with conventional C you will recognize nearly all the other features.

- Handel-C programs
- Parallel programs
- Channel communications
- Scope and variable sharing

2.5.1 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. To execute instructions in parallel, you must use the `par` keyword.

>: 2. Getting started with Handel-C

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.

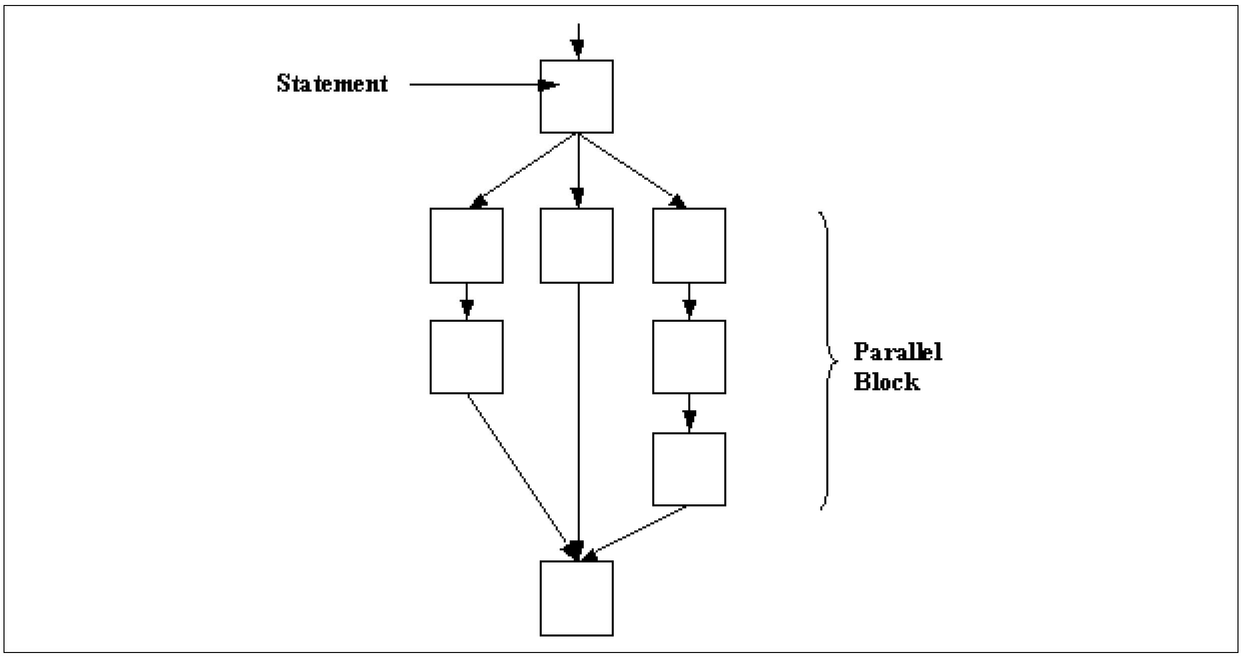
You can express your algorithm in Handel-C without worrying about how the underlying computation engine works. This philosophy makes Handel-C a programming language rather than a hardware description language. In some senses, Handel-C is to hardware what a conventional high-level language is to microprocessor assembly language.

The hardware design that DK1 produces is generated directly from the Handel-C 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.

2.5.2 Parallel programs

The target of the Handel-C compiler is low-level hardware. This means that you get massive performance benefits by using parallelism. It is essential for writing efficient programs to instruct the compiler to build hardware to execute statements in parallel. Handel-C parallelism is true parallelism, not the time-sliced parallelism familiar from general purpose computers. When instructed to execute two instructions in parallel, those two instructions will 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. Any branches that complete early are forced to wait for the slowest branch before continuing.

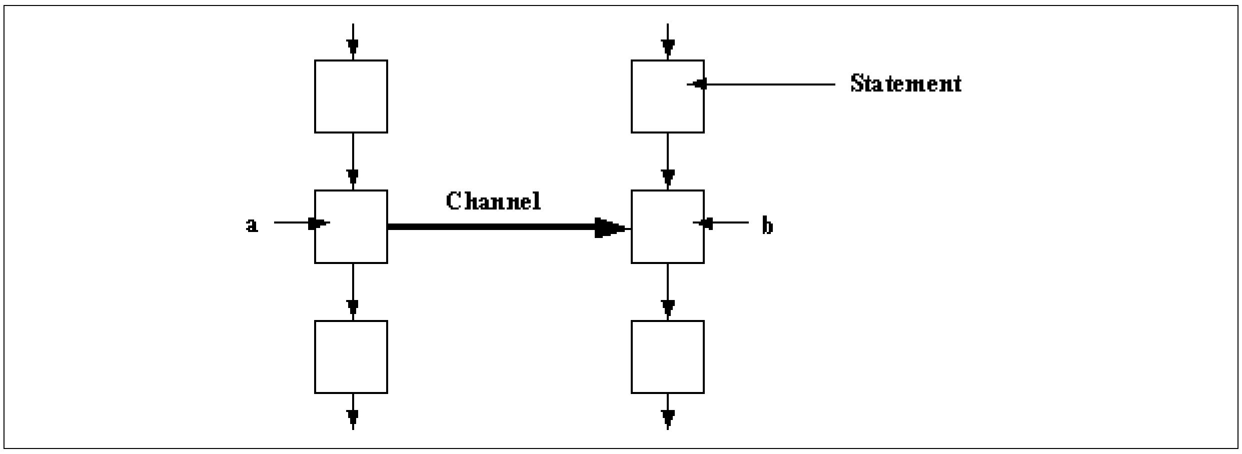


>: 2. Getting started with Handel-C

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

2.5.3 Channel communications: overview

Channels provide a link between parallel branches. One parallel branch outputs data onto the channel and the other branch 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**.

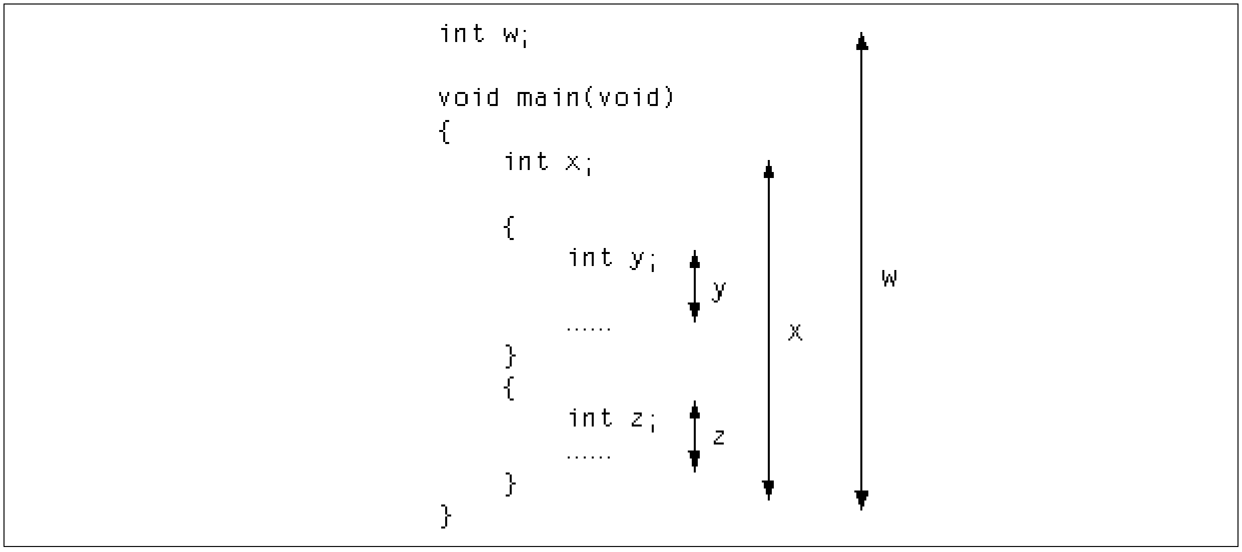
2.5.4 Scope and variable sharing

The scope of declarations is 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.

The scope of variables is illustrated below:

>: 2. Getting started with Handel-C



```
int w;

void main(void)
{
     int x;

     {
          int y;

          ......
     }
     {
          int z;
          ......
     }
}
```

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 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.

If you wish 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 will 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` will be changed by whichever process sends the data first.

A single variable should not be written to by more than one parallel branch.

>: 2. Getting started with Handel-C

2.6 Language basics

Macros and the pre-processor

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.

2.6.1 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 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. This is unlike conventional C, where only one `main` function is permitted. You would only use more than one main function if you needed parts of your program to run at different speeds (and so use different clocks). A `main` function is defined as follows:

>: 2. Getting started with Handel-C

```
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.

2.6.2 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
```

>: 3 Language summary

3.1 Statement summary

Statement	Meaning
`par {...}`	Parallel execution
`seq {...}`	Sequential execution
`par (`*Init* `;` *Test* `;` *Iter*`){...}`	Parallel replication
`seq (`*Init* `;` *Test* `;` *Iter*`){...}`	Sequential replication
Variable `=` *Expression*`;`	Assignment
Variable `++;`	Increment
Variable `--;`	Decrement
`++` *Variable*`;`	Increment
`--` *Variable*`;`	Decrement
Variable `+=` *Expression*`;`	Add and assign
Variable `-=` *Expression*`;`	Subtract and assign
Variable `*=` *Expression*`;`	Multiply and assign
Variable `/=` *Expression*`;`	Divide and assign
Variable `%=` *Expression*`;`	Modulo and assign
Variable `<<=` *Expression*`;`	Shift left and assign
Variable `>>=` *Expression*`;`	Shift right and assign
Variable `&=` *Expression*`;`	Bitwise AND and assign
Variable `|=` *Expression*`;`	Bitwise OR and assign
Variable `^=` *Expression*`;`	Bitwise XOR and assign
Channel `?` *Variable*`;`	Channel input
Channel `!` *Expression*`;`	Channel output
`if (`*Expression*`) {`*statement*`} [else {`*statement*`}]`	Conditional execution
`ifselect (`*Expression*`) {`*statement*`} [else {`*statement*`}]`	Conditional compilation
`while (`*Expression*`) {`*statement*`}`	Iteration
`do {...} while (`*Expression*`);`	Iteration
`for (`*Init* `;` *Test* `;` *Iter*`) {...}`	Iteration

>: 3. Language summary

Statement	Meaning
`break;`	Loop, switch and prialt termination
`continue;`	Resume execution
`return[([`*Expression*`])];`	Return from function
`goto` *label*`;`	Jump to label
`switch (`*Expression*`) {`*statement*`}`	Selection
`prialt {`*statement*`}`	Channel alternation
`releasesema()`	Make semaphore available after use of trysema expression
`try{...} reset(`*Condition*`){`*statement*`}`	Perform statements on reset condition
`delay;`	Single cycle delay

Note: RAM and ROM elements, signals and array elements are included in the set of variables above. However,

```
ram x [3];
x[0]++;
```

is invalid.

The assignment group of operations and the increment and decrement operations are included as statements to reflect the fact that Handel-C expressions cannot contain side effects.

3.2 Operator summary

The following table lists all operators. Entries at the top have the highest precedence and entries at the bottom have the lowest precedence. Entries within the same group have the same precedence. Precedence of operators is as expected from conventional C. For example:

```
x = x + y * z;
```

This performs the multiplication before the addition. Brackets may be used to ensure the correct calculation order as in conventional C.

Note that assignments are not true operators in Handel-C.

Operator	Meaning

>: 3. Language summary

Operator	Meaning
`trysema`	Test if semaphore owned. Take if not
`select(`*Constant*`,` *Expr*`,` *Expr*`)`	Compile-time selection
Expression `[`*Expression*`]`	Array or memory subscripting
Expression `[`*Constant* `]`	Bit selection
Expression `[`*Constant*`:` *Constant*`]`	Bit range extraction. One of the two constants may be omitted (but not both).
functionName `(`*Arguments* `)`	Function call
pointertostructure`->`*member*	Structure reference
structureName`.`*member*	Structure reference
`!` *Expression*	Logical NOT
~ *Expression*	Bitwise NOT
`-` *Expression*	Unary minus
+ *Expression*	Unary plus
& *object*	Yields pointer to operand
`*` *pointer*	Yields object or function that the operand points to
`width(`*Expression*`)`	Width of expression
`(`*Type*`)` *Expression*	Type casting
Expression `<-` *Constant*	Take LSBs
Expression `\\` *Constant*	Drop LSBs
Expression `*` *Expression*	Multiplication
Expression `/` *Expression*	Division
Expression % *Expression*	Modulo arithmetic
Expression + *Expression*	Addition
Expression `-` *Expression*	Subtraction
Expression << *Expression*	Shift left
Expression >> *Expression*	Shift right

>: 3. Language summary

Operator	**Meaning**
Expression @ *Expression*	Concatenation
Expression < *Expression*	Less than
Expression > *Expression*	Greater than
Expression <= *Expression*	Less than or equal
Expression >= *Expression*	Greater than or equal
Expression == *Expression*	Equal
Expression != *Expression*	Not equal
Expression & *Expression*	Bitwise AND
Expression ^ *Expression*	Bitwise XOR
Expression | *Expression*	Bitwise OR
Expression && *Expression*	Logical AND
Expression || *Expression*	Logical OR
Expression ? *Expr* : *Expr*	Conditional selection
`assert`	diagnostic macro to print to stderr

3.3 Type summary

The most common types that may be associated with a variable, and the prefixes for architectural and compound types are listed below:

Common logic types

Type	**Width**
`[signed	unsigned] char`	8 bits
`[signed	unsigned] short`	16 bits
`[signed	unsigned] long`	32 bits
`[signed	unsigned] int`	See *Note 1

>: 3. Language summary

Type	Width
`[signed | unsigned] int n`	n bits
`[signed | unsigned] int undefined`	Compiler infers width
`typeof (Expression)`	Yields type of object

*Note 1: Width will be inferred by compiler unless the 'set intwidth = n' command appears before the declaration.

Architectural types

Prefixes to the above types for different architectural object types are:

Prefix	Object
`chan`	Channel
`chanin`	Simulator channel
`chanout`	Simulator channel
`ram`	Internal or external RAM
`rom`	Internal or external ROM
`signal`	Wire
`wom`	WOM within multi-port memory

Compound types

The compound types are:

Prefix	Object
`struct`	Structure
`mpram`	Multi-port memory

Special types

Type	Object
`interface`	interface to external logic or device
`sema`	Semaphore. Has no width or logic type

`interfaces` connect to logic beyond the Handel-C design, whether on the same or a different device.

>: 4 Declarations

4.1 Introduction to 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.

You 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 ANSI-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.

4.1.1 Handel-C values and widths

A crucial difference between Handel-C and conventional C is Handel-C's ability to handle values of arbitrary width. Since conventional C is targeted at general purpose microprocessors it handles 8, 16 and 32 bit values well but cannot easily handle other widths. When targeting hardware, there is no reason to be tied to these data widths and so Handel-C has been extended to allow types of any number of bits.

Handel-C has also been extended to cope with extracting bits from values and joining values together to form wider values. These operations require no hardware and can provide great performance improvements over software.

>: 4. Declarations

When writing programs in Handel-C, care should be taken that data paths are no wider than necessary to minimize hardware usage. While it may be valid to use 32-bit values for all items, a large amount of unnecessary hardware is produced if none of these values exceed 4 bits.

Care must also be taken that values do not overflow their width. This is more of an issue with Handel-C than with conventional C because variables should be just wide enough to contain the largest value required (and no wider).

4.1.2 String constants

String constants are allowed in Handel-C. A string constant consists of a string of characters delimited by double quotes ("). They will be stored as a null-terminated array of characters (as in ANSI-C). String constants can contain any of the special characters listed below. Arrays and pointers can be initialized with string constants, and string constants can be assigned to pointers. If a string constant is assigned to a pointer, the storage for the string will be created implicitly.

Special characters:

\a	alert
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\\	backslash
\?	question mark
\'	single quote
\"	double quote
\onumber	octal number e.g. \o77
\xnumber	hexadecimal number e.g. \xf3

4.1.3 Constants

Constants may be used in expressions. Decimal constants are written as simply the number while hexadecimal constants must be prefixed with `0x` or `0X`, octal constants must be prefixed with a zero and binary constants must be prefixed with `0b` or `0B`. For example:

```
w = 1234;        /* Decimal     */

x = 0x1234;      /* Hexadecimal */
```

>: 4. Declarations

```
y = 01234;       /* Octal       */

z = 0b00100110;  /* Binary      */
```

The width of a constant may be explicitly given by 'casting'. For example:

```
x = (unsigned int 3) 1;
```

Casting may be necessary where the compiler is unable to infer the width of the constant from its usage.

4.2 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 will attempt to infer a width from use.

Enumeration types (`enums`) allow you 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 you to declare empty parameter lists or functions that do not return a value. The `typeof` type operator allows you to reference the type of a variable.

4.2.1 int

There is only one fundamental type for variables: `int`. By default, integers are signed. The `int` type may be qualified with the `unsigned` keyword to indicate that the variable only contains positive integers or 0. For example:

```
int 5 x;
unsigned int 13 y;
```

These two lines declare two variables: a 5-bit signed integer `x` and a 13-bit non-negative integer `y`. In the second example here, the `int` keyword is optional. Thus, the following two declarations are equivalent.

```
unsigned int 6 x;
unsigned 6 x;
```

You may use the `signed` keyword to make it clear that the default type is used. The following declarations are equivalent.

```
int 5 x;
signed int 5 x;
signed 5 x;
```

>: 4. Declarations

The range of an 8-bit signed integer is -128 to 127 while the range of an 8-bit unsigned integer is 0 to 255 inclusive. This is because signed integers use 2's complement representation.

You may declare a number of variables of the same type and width simultaneously. For example:

```
int 17 x, y, z;
```

This declares three 17-bit wide signed integers x, y and z.

Signed | unsigned syntax

Signed | unsigned is declared in the same way as in ANSI-C except that Handel-C allows the width to be declared. The width may be undefined, an expression, or nothing.

For example:

- `int a;`
- `long b;`
- `unsigned int 7 c;`
- `signed undefined d;`
- `long signed int e;`

Supported types for porting

Handel-C provides support for porting from conventional C by allowing the types `char`, `short` and `long`. For example:

```
unsigned char w;
short y;
unsigned long z;
```

Note that these are fixed-widths in Handel-C, and implementation dependent in ANSI-C. The widths used for each of these types in Handel-C is as follows:

Type	Width
`char`	8 bits (signed)
`short`	16 bits
`long`	32 bits

Smaller and more efficient hardware will be produced by only using variables of the smallest possible width.

>: 4. Declarations

4.2.2 Inferring widths

The Handel-C compiler can infer the width of variables from their usage. It is therefore not always necessary to explicitly define the width of all variables and the `undefined` keyword can be used to tell the compiler to try to infer the width of a variable. For example:

```
int 6 x;
int undefined y;

x = y;
```

In this example the variable x has been declared to be 6 bits wide and the variable y has been declared with no explicit width. The compiler can infer that y must be 6 bits wide from the assignment operation later in the program and sets the width of y to this value.

If the compiler cannot infer all the undefined widths, it will generate errors detailing which widths it could not infer.

The `undefined` keyword is optional, so the two definitions below are equivalent:

```
int x;
int undefined x;
```

Handel-C provides an extension to allow you to override this behaviour to ease porting from conventional C. This allows you to set a width for all variables that have not been assigned a specific width or declared as `undefined`.

This is done as follows:

```
set intwidth = 16;

int x;
unsigned int y;
```

This declares a 16-bit wide signed integer x and a 16-bit wide unsigned integer y. Any width may be used in the `set intwidth` instruction, including `undefined`.

You can still declare variables that must have their width inferred by using the `undefined` keyword. For example:

```
set clock = external "p1";
set intwidth = 27;
```

```
void main(void)
{

  unsigned x;
  unsigned undefined y;
}
```

This example declares a variable x with a width of 27 bits and a variable y that has its width inferred by the compiler. This example also illustrates that the `int` keyword may be omitted when declaring unsigned integers.

You may also set the default width to be undefined:

```
set intwidth = undefined;
```

4.3 Complex types

4.3.1 Arrays

You can declare arrays of variables in the same way that arrays are declared in conventional C. For example:

```
int 6 x[7];
```

This declares 7 registers each of which is 6 bits wide. Accessing the variables is exactly as in conventional C. For example, to access the fifth variable in the array:

```
x[4] = 1;
```

Note that as in conventional C, the first variable has an index of 0 and the last has an index of *n*-1 where *n* is the total number of variables in the array.

When a variable is used as an array index, as is often done when using a for loop, the variable must be declared unsigned.

Multidimensional arrays

You can declare multi-dimensional arrays of variables. For example:

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

This declares 4 * 5 * 6 = 120 variables each of which is 6 bits wide. Accessing the variables is as expected from conventional C. For example:

```
y = x[2][3][1];
```

>: 4. Declarations

Example

This loop initializes all the elements in array `ax` to the value of `index`

```
unsigned int 6 ax[7];
unsigned index;

index=0;
do
{
   ax[index] = (0 @ index);
   index++;
}
while(index <= 6);
```

Note that the width of `index` has to be adjusted in the assignment. This is because its width will be inferred to be 3, from the array dimension (the array has 7 elements, so "index" will only ever need to count as far as 6).

4.3.2 Array indices

When an array is declared, the index has the smallest width possible. For instance, in array[8], the index need only go up to seven and will therefore be a three bit number. If a variable is declared to represent the index, it too will be three bits.

4.3.3 Struct

`struct` defines a data structure; a grouping together of variables under a single name. The format of the structure can be identified by a type name. The variable members of the structure may be of the same or different types. Once a structure has been declared, its type name can be used to define other structures of the same type. Structure members may be accessed individually using the construct

struct_Name`.`*member_Name*

Syntax

A structure type is declared using the format

```
struct [type_Name]
{
   member-list
} [instance_Name {,instance_Name}];
```

member-list is a list of variable definitions terminated by semi-colons.

>: 4. Declarations

The use of *instance_Names* declares variables of that structure type. Alternatively, you may declare variables as follows:

`struct` *type_Name instance_Name*`;`

Storage

- Structures may be passed through channels and signals.
- Structures may be stored in internal memory elements.
- Structures cannot be stored in off-chip RAMs.

If a structure contains a memory element, a channel, or a signal, it cannot be stored in another memory element, it cannot be passed to a function "by value", it cannot be assigned to and it cannot be passed through a channel or a signal.

If a structure contains a memory element, it cannot be assigned (or assigned to) another structure as the assignment cannot be performed in a single clock cycle.

Whole structures may not be sent directly to interfaces.

Example

```
struct human      // Declare human struct type
{
  unsigned int 8 age;   // Declare member types
  int 1 sex;
  char  name[25];
};      // Define human type

struct human sister;
sister.age  = 25;
```

4.3.4 enum

`enum` specifies a list of constant integer values, e.g.

```
enum weekdays {MON, TUES, WED, THURS, FRI};
```

The first name (in this case `MON`) has a value of 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, values increment from the last specified value.

You can declare variables of a specified `enum` type. They are effectively equivalent to `int undefined` or `unsigned undefined`. The signedness is inferred from use.

>: 4. Declarations

To specify enum values

```
enum weekdays {MON = 9, TUES, WED, THURS, FRI};
```

To declare a variable of type `enum`

```
enum weekdays x;
```

(The compiler must be able to infer width of x from its use.)

To assign `enum` values to a variable

```
static int x = MON;
```

Example:

The example below illustrates how to infer the width of an `enum.` The cast ensures the enumerated variable has a width associated with it.

```
set clock = external "P1";
typedef enum
{
   A,
   B,
   C = 43,
   D
} En;


void main(void)
{
   En num;
   int undefined result;

   num = (int 7)D;

   result = num;
}
```

4.3.5 Bit field

A bit field is a type of structure member consisting of a specified number of bits. The length of each field is separated from the field name by a colon (:). Each element can be accessed independently. Since Handel-C allows you to specify the width of integers in bits, a bit field is merely another way of specifying a standard structure. In ANSI-C, bit fields are made up of words, and only the specified bits are accessed, the rest are padded. Padding in ANSI-C is implementation dependent. There is no padding in Handel-C, so nothing can be assumed about it.

Syntax

```
struct [tag_name]
{
   field_Type  field_Name:  field_Width
   ...
} [instance_names] ;
```

Example

This example defines an identical array of flags as a structure and as a bit field

```
struct structure
{
   unsigned int 1 LED;
   unsigned int 1 value;
   unsigned int 1 state;
}outputs;


struct bitfield
{
   unsigned int LED : 1;
   unsigned int value : 1;
   unsigned int state : 1;
}signals;
```

4.4 Pointers

A pointer declaration consists of `*`, the name of the pointer and the type of the variable that it points to.

*type *Name*

They are used to point to variables in conjunction with the unary operator &, which gives the address of an object. To set a pointer to point to a variable, you assign the address of the variable to the pointer. For example

```
int 8 *ptr;      //declare a pointer to an int 8
int 8 object, x;
object = 6;
x = 10;

ptr = &object;   //assigns the address of
                 // object to pointer
x = *ptr;        // x is now 6
*ptr = 12;       //object is now 12
```

In Handel-C, you may only cast void pointers (`void * ` *pointerName*) to a different type. All other pointers may only be cast to change the sign of an object pointed to, and whether it is `const` or `volatile.` These restrictions are the standard casting restrictions in Handel-C.

You can change a void pointer's type by casting, assignment or comparison. Void * must have a consistent type so:

```
void *p;
int 6 *s;
int 7 *t;

p = s;
p = t; //invalid
```

Valid pointer operations are:

- Assign a pointer to another pointer of the same type
- Add or subtract a pointer and an integer
- Subtract or compare a pointer to an array member with another pointer to a member of the same array
- Assign or compare a pointer to NULL

4.4.1 Pointers and addresses

Pointers in Handel-C are similar to those in conventional C. They provide the address of a variable or a piece of code. This enables you to access variables by reference rather than by value.

The indirection operator (`*`) is the same as it is in ANSI-C. It is used to de-reference pointers (i.e. to access objects pointed to by pointers).

The "address of" operator (&) works as it does in ANSI-C.

>: 4. Declarations

4.4.2 Pointers to functions

If you point to code (a function), the address operator is optional. The syntax is

returnType `(`**pointerName*`)(`*parameter list*`);`

The parentheses at the end of the declaration declare the pointer to be a pointer to a function. The * before the *pointerName* declares it to be a pointer declaration.

There is the standard C type ambiguity between the declaration of a function returning a pointer and a pointer to a function. To ensure that * is associated with the pointer name rather than the return type, you need to use parentheses

```
int 8 * functionName(); //function returning pointer
```

and

```
int 8 (* pointerName)(); //pointer to function
```

4.4.3 Pointers to interfaces

When declaring pointers to interfaces, you must ensure that you declare a pointer to an interface sort and then assign a defined interface to it (much as when you declare a pointer to a function). You cannot combine the definition of an object with the declaration of a pointer to it.

The members of the interface must have the same name in the declaration of the pointer type as in the definition of the interface object which you assign the pointer to.

Example

```
//declaration of pointer to interface of sort bus_out

interface bus_out() *p(int 2 x);
interface bus_out() b(int 2 x=y);  //interface definition
p=&b;                              // p now points to b
```

4.4.4 Structure pointers

The structure pointer operator (->) can be used, as in ANSI-C. It is used to access the members of a structure, when the structure is referenced through a pointer.

>: 4. Declarations

```
struct S
{
   int 18 a, b;
} s, *sp;

sp = &s;
s.a = 26;
sp->b = sp->a;
```

The last line accesses the member variables of structure s through pointer sp. Because the pointer is being used to access the structure, the -> operator is used to refer to the member variables.

```
sp->a = (*sp).q
```

You can cast structure pointers between structures with the same member types and names. For example:

```
struct S1
{
  int 6 x;
} st1;

struct S2
{
  int 6 x;
} st2;

set clock = external;

void main (void)
{
   int r;

   struct S1 *structPtr1;
   struct S2 *structPtr2;

   structPtr1 = &st1;
   structPtr2 = (struct S2 *)structPtr1;

   structPtr2->x = 7;

   r = st1.x;  //r = 7
}
```

4.4.5 * operator / & operator

The indirection operator `*` is the same as it is in ANSI C. It is used to de-reference pointers (i.e. to access objects pointed to by pointers).

The address operator (&) works as it does in ANSI-C.

The following can also be used: pointers to arrays, pointers to channels, pointers to signals, pointers to memory elements, pointers to structures and unions, pointers to pointers, arrays of pointers.

Example: pointer assignment

```
unsigned char cha, chb, *chp;

chp = &cha;
cha = 90;

chb = *chp;
chp = &chb;
```

The first line declares two `unsigned` variables (`cha` and `chb`), and a pointer to an `unsigned` (`chp`). The second line assigns the address of `cha` to pointer `chp`. In other words, pointer `chp` now points to variable `cha`. The third line simply assigns a value to `cha`. The fourth line dereferences pointer `chp`, to access what it's pointing to, which is `cha`. In other words, `chb` is assigned the value of the object pointed to by `chp`. The last line assigns the address of `chb` to pointer `chp`. In other words, pointer `chp` now points to variable `chb`.

Example: pointer to pointer assignment

```
struct S
{
   int 6 a, b;
} s1, s2, *sp, **spp;

sp = &s1;
spp = &sp;
s2 = **spp;
```

This declares two variables of type `struct S` (`s1` and `s2`), a pointer to a variable of this type (`sp`), and a pointer to a pointer to a variable of this type (`spp`). The next line assigns the address of structure `s1` to pointer `sp` (pointer `sp` to point to structure `s1`). The following line assigns the address of pointer `sp` to pointer `spp` (pointer `spp` to point to pointer `sp`). The last line dereferences pointer `spp` twice, and it assigns the dereferenced value, which is `s1`, to structure `s2` (i.e. `s2` now equals `s1`).

>: 4. Declarations

4.5 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`)
- `signal` (declares a wire).

The disambiguator < > has been provided to help clarify the definitions of memories, channels and signals.

4.6 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 or if the width is omitted. Channels can also be declared with no explicit type. The compiler infers the type and width of the channel from its usage. For example:

```
set intwidth = undefined;

chan int Link1;
chan unsigned undefined Link2;
chan Link3;
```

Syntax

`chan [` *logicType* `]` *Name*`;`

4.6.1 Arrays of channels

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

>: 4. Declarations

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

This is equivalent to declaring 6 channels each of which is 5 bits wide. A channel can be accessed by specifying its index. As with variable arrays, the index for the *n*th element is *n*-1. For example:

```
x[4] ! 3; // Output 3 on channel x[4]
x[3] ? y; // Input to y from channel x[3]
```

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. Accessing the channels is similar to accessing arrays in conventional C. For example:

```
x[2][3][1] ! 4; // Output 4 on channel
```

4.7 Interfaces: overview

All interfaces, except for external (foreign code or off-chip) RAMs are declared with the `interface` keyword in Handel-C. Interfaces are used to communicate with:

- external devices
- external logic, such as other Handel-C programs, programs written in VHDL etc.

You can communicate between blocks of internal logic using channels

The interface definition is in two parts:

- an interface sort: the name of the black box or primitive that the interface connects to
- an instance name: the name of the instance of the interface sort in Handel-C

Interface definitions may be split into declarations and definitions. You must use a declaration if you want to define multiple instances of the same interface sort, or to use forward references.

The declaration gives the sort name and port names and types associated with that interface sort.

The definition gives the instance name, object specifications and the data transmitted for a single instance of the interface sort.

Only `signed` and `unsigned` types may be passed over interfaces.

>: 4. Declarations

4.7.1 Interface definition

A Handel-C `interface` definition consists of an interface sort, an instance name and data ports, together with information about each port.

The definition defines a single instance of an interface sort. If you want to define multiple instances, or use forward references to the interface, declare the interface, and then make multiple definitions of that interface sort. (You do not need to declare interfaces of predefined sorts.)

The general format of an interface definition is:

```
interface   Sort (ports_in_to_Handel-C)
            InstanceName (ports_out_from_Handel-C )
        with {GeneralSpecs};
```

Sort	Pre-defined interface sort, or used-defined sort. (This should match the sort in the interface declaration, if you are using one.)
ports_in_to_Handel-C	Definitions of one or more ports bringing data into the Handel-C code. (Port definitions are described below.)
InstanceName	User-defined identifier for that instance of the interface. (You can define any number of instances of an interface sort, if you make a declaration of the interface sort.)
ports_out_from_Handel-C	Definitions of one or more ports sending data from the Handel-C code. Each output port should be assigned an expression. The value of the expression will be connected to that port.
GeneralSpecs	Handel-C interface specifications. These specify hardware details of the interface, such as chip pin numbers or are used to specify an external simulator using the `extlib` directive. Interface specifications apply to all ports in the interface. You can also assign specifications to individual ports.

Port definitions

If the interface has been previously declared, the port definitions must be prototyped in their interface declaration, and must have the same types as those in the prototype. The declaration must have at least one port into Handel-C or from Handel-C. Port definitions are delimited by commas. Each port definition consists of:

- the data type that uses it (either defined or inferred from its first use). Only `signed` and `unsigned` types may be passed over interfaces.
- a port name

- port specifications (optional). The port specifications are enclosed in a set of braces {...} and delimited by commas.

Example

```
interface Sort_A (int 4 inPort1, int 4 inPort2)
  interfaceName (unsigned outPort = x)
```

4.7.2 Interface declaration

You need to use an interface declaration if you want to define multiple instances of an interface sort, or to use forward references. If you only want a single instance of an interface sort, you only need to use an interface definition.

Interfaces of pre-defined sorts do not need to be declared.

The general format of the interface declaration is:

```
interface Sort (ports_in_to_Handel-C)
   (ports_out_from_Handel-C);
```

Sort	user-defined name or predefined interface sort
ports_in_to_Handel-C	Optional. One or more prototypes of ports bringing data **into** the Handel-C code.
ports_out_from_Handel-C	Optional. One or more prototypes of ports sending data **from** the Handel-C code.

A port prototype consists of the port type, and the port name. At least one port (whether to Handel-C or from Handel-C) must be declared. Port declarations are delimited by commas. For example:

```
interface MyInterface (int 5 InPort)
   (int 4 OutPort1, int 4 OutPort2);
```

The name of each port in a `port_in` or `port_out` interface must be different, as they will all be built to the top level of the design.

Once you have declared an interface sort, you can define multiple instances of that sort. The interface definition creates a named instance of the interface sort, assigns data to be transmitted to the output ports, and may also specify properties for individual ports**.**

You can declare pointers to an interface declaration and then assign a defined interface to the pointer.

>: 4. Declarations

Old-style declaration-definitions

The style of interface declaration used in Handel-C Version 2 (which omitted port prototypes) is deprecated, but remains for backward compatibility.

4.7.3 Example interface to external code

This example shows an interface declaration used to connect to a piece of foreign code, and the definition that uses this declaration.

```
set clock = external "D17";
set family = XilinxVirtex;
set part = "V1000BG560-4";


// Interface declaration
interface ttl7446(unsigned 7 segments, unsigned 1 rbon)
        (unsigned 1 ltn, unsigned 1 rbin, unsigned 4 digit,
           unsigned 1 bin);

unsigned 1 ltnVal;
unsigned 1 rbinVal;
unsigned 1 binVal;
unsigned 4 digitVal;


// Interface definition
  interface ttl7446(unsigned 7 segments, unsigned 1 rbon)
            decode(unsigned 1 ltn=ltnVal, unsigned 1 rbin=rbinVal,
                   unsigned 4 digit=digitVal, unsigned 1 bin=binVal)
              with {extlib="PluginModelSim.dll",
         extinst="decode; model=ttl7446_wrapper; delay=1"};
```

This declares an interface of sort `tt17446`. The inputs from the interface to the Handel-C design are `segments` and `rbon`. The interface would therefore connect to a black box named `tt17746` with ports `segments`, `rbon`, `ltn`, `rbin`, `digit`, and `bin`.
The instance of the interface is `decode`. The instance specifies the data going into the ports `ltn`, `rbin`, `digit`, and `bin` and connects to a plugin, `PluginModelSim.dll`, for simulation.
If you did not want to use forward references to the interface, and only wanted to define a single instance of the interface sort `tt17446`, you would not need to declare the interface. (The interface definition would be exactly the same as that shown above.)

>: 4. Declarations

4.7.4 Interface specifications

Predefined bus interface specs:	**Meaning:**	**Default:**
`data`	list the pins used for transferring data, MSB to LSB	None
`speed`	set buffer speed (output)	3: Xilinx 4000 series 2: Actel ProASIC/ProASIC+ 1: others
`pull`	set pull-up or pull-down for bus pins	None
`infile`	set file source for input bus data	None
`outfile`	set file destination for output bus data	None

All interface specs:	**Meaning:**	**Default:**
`base`	specify display base for variables in debugger	10
`bind`	bind component to work library	0
`busformat`	text format of exported wires in EDIF netlist	"B_I"
`data`	list the pins used for transferring data, MSB to LSB	None
`dci`	apply Digital Controlled Impedance to buses (Xilinx only)	0 (No)
`extlib`	specify external plugin for simulator	None
`extfunc`	specify external simulator function for this port	`PlugInSet` or `PlugInGet`
`extpath`	specify any direct logic (combinational logic) connections to another port	None
`extinst`	specify connection to external code	None
`intime`	maximum allowable time between a port and the	None

All interface specs:	Meaning:	Default:
	sequential elements it drives (in ns)	
`outtime`	maximum allowable time between a port and the sequential elements it is driven from (in ns)	None
`properties`	parameterize instantiations of external black boxes	None
`standard`	specify I/O standard (electrical characteristics) to use on port(s) in question	LVCMOS33 for Actel ProASIC/ProASIC+, LVTTL for others
`std_logic_vector`	specify `std_logic_vector` port in `port_in, port_out` or generic interface	0
`strength`	specify drive strength (in mA) for output buses	Standard dependent
`warn`	disable some compiler warnings	1 (No)

4.8 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];
static 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.

ROMs must be declared as static or global. RAMs can be declared as static, global or auto (i.e. non-static).

All RAMs and ROMs must be declared as arrays, so to declare a RAM that holds one 4 bit integer, you must declare it as an array with a dimension of 1.

```
ram int 4 ramname[1];
```

RAMs and ROMs may only have one entry accessed in any clock cycle.

>: 4. Declarations

Initialization

Only static or global ROMs or RAMs can be initialized. For example, a global ROM could be initialized as shown below:

```
rom int 16 b[4] = { 23, 46, 69, 92 } with {block = 1};
```

The ROM is initialized with the constants given in the following list in the same way as an array would be initialized in C. In this example, the ROM entries are given the following values:

ROM entry	Value
b[0]	23
b[1]	46
b[2]	69
b[3]	92

Inferring size from use

The Handel-C compiler can also infer the widths, types and the number of entries in RAMs and ROMs from their usage. Thus, it is not always necessary to explicitly declare these attributes. For example:

```
ram int undefined a[123];
ram int 6 b[];
ram c[43];
ram d[];
```

RAMs and ROMs are accessed in the same way as arrays. For example:

```
ram int 6 b[56];

b[7] = 4;
```

This sets the eighth entry of the RAM to the value 4. Note that as in conventional C, the first entry in the memory has an index of 0 and the last has an index of n-1 where n is the total number of entries in the memory.

Differences between RAMs and arrays

RAMs differ from arrays in that an array is equivalent to declaring a number of variables. Each entry in an array may be used exactly like an individual variable, with as many reads, and as many writes to a different element in the array as required within a clock cycle. RAMs, however, are normally more efficient to implement in terms of hardware resources than arrays, but they only allow one location to be accessed in any one clock cycle. Therefore, you should use an array when you wish to access the elements more than once in parallel and you should use a RAM when you need efficiency.

>: 4. Declarations

Devices

Creating internal RAMs can only be done if the target device supports on-chip RAMs. Most devices currently targeted by Handel-C do so (e.g. Altera Flex 10K, APEX, APEXII and Mercury, Xilinx 4000E, 4000EX, 4000L, 4000XL, 4000XV, Spartan, Spartan II and Virtex series devices).

No Actel families support ROMs. Only ProASIC and ProASIC+ Actel devices support RAMs. RAMs on Actel devices may not be initialized.

4.8.1 Multidimensional memory arrays

You can 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 <>.

Syntax

`ram | rom | wom` *logicType entry_width Name*`[[`*const_expression*`]]`
`{[[`*const_expression*`]]}`
`[= {`*initialization*`}];`

Possible logic types are `ints`, `structs`, pointers and arrays.

The last constant expression is the index for the RAM. The other indices give the number of copies of that type of RAM.

Example

```
ram <int 6> a[15][43];
static rom <int 16> b[4][2][2] =
   {  {{1, 2},
       {3, 4}
      },
      {{5, 6},
       {7, 8}
      },
      {{9, 10},
       {11, 12}
      },
      {{13, 14},
       {15, 16}
      }
   };
```

This example constructs 15 RAMs, each consisting of 43 entries of 6 bits wide and 4 * 2 ROMs, each consisting of 2 entries of 16 bits wide. The ROM is initialized with the constants in the following list in the same way as a multidimensional array would be initialized in C.

>: 4. Declarations

The last index (that of the RAM entry) changes fastest. In this example, the ROM entries are given the following values:

ROM entry	Value	ROM entry	Value
`b[0][0][0]`	1	`b[0][0][1]`	2
`b[0][1][0]`	3	`b[0][1][1]`	4
`b[1][0][0]`	5	`b[1][0][1]`	6
`b[1][1][0]`	7	`b[1][1][1]`	8
`b[2][0][0]`	9	`b[2][0][1]`	10
`b[2][1][0]`	11	`b[2][1][1]`	12
`b[3][0][0]`	13	`b[3][0][1]`	14
`b[3][1][0]`	15	`b[3][1][1]`	16

Because of their architecture, RAMs and ROMs are restricted to performing operations sequentially. Only one element of a RAM or ROM may be addressed in any given clock cycle and, as a result, familiar looking statements are often disallowed. For example:

```
ram <unsigned int 8> x[4];
x[1] = x[3] + 1;
```

This code is inadvisable because the assignment attempts to read from the third element of `x` in the same cycle as it writes to the first element.

In a multi-dimensional array, you can access separate elements of the arrays, so long as you are not accessing the same RAM. For example:

`x[2][1]=x[3][0]` is valid

`x[2][1]=x[2][0]` is invalid

Note that arrays of variables do not have these restrictions but may require substantially more hardware to implement than RAMs depending on the target architecture.

4.9 mpram (multi-ported RAMs)

You can create multiple-ported RAMs (MPRAMs) by constructing something similar to an ANSI-C union. You must use the `mpram` keyword.

`mprams` can be used to connect two independent code blocks. The clock of the `mpram` port is taken from the function in which it is used.

The normal declaration of a MPRAM would be to create a dual-ported RAM by declaring two ports of equal width:

- for Altera ApexII or Mercury, both ports can be bi-directional. For other Altera families, one port would be read-only and one write-only
- for Xilinx 4000, one port would be read/write and one read-only
- for Virtex, both ports would be read/write for block RAM, and for LUT RAM, one port would be read/write and one read-only.
- Altera Mercury devices can have up to four ports. You can have (one or two write ports AND one or two read ports) OR two read/write ports. Depending on how you have configured the port, you can have up to four simultaneous accesses of the same block of memory.
- for Actel devices, one port must be read-only, and one write-only.

The `mpram` construct allows the declaration of any number of ports. Your only restriction is the target hardware.

Syntax

`mpram` *MPRAM_name*
`{`
    ram_Type variable_Type RAM_Name`[`*size*`];`
    ram_Type variable_Type RAM_Name`[`*size*`];`
`};`

4.9.1 Initialization of mprams

The first member of the `mpram` can be initialized.

```
static mpram Fred
{
   ram <unsigned 8> ReadWrite[256];   // Read/write port
   rom <unsigned 8> Read[256];      // Read only port
} Mary ={10,11,12,13};
```

This would have the same effect as

```
Mary.ReadWrite[0]=10;
Mary.ReadWrite[1]=11;
Mary.ReadWrite[2]=12;
Mary.ReadWrite[3]=13;
```

The other elements of `Fred.ReadWrite` will be initialized as zero (since `Mary` is `static`). In this case, since `Fred.Read` is the same size as `Fred.ReadWrite`, elements 0 – 3 of `Fred.Read` would be initialized with the same values.

>: 4. Declarations

4.9.2 Mapping of different width mpram ports

If the ports of the mpram are of different widths, they will be mapped onto each other according to the specifications of the chip you are using. If the ports used are of different widths, the widths should have values of 2^n.

Different width ports are available for Xilinx Virtex and SpartanII devices and Altera Apex II devices. They are not available with other Altera devices, Xilinx 4000 series devices or Actel devices.

Xilinx bit mapping

To find the bits that an array element occupies in a Xilinx Virtex or SpartanII RAM, you can use the formula
RAM array `ram` *y* `Name[`*a*`]` will have a start bit of (*y* * (*a*+1)) - 1 and an end bit of *y* * *a* .

Xilinx mapping is little-endian. This means that the address points to the LSB.

The bits between the declarations of RAM are mapped directly across, so that bit 27 in one declaration will have the same value as bit 27 in another declaration, even though the bits may be in different array elements in the different declarations.

```
mpram Joan
{
   ram <unsigned 4> ReadWrite[256];   // Read/write port
   rom <unsigned 8> Read[256];     // Read only port
};
```

`Joan.ReadWrite[100]` will run from 400 to 403.

`Joan.Read[100]` will run from 800 to 807.

`Joan.Read[50]` will run from 400 to 407.

`Joan.ReadWrite[100]` is equivalent to `Joan.Read[50][0:3].`

ApexII bit mapping

To find the bits that an array element occupies in an ApexII RAM, you can use the formula
RAM array `ram` *y* `Name[`*a*`]` will have a start bit of (*y* * (*a*+1)) - 1 and an end bit of *y* * *a* .

ApexII mapping is little-endian. This means that the address points to the LSB.

The bits between the declarations of RAM are mapped directly across, so that bit 27 in one declaration will have the same value as bit 27 in another declaration, even though the bits may be in different array elements in the different declarations.

```
mpram Joan
{
   ram <unsigned 4> ReadWrite[256];   // Read/write port
   rom <unsigned 8> Read[256];      // Read only port
};
```

`Joan.ReadWrite[100]` will run from 400 to 403.

`Joan.Read[100]` will run from 800 to 807.

`Joan.Read[50]` will run from 400 to 407.

`Joan.ReadWrite[100]` is equivalent to `Joan.Read[50][0:3]`.

4.9.3 mprams example

Using an `mpram` to communicate between two independent logic blocks:

File 1:

```
mpram Fred
{
   ram <unsigned 8> ReadWrite[256];   // Read/write port
   rom <unsigned 8> Read[256];      // Read only port
};

mpram Fred Joan ;  /*Declare Joan as an mpram like Fred */

set clock = internal "F8M";

void main(void)
{
 unsigned 8 data;

 Joan.ReadWrite[7] = data;
}
```

File 2:

```
mpram Fred
{
   ram <unsigned 8> ReadWrite[256];   // Read/write port
   rom <unsigned 8> Read[256];     // Read only port
};

extern mpram Fred Joan;
set clock = external "P2";

void main(void)
{
 unsigned 8 data;

 data= Joan.Read[7];
}
```

4.9.4 WOM (write-only memory)

You can declare a write-only memory using the keyword `wom`. The only use of a write-only memory would be to declare an element within a multi-ported RAM. Since `wom`s only exist inside multi-port rams, it is illegal to declare one outside a `mpram` declaration.

Syntax

`wom` *variable_Type variable_Size WOM_Name*`[`*dimension*`] =` *initialize_Values*
        `[ with {`*specs*`}]`

Example

```
mpram connect
{
   wom <unsigned 8> Writeonly[256];  // Write only port
   rom <unsigned 8> Read[256];       // Read only port
}
```

>: 4. Declarations

4.10 Other architectural types

4.10.1 sema

Handel-C provides semaphores for protecting critical areas of code. Semaphores are declared with the `sema` keyword. For example:

```
sema RAMguard;
```

Semaphores have no type or width associated with them. They cannot be assigned to or have their value assigned to anything else. You can only access semaphores through the `trysema(`*semaphore* `)` expression and `releasesema(`*semaphore*`)` statement. `trysema` tests to see if the semaphore is currently taken. If it is not, it takes the semaphore and returns one. If it is taken, it returns zero. `releasesema` releases the semaphore. After you have taken a semaphore, you should ensure that you release it cleanly once you have left the critical area.

Semaphores may be included in structures. They cannot be passed to directly to functions, over channels or interfaces. They may be passed to functions or channels by reference.

Syntax

`sema` N*ame*

Example

```
inline void critRAMaccess(sema *RAMsema, ram int 8 (*danger)[4],
                          unsigned count)
{
   int 8 x;
   while(trysema(*RAMsema)==0) delay; // wait till you've got the
                                       // RAM
   x= (*danger)[count];
   releasesema(*RAMsema);
}
```

4.10.2 signal

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 initialization value. 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, you get 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 cycle before the last line, a is finally assigned the signal's initial value of 690.

4.11 Storage class specifiers

Storage class specifiers define how variables are accessed.

`extern` and `static` are used within functions to allocate storage. `static` gives the declared objects static storage class, and `extern` specifies that the variable is defined elsewhere. For compatibility with ANSI-C, the specifiers `auto` and `register` can be used but have no effect.

The expansion of a function is defined by the specifier `inline`.

The `typedef` specifier does not reserve storage, but allows you to declare new names for existing types.

4.11.1 auto

`auto` defines a local automatic variable. In Handel-C, all local variables default to `auto`. You cannot initialize an `auto` variable, but must assign it a value. The initialization status of `auto` variables is undefined.

Example

```
set clock = external "P1";

void main (void)
{
   auto 8 pig;
   pig = 15;
}
```

4.11.2 extern (external variables)

`extern` declares a variable that is external to all functions; the variable may be accessed by name from any function.

External variables must be defined exactly once outside any function, and declared in each function that wants to access them. The declaration may be an explicit `extern` , or else be implicit from the context (if the variable has been defined outside a function without `static`).

If the variable is used in multiple source files, it is good practice to collect all the extern declarations in a header file, included at the top of each source file using the `#include` *headerFileName* directive.

You may use `extern "`*language*`"` to access variables in C or C++ files.

You cannot access the same variable from different clock domains.

>: 4. Declarations

Example

```
extern int 16 global_fish;
int global_frog = 1234;


main()
{
  global_fish =  global_frog;
  …
}
```

Syntax

`extern` *variable declaration*;

4.11.3 extern language construct

The `extern "`*language*`"` construct allows you to declare that names used in Handel-C code have ANSI-C or C++ linkage.

- For ANSI-C functions, use `extern "C"`
- For C++ functions, use `extern "C++"`

These functions can only be compiled for simulation. They may not be used in targeting devices.

Examples

```
extern “C” int printf(const char *format, ...);
```

declares `printf()` with C linkage.

```
extern “C++”
{
   int 14 x;
}
```

declares a variable, x, with C++ linkage.

```
extern "C"
{
   #include <stdio.h>
}
```

causes everything in `stdio.h` to have C linkage.

>: 4. Declarations

Mapping of types to C/C++

Handel-C types will be mapped to C/C++ types in the following way when inside an `extern` "*language*" construct:

Handel-C type	C/C++ type
`char`	`char`
`short`	`short`
`long`	`long`
`int`	`int` (only valid within an `extern` "*language*" construct)
`int` *width*	`Int`<*width*> (C++ only)
`unsigned int` *width*	`UInt`<*width*> (C++ only)
`struct`	`struct`
type `ram[`*n*`]`	*convertedType*`[`*n*`]`
type `rom[`*n*`]`	*convertedType*`[`*n*`]`
Others	Generates an error

Mapping of types outside extern

Mapping of types outside the `extern` "*language*" construct is the same, except signed and unsigned `int`s must have a specified width.

When outside an `extern` "*language*" construct, an `int` without a specified width will generate an error.

For example, the following Handel-C:

```
extern “C” int printf(const char *format, ...);
extern “C++”
{
   int 14 x;
   long y;
}
char f(long y);  //outside extern construct
```

will map to this C++:

```
int printf(const char *format, ...);
Int<14> x;
long y;
char f(long y);
```

>: 4. Declarations

4.11.4 register

`register` has been implemented for reasons of compatibility with ANSI-C. `register` defines a variable that has local scope. Its initial value is undefined.

Example

```
register int 16 fish;
fish = f(plop);
```

4.11.5 inline

`inline` causes a function to be expanded where it is called. The logic will be generated every time it is invoked. This ensures that the function is not accessed at the same time by parallel branches of code.

If you have a local static variable in an inline function there is one copy of the variable per function instantiation.

By default, functions are assumed to be shared (not inline).

Example

```
inline int 4 knit(int needle, int stitch)
{
  needle = needle + stitch;
  return(needle);
}


int 4 jumper[100];
par(needle = 1; needle < 100; needle = needle+2)
 {
  jumper[needle] = knit(needle, 1);
 }
```

Syntax

`inline` *function_Declaration*

4.11.6 static

`static` gives a variable static storage (its values are kept at all times). This ensures that the value of a variable is preserved across function calls. It also affects the scope of a variable or a function. `static` functions and `static` variables declared outside functions can only be used in the file in which they appear. `static` variables declared within an `inline` function or an array of functions can only be used in the copy of the function in which they appear. Handel-C uses `static` in a different way to C++. In C++, if you have an inline function and a local static variable, one copy of the variable is shared across each function instantiation. In Handel-C, there is one copy of the variable per function instantiation.

`static` variables are the only local variables (excluding `consts`) that can be initialized. To get a default value, initialize the variable.

Example

```
static int 16 local_function (int water, int weed);
static int 16 local_fish = 1234;

void main(void)
{
  int fresh, pondweed;
 local_fish = local_function(fresh, pondweed);
 ...
}
```

Syntax

```
static variable_declaration;
static functionName(parameter-type-list);
```

Static variables in arrays of functions

If a static variable is declared in an arrayed function, each instance of the function will have its own independent copy of the variable.

4.11.7 typedef

`typedef` defines another name for a variable type. This allows you to clarify your code. The new name is a synonym for the variable type.

```
typedef int 4 SMALL_FISH;
```

If the `typedef` is used in multiple source files, it is good practice to collect all the type definitions in a header file, included at the top of each source file using the `#include` *headerFileName* directive. It is conventional to differentiate `typedef` names from standard variable names, so that they are easily recognizable.

Example

```
typedef int 4 SMALL_FISH;
```

```
extern SMALL_FISH stickleback;
```

4.12 typeof

The `typeof` type operator allows the type of an object to be determined at compile time. The argument to `typeof` must be an expression. Using `typeof` ensures that related variables maintain their relationship. It makes it easy to modify code by simplifying the process of sorting out type and width conflicts.

A `typeof`-construct can be used anywhere a type name could be used. For example, you can use it in a declaration, in casts.

Syntax

`typeof (` *expression* `)`

Example

```
unsigned 9 ch;
typeof(ch @ ch) q;
struct
{
   typeof(ch) cha, chb;
} s1;

typeof(s1) s2;

ch = s1.cha + s2.chb;
q = s1.chb @ s2.cha;
```

If the width of variable `ch` were changed in this example, there would be no need to modify any other code.

This is also useful for passing parameters to macro procedures. The code below shows how to use a `typeof` definition to deal with multiple parameter types.

```
macro proc swap (a, b)
{
      typeof(a) t;
      t=a;
      a=b;
      b=t;
}
```

4.12.1 const

`const` defines a variable or pointer or an array of variables or pointers that cannot be assigned to. This means that they keep the initialization value throughout. They may be initialized in the declaration statement. The `const` keyword can be used instead of `#define` to declare constant values. It can also be used to define function parameters which are never modified. The compiler will perform type-checking on `const` variables and prevent the programmer from modifying it.

Example 1

```
const int i = 5;

i = 10; // Error
i++;       // Error
```

Example 2

```
const int *const p;

p = p + 1;    // Error
*p = 3;       // Error
```

4.12.2 volatile

In ANSI-C, `volatile` is used to declare a variable that can be modified by something other than the program.

It is mostly used for hard-wired registers. `volatile` controls optimization by forcing a re-read of the variable. It is only a guide, and may be ignored. The initial value of `volatile` variables is undefined.

Handel-C does nothing with `volatile`. It is accepted for compatibility purposes.

4.13 Complex declarations

It is possible to have extremely complex declarations in Handel-C. You can combine arrays of functions, structs, arrays, and pointers with architectural types. To clarify such expressions, it is wise to use `typedef`.

4.13.1 Macro expressions in widths

If you use a macro expression to provide the width in a type declaration, you must enclose it in parentheses. This ensures that it will be correctly parsed as a macro.

>: 4. Declarations

```
int (mac(x)) y;
```

To declare a pointer to a function returning that type, you get

```
int (mac(x)) (*f)();
```

4.13.2 <> (type clarifier)

< > is a Handel-C extension used to disambiguate complex declarations of architectural types. You cannot use it on logic types. It is good practice to use it whenever you declare channels, memories or signals, to clarify the format of data passed or stored in these variables.

It is required to disambiguate a declaration such as:

```
chan int *x; //pointer to channel or
             //channel of pointers?
```

This should be declared as

```
chan <int *> x; //channel of pointers
```

or

```
chan <int> *x; //pointer to channel
```

Example

```
struct fishtank
{
  int 4 koi;
  int 8 carp;
  int 2 guppy;
} bowl;

signal <struct fishtank> drip;
chan <int 8 (*runwater)()> tap;
```

4.13.3 Using signals to split up complex expressions

You can use signals to 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 you to re-use sub-expressions:

```
unsigned 15 a, b;
signal sig1;

par
{
   sig1 = x + 2;
   a = sig1 * 3;
   b = sig1 / 2;
}
```

4.14 Variable initialization

Global, static and const variables

Global variables (i.e. those declared outside all code blocks) may be initialized with their declaration. For example:

```
static int 15 x = 1234;

static int 7 y = 45 with {outfile = "out.dat"};
```

Variables declared within functions can only be initialized if they have `static` storage or are `consts`.

Global and static variables may only be initialized with constants. If you do not initialize them, they will have a default value of zero.

Global and static variables are constructed from registers in the FPGA or PLD and are initialized with values from the configuration file when the device is programmed. If you use the `set reset` construct, variables will be reset to their initial values. If you use the try...reset construct, variables will not be re-initialized.

All other variables

You cannot initialize non-static local variables. Instead, you must use an explicit sequential or parallel list of assignments following your declarations to achieve the same effect. For example:

```
{
    int 4 x;
    unsigned 5 y;

    x = 5;
    y = 4;
}
```

Local non-static variables have no default initial value; you must explicitly assign them to zero or another value.

Simulation

In simulation, variables (including static variables inside functions) are initialized before the simulation run begins (i.e. before the first clock cycle is simulated).

>: 5 Statements

5.1 Sequential and parallel execution

Handel-C implicitly executes instructions sequentially. When targeting hardware it is extremely important to use parallelism. For this reason, Handel-C has a parallel composition keyword `par` to allow statements in a block to be executed in parallel.

Three assignments that execute in parallel and in the same clock cycle:

```
par
{
    x = 1;
    y = 2;
    z = 3;
}
```

Three assignments that execute sequentially, requiring three clock cycles:

```
x = 1;
y = 2;
z = 3;
```

The `par` example executes all assignments literally in parallel. Three specific pieces of hardware are built to perform these three assignments. This is about the same amount as is needed to execute the assignments sequentially.

Sequential branches

Within parallel blocks of code, sequential branches can be added by using a code block denoted with the `{...}` brackets instead of a single statement. For example:

```
par
{
    x = 1;
    {
        y = 2;
        z = 3;
    }
}
```

In this example, the first branch of the parallel statement executes the assignment to `x` while the second branch sequentially executes the assignments to `y` and `z`. The assignments to `x` and `y` occur in the same clock cycle, the assignment to `z` occurs in the next clock cycle.

>: 5. Statements

The instruction following the `par {...}` will not be executed until all branches of the parallel block complete.

5.1.1 seq

To allow replication, the `seq` keyword exists. Sequential statements can be written with or without the keyword.

The following example executes three assignments sequentially:

```
x = 1;
y = 2;
z = 3;
```

as does this:

```
seq
{
    x = 1;
    y = 2;
    z = 3;
}
```

5.1.2 Replicated par and seq

You can replicate `par` and `seq` blocks by using a counted loop (a similar construct to a `for` loop). The count is defined with a start point (*index_Base* below), an end point (*index_Limit*) and a step size (*index_Count*). The body of the loop is replicated as many times as there are steps between the start and end points. If it is a `par` loop, the replicated processes will run in parallel, if a `seq`, they will run sequentially.

Syntax

```
par | seq (index_Base; index_Limit; index_Count)
{
   Body
}
```

The apparent variables used in *index_Base*, *index_Limit* and *index_Count* are `macro exprs` that are implicitly declared. *index_Base*, *index_Limit* and *index_Count* do not need to be single expressions, for example, you could declare `par (i=0, j=23; i != 76; i++, j--)`. In this case `i` and `j` are implicit `macro exprs`

Example

```
par (i=0; i<3; i++)
    {
       a[i] = b[i];
    }
```

expands to:

```
   par
   {
      a[0] = b[0];
      a[1] = b[1];
      a[2] = b[2];
   }
```

Replicated pipeline example

```
unsigned init;
unsigned q[149];
unsigned 31 out;

init = 57;
par (r = 0; r < 16; r++)
{
   ifselect(r == 0)
      q[r] = init;
   else ifselect(r == 15)
      out = q[r-1];
   else
      q[r] = q[r-1];
}
```

`ifselect` checks for the start of the pipeline, the replicator rules create the middle sections and `ifselect` checks the end. The replicated code expands to:

```
par
{
   q[0] = init;
   q[1] = q[0];
   q[2] = q[1];
   etc...

   q[14] = q[13];
   out = q[14];
}
```

5.1.3 Channel communication

Channels are a way of communicating between processes. When you write to a channel, a copy of the data you write is sent to the receiving process. This allows information to be shared between processes. Since a variable cannot be written to by multiple processes, you can write to the variable in a single process by reading channels that send data from other processes.

Each channel must be written to at one end, and read from at the other. The width and type of data sent down the channel must be of the same width and type of the channel. The channel can be an entry in an array of channels, or be pointed to by a channel pointer.

As with other variables, if no width or type is given to a channel, (or if it is set as `undefined`), the compiler can infer the channel width and type from its use.

Reading from a channel

Channel `?` *Variable*`;`

This assigns the value read from the channel to the variable. The variable may also be a signal, an array element, RAM element or WOM element.

Writing to a channel

Channel `!` *Expression*`;`

This writes the value of the expression to the channel. *Expression* may be any expression.

Example

```
set clock = external;
void main(void)
{
   unsigned 8 Res;
   chan Bill;

   par
   {
      Bill ! 23;
      Bill ? Res;
   }
}
```

Restrictions

No two statements may simultaneously write to or simultaneously read from a single channel.

```
par
{
    out ! 3 // Parallel write to a channel
    out ! 4
}
```

This code is illegal as it attempts to write simultaneously to a single channel. Similarly, the following code is illegal because an attempt is made to read simultaneously from the same channel:

```
par
{
    in ? x; // Parallel read from a channel
    in ? y;
}
```

5.1.4 prialt

The `prialt` statement selects the first channel ready to communicate from a list of channel cases. The syntax is similar to a conventional C `switch` statement.

```
prialt
{
    case CommsStatement:
         Statement

         break;
    ......
    case CommsStatement:
         Statement

         break;
    ......
    [default:
         Statement

         break;]
}
```

`prialt` selects between the communications on several channels depending on the readiness of the other end of the channel. *CommsStatement* must be one of the following:

Channel `?` *Variable*

Channel `!` *Expression*

The case whose communication statement is the first to be ready to transfer data will execute and data will be transferred over the channel. The statements up to the next `break` statement will then be executed.

Restrictions

The `prialt` construct does not allow the same channel to be listed twice in its cases and fall through of cases is prohibited. This means that each `case` must have its own `break` statement.

Priority

If two channels are ready simultaneously, then the first one listed in the code takes priority.

Default

`prialt` with no `default` case:
execution halts until one of the channels becomes ready to communicate.

`prialt` statement with `default` case:
if none of the channels is ready to communicate immediately then the `default` branch statements executes and the `prialt` statement terminates.

>: 5. Statements

5.2 Assignments

Handel-C assignments are of the form:

Variable = *Expression*;

For example:

```
x = 3;
y = a + b;
```

The expression on the right hand side must be of the same width and type (signed or unsigned) as the variable on the left hand side. The compiler generates an error if this is not the case.

The left hand side of the assignment may be any variable, array element or RAM element. The right hand side of the assignment may be any expression.

Short cuts

The following short cut assignment statements cannot be used in expressions as they can in conventional C but only in stand-alone statements. See Introduction: Expressions for more information.

Shortcuts cannot be used with RAM variables, as they contravene the RAM access restrictions

5.3 Control statements

5.3.1 continue

`continue` moves straight to the next iteration of a `for`, `while` or do loop. For do or `while`, this means that the test is executed immediately. In a `for` statement, the increment step is executed. This allows you to avoid deeply nested `if` … `else` statements within loops.

Example

```
for (i = 100; i > 0; i--)
{
   x = f( i );
   if ( x == 1 )
      continue;
   y += x * x;
}
```

You cannot use `continue` to jump out of or into `par` blocks.

5.3.2 goto

`goto` *label* moves straight to the statement specified by *label*. *label* has the same format as a variable name, and must be in the same function as the `goto`. Labels are local to the whole function, even if placed within an inner block. Formally, `goto` is never necessary. It may be useful for extracting yourself from deeply nested levels of code in case of error.

Example

```
for(… )
{
    for(… )
    {
        if(disaster)
            goto Error;
    }
}

Error:
    output ! error_code;
```

You cannot use *`goto`* to jump out of or into *`par`* blocks.

5.3.3 return [expression]

The `return` statement is used to return from a function to its caller. `return` terminates the function and returns control to the calling function. Execution resumes at the line immediately following the function call. `return` can return a value to the calling function. The value returned is of the type declared in the function declaration. Functions that do not return a value should be declared to be of type `void`.

>: 5. Statements

Example

```
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
      p = p * base;
    return(p);
}
```

You cannot use `return` to jump out of `par` blocks.

Statement	Expansion
Variable `++;`	*Variable* `=` *Variable* `+ 1;`
Variable `--;`	*Variable* `=` *Variable* `- 1;`
`++` *Variable*`;`	*Variable* `=` *Variable* `+ 1;`
`--` *Variable*`;`	*Variable* `=` *Variable* `- 1;`
Variable `+=` *Expression*`;`	*Variable* `=` *Variable* `+` *Expression*`;`
Variable `-=` *Expression*`;`	*Variable* `=` *Variable* `–` *Expression*`;`
Variable `*=` *Expression*`;`	*Variable* `=` *Variable* `*` *Expression*`;`
Variable `/=` *Expression*`;`	*Variable* `=` *Variable* `/` *Expression*`;`
Variable `%=` *Expression*`;`	*Variable* `=` *Variable* `%` *Expression*`;`
Variable `<<=` *Expression*`;`	*Variable* `=` *Variable* `<<` *Expression*`;`
Variable `>>=` *Expression*`;`	*Variable* `=` *Variable* `>>` *Expression*`;`
Variable `&=` *Expression*`;`	*Variable* `=` *Variable* `&` *Expression*`;`
Variable `|=` *Expression*`;`	*Variable* `=` *Variable* `|` *Expression*`;`
Variable `^=` *Expression*`;`	*Variable* `=` *Variable* `^` *Expression*`;`

5.3.4 Conditional execution (if … else)

Handel-C provides the standard C conditional execution construct as follows:

```
if (Expression)
    Statement
else
    Statement
```

As in conventional C, the `else` portion may be omitted if not required. For example:

```
if (x == 1)
    x = x + 1;
```

Statement may be replaced with a block of statements by enclosing the block in {...} brackets. For example:

```
if (x>y)
{
    a = b;
    c = d;
}
else
{
    a = d;
    c = b;
}
```

The first branch of the conditional is executed if the expression is true and the second branch is executed if the expression is false. Handel-C treats zero values as false and non-zero values as true. Relational and logical operators return values to match this meaning but it is also possible to use variables as conditions. For example:

```
if (x)
    a = b;
else
    c = d;
```

This is expanded by the compiler to:

```
if (x!=0)
    a = b;
else
    c = d;
```

When executed, if x is not equal to 0 then b is assigned to a. If x is 0 then d is assigned to c.

5.3.5 while loops

Handel-C provides `while` loops exactly as in conventional C:

while (Expression)
        Statement

>: 5. Statements

The contents of the `while` loop may be executed zero or more times depending on the value of *Expression*. While *Expression* is true then *Statement* is executed repeatedly. *Statement* may be replaced with a block of statements. For example:

```
x = 0;
while (x != 45)
{
    y = y + 5;
    x = x + 1;
}
```

This code adds 5 to y 45 times (equivalent to adding 225 to y).

5.3.6 do ... while loops

Handel-C provides `do ... while` loops exactly as in conventional C:

```
do
    Statement
while (Expression);
```

The contents of the `do` ... `while` loop is executed at least once because the conditional expression is evaluated at the end of the loop rather than at the beginning as is the case with `while` loops. *Statement* may be replaced with a block of statements. For example:

```
do
{
    a = a + b;
    x = x - 1;
} while (x>y);
```

5.3.7 for loops

Handel-C provides `for` loops similar to those in conventional C.

```
for (Initialization ; Test ; Iteration)
    Statement
```

The body of the `for` loop may be executed zero or more times according to the results of the condition test. There is a direct correspondence between `for` loops and `while` loops. Because of the benefits of parallelism, it is nearly always preferable to implement a `while` loop instead.

```
for (Init; Test; Inc)
    Body;
```

is directly equivalent to:

```
{
    Init;
    while (Test)
    {
        Body;
        Inc;
    }
}
```

unless the *Body* includes a `continue` statement. In a `for` loop `continue` jumps to before the increment, in a `while` loop `continue` jumps to after the increment.

Unless a specific `continue` statement is needed, it is always faster to implement the *for* loop as a *while* loop with the *Body* and *Inc* steps in parallel rather than in sequence when this is possible.

Each of the initialization, test and iteration statements is optional and may be omitted if not required. Note that `for` loops with no iteration step can cause combinational loops. As with all other Handel-C constructs, *Statement* may be replaced with a block of statements. For example:

```
for ( ; x>y ; x++ )
{
    a = b;
    c = d;
}
```

The difference between a conventional C `for` loop and the Handel-C version is in the initialization and iteration phases. In conventional C, these two fields contain expressions and by using expression side effects (such as ++ and `--`) and the sequential operator '`,`' conventional C allows complex operations to be performed. Since Handel-C does not allow side effects in expressions the initialization and iteration expressions have been replaced with statements. For example:

```
for (x = 0; x < 20; x = x+1)
{
    y = y + 2;
}
```

Here, the assignment of 0 to x and adding one to x are both statements and not expressions. These initialization and iteration statements can be replaced with blocks of statements by enclosing the block in {...} brackets. For example:

```
for ({ x=0; y=23;} ; x < 20; {x+=1; x*=2;} )
{
    y = y + 2;
}
```

5.3.8 switch

Handel-C provides `switch` statements similar to those in conventional C.

```
switch (Expression)
{
    case Constant:
        Statement
        break;
    ......
    default:
        Statement
        break;
}
```

The `switch` expression is evaluated and checked against each of the `case` compile time constants. The statement(s) guarded by the matching constant is executed until a `break` statement is encountered.

If no matches are found, the `default` statement is executed. If no default option is provided, no statements are executed.

Each of the *Statement* lines above may be replaced with a block of statements by enclosing the block in {...} brackets.

As with conventional C, it is possible to make execution drop through case branches by omitting a `break` statement. For example:

```
switch (x)
{
case 10:
    a = b;
case 11:
    c = d;
    break;

case 12:
    e = f;
    break;
}
```

Here, if x is 10, b is assigned to a and d is assigned to c, if x is 11, d is assigned to c and if x is 12, f is assigned to e.

The values following each case branch must be compile time constants.

5.3.9 break

Handel-C provides the normal C `break` statement for:

- terminating loops
- separation of `case` branches in `switch` and `prialt` statements.

`break` cannot be used to jump into or out of `par` blocks.

Loops

When used within a `while`, `do...while` or `for` loop, the loop is terminated and execution continues from the statement following the loop. For example:

```
for (x=0; x<32; x++)
{
    if (a[x]==0)
        break;
    b[x]=a[x];
}
// Execution continues here
```

switch

When used within a `switch` statement, execution of the `case` branch terminates and the statement following the `switch` is executed. For example:

```
switch (x)
{
    case 1:
    case 2:
        y++;
        break;
    case 3:
        z++;
        break;
}
// Execution continues here
```

>: 5. Statements

prialt

When used within a `prialt` statement, execution of the case branch terminates and the statement following the `prialt` is executed. For example:

```
prialt
{
    case a ? x:
        x++;
        break;
    case b ! y:
        y++;
        break;
}
// Execution continues here
```

5.3.10 delay

Handel-C provides a `delay` statement, not found in conventional C, which does nothing but takes one clock cycle to do it. This may be useful to avoid resource conflicts (for example to prevent two accesses to one RAM in a single clock cycle) or to adjust execution timing.

`delay` can also be used to break combinational logic cycles

5.3.11 try... reset

`try...reset` allows you to perform actions on receipt of a reset signal within a specified section of code.

Syntax

```
try
{
  statements
}
reset(condition)
{
  statements
}
```

During the execution of statements within the `try` block, if *condition* is true, the `reset` statement block will be executed immediately, else it will not. The *condition* expression is continually checked. If it occurs in the middle of a function, execution will immediately go to the reset thread. Static variables within the function will remain in the state they were in when the reset condition occurred. Variables and RAMs will not be re-initialized.

Example

```
void main(void)
{
   interface bus_in(int 1 input) resetbus();
   try
   {
      someFunction();
   }
   reset(resetbus.input == 1)
   {
      cleanUpSomeFunction();
   }
}
```

5.3.12 trysema()

`trysema(`*semaphore*`)` tests to see if the semaphore is owned. If not, it returns one and takes ownership of the semaphore. If it is, it returns zero. A semaphore may be freed by using the statement `releasesema(`*semaphore*`)`.

Example

```
inline void critRAMaccess(sema *RAMsema, ram int 8 (*danger)[4],
                   unsigned count)
{
   int 8 x;
   while(trysema(*RAMsema)==0) delay; // wait till you've got the
RAM
    x= (*danger)[count]; releasesema(*RAMsema);
}
```

Note that in 1.1 you can no longer take the semaphore twice without releasing it.

```
while(1)
{
  if (trysema(s)) {...}   // always succeeds because its the same
                          //'trysema' expression
}
```

In version 1, this worked. In version 1.1 and subsequent versions, the second and subsequent `trysema()` will always fail. Instead, use

```
while(1)
{
   if (trysema(s))
   {
      ...
      releasesema(s)
   }
}
```

5.3.13 releasesema()

`releasesema(`*semaphore*`)` releases a semaphore that was previously taken by `trysema(`*semaphore*`)`.

Example

```
inline void critRAMaccess(sema *RAMsema, ram int 8 (*danger)[4],
                  unsigned count)
{
   int 8 x;
   while(trysema(*RAMsema)==0) delay; // wait till you've got the
RAM
   x= (*danger)[count];
   releasesema(*RAMsema);
}
```

>: 6 Expressions

Clock cycles required

Expressions in Handel-C take no clock cycles to be evaluated, and so have no bearing on the number of clock cycles a given program takes to execute.

They affect the maximum possible clock rate for a program: the more complex an expression, the more hardware is involved in its evaluation and the longer it is likely to take because of combinational delays in the hardware. The clock period for the entire hardware program is limited by the longest such evaluation in the whole program.

Because expressions are not allowed to take any clock cycles, expressions with side effects are not permitted in Handel-C. For example;

```
if (a<b++)   /* NOT PERMITTED */
```

This is not permitted because the ++ operator has the side effect of assigning b+1 to b which requires one clock cycle.

Breaking down complex expressions

The longest and most complex C statement with many side effects can be written in terms of a larger number of simpler expressions and assignments. The resulting code is normally easier to read. For example:

```
a = (b++) + (((c-- ? d++ : e--)) , f);
```

can be rewritten as:

```
a = b + f;
b = b + 1;
if (c)
    d = d + 1;
else
    e = e - 1;
c = c - 1;
```

Pre-fix and postfix operators

Handel-C provides the prefix and postfix ++ and `--` operations as statements rather than expressions. For example:

```
a++;
b--;
++c;
--d;
```

is directly equivalent to:

```
a = a + 1;
b = b - 1;
c = c + 1;
d = d - 1;
```

6.1 Casting of expression types

Automatic conversions between signed and unsigned values are not allowed. 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.

You can cast to a type of undefined width. For example:

```
int 4 x;
unsigned int undefined y;

x = (int undefined)y;
```

The compiler will infer that y must be 4 bits wide.

Explanation of signed/unsigned casting

The following piece of Handel-C is invalid:

```
int 4 x;          // Range of x: -8...7
unsigned int 4 y; // Range of y: 0...15

x = y;  // Not allowed
```

This is because x is a signed integer while y is an unsigned integer. When generating hardware, it is not clear what the compiler should do here. It could simply assign the 4 bits of y to the 4 bits of x or it could extend y with an extra zero as its most significant bit to preserve its value and then assign these 5 bits to x assuming x was declared to be 5 bits wide.

To see the difference, consider the case when y is 10. By simply assigning these 4 bits to a signed integer, a result of -6 would be placed in x. A better solution might be to extend y to a five bit value by adding a 0 bit as its MSB to preserve the value of 10.

>: 6. Expressions

A programmer must explicitly cast the variables to the same type. Assuming that they wish to use the 4-bit value as a signed integer, the above example then becomes:

```
int 4 x;
unsigned int 4 y;

x = (int 4)y;
```

It is now clear that the value of x is the result of treating the 4 bits extracted from y as a signed integer.

6.1.1 Restrictions on casting

Casting cannot be used to change the width of values. For example, this is not allowed:

```
unsigned int 7 x;
int 12 y;

y = (int 12)x;  // Not allowed
```

The conversion should be done explicitly:

```
y = (int 12)(0 @ x);
```

Here, the concatenation operation produces a 12-bit unsigned value. The casting then changes this to a 12-bit signed integer for assignment to y.

This is to ensure that the programmer is aware of such conversions.

Explanation

```
int 7 x;
unsigned int 12 y;

x = -5;
y = (unsigned int 12)x;
```

The Handel-C compiler could take two routes. One would be to sign extend the value of x and produce the result 4091. The second would be to zero pad the value of x and produce the value of 123. Since neither method can preserve the value of x in y Handel-C performs neither automatically. Rather, it is left up to the programmer to decide which approach is correct in a particular situation and to write the expression accordingly. You may sign extend using the `adjs` macro and zero-pad using the `adju` macro.

>: 6. Expressions

6.2 Restrictions on RAMs and ROMs

Because of their architecture, RAMs and ROMs are restricted to performing operations sequentially. Only one element of a RAM or ROM may be addressed in any given clock cycle. As a result, familiar looking statements are often disallowed.

Ports within a multi-port RAM are in the same elements of memory so you can only make a single access to any one `mpram` port in a single clock cycle.

Example of disallowed assignment

Only one element of a RAM or ROM may be addressed in any given clock cycle and, as a result, familiar looking statements will often produce unexpected results. For example:

```
ram <unsigned int 8> x[4];
x[1] = x[3] + 1;
```

This code should not be used because the assignment attempts to read from the third element of x in the same cycle as it writes to the first element, and the memory may produce undefined results.

Example of disallowed condition evaluation

```
ram unsigned int 8 x[4];

if (x[0]==0)
    x[1] = 1; //double access, disallowed
```

This code is illegal because the condition evaluation must read from element 0 of the RAM in the same clock cycle as the assignment writes to element 1. Similar restrictions apply to `while` loops, `do` ... `while` loops, `for` loops and `switch` statements.

Incorrect execution of conditional operator

This code will not execute correctly because of the double access.

```
x = y>z ? RamA[1] : RamA[2];
```

The solution is to re-write the code as follows:

```
x = RamA[y>z ? 1 : 2];
```

Here, there is only a single access to the RAM so the problem does not occur.

Arrays of variables do not have these restrictions but may require substantially more hardware to implement than RAMs (depending on the target architecture).

>: 6. Expressions

6.3 assert

`assert` allows you to generate messages at compile-time if a condition is met. The messages can be used to check compile-time constants and help guard against possible problematic code alterations. The user uses an expression to check the value of a compile-time constant, and if the expression evaluates to false, an error message is sent to the standard error channel in the format

filename`:`*line number*`,` *start column* `-` *end column*`::Assertion failed:` *user-defined error string*

The default error message is:

```
"Error : User assertion failed"
```

If the expression evaluates to true, the whole assert expression is replaced by a constant expression.

`assert` can be used as a statement by passing 0 as the *trueValue*. If the condition is true, the whole assert statement is replaced by 0 (a null statement). This is shown in the example below. If the width of x is 3 (the condition is true), the whole statement is replaced by the *trueValue* of 0, so nothing happens.

```
assert (width(x)==3, 0, "Width of x is not 3 (it is %d)", width(x));
```

A more detailed example is given below. `assert` can also be used as an expression, where its return value is assigned to something. This is illustrated in the second example below, where the return value is assigned to *ReturnVal*.

Syntax

`assert(`*condition*`,`*trueValue* `[`*string with format specification(s)* `{`*,argument(s)*`}]);`

If *condition* is true, the whole expression reduces to *trueValue.* If *condition* is false, *string* will be sent to the standard error channel, with each *format specification* replaced by an *argument*. When `assert` encounters the first format specification (if any), it converts the value of the first argument into that format and outputs it. The second argument is formatted according to the second format specification and so on. If there are more expressions than format specifications, the extra expressions are ignored. The results are undefined if there are not enough arguments for all the format specifications.

The format specification is one of:

%c	Display as a character	%s	Display as a string
%d	Display as a decimal	%f	Display as a floating point
%o	Display as an octal	%x	Display as a hexadecimal

>: 6. Expressions

Using assert as a statement

In the example below `assert` is used as a statement.

```
set clock = external "C1";
int f(int x)
{
   assert(width(x)==3, 0, "Width of x is not 3 (it is %d)",
width(x));
   return x+1;
}

void main(void)
{
   int 4 y;
   y = f(y);
}
```

x will be inferred to have a width of 4, so the following message will be displayed.

```
F:\proj\test.hcc(4)(2) : Assertion failed : Width of x is not 3 (it
is 4)
```

Using assert as an expression

In the example below, `assert` is used as an expression.

```
set clock = external "C1";
unsigned func(unsigned p, unsigned q)
{
   macro expr WidthSum(a, b) = width(a) + width(b);
   macro expr CheckWidths(a, b) = assert((WidthSum(a, b)==32
      || WidthSum(a, b)==16), WidthSum(a, b),
      "Sum of widths of function parameters is not 16 or 32 (it is
%d)",
      WidthSum(a, b));
   unsigned 16 ReturnVal;

   ReturnVal = CheckWidths(p, q);

   return ReturnVal;
}
```

```
void main(void)
{
   static unsigned 9 x;
   static unsigned 7 y;
   unsigned result;

   result = func(x, y);
}
```

6.4 Bit manipulation operators

The following bit manipulation operators are provided in Handel-C:

`<<`	Shift left
`>>`	Shift right
`<-`	Take least significant bits
`\\`	Drop least significant bits
`@`	Concatenate bits
`[]`	Bit selection
`width(`*Expression*`)`	Width of expression

6.4.1 Shift operators

The shift operators shift a value left or right by a variable number of bits resulting in a value of the same width as the value being shifted. Any bits shifted outside this width are lost.

When shifting unsigned values, the right shift pads the upper bits with zeros. When right shifting signed values, the upper bits are copies of the top bit of the original value. Thus, a shift right by 1 divides the value by 2 and preserves the sign. For example:

```
static unsigned 4 a = 0b1101;
static unsigned (log2ceil(width(a)+1)) b = 2;

a = a >> b;   //a becomes 0b0011
b--;
a = a >> b;   //a becomes 0b0001
```

The width of b needs to have a width equal to `log2(width(a)+1)` rounded up to the nearest whole number. This can be calculated using the `log2ceil` macro.

>: 6. Expressions

6.4.2 Take /drop operators

The take operator, <-, returns the n least significant bits of a value. The drop operator, \\, returns all but the n least significant bits of a value. *n* must be a compile-time constant. For example:

```
macro expr four = 8 / 2;
unsigned int 8 x;
unsigned int 4 y;
unsigned int 4 z;

x = 0xC7;
y = x <- four;
z = x \\ 4;
```

This results in y being set to 7 and z being set to 12 (or 0xC in hexadecimal).

6.4.3 Concatenation operator

The concatenation operator, @, joins two sets of bits together into a result whose width is the sum of the widths of the two operands. For example:

```
unsigned int 8 x;
unsigned int 4 y;
unsigned int 4 z;

y = 0xC;
z = 0x7;
x = y @ z;
```

This results in x being set to 0xC7. The left operand of the concatenation operator forms the most significant bits of the result.

You may also use the concatenation operator to zero pad a variable to a given width.

```
unsigned int 8 x;
unsigned int 8 y;
unsigned int 16 z;

z = (0 @ x) * (0 @ y); //width of zero constant inferred to be 8
bits
```

>: 6. Expressions

6.4.4 Bit selection

Individual bits or a range of bits may be selected from a value by using the `[]` operator. Bit 0 is the least significant bit and bit *n*-1 is the most significant bit where *n* is the width of the value. For example:

```
unsigned int 8 x;
unsigned int 1 y;
unsigned int 5 z;

x = 0b01001001;
y = x[4];
z = x[7:3];
```

This results in `y` being set to 0 and `z` being set to 9. Note that the range of bits is of the form *MSB*`:`*LSB* and is inclusive. Thus, the range `7:3` is 5 bits wide.

The bit selection values must be fixed at compile time.

Bit selection is allowed in RAM, ROM and array elements. For example:

```
ram int 7 w[23];
int 5 x[4];
int 3 y;
unsigned int 1 z;

y = w[10][4:2];
z = (unsigned 1)x[2][0];
```

The 10 specifies the RAM entry and the `4:2` selects three bits from the middle of the value in the RAM *w* is set to the value of the selected bits.

Similarly, `z` is set to the least significant bit in the `x[2]` variable.

You cannot assign to bit ranges, only read from them.

6.4.5 Width operator

The `width()` operator returns the width of an expression. It is a compile time constant. For example:

```
x = y <- width(x);
```

This takes the least significant bits of `y` and assigns them to `x`. The `width()` operator ensures that the correct number of bits is taken from `y` to match the width of `x`.

>: 6. Expressions

6.5 Arithmetic operators

The following arithmetic operators are provided in Handel-C:

Operator	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulo arithmetic

Any attempt to perform one of these operations on two expressions of differing widths or types results in a compiler error. For example:

```
int 4 w;
int 3 x;
int 4 y;
unsigned 4 z;

y = w + x; // ILLEGAL
z = w + y; // ILLEGAL
```

The first statement is illegal because `w` and `x` have different widths. The second statement is illegal because `w` and `y` are signed integers and `z` is an unsigned integer.

Width of results

All operators return results of the same width as their operands. Thus, all overflow bits are lost. For example:

```
unsigned int 8 x;
unsigned int 8 y;
unsigned int 8 z;

x = 128;
y = 192;
z = 2;

x = x + y;
z = z * y;
```

This example results in `x` being set to 64 and `z` being set to 128.

>: 6. Expressions

By using the bit manipulation operators to expand the operands, it is possible to obtain extra information from the arithmetic operations. For instance, the carry bit of an addition or the overflow bits of a multiplication may be obtained by first expanding the operands to the maximum width required to contain this extra information. For example:

```
unsigned int 8 u;
unsigned int 8 v;
unsigned int 9 w;
unsigned int 8 x;
unsigned int 8 y;
unsigned int 16 z;

w = (0 @ u) + (0 @ v);
z = (0 @ x) * (0 @ y);
```

In this example, `w` and `z` contain all the information obtainable from the addition and multiplication operations. Note that the constant zeros do not require a width specification because the compiler can infer their widths from the usage. The zeros in the first assignment must be 1 bit wide because the destination is 9 bits wide while the source operands are only 8 bits wide. In the second assignment, the zero constants must be 8 bits wide because the destination is 16 bits wide while the source operands are only 8 bits wide.

6.6 Relational operators

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

These operators compare values of the same width and return a single bit wide `unsigned int` value of 0 for false or 1 for true. This means that this conventional C code is invalid:

```
unsigned 8 w, x, y, z;

w = x + (y >z); // NOT ALLOWED
```

Instead, you should write:

```
w = x + (0 @ (y > z));
```

6.6.1 Signed/unsigned compares

Signed/signed compares and unsigned/unsigned compares are handled automatically. Mixed signed and unsigned compares are not handled automatically. For example:

```
unsigned 8 x;
int 8 y;

if (x>y) // Not allowed
    ...
```

To compare signed and unsigned values you must sign extend each of the parameters. The above code can be rewritten as:

```
unsigned 8 x;
int 8 y;

if ((int)(0@x) > (y[7]@y))
    ...
```

6.6.2 Implicit compares

The Handel-C compiler inserts implicit compares with zero if a value is used as a condition on its own. For example:

```
while (1)
{
    ...
}
```

Is directly expanded to:

```
while (1 != 0)
{
    ...
}
```

6.7 Logical operators

Operator	Meaning
&&	Logical and
||	Logical or
!	Logical not

These operators are provided to combine conditions as in conventional C. Each operator takes 1-bit unsigned operands and returns a 1-bit unsigned result.

Note that the operands of these operators need not be the results of relational operators. This feature allows some familiar looking conventional C constructs.

Example:

```
if (x || y > z)
    w = 0;
```

In this example, the variable x need not be 1 bit wide. If it is wider, the Handel-C compiler inserts a compare with 0.

```
if (x != 0 || y > z)
    w = 0;
```

The condition of the `if` statement is true if x is not equal to 0 or y is greater than z.

C-like example

```
while (x || y)
{
    ...
}
```

Again, if the variables are wider than 1-bit, the Handel-C compiler inserts compares with 0.

6.7.1 Bitwise logical operators

Operator	Meaning
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~	Bitwise not

These operators perform bitwise logical operations on values. Both operands must be of the same type and width: the resulting value will also be this type and width. For example:

```
unsigned int 6 w;
unsigned int 6 x;
unsigned int 6 y;
unsigned int 6 z;

w = 0b101010;
x = 0b011100;
y = w & x;
z = w | x;
w = w ^ ~x;
```

This example results in `y` having the value 0b001000, `z` having the value 0b111110 and `w` having the value 0b001001.

6.8 Conditional operator

Handel-C provides the conditional expression construct familiar from conventional C. Its format is:

Expression ? *Expression* : *Expression*

The first expression is evaluated and if true, the whole expression evaluates to the result of the second expression. If the first expression is false, the whole expression evaluates to the result of the third expression. For example:

```
x = (y > z) ? y : z;
```

This sets x to the maximum of y and z. This code is directly equivalent to:

```
if (y > z)
    x = y;
else
    x = z;
```

The advantage of using this construct is that the result is an expression so it can be embedded in a more complex expression. For example:

```
x = ((w==0) ? y : z) + 4;
```

In this case, the signedness and widths of `x`, `y` and `z` must match (as the value of `y` or `z` may be assigned to `x`), but those of `w` need not.

6.9 Member operators (. / ->)

The structure member operator (.) is used to access members of a structure or mpram, or to access a port within an interface.

The structure pointer operator (->) can be used, as in ANSI-C. It is used to access the members of a structure or mpram, when the structure/mpram is referenced through a pointer.

```
mpram Fred
{
  ram <unsigned 8> ReadWrite[256]; // Read/write port
  rom <unsigned 8> Read[256];   // Read only port
} Joan;

mpram Fred *mpramPtr;

mpramPtr = &Joan;

x = mpramPtr->Read[56];
```

If a memory is made up of structures, the structure member operator can be used to reference structure members within the memory

```
ram struct S compRAM[100];
ram struct S (*ramStructPtr)[];
ramStructPtr = &compRAM;
x = (*ramStructPtr)[10].a;
```

>: 7 Functions and macros

7.1 Comparison of functions and macros

Handel-C includes and extends the range of functions and macros offered by ANSI-C.

	Return value?	**Typed return values and parameters?**	**Called by reference?**	**Shared hardware?**
Functions	Can have	Yes	No	Yes
Arrays of functions	Can have	Yes	No	Yes
Inline functions	Can have	Yes	No	No
Preprocessor macros	Can have	No	Yes	No
Macro expressions	Must have	No	Yes	No
Shared expressions	Must have	No	Yes	Yes
Macro procedures	None	No	Yes	No

7.1.1 Functions and macros: language issues

Called by reference or value

Functions employ call-by-value on their parameters, whereas macros effectively employ call-by-reference. Consider the code:

```
void inline f_pseudoswap (int 12 x, int 12 y)
{
   par
   {
      x = y;
      y = x;
   }
}
```

>: 7. Functions and macros

```
macro proc mp_swap (x, y)
{
   par
   {
      x = y;
      y = x;
   }
}
```

If you call `mp_swap(a,b)` the values of a and `b` will be swapped.

If you call call `f_pseudoswap(a,b)` the values a and b are copied to the formal parameters x and y of `f_pseudoswap. x` and `y` are swapped, but a and b are unaffected. The swap function with the same behaviour as the macro procedure is therefore

```
void inline f_swap (int 12 * x, int 12 * y)
{
   par
   {
      * x = * y;
      * y = * x;
   }
}
```

with a call of the form `f_swap(&a,&b)`.

Typed or untyped parameters

Function parameters must have a type, although the width can sometimes be inferred by the compiler.

Macro expressions and procedures are un-typed in the sense that their formal parameters can't be given types. The type of macro parameters is inferred from the type in the call statement.

This means that it is better to use macros for parameterizable code. For example, macro procedures can be used in libraries if you want to create multiple instances of hardware, but leave them untyped to make the code more generic.

Recursion

In Handel-C, functions may not be recursive. Macro procedure and macro expressions can be used to capture compile-time recursion.
If you use recursive macro procedures you need to use `ifselect` to guard the base case (the condition where the recursion terminates). If you use recursive macro expressions, you need to use `select` to guard the base case.

Macro procedure example:

```
unsigned 4 g;
macro proc p(x)
{
   ifselect(width(x) != 0)
   {
      g = 0@x;
      p(x\\1);
   }
   else
   delay;
}

set clock = external;
void main()
{
   unsigned 4 i;
   p(i);
}
```

Macro expression example:

```
macro expr copycat (copies, bits) =
   select (copies <= 0, (unsigned 0) 0,
                    bits @ copycat (copies - 1, bits));
```

7.1.2 Functions and macros: sharing hardware

Calls to functions and shared expressions result in a single shared piece of hardware. This is equivalent to an ANSI-C function resulting in a single shared section of machine code.

Shared hardware will reduce the size of your design, but care is needed if you have parallel code where multiple branches access the shared hardware. Shared hardware may also compromise the speed of your design as it tends to lead to an increase in logic depth.

Each call to an inline function, macro procedure or macro expression results in a separate piece of hardware.

Arrays of functions allow a specified number of copies to be created.

7.1.3 Functions and macros: clock cycles

Macro expressions and shared expressions are evaluated in a single clock cycle, where the expression is assigned to a variable. Functions and macro procedures may involve control logic, and may take many cycles.

>: 7. Functions and macros

7.1.4 Functions and macros: examples

There are many ways in which a much-used code fragment can be expressed. The examples below all multiply a value by 1.5.

Preprocessor macro

```
#define de_sesqui(s) ((s) + ((s) >> 1))
#define dp_sesqui(d,s) ((d) = (s) + ((s) >> 1))
```

Macro expression

```
macro expr me_sesqui (s) = s + (s >> 1);
```

Shared expression

```
shared expr se_sesqui (s) = s + (s >> 1);
```

Macro procedure

```
macro proc mp_sesqui (d, s)
{
   d = s;
   d += (d >> 1);
}
```

Function

```
void f_sesqui (int * d, int s) //"shared" function without return
{
   * d = s;
   * d += ((* d) >> 1);
}
```

```
int rf_sesqui (int s) //"shared" function with return
{
   int ret;
   ret = s;
   ret += (ret >> 1);
   return ret;
}
```

>: 7. Functions and macros

Array of functions

```
void af_sesqui [5] (int * d, int s) //function array without return
{
   * d = s;
   * d += ((* d) >> 1);
}

int arf_sesqui [5] (int s) // function array with return
{
   int ret;
   ret = s;
   ret += (ret >> 1);
   return ret;
}
```

Inline function

```
void inline if_sesqui (int * d, int s) // inline
function without return
{
   * d = s;
   * d += ((* d) >> 1);
}

int inline irf_sesqui (int s) // inline function with return
{
   int ret;
   ret = s;
   ret += (ret >> 1);
   return ret;
}
```

How to call the example macros and functions

The example macros and functions above can be called using code such as:

```
{
   int 5 x, y;
   x = 10;

   y = de_sesqui (x);
   dp_sesqui (y, x);

   y = me_sesqui (x);
   y = se_sesqui (x);

   mp_sesqui (y, x);

   f_sesqui (& y, x);
   y = rf_sesqui (x);

   af_sesqui[2] (& y, x);
   y = arf_sesqui[2] (x);

   if_sesqui (& y, x);
   y = irf_sesqui (x);
}
```

7.1.5 Accessing external names

You can refer to functions, macros and shared expressions that have been defined in another file by prototyping them. You prototype by declaring an object at the top of the file in which it is used.

Function prototypes are in the following format:

returnType functionName`(`*parameterTypeList*`);`

Macro prototypes are of the form:

`macro expr` *Name*`(`*parameterList*`);`

`macro proc` *Name*`(`*parameterList*`);`

Functions and macros may be `static` or `extern`. `static` functions and macros may only be used in the file where they are defined.

You can collect all the prototypes into a single header file and then `#include` it within your code files.

You can access variables declared in other files by using the `extern` keyword.

You cannot use variables to communicate between clock domains. Variables are restricted to a single clock domain. The only items that can connect across separate clock domains are channels and `mprams`.

>: 7. Functions and macros

7.2 Functions

Functions are similar to functions in ANSI-C. A function is compiled to be a single shared piece of hardware, much as a C compiler generates a single shared block of machine code.

Handel-C has been extended to provide arrays of functions and inline functions.

Arrays of functions provide multiple copies of a function. You can select which copy is used at any time.

Inline functions are similar to macros in that they are expanded wherever they are used.

You may also use a macro proc (a parameterized macro procedure).

Functions take arguments and return values. A function that does not return a value is of type `void`. The default return type is `int undefined`. Functions that do not take arguments have `void` as their parameter list, for example:

```
void main(void)
```

As in ANSI-C, function arguments are passed by value. This means that a local copy is created that is only in scope within the function. Changes take place on this copy.

To access a variable outside the function, you must pass the function a pointer to that variable. A local copy will be made of the pointer, but it will still point to the same variable. This is known as passing by reference.

Architectural types (hardware constructs) must be passed by reference (a pointer to or address of the construct). The only architectural type that can be passed to or returned by a function by value is a signal. All others (and structures containing them) must be passed by reference. Arrays and functions can also only be passed by reference.

7.2.1 Function definitions, declarations and prototypes

Function definitions, declarations and prototypes are defined as in ANSI-C. Functions must be declared in every file that they are used in, though they should only be defined once. It is common to put function prototypes into a header file and `#include` that in every file where they are used.

Function definition

The definition of a function consists of its name and parameters plus the code body that it performs when it is called.

>: 7. Functions and macros

returnType Name(*parameterList*)
{
  declarations
  statements
}

Function prototype

A function prototype lists the function name, return type and the types of the parameters.

returnType Name(*parameterType parameter_1, parameterType parameter_n*);

The parameter names in a function prototype are only in scope in the prototype. You can use different names in the definition of the function. The parameter types are used by the compiler to check that the correct types are used for the function arguments within the rest of the file.

Function declaration

A function declaration lists the function name and return type.

returnType Name();

7.2.2 Functions: scope

Functions cannot be defined within other functions. By default, functions are `extern` (they can be used anywhere). Functions can also be defined as `static` (they can only be used in the file in which they are defined).

7.2.3 Arrays of functions

An array of functions is a collection of identical functions. It is not the same as an array of function pointers (each of whose elements can point to a different function). A function array allows you to run different copies of the same function in parallel. Without this construct, the only safe way to run a function in parallel with itself would be to explicitly declare two functions with different names.

Function arrays allow functions to be copied and shared neatly. For example:

```
unsigned func[2](unsigned x, unsigned y)
{
   return (x + y);
}
```

>: 7. Functions and macros

Syntax

The syntax is a normal function declaration, with square brackets added to specify that this is an array declaration as well as a function declaration. The general form of a function array declaration is:

```
returnType Name[Size](parameterList);
```

7.2.4 Using static variables in arrays of functions

In the example below each function in the array has its own copy of the `static` variable 't'. Thus, if `func[0]`'s copy of `t` is modified, `func[1]`'s copy remains unaffected.

```
set clock = external "C1";

unsigned func[2](unsigned a, unsigned b)
{
  static unsigned t = 0;
  t++;
  return a + b + t;
}

 void main(void)
{
  unsigned 7 p, q, r, s, t, u, v, w, x, y, z;

  par
  {
    p = 1;
    q = 1;
    r = 1;
    s = 1;
    t = 1;
    u = 1;
  }
  par
  {
    v = func[0](p, q);    // v = 3   (t in func[0] is 1)
    w = func[1](r, s);    // w = 3   (t in func[1] is 1)
  }
  x = func[0](t, u);      // x = 4   (t in func[0] is 2)
  y = func[0](v, w);      // y = 9   (t in func[0] is 3)

  z = func[1](x, y);      // z = 15  (t in func[1] is 2)
}
```

>: 7. Functions and macros

Function arrays: Example

```
set clock = external "P1";

// Function array prototype

unsigned func[2](unsigned x, unsigned y);

// Main program

void main(void)
{
   unsigned a, b, c, d, e, f;
   unsigned short r1, r2, r3, r4;
   unsigned result;


   par
   {
      a = 12;
      b = 22;
      c = 32;
      d = 42;
      e = 52;
      f = 62;
   }


   par
   {
      r1 = func[0](a, b);
      r2 = func[1](c, d);
   }

   par
   {
      r3 = func[0](e, f);
      r4 = func[1](r1, r2);
   }

   result = func[0](r3, r4);
}
```

```
// Function array definition

unsigned func[2](unsigned x, unsigned y)
{
   return (x + y);
}
```

7.2.5 Function pointers

These are a very powerful, yet potentially confusing feature. In situations where any one of a number of functions can be called at a particular point, it is neater and more concise to use a function pointer, where the alternative might be a long if-else chain, or a long switch statement (see example).

Function pointers can be assigned with or without the address operator & (similar to assigning array addresses). Functions pointed to can be called with or without the indirection operator.

A function name can be assigned to a pointer without the &

```
p = addeven;
```

although the & format is clearer:

```
p = &addeven;
```

A function pointed to can be called by writing

```
(*chk)(a, b);
```

This can also be written in the shorthand form:

```
chk(a, b);
```

The first form is preferable, as it tips off anyone reading the code that a function pointer is being used.

Function pointers: example

Consider the following program:

```
set clock = external "P1";

unsigned 1 check(short int *a, short int *b,
   unsigned 1 (*chk)(const short int *, const short int *));
```

>: 7. Functions and macros

```
unsigned 1 addeven(const short int *x, const short int *y);
unsigned 1 minuseven(const short int *x, const short int *y);
unsigned 1 diveven(const short int *x, const short int *y);
unsigned 1 modeven(const short int *x, const short int *y);

void main(void)
{
   short int m, n;
   unsigned 2 choice;
   unsigned 1 result;
   unsigned 1 (*p)(const short *, const short *);

   par
   {
      m = 19;
      n = 47;
   }

   do
   {
    switch (choice)
      {
      case 0:
         p = addeven;
         break;
      case 1:
         p = minuseven;
         break;
      case 2:
         p = diveven;
         break;
      case 3:
         p = modeven;
         break;
      default:
         delay;
         break;
      }
      par
      {
         result = check(&m, &n, p);
         choice++;
      }
   }
   while(choice);
}

unsigned 1 check(short int *a, short int *b,
            unsigned 1 (*chk)(const short int *, const short int *))
{
   return (*chk)(a, b);
}

unsigned 1 addeven(const short int *x, const short int *y)
{
   return (unsigned)(*x + *y)[0];
}

unsigned 1 minuseven(const short int *x, const short int *y)
{
   return (unsigned) (*x - *y)[0];
}

unsigned 1 diveven(const short int *x, const short int *y)
{
   return (unsigned) (*x / *y)[0];
}

unsigned 1 modeven(const short int *x, const short int *y)
{
   return (unsigned) (*x % *y)[0];
}
```

The function `addeven` checks whether the sum of two numbers is even. Similar checks are carried out by `minuseven` (difference of two numbers), `diveven` (division) and `modeven` (modulus). The function `check` simply calls the function whose pointer it receives, with the arguments it receives. This gives a consistent interface to the *xxx*`even` functions. Pay close attention to the declaration of `check`, and of function pointer `p`. The parentheses around `*p` (and `*chk` in the declaration of `check`) are necessary for the compiler to make the correct interpretation.

>: 7. Functions and macros

Possible code optimization

Inside the main program body, `check` was called like this:
```
check(&m, &n, p);
```

It could have been written like this:
```
check(&m, &n, xxxeven);
```

eliminating the need for an additional pointer variable.

Here is the `main` section written using this form of expression:

```
void main(void)
{
   short int m, n;
   unsigned 2 choice;
   unsigned 1 result;

   par
   {
      m = 19;
      n = 47;
   }
   do
   {
      switch (choice)
      case 0:
         result = check(&m, &n, &addeven);
         break;
      case 1:
         result = check(&m, &n, &multeven);
         break;
      case 2:
         result = check(&m, &n, &diveven);
         break;
      case 3:
         result = check(&m, &n, &modeven);
         break;
      default:
         break;
      choice++;
   }

   while(choice);
}
```

>: 7. Functions and macros

7.2.6 Shared code restrictions

Functions must not be shared by two different parts of the program on the same clock cycle. For example:

```
int func(int x, int y);

void main(void)
{
   int a, b, c, d, e, f, foo;
   // etc ...

   par
   {
      a = func(b, c);
      {
         b = foo;
         d = func(e, f); // NOT ALLOWED
      }
   }
   // etc ...
}


int func(int x, int y);
{
   if (x ==y)
      delay;
   else
   {
      x = x % y;
   }
   x *= 10;

   return(x);
}
```

This is not allowed because part of the single function is used twice in the same clock cycle.

This overlapping usage is not detected by the compiler, as it is a run-time error. It is therefore the programmer's responsibility to ensure that code usage does not overlap. This may be done by declaring functions to be inline (are expanded whenever they are used) or declaring an array of functions, one to be used in each parallel branch.

```
inline int func(x, y);

par
{
    a = func(b, c);
    {
     b = foo;
     d = func(e, f);
    }
}
```

or

```
int func[3](x, y);

par
{
    a = func[0](b, c);
    {
     b = foo;
     d = func[1](e, f);
    }
}
```

7.2.7 Multiple functions in a statement

Because each statement in Handel-C must take a single clock cycle, you cannot have multiple functions in a single statement.

Instead of

```
y = f(g(x));//illegal
```

you can write

```
z=g(x);
y=f(z);
```

Instead of

```
y = f(x) + g(z); //illegal
```

you can write:

>: 7. Functions and macros

```
par
{
   a = f(x);
   b = g(z);
}
y = a+b;
```

7.2.8 Recursion in macros and functions.

Macros can be recursive in Handel-C, but due to the absence of a stack in Handel-C, functions cannot be recursive.

The depth of recursion, though unbounded, must be determinable at compile-time.

7.3 Macros

The Handel-C compiler passes source code through a standard C preprocessor before compilation allowing the use of `#define` to define constants and macros in the usual manner. Since the preprocessor can only perform textual substitution, some useful macro constructs cannot be expressed. For example, there is no way to create recursive macros using the preprocessor.

Handel-C provides additional macro support to allow more powerful macros to be defined (for example, recursive macro expressions). In addition, Handel-C supports shared macro expressions to generate one piece of hardware which is shared by a number of parts of the overall program similar to the way that procedures allow conventional C to share one piece of code between many parts of a conventional program.

7.3.1 Non-parameterized macro expressions

Non-parameterized macro expressions are of two types:

- simple constant equivalent to `#define`
- a constant expression

Constant

This first form of the macro is a simple expression. For example:

```
macro expr DATA_WIDTH = 15;

int DATA_WIDTH x;
```

This form of the macro is similar to the `#define` macro. Whenever `DATA_WIDTH` appears in the program, the constant 15 is inserted in its place.

>: 7. Functions and macros

Constant expression

To provide a more general solution, you can use a real expression. For example:

```
macro expr sum = (x + y) @ (y + z);

v = sum;
w = sum;
```

7.3.2 Parameterized macro expressions

Handel-C allows macros with parameters. For example:

```
macro expr add3(x) = x+3;

y = add3(z);
```

This is equivalent to the following code:

```
y = z + 3;
```

This form of the macro is similar to the `#define` macro in that every time the `add3()` macro is referenced, it is expanded in the manner shown above. In this example, an adder is generated in hardware every time the `add3()` macro is used.

7.3.3 Select operator

The `select(...)` operator is used to mean 'select at compile time'. Its general usage is:

`select(`*Expression1*, *Expression2*, *Expression3*`)`

Expression1 must be a compile time constant. If *Expression1* evaluates to true then the Handel-C compiler replaces the whole expression with *Expression2*. If *Expression1* evaluates to false then the Handel-C compiler replaces the whole expression with *Expression3*.

Comparison with conditional operator

The difference between `select` and the conditional operators is seen in this example:

```
w = (width(x)==4 ? y : z);
```

The example generates hardware to compare the width of the variable `x` with 4 and set `w` to the value of `y` or `z` depending on whether this value is equal to 4 or not.

This is probably not what was intended because both `width(x)` and 4 are constants. What was probably intended was for the compiler to check whether the width of `x` was 4 and then simply replace the whole expression above with y or z according to the value. This can be written as follows:

```
w = select(width(x)==4 , y , z);
```

In this example, the compiler evaluates the first expression and replaces the whole line with either `w=y;` or `w=z;`. No hardware for the conditional is generated.

Combining with macros

This is more useful when macros are combined with this feature.

```
macro expr adjust(x, n) =
    select(width(x) < n, (0 @ x), (x <- n));

unsigned 4 a;
unsigned 5 b;
unsigned 6 c;

b = adjust(a, width(b));
b = adjust(c, width(b));
```

This example is for a macro that equalizes widths of variables in an assignment. If the right hand side of an assignment is narrower than the left hand side then the right hand side must be padded with zeros in its most significant bits. If the right hand side is wider than the left hand side, the least significant bits of the right hand side must be taken and assigned to the left hand side.

The `select(...)` operator is used here to tell the compiler to generate different expressions depending on the width of one of the parameters to the macro. The last two lines of the example could have been written by hand as follows:

```
b = 0 @ a;
b = c <- 5;
```

The macro comes into its own if the width of one of the variables changes. Suppose that during debugging, it is discovered that the variable a is not wide enough and needs to be 8 bits wide to hold some values used during the calculation. Using the macro, the only change required would be to alter the declaration of the variable a. The compiler would then replace the statement `b = 0 @ a;` with `b = a <- 5;` automatically.

This form of macro also comes in useful is when variables of undefined width are used. If the compiler is used to infer widths of variables, it may be tedious to work out by hand which form of the assignment is required. By using the `select(...)` operator in this way, the correct expression is generated without you having to know the widths of variables at any stage.

>: 7. Functions and macros

7.3.4 ifselect

`ifselect` checks the result of a compile-time constant expression at compile time. If the condition is true, the following statement or code block is compiled. If false, it is dropped and an else condition can be compiled if it exists. Thus, whole statements can be selected or discarded at compile time, depending on the evaluation of the expression.

The `ifselect` construct allows you to build recursive macros, in a similar way to `select`. It is also useful inside replicated blocks of code as the replicator index is a compile-time constant. Hence, you can use `ifselect` to detect the first and last items in a replicated block of code and build pipelines.

Syntax

```
ifselect (condition)
      statement 1
[else
      statement 2]
```

Example

```
int 12 a;
int 13 b;
int undefined c;

ifselect(width(a) >= width(b))
   c = a;
else
   c = b;
```

c is assigned to by either a or b, depending on their width relationship.

Pipeline example

```
unsigned init;
unsigned q[15];
unsigned 31 out;

init = 57;
par (r = 0; r < 16; r++)
{
   ifselect(r == 0)
      q[r] = init;
   else ifselect(r == 15)
      out = q[r-1];
   else
      q[r] = q[r-1];
}
```

7.3.5 Recursive macro expressions

Preprocessor macros (those defined with `#define`) cannot generate recursive expressions. By combining Handel-C macros (those defined with `macro expr`) and the `select(`...`)` operator, recursive macros can express complex hardware simply. This type of macro is particularly important in Handel-C where the exact form of the macro may depend on the width of a parameter to the macro.

Variable sign extension example

When assigning a narrow signed variable to a wider variable, the most significant bits of the wide variable should be padded with the sign bit (MSB) of the narrow variable.

Value	4-bit representation	Conversion to 8-bit representation
-2	0b1110	0b11111110
6	0b0110	0b00000110

The following code suffices for a 4-bit to 8-bit conversion

```
int 8 x;
int 4 y;

x = y[3] @ y[3] @ y[3] @ y[3] @ y;
```

but it is tedious for variables that differ by a significant number of bits. It also does not deal with the case when the exact widths of the variables are not known. What is needed is a macro to sign extend a variable. For example:

>: 7. Functions and macros

```
macro expr copy(x, n) =
    select(n==1, x, (x @ copy(x, n-1)));

macro expr extend(y, m) =
    copy(y[width(y)-1], m-width(y)) @ y;

int a;
int b;  // Where b is known to be wider than a

b = extend(a, width(b));
```

The `copy` macro generates n copies of the expression x concatenated together. The macro is recursive and uses the `select(...)` operator to evaluate whether it is on its last iteration (in which case it just evaluates to the expression) or whether it should continue to recurse by a further level.

The `extend` macro concatenates the sign bit of its parameter *m*-*k* times onto the most significant bits of the parameter. Here, *m* is the required width of the expression y and *k* is the actual width of the expression y.

The final assignment correctly sign extends a to the width of b for any variable widths where `width(b)` is greater than `width(a)`.

7.3.6 Recursive macro expressions: a larger example

This example illustrates the generation of large quantities of hardware from simple macros. The example is a multiplier whose width depends on the parameters of the macro. Although Handel-C includes a multiplication operator as part of the language, this example serves as a starting point for generating large regular hardware structures using macros.

The multiplier generates the hardware for a single cycle long multiplication operation from a single macro. The source code is:

```
macro expr multiply(x, y) =  select(width(x) == 0, 0,
               multiply(x \\ 1, y << 1) +
               (x[0] == 1 ? y : 0));
a = multiply (b , c);
```

At each stage of recursion, the multiplier tests whether the bottom bit of the x parameter is 1. If it is then y is added to the ‘running total’. The multiplier then recurses by dropping the LSB of x and multiplying y by 2 until there are no bits left in x. The overall result is an expression that is the sum of each bit in x multiplied by y. This is the familiar long multiplication structure. For example, if both parameters are 4 bits wide, the macro expands to:

```
a = ((b \\ 3)[0]==1 ? c<<3 : 0) +
    ((b \\ 2)[0]==1 ? c<<2 : 0) +
    ((b \\ 1)[0]==1 ? c<<1 : 0) +
    (b[0]==1 ? c : 0);
```

This code is equivalent to:

```
a = ((b & 8)==8 ? c*8 : 0) +
    ((b & 4)==4 ? c*4 : 0) +
    ((b & 2)==2 ? c*2 : 0) +
    ((b & 1)==1 ? c : 0);
```

which is a standard long multiplication calculation.

7.3.7 Shared expressions

By default, Handel-C generates all the hardware required for every expression in the whole program. This can mean that large parts of the hardware are idle for long periods. Shared expressions allow hardware to be shared between different parts of the program to decrease hardware usage.

The shared expression has the same format as a macro expression but does not allow recursion. You can use recursive macro expressions or `let...in` to generate recursive shared expressions.

Example

```
a = b * c;
d = e * f;
g = h * i;
```

This code generates three multipliers. Each one will only be used once and none of them simultaneously. This is a massive waste of hardware. You can improve the hardware efficiency with a shared expression:

```
shared expr mult(x, y) = x * y;

a = mult(b, c);
d = mult(e, f);
g = mult(h, i);
```

In this example, only one multiplier is built and it is used on every clock cycle. If speed is required, you can build three multipliers executing in parallel.

>: 7. Functions and macros

Warning

It is not always the case that less hardware is generated by using shared expressions because multiplexers may need to be built to route the data paths. Some expressions use less hardware than the multiplexers associated with the shared expression.

7.3.8 Using recursion to generate shared expressions

Although shared expressions cannot use recursion directly, macro expressions can be used to generate hardware which can then be shared using a shared expression. For example, to share a recursive multiplier you could write:

```
macro expr multiply(x, y) =  select(width(x) == 0, 0,
               multiply(x \\ 1, y << 1) +
               (x[0] == 1 ? y : 0));

shared expr mult(x, y) = multiply(x, y);

a = mult(b, c);
d = mult(e, f);
```

The macro expression builds a multiplier and the shared expression allows that hardware to be shared between the two assignments.

7.3.9 Restrictions on shared expressions

Shared expressions must not be shared by two different parts of the program on the same clock cycle. For example:

```
shared expr mult(x, y) = x * y;

par
{
    a = mult(b, c);
    d = mult(e, f); // NOT ALLOWED
}
```

This is not allowed because the single multiplier is used twice in the same clock cycle.

You need to ensure that shared expressions in parallel branches are not shared on the same clock cycle.

7.3.10 let ... in

`let` and `in` allow you to declare macro expressions within macro expressions. In this way, complex macros may be broken down into simple ones, whilst still being grouped together in a single block of code. They also provide easy sharing of recursive macros.

>: 7. Functions and macros

The `let` keyword starts the declaration of a local macro; the `in` keyword ends the declaration and defines its scope.

Example

```
macro expr Fred(x) =
  let macro expr y = x*2; in
   y+3;  // Returns x*2+3
```

The top line defines the macro name and parameters. The second line defines y within the macro definition. The last line expresses the value of the macro in full.

Independent `let` …in definitions

```
macro expr op(a, b) =
   let macro expr t2(x) = x * 2; in
   let macro expr d3(x) = x / 3; in
   let macro expr t4(x) = x * 4; in
      t2(a) + d3(b) + t4(a - b) + t2(b - a);
```

is equivalent to writing

```
macro expr op(a, b) = (a * 2) + (b / 3) + ((a-b) * 4) + ((b-a) * 2);
```

Related `let` …in definitions

```
macro expr op(a, b) =
   let macro expr sum(x, y) = x + y; in
      let macro expr mult(x, y) = x * sum(x, y); in
         mult(a, b) - (b * b);
```

sum is defined within the macro definition, then mult is defined using `sum`. This example is equivalent to:

```
macro expr op(a, b) = (a * (a + b)) - (b * b);
```

Shared recursive macro

A recursive multiplier illustrating the way in which `let`…in can be used to share recursive macros.

```
shared expr mult(p, q) =
   let macro expr multiply(x, y) =
      select(width(x) == 0, 0, multiply(x \\ 1, y << 1)
      + (x[0] == 1 ? y : 0)); in
      multiply(p, q)
```

>: 7. Functions and macros

Scope of definitions

The inner macros are not accessible outside the outer macro

```
{
    chanout <unsigned 16> och;
    int 16 i, j, k;
    {
        macro expr Cube(x) =
            let macro expr Sqr(x) = x*x; in
                 x * Sqr(x)
        i = Cube(3) // Correct use
        j = Sqr(3)  // Error - out of scope
    }
    k = Cube(2);    //Error - out of scope
}
```

7.3.11 Macro procedures

Macro procedures may be used to replace complete statements to avoid tedious repetition while coding. A single macro procedure can be expanded into a complex block of code. It generates the hardware for the statement each time it is referenced.

The general syntax of macro procedures is:

`macro proc` *Name*`(`*Params*`)` *Statement*

Macros may be prototyped (like functions). This allows you to declare them in one file and use them in another. A macro prototype consists of the name of the macro plus a list of the names of its parameters. E.g.

```
macro proc work(x, y);
```

If you have local or static declarations within the macro procedure, a copy of the variable will be created for each copy of the macro.

>: 7. Functions and macros

Example

```
macro proc output(x, y)
    {
        out ! x;
        out ! y;
    }

output(a + b, c * d);
output(a + b, c * d);
```

This example writes the two expressions `a+b` and `c*d` twice to the channel `out`. This example also illustrates that the statement may be a code block - in this case two instructions executed sequentially.

It expands to 4 channel output statements.

Macro procedures compared to pre-processor macros

Macro procedures differ from preprocessor macros in that they are not simple text replacements. The statement section of the definition must be a valid Handel-C statement.

The following code is valid as a `#define` pre-processor macro but not as a macro procedure:

```
#define test(x,y) if (x!=(y<<2)) // not valid as a macro procedure
as not a complete statement

test(a,b)
{
    a++;
}
else
{
    b++;
}
```

Incomplete statements will not compile as macro procedures:

```
macro proc test(x,y) if (x!=(y<<2)) // Incomplete statement, will
not compile
```

A complete statement will not successfully replace an incomplete one**:**

>: 7. Functions and macros

```
macro proc test(x,y) if (x!=(y<<2)); // Complete statement will
compile

test(a,b)  // will expand to if (x!=(y<<2));
{
    a++;
}
else      // this else has no associated if
{
    b++;
}
```

Here, the macro procedure is not defined to be a complete statement so the Handel-C compiler generates an error. This restriction provides protection against writing code which is generally unreadable and difficult to maintain.

>: 8 Introduction to timing

A Handel-C program executes with one clock source for each `main` statement. It is important to be aware exactly which parts of the code execute on which clock cycles. This is not only important for writing code that executes in fewer clock cycles but may mean the difference between correct and incorrect code when using Handel-C's parallelism. Experienced programmers can immediately tell which instructions execute on which clock cycles. This information becomes very important when your program contains multiple interacting parallel processes.

Knowing about clock cycles also becomes important when considering interfaces to external hardware. It is important to understand timing issues before moving on to implementing such interfaces because it is likely that the external device will place constraints on when signals should change.

Avoiding certain constructs has a dramatic influence on the maximum clock rate that your Handel-C program can run at.

8.1 Statement timing

The basic rule for working out the number of cycles used in a Handel-C program is:

Assignment and delay take 1 clock cycle. Everything else is free.

- One clock cycle is used every time you write an assignment statement or a `delay` statement. `releasesema` also uses one clock cycle.
  A special case statement is supported of the form:
  `a = f(x);`
  to allow function calls which take multiple clock cycles.
- Channel communications use one clock cycle if both ends are ready to communicate in the same clock domain. This is because the data from the channel must be assigned to a variable. If one of the branches is not ready for the data transfer then execution of the other branch waits until both branches become ready.
- You can write any other piece of code and not use any clock cycles to execute it.

8.1.1 Example timings

Statements

```
x = y;
x = (((y * z) + (w * v))<<2)<-7;
```

>: 8. Introduction to timing

Each of these statements takes one clock cycle.

Notice that even the most complex expression can be evaluated in a single clock cycle. Handel-C builds the combinational hardware to evaluate such expressions; they do not need to be broken down into simpler assembly instructions as would be the case for conventional C.

Parallel statements

```
par
{
    x = y;
    a = b * c;
}
```

This code executes in a single cycle because each branch of the parallel statement takes only one clock cycle. This example illustrates the benefits of parallelism. You can have as many non-interdependent instructions as you wish in the branches of a parallel statement. The total time for execution is the length of time that the longest branch takes to execute. For example:

```
par
{
    x = y;
    {
        a = b;
        c = d;
    }
}
```

This code takes two clock cycles to execute. On the first cycle, `x = y` and `a = b` take place. On the second clock cycle, `c = d` takes place. Since both branches of the `par` statement must complete before the `par` block can complete, the first branch delays for one clock cycle while the second instruction in the second branch is executed.

While loop

```
x = 5;
while (x>0)
{
    x--;
}
```

This code takes a total of 6 clock cycles to execute. One cycle is taken by the assignment of 5 to `x`. Each iteration of the `while` loop takes 1 clock cycle for the assignment of `x-1` to `x` and the loop body is executed 5 times. The condition of the `while` loop takes no clock cycles as no assignment is involved.

>: 8. Introduction to timing

For loop

```
for (x = 0; x < 5; x ++)
{
    a += b;
    b *= 2;
}
```

This code has an almost direct equivalent:

```
{
    x = 0;
    while (x<5)
    {
        a += b;
        b *= 2;
        x ++;
    }
}
```

This code takes 16 clock cycles to execute. One is required for the initialization of `x` and three for each execution of the body. Since the body is executed 5 times, this gives a total of 16 clock cycles.

Decision

```
if (a>b)
{
    x = a;
}
else
{
    x = b;
}
```

This code takes exactly one clock cycle to execute. Only one of the branches of the `if` statement is executed, either `x = a` or `x = b`. Each of these assignments takes one clock cycle. Notice again that no time is taken for the test because no assignment is made. A slightly different example is:

```
if (a>b)
{
    x = a;
}
```

>: 8. Introduction to timing

Here, if a is not greater than b, there is no `else` branch. This code therefore takes either 1 clock cycle if a is greater than b or no clock cycles if a is not greater than b.

Channels

Channel communications are more complex. The simplest example is:

```
par
{
    link ! x; // Transmit
    link ? y; // Receive
}
```

This code takes a single clock cycle to execute because both the transmitting and receiving branches are ready to transfer at the same time. All that is required is the assignment of x to y which, like all assignments, takes 1 clock cycle. A more complex example is:

```
par
{
    {               // Parallel branch 1
        a = b;
        c = d;
        link ! x;
    }

    link ? y;       // Parallel branch 2
}
```

Here, the first branch of the `par` statement takes three clock cycles to execute. However, the second branch of the `par` statement also takes three clock cycles to execute because it must wait for two cycles before the transmitting branch is ready. The usage of clock cycles is as follows:

Cycle	Branch 1	Branch 2
1	`a = b;`	`delay`
2	`c = d;`	`delay`
3	Channel output	Channel input

This approach extends to all the other Handel-C statements. See the summary of statement timings for more detail.

>: 8. Introduction to timing

8.1.2 Statement timing summary

Statement	**Timing**
`{...}`	Sum of all statements in sequential block
`par {...}`	Length of longest branch in block
Function(), `break`, `goto`, `continue`	No clock cycles
`return(`*Expression*`);`	1 clock cycle if *Expression* is assigned on return, otherwise none.
Variable = *Expression*;	1 clock cycle
Variable `++;`	1 clock cycle
Variable `--;`	1 clock cycle
++ Variable;	1 clock cycle
`--` *Variable*`;`	1 clock cycle
Variable += *Expression*`;`	1 clock cycle
Variable `-=` *Expression*`;`	1 clock cycle
Variable `*=` *Expression*`;`	1 clock cycle
Variable `/=` *Expression*`;`	1 clock cycle
Variable %= *Expression*`;`	1 clock cycle
Variable <<= Constant;	1 clock cycle
Variable >>= Constant;	1 clock cycle
Variable &= *Expression*`;`	1 clock cycle
Variable `|=` *Expression*`;`	1 clock cycle
Variable ^= *Expression*`;`	1 clock cycle
Channel ? *Variable*`;`	1 clock cycle when transmitter is ready (in same clock domain)
Channel `!` *Expression*`;`	1 clock cycle when receiver is ready (in same clock domain)
`if (`*Expression*`) {...} else {...}`	Length of executed branch
`while (`*Expression*`) {...}`	Length of loop body * number of iterations
`do {...} while (`*Expression*`);`	Length of loop body * number of iterations
`for (`*Init*`;` *Test*`;` *Iter*`) {...}`	Length of *Init* + (Length of body + length of *Iter*) * number of iterations
`switch (`*Expression*`) {...}`	Length of executed case branch

>: 8. Introduction to timing

Statement	Timing
`prialt {...}`	1 clock cycle for case communication when other party is ready plus length of executed case branch *or* length of default branch if present and no communication case is ready *or* infinite if no default branch and no communication case is ready
`releasesema();`	1 clock cycle
`delay;`	1 clock cycle

The Handel-C compiler may insert `delay` statements to break combinational loops.

8.2 Avoiding combinational loops

If you wish to wait for a variable to be modified in a parallel process before continuing, you might write:

```
while (x!=3); // WARNING!!
```

This is bad Handel-C code because it generates a combinational loop in the logic (This is because of the way that Handel-C expressions are built to evaluate in zero clock cycles.)

This is easier to see if it is written as

```
while (x!=3)
{
            // wait until x == 3
}
```

This empty loop must be broken by changing the code to:

```
while (x!=3)
{
    delay;
}
```

This code takes no longer to execute but does not contain a combinational loop because of the clock cycle delay in the loop body.

The Handel-C compiler spots this form of error, inserts the `delay` statement, and generates a warning. It is considered better practice to include the `delay` statement in the code to make it explicit

>: 8. Introduction to timing

Similar problems occur with do ... `while` loops and `switch` statements in similar circumstances. `for` loops with no iteration step can also cause combinational loops.

Further combinational loop code example

Code may look correct but still include an empty loop. For example:

```
while (x!=3)
{
    if (y>z)
    {
        a++;
    }
}
```

This `if` statement may take zero clock cycles to execute if y is not greater than z so even though this loop body does not look empty a combinational loop is still generated. This is more obvious written as

```
while (x!=3)
{
    if (y>z)
    {
        a++;
    }
    else
    {
           // do nothing
    }
}
```

The solution is to add the `else` part of the if construct as follows:

```
while (x!=3)
{
    if (y>z)
    {
        a++;
    }
    else
    {
        delay;
    }
}
```

>: 8. Introduction to timing

8.3 Parallel access to variables

The rules of parallelism state that the same variable must not be accessed from two separate parallel branches. This avoids resource conflicts on the variables.

The rule may be relaxed to state that the same variable must not be assigned to more than once on the same clock cycle but may be read as many times as required. This gives powerful programming techniques. For example:

```
par
{
    a = b;
    b = a;
}
```

This code swaps the values of a and b in a single clock cycle.

Since exact execution time may be run-time dependent, the Handel-C compiler cannot determine when two assignments are made to the same variable on the same clock cycle. You should therefore check your code to ensure that the relaxed rule of parallelism is still obeyed.

Example

Using this technique, a four-place queue can be written:

```
while(1)
{
    par
    {
        int x[3];

        x[0] = in;
        x[1] = x[0];
        x[2] = x[1];
        out = x[2];
    }
}
```

The value of `out` is the value of `in` delayed by 4 clock cycles. On each clock cycle, values will move one place through the x array. For example:

>: 8. Introduction to timing

Clock	in	x[0]	x[1]	x[2]	out
1	5	0	0	0	0
2	6	5	0	0	0
3	7	6	5	0	0
4	8	7	6	5	0
5	9	8	7	6	5
6	10	9	8	7	6
7	11	10	9	8	7
8	12	11	10	9	8
9	13	12	11	10	9

8.4 Detailed timing example

This is an analyzed example that generates signals tied to real-world constraints. It shows the generation of signals for a real time clock. The signals required are for microseconds, seconds, minutes and hours.

The hardware generated will eventually be driven from an external clock. In order to write the program, the rate of this clock must be known. It has been assumed to be 5 MHz on pin P1.

The loop body takes one clock cycle to execute. The `Count` variable is used to divide the clock by 5 to generate microsecond increments. As each variable wraps round to zero, the next time step up is incremented.

```
set clock = external "P1";
void main(void)
{
    unsigned 20 MicroSeconds;
    unsigned 6 Seconds;
    unsigned 6 Minutes;
    unsigned 16 Hours;
    unsigned 3 Count;
```

>: 8. Introduction to timing

```
    par
    {
        Count = 0;
        MicroSeconds = 0;
        Seconds = 0;
        Minutes = 0;
        Hours = 0;
    }
    while (1)
    {
        if (Count!=4)
            Count++;
        else
            par
            {
                Count = 0;
                if (MicroSeconds!=999999)
                    MicroSeconds++;
                else
                    par
                    {
                        MicroSeconds = 0;
                        if (Seconds!=59)
                            Seconds++;
                        else
                            par
                            {
                                Seconds = 0;
                                if (Minutes!=59)
                                    Minutes++;
                                else
                                    par
                                    {
                                        Minutes = 0;
                                        Hours++;
                                    }
                            }
                    }
            }
    }
}
```

>: 8. Introduction to timing

8.5 Time efficiency of Handel-C hardware

Handel-C requires that the clock period for a program is longer than the longest path through combinational logic in the whole program. This means that, for example, once FPGA or PLD place and route has been completed, the maximum clock rate for the system can be calculated from the reciprocal of the longest path delay in the circuit.

For example, suppose the FPGA place and route tools calculate that the longest path delay between flip-flops in a design is 70ns. The maximum clock rate that that circuit should be run at is then 1/70ns = 14.3MHz.

If this calculated rate is not fast enough for the system performance or real time constraints you can optimize your program to reduce the longest path delay and increase the maximum possible clock rate.

One standard technique for optimizing efficiency is to use pipelining.

8.5.1 Reducing logic depth

Certain operations in Handel-C combine to produce deep logic. Deep logic results in long path delays in the final circuit so reducing logic depth should increase clock speed.

Guidelines for reducing logic depth

- Division and modulo operators produce the deepest logic. Multiplication also produces deep logic. A single cycle divide, mod or multiplier produces a large amount of hardware and long delays through deep logic so you should avoid using them wherever possible.
- Most common division and multiplications can be done with the shift operators. Also consider using a long multiplication with a loop, shift and add routine or a pipelined multiplier.
- Most common modulo operations can be done with the AND operator.
- Wide adders require deep logic for the carry ripple. Consider using more clock cycles with shorter adders.
- Avoid greater than and less than comparisons - they produce deep logic.
- Reduce complex expressions into a number of stages.
- Avoid long strings of empty statements. Empty statements result from, for example, missing `else` conditions from `if` statements.

To reduce a single, 8-bit wide adder to 3, narrower adders:

>: 8. Introduction to timing

```
unsigned 8 x;
unsigned 8 y;
unsigned 5 temp1;
unsigned 4 temp2;

par
{
    temp1 = (0@(x<-4)) + (0@(y<-4));
    temp2 = (x \\ 4) + (y \\ 4);
}
x = (temp2+(0@temp1[4])) @ temp1[3:0];
```

Comparison example

```
while (x<y)
{
    ......
    x++;
}
```

can be replaced with:

```
while (x!=y)
{
    ......
    x++;
}
```

The == and != comparisons produce much shallower logic although in some cases it is possible to remove the comparison altogether. Consider the following code:

```
unsigned 8 x;

x = 0;
do
{
    ......
    x = x + 1;
} while (x != 0);
```

This code iterates the loop body 256 times but it can be re-written as follows:

>: 8. Introduction to timing

```
unsigned 9 x;

x = 0;
do
{
    ......
    x = x + 1;
} while (!x[8]);
```

By widening x by a single bit and just checking the top bit, we have removed an 8-bit comparison.

Complex expression example

```
x = a + b + c + d + e + f + g + h;
```

reduces to:

```
par
{
    temp1 = a + b;
    temp2 = c + d;
    temp3 = e + f;
    temp4 = g + h;
}
par
{
    temp1 = temp1 + temp2;
    temp3 = temp3 + temp4;
}
x = temp1 + temp3;
```

This code takes three clocks cycles as opposed to one but each clock cycle is much shorter and so the rest of the circuit should be speeded up by the faster clock rate permitted.

Empty statement example

```
if (a>b)
    x++;
if (b>c)
    x++;
if (c>d)
    x++;
```

```
if (d>e)
    x++;
if (e>f)
    x++;
```

If none of these conditions is met then all the comparisons must be made in one clock cycle. By filling in the `else` statements with `delays`, the long path through all these `if` statements can be split at the expense of having each `if` statement take one clock cycle whether the condition is true or not.

8.5.2 Pipelining

A classic way to increase clock rates in hardware is to pipeline. A pipelined circuit takes more than one clock cycle to calculate any result but can produce one result every clock cycle. The trade off is an increased latency for a higher throughput so pipelining is only effective if there is a large quantity of data to be processed: it is not practical for single calculations.

Pipelined multiplier example

```
unsigned 8 sum[8];
unsigned 8 a[8];
unsigned 8 b[8];
chanin inputa with {infile = "ina.dat"};
        //dummy data file. User must provide their own
chanin inputb with {infile = "ina.dat"};
        //dummy data file. User must provide their own
chanout output with {outfile = "out.dat"};


par
{
    while(1)
        inputa ? a[0];

    while(1)
        inputb ? b[0];

    while(1)
        output ! sum[7];
```

>: 8. Introduction to timing

```
    while(1)
    {
        par
        {
            macro proc level(x)
                par
                {
                    sum[x] = sum[x - 1] +
                             ((a[x][0] == 0) ? 0 : b[x]);
                    a[x] = a[x - 1] >> 1;
                    b[x] = b[x - 1] << 1;
                }

            sum[0] = ((a[0][0] == 0) ? 0 : b[0]);
           par ( i=1; i <=7; i++)
           {
                level (i);
           }
        }
    }
}
```

This multiplier calculates the 8 LSBs of the result of an 8-bit by 8-bit multiply using long multiplication. The multiplier produces one result per clock cycle with a latency of 8 clock cycles. This means that although any one result takes 8 clock cycles, you get a throughput of 1 multiply per clock cycle. Since each pipeline stage is very simple, combinational logic is shallow and a much higher clock rate is achieved than would be possible with a complete single cycle multiplier.

At each clock cycle, partial results pass through each stage of the multiplier in the `sum` array. Each stage adds on 2^n multiplied by the b operand if required. The LSB of the a operand at each stage tells the multiply stage whether to add this value or not. Stages are generated with a macro procedure instantiated several times using a replicator

Operands are fed in on every clock cycle through `a[0]` and `b[0]`. Results appear 8 clock cycles later on every clock cycle through `sum[7]`.

>: 9. Clocks

>: 9 Clocks

You can have multiple clocks interfacing with your design. Each `main()` function must be associated with a single clock.

Clocks may be generated internally, fed from expressions, or fed from a pin (external clocks).

The current clock may be referred to using the keyword `__clock`

You can specify the maximum delay in ns allowed between components fed from a clock by using the rate specification.

The general syntax of the clock specification is:

`set clock =` *Location* `with {`*RateSpec*`};`

Communicating between multiple clock domains

It is not legal to access the same variable from different clock domains since there are metastability issues around such communications. Instead, you must transmit data between clock domains using a channel or a port.

9.1 Locating the clock

Since each Handel-C `main()` code block generates synchronous hardware, a single clock source is required for each one.

The general syntax of the clock specification is:

`set clock =` *Location*`;`

Location may be any of the following:

Location	Meaning
`internal` *Frequency*	Clock from internal clock generator (Xilinx 4000 series devices only)
`internal_divide` *Frequency Factor*	Clock from internal clock generator with integer division (Xilinx 4000 series devices only)
`internal` *Expression*	Clock from expression
`internal_divide` *Expression Factor*	Clock from expression with integer division
`external` [*Pin*]	Clock from device pin
`external_divide` [*Pin*] *Factor*	Clock from device pin with integer division

>: 9. Clocks

9.1.1 External clocks

External clocks may be accessed by associating the clock with a specific pin using `set clock external = "`*pin_Name*`"` or `set clock external_divide = "`*pin_Name*`"` *factor*, where the `external_divide` keyword is a constant integer. For example:

```
set clock = external "P35";
set clock = external_divide "P35" 3;
set clock = external_divide 3;
```

The first of these examples specifies a clock taken from pin P35. The second specifies a clock taken from pin P35 which is divided on the FPGA by a factor of 3. The third option shows a clock divided by 3 with no pin number specified.

When the pin number is omitted, the place and route tools will choose an appropriate pin. Omitting pin specifications can speed up the clock rate of the design.

You can also define an `interface` that reads an external clock. If the clock is associated with a specific pin, you can use the `interface` sort `bus_in`. You would only need to do this if the external clock has been divided, otherwise you can use the intrinsic `__clock`.

Example

```
interface bus_in(unsigned 1 in with {clockport=1}) InputBus()
                with {data={"Pin1"}};
set clock = external_divide "Pin1" 4;
```

You can now use `InputBus.in` to get an undivided external clock.

9.1.2 Internal clocks fed from expressions

You can set the clock to be any expression or any expression divided by a given factor.

`set clock = internal` *<Expression>*`;`

`set clock = internal_divide` *<Expression> factor*`;`

The clock division factor specified with the `internal_divide` keyword must be a constant integer.

Example

This allows you to set the clock to a value read from an `interface`.

```
interface port_in(unsigned 1 clk with {clockport = 1}) ClockPort();
set clock = internal ClockPort.clk;
```

>: 9. Clocks

9.1.3 Internally generated clocks

For Xilinx 4000 series chips, you can set the clock to be a value read from the on-chip clock generator.

`set clock = internal` *Frequency*`;`

`set clock = internal_divide` *Frequency Factor*`;`

The clock division factor specified by *Factor* must be a constant integer.

For example:

```
set clock = internal "F8M";
set clock = internal_divide "F8M" 3;
```

The frequency of the internal clock may take one of the following values:

Specification string	Frequency
"F15"	15Hz
"F490"	490Hz
"F16K"	16kHz
"F500K"	500kHz
"F8M"	8MHz

The tolerance for these values is -50% to +25% so you should not rely on the internal clock being at exactly these frequencies.

9.2 Current clock

The current clock used by a function can be referenced using the keyword `__clock`. This allows the function to pass the current clock to an external interface. The value of the system variable `__clock` will be the value after any divide. The clock may be an internal or an external clock.

Example

The code below shows the assignment of the current clock to a port in an interface.

```
interface reg32x1k() registers(unsigned addr=address,
     unsigned data=data_in, unsigned 1 clk = __clock,
     unsigned out = write);
```

>: 9. Clocks

9.3 Channels communicating between clock domains

Channels that connect between clock domains must be uni-directional point-to-point. This means that their first use defines their direction and the domains in which they transmit and receive. If you attempt to re-use the channel in a different direction or to or from a different clock domain, the compiler generates an error.

Channels used between clock domains must be defined in one file and then declared as `extern` in another.

The timing between domains is unspecified, but the transmission is guaranteed to occur, and both sides will wait until the transmission has completed. For example:

```
//File: transmit.hcc
chan 8 c; // channel must have global scope

main()
{
 int 8 x, y;
 c ! x; //program will wait until data
    //successfully transmitted
 c ! y;
}

//File: receive.hcc
extern chan c;

main()
{
 int 8 p, q;

 c ? p;
 c ? q;
}
```

>: 10. Targeting hardware

>: 10 Targeting hardware

10.1 Interfacing with the simulator

Communication with the simulator takes place over channels. They are declared using the keywords `chanin` and `chanout`. Standard channel communication statements can then be used to transfer data. It is assumed that channels to and from the simulator never block and will always complete a transfer in one clock cycle.

Channels to and from the simulator are declared using `chanin` and `chanout` instead of `chan`.

The special channels `chanin` and `chanout` are normally connected to files. An unconnected channel that outputs data to the simulator will be displayed in the debug window. You can declare multiple channels for input and output.

If the simulation is still running when the end of the file has been reached, the simulator will read in zeroes.

Simple example

```
chanin unsigned Input with {infile = "../Data/source.dat"};
chanout unsigned Output;

input ? x;
output ! y;
```

This example declares two channels: one for input from the simulator and one for output to the simulator. The input channel connects to a file managed by the simulator; the output channel connects to the simulator's standard output (the debug window in the DK1 GUI).

Multiple channel example

```
chanin int 8 input_1 with
     {infile = "../Data/source_1.dat"};
chanin int 16 input_2 with
     {infile = "../Data/source_2.dat"};
chanout unsigned 3 output_1;
chanout char output_2;


int 8 a;
int 16 b;
```

```
input_1 ? a;
input_2 ? b;
output_1 ! (unsigned 3)(((0 @ a) + b) <- 3);
output_2 ! a;
```

When simulated, such a program displays the name of channels before outputting their value on the simulating computer screen.

10.1.1 Simulator input file format

The data input file should have one number per line separated by newline characters (either DOS or UNIX format text files may be used). Each number may be in any format normally used for constants by Handel-C. Blank lines are ignored as are lines prefixed by `//` (comments). For example:

```
56
0x34
0654
0b001001

//is a comment, blank lines ignored
27
```

If EOF file is reached while reading an input file, zeroes will be read in until the simulation completes.

10.1.2 Block data transfers

The Handel-C simulator has the ability to read data from a file and write results to another file. For example:

```
chanin int 16 input with {infile = "in.dat"};
chanout int 16 output with {outfile = "out.dat"};

void main (void)
{
    while (1)
    {
        int value;

        input ? value;
        output ! value+1;
    }
}
```

>: 10. Targeting hardware

This program reads data from the file `in.dat` and writes its results to the file `out.dat`. The simulator will open and close the specified files for reading or writing as appropriate. If EOF file is reached while reading an `infile` file, zeroes will be read in until the simulation completes.

If the `in.dat` file consists of:

```
56
0x34
0654
0b001001
```

the `out.dat` will contain the decimal results as follows:

```
57
53
429
10
```

The `base` specification can be used to write to the outfile in different formats.

Block data transfers allow algorithms to be debugged and tested without needing to build actual hardware. For example, an image processing application may store a source image in a file and place its results in a second file. All that need be done outside the Handel-C compiler is a conversion from the image (e.g. JPEG file) into the text file (which can then be used by the simulator) and a conversion back from the output file to the image format. The results can then be viewed and the correct operation of the Handel-C program confirmed.

10.2 Targeting FPGA and PLD devices

The Handel-C language is designed to target real hardware devices. To do this, you must supply this information to the compiler:

- the FPGA/PLD family and part that the design will be implemented in

  These are supplied through the **Project**>**Settings** dialog. They can be supplied to the via the source code using the `set` statement or they can be supplied to the command line using the -f (family) and -p (part) switches. They will be passed to the FPGA/PLD place and route tool to inform it of the device it should target.

- the location of a clock source

  The clock source is specified using the `set` command.

>: 10. Targeting hardware

10.2.1 Summary of supported devices

In order to target a specific FPGA or PLD, the compiler must be supplied with the part number. Ultimately, this information is passed to the place and route tool to inform it of the device it should target.

You can specify your target device using the **Chip** tab on the **Project Settings** dialog, or within your source code.

Recognized families are:

Description	On-chip asynchronous RAMs	On-chip synchronous RAMs
Xilinx 4000E series FPGAs	SelectRAM, dual-port	-
Xilinx 4000L series FPGAs	SelectRAM, dual-port	-
Xilinx 4000EX series FPGAs	SelectRAM, dual-port	-
Xilinx 4000XL series FPGAs	SelectRAM, dual-port	-
Xilinx 4000XV series FPGAs	SelectRAM, dual-port	-
Xilinx Spartan series FPGAs	SelectRAM, dual-port	-
Xilinx Spartan XL series FPGAs	SelectRAM, dual-port	-
Xilinx SpartanII series FPGAs	SelectRAM, dual-port	Block RAM
Xilinx Virtex series FPGAs	SelectRAM, dual-port	Block RAM, dual-port
Xilinx VirtexE series FPGAs	SelectRAM, dual-port	Block RAM, dual-port
Xilinx Virtex-II series FPGAs	SelectRAM, dual-port	Block RAM, dual-port
Xilinx Virtex-II Pro series FPGAs	SelectRAM, dual-port	Block RAM, dual-port
Altera Apex 20K series PLDs	Block RAM (in ESBs), dual-port	Block RAM (in ESBs), dual-port
Altera Apex 20KE series PLDs	Block RAM (in ESBs), dual port	Block RAM (in ESBs), dual port
Altera Apex 20KC series PLDs	Block RAM (in ESBs), dual port	Block RAM (in ESBs), dual port
Altera ApexII series PLDs	Block RAM (in ESBs), dual-port	Block RAM (in ESBs), dual-port
Altera Excalibur ARM series PLDs	Block RAM (in ESBs), dual-port	Block RAM (in ESBs), dual-port
Altera Flex10K series PLDs	Block RAM (in EABs), dual-port	Block RAM (in EABs), dual-port

Description	On-chip asynchronous RAMs	On-chip synchronous RAMs
Altera Flex10KA series PLDs	Block RAM (in EABs), dual-port	Block RAM (in EABs), dual-port
Altera Flex10KB series PLDs	Block RAM (in EABs), dual-port	Block RAM (in EABs), dual-port
Altera Flex10KE series PLDs	Block RAM (in EABs), dual-port	Block RAM (in EABs), dual-port
Altera Mercury series ASSPs	Block RAM (in ESBs), dual-port, quad-port	Block RAM (in ESBs), dual-port, quad-port
Actel eX series FPGAs	None	None
Actel 54SX series FPGAs	None	None
Actel 54SX-A series FPGAs	None	None
Actel RT54SX series FPGAs	None	None
Actel RT54SX-S series FPGAs	None	None
Actel ProASIC series FPGAs	Block RAM, dual-port	Block RAM, dual-port
Actel ProASIC+ series FPGAs	Block RAM, dual-port	Block RAM, dual-port

10.2.2 Targeting specific devices via source code

If you are not using the GUI to specify the target device, you must insert lines in the code to specify it. In order to target a specific FPGA or PLD, the compiler must be supplied with the FPGA part number. Ultimately, this information is passed to the FPGA/PLD place and route tool to inform it of the device it should target.

Targeting devices is in two parts: specifying the target family and the target device. The general syntax is:

```
set family = Family;
set part = Chip Number;
```

Recognized families are:

Family Name	Description
`Xilinx4000E`	4000E series Xilinx FPGAs
`Xilinx4000L`	4000L series Xilinx FPGAs
`Xilinx4000EX`	4000EX series Xilinx FPGAs
`Xilinx4000XL`	4000XL series Xilinx FPGAs

>: 10. Targeting hardware

Family Name	Description
`Xilinx4000XV`	4000XV series Xilinx FPGAs
`XilinxVirtex`	Virtex Xilinx FPGAs
`XilinxVirtexE`	VirtexE Xilinx FPGAs
`XilinxVirtexII`	Virtex-II Xilinx FPGAs
`XilinxVirtexIIPro`	Virtex-II Pro Xilinx FPGAs
`XilinxSpartan`	Spartan Xilinx FPGAs
`XilinxSpartanXL`	SpartanXL Xilinx FPGAs
`XilinxSpartanII`	SpartanII Xilinx FPGAs
`AlteraFlex10K`	Flex10K series Altera PLDs
`AlteraFlex10KA`	Flex10KA series Altera PLDs
`AlteraFlex10KB`	Flex10KB series Altera PLDs
`AlteraFlex10KE`	Flex10KE series Altera PLDs
`AlteraApex20K`	Apex 20K series Altera PLDs
`AlteraApex20KE`	Apex 20KE series Altera PLDs
`AlteraApex20KC`	Apex 20KC series Altera PLDs
`AlteraApexII`	Apex II series PLDs
`AlteraMercury`	Altera Mercury series PLDs
`AlteraExcaliburARM`	Altera Excalibur ARM series PLDs
`ActelEX`	Actel eX series FPGAs
`Actel54SX`	Actel 54SX series FPGAs
`Actel54SXA`	Actel 54SX-A series FPGAs
`ActelRT54SX`	Actel RT54SX series FPGAs
`ActelRT54SXS`	Actel RT54SX-S series FPGAs
`Actel500K`	Actel ProASIC series FPGAs
`ActelPA`	Actel ProASIC+ series FPGAs
Deprecated	
`Altera10K`	Flex10K series Altera PLDs
`Altera10KA`	Flex10KA series Altera PLDs
`Altera10KB`	Flex10KB series Altera PLDs
`Altera10KE`	Flex10KE series Altera PLDs
`Altera20K`	Flex20K series Altera PLDs
`Altera20KE`	Flex20KE series Altera PLDs

>: 10. Targeting hardware

The part string is the complete Xilinx, Altera or Actel device string. For example:

```
set family = Xilinx4000E;
set part = "4010EPC84-1";
```

This instructs the compiler to target a XC4010E device in a PLCC84 package. It also specifies that the device is a -1 speed grade. This last piece of information is required for the timing analysis of your design by the Xilinx tools.

The family is used to inform the compiler of which special blocks it may generate.

To target Altera Flex 10K devices:

```
set family = AlteraFlex10K;
set part = "EPF10K20RC240-3";
```

This instructs the compiler to target an Altera Flex 10K20 device in a RC240 package. It also specifies that the device is a -3 speed grade. This last piece of information is required for the timing analysis of your design by the Altera Max Plus II or Quartus tools. Note that when performing place and route on the resulting design, the device and package must also be selected via the menus in the Max Plus II or Quartus software.
To target Actel RT54SX-S devices:

```
set family = ActelRT54SXS;
set part = "RT54SX32S-1CQ256B";
```

This instructs the compiler to target an Actel RT54SX-S device with 32,000 gates in a CQ256 package (CQFP with 256 pins). It also specifies that the device is a -1 speed grade, and that the device is to be used for a military application: the “B” at the end of the part string specifies that the device is to conform to military temperature range standards. The speed information is required for the timing analysis of your design by the Actel Designer tools. The application information (“military” in this example) is required for place and route of your design by the Actel Designer tools. Note that when performing place and route on the resulting design, the device and package must also be selected via the menus in the Designer software.

10.2.3 Specifying a global reset

`set reset` allows you to reset your device into a known state without reconfiguring it. You can also use it to set up devices which are not in a known state at start up.

`set reset` causes the program to return to its initial state and resets variables to their initial values. However, it does not reset any RAMs (distributed or block).

set reset = internal <*expression*>;

set reset = external <*Pin*>;

`reset` is active high.

>: 10. Targeting hardware

Examples

```
signal x, y;
set reset = internal x[0] & y[0];

set reset = external "P1"; // resets when a signal sent to named pin

set reset = external; // connects to a pin, but doesn't specify
which
```

Current reset value

The current reset state can be referenced using the `__reset` keyword. You can use the `__reset` keyword to pass a reset condition to a black box.

For example:

```
set reset = external "P1";

interface UserBlock(unsigned 1 Status)
        UserBlockInstance(unsigned 1 reset_port = __reset);
```

You must specify a reset pin using set reset if you are targeting Altera devices.

10.3 Use of RAMs and ROMs with Handel-C

Handel-C provides support for:

- interfacing to on-chip and off-chip RAMs and ROMs using the `ram` and `rom` keywords.
- specifying RAMs and ROMs external to the Handel-C code by using the `ports` specification keyword.
- controlling the timing for read/write cycles by using specification keywords that define the relationship between the RAM strobe and the Handel-C clock.

The usual technique for specifying timing in synchronous and asynchronous RAM is to have a fast external clock which is divided down to provide the Handel-C clock and used directly to provide the pulses to the RAM.

10.4 Asynchronous RAMs

There are three techniques for timing asynchronous RAMs, depending on the clock available

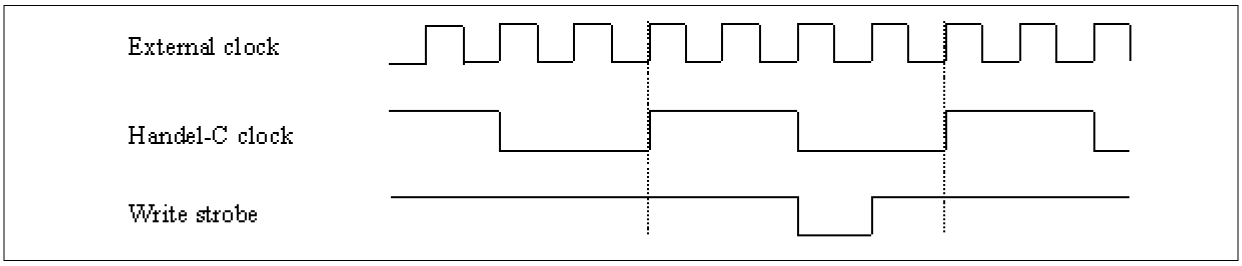
- Fast external clock. Use the Handel-C `westart` and `welength` specifications to position the write strobe.
- External clock at the same speed as the Handel-C clock. Use multiple reads to give the RAM enough time to respond.
- Use the `wegate` specification to position the write enable signal within the Handel-C clock.

10.4.1 Fast external clock

This method of timing asynchronous RAMs depends on having an external clock that is faster than the internal clock (i.e. the location of the clock is `internal_divide` or `external_divide` with a division factor greater than 1). If so, Handel-C can generate a write strobe for the RAM which is positioned within the Handel-C clock cycle. This is done with the `westart` and `welength` specifications. For example:

```
set clock = external_divide "P78" 4;
ram unsigned 6 x[34] with { westart = 2,
                            welength = 1 };
```

The write strobe can be positioned relative to the Handel-C clock cycle by half cycle lengths of the external (undivided) clock. The above example starts the pulse 2 whole external clock cycles into the Handel-C clock cycle and gives it a duration of 1 external clock cycle. Since the external clock is divided by a factor of 4, this is equivalent to a strobe that starts half way through the internal clock cycle and has a duration of one quarter of the internal clock cycle. This signal is shown below:



TIMING DIAGRAM: POSITIONED WRITE STROBE

This timing allows half a clock cycle for the RAM setup time on the address and data lines and one quarter of a clock cycle for the RAM hold times. This is the recommended way to access asynchronous RAMs.

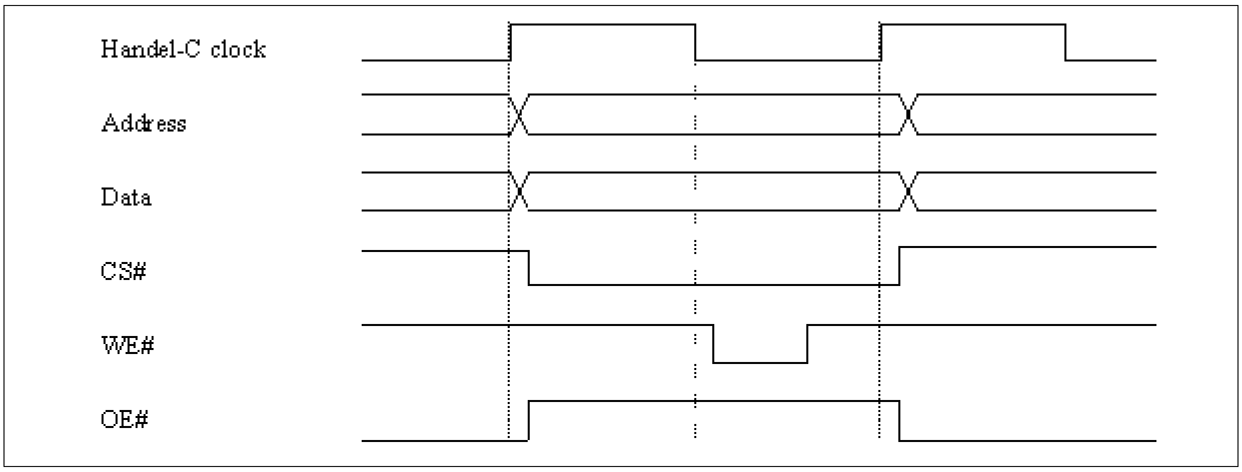
>: 10. Targeting hardware

10.4.2 Asynchronous RAMs: fast external clock example

To declare a 16Kbyte by 8-bit RAM:

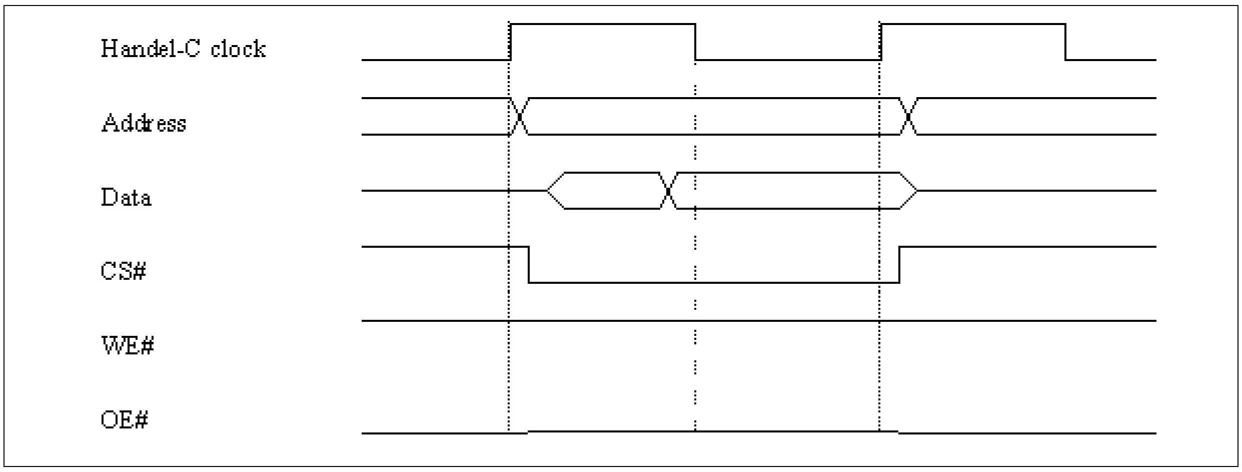
```
set clock = external_divide "P99" 4;

ram unsigned 8 ExtRAM[16384] with {
       offchip = 1,
       westart = 2,
       welength = 1,
       data = {"P1", "P2", "P3", "P4",
               "P5", "P6", "P7", "P8"},
       addr = {"P9", "P10", "P11", "P12",
               "P13", "P14", "P15", "P16",
               "P17", "P18", "P19", "P20",
               "P21", "P22"},
       we = {"P23"},
       oe = {"P24"},
       cs = {"P25"}};
```

The compiled hardware generates the following cycle for a write to external RAM:



>: 10. Targeting hardware

The compiled hardware generates the following cycle for a read from external RAM:



10.4.3 Same rate external clock

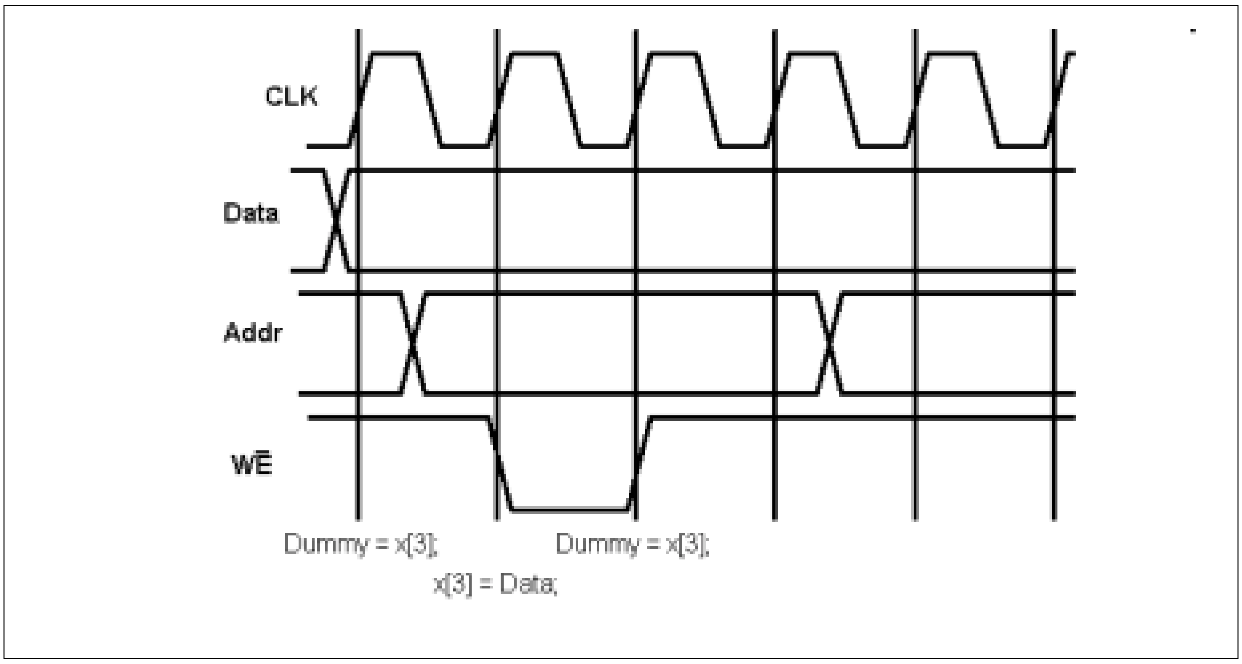
This method of timing asynchronous RAMs uses multiple Handel-C RAM accesses to meet the setup and hold times of the RAM.

```
ram unsigned 6 x[34];

Dummy = x[3];
x[3] = Data;
Dummy = x[3];
```

This code holds the address constant around the RAM write cycle, enabling a write to an asynchronous RAM.

The timing diagram below shows the address being held constant during the write strobe. It is held constant by the two assignments to `Dummy`.

>: 10. Targeting hardware



10.4.4 Undivided external clock

This method of accessing asynchronous RAMs is a compromise between the other two methods (fast external clock and multiple RAM accesses). `wegate` is used with an undivided external clock and keeps the write strobe in the first or second half of the clock cycle. It is still necessary to hold the address constant either in the clock cycle before or in the clock cycle after the access. For example:

```
ram unsigned 6 x[34] with { wegate = 1 };

x[3] = Data;
Dummy = x[3];
```

This places the write strobe in the second half of the clock cycle (use a value of -1 to put it in the first half) and holds the address for the clock cycle after the write. The RAM therefore has half a clock cycle of setup time and one clock cycle of hold time on its address lines.

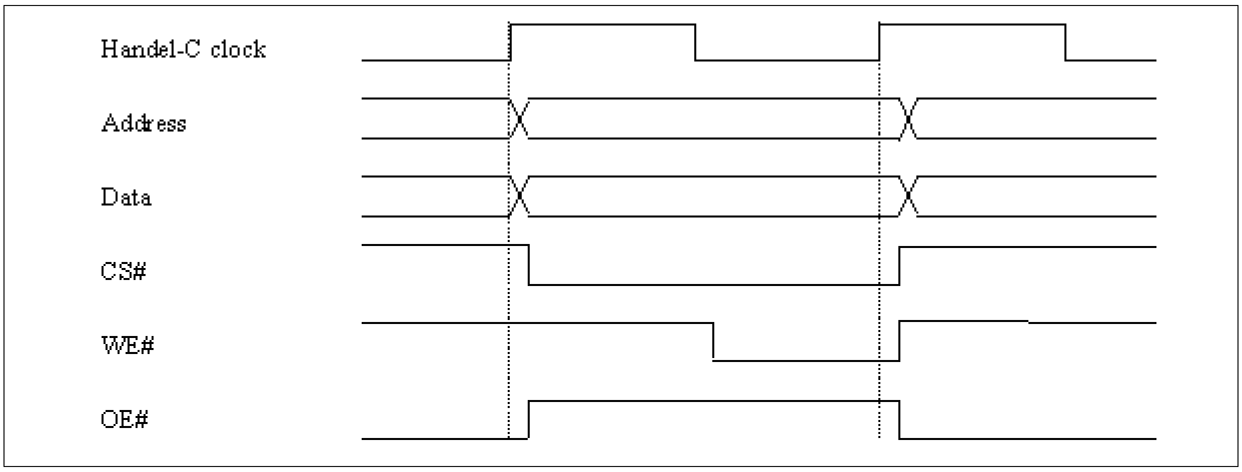
10.4.5 Asynchronous RAMs: wegate example

The `wegate` specification may be used when a divided clock is not available. For example, to declare a 16Kbyte by 8-bit RAM:

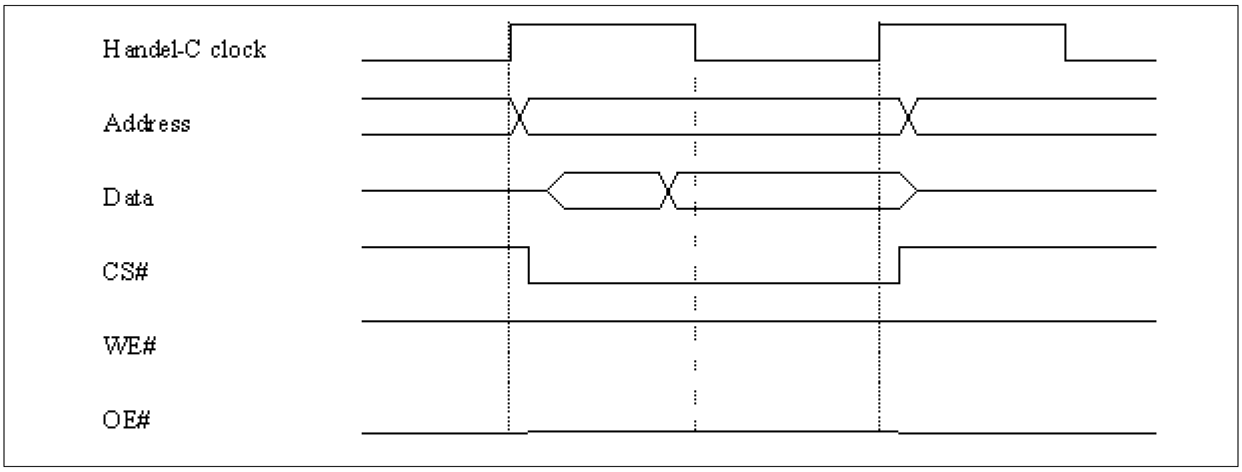
>: 10. Targeting hardware

```
ram unsigned 8 ExtRAM[16384] with {
       offchip = 1,
       wegate = 1,
       data = {"P1", "P2", "P3", "P4",
               "P5", "P6", "P7", "P8"},
       addr = {"P9", "P10", "P11", "P12",
               "P13", "P14", "P15", "P16",
               "P17", "P18", "P19", "P20",
               "P21", "P22"},
       we = {"P23"},
       oe = {"P24"},
       cs = {"P25"}};
```

The compiled hardware generates the following cycle for a write to external RAM:



The compiled hardware generates the following cycle for a read from external RAM:



Note that the timing diagram above may violate the hold time for some asynchronous RAM devices. If the delay between rising clock edge and rising write enable is longer than the delay between rising clock edge and the change in data or address then corruption in the write may occur in these devices. The two cycle access does not solve the problem since it is not possible to hold the data lines constant beyond the end of the clock cycle. If this causes a problem then a multiplied external clock must be used as described above.

Using the wegate specification may violate the hold time for some asynchronous RAM devices.

10.5 Synchronous RAMs

SSRAM clocks

Handel-C timing semantics require that any assignment takes one clock cycle. Typically, SSRAMs have a latency of at least one clock cycle. Therefore, in order for accesses to a SSRAM device to conform to Handel-C's one-clock-cycle-per-assignment rule, the SSRAM clock needs to be offset from the Handel-C clock. If the SSRAM has a latency of more than one clock cycle, its clock needs to be faster than the Handel-C clock, as well as being offset from it.

This is done by using an independent fast clock (RAMCLK) to match the SSRAM timings with the Handel-C timing constraints.

A fast external clock (CLK) is divided to provide the Handel-C clock (HCLK), and is also used to generate pulses to clock the SSRAM, where the pulses can be placed within a single HCLK cycle. This placed clock is the RAMCLK. It can be carried to an external SSRAM using the `clk` specification.

By default, the Handel-C compiler uses an inverted copy of the Handel-C clock to drive synchronous on-chip memories. This may mean you need to run your design at a lower clock frequency than you want to. You can increase the efficiency of your design by using the clock position specifications to alter the position of the RAM clock relative to the Handel-C clock. For example, you might want to advance the write-clock, or delay the read-clock.

SSRAM devices supported

Handel-C supports ZBT-compatible (Zero Bus Turnaround) flow-through and pipelined output devices. DDR (double data rate) and QDR (quad data rate) devices are not supported directly; you can write your own interfaces.

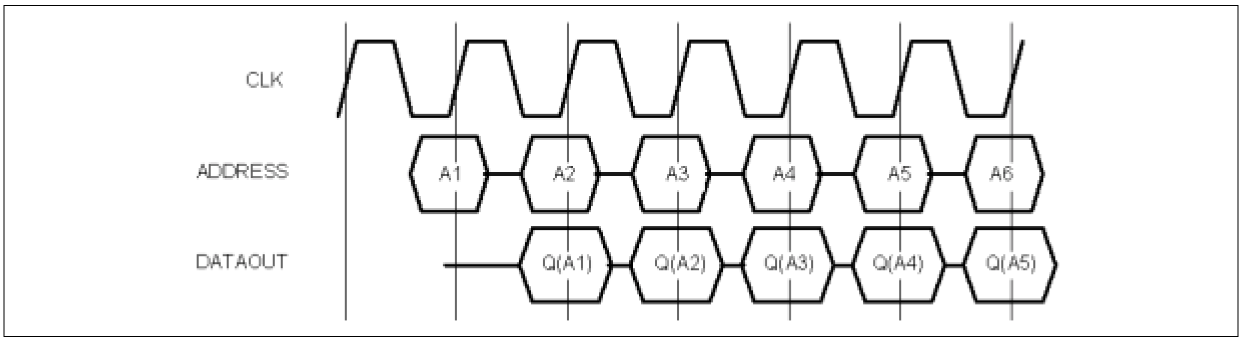
SSRAM write-enable

The Handel-C compiler checks the `block` and `offchip` specifications to find out what type of RAM is being built and generates the appropriate write-enable signal (e.g. active low for ZBT SSRAM devices and active-high for block RAMs within Xilinx Virtex chips).

>: 10. Targeting hardware

10.5.1 SSRAM read and write cycles

The inputs to most inputs to SSRAMs are captured on the rising edge of the input clock. During a read cycle there is a latency of at least one clock cycle between an address being captured at the input and data becoming available at the output. This is also true for the write cycle in many devices: an address is captured on one clock cycle, and data on the next. A diagram of a typical timing for a read (or write) cycle for an SSRAM device is shown below.



TIMING DIAGRAM: SSRAM READ AND WRITE

10.5.2 Specifying SSRAM timing

You can place the RAM clock pulses at different points within the Handel-C clock if a fast divided clock is available. If you have a fast undivided clock CLK, a divided clock HCLK, and you want to generate a RAM clock RAMCLK, the following applies:

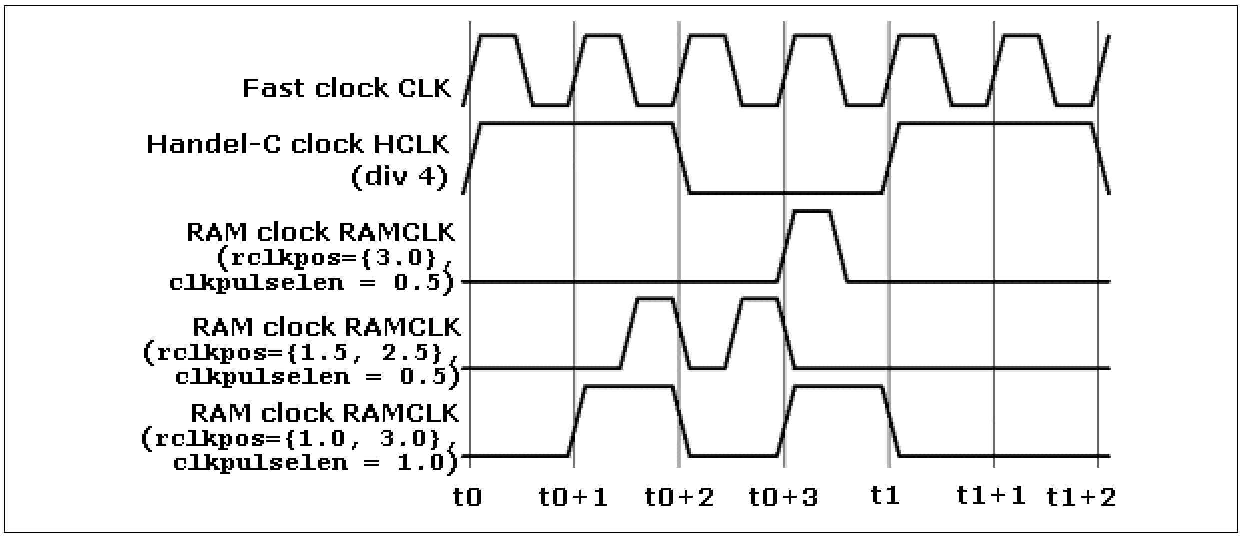
The SSRAM clock (RAMCLK) is generated from the fast clock (CLK) according to the specifications: `rclkpos` , `wclkpos` and `clkpulselen`. These specifications can be in whole or half cycles of the external clock (i.e. the specifications are in multiples of 0.5).

`rclkpos` specifies the positions of the clock cycles of clock RAMCLK for a read cycle. These positions are specified in terms of cycles and half-cycles of CLK, counting forwards from a HCLK rising edge.

`wclkpos` specifies the positions of the clock cycles of RAMCLK for a write cycle. These are also counted forward from an HCLK rising edge.

`clkpulselen` specifies the length of the RAMCLK pulses in CLK cycles. This is specified once per RAM. It applies to both the read and write clocks.

>: 10. Targeting hardware


Fast clock CLK
Handel-C clock HCLK (div 4)
RAM clock RAMCLK (rclkpos={3.0}, clkpulselen = 0.5)
RAM clock RAMCLK (rclkpos={1.5, 2.5}, clkpulselen = 0.5)
RAM clock RAMCLK (rclkpos={1.0, 3.0}, clkpulselen = 1.0)
t0
t0+1
t0+2
t0+3
t1
t1+1
t1+2

TIMING DIAGRAM: SSRAM READ CYCLE USING GENERATED RAMCLK

The pulse positions and lengths are specified in cycles and half-cycles of CLK. The `westart` and `welength` specs are used to place the write enable strobe where it is required.

10.5.3 Flow-through SSRAM example

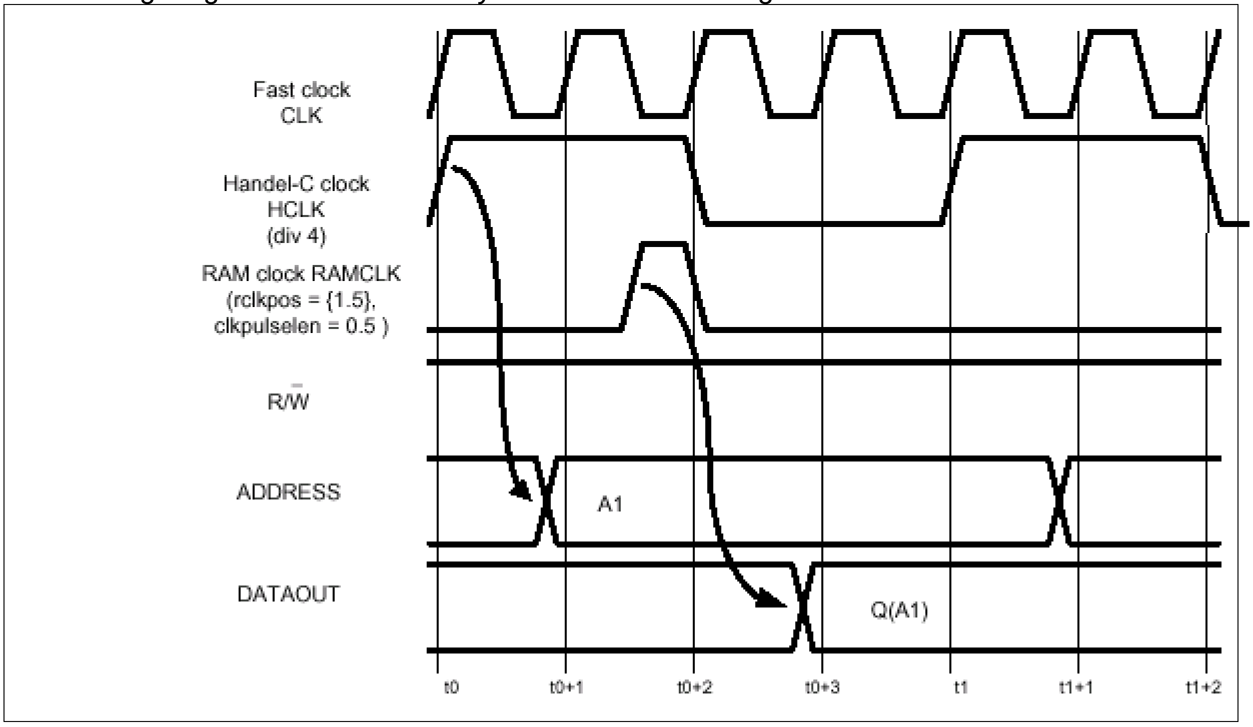
```
ram unsigned 18 FlowBank[1024] with {block = 1,
                westart = 2,
                welength = 1,
                rclkpos = {1.5},
                wclkpos = {2.5, 3.5},
                clkpulselen = 0.5};
```

This code instructs the compiler to build hardware to generate SSRAM control signals as shown below. It is also applicable for reading from block RAMs in Xilinx FPGAs and Altera PLDs.

>: 10. Targeting hardware

Read cycle for a flow-through SSRAM

The timing diagram shows a read-cycle from a flow-through SSRAM.



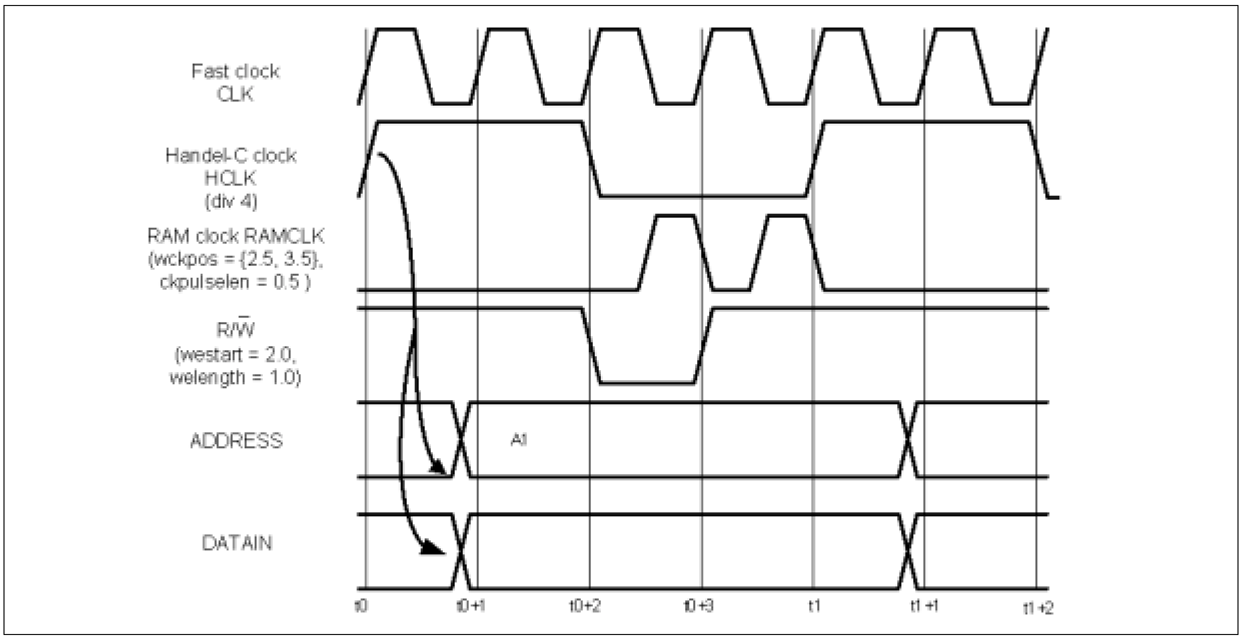
The rising HCLK edge at t0 initiates the read cycle. Some time later, the address A1 is set up, which is sampled somewhere in the middle of the HCLK cycle: t0+1.5 in this case. By the time the next HCLK rising edge occurs at t1, the data is available for reading. The cycle completes within one Handel-C clock cycle.

Write cycle for a flow-through SSRAM

Flow-through SSRAMs perform a "late" write cycle; the data is clocked in one clock cycle after the address is sampled.

>: 10. Targeting hardware

The timing diagram shows the complete write cycle.



The HCLK rising edge at t0 initiates the write cycle, causing the ADDRESS and DATAIN signals to change. Two cycles of RAMCLK are needed to clock the new data into the RAM at the specified address: the first to sample the address, the second to sample the data. However, since we're not expecting to read from the RAM's output, we can wait until the last possible moment. In this case, the two rising edges of RAMCLK occur at t0+2.5 and t0+3.5.

The write enable signal must be low during the rising edge of RAMCLK that samples the address, but not during the one that samples the data. This can be done by setting `westart` and `welength` as shown. The entire cycle completes within a single Handel-C clock cycle.

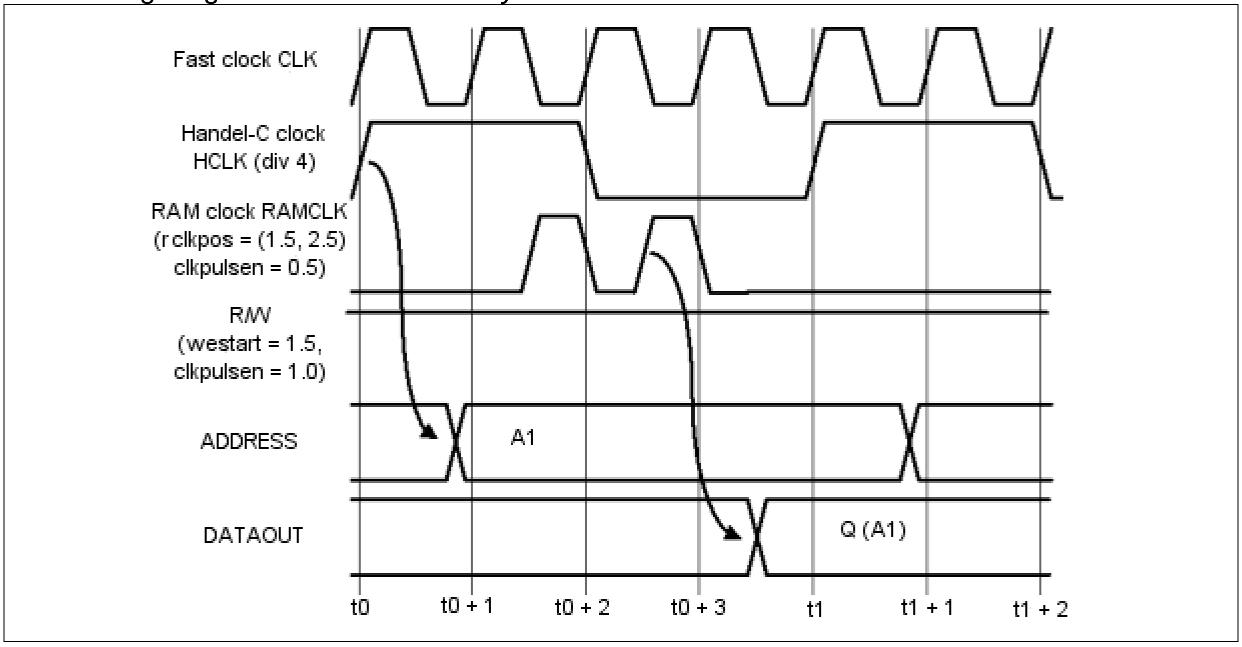
10.5.4 Pipelined-output SSRAM timing example

```
ram unsigned 18 PipeBank[1024] with {block = 1,
                westart = 1.5,
                welength = 1,
                rclkpos = {1.5, 2.5},
                wclkpos = {1.5, 2.5, 3.5},
                clkpulselen = 0.5};
```

>: 10. Targeting hardware

Read cycle for a pipelined-output SSRAM

The timing diagram shows the read cycle


Fast clock CLK
Handel-C clock HCLK (div 4)
RAM clock RAMCLK (rclkpos = (1.5, 2.5) clkpulsen = 0.5)
R/W (westart = 1.5, clkpulsen = 1.0)
ADDRESS
A1
DATAOUT
Q (A1)
t0
t0 + 1
t0 + 2
t0 + 3
t1
t1 + 1
t1 + 2

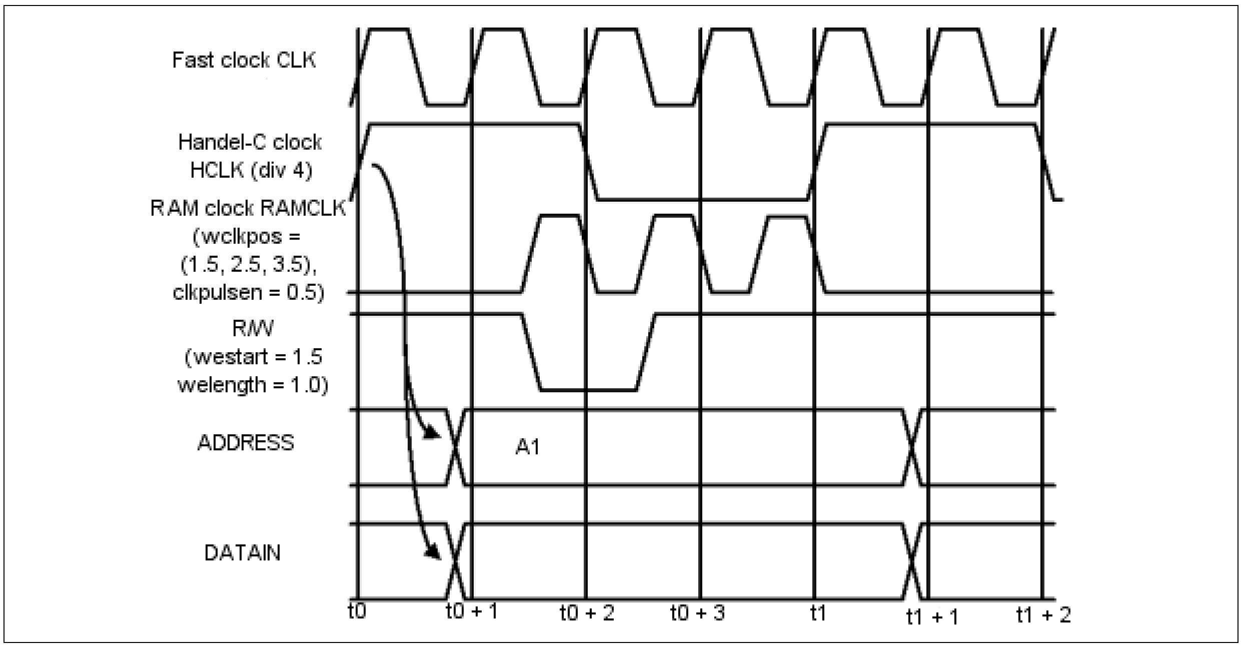
This read cycle is very similar to that for a flow through RAM. The rising HCLK edge at t0 initiates the read cycle. Some time later, the address A1 is set up, which is sampled somewhere near the middle of the HCLK cycle: (t0+1.5 in this case). The RAM contents at address A1 are then piped to the RAM's output register; it must be made available at the RAM output. A second RAMCLK pulse (at t0+2.5 in this case) is used to do this. By the time the next HCLK rising edge occurs at t1, the data is available for reading by the Handel-C design. The cycle completes within one Handel-C clock cycle.

Write cycle for a pipelined-output SSRAM

Pipelined-output SSRAMs perform a "late-late" write cycle. This means that data is written to memory two clock cycles after the address is sampled.

The timing diagram shows the complete cycle:

>: 10. Targeting hardware


Fast clock CLK
Handel-C clock HCLK (div 4)
RAM clock RAMCLK (wclkpos = (1.5, 2.5, 3.5), clkpulsen = 0.5)
R/W (westart = 1.5 welength = 1.0)
ADDRESS
A1
DATAIN
t0
t0 + 1
t0 + 2
t0 + 3
t1
t1 + 1
t1 + 2

The HCLK rising edge at t0 initiates the write cycle, causing the ADDRESS and DATAIN signals to change. Three cycles of RAMCLK are needed to clock the new data into the RAM at the specified address: the first to sample the address and the third to sample the data. Since you will not read from the RAM on a write strobe, you can sample the data as late as possible to give the circuit maximum time to set up the data. In this case, the three rising edges of RAMCLK occur at t0+1.5, t0+2.5 and t0+3.5.

The write enable signal must be low during the rising edge of RAMCLK that samples the address, but not during the one that samples the data. This can be done by setting `westart` and `welength` as shown. The entire cycle completes within a single Handel-C clock cycle.

10.6 Using on-chip RAMs in specified devices

10.6.1 Using on-chip RAMs in Xilinx devices

Xilinx 4000 series devices can implement RAMs and ROMs in the look up tables on the device. Handel-C supports the synchronous RAMs on the 4000E, 4000EX, 4000L, 4000XL, 4000XV and Virtex series parts directly simply by declaring a RAM or ROM. For example:

```
ram unsigned 6 x[34];
```

This will declare a RAM with 34 entries, each of which is 6 bits wide.

>: 10. Targeting hardware

10.6.2 Using on-chip RAMs in Altera devices

On-chip RAMs in Altera Flex10K devices use the EAB structures. These blocks can be configured in a number of data width/address width combinations. When writing Handel-C programs, you must be careful not to exceed the number of EAB blocks in the target device or the design will not place and route successfully. While it is possible to use RAMs that do not match one of the data width/address width combinations, EAB space may be wasted by such a RAM.

RAM blocks in Flex 10K and Apex 20K parts can be configured to be either synchronous or asynchronous.

Synchronous access

If no clock specification is given and none of `westart`, `welength` or `wegate` is specified, Handel-C will use synchronous access as a default, by utilizing the falling edge of the clock as the input clock signal to the RAM. However, to obtain the best performance use the clock specification to tailor the RAM access timing to the needs of your design.

Asynchronous access

If you use one of the `westart`, `welength` or `wegate` specifications described in the previous section, the Handel-C compiler will generate an asynchronous RAM, as long as no clock specification is given.

Initialization

RAM/ROM initialization files with a `.mif` extension will be generated on compilation to feed into the Max Plus II or Quartus software. This process is transparent as long as they are in the same directory as the EDIF (`.edf` extension) file generated by the Handel-C compiler

10.6.3 Using on-chip RAMs in Actel devices

On-chip RAMs in Actel ProASIC and ProASIC+ devices use the embedded memory structures, which are of a fixed width and depth. These blocks can be combined to create deeper and wider memory spaces. When writing Handel-C programs, you must be careful not to exceed the number of memory blocks in the target device or the design will not place and route successfully. While it is possible to use RAMs that do not match one of the width/depth combinations, memory space may be wasted.

Memory blocks in ProASIC and ProASIC+ parts can be configured to be either synchronous or asynchronous.

Synchronous access

If no clock specification is given and none of `westart`, `welength` or `wegate` is specified, Handel-C will use synchronous access as a default, by utilizing the falling edge of the Handel-C clock as the input clock signal to the RAM. However, to obtain the best performance use the clock specifications to tailor the RAM access timing to the needs of your design.

>: 10. Targeting hardware

Asynchronous access

If you use one of the `westart`, `welength` or `wegate` specifications, the Handel-C compiler will generate an asynchronous RAM, as long as no clock specifications is given.

Initialization

Actel memories may not be initialized.

10.7 Targeting external RAMs

10.7.1 Targeting external asynchronous RAMs

Handel-C provides support for accessing off-chip static RAMs in the same way as you access internal RAMs. The syntax for an external RAM declaration is:

```
ram Type Name[Size] with {
                          offchip = 1,
                          data = Pins,
                          addr = Pins,
                          we = Pins,
                          oe = Pins,
                          cs = Pins};
```

To declare a 16Kbyte by 8-bit RAM:

```
ram unsigned 8 ExtRAM[16384] with {
       offchip = 1,
       data = {"P1", "P2", "P3", "P4",
               "P5", "P6", "P7", "P8"},
       addr = {"P9", "P10", "P11", "P12",
               "P13", "P14", "P15", "P16",
               "P17", "P18", "P19", "P20",
               "P21", "P22"},
       we = {"P23"},
       oe = {"P24"},
       cs = {"P25"}};
```

Note that the lists of address and data pins are in the order of most significant to least significant. It is possible for the compiler to infer the width of the RAM (8 bits in this example) and the number of address lines used (14 in this example) from the RAM's usage. This is not recommended since this declaration deals with real external hardware which has a fixed definition.

>: 10. Targeting hardware

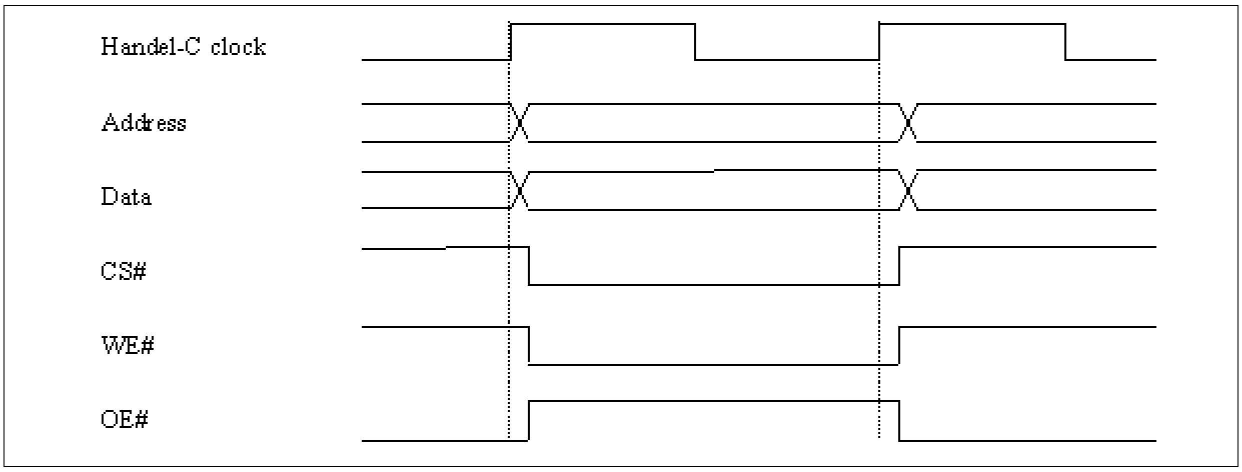
Accessing RAM

Accessing the RAM is the same as for accessing internal RAM. For example:

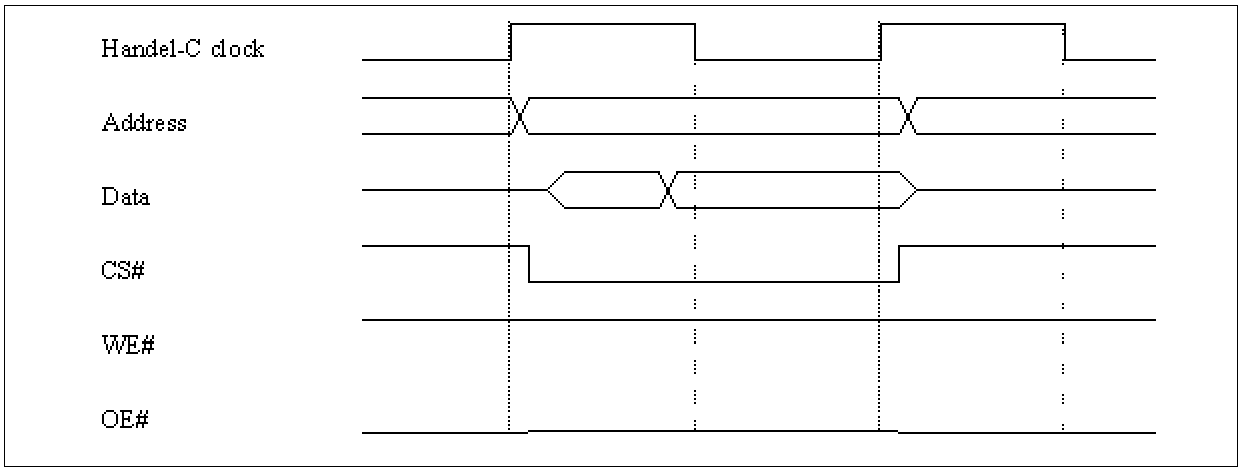
```
ExtRAM[1234] = 23;
y = ExtRAM[5678];
```

Similar restrictions apply as with internal RAM - only one access may be made to the RAM in any one clock cycle.

The compiled hardware generates the following cycle for a write to external RAM:



The compiled hardware generates the following cycle for a read from external RAM:



This cycle may not be suitable for the RAM device in use. In particular, asynchronous static RAM may not work with the above cycle due to setup and hold timing violations. For this reason, the `westart`, `welength` and `wegate` specifications may also be used with external RAM declarations.

>: 10. Targeting hardware

10.7.2 Targeting external synchronous RAMs

Off-chip synchronous SRAMs can be specified in exactly the same way as on-chip synchronous SRAMs, with the addition of the `rclkpos`, `wclkpos`, `clkpulselen` and `clk` specifications. `clk` specifies the pin on which the generated RAMCLK will appear, when the SSRAM in question is external (`offchip = 1`).

Example

```
macro expr addressPins = {Pin List...};
macro expr dataPins = {Pin List...};
macro expr csPins = {Pin List...};
macro expr wePins = {Pin List...};
macro expr oePins = {Pin List...};
macro expr clkPins = {Pin List...};

ram unsigned 32 ExtBank[1024] with {offchip = 1,
                                addr = addressPins,
                                data = dataPins,
                                cs = csPins,
                                we = wePins,
                                oe = oePins,
                                westart = 2,
                                welength = 1,
                                rclkpos = {1.5, 2.5},
                                wclkpos = {1.5, 2.5, 3.5},
                                clkpulselen = 0.5,
                                clk = clkPins};
```

10.7.3 Using external ROMs

An external ROM is declared as an external RAM with an empty write enable pin list. For example:

```
ram unsigned 8 ExtROM[16384] with {
       offchip = 1,
       data = {"P1", "P2", "P3", "P4",
               "P5", "P6", "P7", "P8"},
       addr = {"P9", "P10", "P11", "P12",
               "P13", "P14", "P15", "P16",
               "P17", "P18", "P19", "P20",
               "P21", "P22"},
       we = {},
       oe = {"P24"},
       cs = {"P25"}};
```

Note that no `westart`, `welength` or `wegate` specification is required since there is no write strobe signal on a ROM device.

10.7.4 Connecting to RAMs in foreign code

You can create ports to connect to a RAM by using the `ports = 1` specification to your memory definition. This will generate VHDL or EDIF wires which can be connected to a component created elsewhere. The ports specification cannot be used in conjunction with the `offchip=1` specification, but all other specifications will apply.

The interface generated will have separate read (output) and write (data) ports, write enable, data enable and clock wires. This ensures that it can be connected to any device. Pin names provided in the `addr, data, cs,we, oe,` and `clk` specifications will be passed through to the generated EDIF. They are not passed through to VHDL, since VHDL interfaces are generated as n-bit wide buses rather than n 1-bit wide wires. This means that it is ambiguous to specify a separate identifier for each wire. If they are used when compiling to VHDL, the compiler issues a warning.

For VHDL or Verilog output, the compiler generates meaningful port names. For example, with the following RAM declaration compiled to VHDL:

```
ram unsigned 4 rax[4] with
   {ports = 1, data = dataPins, addr = addrPins,
      we = wePins, cs = csPins, oe = oePins};
```

the compiler will warn that all the pins specifications have been ignored, and will generate an interface in VHDL with the following ports:

```
component rax_SPPort
port(
rax_SPPort_addr: in unsigned(1 downto 0);
rax_SPPort_clk: in std_logic;
rax_SPPort_cs: in std_logic;
rax_SPPort_data_en: in std_logic;
rax_SPPort_data_in: out unsigned(3 downto 0);
rax_SPPort_data_out: in unsigned(3 downto 0);
rax_SPPort_oe: in std_logic;
rax_SPPort_we: in std_logic
);
```

The port names consist of the memory name (`rax` in this case), description of the memory type (`SPPort` : single port in this case) and an identifier describing the ports function.

A clock port will always be generated.

If ylou use the `ports` specification with an MPRAM, a separate interface will be generated for each port.

>: 10. Targeting hardware

Generating an interface to a foreign code RAM: Example

```
set family = Xilinx4000XV;
set part = "V1000BG560-4";
set clock = external "C1";

unsigned 4 a;
ram unsigned 4 rax[4] with {ports = 1};

void main(void)
{
  static unsigned 2 i = 0;

  while(1)
  {
    par
    {
      i++;
      a++;
      rax[i] = a;
    }
    a = rax[i];
  }
}
```

The declaration of `rax` would produce wires

```
rax_SPPort_addr<0>        // Address
rax_SPPort_addr<1>
rax_SPPort_data_in<0>     // Data In
rax_SPPort_data_in<1>
rax_SPPort_data_in<2>
rax_SPPort_data_in<3>
rax_SPPort_data_out<0>    // Data Out
rax_SPPort_data_out<1>
rax_SPPort_data_out<2>
rax_SPPort_data_out<3>
rax_SPPort_data_en        // Data Enable
rax_SPPort_clk            // Clock
rax_SPPort_cs             // Chip Select
rax_SPPort_oe             // Output Enable
rax_SPPort_we             // Write Enable
```

>: 10. Targeting hardware

Generating an interface to a foreign code MPRAM: Example

```
set family = XilinxVirtex;
set part = "V1000BG560-4";
set clock = external "C1";

unsigned 4 a;

mpram Mpaz
{
  wom unsigned 4 wox[4];
  rom unsigned 4 rox[4];
} mox with {ports = 1};

void main(void)
{
  static unsigned 2 i = 0;

  while(1)
  {
    par
    {
      i++;
      a++;
      mox.wox[i] = a;
    }
    a = mox.rox[i];
  }
}
```

The declaration of the read only port `rox` would produce wires

```
mox_rox_addr_0 // Address
mox_rox_addr_1
mox_rox_clk // Clock
mox_rox_cs // Chip select
mox_rox_data_en // Data enable
mox_rox_oe // Output enable
mox_rox_we // Write enable
mox_rox_data_in_0 // Data In (into Handel-C, out from foreign code
memory)
mox_rox_data_in_1
mox_rox_data_in_2
mox_rox_data_in_3
```

The declaration of the read only port wox would produce wires

```
mox_wox_addr_0 // Address
mox_wox_addr_1
mox_wox_clk // Clock
mox_wox_cs // Chip select
mox_wox_data_en // Data enable
mox_wox_data_out_0 // Data Out (from Handel-C, into foreign code
memory)
mox_wox_data_out_1
mox_wox_data_out_2
mox_wox_data_out_3
mox_wox_oe // Output enable
mox_wox_we // Write enable
```

10.8 Using other RAMs

The interface to other types of RAM such as DRAM should be written by hand using interface declarations. Macro procedures can then be written to perform complex or even multi-cycle accesses to the external device.

>: 11 External hardware and logic

11.1 Interfacing with external hardware and logic

All off-chip accesses are based on the idea of a bus which is just a collection of external pins. Handel-C provides the ability to read the state of pins for input from the outside world and set the state of pins for writing to the outside world. Tri-state buses are also supported to allow bi-directional data transfers through the same pins.

The pins used may be defined in Handel-C by using pin specifications (e.g. `data`). If this is omitted, the pins will be left unconstrained and can be assigned by the place and route tools.

Note that Handel-C provides no information about the timing of the change of state of a signal within a Handel-C clock cycle. Timing analysis is available from the FPGA or PLD manufacturer's place-and-route tools.

Handel-C programs can also interface to external logic (other Handel-C programs, programs written in VHDL or Verilog etc.) by using user-defined interfaces or Handel-C ports.

11.2 Interface sorts

Handel-C provides a number of predefined interface sorts.

"bus-type" interfaces `(bus_*)` generate the hardware for buses connected to pins.

"port-type" interfaces `(port_*)` generate the hardware for floating ports (buses which are not connected to pins). These can be of any width, and can carry signals between different sections of Handel-C code, or to software or hardware beyond the Handel-C program.

You can also define your own sorts to interface to external blocks of code ("generic" or custom interface sorts).

>: 11. External hardware and logic

Predefined interface sorts

Sort identifier	Description
`bus_in`	Input bus from pins
`bus_latch_in`	Registered input bus from pins
`bus_clock_in`	Clocked input bus from pins
`bus_out`	Output bus to pins
`bus_ts`	Bi-directional tri-state bus
`bus_ts_latch_in`	Bi-directional tri-state bus with registered input
`bus_ts_clock_in`	Bi-directional tri-state bus with clocked input
`port_in`	Input port from logic
`port_out`	Output port to logic

Custom or generic interface sorts

You can define your own interface sorts to connect to non-Handel-C objects:

- Hardware descriptions written in another language.
  VHDL, Verilog and EDIF are currently supported. For a VHDL code interface, the interface sort would be the name of the VHDL entity. For a Verilog code interface, the interface sort would be the name of the Verilog module.

- Native PC object code used in simulation.
  Programs that run on your PC for simulation and connect to a Handel-C interface are known as plugins. There are special port specifications to enable you to connect user-defined interfaces with a plugin for simulation. These are `extlib`, `extfunc`, and `extinst`.

11.2.1 Reading from external pins: bus_in

The `bus_in` interface sort allows Handel-C programs to read from external pins. Its general usage is:

```
interface bus_in(type portName)
   Name()
      with {data = {Pin List}};
```

Reading the bus is performed by accessing the identifier *Name*`.`*portName* as a variable which will return the value on those pins at that clock edge. If no input port name is given, the port name defaults to `in`.

Example

```
interface bus_in(int 4 To) InBus() with {data =
                    {"P4", "P3", "P2", "P1"}};
int 4 x;

x = InBus.To;
```

This declares a bus connected to pins P1, P2, P3 and P4 where pin P4 is the most significant bit and pin P3 is the least significant bit.

The variable x is set to the value on the external pins. The type of `InBus.To` is `int 4` as specified in the type list after the `bus_in` keyword.

11.2.2 Registered reading from external pins: bus_latch_in

The `bus_latch_in` interface sort is similar to `bus_in` but allows the input to be registered on a condition. This may be required to sample the signal at particular times. Its general usage is:

```
interface bus_latch_in(type portName)
      Name(type conditionPortName=Condition)
          with {data = {Pin List}};
```

Reading the bus is performed by accessing the identifier *Name*`.`*portName* as a variable which will return the value on those pins at that clock edge. If no input port name is given, the port name defaults to `in`. *Condition* specifies a signal that is used to clock the input registers in the FPGA or PLD. The rising edge of this signal clocks the external signal into the internal value.

Example

```
int 1 get;
int 4 x;

interface bus_latch_in(int 4 To)
        InBus(int 1 condition = get)
        with {data = {"P4", "P3", "P2", "P1"}};

get = 0;
get = 1;      // Register the external value
x = InBus.To; // Read the registered value
```

>: 11. External hardware and logic

11.2.3 Clocked reading from external pins: bus_clock_in

The `bus_clock_in` interface sort is similar to the `bus_in` interface sort but allows the input to be clocked continuously from the Handel-C global clock. This may be required to synchronize the signal to the Handel-C clock. Its general usage is:

```
interface bus_clock_in(type portName) Name()
   with {Specs};
```

Reading the bus is performed by accessing the identifier *Name*`.`*portName* as a variable which will return the value on those pins at that clock edge. If no input port name is given, the port name defaults to `in`. The rising edge of the Handel-C clock clocks the external signal into the internal value. For example:

```
interface bus_clock_in(int 4 InTo) InBus() with
             {data = {"P4", "P3", "P2", "P1"}};

x = InBus.InTo; // Read flip-flop value
```

11.2.4 Writing to external pins: bus_out

The `bus_out` interface sort allows Handel-C programs to write to external pins. Its general usage is:

```
interface bus_out()
   Name(type portName=Expression)
      with {data = {Pin List}};
```

A specific example is:

```
interface bus_out () OutBus(int 4 OutPort=x+y)
                 with {data = {"P4", "P3", "P2", "P1"}};
```

This declares a bus connected to pins 1, 2, 3 and 4 where pin 4 is the most significant bit and pin 1 is the least significant bit. The value appearing on the external pins is the value of the expression `x+y` at all times.

11.2.5 Bi-directional data transfer: bus_ts

The `bus_ts` interface sort allows Handel-C programs to perform bi-directional off-chip communications via external pins. Its general usage is:

```
interface bus_ts (type inPortName)
   Name(type outPortName = Value, type conditionPortName = Condition)
   with {Specs};
```

>: 11. External hardware and logic

Value is an expression giving the value to output on the pins. *Condition* is an expression giving the condition for driving the pins. When *Condition* is non-zero (i.e. true), the value of *Value* is driven on the pins. When the value of *Condition* is zero, the pins are tri-stated and the value of the external bus can be read using the identifier *Name*`.`*inPortName*. If *inPortName* is not defined, the port name defaults to `in`.

Example

```
unsigned 1 condition;
int 4 x;

interface bus_ts(int 4 read)
           BiBus(int write=x+1, unsigned 1 enable= condition)
             with {data = {"P4", "P3", "P2", "P1"}};

condition = 0;      // Tri-state the pins
x = BiBus.read;     // Read the value
condition = 1;      // Drive x+1 onto the pins
```

This example reads the value of the external bus into variable `x` and then drives the value of `x + 1` onto the external pins.

Take care when driving tri-state buses that the FPGA/PLD and another device on the bus cannot drive simultaneously as this may result in damage to one or both of them.

11.2.6 Bi-directional data transfer with registered input

The `bus_ts_latch_in` interface sort allows Handel-C programs to perform bi-directional off-chip communications via external pins with inputs registered on a condition. Its general usage is:

```
interface bus_ts_latch_in (type inPortName)
   Name(type outPortName = Value,
      type conditionPortName = Condition,
      type clockPortName = Clock)
   with {Specs};
```

Value is an expression giving the value to output on the pins. *Condition* is an expression giving the condition for driving the pins. *Clock* is an expression giving the signal to clock the input from the pins. When *Condition* is non-zero (i.e. true), the value of *Value* is driven on the pins. When the value of *Condition* is zero, the pins are tri-stated and the registered value of the external bus can be read using the identifier *Name*`.`*inPortName*. If *inPortName* is not defined, the port name defaults to `in`.

>: 11. External hardware and logic

The rising edge of the value of the third expression clocks the external values through to the internal values on the chip.

Example

```
int 1 get;
unsigned 1 condition;
int 4 x;

interface bus_ts_latch_in(int 4 read)
    BiBus(int write = x+1,
       unsigned 1 enable = condition,
          unsigned 1 clock_port = get)
       with {data = {"P4", "P3", "P2", "P1"}};

condition = 0;  // Tri-state external pins
get = 0;
get = 1;        // Register external value
x = BiBus.read; // Read registered value
condition = 1;  // Drive x+1 onto external pins
```

This example samples the external bus and reads the registered value into variable `x` and then drives the value of `x + 1` onto the external pins.

Take care when driving tri-state buses that the FPGA/PLD and another device on the bus cannot drive simultaneously as this may result in damage to one or both of them.

11.2.7 Bi-directional data transfer with clocked input

The `bus_ts_clock_in` interface sort allows Handel-C programs to perform bi-directional off-chip communications via external pins with inputs clocked continuously with the Handel-C clock. Its general usage is:

```
interface bus_ts_clock_in (type inPortName)
      Name(type outPortName = Value,
          type conditionPortName = Condition)
                 with {Specs};
```

Value is an expression giving the value to output on the pins. *Condition* is an expression giving the condition for driving the pins. When *Condition* is non-zero (i.e. true), the value of *Value* is driven on the pins. When the value of *Condition* is zero, the pins are tri-stated and the value of the external bus can be read using the identifier *Name*`.`*inPortName*. If *inPortName* is not defined, the port name defaults to `in`

>: 11. External hardware and logic

The rising edge of the Handel-C clock reads the external values into the internal flip-flops on the chip. For example:

```
unsigned 1 condition;
int 4 x;

interface bus_ts_clock_in (int 4 read)
     BiBus(int 4 writePort=x+1,
        unsigned 1 enable=condition)
       with {data = {"P4", "P3", "P2", "P1"}};

condition = 0;  // Tri-state external pins
x = BiBus.read; // Read registered value
condition = 1;  // Drive x+1 onto external pins
```

This example reads the value from the flip-flop into variable x and then drives the value of x + 1 onto the external pins.

Take care when driving tri-state buses that the FPGA/PLD and another device on the bus cannot drive simultaneously as this may result in damage to one or both of them.

11.2.8 Example hardware interface

The example shows the use of buses. The scenario is of an external device connected to the FPGA which may be read from or written to. The device has a number of signals connected to the FPGA.

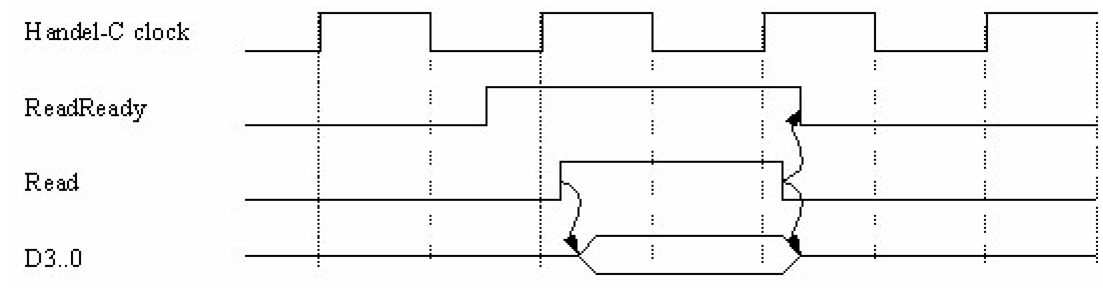
Signals connected

Signal Name	FPGA pin	Description
`D3..0`	1, 2, 3, 4	Data Bus
`Write`	5	Write strobe
`Read`	6	Read strobe
`WriteRdy`	7	Able to write to device
`ReadRdy`	8	Able to read from device

Read cycle timing

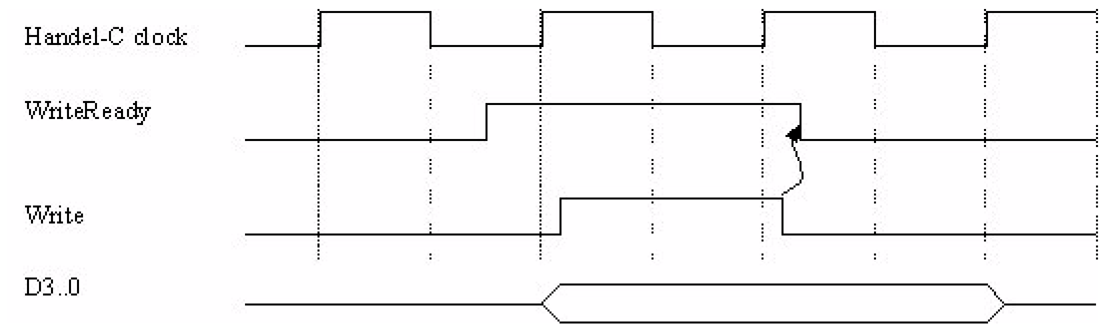
A read from the device is performed by waiting for `ReadRdy` to become active (high). The Read signal is then taken high for one clock cycle and the data sampled on the falling edge of the strobe.

>: 11. External hardware and logic



Write cycle timing

A write to the device is performed by waiting for `WriteRdy` to become active (high). The `Write` signal is then taken high for one clock cycle while the data is driven to the device by the FPGA. The device samples the data on the falling edge of the Write signal.



Bus declarations

The first stage of the code declares the buses associated with each of the external signals.

```
int 4 Data;
int 1 En = 0;
interface bus_ts_clock_in(int 4 DataIn)
          dataB(int outPort=Data, int EnableSignal=En) with
            {data = {"P4", "P3", "P2", "P1"}};

int 1 Write = 0;
interface bus_out() writeB(int WriteSignal = Write) with
              {data = {"P5"}};

int 1 Read = 0;
interface bus_out() readB(int readSignal=Read) with
              {data = {"P6"}};

interface bus_clock_in(int 1 wr)
               WriteReady() with {data = {"P7"}};
```

>: 11. External hardware and logic

```
interface bus_clock_in(int 1 readySignal)
                ReadReady() with {data = {"P8"}};


void main (void)
{
    int 4 Data, Reg;

    // Read word from external device
    while (ReadReady.readySignal == 0)
        delay;

    Read = 1; // Set the read strobe
    par
    {
        Data = dataB.DataIn; // Read the bus
        Read = 0; // Clear the read strobe
    }

    // Write one word back to external device
    Reg = Data + 1;
    while (WriteReady.wr == 0)
        delay;

    par
    {
        En = 1; // Drive the bus
        Write = 1; // Set the write strobe
    }

    Write = 0; // Clear the write strobe
    En = 0; // Stop driving the bus
}
```

Writing data

You can change the values on the output buses by setting the values of the `Data`, `Write` and `Read` variables. You can drive the data bus with the contents of `Data` by setting `En` to 1.

The variables that drive buses have been initialized to 0. That means that these variables must be static or global. This may be important when driving write strobes. Care should be taken during configuration that the FPGA pins are disconnected in some way from the external devices because the FPGA pins become tri-state during this time.

>: 11. External hardware and logic

The main program

The main program reads a word from the external device before writing one word back.

```
void main (void)
{
    int 4 Data, Reg;

    // Read word from external device
    while (ReadReady.readySignal == 0)
        delay;
    Read = 1;      // Set the read strobe
    par
    {
        Data = dataB.DataIn; // Read the bus
        Read = 0;  // Clear the read strobe
    }

    // Write one word back to external device
    Reg = Data + 1;
    while (WriteReady.wr == 0)
        delay;
    par
    {
        En = 1;    // Drive the bus
        Write = 1; // Set the write strobe
    }
    Write = 0;     // Clear the write strobe
    En = 0;        // Stop driving the bus
}
```

Note that during the write phase, the data bus is driven for one clock cycle after the write strobe goes low to ensure that the data is stable across the falling edge of the strobe.

11.3 Merging pins

It is possible to merge pins.

- merge input pins with double declarations of input bus interfaces
- merge tri-state pins

>: 11. External hardware and logic

Input pins

Input pins can be merged so that pins can be read simultaneously into multiple variables. This can be done by specifying multiple interfaces (`bus_in`, `bus_clock_in`, `bus_latch_in`) which have some pins in common. If required, a different subset of pins can be specified for each instance of the interface. For example:

```
interface bus_in(int 8 wide) wideDataBus() with
  {data ={"P7", "P6", "P5", "P4", "P3",
     "P2", "P1", "P0"}};
interface bus_in(int 3 thin) thinDataBus() with
  {data ={"P5", "P4", "P3"}};
```

`wideDataBus.wide` would give the values of pins P0 – P7, whereas `thinDataBus.thin` would give the three bit value on pins P3, P4 and P5.

Tri-state bus pins

Tri-state bus pins can be merged, though doing so will generate a compiler warning, as the compiler cannot detect whether there is a conflict in the use of the merged pins. You might wish to merge output pins for a tri-state bus if you wished to switch the circuit connections from one external piece of logic to another. For example:

```
int 1 en1, en2;
int 4 x, y;
interface bus_ts_clock_in (int 4 read)
      BiBus1(int 4 writePort=x+1, unsigned 1 enable = (en1==1))
      with {data = {"P4", "P3", "P2", "P1"}};
interface bus_ts_clock_in (int 4 read)
      BiBus2(int 4 writePort=y+1, unsigned 1 enable = (en2==1))
      with {data = {"P4", "P3", "P2", "P1"}};
```

Take care when driving tri-state buses that the FPGA/PLD and another device on the bus cannot drive simultaneously as this may result in damage to one or both of them.

11.4 Buses and the simulator

The Handel-C simulator cannot simulate buses directly, because the simulation of buses cannot determine when input and output should occur. The recommended process for debugging is:

For simple data, use a channel or a `chanin/chanout` to connect to a file. This is the simplest method.

>: 11. External hardware and logic

For more complex buses/interfaces, write a C/C++ function and call it to bring in data. This allows you to operate on the data or read it in a complex format. This models functionality but not hardware.

To model buses accurately, use the Plugin Library to write a plugin which can be co-simulated. This is precise and allows you to read I/O signals using the waveform analyzer, but can be slow and cumbersome.

Using preprocessor definitions

By using the `#define` and `#ifdef`...`#endif` constructs of the preprocessor, it is possible to combine both the simulation and hardware versions of your program into one.

Channel example

```
#define SIMULATE
#ifdef SIMULATE
    input ? value;
#else
    value = BusIn.in;
#endif
```

External function call example

```
#define SIMULATE

#ifdef SIMULATE
    extern "C++" int 8 bus_input_function(void);
    data_in = bus_input_function();
#else
    interface bus_in(int 8 in) BusIn();
    data_in = BusIn.in;
#endif
```

Example with plugin

To simulate a tri-state bus:

```
interface bus_ts (int 32 in with
   {extlib = "MyPlugin.dll", extinst = "1", extfunc = "DataBusIn"})
   DataBus(int 32 out = DataOut with {extlib = "MyPlugin.dll",
      extinst = "1", extfunc = "DataBusOut"},
      int 1 enable = !WriteBus.in with {extlib = "MyPlugin.dll",
      extinst = "1", extfunc = "DataBusEnable"})
      with {data = pinList};
```

>: 11. External hardware and logic

In this case, the functions `DataBusIn`, `DataBusOut` and `DataBusEnable` would be provided in the plugin `MyPlugin.dll` and called by the simulator. The `extlib`, `extfunc` and `extinst` specifications are ignored if compiled to HDL so the interface definition does not have to be within an `#ifdef`.

11.5 Timing considerations for buses

`bus_in` interfaces

This form of bus is built with no register between the external pin and the points inside the FPGA or PLD where the data is used. If the value on the external pin changes asynchronously with the Handel-C clock then routing delays within the FPGA can cause the value to be read differently in different parts of the circuit. The solution to this problem is to use either a `bus_latch_in` or a `bus_clock_in` interface sort.

`bus_out` interfaces

The output value on pins cannot be guaranteed except at rising Handel-C clock edges. In between clock edges, the value may be in the process of changing. Since the routing delays through different parts of the logic of the output expression are different, some pins may change before others giving rise to intermediate values appearing on the pins. This is particularly apparent in deep combinational logic. Adding a flip-flop to the output (as shown in the `bus_out` example) will minimize these effects.

Race conditions within the combinational logic can lead to glitches on output pins between clock edges. When this happens, pins may glitch from 0 to 1 and back to zero or vice versa as signals propagate through the combinational logic. Adding a flip-flop at the output removes these effects.

Bi-directional tri-state buses

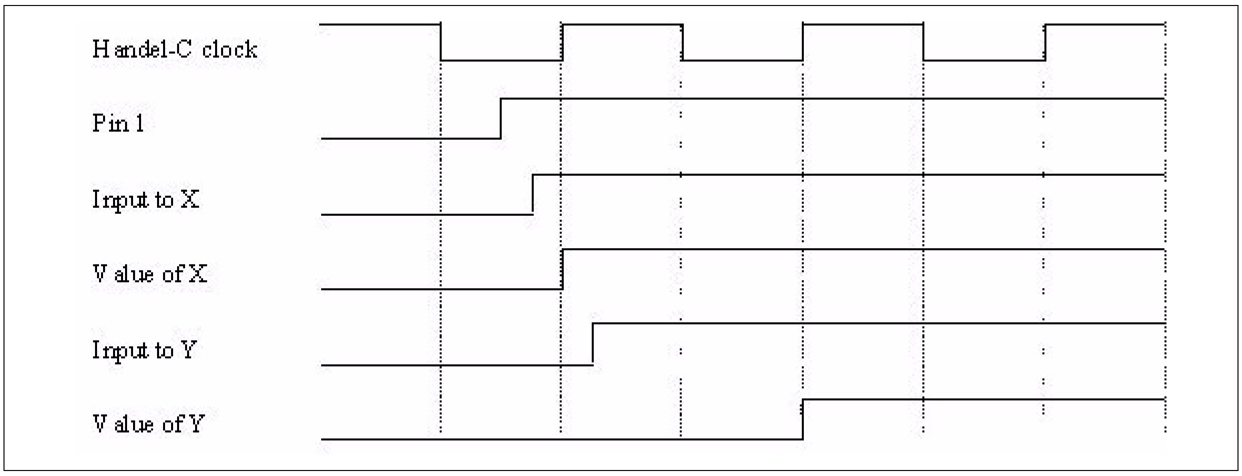
The timing considerations for `bus_in` and `bus_out` interfaces should also be taken into account when using bi-directional tri-state buses since these are effectively a combination of an input bus and an output bus.

11.5.1 Example timing considerations for input buses

```
interface bus_in(int 1 read) a() with
   {data = {"P1"}};

par
{
    x = a.read;
    y = a.read;
}
```

>: 11. External hardware and logic

Even though `a.read` is assigned to both x and y on the same clock cycle, if the delay from pin 1 to the flip-flop implementing the x variable is significantly different from that between pin 1 and the flip-flop implementing the y variable then x and y may end up with different values.



The delay between pin 1 and the input of y is slightly longer than the delay between pin 1 and the input to x. As a result, when the rising edge of the clock registers the values of x and y, there is one clock cycle when x and y have different values.

This effect can also occur in places that are more obscure.

```
interface bus_in(int 1 read) a() with
                              {data = {"P1"}};

while (a.read==1)
{
    x = x + 1;
}
```

Although `a.read` is only apparently used once, the implementation of a `while` loop requires the signal to be routed to two different locations giving the same problem as before. The solution to this problem is to use either a `bus_latch_in` or a `bus_clock_in` interface sort.

The compiler will detect any occurrences of a pin feeding more than one register, and issue a warning.

11.5.2 Example of timing considerations for output buses

```
int 8 x;
int 8 y;

interface bus_out() output(int out = x * y)
      with {data = {"P7", "P6", "P5", "P4",
                    "P3", "P2", "P1", "P0"}};
```

A multiplier contains deep logic so some of the 8 pins may change before others leading to intermediate values. It is possible to minimize this effect (although not eliminate it completely) by adding a variable before the output. This effectively adds a flip-flop to the output. The above example then becomes:

```
int 8 x;
int 8 y;
int 8 z;

interface bus_out() output(int out = z)
      with {data = {"P7", "P6", "P5", "P4",
                    "P3", "P2", "P1", "P0"}};

z = x * y;
```

You must now take care to update the value of z whenever the value output on the bus must change.

11.6 Metastability

The output of a digital logic gate is a voltage level that normally represents either ‘0’ or ‘1’. If the voltage is below the low threshold, it represents 0 and if it is above the high threshold, it represents 1. However, if the voltage input to a register or latch is between these thresholds on the clock edge, then the output of that register will be indeterminate for a time before reverting to one of the normal states. The state to which it reverts and the time at which it reverts cannot be predicted. This is called metastability, and can occur when data is clocked into a register during the time when the data is changing between the two normal voltage levels representing 0 and 1. It is therefore an important consideration for Handel-C programs that may clock in data when the data is changing state.

Techniques to minimize the problem

- examine the metastability characteristics of the device used
- use extra registers to stabilize the data
- decouple the FPGA/PLD from external synchronous hardware by using external buffer storage

>: 11. External hardware and logic

Metastability characteristics of devices

The metastability characteristics of digital logic devices vary enormously. For a discussion of Xilinx FPGAs see the Xilinx FPGA data sheet (reference 2). This document puts the problem into perspective. For example a XC4000E device clocking a 1MHz data signal with a 10MHz clock is expected only once in a million years to take longer than 3ns to recover from a metastable state to a stable state. So when designing a system examine the metastability characteristics of the devices under the conditions in which they will be used to determine whether any precautions need be taken.

Stabilize the data

The ideal system is designed such that when data is clocked into a register it is guaranteed to be stable. This can be achieved by using intermediate buffer storage between the two systems that are transferring data between each other. This storage could be a single dual-port register, dual-port memory, FIFO, or shared memory. Handshaking flags are used to indicate that data is ready, and that data has been read.

However even in this situation sampling of the flags could cause metastability. The solution is to clock the flag into the Handel-C program more than once, so it is clocked into one register, and the output of that register is then clocked into another register. On the first clock the flag could be changing state so the output could be metastable for a short time after the clock. However, as long as the clock period is long relative to the possible metastable period, the second clock will clock stable data. Even more clocks further reduce the possibility of metastable states entering the program, however the move from one clock to two clocks is the most significant and should be adequate for most systems.

The example below has 4 clocks. The first is in the `bus_clock_in` procedure, and the next 3 are in the assignments to the variables `x`, `y`, and `z`.

```
int 4 x,y,z;

interface bus_clock_in(int 4 read) InBus() with
            {data = {"P4", "P3", "P2", "P1"}};

par
{
    while(1)
        x = InBus.read;

    while(1)
        y = x;

    {
        ......
        z = y;
    }
}
```

Design the system to minimize the problem

Remember to keep the problem in perspective by examining the details of the system to estimate the probability of metastability. Design the system in the first place to minimize the problem by decoupling the FPGA from external synchronous hardware by using external buffer storage.

11.6.1 Metastability across clock domains

There are particular metastability issues when dealing with communications across clock domains.

Channels between clock domains

Channels that connect between clock domains are uni-directional point-to-point. The timing between domains is unspecified, but the transmission is guaranteed to occur, and both sides will wait until the transmission has completed. For example:

```
//File: transmit.hcc
chan 8 c; // channel must have global scope

set clock = external "P1";
void main(void)
{
   int 8 x, y;
   c ! x; //program will wait until data successfully transmitted
   c ! y;
}

//File: receive.hcc
extern chan c;

set clock = external "P2";
void main(void)
{
   int 8 p, q;

   c ? p;
   c ? q;
}
```

Interfaces between hardware components in separate clock domains

If you are dealing with hardware components in separate clock domains, you will need to insert resynchronizing hardware if it is not included in the components. For example, if data is sent from `port_out A` in domain bbA and received from `port_in B` in domain bbB, the data must be resynchronized to the clock in domain bbB. This can be done by registering the data at least once in the Handel-C wrapper file.

11.6.2 Metastability in separate clock domains: example

External resynchronizing example

This example shows the three files required to connect two EDIF blocks (`bbA` and `bbB`) which use different clocks. The small files `bbA.hcc` and `bbB.hcc` compile to the EDIF code using the `port_out from` and `port_in to` interfaces. The `metastable.hcc` file connects the two together and generates one flip –flop that resynchronizes the data by reading the value from bbA into a variable.

File: metastable.hcc

```
/*
* Black box code to resynchronize
* Needs to be clocked from the reading clock
* (i.e. bbB.hcc's clock)
*/

int 1 x;
interface bbA(int 1 from) A();
interface bbB() B(int 1 to=x);

set clock = external "P1";
void main(void)
{
  while(1)
  {
  /* stabilize the data by adding
  * resynchronization FF
  */
     x = A.from;
  }
}
```

>: 11. External hardware and logic

File: bbA.hcc

```
/*
* Domain bbA
* Compiles to bbA.edf
*/

set clock = external "P2";
void main(void)
{
  int 1 y;
  interface port_out() from (int 1 from = y);
}
```

File: bbB.hcc

```
/*
*Domain bbB
* Compiles to bbB.edf
*/

set clock = external "P3";
void main(void)
{
  int 1 q;

  interface port_in(int 1 to) to();
  par
  {
    while(1)
    {
      q = to.to; // Read data
    }
  }
}
```

Internal resynchronizing example

The resynchronizing flip-flop can be placed in the file that reads the data from the foreign code block.

This example shows the three files required to connect two EDIF blocks (bbA and bbB) which use different clocks. The small files `bbA.hcc` and `bbB.hcc` compile to the EDIF code using the `port_out from` and `port_in to` interfaces. The `toplevel.hcc` file connects them together. The data is resynchronized in the `bbB.hcc` file.

File: toplevel.hcc

```
/*
* Code to connect data between two cores
*/

interface bbA(int 1 from) A();
interface bbB() B(int 1 to=A.from);
```

File: bbA.hcc

```
/*
* Domain bbA
* Compiles to bbA.edf
*/
set clock = external "P1";
void main(void)
{
  int 1 y;
  interface port_out() from (int 1 from = y);
}
```

File: bbB.hcc

```
/*
*Domain bbB
* Complies to bbB.edf
*/
set clock = external "P2";
void main(void)
{
   int 1 q, y;

   interface port_in(int 1 to) to();
   while(1)
   {
      par
      {
         q = to.to;  // Resynchronize data
         y = q;
      }
   }
}
```

11.7 Ports: interfacing with external logic

Handel-C provides the interface sorts `port_in` and `port_out`. These allow you to have a set of wires, unconnected to pins, which you can use to connect to a simulated device or to another function within the FPGA or PLD. Handel-C supplies the interface declaration for these sorts, and you supply the instance definition.

port_in

For a `port_in`, you define the port(s) carrying data to the Handel-C code and any associated specifications.

```
interface port_in(Type data_TO_hc [with {port_specs}])
         Name() [with {Instance_specs}];
```

For example:

```
interface port_in(int 4 signals_to_HC) read();
```

You can then read the input data from the variable *Name*`.`*data_TO_hc*, in this case `read.signals_to_HC`

port_out

For a `port_out`, you define the port(s) carrying data from the Handel-C code, the expression to be output over those ports, and any associated specifications.

```
interface port_out() Name(Type data_FROM_hc =
   output_Expr[with {port_specs}])
         [with {Instance_specs}];
```

For example:

```
int X_out;
interface port_out()
   drive(int 4 signals_from_HC = X_out);
```

In this case, the width of `X_out` would be inferred to be 4, as that is the width of the port that the data is sent to.

Port names

The name of each port in a `port_in` or `port_out` interface must be different, as they will all be built to the top level of the design.

The examples below would both generate a compiler error.

Example 1:

```
interface port_in(unsigned 1 soggy) In1();
interface port_in(unsigned 1 soggy) In2();
```

Example 2:

```
interface port_in(unsigned 1 soggy) In1();
void main(void)
{
interface port_in(unsigned 1 soggy) In2();
…
}
```

Both examples build two ports to the top level of the design called `soggy`. When they were integrated with external code, the PAR tools wouldn't know which `soggy` to use where.

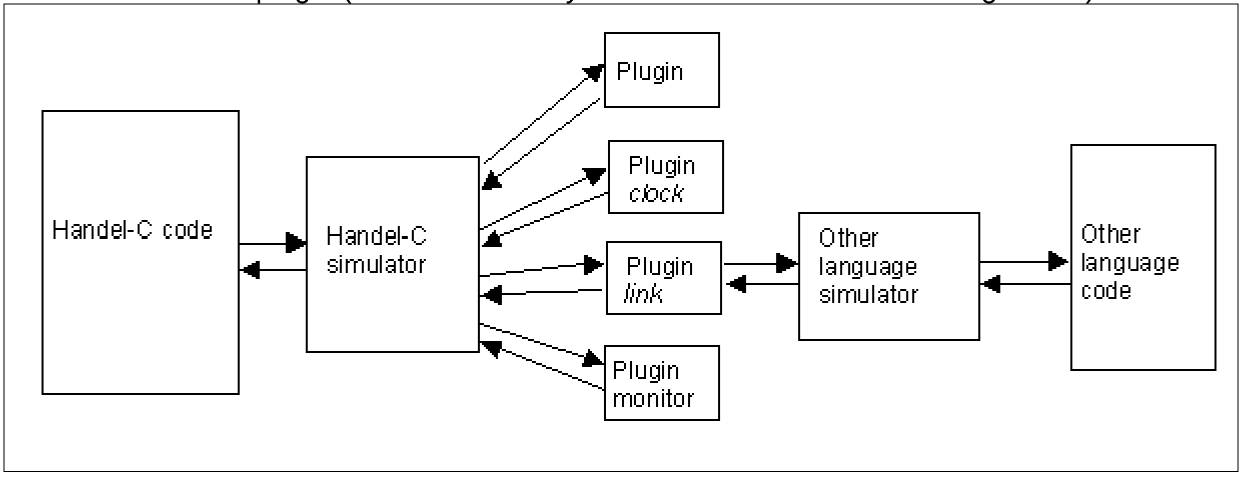
11.8 Specifying the interface

You can specify your own interface format. This allows you to communicate with code written in another language such as VHDL, Verilog or EDIF and allows the Handel-C simulator to communicate with an external plugin program (e.g., a connection to a VHDL simulator).

The expected use for this is to allow you to incorporate bought-in or handcrafted pieces of low-level code in your high-level Handel-C program. It also allows your Handel-C program code to be incorporated within a large EDIF, VHDL or Verilog program. You can also use it to communicate with programs running on a PC that simulate external devices.

>: 11. External hardware and logic

To use such a piece of code requires that you include an `interface` definition in the Handel-C code to connect it to the external code block. This interface definition also tells the simulator to call a plugin (which in turn may invoke a simulator for the foreign code).



11.9 Targeting ports to specific tools

When compiling to EDIF, Handel-C has the capacity to format the names of wires to external logic according to the different syntaxes used by any external components generated by foreign tools. You can do this using the `busformat` specification to a port. This allows you to specify how the bus name and wire number are formatted.

To specify a format, you use the syntax

`with {busformat = "`*formatString*`"}`

formatstring can be one of the following strings. `B` represents the bus name, and `I` represents the wire number.

```
BI
B_I
B[I]
B(I)
B<I>
```

`B`	specifies a bus

`B[N:0]`, `B<N:0>` or `B(N:0)` specify a bus of width (`N+1`).

Example format B[I]

```
interface port_in(int 4 signals_to_HC with
                      {busformat="B[I]"}) read();
```

>: 11. External hardware and logic

would produce wires

```
signals_to_HC[0]
signals_to_HC[1]
signals_to_HC[2]
signals_to_HC[3]
```

Example format B<I>

```
ram unsigned 4 rax[4] with {ports = 1, busformat="B<I>"};
```

would produce wires

```
rax_SPPort_addr<0>       // Address
rax_SPPort_addr<1>
rax_SPPort_data_in<0>    // Data In
rax_SPPort_data_in<1>
rax_SPPort_data_in<2>
rax_SPPort_data_in<3>
rax_SPPort_data_out<0>   // Data Out
rax_SPPort_data_out<1>
rax_SPPort_data_out<2>
rax_SPPort_data_out<3>
rax_SPPort_data_en       // Data Enable
rax_SPPort_clk           // Clock
rax_SPPort_cs            // Chip Select
rax_SPPort_oe            // Output Enable
rax_SPPort_we            // Data In
```

>: 12. Object specifications

>: 12 Object specifications

Handel-C provides the ability to add 'tags' to certain objects (variables, channels, ports, buses, RAMs, ROMs, mprams and signals) to control their behaviour. These tags or specifications are listed after the declaration of the object using the `with` keyword.

When declaring multiple objects, the specification must be given at the end of the line and applies to all objects declared on that line. For example:

```
unsigned x, y with {show=0};
```

This attaches the `show` specification with a value of 0 to both `x` and `y` variables.

Compiler attributes

Specification	Possible Values	Default	Applies to	Meaning
`warn`	0, 1	1	variables memories channels interfaces	Enable warnings for object
`extpath`	Name of port TO Handel-C on the same `interface`	None	port FROM Handel-C	Specify any direct logic (combinational logic) connections to another port

Simulator attributes

Specification	Possible Values	Default	Applies to	Meaning
`show`	0, 1	1	variables channels o/p interfaces tri-state interfaces	Show variable during simulation
`base`	2, 8, 10, 16	10	variables chanouts o/p interfaces tri-state interfaces	Print variable in specified base
`infile`	Any valid filename	None	chanins i/p interfaces tri-state interfaces	Redirect from file

>: 12. Object specifications

Specification	Possible Values	Default	Applies to	Meaning
`outfile`	Any valid filename	None	chanouts o/p interfaces tri-state interfaces, variables	Redirect to file
`extlib`	Name of a plugin `.dll`	None	interface or port	Specify external plugin for simulator
`extfunc`	Name of a function within the plugin	`PlugInSet` or `PlugInGet` depending on port direction	interface or port	Specify external function within the simulator for this port
`extinst`	Instance name (with optional parameters)	None	interface or port	Specify simulation instance used

Interface attributes

Specification	Possible Values	Default	Applies to	Meaning
`bind`	0,1	0	interface, port	Bind component to work library
`properties`	string-value pair OR string-value-string triplet	None	generic interfaces	Parameterize instantiations of external black boxes
`std_logic_vector`	0, 1	0	`port_in`, `port_out` or generic interfaces	Creates a `std_logic_vector` port instead of an `unsigned` port in VHDL output

>: 12. Object specifications

Interface and memory attributes

Specification	Possible Values	Default	Applies to	Meaning
`speed`	0, 1, 2, 3 (Xilinx 4000 only) 0, 1, 2 (Actel ProASIC only) 0, 1 (Xilinx and Altera)	3 for Xilinx 4000 series 2 for Actel ProASIC and ProASIC+ 1 for Xilinx VirtexE and Spartan2 series and Altera	o/p or tri-state interfaces	Set buffer speed
`intime`	Any floating point ns delay	None	input port or interfaces or tri-state interfaces external RAMs	Maximum allowable delay between interface and variable
`outtime`	Any floating point ns delay	None	output port or interfaces or tri-state interfaces external RAMs	Maximum allowable delay between variable and interface
`standard`	Specified keywords representing I/O standards	LVCMOS33:ProASIC / ProASIC+ LVTLL: other	any external interface (dependent on FPGA type) and off-chip memories	I/O standard used (electrical characteristics)
`strength`	2, 4, 6, 8, 12, 16, 24 OR 0 (Min), -1 (Max)	Various, refer to table of supported values	external interfaces and off-chip memories	Signal strength.
`dci`	0, 1	0 (No DCI)	external interfaces (Virtex II only) and off-chip memories	Digital control impedance enabled (only valid with some standards)

>: 12. Object specifications

Specification	Possible Values	Default	Applies to	Meaning
`busformat`	Format string	B_I	interface, port or memories in external logic	Specify the way that wire names are formatted in EDIF
`pull`	0, 1	None	Xilinx and ApexII interfaces	Add pull up or pull down resistor(s)
`data`	Any valid pin list	None	memories interfaces	Set data pins

Memory attributes

Specification	Possible Values	Default	Applies to	Meaning
`offchip`	0, 1	0	memories	Set RAM/ROM to be off chip. Cannot be used in conjunction with `ports`
`ports`	0, 1	0	memories	Set RAM/ROM to be in external code. Cannot be used in conjunction with `offchip`
`block`	0, 1 for Xilinx, Altera 1 for Actel	0 for Xilinx 1 for Altera, Actel	memories (on-chip)	Set RAM/ROM to be in block memory
`wegate`	-1, 0, 1	0	RAMs	Place write enable signal
`westart`	in multiples of 0.5 to (clock division -0.5)	None	RAMs	Position write enable signal
`welength`	in multiples of 0.5 to clock division	None	RAMs	Set length of write enable signal
`rclkpos`	in multiples of 0.5 to (clock division -0.5)	None	memories	Set read cycle position of SSRAM clock
`wclkpos`	in multiples of 0.5 to (clock division -0.5)	None	memories	Set write cycle position of SSRAM clock
`clkpulselen`	in multiples of 0.5 to clock division	None	memories	Set pulse length of SSRAM clock

>: 12. Object specifications

Specification	Possible Values	Default	Applies to	Meaning
`clk`	Any valid pin list	None	memories (off-chip)	Set clock pins for external SSRAM clock
`addr`	Any valid pin list	None	memories (off-chip)	Set address pins
`oe`	Any valid pin list	None	memories (off-chip)	Set output enable pin(s)
`we`	Any valid pin list	None	RAMs (off-chip)	Set write enable pin(s)
`cs`	Any valid pin list	None	memories (off-chip)	Set chip select pin(s)

Clock attributes

Specification	Possible Values	Default	Applies to	Meaning
`clockport`	0, 1	0 for a port on an interface, 1 for a clock declaration	ports on interfaces, external clocks	Mark port as feeding a clock
`fastclock`	0,1	1	external clocks	Use a fast clock buffer
`rate`	Any floating point frequency in MHz	None	clocks	Minimum frequency at which the clock in question should be capable of running

Example

Specifications can be added to objects as follows:

```
unsigned 4 w with {show=0};
int 5 x with {show=0, base=2};
chanout char y with {outfile="output.dat"};
chanin int 8 z with {infile="input.dat"};
interface bus_clock_in(int 4 in) InBus() with
          { pull = 1,
            data = {"P4", "P3", "P2", "P1"}
          };
```

>: 12. Object specifications

12.1 base specification

The `base` specification may be given to variable, output channel, output bus and tri-state bus declarations. The value that this specification is set to tell the Handel-C compiler which base to display the value of the object in. Valid bases are 2, 8, 10 and 16 for binary, octal, decimal and hexadecimal respectively.

The default value of this specification is 10. If you write `with {base = 0}` this is equivalent to not specifying a base.

Example

```
int 5 x with {base=2};
```

12.2 bind specification

The bind specification may be given to an user-defined interface that connects to a component in external logic. It only has meaning when instantiating an external block of code from Handel-C generated VHDL or Verilog. If `bind` is set to 1, it is assumed that the definition of the component exists in HDL elsewhere. If it is set to 0, it does not and the component is assumed to be a black box.

In VHDL, setting `bind` to 1 instantiates the component and generates a declaration of this component of which the definition is assumed to be within the work library. Setting `bind` to 0 (default) instantiates the component and generates a black box component declaration.

In Verilog, setting `bind` to 1 instantiates the component but does not declare it. Setting `bind` to 0 instantiates the component and generates a black box component declaration. This black box component declaration is an empty module, which merely describes the interfaces of the component.

VHDL example 1: with `bind` set to 0:

```
interface Bloo(unsigned 1 myin) B(unsigned 1 myout = x)
    with {bind = 0};
```

results in Handel-C generating this VHDL instantiation of the `Bloo` component:

```
component Bloo
port (
   myin : out std_logic;
   myout : in std_logic
);
end component;
```

>: 12. Object specifications

VHDL example 2: with `bind` set to 1:

```
interface Bloo(unsigned 1 myin) B(unsigned 1 myout = x)
    with {bind = 1};
```

results in Handel-C generating this VHDL instantiation/declaration of the `Bloo` component:

```
component Bloo
port (
   myin : out std_logic;
   myout : in std_logic
);
end component;
for all : Bloo use entity work.Bloo;
```

In this case `Bloo` is bound to the work library.

Verilog example 1: with `bind` set to 0:

```
interface Bloo(unsigned 1 myin) B(unsigned 1 myout = x)
     with {bind = 0};
```

results in Handel-C generating this Verilog instantiation of the `Bloo` component:

```
module Bloo;
   input myin;
   output myout;
endmodule;

module MyModule;
   ...
   wire a, b;
   ...
   Bloo MyInstance (.myin(a), .myout(b));
   ...
endmodule;
```

Note that the code includes a black box declaration of `Bloo`.

Verilog example 2: with `bind` set to 1:

```
interface Bloo(unsigned 1 myin) B(unsigned 1 myout = x)
    with {bind = 1};
```

results in Handel-C generating this Verilog instantiation of the `Bloo` component:

>: 12. Object specifications

```
module MyModule;
   ...
   wire a, b;
   ...
   Bloo MyInstance (.myin(a), .myout(b));
   ...
endmodule;
```

(The VHDL or Verilog synthesizer expects the declaration of `Bloo` to be provided in another block of HDL.)

12.3 block specification

The `block` specification may be given to a RAM or ROM declaration. You can specify that a block RAM is created by using the specification `block = 1`. E.g.

```
ram int 8 a[15][43] with {block = 1};
```

The default value for the block specification varies according to the target device:

- For Xilinx, the default value is 0, which means that the memory will be built from LUTs rather than from block memory.
- For Altera, the default is 1.

The implementation of parallel read/writes to block memory is different on Xilinx and Altera devices.

If you want to build a ROM from look-up tables (distributed memory) in Altera devices, you need to declare the ROM with `{block = 0}`.

Example

```
static ram 8 blockRAM[2] = {12, 49} with {block = 1} ;
```

Issues with Xilinx Virtex and VirtexE

Due to the pipelined nature of Virtex block RAM, if you attempt to read from one bank of block RAM and write the value into another on a single cycle, the value read is the value in block RAM on the previous clock cycle, not the current cycle.

>: 12. Object specifications

Code example with timing issues

```
ram unsigned 8 RAM1[4] = {0,1,2,3} with {block=1};
ram unsigned 8 RAM2[4] with {block=1};
signal s;
unsigned x;
unsigned i;

while(1)
{
   par
   {
      s = RAM1[i];
      RAM2[i] = s;
      x = s;
      i++;
   }
}
```

Here, x and `Block1[i]` get different values. s changes on the falling edge. x is written to on the rising edge. `Block1[i]` is written to on the falling edge.

Therefore, `Block1[i]` gets the value of `Block0[i-1]` and x gets the value of `Block0[i]`.

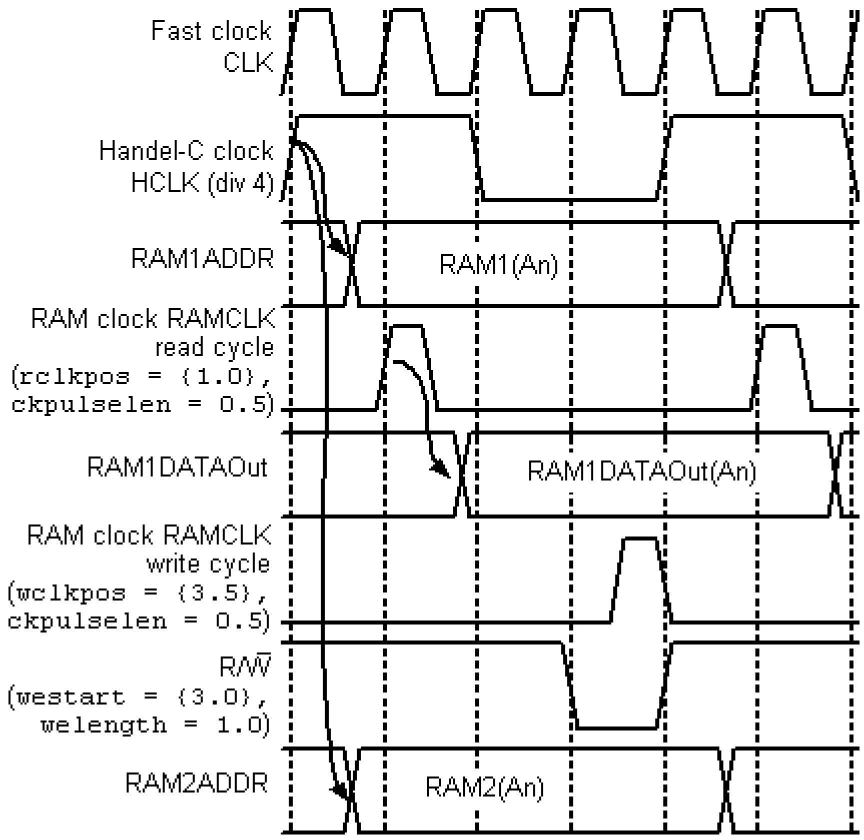
To alter this, you must use the `rclkpos`, `wclkpos` and `clkpulselen` specifications to set the RAM clock cycle positions.

Solution to timing problem

```
//divide CLK by four to give Handel-C clock
set clock = external_divide “C1” 4;

ram unsigned 8 RAM1[4] with {block = 1,
                       rclkpos = {1.0},
                       wclkpos = {3.5},
                       clkpulselen = 0.5,
                       westart = 3.0,
                       welength = 1.0};
```

>: 12. Object specifications

```
ram unsigned 8 RAM2[4] with {block = 1,
                       rclkpos = {1.0},
                       wclkpos = {3.5},
                       clkpulselen = 0.5,
                       westart = 3.0,
                       welength = 1.0};
```



HCLK initiates the parallel read from and write to the different blocks of RAM

The settings of `rclkpos` and `ckpulselen` delays the read cycle until the address is stable. (Read clock pulse 1CLK pulse after HCLK, held for 0.5 CLK pulses)

The settings of `wclkpos` and `ckpulselen` delays the write cycle until after the data has been read and is stable. The settings of `westart` and `welength` positions the write enable appropriately.

>: 12. Object specifications

12.4 busformat specification

The `busformat` specification may be given to

- generic and port-type (`port_in` and `port_out`) interfaces (but not bus-type interfaces)
- port memories (memories using `with {ports = 1}` to connect to external code)

`busformat` specifications are ignored for VHDL and Verilog output and for bus-type interfaces (`bus_in`, `bus_ts` etc).

When compiled to EDIF, the `busformat` string defines the format of the wire names. Valid values for the `busformat` string are

```
BI B_I   B[I]   B(I) B<I>
```

B represents the bus name and `I` the wire number. The default format is `B_I`

If you want to specify a single port for the entire bus, use

```
B        B[N:0]  B<N:0>     B(N:0)
```

B specifies a bus without specifying a width and `B[N:0]` and `B<N:0>` specify a bus of width (N +1). A 6-bit port could therefore be generated as `port`, `port[5:0]`or `port<5:0>` depending on the value of `busformat`.

If `data` specifications are used with `busformat`, they are ignored and a warning is issued.

You can place the `busformat` specification after any port, or at the end of an interface statement. If you place a specification at the end of the interface declaration, it will apply to all ports in the declaration, except for any ports that have their own specification. For example:

```
interface Bloo (unsigned 4 in) InstBloo (unsigned 4 out = x with
{busformat = "BI"})
                   with {busformat = "B(I)"};
// first port has spec B(I) and second port has spec BI
```

Examples

```
interface port_in(int 4 signals_to_HC with {busformat="B[I]"}) read
();
```

creates four ports named `signals_to_HC[0]`, `signals_to_HC[1]`, `signals_to_HC[2]` and `signals_to_HC[3]`:

```
interface port_in(unsigned 6 myvar) MyFunction() with {busformat =
"B[N:0]"};
```

creates a single 6-bit port: `myvar[5:0]`

```
unsigned 6 x;
interface ExtThing(unsigned 6 myvar)
     Inst1ExtThing(unsigned 6 anothervar = x)
     with {busformat = "B[N:0]"};
```

creates two ports: `myvar[5:0]` and `anothervar[5:0]`

12.5 clockport specification

The `clockport` specification can be used when declaring a port on an interface, or when declaring a clock.

Port declaration

You can use the `clockport` specification to indicate that a port on an interface is used to drive a clock in the Handel-C design. This is useful when the clock for the Handel-C design originates in an external 'black box' component. For example

```
unsigned 1 En;
interface BlackBox(unsigned 1 CLK with {clockport=1})
Instance(unsigned 1 Enable = En);

set clock = internal Instance.CLK;
```

If you don't use the `clockport` specification you may end up with combinational loops.

Clock declaration

You can use the `clockport` specification, `with {clockport=1}`, when declaring external clocks to assign the clock to a dedicated clock input resource on the target device.

If you apply the `clockport` specification to Xilinx Virtex parts, you can use it to specify a particular "input" clock buffers.

If `clockport` is set to 0, the clock is assigned to a pin that is not a dedicated clock input and the IO `standard` and `DCI` specifications are not available.

>: 12. Object specifications

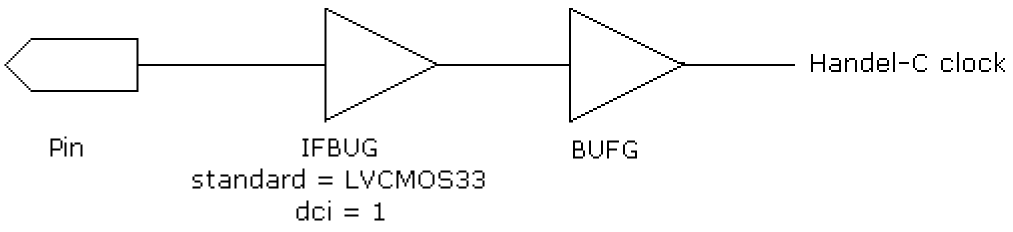
Example clock declarations

```
set family = XilinxVirtexII;
set clock = external with {standard = “LVCMOS33”, dci = 1};
```

OR

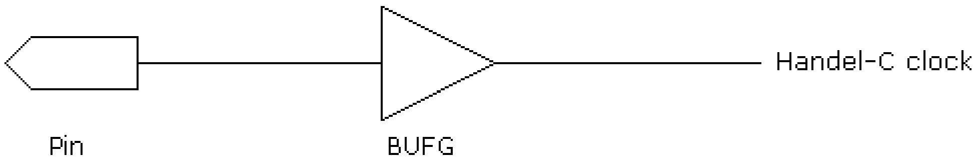
```
set family = XilinxVirtexII;
set clock = external with {clockport = 1, standard = “LVCMOS33”,
dci = 1};
```

both instruct the compiler to build an external clock interface, using a dedicated Virtex-II clock input (IBUFG) resource. That is, the clock interface logic built will be:



```
set family = XilinxVirtexII;
set clock = external with {clockport = 0, standard = “LVCMOS33”,
dci = 1};
```

This instructs the compiler to build an external clock interface, without using a dedicated Virtex-II clock input resource. That is, the clock interface logic built will be:



12.6 data specification (pin constraints)

The `data` specification can be used to constrain pin location or name ports:

- When applied to bus-type interfaces or off-chip memories, `data` specifies pin locations as a list of pin numbers separated by commas
- When applied to foreign code memories (using `with {ports=1}`), port-type interfaces and generic interfaces, `data` specifies port names as a list of names separated by commas

>: 12. Object specifications

If the `data` specification is omitted for bus-type interfaces or off-chip memories, the place and route tools will assign the pins. The pins are listed in order MSB to LSB, but the LSB pin (rightmost element of list) is assigned first. If you do not assign all the pins used, the MSB pins remain unassigned.

If you are targeting EDIF output, the `data` specification can also be used for a `port_in` or `port_out` interface to specify the names of the ports to be exported. (This part of the `data` specification is ignored for VHDL or Verilog output.)

If you are compiling your Handel-C code to VHDL or Verilog, you can only use the `data` specification to constrain pin locations for LeonardoSpectrum, FPGAExpress and Synplify outputs. If you compile for ModelSim, the `data` specification is ignored.

In LeonardoSpectrum VHDL or Verilog output, pin constraints are implemented using the `pin_number` attribute. In Synplify output, pin constraints are implemented using the `loc` attribute. If you compile your VHDL or Verilog for FPGAExpress, the pin constraints specified by data are put into a FES file (FPGAExpress script file) with the same name as your top-level VHDL or Verilog file (e.g. `MyProject.fes`).

If the `busformat` specification is used as well as `data` specifications for port-type or generic interfaces, the `data` specifications are ignored and a warning is issued.

Bus-type interface example

```
macro expr dataPins = {"P3", "P2", "P1", "P0"};
interface bus_in(unsigned 4 inPort) hword() with
     {data = dataPins, intime = 5};
```

Port-type interface example

```
macro expr dataInNames = {“I3”, “I2”, “I1”, “I0”};
macro expr dataOutNames = {“O3”, “O2”, “O2”, “O1”};

unsigned 4 x;
interface port_in(unsigned 4 in) Ig() with {data = dataInNames};
interface port_out() Og(unsigned 4 out = x) with {data =
dataOutNames};
```

>: 12. Object specifications

Generic interface example

```
macro expr dataInNames = {“I3”, “I2”, “I1”, “I0”};
macro expr dataOutNames = {“O3”, “O2”, “O2”, “O1”};

unsigned 4 x;
interface Igator
     (
     unsigned 4 in with {data = dataInNames}
     )
   InstIgator
     (
     unsigned 4 out = x with {data = dataOutNames}
     );
```

12.7 dci specification

The `dci` specification may be used with the `standard` specification on external bus interface connected to pins (not `port_in` or `port_out`) to select whether Digital Controlled Impedance is to be used on all pins of that interface. It may also be applied to off-chip memories.

The only devices that currently support DCI are Xilinx Virtex-II and Virtex-II Pro. For more information on DCI, please refer to the Xilinx Data Book.

If you have used the `clockport` specification and set it to 0, `dci` specifications will be ignored. (The default for `clockport` is 1.)

Standards supporting `dci` are:

GTL	GTL+		
HSTL Class I	HSTL Class II	HSTL Class III	HSTL Class IV
LVCMOS33	LVCMOS25	LVCMOS18	LVCMOS15
SSTL2 Class I	SSTL2 Class II	SSTL3 Class I	SSTL3 Class II

The possible values for the `dci` specification are:

`0`	No DCI (default)
`1`	DCI with single termination
`0.5`	DCI with split termination. This can only be used with LVCMOS standards.

>: 12. Object specifications

If `dci` is used on a device or standard that does not support it, a warning is issued and the specification is ignored.

Example:

```
// Use dci on all pins
interface bus_out() Eel(outPort = x) with {data = dataPinsO,
standard = "HSTL2_I", dci=1};
```

12.8 extlib, extfunc, extinst specifications

The `extlib extfunc and extinst` specifications are used when connecting a Handel-C interface to a simulation .dll. There is a default value for `extfunc`, but `extlib` and `extinst` must both be specified.

Specification	Possible Values	Default	Meaning
`extlib`	Name of a plugin `.dll`	None	Specify external plugin for simulator
`extfunc`	Name of a function within the plugin	`PlugInSet` or `PlugInGet` depending on port direction	Specify external function within the simulator for this port
`extinst`	Instance name (with optional parameters)	None	Specify simulation instance used

extlib

`extlib` takes the name of a `.dll`. It specifies that the named `.dll` plugin will be connected to the port or interface.

extfunc

`extfunc` specifies the name of an external function within the `.dll`.

On output ports, this function is called by the simulator to pass data from the Handel-C simulator to the plugin (default `PlugInSet`). It is guaranteed to be called every time the value on the port changes but may be called more often than that.

On input ports, this function is called by the simulator to get data from the plugin (default `PlugInGet`). It is guaranteed to be called at least once every clock cycle.

extinst

`extinst` takes a string, which is passed to the `PlugInOpenInstance` function within the plugin. If parameters must be passed to the `.dll` instance, they can be done so in the string. A new instance of the plugin will be generated for each unique `extinst` string.

Examples

```
interface bus_out() MyBusOut(outPort=MyOutExpr) with
     {extlib="pluginDemo.dll", extinst="0", extfunc="MyBusOut"};


interface TTL7446(unsigned 7 segments, unsigned 1 rbon)
     decode(unsigned 1 ltn=ltnVal, unsigned 1 rbin=rbinVal,
          unsigned 4 digit=digitVal, unsigned 1 bin=binVal)
             with {extlib="PluginModelSim.dll",
                extinst="decode; model=TTL7446_wrapper; delay=1"};
```

12.9 extpath specification

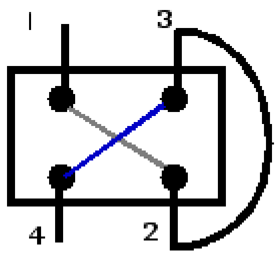
The `extpath` specification is used when connecting a Handel-C interface to external (black-box) logic. Its usage is

portName `with {extpath={`*portNameList*`}}`

portNameList is a comma-separated list of port names.

It specifies that a Handel-C output port on an interface will have direct logic connections via the black box to one or more input ports on the same interface. It is used during simulation to tell the simulator what order to update the ports in.

Example



There are direct logic connections between ports 1 and 2 and ports 3 and 4 on a black box interface. Port 2 (output) is directly connected to port 3 (input) via the Handel-C.

The interface definition would be:

>: 12. Object specifications

```
interface blackBox(int 1 Two, int 1 Four)
        bb1(int 1 One = out with {extpath = {bb1.Two}},
             int 1 Three = bb1.Two with {extpath={bb1.Four}});
```

12.10 fastclock specification

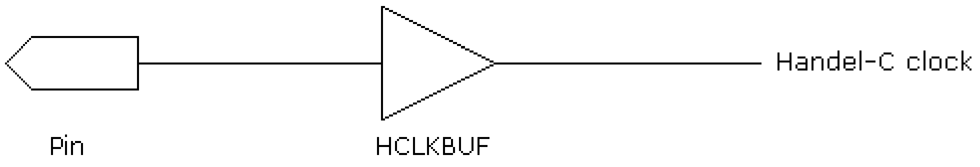
The `fastclock` specification can only be applied to clock declarations. If `fastclock` is set to 1, this specifies that an external clock should use a fast clock buffer. The default value is 0.

The fastclock specification currently only applies to Actel antifuse devices (eX, 54SX, 54SX-A, RT54SX, RT54SX-S). It is ignored for all other devices.

Examples

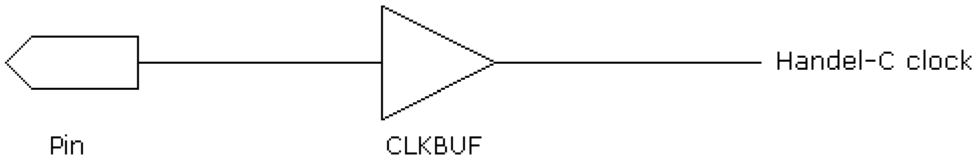
```
set family = Actel54SXA;
set clock = external with {fastclock = 1};
```

This clock definition instructs the compiler to build an external clock interface, using a fast clock input (HCLKBUF) resource. That is, the clock interface logic built will be:



```
set family = Actel54SXA;
set clock = external;

set family = Actel54SXA;
set clock = external with {fastclock = 0};
```

Both of these clock definitions instruct the compiler to build an external clock interface, using a regular clock input (CLKBUF) resource. That is, the clock interface logic built will be:



>: 12. Object specifications

12.11 infile and outfile specifications

The `infile` specification may be given to `chanin`, `port_in`, `port_out`, `bus_in`, `bus_latch_in`, `bus_clock_in`, `bus_ts`, `bus_ts_latch_in` and `bus_ts_clock_in` declarations. The `outfile` specification may be given to `chanout`, `bus_out`, `bus_ts`, `bus_ts_latch_in` and `bus_ts_clock_in` declarations. The strings that these specifications are set to will inform the simulator of the file that data should be read from (`infile`) or the file that data should be written to (`outfile`).

When applied to a variable, the state of that variable at each clock cycle is placed in that file when simulation takes place. Note that when applying the `outfile` specification, it should not be given to multiple variables or channels. For example, the following declarations are allowed, but it would be better to place them in separate files to avoid undefined results:

```
int x, y with {outfile="out.dat"};
chanout a, b with {outfile="out.dat"};
```

The filename passed to infile and outfile is a standard string and follows all string rules, including the need to specify the backslash character as '\\'.

12.12 intime and outtime specifications

The `intime` specification may be given to an input port or bus, tri-state bus, foreign code memory or off-chip memory. The `outtime` specification may be given to an output port or bus, tri-state bus, foreign code memory or off-chip memory

`intime` specifies the maximum delay in ns allowed between an interface or memory interface and the sequential elements it feeds. `outtime` specifies the maximum delay in ns allowed between an interface or memory interface and the sequential elements it is fed from. They can be floating point numbers. For example:

```
macro expr memoryPins = {"P13", "P12", "P11",
           "P10", "P9", "P8", "P7", "P6"};
macro expr dataPins = {"P4", "P3", "P2", "P1"};

interface bus_in(unsigned 4 dataIn) hword() with {data = dataPins,
intime = 5};
interface port_out()
   new_hword(unsigned 4 out = hword.dataIn + 1)
      with {outtime = 5.2};
ram int 8 a[15][43] with {outtime = 5.2,
                 offchip = 1,
                 data = memoryPins};
```

When applied to Xilinx chips, `intime` and `outtime` specifications cause Handel-C to generate a Netlist Constraints File (NCF) for the design. When an Altera device is the target,

>: 12. Object specifications

Handel-C generates ACF or TCL files. When an Actel device is targeted, Handel-C generates DCF or GCF files. These files are used by the place-and-route tools to constrain the relevant paths.

12.13 offchip specification

The `offchip` specification may be given to a RAM or ROM declaration. When set to 1, the Handel-C compiler builds an external memory interface for the RAM or ROM using the pins listed in the `clk`, `addr`, `data`, `cs`, `we` and `oe` specifications. When set to 0, the Handel-C compiler builds the RAM or ROM on the FPGA or PLD and ignores any pins given with other specifications.

```
ram int 8 a[15][43] with {offchip = 1};
```

12.14 Pin specifications

The `addr`, `data`, `we`, `cs` and `oe` specifications each take a list of device pins and are used to define the connections between the FPGA and external devices. If the specifications are omitted, the place and route tools will assign the pins. The specifications apply to the following objects:

Specification	Input bus	Output bus	Tri-state bus	RAM	ROM
`addr`	-	-	-	●	●
`data`	●	●	●	●	●
`we`	-	-	-	●	-
`cs`	-	-	-	●	●
`oe`	-	-	-	●	●
`clk`	-	-	-	●	●

Pin lists are always given in the order most significant to least significant. Multiple write enable, chip select and output enable pins can be given to allow external RAMs and ROMs to be constructed from multiple devices. For example, when using two 4-bit wide chips to make an 8-bit wide RAM, the following declaration could be used:

```
ram unsigned 8 ExtRAM[256] with {offchip=1,
     addr={"P1", "P2", "P3", "P4", "P5", "P6", "P7", "P8"},
     data={"P9", "P10", "P11", "P12", "P13", "P14", "P15", "P16"},
     we={"P17", "P18"},
     cs={"P19", "P20"},
     oe={"P21", "P22"}
};
```

12.15 ports specification

The `ports` specification may be given to a RAM, ROM or MPRAM declaration. When set to 1, the Handel-C compiler builds an external memory interface for the RAM, ROM or MPRAM using the ports defined in the `addr`, `data`, `cs`, `we`, `oe` and `clk` specifications. This allows you to connect to RAMs in external code. The compiler generates an error if the `ports` and `offchip` specification are both set to 1 for the same memory. All other specifications can be applied.

If you use the `ports` specification with an MPRAM, a separate interface will be generated for each port.

Examples

```
mpram
{
     ram <unsigned 8> ReadWrite[256];      // Read/write port
     rom <unsigned 8> Read[256];            // Read only port
} Joan with {ports = 1, busformat = "B<I>"};
```

generates EDIF ports with names prefixed by `Joan_Read` and `Joan_ReadWrite`. For example:

```
(interface
 (port Joan_Read_addr<0> (direction INPUT))
 (port Joan_Read_addr<1> (direction INPUT))

......

(interface
 (port Joan_ReadWrite_addr<0> (direction INPUT))
 (port Joan_ReadWrite_addr<1> (direction INPUT))
.....
```

>: 12. Object specifications

12.16 properties specification

The `properties` specification can be given to generic interfaces and is used to parameterize instantiations of external black boxes.

Properties are specified as a list of property items, where each item comprises two or three values:

{*property_name*, *property_value* [, *property_type*]}

- *property_name* is a string
- *property_value* can be a string or an integer
- *property_type* is optional, with 3 possible values (all strings): `"integer"`, `"boolean"` or `"string"`

If your property is a boolean, you need to specify 0 (false) or 1 (true) as the property value, and specify `"boolean"` as the type.

If your property is an integer or string, the type can be inferred from the property value and you do not need to specify it.

Each valid property is propagated through to the EDIF netlist as an EDIF property. `properties` specifications are ignored for VHDL and Verilog output.

Compiler warnings are issued if illegal values are entered, or if there is a mismatch between the property type and property value.

Example

```
unsigned 6 x;
interface ExtThing(unsigned 6 myvar)
     Inst1ExtThing(unsigned 6 anothervar = x)
     with {properties = {{“LPM_TYPE”, “LPM_RAM_DQ”},
     {“LPM_WIDTH”, 6, “integer”}}, busformat = “B[N:0]”};
```

12.16.1 Using properties: example LVDS interface

This example shows how to use the `properties` specification to create a LVDS interface for a Xilinx Virtex-II Pro.

`properties` are used to constrain the pin locations and to specify LVDS as the IO standard.

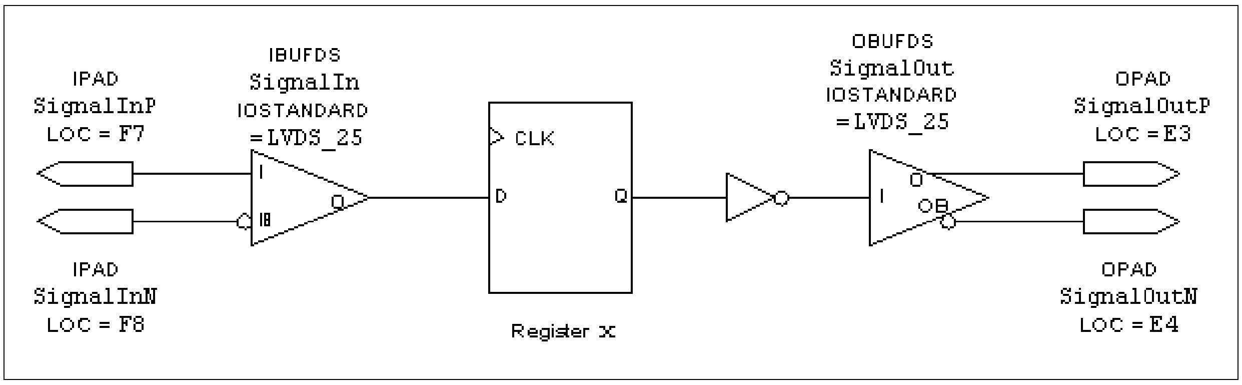
>: 12. Object specifications

```
set family = XilinxVirtexIIPro;
set part = "XC2VP20-6FF1152C";
set clock = external "D17";

/*
 * LVDS input interface
 */
interface IPAD(unsigned 1 IPAD) SignalInP() with {properties =
{{"LOC", "F7"}}};
interface IPAD(unsigned 1 IPAD) SignalInN() with {properties =
{{"LOC", "F8"}}};
interface IBUFDS
(
    unsigned 1 O
)
SignalIn
(
    unsigned 1 I = SignalInP.IPAD,
    unsigned 1 IB = SignalInN.IPAD
)
with
{
    properties = {{"IOSTANDARD", "LVDS_25"}}
};

unsigned 1 x;

/*
 * LVDS output interface
 */
interface OBUFDS
(
    unsigned 1 O,
    unsigned 1 OB
)
SignalOut
(
    unsigned 1 I = ~x
)
with
{
    properties = {{"IOSTANDARD", "LVDS_25"}}
};
```

>: 12. Object specifications

```
interface OPAD
   ()
   SignalOutP
   (unsigned 1 OPAD = SignalOut.O)
      with {properties = {{"LOC", "E3"}}};

interface OPAD
   ()
    SignalOutN
   (unsigned 1 OPAD = SignalOut.OB)
         with {properties = {{"LOC", "E4"}}};

void main(void)
{
    while(1)
    {
        x = SignalIn.O;
    }
}
```

The example creates the following circuit:



12.17 pull specification

The `pull` specification may be given to an input or tri-state bus. When set to 1, a pull up resistor is added to each of the pins of the bus. When set to 0, a pull down resistor is added to each of the pins of the bus. When this specification is not given for a bus, no pull up or pull down resistor is used.

Most Altera devices do not have pull-up or pull-down resistors. The ApexII and Mercury devices have a pull-up resistor but no pull-down resistor. Refer to the appropriate data sheet for details.

>: 12. Object specifications

Most Actel devices do not have pull-up or pull-down resistors. ProASIC and ProASIC+ devices have a pull-up resistor but no pull-down resistor. Refer to the appropriate data sheet for details.

Refer to the Xilinx FPGA data sheet for details of pull up and pull down resistors.

By default, no pull up or pull down resistors are attached to the pins.

Example

```
interface bus_clock_in(int 4 in) InBus() with
          { pull = 1,
            data = {"P4", "P3", "P2", "P1"}
          };
```

12.18 rate specification

The `rate` specification may be given to a clock, and is used to specify the frequency (in MHz) at which the clock will need to be driven. This specification causes Handel-C to generate one of the following:

- a Netlist Constraints File (NCF) for Xilinx devices
- an Assignments and Constraints File (ACF) for use with Max+PlusII for non-Apex Altera devices
- a TCL script (for use with Quartus) for Altera Apex devices.
- a Gate-field Constraints File (GCF) for Actel ProASIC and ProASIC+
- a Delay Constraints File (DCF) for Actel antifuse devices

The place-and-route tools then use these timing requirements to constrain the relevant paths so that the part of the design connected to the clock in question can be clocked at the specified rate. In the example below, the clock will need to run at 17.5MHz.

```
set clock = external_divide "D17" 4 with
     {rate = 17.5};
```

When `rate` is applied to a divided clock (as shown), it is the divided clock that will be constrained by the specification, not the external clock. Undivided clocks are also constrained to the appropriate value as calculated from the specified rate and the division factor.

>: 12. Object specifications

12.19 clkpos, wclkpos, clkpulselen and clk specifications (SSRAM timing)

The `rclkpos`, `wclkpos` and `clkpulselen` may be given to internal or external SSRAM declarations. They are specified as floating-point numbers in multiples of 0.5. The `clk` specification is used for external SSRAM declarations. To use these specifications, you must be using the `external_divide` or `internal_divide` clock types with a division factor of 2 or more.

`rclkpos` specifies the positions of the clock cycles of the RAM clock, for a read cycle. These positions are specified in terms of cycles of a fast external clock, counting forwards from the rising edge of the divided Handel-C clock rising edge.

`wclkpos` specifies the positions of the clock cycles of the RAM clock, for a write cycle.

`clkpulselen` specifies the length of the pulses of the RAM clock, in terms of cycles of a fast external clock. This is specified only once for a RAM. It thus applies to both the read and write clocks.

`clk` specifies the pin(s) that carry the RAM clock to the external SSRAM.

12.20 show specification

The `show` specification may be given to variable, channel, output bus and tri-state bus declarations. When set to 0, this specification tells the Handel-C simulator not to list this object in its output. This means that it will not appear in the Variables debug window in the GUI, but it can be seen in the Watch window.

The default value of this specification is 1.

```
int 5 x with {show=0};
```

12.21 speed specification

The `speed` specification may be given to an output or tri-state bus. The value of this specification controls the slew rate of the output buffer for the pins on the bus. For Xilinx 4000 series devices, 0 is slow, 3 is fast, and the default value is 3. For Xilinx Virtex series, Xilinx Spartan 2 series and Altera, 0 is slow, 1 is fast, and the default value is 1. Refer to the Xilinx or Altera data sheets for details of slew rate control.

For Actel, the `speed` specification is only supported for ProASIC and ProASIC+ devices, and there are three possible values: 0 (slow), 1 (normal) and 2 (fast – default value).

>: 12. Object specifications

Example

```
interface bus_out()
  drive(int 4 signals_from_HC = X_out) with {speed=0};
```

12.22 standard specification

The `standard` specification may be applied to any external bus interface connected to pins (not `port_in` or `port_out`) to select the I/O standard to be used on all pins of that interface. It may also be applied to off-chip memories. If the standard supports it, you can use the `strength` specification to set the drive current and the `dci` specification to set digital controlled impedance.

`standard` and `dci` specifications are ignored if you have used the `clockport` specification and set it to 0. (The default for `clockport` is 1.)

Different device families support different standards. Consult the data sheet for a specific device for details of which standard it supports. The compiler will issue errors if a non-supported standard is selected for a particular device, or if the `standard` specification is used on a family not supporting selectable I/O standards.

Available I/O standards

IO Standard	Handel-C keyword	IO Standard	Handel-C keyword	IO Standard	Handel-C keyword
LVTTL	`"LVTTL"`	HSTL Class I	`"HSTL_I"`	LVDS - see note 1	`"LVDS"`
LVCMOS (3.3 V)	`"LVCMOS33"`	HSTL Class II	`"HSTL_II"`	LVPECL - see note 1	`"LVPECL"`
LVCMOS (2.5 V)	`"LVCMOS25"`	HSTL Class III	`"HSTL_III"`	LVDCI (3.3 V) - see note 2	`"LVDCI_33"`
LVCMOS (1.8 V)	`"LVCMOS18"`	HSTL Class IV	`"HSTL_IV"`	LVDCI (2.5 V) - see note 2	`"LVDCI_25"`
LVCMOS (1.5 V)	`"LVCMOS15"`	SSTL2 Class I	`"SSTL2_I"`	LVDCI (1.8 V) - see note 2	`"LVDCI_18"`
PCI (33 MHz, 3.3 V)	`"PCI33_3"`	SSTL2 Class II	`"SSTL2_II"`	LVDCI (1.5 V) - see note 2	`"LVDCI_15"`
PCI (33 MHz, 5.0 V)	`"PCI33_5"`	SSTL3 Class I	`"SSTL3_I"`	LVDCI (3.3 V, split termination) - see note 3	`"LVDCI_DV2_33"`

>: 12. Object specifications

IO Standard	Handel-C keyword	IO Standard	Handel-C keyword	IO Standard	Handel-C keyword
PCI (66 MHz, 3.3 V)	`"PCI66_3"`	SSTL3 Class II	`"SSTL3_II"`	LVDCI (2.5 V, split termination) - see note 3	`"LVDCI_DV2_25"`
PCI-X	`"PCIX"`	CTT	`"CTT"`	LVDCI (1.8 V, split termination) - see note 3	`"LVDCI_DV2_18"`
GTL	`"GTL"`	AGP (1x)	`"AGP-1X"`	LVDCI (1.5 V, split termination) - see note 3	`"LVDCI_DV2_15"`
GTL+	`"GTL+"`	AGP (2x)	`"AGP-2X"`		

Notes:

1. `LVDS` and `LVPECL` are not yet supported by Handel-C. Interfaces can be created using the `properties` specification.
2. `LVDCI` standards are equivalent to using `LVCMOS` standards with a `dci` specification of 1
3. `LVDCI` split termination standards are equivalent to using `LVCMOS` standards with a `dci` specification of 0.5

Example:

```
interface bus_out() Eel(outPort=x) with {data = dataPinsO, standard
= "HSTL2_I"};

interface bus_ts(unsigned 3) Baboon(ape1=y, ape2 =
en) with {data = dataPinsT, standard = "LVTTL",
strength = 24};
```

If no I/O standard is specified, the default for Actel ProASIC and ProASIC+ is `LVCMOS33` (with drive strength “High” or “Max”). The default for all other devices is `LVTTL` (with a drive current of 12mA in the case of the Xilinx families supporting Select I/O).

Datasheets

Datasheets for Xilinx devices can be found at http://www.xilinx.com/partinfo/databook.htm

Datasheets for Altera devices can be found at http://www.altera.com/literature/lit-ds.html

Datasheets for Actel devices can be found at http://www.actel.com/techdocs/ds/index.html

>: 12. Object specifications

I/O standard	Handel-C keyword	Xilinx devices supporting standard	Altera devices supporting standard
LVTTL	`"LVTTL"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20K, Apex20KC, Apex20KE, ApexII, Mercury
LVCMOS (3.3V)	`"LVCMOS33"`	Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury
LVCMOS (2.5V)	`"LVCMOS25"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20K, Apex20KC, Apex20KE, ApexII, Mercury
LVCMOS (1.8V)	`"LVCMOS18"`	VirtexE, Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury
LVCMOS (1.5V)	`"LVCMOS15"`	Virtex-II	ApexII
PCI (33 MHz, 3.3V)	`"PCI33_3"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20K, Apex20KC, Apex20KE, ApexII, Mercury
PCI (33MHz, 5V)	`"PCI33_5"`	Spartan II, Virtex, VirtexE	-
PCI (66MHz, 3.3V)	`"PCI66_3"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20K, Apex20KC, Apex20KE, ApexII, Mercury
PCI-X	`"PCIX"`	Virtex-II	ApexII, Mercury
GTL	`"GTL"`	Spartan II, Virtex, VirtexE, Virtex-II	-
GTL+	`"GTL+"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury
HSTL Class I	`"HSTL_I"`	Spartan II, Virtex, VirtexE, Virtex-II	ApexII, Mercury
HSTL Class II	`"HSTL_II"`	Virtex-II	ApexII, Mercury
HSTL Class III	`"HSTL_III"`	Spartan II, Virtex, VirtexE, Virtex-II	-
HSTL Class IV	`"HSTL_IV"`	Spartan II, Virtex, VirtexE, Virtex-II	-
SSTL2 Class I	`"SSTL2_I"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury
SSTL2 Class II	`"SSTL2_II"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury
SSTL3 Class I	`"SSTL3_I"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury
SSTL3 Class II	`"SSTL3_II"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury

>: 12. Object specifications

I/O standard	**Handel-C keyword**	**Xilinx devices supporting standard**	**Altera devices supporting standard**
CTT	`"CTT"`	Spartan II, Virtex, VirtexE	Apex20KC, Apex20KE, ApexII, Mercury
AGP(1x)	`"AGP-1X"`	-	ApexII, Mercury
AGP(2x)	`"AGP-2X"`	Spartan II, Virtex, VirtexE, Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury
LVDS (see note 1)	`"LVDS"`	VirtexE, Virtex-II	Apex20KC, Apex20KE, ApexII, Mercury
LVPECL (see note 1)	`"LVPECL"`	VirtexE	-
LVDCI (3.3V)	`"LVDCI_33"`	Virtex-II	-
LVDCI (3.3V split termination)	`"LVDCI_DV2_33"`	Virtex-II	-
LVDCI (2.5V)	`"LVDCI_25"`	Virtex-II	-
LVDCI (2.5V split termination)	`"LVDCI_DV2_25"`	Virtex-II	-
LVDCI (1.8V)	`"LVDCI_18"`	Virtex-II	-
LVDCI (1.8V split termination)	`"LVDCI_DV2_18"`	Virtex-II	-
LVDCI (1.5V)	`"LVDCI_15"`	Virtex-II	-
LVDCI (1.5V split termination)	`"LVDCI_DV2_15"`	Virtex-II	-

Amongst Actel devices, only ProASIC and ProASIC+ support selectable I/O standards, and the only standards supported are `LVCMOS33(default)` and `LVCMOS25`.

12.22.1 I/O standard details

LVTTL – Low Voltage TTL

The Low-Voltage TTL, or LVTTL standard is a single ended, general purpose standard for 3.3V applications that uses an LVTTL input buffer and a Push-Pull output buffer. The LVTTL interface is defined by JEDEC Standard JESD 8-A, *Interface Standard for Nominal 3.0 V/3.3 V Supply Digital Integrated Circuits*. This standard requires a 3.3V output source voltage, but does not require the use of a reference voltage or a termination voltage.

LVCMOS (3.3 V) – 3.3 Volt Low-Voltage CMOS

This standard is an extension of the LVCMOS standard and is defined in JEDEC Standard JESD 8-A, *Interface Standard for Nominal 3.0 V/3.3 V Supply Digital Integrated Circuits*. This is a single-ended general-purpose standard also used for 3.3V applications. It uses a 5V-tolerant CMOS input buffer and a Push-Pull output buffer. This standard requires a 3.3V

>: 12. Object specifications

input/output source voltage, but does not require the use of a reference voltage or a board termination voltage.

LVCMOS (2.5 V) – 2.5 Volt Low-Voltage CMOS

This standard is an extension of the LVCMOS standard and is documented by JEDEC Standard JESD 8-5, 2.5 V ± 0.2 V (Normal Range) and 1.7 V to 2.7 V (Wide Range) Power Supply Voltage and Interface Standard for Non-terminated Digital Integrated Circuit. This is a single-ended general-purpose standard, used for 2.5V (or lower) applications. It uses a 5V-tolerant CMOS input buffer and a Push-Pull output buffer. This standard requires a 2.5V input/output source voltage, but does not require the use of a reference voltage or a board termination voltage. Altera documentation refers to this standard as simply “2.5 V”.

LVCMOS (1.8 V) – 1.8 Volt Low-Voltage CMOS

This standard is an extension of the LVCMOS standard and is documented by JEDEC Standard JESD 8-7, 1.8 V ± 0.15 V (Normal Range) and 1.2 V to 1.95 V (Wide Range) Power Supply Voltage and Interface Standard for Non-terminated Digital Integrated Circuit. This is a single-ended general-purpose standard, used for 1.8V power supply levels and reduced input and output thresholds. It uses a 5V-tolerant CMOS input buffer and a Push-Pull output buffer. This standard does not require the use of a reference voltage or a board termination voltage. Altera documentation refers to this standard as simply “1.8 V”.

LVCMOS (1.5 V) – 1.5 Volt Low-Voltage CMOS

This standard is an extension of the LVCMOS standard. This is a single-ended general-purpose standard, used for 1.5V applications. It uses a 5V-tolerant CMOS input buffer and a Push-Pull output buffer. This standard does not require the use of a reference voltage or a board termination voltage. Altera documentation refers to this standard as simply “1.5 V”.

PCI (33 MHz, 3.3 V) & PCI (66 MHz, 3.3 V) – 3.3 Volt PCI

The PCI standard specifies support for 33 MHz, 66 MHz and 133 MHz PCI bus applications. It uses a LVTTL input buffer and a Push-Pull output buffer. This standard requires a 3.3V input output source voltage, but not the use of input reference voltages or termination.

PCI (33 MHz, 5.0 V) – 5.0 Volt PCI

Some Xilinx devices may be configured in this mode (an extension of the 3.3 Volt PCI standard), which makes them 5V tolerant. No Altera devices currently support this mode.

PCI-X

The PCI-X standard is an enhanced version of the PCI standard that can support higher average bandwidth and has more stringent requirements.

GTL – Gunning Transceiver Logic Terminated

The GTL standard is a high-speed bus standard (JESD 8-3) invented by Xerox. Xilinx has implemented the terminated variation for this standard (Altera has not). This standard requires a differential amplifier input buffer and an Open Drain output buffer.

>: 12. Object specifications

GTL+ – Gunning Transceiver Logic Plus

The GTL+ standard is a high-speed bus standard (JESD 8-3) first used by Intel Corporation for interfacing with the Pentium Pro processor and is often used for processor interfacing or communication across a backplane. GTL+ is a voltage-referenced standard requiring a 1.0 V input reference voltage and board termination voltage of 1.5 V. The GTL+ standard is an open-drain standard that requires a minimum input/output source voltage of 3.0 V.

HSTL – High-speed Transceiver Logic

The HSTL standard, specified by JEDEC Standard JESD 8-6, High-Speed Transceiver Logic (HSTL), is a 1.5 V output buffer supply voltage based interface standard for digital integrated circuits. This is a voltage-referenced standard, and has four variations or classes. Classes I & II require a reference voltage of 0.75 V and a termination voltage of 0.75 V; classes III & IV require a reference voltage of 0.9 V and a termination voltage of 1.5 V. All four classes require an input/output source voltage of 1.5 V. This standard requires a Differential Amplifier input buffer and a Push-Pull output buffer.

SSTL2 – Stub Series Terminated Logic for 2.5 V

The SSTL2 standard, specified by JEDEC Standard JESD 8-9, *Stub-Series Terminated Logic for 2.5 Volts (SSTL-2)*, is a general purpose 2.5 V memory bus standard sponsored by Hitachi and IBM. This is a voltage-referenced standard, and has two variations or classes, both of which require a reference voltage of 1.25 V, an input/output source voltage of 2.5 V and a termination voltage of 1.25 V. This standard requires a Differential Amplifier input buffer and a Push-Pull output buffer. SSTL2 is used for high-speed SDRAM interfaces.

SSTL3 – Stub Series Terminated Logic for 3.3 V

The SSTL2 standard, specified by JEDEC Standard JESD 8-8, *Stub-Series Terminated Logic for 3.3 Volts (SSTL-3)*, is a general purpose 3.3 V memory bus standard sponsored by Hitachi and IBM. This is a voltage-referenced standard, and has two variations or classes, both of which require a reference voltage of 1.5 V, an input/output source voltage of 3.3 V and a termination voltage of 1.5 V. This standard requires a Differential Amplifier input buffer and an Push-Pull output buffer. SSTL3 is used for high-speed SDRAM interfaces.

CTT – Centre Tap Terminated

The CTT standard is a 3.3V memory bus standard, specified by JEDEC Standard JESD 8-4, *Center-Tap-Terminated (CTT) Low-Level, High-Speed Interface Standard for Digital Integrated Circuits*, and sponsored by Fujitsu. CTT is a voltage-referenced standard requiring a reference voltage of 1.5 V, an input/output source voltage of 3.3 V and a termination voltage of 1.5 V. The CTT standard is a superset of LVTTL and LVCMOS. CTT receivers are compatible with LVCMOS and LVTTL standards. CTT drivers, when un-terminated, are compatible with the AC and DC specifications for LVCMOS and LVTTL. This standard requires a Differential Amplifier input buffer and a Push-Pull output buffer.

AGP (1x, 2x) – Advanced Graphics Port

The AGP standard is specified by the A*dvanced Graphics Port Interface Specification Revision 2.0* introduced by Intel Corporation for graphics applications. AGP is a voltage-referenced standard requiring a reference voltage of 1.32 V, an input/output source voltage of 3.3 V and no termination. This standard requires a Differential Amplifier input buffer and a Push-Pull output buffer.

LVDS – Low Voltage Differential Signal

LVDS is a differential I/O standard. It requires that one data bit be carried through two signal lines. The LVDS I/O standard is used for very high-performance, low-power-consumption data transfer. Two key industry standards define LVDS: IEEE 1596.3 SCI-LVDS and ANSI/TIA/EIA-644. Both standards have similar key features, but the IEEE standard supports a maximum data transfer of 250 Mbps. The use of a reference voltage or a board termination voltage is not required, but a 100Ω termination resistor is required between the two traces at the input buffer.

LVPECL – Low Voltage Positive Emitter Coupled Logic

LVDS is a differential I/O standard. It requires that one data bit be carried through two signal lines. The LVPECL standard is similar to LVDS. In LVPECL, the voltage swing between the two differential signals is approximately 850 mV. The use of a reference voltage or a board termination voltage is not required, but an external termination resistor is required.

LVDCI - Low Voltage Digital Controlled Impedance

Xilinx Virtex II devices are able provide controlled impedance input buffers and output drivers that eliminate reflections without an external source termination. Output drivers can be configured as controlled impedance drivers, or as controlled impedance drivers with half impedance. Inputs can be configured to have termination to V_{CCO} or to $V_{CCO}/2$ (split termination), where V_{CCO} is the input/output source voltage. All of these are available at four voltage levels: 1.5 V, 1.8 V, 2.5 V and 3.3 V. For further details, please refer to the Xilinx Data Book.

12.23 std_logic_vector specification

The `std_logic_vector` specification may be given to `port_in, port_out` or generic interfaces, where you want to use a `std_logic_vector` port instead of an `unsigned` port in VHDL. Set `std_logic_vector` to 1 if you want to:

- instantiate an external block of code in Handel-C generated VHDL, and the external block uses one or more `std_logic_vector` ports
- produce a block of VHDL that will be linked into another VHDL block that uses one or more `std_logic_vector` ports.

The default value for `std_logic_vector` is 0. You can apply the `std_logic_vector` specification to an individual port. If you place the specification at the end of the interface statement, it will be applied to all the ports.

The `std_logic_vector` specification is ignored for all outputs except for VHDL

>: 12. Object specifications

Example 1: Handel-C instantiation of a `Bloo` component with `std_logic_vector` set to 0 (default):

```
interface Bloo(unsigned 1 myin) B(unsigned 4 myout = x) with
{std_logic_vector = 0};
```

results in Handel-C generating this VHDL instantiation of the `Bloo` component:

```
component Bloo
port (
   myin : out std_logic;
   myout : in unsigned (3 downto 0)
);
end component;
```

Example 2: Handel-C instantiation of a `Bloo` component with `std_logic_vector` set to 1:

```
interface Bloo(unsigned 1 myin) B(unsigned 4 myout = x) with
{std_logic_vector = 1};
```

results in Handel-C generating this VHDL instantiation of the `Bloo` component:

```
component Bloo
port (
   myin : out std_logic_vector (0 downto 0);
   myout : in std_logic_vector (3 downto 0)
);
end component;
```

12.24 strength specification

The `strength` specification may be used in conjunction with the `standard` specification on any external bus interface connected to pins (not `port_in` or `port_out`) to select the drive current to be used on all pins of that interface. It may also be applied to off-chip memories.

Different device families support different values, as shown in the table below. The compiler will issue warnings if a non-supported value is selected for a particular device.

>: 12. Object specifications

I/O Standard	Xilinx SpartanII, Virtex, VirtexE	Xilinx Virtex-II and Virtex-II Pro	Altera ApexII	Altera Mercury	Actel ProASIC and ProASIC+
LVTTL	2, 4, 6, 8, 12, 16, 24 Default: 12	2, 4, 6, 8, 12, 16, 24 Default: 12	-	-	-
LVCMOS (3.3 V)	-	2, 4, 6, 8, 12, 16, 24 Default: 12	2, 4, 6, 8, 12, 16, 24 No default	-2, 4, 6, 8, 12, 16, 24 No default	'0' (min) '-1' (max) Default: max
LVCMOS (2.5 V)	-	2, 4, 6, 8, 12, 16, 24 Default: 12	2, 4, 6, 8, 12, 16, 24 No default	2, 4, 6, 8, 12, 16, 24 No default	'0' (min) '-1' (max) Default: max
LVCMOS (1.8 V)	-	2, 4, 6, 8, 12, 16, 24 Default: 12	2, 4, 6, 8, 12, 16, 24 No default	2, 4, 6, 8, 12, 16, 24 No default	-
LVCMOS (1.5 V)	-	2, 4, 6, 8, 12, 16, 24 Default: 12	2, 4, 6, 8, 12, 16, 24 No default	-	-
GTL+	-	-	'0' (min) only	'0' (min) '-1' (max) No default	-
HSTL Class I	-	-	'0' (min) only	'0' (min) '-1' (max) No default	-
HSTL Class II	-	-	'0' (min) only	'0' (min) '-1' (max) No default	-
SSTL2 Class I	-	-	'0' (min) only	'0' (min) '-1' (max) No default	-
SSTL2 Class II	-	-	'0' (min) only	'0' (min) '-1' (max) No default	-
SSTL3 Class I	-	-	'0' (min) only	'0' (min) '-1' (max) No default	-
SSTL3 Class II	-	-	'0' (min) only	'0' (min) '-1' (max) No default	-

Example:

```
interface bus_out() Eel(outPort = x)
   with {data = dataPinsO, standard = "HSTL2_I", strength = -1};

interface bus_ts(unsigned 3 inPort) Baboon(ape1 = y, ape2 = en)
   with {data = dataPinsT, standard = "LVTTL", strength = 24};
```

12.25 warn specification

The `warn` specification may be given to a variable, RAM, ROM, channel or bus. When set to zero, certain non-crucial warnings will be disabled for that object. When set to one (the default value), all warnings for that object will be enabled.

```
int 5 x with {warn=0};
```

12.26 wegate specification

The `wegate` specification may be given to external or internal RAM declarations to force the generation of an asynchronous RAM.

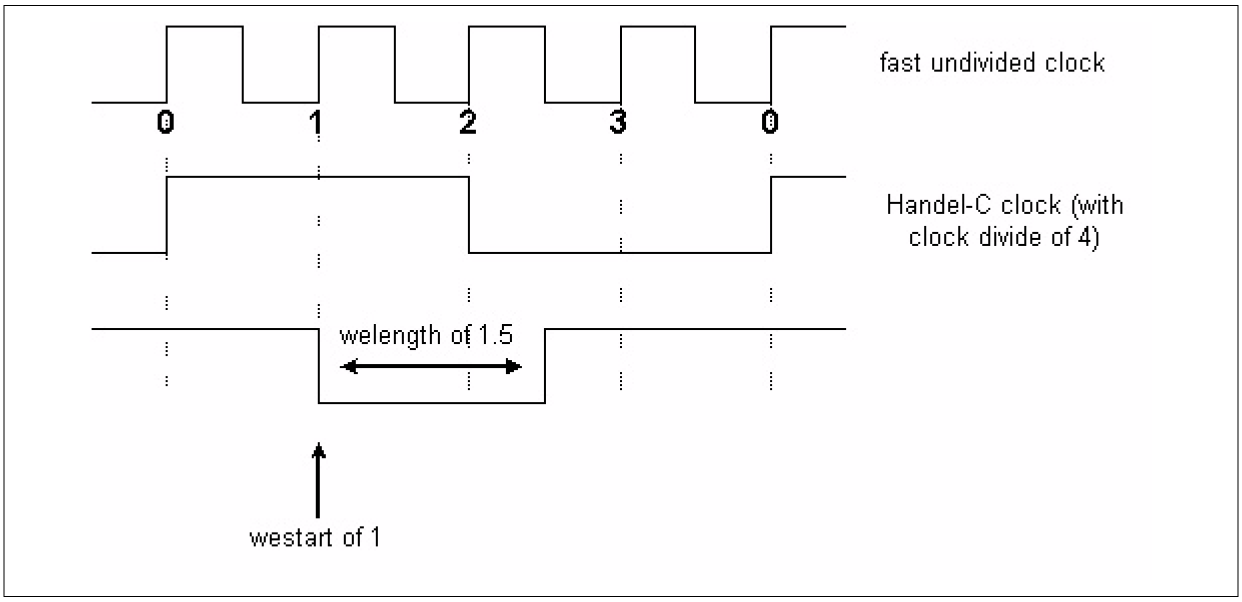
When set to 0, the write strobe will appear throughout the Handel-C clock cycle. When set to -1, the write strobe will appear only in the first half of the Handel-C clock cycle. When set to 1, the write strobe will appear only in the second half of the Handel-C clock cycle.

12.27 westart and welength specifications

The `westart` and `welength` specifications may be given to internal or external RAM declarations. You can only use these specifications together with `external_divide` or `internal_divide` clock types with a division factor greater than 1.

The `westart` and `welength` specifications position the write enable strobe within the Handel-C clock cycle. `westart` is used to specify the starting position of the write enable strobe, and `welength` is used to specify its length. For both of these specifications, a unit value corresponds to a single cycle of the fast clock which has been divided in order to generate the Handel-C clock. The size of `welength` and `westart` can be given in multiples of 0.5, but (`westart` + `welength`) must not exceed the clock divide.

>: 12. Object specifications



WRITE ENABLE STROBE WITH A WESTART OF 1, A WELENGTH OF 1.5, AND A CLOCK DIVIDE OF 4

>: 13 DK1 preprocessor

The preprocessor is invoked by the Handel-C compiler as the first stage in the compilation process, and is used to manipulate the text of source code files. Correct use of this tool can simplify code development and the subsequent maintenance process. There are a number of functions performed by the preprocessor:

- Macro substitution
- File inclusion
- Conditional compilation
- Line splicing
- Line control
- Concatenation
- Error generation
- Predefined macro substitution

Communication with the preprocessor occurs through the use of *directives*. Directives are lines within source code which begin with the # character, followed by an identifier known as the *directive name.* For example, the directive to define a macro is '`#define`'.

13.1 Preprocessor macros

Simple macros

The preprocessor supports several types of macros. Simple macros (or manifest constants) involve the simplest form of macro substitution and are defined with the form:

`#define` *name sequence-subsitute*

Any occurrences of the token *name* found in the source code are replaced with the token sequence *sequence-substitute*, which may include spaces. All leading and trailing white spaces around the replacement sequence are removed. For example:

```
#define FOO   1024
#define loop_forever  while (1)
```

Parameterized macros

You can also define macros with arguments. This allows replacement text to be passed as parameters. For example:

```
#define mul(A, B)   A*B
```

>: 13. DK1 preprocessor

This will replace

```
x = mul (2, 3);
```

with

```
x = 2 * 3;
```

Take care to preserve the intended order of evaluation when passing parameters. For example the line

```
x = mul (a – 2, 3);
```

will be expanded into

```
x = a – 2 * 3;
```

The multiplication is evaluated first, then the result subtracted from variable a. This is almost certainly not the intention, and errors of this type may be difficult to locate.

If a parameter name is preceded by a # when declared as part of a macro, it is expanded into a quoted string by the preprocessor. E.g., if a macro is defined:

```
#define quickassert(X) assert (width(X)==1,0 "Width of " #X " is not
1!\n");
```

The line:

```
quickassert(length);
```

will expand into:

```
assert (width(X)==1,0 "Width of length is not 1!\n");
```

Undefining identifiers

To undefine an identifier, the `#undef` directive may be used. E.g.

```
#undef FOO
```

Note that no error will occur if the identifier has not previously been defined.

Preprocessor directives cannot be used unexpanded in a library; use macro procedures instead.

>: 13. DK1 preprocessor

13.2 File inclusion

File inclusion makes it possible to easily manage and reuse declarations, macro definitions, and other code. The feature is helpful when writing general purpose functions and declarations which can be reused for a number of designs. File inclusion is achieved using directives of the form:

`#include` "*filename*"

or

`#include` <*filename*>

Such lines are replaced by the contents of the file indicated by *filename*. If the filename is enclosed by quotation marks, the preprocessor looks for the file in the directory containing source code for the current design. If the file cannot be found there, or the filename is enclosed with angular brackets, the search examines user-defined include file directories (specified using **Tools**>**Options**>**Directories**), and the main DK1 include file directory.

13.3 Conditional compilation

Conditional directives

You can control preprocessing with conditional directives. These statements can add a great deal of flexibility to source code. For example, they may be used to alter the behaviour of a design, depending upon whether a macro definition is present. Conditional statements must begin with an `#if` directive and an expression to be evaluated, and end with the `#endif` directive. Valid directives are:

`#if` *expression*

`#elif` *expression*
`#else`

`#endif`

>: 13. DK1 preprocessor

Example

```
#if a==b
   /* include this section if a is equal to b */
#elif a>b
   /* include this section if a is greater than b */
#else
   /* otherwise include this section */
#endif
```

If the expression is evaluated to be zero, then any text following the directive will be discarded until a subsequent `#elif`, `#else`, or `#endif` statement is encountered; otherwise the lines will be included as normal. Note that each directive should be placed individually on its own line starting at column 0.

A useful application for conditional directives is easy exclusion of code without the use of comments. For example:

```
#if (0)
   /* Code for debugging purposes*/
#endif
   /* Code continues */
```

By amending the above evaluation to (1), the code can quickly be included during compilation.

Conditional definition

To test for the existence of a macro definition, use the following directives:

`#ifdef`	*identifier*	(equivalent to #if defined (identifier))
`#ifndef`	*identifier*	(equivalent to #if !defined (identifier))

These are used in the same way as `#if`, but are followed by an identifier, rather than an expression. The `#ifndef` directive is often used to ensure that source code is only included once during compilation. E.g.

```
#ifndef UTILCODE
#define UTILCODE

   /* Utility code is written here */

#endif
```

>: 13. DK1 preprocessor

13.4 Line splicing

You can splice multiple lines together by placing a backslash character ('\') followed by a carriage return between them. This feature allows you to break lines for aesthetic purposes when writing code, which are then joined by the preprocessor prior to compilation. For example, if a macro is defined:

```
#define ERRORCHECK(error) \
   if (error!=0)  \
      return (error)
```

The line:

```
ERRORCHECK(i);
```

Expands to:

```
if (i!=0)
   return i;
```

13.5 Line control

A directive of the form:

`#line` *integer*

instructs the compiler that the next source line is the line number specified by *integer*. If a filename token is also present:

`#line` *integer* `"`*filename*`"`

the compiler will additionally regard *filename* as the name of the current input file.

13.6 Concatenation in macros

If a macro is defined with a token sequence containing a ## operator, each instance of ## is removed (along with any surrounding white space), thus concatenating adjacent tokens into one. For example, if the macro below was declared:

```
   #define million(X) X ## e6
```

then

```
   i = million (3);
```

is expanded into:

```
i = 3e6;
```

Take care when specifying parameters. In the example above, if 3e6 was passed instead of 3, then the line would be expanded into:

```
i = 3e6e6;
```

which would result in an error.

13.7 Error generation

Fatal error messages may be reported during preprocessing using the directive:

`#error` *error_message*

This may be useful with conditional compilation if your design only supports certain combinations of parameter definitions.

13.8 Predefined macro substitution

The preprocessor contains a number of useful predefined macros which may be placed into source code:

`_ _FILE_ _`	Expands to the name of the current file being compiled
`_ _LINE_ _`	Expands to the number of the current source line
`_ _TIME_ _`	Expands to the current time of compilation in the form hh:mm:ss
`_ _DATE_ _`	Expands to the current date of compilation in the form mmm dd yyyy

>: 14. Language syntax

>: 14 Language syntax

The complete Handel-C language syntax is given in BNF-like notation.

The overall syntax for the program is:

program `::=` {*global_declaration*}

```
void main(void) {
        {declaration}

        {statement}

        }
```

Language

```
external_declaration  ::=  function_definition
                         |  declaration
                         |  set_statement ;
```

14.1 Language syntax conventions

BNF (Backus-Naur Format) is a way to describe the syntax of file formats. It consists of definitions of the form

identifier ::= *definition*

The *identifier* is a word which describes this part of the syntax.
The ::= represents "consists of".
The *definition* lists the permitted contents of the *identifier*.

The conventions used in this language reference are:

- Terminal symbols are set in typewriter font `like this`.
- Non-terminal symbols are set in italic font *like this*.
- Square brackets [...] denote optional components.
- Braces {...} denotes zero, one or more repetitions of the enclosed components.
- Braces with a trailing plus sign {...}$^+$ denote one or several repetitions of the enclosed components.
- Parentheses (...) denote grouping.

>: 14. Language syntax

14.2 Keyword summary

The keywords listed below are reserved and cannot be used for any other purpose.

Keyword	Meaning	ANSI-C/C++ ?
=	assignment operator	Yes
;	statement terminator	Yes
,	C only	Yes
{ }	code block delimiters	Yes
<>	type specialization	No
(	open delimiter	Yes
)	close delimiter	Yes
[ ]	array index delimiters, bit selection	Yes
[:]	bit range selection	No
!	logical NOT operator	Yes
!	output to channel	No
~	bitwise NOT	Yes
+	addition operator	Yes
-	subtraction operator	Yes
-	unary minus operator	Yes
*	multiplication operator	Yes
/	division operator	Yes
%	modulo operator	Yes
\\	drop LSB	No
<-	take LSBs	No
?	read from channel	No
?	conditional expression	Yes
^	Bitwise XOR	Yes
&	Bitwise AND	Yes
|	Bitwise OR	Yes
&&	Logical AND	Yes
||	Logical OR	Yes
.	structure member operator	Yes
<<	left-shift operator	Yes
>>	right shift operator	Yes
<	less than operator	Yes
>	greater than operator	Yes

>: 14. Language syntax

Keyword	Meaning	ANSI-C/C++ ?
<=	less or equal operator	Not standard
>=	greater or equal operator	Not standard[1]
==	equality operator	Not standard[1]
!=	inequality operator	Not standard[1]
++	increment operator	Not standard
--	decrement operator	Not standard
+=	assignment operator	Not standard
-=	assignment operator	Not standard
*=	assignment operator	Not standard
/=	assignment operator	Not standard
%=	assignment operator	Not standard
<<=	assignment operator	Not standard
>>=	assignment operator	Not standard
&=	assignment operator	Not standard
|=	assignment operator	Not standard
^=	assignment operator	Not standard
...	Reserved. Not valid in Handel-C, but can be used for C/C++ calls.	Yes
->	structure pointer operator	Yes
@	concatenation operator	No

[1] Note, the results of these tests are a single bit `unsigned int`

Keyword	Meaning	ANSI-C/C++ ?
`assert`	diagnostic macro to print to stderr	Not standard
`auto`	auto variable	Yes
`break`	immediate exit from code block	Yes
`case`	selection within switch and prialt	Yes
`chan`	define channel variable	No
`chanin`	simulator channel in	No
`chanout`	simulator channel out	No
`char`	8-bit variable	Yes
`clock`	define clock	No
`const`	specify that variable's value will not change	Yes

>: 14. Language syntax

Keyword	Meaning	ANSI-C/C++ ?
`continue`	force next iteration of loop	Yes
`default`	default case within switch, prialt	Yes
`delay`	wait one clock cycle	No
`do`	start do while loop	Yes
`double`	Reserved. Not valid in Handel-C	C-only
`else`	conditional execution	Yes
`enum`	enumeration constant	Yes
`expr`	define macro as expression	No
`extern`	define global variable	Yes
`external`	clock from device pin	No
`external_divide`	clock from device pin with integer division	No
`family`	define target device's family	No
`float`	Reserved. Not valid in Handel-C	C-only
`for`	for loop iteration	Yes
`goto`	jump to specified label	Yes
`if`	conditional execution	Yes
`ifselect`	conditional compilation on compile-time selection	No
`in`	define scope for local macro expression declaration	No
`inline`	declaration of inline function	No
`int`	definable width variable	Yes
`interface`	declaration of off-chip interface	No
`internal`	use internal clock	No
`internal_divide`	internal clock with integer division	No
`intwidth`	set integer width	No
`let`	start declaration of local macro expression	No
`long`	declare 32-bit variable	Yes
`macro`	declare a macro	No
`mpram`	declare a multi-port RAM	No
`par`	execute statements in parallel	No
`part`	define target hardware	No
`prialt`	execute first ready channel	No
`proc`	define macro as procedure	No
`ram`	declare a RAM (array)	No
`register`	declare register variable	Yes

>: 14. Language syntax

Keyword	Meaning	ANSI-C/C++ ?
`releasesema`*(semaphore)*	free *semaphore*	No
`reset`	reset design	No
`return`	return from function	Yes
`rom`	declare a ROM (array)	No
`select`	select expression or macro expr at compile time	No
`sema`	declare a semaphore	No
`set`	specify device family or part, int width, target, reset or clock	No
`seq`	execute statements in sequence	No
`shared`	declare a shared expression	No
`short`	declare 16-bit variable	Yes
`signal`	declare a signal object	No
`signed`	declare a signed variable	Yes
`sizeof`	Reserved. Not valid in Handel-C	Yes
`static`	specify variable with limited scope	Yes
`struct`	declare a structure variable	Yes
`switch`	switch statement (between cases)	Yes
`try reset(`*Condition*`){...}`	execute statements if *Condition* is true during execution within related try block	No
`trysema`	Test if semaphore owned. Take if not.	No
`typedef`	define type	Yes
`typeof`	return type of expression	No
`undefined`	specify a variable of undefined width	No
`union`	Reserved. Not valid in Handel-C	Yes
`unsigned`	declare an unsigned variable	Yes
`void`	specify void return type,	Yes
`volatile`	declare volatile variable	Yes
`while`	loop statement	Yes
`width`	return integer width	No
`with`	specify interface, signals, channels, RAM and ROM types, variables etc.	No
`wom`	declare a WOM (array)	No

The following character sequences are also reserved:
/*   */   //   #   "   '

>: 14. Language syntax

14.3 Constant expressions

The following constants are available in Handel-C

- Identifiers
- Integer constant
- Character constants
- String constant
- Floating point constants

14.3.1 Identifiers: syntax

Identifiers are sequences of letters, digits and _, starting with a letter. All characters in an identifier are meaningful and all identifiers are case sensitive.

identifier `::=` *letter* `{`*letter* **|** `0`**...**`9}`

letter `::=` `A`**...**`Z` **|** `a`**...**`z` **|** `_`

14.3.2 Integer constants: syntax

integer_constant `::= [-]{1...9}`$^+$`{0...9}`
```
        | [-](0x | 0X){0...9 | A...F | a...f}+
        | [-](0){0...7}
        | [-](0b | 0B){0...1}+
```

14.3.3 Character constants: syntax

character is any printable character or any of the following escape codes.

Escape Code	ASCII Value	Meaning
`\a`	7	Bell (alert)
`\b`	8	Backspace
`\f`	12	Form feed
`\t`	9	Horizontal tab
`\n`	10	Newline
`\v`	11	Vertical tab
`\r`	13	Carriage return
`\"`	-	Double quote mark

>: 14. Language syntax

`\0`	0	String terminator
`\\`	-	Backslash
`\'`	-	Single quote mark
`\?`	-	Question mark

14.3.4 Strings: syntax

```
string ::= "{character}"
```

14.3.5 Floating point constants: syntax

float_constant`::=`
```
     [{0...9}+].{0...9}+[(e | E)[+|-]{0...9}+][f | F | l | L]
   | {0...9}+.[(e | E)[+|-]{0...9}+][f | F | l | L]
   | {0...9}+(e | E)[+|-]{0...9}+[f | F | l | L]
```

14.4 Functions and declarations

function_definition ::= *declaration_specifiers declarator compound_statement* `[ with` *initializer* `;]`
| *declarator compound_statement* `[ with` *initializer ;*`]`

declaration ::= declaration_specifiers `[` *init_declarator_list*`]` `[with` *initializer* `] ;`
| *interface_declaration*
| *macro_declaration*

declaration_specifiers ::= storage_class_specifier `[` *declaration_specifiers*`]`
| *type_specifier* `[` *declaration_specifiers*`]`
| *type_qualifier* `[` *declaration_specifiers*`]`

storage_class_specifier ::= `auto`
```
| register
| inline
| typedef
| extern
| static
```

>: 14. Language syntax

type_specifier ::= `void`
  | `char`
  | `short`
  | `int`
  | `long`
  | `float`
  | `double`
  | `signed`
  | `unsigned`
  | `typeof (` *expression* `)`
  | *signal_specifier*
  | *channel_specifier*
  | *ram_specifier*
  | *struct_or_union_specifier*
  | *enum_specifier*
  | *typedef_name*

type_qualifier ::= `const`
  | `volatile`

typedef_name ::= *identifier*

init_declarator_list ::= *declarator* [= *initializer*] { ,*declarator* [= *initializer*]}

14.5 Macro / shared expressions / procedures: syntax

macro_declaration ::= *macro_proc_decl*
  | *macro_expr_decl*

macro_proc_decl ::= [`static` | `extern`] *macro_proc_spec identifier*
  [([*macro_param*{, *macro_param*}])] *statement*
  [`with` *initializer* ;]

macro_expr_decl ::= [`static` | `extern`] *macro_expr_spec identifier*
  [([*macro_param*{, *macro_param*}])] ;
  | [`static` | `extern`] *macro_expr_spec identifier*
  [([*macro_param*{, *macro_param*}])] = *let_initializer*
  [`with` *initializer*] ;

>: 14. Language syntax

macro_proc_spec ::= `macro proc`
        | `shared proc`

macro_expr_spec ::= `macro expr`
        | `shared expr`

let_initializer ::= *initializer*
        | `let` *macro_expr_decl* `in` *let_initializer*

macro_param ::=   *identifier*

14.6 Interfaces: syntax

interface_declaration ::= interface identifier ([*int_parameter_declaration* { , *int_parameter_declaration*}])
        identifier ([*assignment_expr_spec* {, *assignment_expr_spec*}]) [`with` *initializer];*
        | *interface_type_declarator*
        | *old_style_interface_declarator*

interface_type_declarator :: = *interface identifier* ([*int_parameter_proto*{ , *int_parameter_proto*}])
        identifier ([*int_init_parameter_declaration* { , *int_init_parameter_declaration*}])

This format is deprecated but retained for compatibility reasons

old_style_interface_declarator ::= interface identifier ([*int_parameter_declaration* {,*int_parameter_declaration*}])
        identifier ([*assignment_expr_spec* {,*assignment_expr_spec*})
        [`with` *initializer*] ;

interface ::= [`static` | `extern`] `interface`

int_parameter_proto::= declaration_specifiers
        | *declaration_specifiers declarator*
        | *declaration_specifiers abstract_declarator*
        | *declaration_specifiers width*

>: 14. Language syntax

int_parameter_declaration ::= declaration_specifiers [with *initializer*]
| *declaration_specifiers declarator* [with *initializer*]
| *declaration_specifiers abstract_declarator* [with *initializer*]
| *declaration_specifiers width* [with *initializer*]

int_init_parameter_declaration ::= int_parameter_declaration
| *declaration_specifiers declarator* [= *initializer]* [with *initializer*]

assignment_expr_spec ::= assignment_expression [with *initializer*]

14.7 Structures and unions: syntax

struct_or_union_specifier ::= aggregate_form [*identifier]* { {*struct_declaration*}+ }
| *aggregate_form identifier*

aggregate_form ::= struct
| union
| mpram

struct_declaration ::= { *type_specifier* | *type_qualifier}+*
{*struct_declarator*}+[with *initializer*];

struct_declarator ::= declarator
| [*declarator*] : *constant_expression*

The current version of Handel-C does not support unions.

14.8 Enumerated types: syntax

enum_specifier ::= enum [*identifier*] { *enumerator* {,[*enumerator*]} }
| enum *identifier*

enumerator ::= identifier
| *identifier = constant_expression*

>: 14. Language syntax

14.9 Signal specifiers: syntax

signal_specifier ::= `signal` < *type_name* >
        | `signal`

14.10 Channel syntax

channel_specifier ::= `chan` [< *type_name* >]
        | `chanin` [< *type_name* >]
        | `chanout` [< *type_name* >]

14.11 Ram specifiers: syntax

ram_specifier ::= `ram` [< *type_name* >]
        | `rom` [< *type_name* >]
        | `wom` [< *type_name* >]

14.12 Declarators: syntax

declarator ::= [*width*] *pointer direct_declarator*

width ::= `undefined`
        | *primary_expression*

direct_declarator ::= *identifier*
        | (*pointer direct_declarator*)
        | *direct_declarator* [[*constant_expression*]]
        | *direct_declarator* ([{*parameter_declaration*}+])

pointer ::= *
        | * *type_qualifier*
        | * *pointer*
        | * *type_qualifier pointer*

>: 14. Language syntax

14.13 Function parameters: syntax

parameter_declaration ::= declaration_specifiers
| *declaration_specifiers width*
| *declaration_specifiers abstract_declarator*
| *declaration_specifiers declarator*

14.14 Type names and abstract declarators: syntax

type_name ::= { *type_specifier* | *type_qualifier*}+
| { *type_specifier* | *type_qualifier*}+ *abstract_declarator*
| { *type_specifier* | *type_qualifier*}+ *width*

abstract_declarator ::= [*width*] *pointer direct_abstract_declarator*

direct_abstract_declarator ::= (*pointer direct_abstract_declarator*)
| [*direct_abstract_declarator*][[*constant_expression*]]
| [*direct_abstract_declarator*] ([{*parameter_declaration*}+])

14.15 Statements: syntax

statement ::= semi_statement ;
| *non_semi_statement*

semi_statement ::= expression_statement
| `do` *statement* `while` (*expression*)
| *jump_statement*
| `assert` (*constant_expression* [, *assignment_expression*{,
`assignment_expression}] )`
| `delay`
| *channel_statement*
| *set_statement*

non_semi_statement ::= labeled_statement
| *compound_statement*
| *selection_statement*
| *iteration_statement*

>: 14. Language syntax

The following statements can appear in `for` start/end conditions

for_statement ::= non_semi_statement
  | *expression_statement*
  | `do` *statement* `while` (*expression*)
  | `assert` (*constant_expression* , *constant expression*
      `[,` *assignment_expression*`{, assignment_expression}] )`
  | `delay`
  | *channel_statement*

These are the statements that can appear in `prialt` blocks

prialt_statement ::= semi_statement `;`
  | *non_semi_prialt_statement*

non_semi_prialt_statement ::= prialt_labeled_statement
  | *compound_statement*
  | *selection_statement*
  | *iteration_statement*

labeled_statement ::= identifier `:` *statement*
  | `case` *constant_expression* `:` *statement*
  | `default` : *statement*

prialt_labeled_statement ::= identifier `:` *prialt_statement*
  | `case` *channel_statement* `:` *prialt_statement*
  | `default` : *prialt_statement*

expression_statement ::= `[`*expression*`]`

channel_statement ::= unary_expression `!` *expression*
  | *logical_or_expression* ? *expression*

jump_statement ::= `goto` *identifier*
  | `continue`
  | `break`
  | `return`
  | `return` *expression*

>: 14. Language syntax

selection_statement ::= `if` (*expression*) *statement* `if`
 | `if` (*expression*) *statement* `else` *statement*
 | `ifselect` (*constant_expression*) *statement* `if`
 | `ifselect` (*constant_expression*) *statement* `else` *statement*
 | `switch` (*expression*) *statement*
 | `prialt` { [{*prialt_statement*}+] }

set_statement ::= `set part` = *STRING*
 | `set clock` = *clock*
 | `set family` = *identifier*
 | `set intwidth` = *constant_expression*
 | `set intwidth` = `undefined`
 | `set reset` = *reset*

clock ::= `internal` *expression* [`with` *initializer*]
 | `external` *expression* [`with` *initializer*]
 | `internal_divide` *expression expression* [`with` *initializer*]
 | `external_divide` *expression expression* [`with` *initializer*]

reset ::= `internal` *expression*
 | `external` *expression*

`iteration_statement ::= while` (*expression*) *statement*
 | `for` ([*for_statement*] ; [*expression]* ; [*for_statement*]) *statement*

14.15.1 Compound statements with replicators

compound_statement ::= [`seq | par`] {{ *declaration*} {*statement*} }
 | [`seq | par`] ([*repl_macro_param*{`,` *repl_macro_param*}]
;*constant_expression;*
 [*repl_update_param* {`,` *repl_update_param*}]) {{*declaration*}
{*statement*} }

14.16 Replicator syntax

Replicator initialization definitions

repl_macro_param `::=` *repl_param* = *initializer*
 | (*repl_param* = *initializer*)

>: 14. Language syntax

Replicator update definitions

repl_update_param ::= repl_update_param_body
| (*repl_update_param*)

repl_update_param_body ::= repl_param assignment_operator initializer
| ++ *repl_param*
| *repl_param* ++
| -- *repl_param*
| *repl_param* --

repl_param ::= identifier
| (*repl_param*)

14.17 Expressions: syntax

constant_expression ::= *assignment_expression*

expression ::= *assignment_expression*
| *expression*, *assignment_expression*}

assignment_expression ::= *conditional_expression*
| *unary_expression assignment_operator assignment_expression*

assignment_operator ::= = | *= | /= | %= | += | -= | <<= |
>>= | &=
| ^= | |=

initializer ::= *assignment_expression*

conditional_expression ::= *logical_or_expression*
| *logical_or_expression* ? *expression* : *conditional_expression*

logical_or_expression ::= *logical_and_expression*
| *logical_or_expression* || *logical_and_expression*

logical_and_expression ::= *inclusive_or_expression*
| *logical_and_expression* && *inclusive_or_expression*

inclusive_or_expression `::=` *exclusive_or_expression*
| *inclusive_or_expression* `|` *exclusive_or_expression*

exclusive_or_expression `::=` *and_expression*
| *exclusive_or_expression* `^` *and_expression*

and_expression `::=` *equality_expression*
| *and_expression* `&` *equality_expression*

equality_expression `::=` *relational_expression*
| *equality_expression* `==` *relational_expression*
| *equality_expression* `!=` *relational_expression*

relational_expression `::=` *cat_expression*
| *relational_expression* `<` *cat_expression*
| *relational_expression* `>` *cat_expression*
| *relational_expression* `<=` *cat_expression*
| *relational_expression* `>=` *cat_expression*

cat_expression `::=` *shift_expression*
| *cat_expression* `@` *shift_expression*

shift_expression `::=` *additive_expression*
| *shift_expression* `<<` *additive_expression*
| *shift_expression* `>>` *additive_expression*

additive_expression `::=` *multiplicative_expression*
| *additive_expression* `+` *multiplicative_expression*
| *additive_expression* `-` *multiplicative_expression*

multiplicative_expression `::=` *take_drop_expression*
| *multiplicative_expression* `*` *take_drop_expression*
| *multiplicative_expression* `/` *take_drop_expression*
| *multiplicative_expression* `%` *take_drop_expression*

take_drop_expression `::=` *cast_expression*
| *take_drop_expression* `<-` *cast_expression*
| *take_drop_expression* `\\` *cast_expression*

>: 14. Language syntax

```
cast_expression ::= unary_expression
        |   ( type_name ) cast_expression

unary_expression ::= postfix_expression
        |   ++ unary_expression
        |   -- unary_expression
        |   unary_operator cast_expression
        |   sizeof unary_expression
        |   sizeof ( type_name )
        |   width ( expression )

unary_operator ::= &  |  +  |  -  |  ~  |  !  |  *

postfix_expression ::= select_expression
        |   postfix_expression [ expression ]
        |   postfix_expression [ expression : expression ]
        |   postfix_expression [ : expression ]
        |   postfix_expression [ expression : ]
        |   postfix_expression [ ]
        |   postfix_expression ( [assignment_expression
                  {, assignment_expression}] )
        |   postfix_expression . identifier
        |   postfix_expression -> identifier
        |   postfix_expression ++
        |   postfix_expression --

select_expression ::= primary_expression
        |   select ( constant_expression , constant_expression ,
              constant_expression )

primary_expression ::= identifier
        |   constant
        |   ( expression )
        |   { }
        |   {[initializer {, initializer}[, ] ]}

constant ::= integer_constant
        |   character_constant
        |   string_constant
```

>: 14. Language syntax

integer_constant `::=` *NUMBER*

character_constant `::=` *CHARACTER*

string_constant `::=` *STRING*

Appendix

Appendix: Changes in Handel-C version 3

This chapter describes the changes between Handel-C version 2.1 and Handel-C version 3.0. Handel-C version 3.0 was released with the first release of DK1.

A.1 Operators: changes in version 3

Operator	Meaning	ANSI-C	Change in Version 3
`[ ]`	array index delimiters, bit selection	Extended	Array index may be a variable, bit selection may not
`[ : ]`	bit range selection	Not standard	Extended: the value before or after ':' can be omitted
.	structure, and multi-port RAM member operator, interface port operator	Yes	`struct` variables have been added
`*`	indirection operator	Yes	New
&	address operator	Yes	New
`/`	division operator	Yes	Extended: division of variables
`%`	modulo operator	Yes	Extended: modulo of variables
<<	left-shift operator	Yes	Extended: shift by variable amounts
>>	right shift operator	Yes	Extended: shift by variable amounts
`assert`	diagnostic macro to print to `stderr`	Not standard	Print string to standard error channel

Appendix

A.2 Declarations: changes in version 3

Keyword	Meaning	ANSI-C	Change in v.3
`<>`	disambiguator	No	New
`auto`	auto variable	Yes	New
`const`	specify that variable's value will not change	Yes	New
`enum`	enumeration constant	Yes	New
`extern`	define global variable	Yes	New
`inline`	declaration of inline function	No	New
`interface`	declaration of off-chip interface	No	Extended: you can now create interfaces to foreign code
`mpram`	declare a multi-port RAM	No	Create dual-ported RAMs
`ram`	declare a RAM	No	Extended to specify block memory
`register`	declare register variable	Yes	New
`rom`	declare a ROM	No	Extended to specify block memory
`sema`	declare a semaphore	No	New
`signal`	declare a signal object	No	New
`signed`	declare a signed variable	Yes	New
`static`	specify variable with limited scope	Yes	New
`struct`	declare a structure variable	Yes	New
`typedef`	define type	Yes	New
`void`	specify void return type or empty parameter list	Yes	New
`volatile`	declare volatile variable	Yes	New
`wom`	declare a WOM (array)	No	Specify an area of write-only memory

Appendix

A.3 Statements: changes in version 3

Statement	**Meaning**	**ANSI-C**	**Change in v.3**
`continue`	continue execution outside code block	Yes	New
`goto`	jump to specified label	Yes	New
`ifselect`	conditional compilation on compile time selection	No	Compile following code if selected, else…
`par`	execute statements in parallel	No	Extended: parallel statements can be replicated
`releasesema` (*semaphore*)	free semaphore	No	New
`return`	*return from function*	*Yes*	*New*
`seq`	execute statements in sequence	No	New: seq blocks can also be replicated
`try reset`(*Condition*) {*statement*}	execute statements if *Condition* is true during execution within related try block	No	New
`trysema` *(semaphore)*	test semaphore	No	New
`typeof`	return type of operator	No	As in GNU C

A.4 Macros: changes in version 3

Keyword	**Meaning**	**ANSI-C**	**Change in version 3**
`in`	define scope for local macro expression declaration	No	New: `let macro expr` *name* **=** *expression* **`in`** *macro expression*
`let`	start declaration of local macro expression	No	New: **`let`** `macro expr` *name* **=** *expression* `in` *macro expression*

Appendix

In version 3, macro procedures that don't take any parameters require an empty parameter list. This was not required in version 2.1**.**

A.5 Clocks: changes in version 3

Keyword	**Meaning**	**ANSI-C**	**Change in version 3**
`internal`	use internal clock	No	Extended: can use any expression
`internal_divide`	use divided internal clock	No	Extended: can use any expression
`__clock`	use current clock	No	New

The `rate` specification been added, allowing you to specify the clock rate that the design should run at. This gives a maximum delay between components. It does not appear in ANSI-C.

A.6 Other language changes in version 3

Linker changes

Multiple files can be linked together and loaded into a single FPGA. This allows you to create and access library files.

You can load a single chip with multiple main functions. This means that you can have independent logic blocks using different clocks running within the same FPGA. The clock can be internal or external. External clocks may be user specified.

ANSI-C compatible extensions

Compatibility with ANSI standard C has been increased, so most standard types and derived types are supported. This includes pointers and structures but does not include floats. `goto`, `continue` and `return` are supported. (Note that you cannot use `goto`, `continue`, `break` or `return` to enter or exit from a `par` statement.)

Handel-C now supports functions. These can be used instead of macros. Functions can be immediately expanded using the `inline` keyword

To support the multiple files system, prototypes are supported, as are the ANSI-C keywords `extern` and `static`.

You can send messages to the standard error channel during compilation using the `assert` directive.

Appendix

Macro changes

You can now declare local variables inside a macro expression with `let ... in`

There is a new statement, `ifselect`, which permits conditional compilation according to the result of a test at compile time.

Statements

The Handel-C language has been extended to allow code to be *replicated* using a construct similar to a `for` loop. This means that you can generate multiple identical copies of the same block of code, either in sequence or in parallel.

Initialization

Only static or global variables may be initialized in version 3. In previous versions all memories could be initialized.

Architecture

There is a new type to represent signals.

You can have multi-dimensional arrays of RAMs and dual-ported RAMs.

Interfaces have been extended to allow you to connect to input or output ports as well as pins. You can also define your own interface sorts and use them to link to blocks of external code (currently VHDL, Verilog or EDIF). Interface declarations have changed, and the previous style is deprecated. The names of ports within interfaces can no longer be omitted. For example, in Handel-C v2.1 you could declare '`interface bus_ts() abus(x,y)`' and the default port name `abus.in` would be used. You can no longer do this in version 3.0.

Pins no longer need to be assigned. You can omit the `data` specification to leave the pin assignment unconstrained. In this case, the place and route tools will assign the pins.

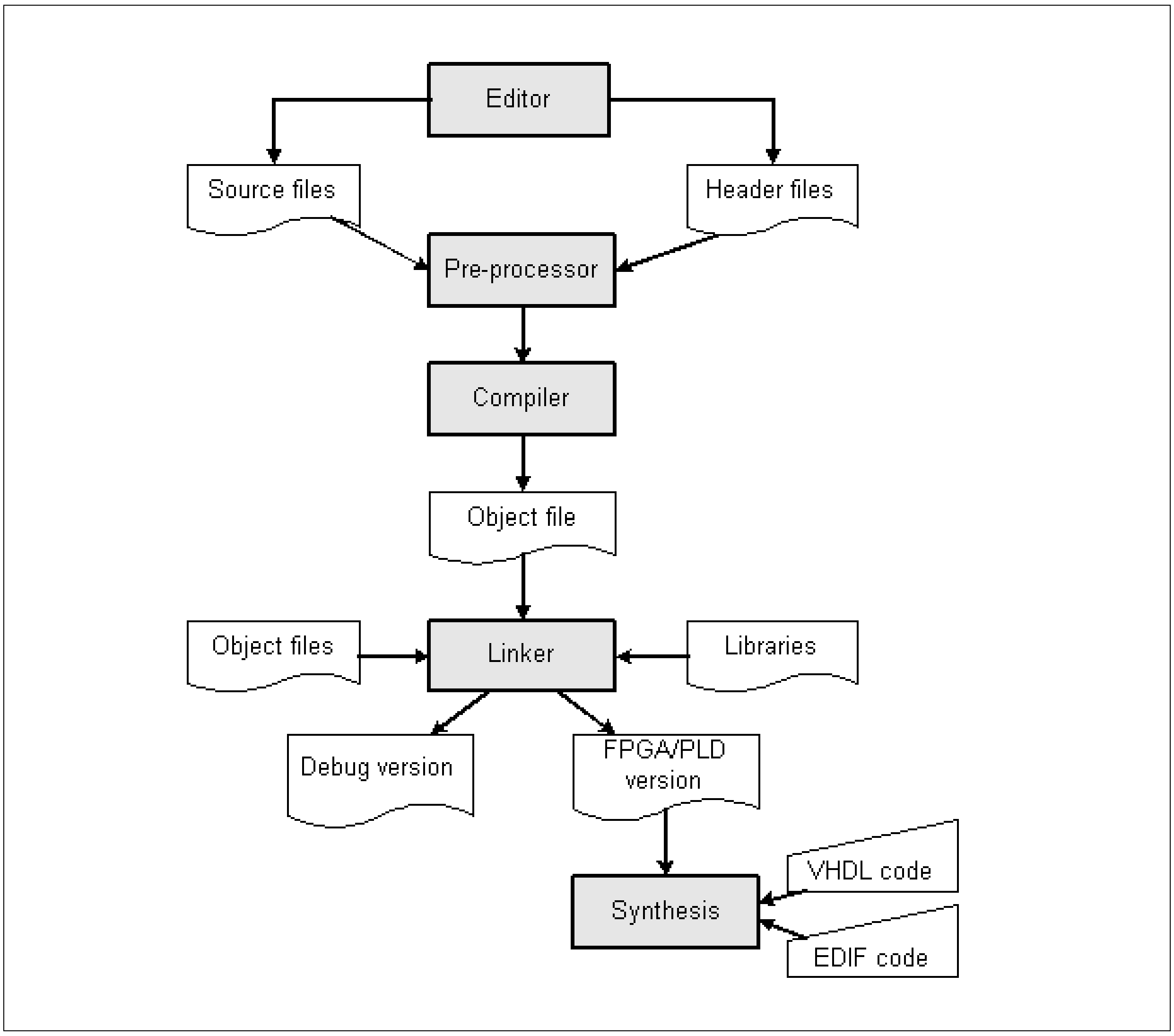
You can have multiple clocks within a system, and refer to the current clock by using `__clock`.

A.7 Linking multiple files to a single output module

The Handel-C compiler has a linker, allowing you to have multiple input files and links to library files.

Multiple files can now be linked into a single output module. These files can be pre-compiled core modules, libraries or header files. The `extern` keyword allows you to reference a function or variable in another file.

Appendix



LINKING MULTIPLE FILES TO A SINGLE OUTPUT MODULE

Linking is carried out during a build. You define the files to link by adding files to a project within the GUI.

A.8 Symbol scoping rules

The rules for scoping for `macro expr` and `macro proc` constructs have changed between version 2.1 and 3.0. Version 2.1 expands macros in the scope of their use. Version 3.0 expands macros in the scope of their declaration. This is consistent with C scoping rules. For example:

```
int x;        // Version 3.0 will use this x
macro expr a = x;

void main(void)
{
  int x;   // Version 2.1 will use this x

  y = a;
}
```

This may lead to undeclared identifier errors. For example, the following code is valid in version 2.1 but not in version 3.0:

```
macro proc a(x)
{
   b(x); //undefined in v 3.0
}
macro proc b(y)
{
   y++;
}

void main(void)
{
  int 4 z;
  a(z);
}
```

A.9 Using macro expressions in widths

Version 3.0 requires disambiguating brackets around macro expressions used in variable widths. For example:

```
int log2ceil(64) x;
```

must be rewritten as:

```
int (log2ceil(64)) x;
```

Appendix

A.10 New keywords clashing with variable names

Version 3 contains a number of new keywords which may clash with variable names in version 2.1 code. The list of new keywords is:

`assert`	`auto`	`const`	`continue`	`double`
`enum`	`extern`	`float`	`goto`	`ifselect`
`in`	`inline`	`let`	`mpram`	`register`
`releasesema`	`reset`	`return`	`sema`	`seq`
`signal`	`sizeof`	`static`	`struct`	`try`
`trysema`	`typedef`	`typeof`	`union`	`volatile`
`wom`				

A.11 Additional combinational loops

Version 2.1 uses approximations when checking for combinational loops in the generated logic. Version 3.0 does not use such approximations and may report unbreakable combinational loops in programs which compile with version 2.1.

A.12 Clock is required for simulation

Version 3.0 requires that a clock is specified when generating simulation output. A dummy clock such as '`set clock = external "P1";`' is valid.

A.13 Variable and interface name conflicts

You can no longer give a variable and an interface the same name. In version 2.1, this was possible because variable and interface names were specified in different namespaces. In version 3, these are stored in the same namespace. Giving a variable and an interface the same name causes a compiler error in version 3.

Example

```
int 3 x;
interface bus_in (int 3) x(); //not allowed in v.3
```

Index

>: Index

>: Index

>: Index

>: Index

>: Index

>: Index

>: Index

>: Index

>: Index

>: Index

>: Index

Customer Support at support@celoxica.com and +44 (0)1344 663649.

Celoxica Ltd. 20 Park Gate Milton Park Abingdon Oxfordshire OX14 4SH United Kingdom Tel: +44 (0) 1235 863 656 Fax: +44 (0) 1235 863 648	Celoxica, Inc 900 East Hamilton Avenue Campbell, CA 95008 USA Tel: +1 800 570 7004 Tel: +1 408 626 9070 Fax: +1 408 626 9079	Celoxica Japan KK YBP West Tower 11F 134 Godo-cho, Hodogaya-ku Yokohama 240-0005 Japan Tel: +81 (0) 45 331 0218 Fax: +81 (0) 45 331 0433	Celoxica Pte Ltd Unit #05-03 31 Int'l Business Park Singapore 609921 Tel: (65) 6896 4838 Fax: (65) 6566 9213