EMBEDDED
SOLUTIONS

Handel-C

Language Reference Manual

Xilinx, XBLOX and XACTStep are trademarks of Xilinx Corp.
Altera, MAX+PLUS II, FLEX, FLEX 10K and FLEX 8000 are trademarks and/or service marks of Altera Corp.
Microsoft and MS-DOS are registered trademarks and Windows, Windows 95 and Windows NT are trademarks of Microsoft Corporation.

This manual was written by Matthew Bowen

© Embedded Solutions Limited. All rights reserved
Version 2.1

Table of Contents

Conventions viii

1. OVERVIEW 1

1.1 Introduction 2

1.2 References 3

1.3 Chapter Summary 4

1.4 Basic Concepts 5
- 1.4.1 Handel-C Programs 5
- 1.4.2 Parallel Programs 5
- 1.4.3 Channel Communications 6
- 1.4.4 Scope and Variable Sharing 7

1.5 Design Flow Overview 9

2. LANGUAGE BASICS 11

2.1 Introduction 12

2.2 Program Structure 13
- 2.2.1 Using the Preprocessor 14
- 2.2.2 Comments 14

2.3 Declarations 15
- 2.3.1 Handel-C Values and Widths 15
- 2.3.2 Constants 15
- 2.3.3 Variables 16
- 2.3.4 Setting the Width of Integers 17
- 2.3.5 Variable Initialisation 18
- 2.3.6 Pointers 19
- 2.3.7 Arrays 19
- 2.3.8 Channels 19
- 2.3.9 Arrays of Channels 20
- 2.3.10 Internal RAMs and ROMs 21

2.4 Statements 23
- 2.4.1 Sequential and Parallel Execution 23
- 2.4.2 Assignments 24
- 2.4.3 Channel Communication 25
- 2.4.4 Conditional Execution 26
- 2.4.5 While Loops 27
- 2.4.6 Do ... While Loops 27
- 2.4.7 For Loops 28
- 2.4.8 Switch Statements 29
- 2.4.9 Prialt Statements 30
- 2.4.10 Break 31
- 2.4.11 Delay 32

2.5 Expressions **33**
2.5.1 Restrictions on RAMs and ROMs 34
2.5.2 Bit Manipulation Operators 35
2.5.3 Arithmetic Operators 37
2.5.4 Relational Operators 39
2.5.5 Relational Logical Operators 40
2.5.6 Bitwise Logical Operators 41
2.5.7 Conditional Operator 42
2.5.8 Casting of Expression Types 42
2.5.9 Compile Time Constant Expressions 44

2.6 Summary **45**
2.6.1 Type Summary 45
2.6.2 Statement Summary 46
2.6.3 Operator Summary 47

3. BASIC EXAMPLES **49**

3.1 Introduction **50**

3.2 The Accumulator Example **51**
3.2.1 Source Code Listing 51
3.2.2 Compiling and Simulating the Program 52
3.2.3 Detailed Explanation 53
3.2.4 Summary 54

3.3 The Divider Example **55**
3.3.1 Source Code Listing 55
3.3.2 Compiling and Simulating the Program 56
3.3.3 Detailed Explanation 57
3.3.4 Summary 57

3.4 The Queue Example **58**
3.4.1 Source Code Listing 58
3.4.2 Compiling and Simulating the Program 59
3.4.3 Detailed Explanation 60
3.4.4 Summary 60

3.5 The Microprocessor Example **61**
3.5.1 Source Code Listing 61
3.5.2 Compiling and Simulating the Program 63
3.5.3 Detailed Explanation 64
3.5.4 Summary 65

4. MACROS **67**

4.1 Introduction **68**

4.2 Macro Expressions **69**
4.2.1 Constant Macro Expressions 69
4.2.2 Parameterised Macro Expressions 69
4.2.3 The select Operator 70
4.2.4 Recursive Macro Expressions 71
4.2.5 Recursive Macro Expressions - A Larger Example 72

4.2.6 Shared Expressions 73
4.2.7 Using Recursion to Generate Shared Expressions 74
4.2.8 Restrictions on Shared Expressions 75

4.3 Macro Procedures 76

5. TIMING AND EFFICIENCY INFORMATION 79

5.1 Introduction 80

5.2 Clock Cycle Timing of Language Constructs 81
5.2.1 Statement Timing 81
5.2.2 Avoiding Combinatorial Loops 85
5.2.3 Parallel Access to Variables 87
5.2.4 Multiple Simultaneous Use of RAMs and ROMs 88
5.2.5 Detailed Timing Example 89

5.3 Time Efficiency of Handel-C Hardware 91
5.3.1 Reducing Logic Depth 91
5.3.2 Pipelining 94

6. TARGETTING HARDWARE 99

6.1 Introduction 100

6.2 Interfacing with the Simulator 101
6.2.1 Single Word Transfers 101
6.2.2 Block Data Transfers 102

6.3 Targeting FPGA Devices 104
6.3.1 Targeting Specific Devices 104
6.3.2 Locating the Clock 105

6.4 Use of RAMs and ROMs with Handel-C 107
6.4.1 Using On-Chip RAMs in Xilinx Devices 107
6.4.2 Using On-Chip RAMs in Altera Devices 108
6.4.3 Using External RAMs 109
6.4.4 Using External ROMs 114
6.4.5 Using Other RAMs 114

6.5 Interfacing With External Hardware 115
6.5.1 Off-chip Interfaces 115
6.5.2 Reading from External Pins 116
6.5.3 Latched Reading from External Pins 116
6.5.4 Clocked Reading from External Pins 117
6.5.5 Writing to External Pins 117
6.5.6 Bi-directional Data Transfer 118
6.5.7 Bi-directional Data Transfer with Latched Input 119
6.5.8 Bi-directional Data Transfer with Clocked Input 120
6.5.9 Buses and the Simulator 121
6.5.10 Timing Considerations of Buses 121
6.5.11 Metastability 124

6.6 Object Specifications 126

6.6.1 The show Specification ... 127
6.6.2 The base Specification ... 127
6.6.3 The infile and outfile Specifications ... 127
6.6.4 The warn Specification ... 128
6.6.5 The speed Specification ... 128
6.6.6 The pull Specification ... 128
6.6.7 The offchip Specification ... 129
6.6.8 The wegate Specification ... 129
6.6.9 The westart and welength Specifications ... 129
6.6.10 Specifying Pinouts ... 130

6.7 An Example Hardware Interface ... 131

7. STANDARD MACRO EXPRESSIONS ... 135

7.1 Introduction ... 136

7.2 Constant Definitions ... 137

7.3 Bit Manipulation Macros ... 138
7.3.1 adjs ... 138
7.3.2 adju ... 139
7.3.3 copy ... 140
7.3.4 lmo ... 141
7.3.5 lmz ... 142
7.3.6 population ... 143
7.3.7 rmo ... 144
7.3.8 rmz ... 145
7.3.9 top ... 146

7.4 Arithmetic Macros ... 147
7.4.1 abs ... 147
7.4.2 addsat ... 148
7.4.3 decode ... 149
7.4.4 div ... 150
7.4.5 exp2 ... 151
7.4.6 incwrap ... 152
7.4.7 log2ceil ... 153
7.4.8 log2floor ... 154
7.4.9 mod ... 155
7.4.10 sign ... 156
7.4.11 subsat ... 157

8. PORTING C TO HANDEL-C ... 159

8.1 Introduction ... 160

8.2 General Porting Issues ... 161

8.3 Comparison Between Conventional C and Handel-C ... 162
8.3.1 Types, Type Operators and Objects ... 162
8.3.2 Statements ... 162
8.3.3 Expressions ... 163

8.4 Porting Example - An Edge Detector 164
8.4.1 The Original Program 164
8.4.2 The Target Architecture 166
8.4.3 Mapping to the Target Architecture 166
8.4.4 First Attempt Handel-C Program 167
8.4.5 First Optimisations of the Handel-C Program 169
8.4.6 Adding Fine Grain Parallelism 170
8.4.7 Further Fine Grain Parallelism 174
8.4.8 Adding the Hardware Interfaces 176

8.5 Summary 180

9. COMPLETE LANGUAGE SYNTAX 183

9.1 Introduction 184

9.2 Keywords 185

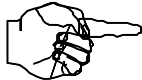
9.3 Complete Language Syntax 186
9.3.1 Identifiers 186
9.3.2 Integer Literals 186
9.3.3 Strings 186
9.3.4 Types 187
9.3.5 Hardware Control 188
9.3.6 Declarations 189
9.3.7 Variable Declarations 189
9.3.8 Channel Declarations 190
9.3.9 Interface Declarations 191
9.3.10 RAM and ROM Declarations 192
9.3.11 Object Specifications 193
9.3.12 Macro Expression Declarations 193
9.3.13 Shared Expression Declarations 194
9.3.14 Macro Procedure Declarations 194
9.3.15 Expressions 194
9.3.16 Statements 196
9.3.17 Program 197

INDEX 199

Conventions

A number of conventions will be used throughout this document. These conventions are detailed below.

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



Information Note. These messages appear to 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*

1. Overview

1.1 Introduction

Handel-C is a programming language designed to enable the compilation of programs into synchronous hardware. Handel-C is not a hardware description language though; rather it is a programming language aimed at compiling high level algorithms directly into gate level hardware.

The Handel-C syntax is based on that of conventional C so programmers familiar with conventional C will recognise almost all the constructs in the Handel-C language. Readers unfamiliar with conventional C should first read one of the standard texts on that language (see reference 1).

This document describes the Handel-C language. The Handel-C compiler and some detailed examples of its usage are described in the Handel-C Compiler Reference Manual.

1.2 References

This document contains references to the following documents.

1. The C Programming Language
   Kernighan, B. and Ritchie, D.
   Prentice-Hall, 1988

2. The Programmable Logic Data Book
   Xilinx 1996

3. Altera Databook
   Altera 1996

4. Handel-C Preprocessor Reference Manual
   Embedded Solutions Limited 1998

5. Handel-C Compiler Reference Manual
   Embedded Solutions Limited 1998

1.3 Chapter Summary

The following chapters are designed to lead the reader through the Handel-C language from the simple expressions and statements up to full programs containing hardware interfaces.

Chapter 1 (this chapter) covers some fundamental concepts of the language including the similarity with conventional C and some additional features such as parallelism and channel communication.

Chapter 2 covers the language data types, expressions and statements in detail.

Chapter 3 presents some examples illustrating the basic data types, expressions and statements. It also introduces the hardware simulator to test Handel-C programs.

Chapter 4 covers macro expressions and procedures which allow complex hardware to be used as 'subroutines' to the main program.

Chapter 5 details clock cycle timing of Handel-C programs and how to improve the performance of Handel-C code.

Chapter 6 details how to connect Handel-C programs to external hardware such as RAMs, ROMs, custom and standard buses.

Chapter 7 details the set of standard macros supplied with the Handel-C compiler.

Chapter 8 presents a detailed example of porting a conventional C program to Handel-C.

Chapter 9 is a reference for the complete Handel-C language syntax.

1.4 Basic Concepts

This section deals with some of the basics behind the Handel-C language. Sequential programs can be written in Handel-C just as in conventional C but to gain the most benefit in performance from the target hardware its inherent parallelism must be exploited. Handel-C therefore includes parallel constructs and these may be new to some readers. Readers familiar with conventional C should recognise virtually all the other language features.

As with conventional high level languages, Handel-C is designed to allow you to express your algorithm without worrying too much about exactly 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.

1.4.1 Handel-C Programs

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

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

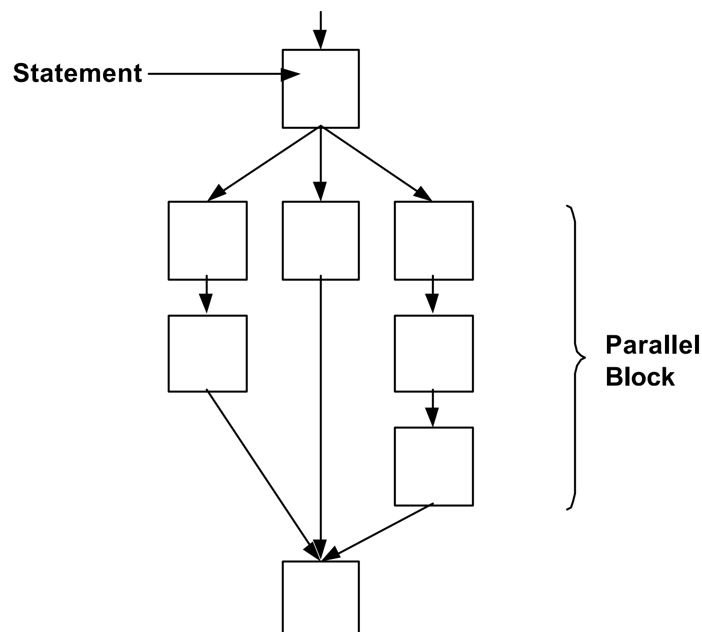
It is important to note that the hardware design that Handel-C produces is exactly the hardware specified in the source program. There is no intermediate 'interpreting' layer as exists in assembly language when targeting general purpose microprocessors. The logic gates that make up the final Handel-C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

1.4.2 Parallel Programs

Because the target of the Handel-C compiler is low-level hardware, massive performance benefits are made possible by the use of parallelism. Although Handel-C is inherently sequential, it is possible (and indeed essential for efficient programs) to instruct the compiler to build hardware to execute statements in parallel.

Handel-C parallelism is true parallelism - it is not the time-sliced parallelism familiar from general purpose computers. In other words, 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 in 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. This is illustrated in the diagram below.

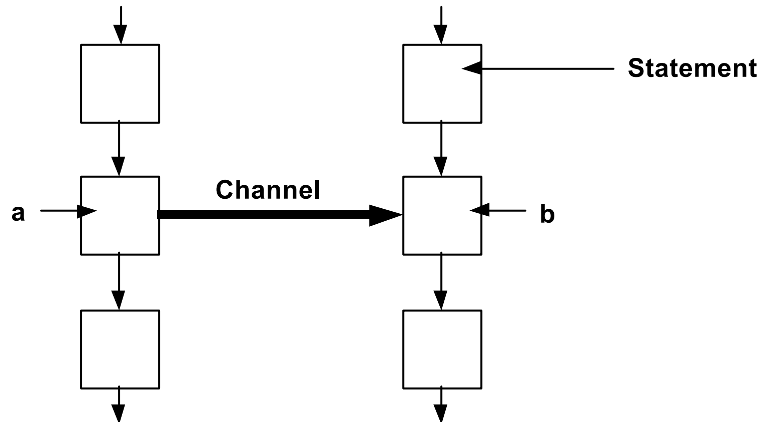


This diagram illustrates the branching and re-joining of the execution flow. The left hand and middle branches must wait to ensure that all branches have completed before the instruction following the parallel construct can be executed.

1.4.3 Channel Communications

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 synchronisation 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.

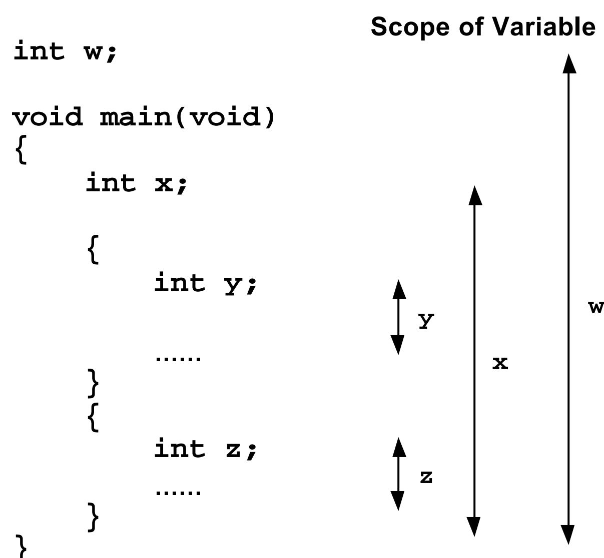
The link between parallel branches can be seen from the diagram below.



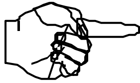
Here, the channel is shown transferring data from the left branch to the right branch. If the left branch reaches point a before the right branch reaches point b, the left branch waits at point a until the right branch reaches point b.

1.4.4 Scope and Variable Sharing

The scope of declarations is, as in conventional C, based around code blocks. A code block is denoted with {...} brackets. Basically, this means that global variables must be declared outside all code blocks and that an identifier is in scope within a code block and any sub-blocks of that block. The scope of variables is illustrated below:



Since parallel constructs are simply code blocks (see chapter 2), variables can be in scope in two parallel branches of code. This can lead to resource conflicts if the variable is accessed simultaneously by more than one of the branches. Handel-C syntax states that a single variable must not be accessed by more than one parallel branch. Unfortunately, as is shown in chapter 5 on timing and efficiency, this precludes the use of some powerful operations so in reality this rule is relaxed. See chapter 5 for further details.

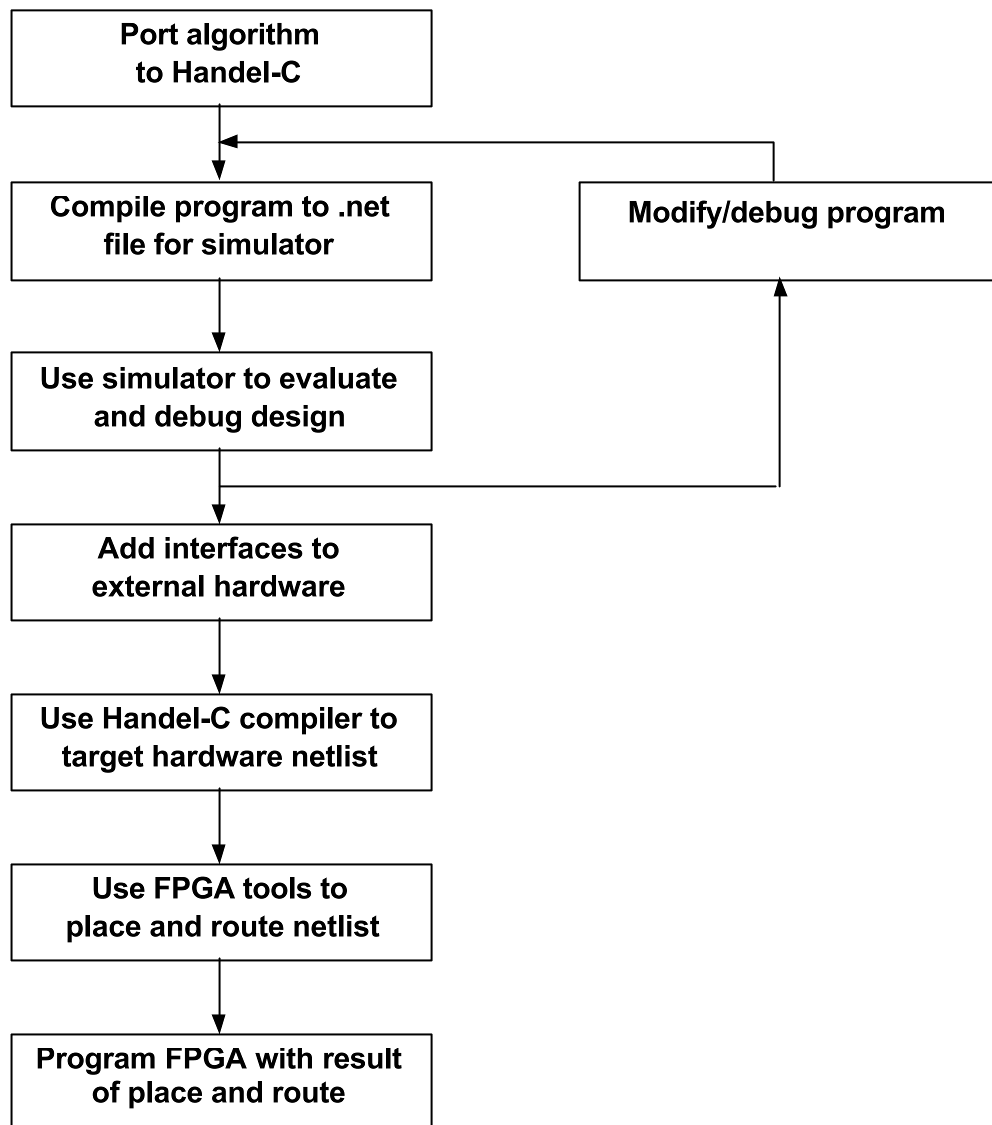


A single variable should not normally be accessed in more than one parallel branch.

1.5 Design Flow Overview

This section will give an overview of the design flow for a Handel-C program. For a detailed look at how to design hardware using Handel-C, consult the Handel-C compiler reference manual.

The basic design flow is presented below:



2. Language Basics

2.1 Introduction

This chapter deals with the basics of producing Handel-C programs including their overall structure, declarations, expressions and statements. The next chapter details a number of examples that use only the basic language constructs described here. Later chapters discuss more complex language constructs and interfaces with the outside world.

2.2 Program Structure

Just like 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 to inform the compiler of 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 variables and expressions similar to conventional C although there are a number of restrictions to these where operations are not appropriate to hardware implementation and some extensions where hardware implementation allows additional functionality.

Unlike conventional C, Handel-C programs can also have statements that execute in parallel. This feature is crucial when targeting hardware because parallelism is the main way to increase performance by using hardware.

The overall program structure is as follows:

```
Global Declarations

void main(void)
{
        Local Declarations

        Body Code
}
```

Note that 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.

2.2.1 Using the Preprocessor

As with conventional C, the Handel-C source code is first 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. See the Handel-C Preprocessor Reference Manual for further details.

Handel-C also provides additional support for macros which cannot be achieved by using the preprocessor. See chapter 4 on macros for further details.

2.2.2 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
```

2.3 Declarations

This section details the types of declarations that can be made. It is important to understand how the type system differs from that of conventional C so this is also dealt with here also.

2.3.1 Handel-C Values and Widths

A crucial difference between Handel-C and conventional C is its 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 values of any number of bits.

In addition, Handel-C has 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.

When writing programs in Handel-C, care should be taken that data paths are no wider than necessary to minimise 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 only just wide enough to contain the largest value.

2.3.2 Constants

Constants may be used within expressions where required. 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 */
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. See section 2.5.8 for further details of casting.

2.3.3 Variables

There is only one fundamental type for variables: **int**. In addition, the **int** type may be qualified with the **unsigned** keyword to indicate that the variable only contains positive integers. 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 positive only 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;
```

Note that 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.

It is also possible to 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**.

The Handel-C compiler can sometimes 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 has been added to indicate that the compiler should attempt 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 while 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.

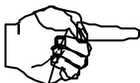
If the compiler cannot infer all the undefined widths, it will generate errors detailing which widths it could not infer.

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

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

The widths assumed for each of these types is as follows:

Type	Width
char	8 bits
short	16 bits
long	32 bits



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

2.3.4 Setting the Width of Integers

As mentioned above, the following line will declare an integer of undefined width:

```
int undefined x;
```

The compiler attempts to infer the width of the variable from its usage. By default, the following declaration will also declare a variable of undefined width and is directly equivalent to the example above:

```
int x;
```

In other words, by default the **undefined** keyword is optional. Handel-C provides an extension to allow you to override this behaviour to ease porting from conventional C. 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** although any width may be used in the **set intwidth** instruction.

It is possible to set integers to be a particular width in this way and still declare variables that must have their width inferred by using the **undefined** keyword. For example:

```
set intwidth = 27;

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.

2.3.5 Variable Initialisation

Global variables (i.e. those declared outside the **main()** function) may be initialised with their declaration. For example:

```
int 15 x = 1234;
```

Note that variables declared within the **main()** function may not be initialised in this way. Rather, you should 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;
}
```

2.3.6 Pointers

Pointers do not exist in Handel-C.

2.3.7 Arrays

It is possible to 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;
```

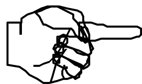
Note that as in conventional C, the first variable has an index of 0 and the last has an index of n-1 where n is the total number of variables in the array.

It is also possible to declare multi-dimensional arrays of variables. For example:

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

This declares 4x5x6 = 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];
```



When accessing an array, the index must be a compile time constant. If you require random access of an array of values, consider using a RAM or ROM as described below.

2.3.8 Channels

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

Channels are declared with the `chan` keyword. For example:

```
chan int 7 link;
```

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

```
set intwidth = undefined;

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

The compiler generates warnings if any single process uses a channel for both input and output or if more than one parallel process uses the same channel for either input or output.

See section 2.4.3 for details of how to communicate via channels.

2.3.9 Arrays of Channels

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

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

This is equivalent to declaring 6 channels each of which is 5 bits wide. Channels can then be accessed by specifying the index of the channel required similar to the way that arrays of variables are de-referenced. For example:

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

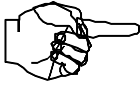
It is also possible to declare multi-dimensional arrays of channels. For example:

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

This declares 4x5x6 = 120 channels each of which is 6 bits wide. Accessing the channels is as expected from conventional C. For example:

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

See section 2.4.3 for details of how to communicate via channels.



As with arrays of variables, the index of an array of channels must be a compile time constant.

2.3.10 Internal RAMs and ROMs

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

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

This example constructs a RAM consisting of 43 entries each of which is 6 bits wide and a ROM consisting of 4 entries each of which is 16 bits wide. The ROM is initialised with the constants given in the following list in much the same way as an array would be initialised 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

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 much the same way that arrays are accessed in conventional C. For example:

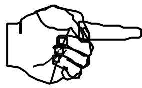
```
ram int 6 b[56];

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

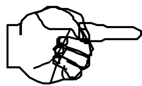
the last has an index of n-1 where n is the total number of entries in the RAM.

Note that 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 writes in a clock cycle as required. RAMs, however, are normally more efficient to implement in terms of hardware resources than arrays and also allow a non-constant index. 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 wish to have random access to the elements.



When accessing a RAM or ROM, the index need not be a compile time constant.

Writing to internal RAMs can only be done in this way on Altera or Xilinx devices with synchronous on-chip RAMs. This includes Altera Flex 10K, Xilinx 4000E, 4000EX, 4000L, 4000XL and 4000XV series devices. See chapter 6 for further details of RAMs and ROMs in Hanel-C.



RAMs and ROMs may only have one entry accessed in any clock cycle. The Handel-C compiler generates warnings if this condition is violated. This restriction is discussed in more detail in section 2.5.1.

2.4 Statements

As with conventional C, the execution flow of a Handel-C program is expressed as a series of statements such as assignment, conditional execution and iteration. Handel-C includes most of the statements from conventional C and these are detailed below.

2.4.1 Sequential and Parallel Execution

Handel-C implicitly executes instructions sequentially but when targeting hardware it is extremely important to make as much use as possible of parallelism. For this reason, Handel-C also has a parallel composition keyword to allow statements in a block to be executed in parallel.

The following example executes three assignments sequentially:

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

In contrast, the following example executes all three assignments in parallel and in the same clock cycle:

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

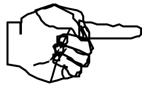
It should be noted that the second example executes all assignments literally in parallel - this is not the time-sliced pseudo parallelism of a conventional microprocessor implementation but rather three specific pieces of hardware built just to perform these three assignments.

Detailed timing analysis will be dealt with in chapter 5 but for now it is enough to state that the first example executes in 3 clock cycles while the second generates a similar quantity of hardware but executes in 1 clock cycle. Therefore, it is obvious that parallelism is a very important construct for targeting hardware.

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

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

In this example, the first branch of the parallel statement executes the assignment of **x** while the second branch sequentially executes the assignments of **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.



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

2.4.2 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 described in section 2.5.

Handel-C also provides a number of short cut assignment statements. Note that these cannot be used in expressions as they can in conventional C but only in stand alone statements. See section 2.5 for further details. These short cuts are:

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 **<<=** *Constant***;**	*Variable = Variable* << *Constant;*
Variable **>>=** *Constant***;**	*Variable = Variable* >> *Constant;*
Variable **&=** *Expression***;**	*Variable = Variable* **&** *Expression;*
Variable **|=** *Expression***;**	*Variable = Variable* **|** *Expression;*
Variable **^=** *Expression***;**	*Variable = Variable* **^** *Expression;*

See section 2.5 for further details of each of these expansions.

2.4.3 Channel Communication

Reading from a channel is done as follows:

Channel **?** *Variable***;**

This assigns the value read from the channel to the variable. The variable may also be an array element or RAM element. Writing to a channel is as follows:

Channel **!** *Expression*

This writes the value of the expression to the channel. *Expression* may be any expression described in section 2.5. In both these statements, *Channel* may also be an entry in an array of channels.

In both cases the width and type (signed or unsigned) of the channel must be the same as the width and type of the variable or expression although the compiler can infer widths if the **undefined** keyword was used when declaring the channel or variable(s).

No two statements may simultaneously write to a single channel and no two statements may simultaneously read from a single channel . For example, the following piece of code is illegal:

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

Here, an attempt is made 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;
}
```

2.4.4 Conditional Execution

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;
```

Here, and throughout the rest of this document, *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. As will be seen in sections 2.5.4 and 2.5.5, the relational logical operators return values to match this 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`.

2.4.5 While Loops

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

```
while (Expression)
    Statement
```

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. Again, *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`).

For details of conditional expressions, see sections 2.5.4 and 2.5.5.

2.4.6 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. Again, *Statement* may be replaced with a block of statements. For example:

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

2.4.7 For Loops

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

```
for (Initialisation ; 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.

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

Is directly equivalent to:

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

Each of the initialisation, test and iteration statements are optional and may be omitted if not required. 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 initialisation 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 (see section 2.5 below) the initialisation 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 initialisation and iteration statements can also 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;
}
```

2.4.8 Switch Statements

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 guarded by the matching constant is executed until a **break** statement is encountered.

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

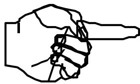
Each of the *Statement* lines above may be replaced with a block of statements by enclosing the block in {...} brackets.

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.

2.4.9 Prialt Statements

Handel-C provides a **`prialt`** statement not found in conventional C for selective channel communication.

```
prialt
{
    case CommsStatement:
        Statement
        break;
    ......
    default:
        Statement
        break;
}
```

The **`prialt`** statement can be used to select between communication on a number of channels depending on the

readiness of the other end of the channel communication. *CommsStatement* must be one of the following forms:

Channel **?** *Variable*
Channel **!** *Expression*

The first communication statement in the list of cases which becomes ready to transfer data will execute. The statements up to the next **break** statement will then be executed.

In a **prialt** statement with no **default** case, execution will halt until one of the channels becomes ready to communicate. In a **prialt** statement with a **default** case, if none of the channels is ready to communicate immediately then the **default** branch statements will execute and the **prialt** statement will terminate.

The **prialt** construct does not allow the same channel to be listed twice in its cases and fallthrough of cases is prohibited. This means that each **case** must be paired with its own **break** statement.

2.4.10 Break

Handel-C provides the normal C **break** statement both for terminating loops and separation of **case** branches in **switch** and **prialt** statements.

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
```

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
```

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
```

2.4.11 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 combinatorial logic cycles. See chapter 5 on timing and efficiency for details of this.

2.5 Expressions

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 do 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 combinatorial delays in the hardware. The clock period for the entire hardware program is limited by the longest such evaluation in the whole program. See chapter 5 for more details on timing and efficiency considerations.

As a result of expressions not being allowed to take any clock cycles, expressions with side effects are not permitted in Handel-C. For example;

```
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.

Note that even the longest and most complex C expression with many side effects can be written in terms of a larger number of simpler expressions. 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;
```

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

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

This example is directly equivalent to:

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

2.5.1 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 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 illegal because the assignment attempts to read from the third element of **x** in the same cycle as it writes to the first element. The compiler generates errors for this form of statement.

The following code is also disallowed:

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

if (x[0]==0)
    x[1] = 1;
```

This is 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 - see chapter 5 for details of the timing of Handel-C programs.

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. RAMs and ROMs also have the advantage of random access - the index need not be a constant as is the case with arrays of variables.

2.5.2 Bit Manipulation Operators

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

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

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

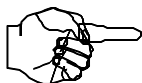
When shifting unsigned values, the right shift pads the upper bits with zeros. When right shifting signed values, the upper bits are copies of the top bit of the original value. Thus, a shift right by 1 divides the value by 2 and preserves the sign. For example:

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

x = 192;
y = -8;

x = x >> 1;
y = y >> 1;
```

This results in **`x`** being set to 96 and **`y`** being set to -4.



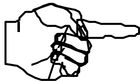
The value following the shift operator must be a compile time constant.

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. For example:

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

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

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



The value following the take and drop operators must be a compile time constant.

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;
```

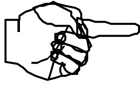
This results in `x` being set to 0xC7. The left operand of the concatenation operator forms the most significant bits of the result.

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 index value and range values for bit selection must be compile time constants.

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

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

y = w[10][4:2];
z = x[2][0];
```

Here, the 10 is the entry in the RAM and the `4:2` selects three bits from the middle of the value in the RAM. Similarly, `z` is set to the least significant bit in the `x[2]` variable.

The `width()` operator returns the width of an expression which 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 are taken from `y` to match the width of `x`.

2.5.3 Arithmetic Operators

The following arithmetic operators are provided in Handel-C:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication

No division operator is provided because the hardware required for the divide would be prohibitively large. See chapters 3 and 7 for examples of how to write division routines in Handel-C.

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. See section 2.5.8 for details of changing types of expressions.

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.

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 form 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.

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. See section 2.6.3 for details of operator precedence.

2.5.4 Relational Operators

The following relational operators are provided in Handel-C:

Operator	Meaning
==	Equal
!=	Not equal
<	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 the following conventional C code is invalid:

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

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

Rather, you should write:

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

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. For example, the above code can be rewritten as:

```
unsigned 8 x;
int 8 y;

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

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)
{
    ......
}
```

2.5.5 Relational Logical Operators

The following relational logical operators are provided in Handel-C:

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

These operators are provided to combine conditions as in conventional C. Each operator takes two 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. For 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. As in conventional C, the condition of the **if** statement is true if **x** is not equal to 0 or **y** is greater than **z**. This feature allows some familiar looking conventional C constructs. For example:

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

2.5.6 Bitwise Logical Operators

The following bitwise logical operators are provided in Handel-C:

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

These operators perform bitwise logical operations on values. Each operand must be of the same width and the resulting value is also of the same 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.

2.5.7 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 second expression. If the first expression is false, the whole expression evaluates to 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 = ((y > z) ? y : z) + 4;
```

2.5.8 Casting of Expression Types

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.

The solution adopted by Handel-C is not to allow automatic conversions between signed and unsigned values to avoid this confusion. 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. 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 just the 4 bits extracted from **y**. It is also possible to cast to a type of undefined width. For example:

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

x = (int undefined)y;
```

Here, the compiler will infer that **y** must be 4 bits wide.

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
```

Instead, the conversion should be done by hand as follows:

```
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.

Again, this is to ensure that the programmer is aware of such conversions. To illustrate why this is important, consider the following example:

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

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

Here, there are two equally viable routes that the Handel-C compiler could take. 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 for their program and to write the expression accordingly.

2.5.9 Compile Time Constant Expressions

In addition to all the operators listed in the previous sections, Handel-C provides two extra operators for expressions consisting only of compile time constants. These are:

Operator	Meaning
/	Division
%	Modulo arithmetic

Handel-C does not build hardware to evaluate these expressions but the compiler can calculate constant expressions with these operators. For example:

```
unsigned int 16 x;
unsigned int (width(x)/2) y;

y = 63552 % 256;
```

The compiler generates an error if one or both of the operands is not a compile time constant.

2.6 Summary

This section summarises the previous sections by listing all the Handel-C types, statements and operators. For a full language description, refer to chapter 9

2.6.1 Type Summary

The following table lists all types that may be associated with a variable.

Type	Width
`char`	8 bits
`unsigned char`	8 bits
`short`	16 bits
`unsigned short`	16 bits
`long`	32 bits
`unsigned long`	32 bits
`int`	See note 1
`unsigned int`	See note 1
`int` *n*	*n* bits
`unsigned int` *n*	*n* bits
`int undefined`	Compiler infers width
`unsigned int undefined`	Compiler infers width
`unsigned`	See note 1
`unsigned` *n*	*n* bits
`unsigned undefined`	Compiler infers width

Note 1: Width will be inferred by compiler unless the '**set intwidth =** *n*' command appears before the declaration. See Section 2.3.4 for details.

The following table lists all prefixes to the above types for different object types.

Prefix	Object
`chan`	Channel
`chanin`	Simulator channel (see chapter 6)
`chanout`	Simulator channel (see chapter 6)
`ram`	Internal or external RAM
`rom`	Internal or external ROM

2.6.2 Statement Summary

The following table lists all statements in the Handel-C language.

Note that 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.

Statement	Meaning
par {...}	Parallel composition
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 <<= *Constant*;	Shift left and assign
Variable >>= *Constant*;	Shift right and assign
Variable &= *Expression*;	AND and assign
Variable |= *Expression*;	OR and assign
Variable ^= *Expression*;	XOR and assign
Channel ? *Variable*;	Channel input
Channel ! *Expression*;	Channel output
if (*Expression*) {...} **else** {...}	Conditional execution
while (*Expression*) {...}	Iteration
do {...} **while** (*Expression*);	Iteration
for (*Init* ; *Test* ; *Iter*) {...}	Iteration
break;	Loop termination
switch (*Expression*) {...}	Selection
prialt {...}	Channel alternation
delay;	Single cycle delay

Note: Internal RAM and ROM elements and array elements are included in the set of variables in this table.

2.6.3 Operator Summary

The following table lists all operators in the Handel-C language.

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>Array [Constant]
Expression [Constant]
Expression [Constant:Constant]
RAM [Expression]</td><td>Array subscripting
Bit selection
Bit range extraction
RAM/ROM subscript</td></tr>
<tr><td>! Expression
~ Expression</td><td>Logical NOT
Bitwise NOT</td></tr>
<tr><td>- Expression</td><td>Unary minus</td></tr>
<tr><td>(Type)Expression</td><td>Type casting</td></tr>
<tr><td>Expression <- Constant
Expression \\ Constant</td><td>Take LSBs
Drop LSBs</td></tr>
<tr><td>Expression * Expression
Constant / Constant
Constant % Constant</td><td>Multiplication
Division
Modulo arithmetic</td></tr>
<tr><td>Expression + Expression
Expression - Expression</td><td>Addition
Subtraction</td></tr>
<tr><td>Expression << Constant
Expression >> Constant</td><td>Shift left
Shift right</td></tr>
<tr><td>Expression @ Expression</td><td>Concatenation</td></tr>
<tr><td>Expression < Expression
Expression > Expression
Expression <= Expression
Expression >= Expression</td><td>Less than
Greater than
Less than or equal
Greater than or equal</td></tr>
<tr><td>Expression == Expression
Expression != Expression</td><td>Equal
Not equal</td></tr>
<tr><td>Expression & Expression</td><td>Bitwise AND</td></tr>
<tr><td>Expression ^ Expression</td><td>Bitwise XOR</td></tr>
<tr><td>Expression | Expression</td><td>Bitwise OR</td></tr>
<tr><td>Expression && Expression</td><td>Logical AND</td></tr>
<tr><td>Expression || Expression</td><td>Logical OR</td></tr>
<tr><td>Expression ? Expr : Expr</td><td>Conditional selection</td></tr>
<tr><td>width(Expression)</td><td>Width of expression</td></tr>
<tr><td>select(Constant, Expr, Expr)</td><td>Compile-time selection</td></tr>
</table>

Note: Here, *Constant* means a compile time constant. The `select` construct is described in chapter 4.

In this table, 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.

3. Basic Examples

3.1 Introduction

In this chapter the basic language features discussed in the previous chapter are used in various examples designed to illustrate the usage of the language.

The first example simply takes a number of values from the user and calculates the sum of those values.

The second example divides one integer by another and calculates the integer result.

The third example illustrates the use of channels by implementing a queue.

Finally, a complete simple microprocessor example is presented which illustrates that complex hardware can be generated from very simple Handel-C programs.

The examples in this chapter all use the simulator provided with the Handel-C compiler to execute the programs and so do not require any additional hardware platform.

All the examples in this chapter are provided on the disk with the compiler. Refer to the Handel-C Compiler Reference Manual for details of the directory structure.

3.2 The Accumulator Example

This program takes a number of values from the user and calculates the sum of those values. It illustrates the basics of producing a Handel-C program and demonstrates the use of the simulator to help get programs correct before implementation in hardware.

3.2.1 Source Code Listing

The complete Handel-C listing is shown below. This program is also provided on the disk with the Handel-C compiler.

```
void main(void)
{
    unsigned int 16 sum;
    unsigned int 8 data;
    chanin input;
    chanout output;

    sum = 0;
    do
    {
        input ? data;
        sum = sum + (0 @ data);
    } while (data!=0);

    output ! sum;
}
```

3.2.2 Compiling and Simulating the Program

Compilation and simulation of the program is performed by typing the following at the command prompt:

```
> handelc -s sum.c
```

This will generate some lines something like this:

```
 Compiled :     0 gates,     0 inverters,    4 latches,   26 others
Optimised :     0 gates,     0 inverters,    4 latches,   23 others
 Expanded :    75 gates,    19 inverters,   30 latches,    8 others
Optimised :    26 gates,     4 inverters,   30 latches,    8 others
```

This is the result of the compilation phase and details the number of hardware gates required to implement the program. The simulator then starts immediately and prompts you for some input. Continue to type in numbers until you wish to quit when you should enter zero. The simulator then asks you whether you are ready for the output from the program. When you respond with a 'y', the simulator returns the sum of the numbers that you typed in. An example session might look something like this:

```
 0: sum=0 data=0
 1: sum=0 data=0
 1: Input to `input' ? 1
 2: sum=0 data=1
 3: sum=1 data=1
 3: Input to `input' ? 2
 4: sum=1 data=2
 5: sum=3 data=2
 5: Input to `input' ? 3
 6: sum=3 data=3
 7: sum=6 data=3
 7: Input to `input' ? 4
 8: sum=6 data=4
 9: sum=10 data=4
 9: Input to `input' ? 0
10: sum=10 data=0
11: sum=10 data=0
11: Ready to accept output from `output' ? (y/n) y
11: Output from channel `output' = 10
12: sum=10 data=0
```

3.2.3 Detailed Explanation

At this point, a detailed explanation of the example is required. Looking at the program the first line simply declares the **`main()`** function as you would expect in conventional C. Note that the **`main()`** function in Handel-C takes no parameters and returns no value as you would expect in a hardware implementation - the compiler does not know where data should come from and where it should go. If parameters and return values are required for your program you should explicitly transfer the values to your surrounding environment. See chapter 6 for how to interface your Handel-C programs with the outside world.

The next couple of lines of the program declare two variables, **`sum`** and **`data`**. Both variables are unsigned integers - **`sum`** is 16 bits wide and **`data`** is 8 bits wide.

Next, two channels are declared, **`input`** and **`output`**. Note that the channels are declared with the **`chanin`** and **`chanout`** keywords rather than the normal **`chan`** keyword. This tells the compiler to connect the input or output of the channel to the simulator so that you can type values to be passed into the channel and see the outputs from channels on the screen. Interfacing with the simulator is discussed in more detail in chapter 6 but for now it is enough to see how to pass single words between your program and the simulator.

Note that the channels have been declared without a specific width or type. The Handel-C compiler creates channels of the correct width and type for the usage of the channels. In this case, the compiler can infer that the input channel must be 8 bits wide and unsigned from the input statement:

```
input ? data;
```

Since the variable **`data`** is explicitly defined to be 8 bits wide, the input channel must also be 8 bits wide. If the **`data`** variable declaration were modified to make it 10 bits wide, the **`input`** channel would then become 10 bits wide also. Thus, you can see that you need not explicitly declare the width of all items in the program although you may if you wish.

Using similar arguments, the **`output`** channel in this example is inferred to be 16 bits wide.

The line following the channel declarations initialises the variable **sum** to zero. Note that there is no width associated with the constant 0. The compiler can infer that its width must be equal to the width of the **sum** variable - in this case 16 bits.

The main part of the program is the **do**...**while** loop. This construct should be familiar from conventional C. The loop is executed until the **data** variable becomes 0. Since the loop is a **do**...**while** construct rather than a **while** loop, it is executed at least once.

The first statement in the loop reads a value from the **input** channel into the variable **data**. As discussed earlier, the **input** channel is connected to the simulator because it was declared with the **chanin** keyword so whenever this statement is executed, the user is prompted for some input to the channel.

The second statement in the loop adds the value read from the **input** channel to the running total stored in the variable **sum**. The variables **data** and **sum** are not of the same width so the **data** variable must be padded with zeros in its most significant bits before being added to the previous value in **sum**. The constant zero does not have a width associated with it - the compiler can infer that it must be 8 bits wide from the fact that **sum** is 16 bits and **data** is 8 bits wide.

Finally, the last statement simply outputs the sum on the **output** channel. Since the output channel was declared with the **chanout** keyword, this causes the simulator to wait for the user to become ready before displaying the result on the screen.

3.2.4 Summary

Although this example is extremely simple it illustrates most of the important points of writing programs in Handel-C. It covers variables of differing widths, channels, connecting to the simulator and basic statements and expressions. The next examples build on this to introduce more complex features of the language.

3.3 The Divider Example

This program does simple integer division using the long division method. The program is an infinite loop so multiple runs can be performed without recompiling. It inputs two values, a and b, and returns the integer part of a/b. Although not very useful in itself, the body of the program could be run in parallel with some other task, giving Handel-C programs access to division which is not provided as one of the standard arithmetic operators.

3.3.1 Source Code Listing

The complete Handel-C listing is shown below. This program is also provided on the disk with the Handel-C compiler.

```
#define DATA_WIDTH 16
void main(void)
{
    unsigned int DATA_WIDTH a, mult, result;
    unsigned int (DATA_WIDTH*2 - 1) b;
    chanin input;
    chanout output;

    while (1)
    {
        input ? a;
        input ? result;
        b = result @ 0;
        mult = 1<<(DATA_WIDTH-1);
        result = 0;
        while (mult!=0)
        {
            if ((0 @ a) >= b)
                par
                {
                    a -= b <- width(a);
                    result |= mult;
                }
            par
            {
                b = b >> 1;
                mult = mult >> 1;
            }
        }
        output ! result;
    }
}
```

3.3.2 Compiling and Simulating the Program

Compilation and simulation of the program is performed by typing the following at the command prompt:

```
> handelc -s divide.c
```

This generates some lines something like this:

```
 Compiled :     1 gate,      0 inverters,   10 latches,   60 others
Optimised :     1 gate,      0 inverters,    7 latches,   52 others
 Expanded :   521 gates,   179 inverters,   89 latches,   14 others
Optimised :   331 gates,    72 inverters,   89 latches,   12 others
```

This is the result of the compilation phase and details the number of hardware gates required to implement the program. The simulator then starts immediately and prompts you for some input. The first input is the value of **a**, the second is the value of **b**. The simulator then executes the program and returns the integer value of **a/b**. An example session might look something like this:

```
 0: a=0 mult=0 result=0 b=0
 0: Input to `input' ? 56
 1: a=56 mult=0 result=0 b=0
 1: Input to `input' ? 6
 2: a=56 mult=0 result=6 b=0
 3: a=56 mult=0 result=6 b=196608
 4: a=56 mult=32768 result=6 b=196608
 5: a=56 mult=32768 result=0 b=196608
 6: a=56 mult=16384 result=0 b=98304
 7: a=56 mult=8192 result=0 b=49152
 8: a=56 mult=4096 result=0 b=24576
 9: a=56 mult=2048 result=0 b=12288
10: a=56 mult=1024 result=0 b=6144
11: a=56 mult=512 result=0 b=3072
12: a=56 mult=256 result=0 b=1536
13: a=56 mult=128 result=0 b=768
14: a=56 mult=64 result=0 b=384
15: a=56 mult=32 result=0 b=192
16: a=56 mult=16 result=0 b=96
17: a=56 mult=8 result=0 b=48
18: a=8 mult=8 result=8 b=48
19: a=8 mult=4 result=8 b=24
20: a=8 mult=2 result=8 b=12
21: a=8 mult=1 result=8 b=6
22: a=2 mult=1 result=9 b=6
23: a=2 mult=0 result=9 b=3
23: Ready to accept output from `output' ? (y/n) y
23: Output from channel `output' = 9
24: a=2 mult=0 result=9 b=3
```

The program has correctly calculated that the integer part of 56/6 is 9.

3.3.3 Detailed Explanation

Although the algorithm is a standard long division, some explanation of the details is now in order. The two values corresponding to the dividend and divisor are input from the simulator in that order via the `input` channel.

The variable `mult` contains a single bit, initially in the most significant bit, which is shifted right by 1 place at each iteration. The variable `b` is maintained to be the divisor multiplied by `mult` at each stage.

The comparison in the `if` statement checks whether `a` is greater than `mult` multiplied by the divisor (which is stored in `b`). If this is the case then `mult` multiplied by the divisor (stored in `b`) is subtracted from `a` and `mult` is added to the result.

When `mult` reaches zero, the division is complete and the result is output on the `output` channel.

For clarification, refer to the example traced out by the simulator for 56/6. The `while` loop starts at cycle 5 and ends at cycle 22. You can see that at each stage `mult` is divided by 2 and that `b` is maintained as `mult` multiplied by the divisor, 6. At each stage where `a` is greater than `b`, `result` has `mult` added to it and `a` has `b` subtracted from it.

By cycle 23, `result` contains 9 which is the correct answer for this calculation and `a` is the remainder of the division - i.e. 2.

3.3.4 Summary

While this example may seem trivial and tedious you should bear in mind what has been achieved. This program, with the help of the Handel-C compiler, has generated the hardware necessary for a 16 bit integer divider that executes in around 16 clock cycles (the actual execution time depends on the values passed in this example).

The way that this was achieved was by expressing the required algorithm in the Handel-C language much as it could have been done if targeting software. At no time was any knowledge of the actual hardware of a divider required to produce the result.

3.4 The Queue Example

This program illustrates the use of parallel tasks and channel communications by implementing a simple four place queue. Each task holds one piece of data and has an input channel connected to the previous queue location and an output channel connected to the next queue location.

3.4.1 Source Code Listing

The complete Handel-C listing is shown below. This program is also provided on the disk with the Handel-C compiler.

```
void main(void)
{
    chan unsigned int undefined link[3];
    chanin unsigned int 8 input;
    chanout unsigned int 8 output;
    unsigned int undefined state[4];

    par
    {
        // First queue location
        while (1)
        {
            input ? state[0];
            link[0] ! state[0];
        }
        // Second queue location
        while (1)
        {
            link[0] ? state[1];
            link[1] ! state[1];
        }
        // Third queue location
        while (1)
        {
            link[1] ? state[2];
            link[2] ! state[2];
        }
        // Fourth queue location
        while (1)
        {
            link[2] ? state[3];
            output ! state[3];
        }
    }
}
```

3.4.2 Compiling and Simulating the Program

Compilation and simulation of the program is performed by typing the following at the command prompt:

```
> handelc -s queue.c
```

This generates some lines something like this:

```
 Compiled :     0 gates,     0 inverters,    8 latches,   52 others
Optimised :     0 gates,     0 inverters,    8 latches,   41 others
 Expanded :    63 gates,    13 inverters,   52 latches,    7 others
Optimised :    32 gates,     9 inverters,   52 latches,    7 others
```

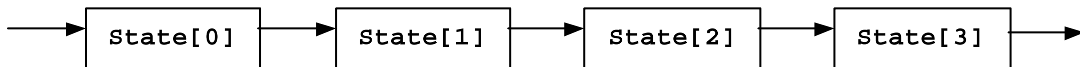
This is the result of the compilation phase and details the number of hardware gates required to implement the program. The simulator then starts immediately and prompts you for some input. You should continue to input data and read data from the queue and find that the output is delayed by 4 clock cycles from the input. An example session might look something like this:

```
 0: state[0]=0 state[1]=0 state[2]=0 state[3]=0
 0: Input to `input' ? 1
 1: state[0]=1 state[1]=0 state[2]=0 state[3]=0
 2: state[0]=1 state[1]=1 state[2]=0 state[3]=0
 2: Input to `input' ? 2
 3: state[0]=2 state[1]=1 state[2]=1 state[3]=0
 4: state[0]=2 state[1]=2 state[2]=1 state[3]=1
 4: Ready to accept output from `output' ? (y/n) y
 4: Output from channel `output' = 1
 4: Input to `input' ? 3
 5: state[0]=3 state[1]=2 state[2]=2 state[3]=1
 6: state[0]=3 state[1]=3 state[2]=2 state[3]=2
 6: Ready to accept output from `output' ? (y/n) y
 6: Output from channel `output' = 2
 6: Input to `input' ? 4
 7: state[0]=4 state[1]=3 state[2]=3 state[3]=2
 8: state[0]=4 state[1]=4 state[2]=3 state[3]=3
 8: Ready to accept output from `output' ? (y/n) y
 8: Output from channel `output' = 3
 8: Input to `input' ? 5
 9: state[0]=5 state[1]=4 state[2]=4 state[3]=3
10: state[0]=5 state[1]=5 state[2]=4 state[3]=4
10: Ready to accept output from `output' ? (y/n) y
10: Output from channel `output' = 4
10: Input to `input' ? 6
11: state[0]=6 state[1]=5 state[2]=5 state[3]=4
12: state[0]=6 state[1]=6 state[2]=5 state[3]=5
12: Ready to accept output from `output' ? (y/n) y
12: Output from channel `output' = 5
```

Note how data is delayed by 4 clock cycles. For example, the value 3 was input on clock cycle 4 and output on clock cycle 8.

3.4.3 Detailed Explanation

This example uses four parallel tasks each containing one word of data as its state. At each iteration, one word is passed between each of these tasks in a chain like this:



The links between the processes are entries in the **`links`** array of channels while the input and output to and from the system are connected to the simulator using the now familiar **`chanin`** and **`chanout`** keywords.

The queue only reads data and writes data on every other clock cycle.

3.4.4 Summary

This example has shown how to create parallel tasks and how to communicate between those tasks. It has also illustrated arrays of variables and arrays of channels.

Also, the queue presented here is parameterised on the width of the input and output channels because the width of all internal variables are undefined and inferred by the compiler.

3.5 The Microprocessor Example

As a final example for this chapter, Handel-C is used to implement a simple microprocessor. This microprocessor executes a program stored in ROM to calculate the first few members of the Fibonacci number sequence.

3.5.1 Source Code Listing

The complete Handel-C listing is shown below. This program is also provided on the disk with the Handel-C compiler.

```
chanin  input;
chanout output;

// Parameterisation
#define dw 32       /* Data width    */
#define opcw 4      /* Op-code width */
#define oprw 4      /* Operand width */

#define rom_aw 4    /* Width of ROM address bus */
#define ram_aw 4    /* Width of RAM address bus */

// The opcodes
#define HALT    0
#define LOAD    1
#define LOADI   2
#define STORE   3
#define ADD     4
#define SUB     5
#define JUMP    6
#define JUMPNZ  7
#define INPUT   8
#define OUTPUT  9

// The assembler macro
#define _asm_(opc, opr) (opc + (opr << opcw))

// Rom program data
rom unsigned int undefined program[] =
{
  _asm_(LOADI,  1),  /* 0 */  /* Get a one  */
  _asm_(STORE,  3),  /* 1 */  /* Store this */
  _asm_(STORE,  1),  /* 2 */
  _asm_(INPUT,  0),  /* 3 */  /* Read value from user */
  _asm_(STORE,  2),  /* 4 */  /* Store this */
  _asm_(LOAD,   1),  /* 5 */  /* Loop entry point */
  _asm_(ADD,    0),  /* 6 */  /* Make a fib number */
  _asm_(STORE,  0),  /* 7 */  /* Store it */
  _asm_(OUTPUT, 0),  /* 8 */  /* Output it */
  _asm_(ADD,    1),  /* 9 */  /* Make a fib number */
  _asm_(STORE,  1),  /* a */  /* Store it */
  _asm_(OUTPUT, 0),  /* b */  /* Output it */
  _asm_(LOAD,   2),  /* c */  /* Decrement counter */
  _asm_(SUB,    3),  /* d */
  _asm_(JUMPNZ, 4),  /* e */  /* Repeat if not zero */
  _asm_(HALT,   0)   /* f */
```

```
};

/* RAM for processor */
ram unsigned int dw data[1 << ram_aw];

/* Processor registers */
unsigned int rom_aw pc;      /* Program counter      */
unsigned int (opcw+oprw) ir; /* Instruction register */
unsigned int dw x;           /* Accumulator          */

/* Macros to extract opcode and operand fields */
#define opcode (ir <- opcw)
#define operand (ir \\ opcw)

/* Main program */
void main(void)
{
  pc = 0;

  // Processor loop
  do
  {
    // fetch
    par
    {
      ir = program[pc];
      pc = pc + 1;
    }

    // decode and execute
    switch (opcode)
    {
      case LOAD   : x = data[operand<-ram_aw]; break;
      case LOADI  : x = 0 @ operand; break;
      case STORE  : data[operand<-ram_aw] = x; break;
      case ADD    : x = x+data[operand<-ram_aw]; break;
      case SUB    : x = x-data[operand<-ram_aw]; break;
      case JUMP   : pc = operand<-rom_aw; break;
      case JUMPNZ : if (x!=0) pc=operand<-rom_aw; break;
      case INPUT  : input  ? x; break;
      case OUTPUT : output ! x; break;
      default     : while(1) delay;    // unknown opcode
    }

  } while (opcode != HALT);
}
```

3.5.2 Compiling and Simulating the Program

Compilation and simulation of the program is performed by typing the following at the command prompt:

```
> handelc -s proc.c
```

This generates some lines something like this:

```
 Compiled :     1 gate,      0 inverters,   13 latches,  120 others
Optimised :     1 gate,      0 inverters,   12 latches,  101 others
 Expanded :   509 gates,    55 inverters,   58 latches,   23 others
Optimised :   369 gates,    79 inverters,   57 latches,   22 others
```

This is the result of the compilation phase and details the number of hardware gates required to implement the program.

The simulator then starts immediately and prompts you for some input. You should enter a number here. The simulator then continues and outputs 2xn numbers from the Fibonacci series where n is the number you entered.

```
 0: pc=0 ir=0 x=0
 1: pc=0 ir=0 x=0
 2: pc=1 ir=18 x=0
 3: pc=1 ir=18 x=1
 4: pc=2 ir=51 x=1
 5: pc=2 ir=51 x=1
 6: pc=3 ir=19 x=1
 7: pc=3 ir=19 x=1
 8: pc=4 ir=8 x=1
 8: Input to `input' ? 2
 9: pc=4 ir=8 x=2
10: pc=5 ir=35 x=2
11: pc=5 ir=35 x=2
12: pc=6 ir=17 x=2
13: pc=6 ir=17 x=1
14: pc=7 ir=4 x=1
15: pc=7 ir=4 x=1
16: pc=8 ir=3 x=1
17: pc=8 ir=3 x=1
18: pc=9 ir=9 x=1
18: Ready to accept output from `output' ? (y/n) y
18: Output from channel `output' = 1
19: pc=9 ir=9 x=1
20: pc=10 ir=20 x=1
21: pc=10 ir=20 x=2
22: pc=11 ir=19 x=2
23: pc=11 ir=19 x=2
24: pc=12 ir=9 x=2
24: Ready to accept output from `output' ? (y/n) y
24: Output from channel `output' = 2
25: pc=12 ir=9 x=2
```

```
26: pc=13 ir=33 x=2
27: pc=13 ir=33 x=2
28: pc=14 ir=53 x=2
29: pc=14 ir=53 x=1
30: pc=15 ir=71 x=1
31: pc=4 ir=71 x=1
32: pc=5 ir=35 x=1
33: pc=5 ir=35 x=1
34: pc=6 ir=17 x=1
35: pc=6 ir=17 x=2
36: pc=7 ir=4 x=2
37: pc=7 ir=4 x=3
38: pc=8 ir=3 x=3
39: pc=8 ir=3 x=3
40: pc=9 ir=9 x=3
40: Ready to accept output from `output' ? (y/n) y
40: Output from channel `output' = 3
41: pc=9 ir=9 x=3
42: pc=10 ir=20 x=3
43: pc=10 ir=20 x=5
44: pc=11 ir=19 x=5
45: pc=11 ir=19 x=5
46: pc=12 ir=9 x=5
46: Ready to accept output from `output' ? (y/n) y
46: Output from channel `output' = 5
47: pc=12 ir=9 x=5
48: pc=13 ir=33 x=5
49: pc=13 ir=33 x=1
50: pc=14 ir=53 x=1
51: pc=14 ir=53 x=0
52: pc=15 ir=71 x=0
53: pc=0 ir=0 x=0
```

Thus, the first 4 numbers, 1, 2, 3 and 5 have been returned correctly.

3.5.3 Detailed Explanation

The system described in this example consists of a ROM containing the program to execute, a RAM containing some scratch variables and a processor that understands 10 opcodes. Each instruction is made up of a 4 bit opcode and a 4 bit operand. The **`_asm_`** preprocessor macro is the assembler for this language and is used to fill in the entries in the **`program`** ROM declaration.

The processor has three registers: a program counter, **`pc`**, that points to the next instruction to be fetched from the ROM, an instruction register, **`ir`**, containing the instruction being executed and an accumulator register, **`x`**, used as one input to the 'ALU'.

The instructions that the processor can execute are:

Opcode	Description
`HALT`	Stop processing
`LOAD`	Load a value from RAM into x
`LOADI`	Load a constant into x
`STORE`	Store x to RAM
`ADD`	Add a value from RAM to x
`SUB`	Subtract a value from RAM from x
`JUMP`	Unconditional jump to a ROM location
`JUMPNZ`	Jump to a ROM location if x is not 0
`INPUT`	Read a word from user into x
`OUTPUT`	Write x to the user

Using these instructions, a ROM is built containing a program to generate the Fibonacci numbers.

The execution unit of the processor simply fetches instructions from the **`program`** ROM and executes them using a **`switch`** statement.

3.5.4 Summary

This example has demonstrated a large number of the most common constructs used in Handel-C. While it may appear to be a simple example it should be easy to see how this example could be extended to implement a more complex processor.

What we have produced here is a processor which only contains the instructions necessary to calculate Fibonacci numbers. It is equally possible to produce processors which contain specialised instructions for any application. Thus, you could use Handel-C to develop processors capable of executing programs for specialised applications with the minimum of effort.

4. Macros

4.1 Introduction

As mentioned in previous chapters, 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. There are some limitations to this approach. 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 as part of the language which allows more powerful macros to be defined (for example, recursive macro expressions). In addition, Handel-C supports shared macros 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.

This chapter details how to define macros and shared hardware.

4.2 Macro Expressions

Macros may be used to replace expressions to avoid tedious repetition. Handel-C provides some powerful macro constructs to allow complex expressions to be generated simply.

4.2.1 Constant Macro Expressions

This first form of the macro is an 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.

More generally, real expressions can be used. For example:

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

v = sum;
w = sum;
```

4.2.2 Parameterised Macro Expressions

Handel-C also 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;
```

Again, 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 other words, in this example, an adder is generated in hardware every time the **add3()** macro is used.

4.2.3 The select Operator

Handel-C provides a **select(...)**operator which is used to mean 'select at compile time'. Its general usage is:

select(*Expression***,** *Expression***,** *Expression***)**

Here, the first expression must be a compile time constant. If the first expression evaluates to true then the Handel-C compiler replaces the whole expression with the second expression. If the first expression evaluates to false then the Handel-C compiler replaces the whole expression with the second expression. The difference between this and the **? :** operators is best illustrated with an example.

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

This 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 in this case 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.

A more useful example can be seen when macros are combined with this feature. For example:

```
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 equalises 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;
```

However, the macro comes into its own if the width of one of the variables changes. For example, 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. By 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.

Another example of where this form of macro 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.

4.2.4 Recursive Macro Expressions

A serious limitation with preprocessor macros (those defined with **#define**) is their inability to generate recursive expressions. By combining Handel-C macros (those defined with **macro expr**) and the **select(...)** operator discussed above, recursive macros can be used to simply express complex hardware. 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.

As an example, let us take sign extension of a variable. 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. For example, the 4 bit representation of -2 is 0b1110. When assigned to an 8 bit wide variable, this should become 0b11111110. In contrast, the 4 bit representation of 6 is 0b0110. When assigned to an 8 bit wide variable, this should become 0b00000110.

In this example, the following code would suffice:

```
int 8 x;
int 4 y;

x = y[3] @ y[3] @ y[3] @ y[3] @ y;
```

As you can see, this can rapidly become tedious for variables that differ by a significant number of bits. Also, what if the exact widths of the variables are not known? What is needed is a macro to sign extend a variable. For example:

```
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));
```

Here, 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 simply 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)`.

4.2.5 Recursive Macro Expressions - A Larger Example

A second example of the use of recursive macro expressions is now given to illustrate the generation of large quantities of hardware from simple macros. The example used is that of 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** which 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)==1 ? c*8 : 0) +
    ((b & 4)==1 ? c*4 : 0) +
    ((b & 2)==1 ? c*2 : 0) +
    ((b & 1)==1 ? c : 0);
```

which is a standard long multiplication calculation.

4.2.6 Shared Expressions

By default, Handel-C generates all the hardware required for every expression in the whole program. In many programs, this means that large parts of the hardware will be idle for long periods of time. The shared expression allows 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.

An example program where shared expressions are extremely useful is:

```
a = b * c;
d = e * f;
g = h * i;
```

Here, three multipliers will be generated but each one will only be used once and none of them simultaneously. This is a massive waste of hardware. The way to improve this program is:

```
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 which is a better use of hardware. (In fact, the above example could be built as three multipliers executing in parallel if the maximum performance is required).

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.

4.2.7 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 the recursive multiplier macro example above 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);
```

Here, the macro expression builds a multiplier and the shared expression allows that hardware to be shared between the two assignments.

4.2.8 Restrictions on Shared Expressions

A limitation to shared expressions is that they 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. The compiler generates a warning if you attempt to perform such an operation.

Refer to chapter 5 for more details on timing of Handel-C programs and for details of how you can tell which clock cycle operations are performed on. This becomes an important skill when using shared expressions.

4.3 Macro Procedures

Macros may be used to replace statements to avoid tedious repetition. Handel-C provides simple macro constructs to expand single statements into complex blocks of code.

The general syntax of macro procedures is:

macro proc *Name*(*Params*) *Statement*

For 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.

Macro procedures generates the hardware for their statement every time they are referenced. The above example expands to 4 channel output statements.

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. For example:

```
#define test(x,y) if (x!=(y<<2))

test(a,b)
{
    a++;
}
else
{
    b++;
}
```

This is a valid preprocessor macro definition. However, the following code is not allowed:

```
macro proc test(x,y) if (x!=(y<<2));

test(a,b)  // NOT ALLOWED
{
    a++;
}
else
{
    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 such as these examples which is generally unreadable and difficult to maintain.

5. Timing and Efficiency Information

5.1 Introduction

A Handel-C program executes with one clock source for the whole program. 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.

Knowing about clock cycles also becomes important when considering interfaces to external hardware. This subject is covered in greater detail in chapter 6 but it is important to understand timing issues before moving on to implementing such interfaces because it likely that the external device places constraints on when signals should change.

This chapter also deals with the subject of overall performance. We shall see that avoiding certain constructs has a dramatic influence on the maximum clock rate that your Handel-C program can run at and some guidelines are given for improving your hardware performance.

An example is given that covers the considerations for real time constraints on a system.

5.2 Clock Cycle Timing of Language Constructs

This section deals with the analysis of a program in terms of the number of clock cycles it takes to execute. The Handel-C language has been designed so that an experienced programmer can immediately tell which instructions execute on which clock cycles. This information becomes particularly important when your program contains multiple interacting parallel processes.

5.2.1 Statement Timing

The basic rule for cycles used in a Handel-C program is:

> ***Assignment and delay take 1 clock cycle.***
> ***Everything else is free.***

What this means is that every time you write an assignment statement or a `delay` statement, you use one clock cycle but you can write any other piece of code and not use any clock cycles to execute it.

The only exception is channel communication which takes one clock cycle only if both parties are ready to communicate. This means that if one parallel branch is ready to output on a channel when another branch is ready to receive then it takes one clock cycle for the data to be assigned to the variable in the input statement. If one of the branches is not ready for the data transfer then execution of the other branch waits until both branches become ready. Even if both branches are ready for the transfer then one clock cycle is used because channel input is a form of assignment.

Some simple examples with their timings are shown below.

```
x = y;
x = (((y * z) + (w * v))<<2)<-7;
```

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 simply builds the combinatorial 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.

```
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 and the total time for execution is just 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.

```
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. Notice how the condition of the `while` loop takes no clock cycles because there is no assignment involved.

```
for (x = 0; x < 5; x ++)
{
    a += b;
    b *= 2;
}
```

As discussed in chapter 2, this code has a direct equivalent which is:

```
{
    x = 0;
    while (x<5)
    {
        a += b;
        b *= 2;
        x ++;
    }
}
```

This code takes 16 clock cycles to execute. One is required for the initialisation of **a** and three for each execution of the body. Since the body is executed 5 times, this gives a total of 16 clock cycles.

```
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;
}
```

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**.

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 would be:

```
par
{
    {                   // Parallel branch 1
        a = b;
        c = d;
        link ! x;
    }

    link ? y;           // Parallel branch 2
}
```

Here, the first branch of the **par** statement takes a total of 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. A summary of statement timings is given below.

Statement	Timing
`{...}`	Sum of all statements in sequential block
`par {...}`	Length of longest branch in block
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 `<<=` *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
Channel `!` *Expression*`;`	1 clock cycle when receiver is ready
`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
`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
`delay;`	1 clock cycle

Note: The Handel-C compiler may insert **`delay`** statements to break combinatorial loops. See next section for details.

5.2.2 Avoiding Combinatorial Loops

Consider the following example:

```
while (x!=3); // WARNING!!
```

If **`x`** is modified in a parallel process then this loop should wait for **`x`** to become 3 before allowing the program to continue. However, this code is not allowed in Handel-C because it generates a

combinatorial loop in the logic because of the way that Handel-C expressions are built to evaluate in zero clock cycles. This loop must be broken by changing the code to:

```
while (x!=3)
{
    delay;
}
```

This loop takes no longer to execute than the first one but does not contain a combinatorial loop because of the clock cycle delay in the loop body.

In actual fact, the Handel-C compiler spots this form of error and inserts the **delay** statement itself and generates a warning. It is considered better practice to include the **delay** statement in the code to make it more readable.

Beware of code which may look correct but has the same error. For example:

```
while (x!=3)
{
    if (y>z)
    {
        a++;
    }
}
```

As seen above, 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 combinatorial loop is still generated. The solution in this example is to add the **else** part of the if construct as follows:

```
while (x!=3)
{
    if (y>z)
    {
        a++;
    }
    else
    {
        delay;
    }
}
```

Similar problems occur with **`do ... while`** loops and **`switch`** statements.

5.2.3 Parallel Access to Variables

As discussed in chapter 1, the rules of parallelism state that the same variable must not be accessed from two separate parallel branches. This rule is there to avoid resource conflicts on the variables. However, if care is taken then this 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. The analysis presented in this chapter allows the programmer to determine precisely when an assignment will take place and avoid such conflicts.

This relaxation allows some useful and 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 dependant, the Handel-C compiler cannot determine when two assignments are made to the same variable on the same clock cycle. It therefore generate warnings based on the strict rule that the same variable may not be used in more than one parallel branch. These warnings should be taken seriously and each one checked to ensure that the relaxed rule of parallelism is still obeyed.

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];
    }
}
```

Here, 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:

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
5	10	9	8	7	6
6	11	10	9	8	7
7	12	11	10	9	8
8	13	12	11	10	9

5.2.4 Multiple Simultaneous Use of RAMs and ROMs

Beware of the following code:

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

This code does not execute correctly because of the multiple use of the RAM in the expression. The Handel-C compiler generates a warning for this code and the simulator fails if it encounters it. 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.

5.2.5 Detailed Timing Example

We now illustrate the analysis of Handel-C programs by an example that generates signals tied to real-world constraints. The example used is the generation of a 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 so we assume a value of 5 MHz. The Handel-C program is shown below.

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.

```
void main(void)
{
    unsigned 20 MicroSeconds;
    unsigned 6 Seconds;
    unsigned 6 Minutes;
    unsigned 16 Hours;
    unsigned 3 Count;

    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++;
                                    }
                            }
                    }
            }
    }
```

5.3 Time Efficiency of Handel-C Hardware

Handle-C requires that the clock period for a program is longer than the longest path through combinatorial logic in the whole program. This means that, for example, once FPGA 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.

But what if this calculated rate is not fast enough for the system performance or real time constraints? This section deals with optimisations that can be made to your program to reduce the longest path delay and increase the maximum possible clock rate.

5.3.1 Reducing Logic Depth

It is important to remember when designing Handel-C programs which operations combine to produce deep logic. Deep logic results in long path delays in the final circuit so reducing logic depth should help to increase clock speed. Some guidelines will now be given for reducing logic depth.

1. The operator that produces the deepest logic is multiplication. A single cycle multiplier produces a large amount of hardware and long delays through deep logic so you should avoid using multipliers wherever possible. Remember that most common 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 (see section 5.3.2).

2. Wide adders require deep logic for the carry ripple. Consider using more clock cycles with shorter adders. For example, to reduce a single, 8 bit wide adder to 3, narrower adders:

```
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];
```

3. Avoid greater than and less than comparisons - they produce deep logic. For 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:

```
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.

4. Reduce complex expressions into a number of stages. For 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.

5. Avoid long strings of empty statements. Empty statements result from, for example, missing `else` conditions from `if` statements. For 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.

5.3.2 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. An example of a pipelined multiplier is given below.

```
unsigned 8 sum[8];
unsigned 8 a[8];
unsigned 8 b[8];
chanin inputa;
chanin inputb;
chanout output;

par
{
    while(1)
        inputa ? a[0];

    while(1)
        inputb ? b[0];

    while(1)
        output ! sum[7];

    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]);
            level(1);
            level(2);
            level(3);
            level(4);
            level(5);
            level(6);
            level(7);
        }
    }
}
```

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, combinatorial 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 to avoid tedious repetition of code.

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]**.

6. Targetting Hardware

6.1 Introduction

The previous chapters have covered most aspects of writing Handel-C programs. What has not yet been discussed is how to get data into and out of those programs. One of the major advantages of using custom hardware such as that which can be produced with Handel-C is its ability to interface directly with external components such as RAM, custom and non-custom buses.

This chapter deals with getting data into and out from your Handel-C program. We start with a discussion of using the simulator provided with the Handel-C compiler to ensure that your program is correct before moving on to detail interfacing with real hardware devices connected to the pins of the chip containing your hardware.

The simulator executes Handel-C programs on the compiling machine without any additional hardware. It allows output to the screen or a file and input from the keyboard or a file. It is a powerful tool that allows programs to be tested thoroughly before custom hardware is manufactured.

While no specific hardware platform is detailed here, a number of examples are given of interfacing to theoretical hardware.

6.2 Interfacing with the Simulator

The earlier chapter of simple examples (chapter 3) briefly described how to get single words from the simulator to your Handel-C program and how to get results back to the screen. In this section, this procedure is covered in more detail and is extended to cover transferring blocks of data through your program to allow debugging with real data.

6.2.1 Single Word Transfers

Communication with the simulator takes place over channels. Special channels must be defined for inputting information from the simulator and outputting information back to the simulator. For example:

```
chanin int 8 input;
chanout int 15 output;

input ? x;
output ! y;
```

This example declares two channels - one for input from the simulator and one for output to the simulator. The standard channel communication statements can then be used to transfer data from and to the simulator.

Note that channels connected for input from the simulator are declared with the keyword **chanin** rather than **chan** as would be used for internal channels. Similarly, channels connected for output to the simulator are declared with the keyword **chanout** rather than **chan**.

It is also valid to declare multiple channels for input and output. For example:

```
chanin int 8 input_1;
chanin int 16 input_2;
chanout unsigned 3 output_1;
chanout char output_2;

input_1 ? a;
input_2 ? b;
output_1 ! (unsigned 3)(((0 @ a) + b) <- 3);
output_2 ! a;
```

When simulated, such a program prompts for input to the named channels from the simulating computer and displays the name of channels before outputting their value on the simulating computer screen.

6.2.2 Block Data Transfers

When processing large quantities of data or repeatedly running programs such as might be required for debugging, typing individual words into the simulator rapidly becomes tedious. The Handel-C simulator also 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;
    }
}
```

This program reads data from the file **in.dat** and writes its results to the file **out.dat**. The **in.dat** 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. For example:

```
56
0x34
0654
0b001001
```

When simulated with this input file, the above program generates a file **out.dat** containing the decimal results as follows:

```
57
53
429
10
```

This feature allows 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 taken by the simulator and a conversion back from the output file to an image format. The results can then be viewed and the correct operation of the Handel-C program confirmed. Chapter 8 demonstrates just such a process by implementing an edge detector in Handel-C and using the simulator to debug the program.

6.3 Targeting FPGA Devices

The Handel-C language is designed to target real hardware devices. There are a number of important pieces of information that must be supplied to the compiler to allow it to do this. These are the FPGA part that the design is to be implemented in and the location of a clock source. These parameters are specified using the '`set`' command.

6.3.1 Targeting Specific Devices

In order to target a specific FPGA, the compiler must be supplied with the FPGA part number. Ultimately, this information is passed to the FPGA place and route tool to inform it of the device it should target.

Targeting devices consists of two parts - the target family and the target device. The general syntax is:

set family = *Family***;**
set part = *Chip Number***;**

Recognised families are:

Family Name	Description
`Xilinx3000`	3000 series Xilinx FPGAs
`Xilinx4000`	4000 series Xilinx FPGAs
`Xilinx4000A`	4000A series Xilinx FPGAs
`Xilinx4000D`	4000D series Xilinx FPGAs
`Xilinx4000H`	4000H series Xilinx FPGAs
`Xilinx4000E`	4000E series Xilinx FPGAs
`Xilinx4000L`	4000L series Xilinx FPGAs
`Xilinx4000EX`	4000EX series Xilinx FPGAs
`Xilinx4000XL`	4000XL series Xilinx FPGAs
`Xilinx4000XV`	4000XV series Xilinx FPGAs
`Altera6K`	Flex6K series Altera FPGAs
`Altera8K`	Flex8K series Altera FPGAs
`Altera10K`	Flex10K series Altera FPGAs

The chip number is the complete Xilinx or Altera 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. The **Xilinx3000** family covers all Xilinx 3000 devices with any suffix.

To target Altera devices:

```
set family = Altera10K;
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 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 software. Refer to chapter 4 in the Compiler Reference Manual for further details of selecting FPGA part numbers.

6.3.2 Locating the Clock

Since Handel-C generates synchronous hardware, a single clock source is required to run your program. The clock is normally provided on one of the external pins of the FPGA but may also be generated internally on Xilinx 4000 devices. 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).
external *Pin*	Clock from device pin.
external_divide *Pin Factor*	Clock from device pin with integer division.

Examples of clocks taken from external device pins are:

```
set clock = external “P35”;
set clock = external_divide “P35” 3;
```

The first of these examples specifies a clock taken from pin P35. The second of these examples specifies a clock taken from pin P35 which is divided on the FPGA by a factor of 3.

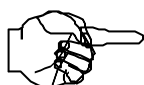
Examples of clocks taken from the Xilinx 4000 series internal clock generator are:

```
set clock = internal “F8M”;
set clock = internal_divide “F8M” 3;
```

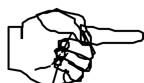
Currently, the frequency of the internal clock may take one of the following values:

Specification String	Frequency
“F15”	15Hz
“F490”	490Hz
“F16K”	16kHz
“F500K”	500kHz
“F8M”	8MHz

Note that the tolerance for these values is -50% to +25% so you should not rely on the internal clock being at exactly these frequencies.



Internal clocks are only supported on Xilinx 4000 series parts.



The clock division specified with the internal_divide and external_divide keywords must be a constant integer.

6.4 Use of RAMs and ROMs with Handel-C

Handel-C provides support for interfacing to both on-chip and off-chip RAMs and ROMs using the **ram** and **rom** keywords.

6.4.1 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 and 4000XV series parts directly simply by declaring a RAM or ROM in the way described in section 2.3.10. For example:

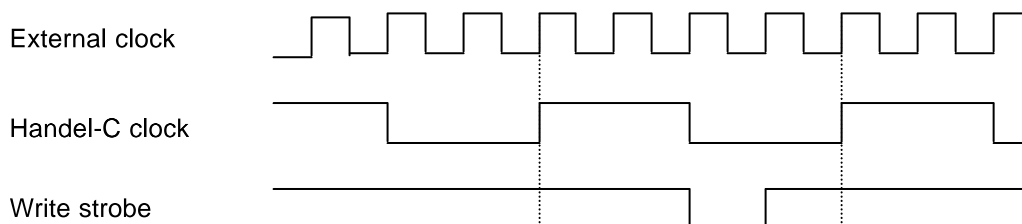
```
ram unsigned 6 x[34];
```

This will declare a RAM with 34 entries, each of which is 6 bits wide. For simplicity, it is recommended that new designs use Xilinx parts with synchronous RAMs.

RAM on other Xilinx 4000 series devices is asynchronous but can still be accessed in one of three ways. If the external clock 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) then 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 with the granularity of the external (undivided) clock. The above example starts the pulse 2 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:



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.

The second method of accessing asynchronous RAMs should be used when the external clock runs at the same rate as the Handel-C clock. It involves using multiple Handel-C RAM accesses to meet the setup and hold times of the RAM. For example, to write to an asynchronous RAM, the following code could be used.

```
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.

The third method of accessing asynchronous RAMs is half way between the two previous methods. The **wegate** specification allows the write strobe to be placed in either the first half or the second half of an undivided clock. 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.

6.4.2 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.

As with Xilinx devices, the RAM blocks in Flex 10K parts can be configured to be either synchronous or asynchronous. By default, Handel-C will use a synchronous access by utilising the falling edge of the clock as the input clock signal to the RAM. This is the recommended method for using RAMs.

By adding one of the **westart**, **welength** or **wegate** specifications described in the previous section, the Handel-C compiler will generate an asynchronous RAM. This may help with the timing characteristics of the design.

RAM/ROM initialisation files with a **.mif** extension will be generated on compilation to feed into the Max+Plus II 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.

6.4.3 Using External 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};
```

For example, 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

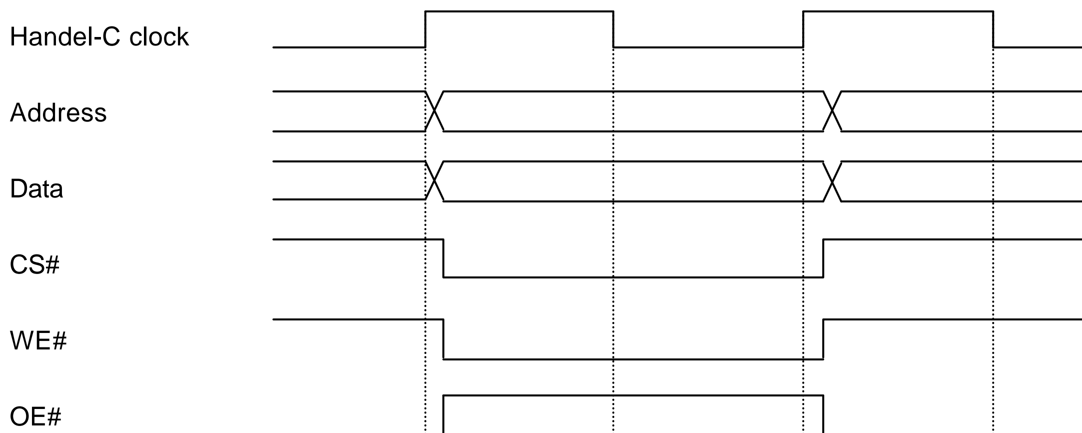
the width of the RAM (8 bits in this example) and the number of address lines used (14 in this example) from the RAM's usage. However, this is not recommended since this declaration deals with real external hardware which has a fixed definition.

Accessing the RAM is the same as for accessing internal RAM. For example:

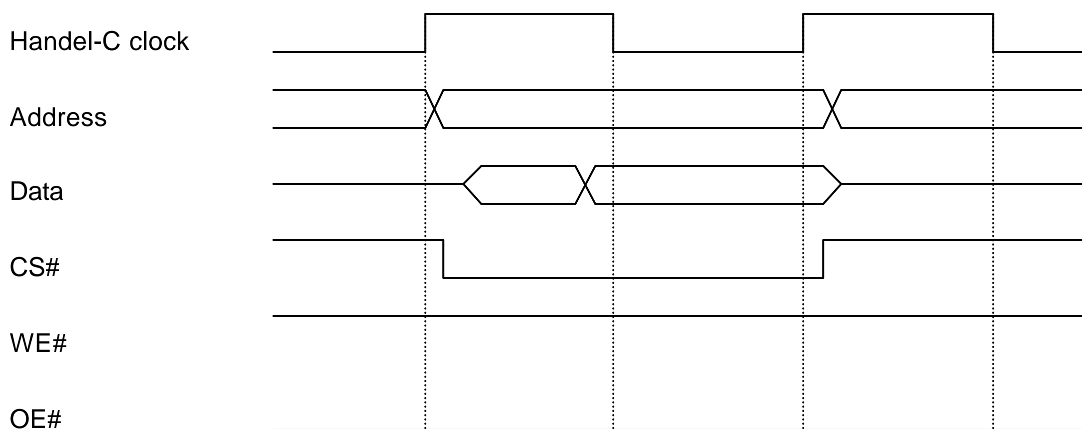
```
ExtRAM[1234] = 23;
y = ExtRam[5678];
```

Similar restrictions apply as with internal RAM - only one access may be made to the RAM in any one clock cycle. See chapter 2 for details of this restriction.

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:



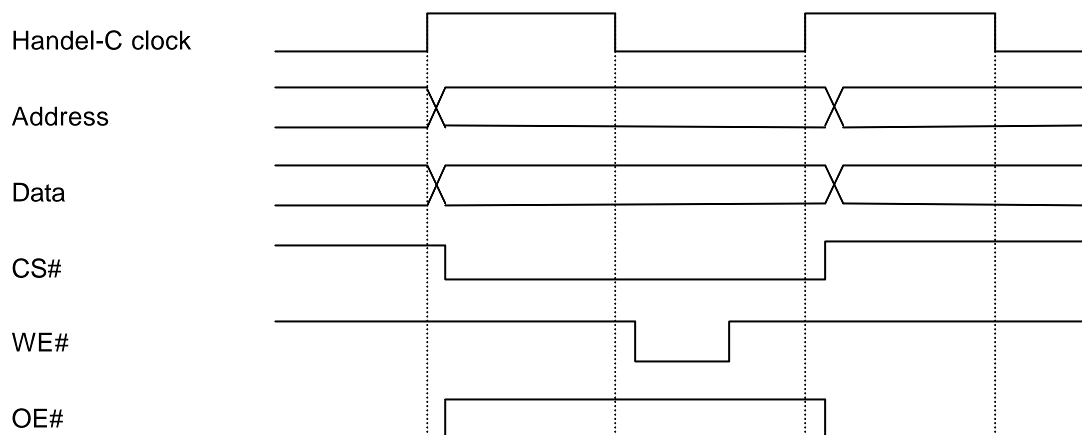
This cycle may not be suitable for the RAM device in use. In particular, asynchronous static RAM may not work with the above cycle due to setup and hold timing violations. For this reason, the **westart**, **welength** and **wegate** specifications described in section 6.4.1 may also be used with external RAM declarations.

For example, to declare a 16Kbyte by 8 bit RAM with the same strobe characteristics described in section 6.4.1:

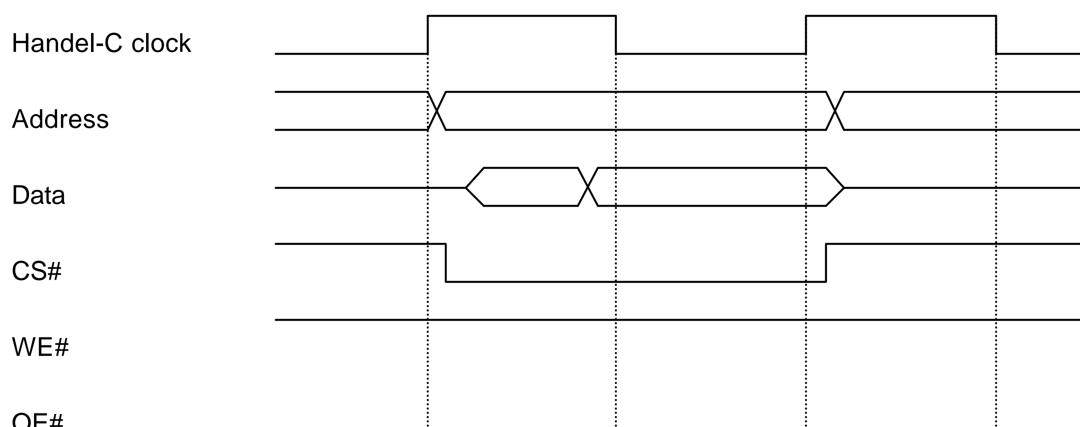
```
set clock = external_divide "P99" 4;

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:



The compiled hardware generates the following cycle for a read from external RAM:



Accessing the RAM is the same as for accessing internal RAM. For example:

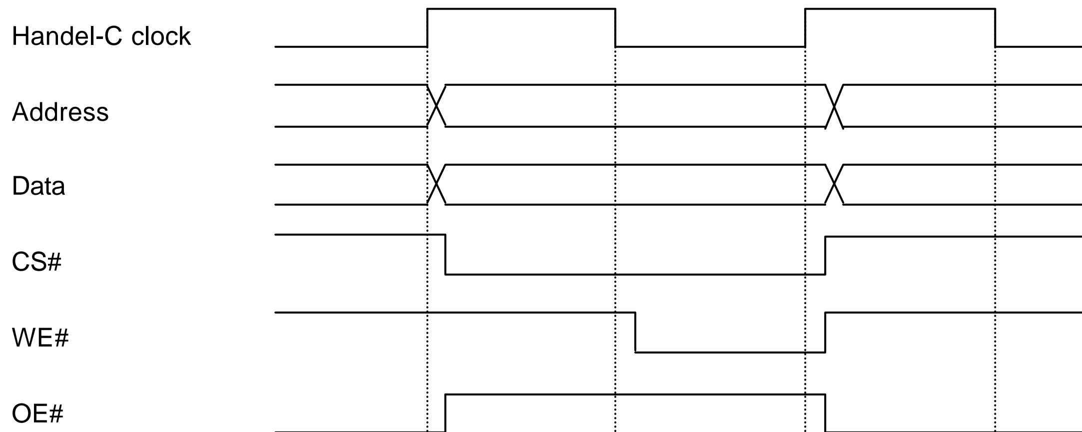
```
ExtRAM[1234] = 23;
y = ExtRam[5678];
```

Similar restrictions apply as with internal RAM - only one access may be made to the RAM in any one clock cycle. See section 2 for details of this restriction.

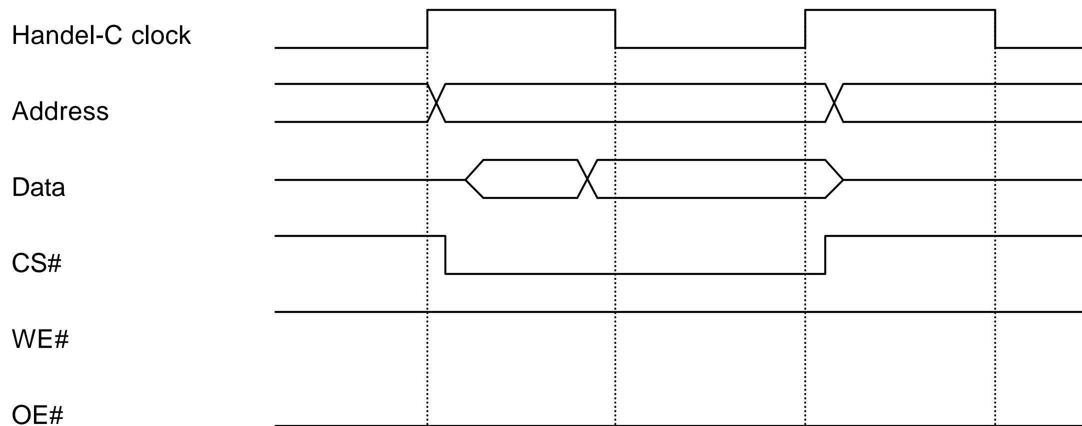
While this is the recommended method for accessing external RAMs, the **wegate** specification may be used when a multiplied clock is not available. For example, to declare a 16Kbyte by 8 bit RAM:

```
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:



Accessing the RAM is the same as for accessing internal RAM. For example:

```
ExtRAM[3] = Data;
Dummy = ExtRAM[3];
```

Similar restrictions apply as with internal RAM - only one access may be made to the RAM in any one clock cycle. See section 2 for details of this restriction.

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 this 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.

6.4.4 Using External ROMs

External ROMs are simply 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 not write strobe signal on a ROM device.

6.4.5 Using Other RAMs

The interface to other types of RAM such as DRAM or synchronous pipelined SRAM should be written by hand using interface declarations described in the following sections. Macro procedures can then be written to perform complex or even multi-cycle accesses to the external device.

6.5 Interfacing With External Hardware

While the simulator allows debugging of Handel-C programs, the real target of the compiler is hardware. It is therefore essential that the compiler can generate hardware that interfaces with external components. These next sections detail the building blocks of such hardware interfaces.

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.

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 manufacturer's place-and-route tools.

6.5.1 Off-chip Interfaces

All off-chip interfaces other than RAMs are declared with the **interface** keyword. The general syntax of interfaces is as follows:

interface *Sort*(*Types*) *Name*(*Args*) **with** {*Specs*};

Here, the *Sort* field specifies what sort of interface is required, *Types* describes the types of values associated with objects coming from the interface, *Name* specifies an identifier for the interface, *Args* specifies any parameters that the interface may require and *Specs* give hardware details of the interface such as chip pin numbers.

Handel-C currently provides the following interface sorts:

Type Identifier	Description
bus_in	Input bus from pins
bus_latch_in	Latched 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 latched input
bus_ts_clock_in	Bi-directional tri-state bus with clocked input

6.5.2 Reading from External Pins

The **bus_in** interface sort allows Handel-C programs to read from external pins. Its general usage is:

```
interface bus_in(Type) Name()
                              with {data = {Pin List}};
```

A specific example might be:

```
interface bus_in(int 4) InBus() with {data =
                            {"P1", "P2", "P3", "P4"}};
```

This declares a bus connected to pins P1, P2, P3 and P4 where pin P1 is the most significant bit and pin P4 is the least significant bit. Reading the bus is performed by accessing the identifier *Name*.**in** as a variable which will return the value on those pins at that clock edge. For example:

```
int 4 x;

x = InBus.in;
```

This sets the variable **x** to the value on the external pins. The type of **InBus.in** is **int 4** as specified in the type list after the **bus_in** keyword.

6.5.3 Latched Reading from External Pins

The **bus_latch_in** interface sort is similar to the **bus_in** interface sort but allows the input to be latched on a condition. This may be required to sample the signal at particular times. Its general usage is:

```
interface bus_latch_in(Type) Name(Condition)
                              with {data = {Pin List}};
```

Its usage is exactly like the **bus_in** interface sort except that *Condition* specifies a signal that is used to clock the input latches in the FPGA. The rising edge of this signal clocks the external signal to the internal value. For example:

```
int 1 get;
int 4 x;

interface bus_latch_in(int 4) InBus(get) with
          {data = {"P1", "P2", "P3", "P4"}};

get = 0;
get = 1;      // Latch the external value
x = InBus.in; // Read the latched value
```

6.5.4 Clocked Reading from External Pins

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 synchronise the signal to the Handel-C clock. Its general usage is:

```
interface bus_clock_in(Type) Name()
                          with {data = {Pin List}};
```

Its usage is exactly like the **bus_in** interface sort. The rising edge of the Handel-C clock clocks the external signal to the internal value. For example:

```
interface bus_clock_in(int 4) InBus() with
           {data = {"P1", "P2", "P3", "P4"}};

x = InBus.in; // Read latched value
```

6.5.5 Writing to External Pins

The **bus_out** interface sort allows Handel-C programs to write to external pins. Its general usage is:

```
interface bus_out() Name(Expression)
                          with {data = {Pin List}};
```

A specific example might be:

```
interface bus_out () OutBus(x+y) with {data =
                        {"P1", "P2", "P3", "P4"}};
```

This declares a bus connected to pins 1, 2, 3 and 4 where pin 1 is the most significant bit and pin 4 is the least significant bit. The value appearing on the external pins is the value of the expression **x+y** at all times.

6.5.6 Bi-directional Data Transfer

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) Name(Value,
                Condition) with {data = {Pin List}};
```

Here, *Value* and *Condition* are two expressions. *Value* refers to the value to output on the pins and *Condition* refers to the condition for driving the pins. When the second expression is non-zero (i.e. true), the value of the first expression is driven on the pins. When the value of the second expression is zero, the pins are tri-stated and the value of the external bus can be read using the identifier *Name*.**in** in much the same way that **bus_in** interfaces work.

A specific example might be:

```
int 1 enable;
int 4 x;

interface bus_ts(int 4) BiBus(x+1, enable==1)
        with {data = {"P1", "P2", "P3", "P4"}};

enable = 0;   // Tri-state the pins
x = BiBus.in; // Read the value
enable = 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.

The type of **BiBus.in** is **int 4** as specified in the type list after the **bus_ts** keyword.

Take care when driving tri-state buses that the FPGA and another device on the bus cannot drive simultaneously as this may result in damage to one or both of them.

6.5.7 Bi-directional Data Transfer with Latched Input

The **bus_ts_latch_in** interface sort allows Handel-C programs to perform bi-directional off-chip communications via external pins with inputs latched on a condition. Its general usage is:

```
interface bus_ts_latch_in (Type)
      Name(Value, Condition, Clock)
                          with {data = {Pin List}};
```

Here, *Value*, *Condition* and *Clock* are all expressions. *Value* refers to the value to output on the pins, *Condition* refers to the condition for driving the pins and *Clock* refers to the signal to clock the input from the pins. When the second expression is non-zero, the value of the first expression is driven on the pins. When the value of the second expression is zero, the pins are tri-stated and the value of the external bus can be read using the identifier *Name***.in** in much the same way that **bus_in** interfaces work.

The rising edge of the value of the third expression latches the external values through to the internal values on the chip. For example:

```
int 1 get;
int 1 enable;
int 4 x;

interface bus_ts_latch_in(int 4) BiBus(
     x+1, enable==1, get)
       with {data = {"P1", "P2", "P3", "P4"}};

enable = 0;   // Tri-state external pins
get = 0;
get = 1;      // Latch external value
x = BiBus.in; // Read latched value
enable = 1;   // Drive x+1 onto external pins
```

This example samples the external bus and reads the latched value into variable **x** and then drives the value of **x + 1** onto the external pins.

The type of **BiBus.in** is **int 4** as specified in the type list after the **bus_ts_latch_in** keyword.

Take care when driving tri-state buses that the FPGA and another device on the bus cannot drive simultaneously as this may result in damage to one or both of them.

6.5.8 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)
        Name(Value, Condition)
                          with {data = {Pin List}};
```

Here, *Value* and *Condition* are expressions. Value refers to the value to output on the pins and *Condition* refers to the condition for driving the pins. When the *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*.**in** in much the same way that **bus_in** interfaces work.

The rising edge of the Handel-C clock latches the external values through to the internal values on the chip. For example:

```
int 1 enable;
int 4 x;

interface bus_ts_clock_in (int 4) BiBus(
     x+1, enable==1)
       with {data = {"P1", "P2", "P3", "P4"}};

enable = 0;    // Tri-state external pins
x = BiBus.in;  // Read latched value
enable = 1;    // Drive x+1 onto external pins
```

This example reads the latched value into variable **x** and then drives the value of **x + 1** onto the external pins.

The type of **BiBus.in** is **int 4** as specified in the type list after the **bus_ts_clock_in** keyword.

Take care when driving tri-state buses that the FPGA and another device on the bus cannot drive simultaneously as this may result in damage to one or both of them.

6.5.9 Buses and the Simulator

The Handel-C simulator is capable of limited simulation of buses. The recommended process for debugging is to use the channel method outlined earlier in this chapter. This is because the simulation of buses cannot determine when input and output should occur. Rather, the simulator asks for and presents information at each clock cycle which can be tedious.

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. For example:

```
#define SIMULATE
#ifdef SIMULATE
    input ? value;
#else
    value = BusIn.in;
#endif
```

Refer to the Handel-C Preprocessor Reference Manual for details of conditional compilation.

Simulation of buses may be important when debugging your interface with the outside world. No extra work is required to allow this - the simulator simply connects automatically to the buses declared in your program. The simulator prompts for input when required. The output from buses can be read out from amongst the variable states at each clock.

To see how the simulator handles buses, try simulating the full example given later in this chapter.

6.5.10 Timing Considerations of Buses

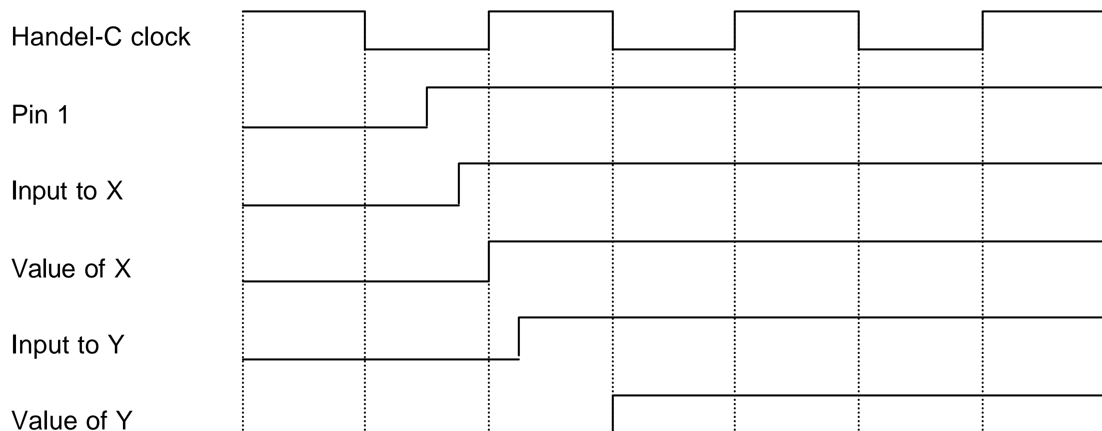
It is sometimes important to be aware of the timing of the external interfaces. While Handel-C without hardware libraries does not allow you to control exact timings, some care when writing code can allow enough control to make such interfaces work.

The first consideration is for **bus_in** interfaces. This form of bus is built with no latch between the external pin and the points inside the FPGA where the data is used. Thus, 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. For example:

```
interface bus_in(int 1) a() with
                               {data = {"P1"}};

par
{
    x = a.in;
    y = a.in;
}
```

Even though **a.in** is assigned to both **x** and **y** on the same clock cycle, if the delay from pin 1 to the latch implementing the **x** variable is significantly different from that between pin 1 and the latch implementing the **y** variable then **x** and **y** may end up with different values. This can be seen by considering the timing of some signals.



Here, 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 latches the values of **x** and **y**, there is one clock cycle when **x** and **y** have different values.

This effect can also occur in more obscure places. For example:

```
interface bus_in(int 1) a() with
                               {data = {"P1"}};

while (a.in==1)
{
    x = x + 1;
}
```

In this example, although **a.in** 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

Handel-C compiler generates warnings for these programs but it is advised that asynchronous signals are never used in this way. The solution to this problem is to use either a **bus_latch_in** or **bus_clock_in** interface sort.

There is also a timing issue with output buses that you should be careful with when designing interface hardware. In this case, the value output 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 combinatorial logic. For example:

```
int 8 x;
int 8 y;

interface bus_out() output(x * y) with {data =
                   {"P1", "P2", "P3", "P4",
                    "P5", "P6", "P7", "P8"}};
```

Here, a multiplier contains deep logic so some of the 8 pins may change before others leading to intermediate values. It is possible to minimise this effect (although not eliminate it completely) by adding a variable before the output. This effectively adds a latch to the output. The above example then becomes:

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

interface bus_out() output(z) with {data =
                   {"P1", "P2", "P3", "P4",
                    "P5", "P6", "P7", "P8"}};

z = x * y;
```

Care must now be taken because the value of **z** must be updated whenever the value output on the bus must change.

Race conditions within the combinatorial logic can lead to glitches on output pins between clock edges. When this happens, a pin may glitch from 0 to 1 and back to zero or vice versa as signals propogate through the combinatorial logic. Adding a latch at the output in the manner described above removes these effects.

These considerations 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.

6.5.11 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 at a point where the data is changing state.

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 metastable characteristics of the devices under the conditions in which they will be used to determine whether any precautions need be taken.

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) InBus() with
            {data = {“P1”, “P2”, “P3”, “P4”}};

par
{
    while(1)
        x = InBus.in;

    while(1)
        y = x;

    {
        ......
        z = y;
    }
}
```

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.

6.6 Object Specifications

Handel-C provides the ability to add 'tags' to certain objects (variables, channels, buses, RAMs and ROMs) to control their behaviour. These tags or specifications are listed after the declaration of the object using the **with** keyword. This keyword takes one or more of the following attributes.

Specification	Possible Values	Meaning
show	0, 1	Show variable during simulation
base	2, 8, 10, 16	Print variable in specified base
infile	Any valid filename	Redirect from file
outfile	Any valid filename	Redirect to file
warn	0, 1	Disable warnings for object
speed	0, 1, 2, 3	Set buffer speed
pull	0, 1	Add pull up or pull down resistor(s)
data	Any valid pin list	Set data pins
offchip	0, 1	Set RAM/ROM to be off chip
wegate	-1, 0, 1	Asynchronous write enable signal
westart	0 to clock division -1	Asynchronous write enable signal
welength	1 to clock division	Asynchronous write enable signal
addr	Any valid pin list	Set address pins
oe	Any valid pin list	Set output enable pin(s)
we	Any valid pin list	Set write enable pin(s)
cs	Any valid pin list	Set chip select pin(s)

The previous sections in this chapter have already shown briefly how to use some of these specifications but this section details these in more detail and covers the other specifications in the table above.

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 clock_busin(int 4) InBus() with
          { pull = 1,
            data = {"P1", "P2", "P3", "P4"} };
```

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};
```

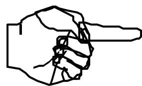
This attaches the **show** specification with a value of 0 to both **x** and **y** variables.

Details of each of the specifications is given below.

6.6.1 The 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 may be useful to avoid clutter in the output from the simulator or to just list the results from the program rather than the full list of variables at each clock step.

The default value of this specification is 1.



Reducing the number of items displayed in the output list from the simulator produces a noticeable speed up in simulation.

6.6.2 The 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 tells 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.

6.6.3 The infile and outfile Specifications

The **infile** specification may be given to **chanin**, **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 not allowed:

```
int x, y with {outfile="out.dat"};
chanout a, b with {outfile="out.dat"};
```

For details of connecting channels to files, see section 6.2.

By default, no input or output files are used.

6.6.4 The 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.

For example, when it is known that a safe parallel access is being made to a variable then adding the **warn=0** specification to the variable declaration will disable the warning that the compiler would normally generate.

6.6.5 The speed Specification

The **speed** specification may be given to an output or tri-state bus. The value that this specification is set to controls the slew rate of the output buffer for the pins on the bus. For Xilinx devices, 0 is slow and 3 is fast and the default value is 3. For Altera devices, 0 is slow and 1 is fast and the default value is 1.

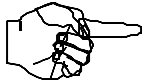
Refer to the Xilinx or Altera FPGA data sheets for details of slew rate control.

6.6.6 The pull Specification

The **pull** specification may be given to an input, output 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.

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.



Pull up and pull down resistors are not available on Altera devices.

6.6.7 The 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 **`addr`**, **`data`**, **`cs`**, **`we`** and **`oe`** specifications (see below). When set to 0, the Handel-C compiler builds the RAM or ROM on the FPGA and ignores any pins given with other specifications.

See section 6.4.3 for details of how to interface with external RAMs and ROMs.

6.6.8 The 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.

Refer to section 6.4 for further details of interfacing with asynchronous RAM devices.

6.6.9 The westart and welength Specifications

The **`westart`** and **`welength`** specifications may be given to internal or external RAM declarations. To use these specifications, you must be using the **`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.

Refer to section 6.4 for further details of interfacing with asynchronous RAM devices.

6.6.10 Specifying Pinouts

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. The specifications apply to the following objects:

Specification	Input bus	Output bus	Tri-state bus	RAM	ROM
addr				✓	✓
data	✓	✓	✓	✓	✓
we				✓	
cs				✓	✓
oe				✓	✓

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:

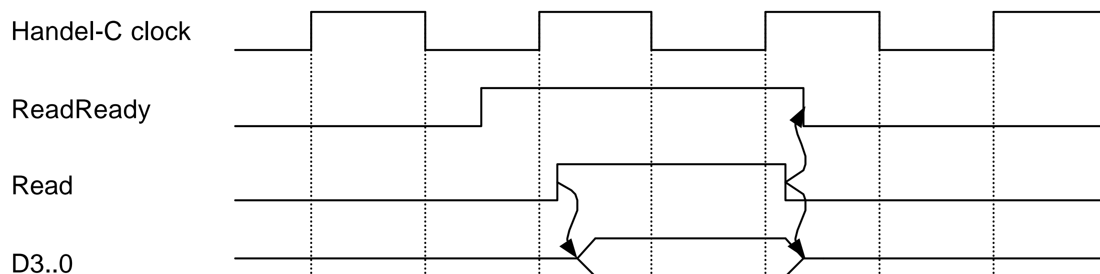
```
ram unsigned 8 ExtRAM[256] with {offchip=1,
              addr={"P1", "P2", "P3", "P4",
                    "P5", "P6", "P7", "P8"},
              data={"P9", "P10", "P11", "P12",
                    "P13", "P14", "P15", "P16"},
              we={"P17", "P18"},
              cs={"P19", "P20"},
              oe={"P21", "P22"}};
```

6.7 An Example Hardware Interface

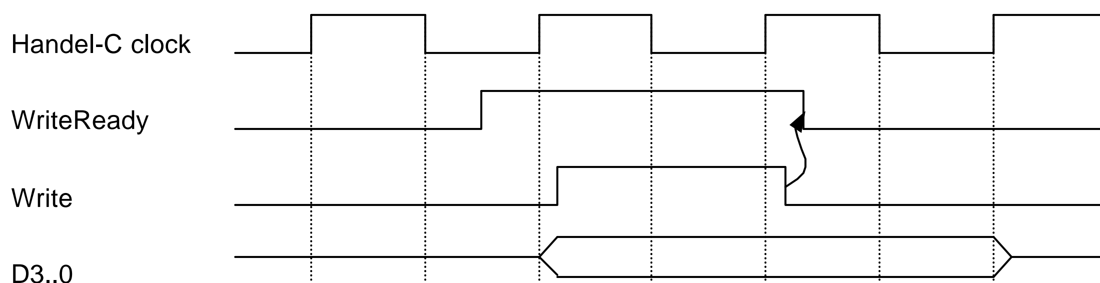
An example, theoretical interface is now described to illustrate the use of buses. The scenario is of an external device connected to the FPGA which may be read from or written to. The device has a number of signals connected to the FPGA. These are listed below:

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

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. Thus, the read cycle looks like this:



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. Thus, the write cycle looks like this:



The first stage of the code must declare the buses associated with each of the external signals. The following code does this:

```
int 4 Data;
int 1 En = 0;
interface bus_ts_clock_in(int 4)
          dataB(Reg, En==1) with
            {data = {"P1", "P2", "P3", "P4"}};

int 1 Write = 0;
interface bus_out() writeB(Write) with
              {data = {"P5"}};

int 1 Read = 0;
interface bus_out() readB(Read) with
              {data = {"P6"}};

interface bus_clock_in(int 1)
        WriteReady() with {data = {"P7"}};

interface bus_clock_in(int 1) ReadReady() with
              {data = {"P8"}};
```

Now we can change the values on the output buses by setting the values of the **Data**, **Write** and **Read** variables. In addition, we can drive the data bus with the contents of **Data** by setting **En** to 1.

Note that we have initialised the variables that drive buses to 0 so these variables must be global. This may be important when driving write strobes as we are here. 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.

The main program reads a word from the external device before writing one word back.

```
void main (void)
{
    int 4 Data;

    // Read word from external device
    while (ReadReady == 0);
    Read = 1;       // Set the read strobe
    par
    {
        Data = dataB.in; // Read the bus
        Read = 0;  // Clear the read strobe
    }

    // Write one word back to external device
    Reg = Data + 1;
    while (WriteReady == 0);
    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.

7. Standard Macro Expressions

7.1 Introduction

The Handel-C compiler is provided with a standard header file containing a collection of useful macro expressions. This header file may be used by simply including it in your Handel-C program with the following line:

```
#include <stdlib.h>
```

Note that this header file is not the same as the conventional C **stdlib.h** header file but contains a standard collection of definitions useful to the Handel-C programmer.

The following sections describe each macro in detail. Examine the **stdlib.h** header file to see the source code for the macros which also serve as additional examples of how to write macro expressions.

7.2 Constant Definitions

The **stdlib.h** header file contains the following constant definitions:

Constant Name	Definition
TRUE	1
FALSE	0

These definitions often lead to cleaner and more readable code. For example:

```
int 8 x with { show=FALSE };

while (TRUE)
{
    ......
}

if (a==TRUE)
{
    ......
}
```

7.3 Bit Manipulation Macros

The **stdlib.h** header file contains a number of macro expressions used to manipulate bits and bitfields listed below.

7.3.1 adjs

Usage: **adjs(** *Expression***,** *Width* **)**

Parameters:

Expression	Expression to adjust (must be signed integer)
Width	Width to adjust to

Returns:

Signed integer of width *Width*.

Description:

Adjusts width of signed expression up or down. Sign extends MSBs of expression when expanding width. Drops MSBs of expression when reducing width.

Example:

```
int 4 x;
int 5 y;
int 6 z;

y = 15;
x = adjs(y, width(x)); // x = 7
y = -4;
z = adjs(y, width(z)); // z = -4
```

7.3.2 adju

Usage: `adju(` *Expression*`,` *Width* `)`

Parameters:

Expression	Expression to adjust (must be unsigned integer)
Width	Width to adjust to

Returns:

Unsigned integer of width *Width*.

Description:

Adjusts width of unsigned expression up or down. Zero pads MSBs of expression when expanding width. Drops MSBs of expression when reducing width.

Example:

```
unsigned 4 x;
unsigned 5 y;
unsigned 6 z;

y = 14;
x = adju(y, width(x)); // x = 14
z = adju(y, width(z)); // z = 14
```

7.3.3 copy

Usage: **`copy(`** *Expression***`,`** *Count* **`)`**

Parameters:

Expression	Expression to copy
Count	Number of times to copy

Returns:

Expression duplicated *Count* times.
Returned expression is of same type as *Expression.*
Returned width is *Count* * **`width(`***Expression***`)`**.

Description:

Duplicates a bit field multiple times.

Example:

```
unsigned 32 x;
unsigned 4 y;

y = 0xA;
x = copy(y, 8); // x = 0xAAAAAAAA
```

7.3.4 lmo

Usage: `lmo(` *Expression* `)`

Parameters:

Expression Expression to calculate left most one of.

Returns:

Bit position of left most one in *Expression* or `width(`*Expression*`)` if *Expression* is zero. Return value is $\log_2$(`width(`*Expression*`))`+1 bits wide.

Description:

Finds the position of the most significant 1 bit in an expression.

Example:

```
int 4 x;
int 3 y;

x = 3;
y = lmo(x); // y = 1
x = 0;
y = lmo(x); // y = 4;
```

7.3.5 lmz

Usage: `lmz(` *Expression* `)`

Parameters:

Expression Expression to calculate left most zero of.

Returns:

Bit position of left most zero in *Expression* or `width(`*Expression*`)` if *Expression* is all ones. Return value is $\log_2$(`width(`*Expression*`)`)+1 bits wide.

Description:

Finds the position of the most significant 0 bit in an expression.

Example:

```
int 4 x;
int 3 y;

x = 3;
y = lmz(x); // y = 2
x = 15;
y = lmz(x); // y = 4;
```

7.3.6 population

Usage: `population(` *Expression* `)`

Parameters:

Expression Expression to calculate population of.

Returns:

Value of same type as *Expression*.

Description:

Counts the number of 1 bits (population) in *Expression*.

Example:

```
int 4 x;
int 3 y;

x = 0b1011;
y = population(x); // y = 3
```

7.3.7 rmo

Usage: `rmo(` *Expression* `)`

Parameters:

Expression Expression to calculate right most one of.

Returns:

Bit position of right most one in *Expression* or `width(`*Expression*`)` if *Expression* is zero. Return value is $\log_2$(`width(`*Expression*`)`)+1 bits wide.

Description:

Finds the position of the least significant 1 bit in an expression.

Example:

```
int 4 x;
int 3 y;

x = 3;
y = rmo(x); // y = 0
x = 0;
y = rmo(x); // y = 4;
```

7.3.8 rmz

Usage: **`rmz(`** *Expression* **`)`**

Parameters:

Expression Expression to calculate right most zero of.

Returns:

Bit position of right most zero in *Expression* or **`width(`***Expression***`)`** if *Expression* is all ones. Return value is $\log_2$(**`width(`***Expression***`)`**)+1 bits wide.

Description:

Finds the position of the least significant 0 bit in an expression.

Example:

```
int 4 x;
int 3 y;

x = 3;
y = rmz(x); // y = 2
x = 15;
y = rmz(x); // y = 4;
```

7.3.9 top

Usage: `top(` *Expression* `,` *Width* `)`

Parameters:

Expression	Expression to extract bits from.
Width	Number of bits to extract.

Returns:

Value of same type as *Expression*.

Description:

Extracts the most significant *Width* bits from an expression.

Example:

```
int 32 x;
int 8 y;

x = 0x12345678;
y = top(x, width(y)); // y = 0x12
```

7.4 Arithmetic Macros

The **stdlib.h** header file contains a number of macro expressions for mathematical calculations listed below.

7.4.1 abs

Usage: **abs(** *Expression* **)**

Parameters:

Expression Signed expression to get absolute value of.

Returns:

Signed value of same width as *Expression*.

Description:

Obtains the absolute value of an expression.

Example:

```
int 8 x;
int 8 y;

x = 34;
y = -18;
x = abs(x); // x = 34
y = abs(y); // y = 18
```

7.4.2 addsat

Usage: `addsat(` *Expression1* `,` *Expression2* `)`

Parameters:

Expression1 Unsigned operand 1.
Expression2 Unsigned operand 2. Must be of same width as *Expression1*.

Returns:

Unsigned value of same width as *Expression1* and *Expression2*.

Description:

Returns sum of *Expression1* and *Expression2*. Addition is saturated and result will not be greater than maximum value representable in the width of the result.

Example:

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

x = 34;
y = 18;
z = addsat(x, y); // z = 52
x = 34;
y = 240;
z = addsat(x, y); // z = 255
```

7.4.3 decode

Usage: `decode(` *Expression* `)`

Parameters:

Expression Unsigned operand.

Returns:

Unsigned value of width $2^{\texttt{width(}Expression\texttt{)}}$

Description:

Returns $2^{Expression}$.

Example:

```
unsigned 4 x;
unsigned 16 y;

x = 8;
y = decode(x); // y = 0b100000000
```

7.4.4 div

Usage: `div(` *Expression1* `,` *Expression2* `)`

Parameters:

Expression1	Unsigned operand 1.
Expression2	Unsigned operand 2. Must be of the same width as *Expression1*.

Returns:

Unsigned value of same width as *Expression1* and *Expression2*.

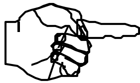
Description:

Returns integer value of *Expression1*/*Expression2*.

Example:

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

x = 56;
y = 6;
x = div(x, y); // z = 9
```



Warning! Division requires a large amount of hardware and should be avoided unless absolutely necessary. See chapter 3 for details of an alternative division routine.

7.4.5 exp2

Usage: `exp2(` *Constant* `)`

Parameters:

Constant Operand.

Returns:

Constant of width `width(`*Constant*`)+1`.

Description:

Used to calculate $2^{Constant}$. Similar to **`decode`** but may be used with constants of undefined width.

Example:

```
unsigned 4 x;
unsigned (exp2(width(x))) y; // y of width 16
```

7.4.6 incwrap

Usage: **`incwrap(`** *Expression1***`,`** *Expression2* **`)`**

Parameters:

Expression1 Operand 1.

Expression2 Operand 2. Must be of same width as *Expression1*.

Returns:

Value of same type and width as *Expression1* and *Expression2*.

Description:

Used to increment a value with wrap around at a second value. Returns *Expression1*+1 or 0 if *Expression1*+1 is equal to *Expression2*.

Example:

```
unsigned 8 x;

x = 74;
x = incwrap(x, 76); // x = 75
x = incwrap(x, 76); // x = 0
x = incwrap(x, 76); // x = 1
```

7.4.7 log2ceil

Usage: `log2ceil(` *Constant* `)`

Parameters:

Constant Operand.

Returns:

Constant value of ceiling($\log_2$(*Constant*)).

Description:

Used to calculate $\log_2$ of a number and rounds the result up. Useful to determine the width of a variable needed to contain a particular value.

Example:

```
unsigned (log2ceil(5768)) x; // x 13 bits wide
unsigned 8 y;

y = log2ceil(8); // y = 3
y = log2ceil(7); // y = 3
```

7.4.8 log2floor

Usage: `log2floor(` *Constant* `)`

Parameters:

Constant Operand.

Returns:

Constant value of floor($\log_2$(*Constant*)).

Description:

Used to calculate $\log_2$ of a number and rounds the result down.

Example:

```
unsigned 8 y;

y = log2floor(8); // y = 3
y = log2floor(7); // y = 2
```

7.4.9 mod

Usage: `mod(` *Expression1* `,` *Expression2* `)`

Parameters:

Expression1 Unsigned operand 1.
Expression2 Unsigned operand 2. Must be of the same width as *Expression1*.

Returns:

Unsigned value of same width as *Expression1* and *Expression2*.

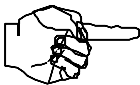
Description:

Returns remainder of *Expression1* divided by *Expression2*.

Example:

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

x = 56;
y = 6;
x = mod(x, y); // z = 2
```



Warning! Modulo arithmetic requires a large amount of hardware and should be avoided unless absolutely necessary.

7.4.10 sign

Usage: `sign(` *Expression* `)`

Parameters:

Expression Signed operand.

Returns:

Unsigned integer 1 bit wide.

Description:

Used to obtain the sign of an expression. Returns zero if *Expression* is positive or one if *Expression* is negative.

Example:

```
int 8 y;
unsigned 1 z;

y = 53;
z = sign(y); // z = 0
y = -53;
z = sign(y); // z = 1
```

7.4.11 subsat

Usage: `subsat(` *Expression1*`,` *Expression2* `)`

Parameters:

Expression1 Unsigned operand 1.
Expression2 Unsigned operand 2. Must be of same width as *Expression1*.

Returns:

Unsigned value of same width as *Expression1* and *Expression2*.

Description:

Returns difference between *Expression1* and *Expression2*. Subtraction is saturated and result will not be less than 0.

Example:

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

x = 34;
y = 18;
z = subsat(x, y); // z = 16
x = 34;
y = 240;
z = subsat(x, y); // z = 0
```

8. Porting C to Handel-C

8.1 Introduction

This chapter illustrates the general process of porting an existing conventional C routine to Handel-C. The general issues are discussed first and then illustrated with the particular example of an edge detection routine. This example illustrates the whole conversion process from conventional C program to optimised Handel-C program and also shows how to map conventional C onto real hardware.

There is also a section detailing the differences between conventional C and Handel-C.

8.2 General Porting Issues

In general, there are a number of stages to porting and mapping a conventional C program to hardware. These are:

1. Decide on how the software system maps onto the target hardware platform. For example, external RAM connected to the FPGA can be used to hold buffers used in the conventional C program. This mapping may also include partitioning the algorithm between multiple FPGAs and, hence, splitting the conventional C into multiple Handel-C programs.
2. Port conventional C to Handel-C and use the simulator to check correctness. Remember that there may be optimisations that can be made to the algorithm given that a Handel-C program is parallel. For example, you can sort numbers more efficiently in parallel by using a sorting network. This form of coarse grain parallelism can provide massive performance gains so time should be spent on this step.
3. Modify code to take advantage of extra operators available in Handel-C. For example concatenation and bit selection can be used where conventional C may use shifts and masks. Simulate again to ensure program is still correct.
4. Add fine grain parallelism such as making parallel assignments or executing individual instructions in parallel to fine tune performance. Again, simulate to ensure that the program still functions correctly.
5. Add the hardware interfaces necessary for the target architecture and map the simulator channel communications onto these interfaces. If possible, simulate to ensure mapping has been performed correctly.
6. Use the FPGA place and route tools to generate the FPGA image(s).

These steps are obviously guidelines only - some of the stages may not be relevant to your design or you may require extra stages if your design does not fit this example flow. This list provides a starting point and guidelines for how to approach the process of porting your code that is now illustrated with a full example.

8.3 Comparison Between Conventional C and Handel-C

This section details the types, operators, and statements available in conventional C and Handel-C. These tables should be used to get an idea of which parts of your conventional C program need to be altered.

8.3.1 Types, Type Operators and Objects

In Both	In Conventional C Only	In Handel-C Only
int	double	chan
unsigned	float	ram
char	enum	rom
long	register	chanin
short	static	chanout
	extern	undefined
	struct	interface
	volatile	
	void	
	const	
	union	

8.3.2 Statements

In Both	In Conventional C Only	In Handel-C Only
{;}	continue	par
switch	return	delay
do ... while	goto	?
while	typedef	!
if ... else		prialt
for (;;)		
break		

8.3.3 Expressions

<table>
<tr><th>In Both</th><th>In Conventional C Only</th><th>In Handel-C Only</th></tr>
<tr><td>-</td><td>-></td><td>select(...)</td></tr>
<tr><td>+</td><td>.</td><td>width(...)</td></tr>
<tr><td>* (multiplication)</td><td>* (pointer indirection)</td><td>@</td></tr>
<tr><td><<</td><td>& (address of)</td><td>\\</td></tr>
<tr><td>>></td><td>sizeof</td><td><-</td></tr>
<tr><td>></td><td>/ (for variables)</td><td>[:]</td></tr>
<tr><td><</td><td>% (for variables)</td><td></td></tr>
<tr><td>>=</td><td>/=</td><td></td></tr>
<tr><td><=</td><td>%=</td><td></td></tr>
<tr><td>==</td><td></td><td></td></tr>
<tr><td>!=</td><td></td><td></td></tr>
<tr><td>& (bitwise and)</td><td></td><td></td></tr>
<tr><td>^</td><td></td><td></td></tr>
<tr><td>|</td><td></td><td></td></tr>
<tr><td>? :</td><td></td><td></td></tr>
<tr><td>[]</td><td></td><td></td></tr>
<tr><td>!</td><td></td><td></td></tr>
<tr><td>&&</td><td></td><td></td></tr>
<tr><td>||</td><td></td><td></td></tr>
<tr><td>~</td><td></td><td></td></tr>
</table>

Note that `%` and `/` are provided in Handel-C for compile time constants only.

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 stand alone statements and not in the middle of more complex expressions. See section 2.5 for further details on expressions and side effects.

<table>
<tr><th>In Both (with restrictions)</th></tr>
<tr><td>=</td></tr>
<tr><td>+=</td></tr>
<tr><td>-=</td></tr>
<tr><td>*=</td></tr>
<tr><td><<=</td></tr>
<tr><td>>>=</td></tr>
<tr><td>&=</td></tr>
<tr><td>|=</td></tr>
<tr><td>^=</td></tr>
<tr><td>++</td></tr>
<tr><td>--</td></tr>
</table>

8.4 Porting Example - An Edge Detector

The example used in this section to illustrate the porting process is that of a simple edge detector. Each of the stages outlined in the previous section is illustrated with complete code listings.

8.4.1 The Original Program

The original conventional C program is given below.

```
#include <stdio.h>
#include <stdlib.h>

#define WIDTH 256
#define HEIGHT 256
#define THRESHOLD 16

void edge_detect(unsigned char *Source, unsigned char *Dest)
{
    int x, y;

    for (y=1; y<HEIGHT; y++)
        for (x=1; x<WIDTH; x++)
        {
            if (abs(Source[x + y*WIDTH] -
                        Source[x-1 + y*WIDTH])>THRESHOLD ||
                abs(Source[x + y*WIDTH]-
                        Source[x + (y-1)*WIDTH])>THRESHOLD)
                Dest[x + y*WIDTH]=0xFF;
            else
                Dest[x + y*WIDTH]=0;
        }
}

main()
{
    unsigned char *Source = malloc(WIDTH*HEIGHT);
    unsigned char *Dest = malloc(WIDTH*HEIGHT);
    FILE *FilePtr;

    FilePtr = fopen("Source.raw", "rb");
    fread(Source, sizeof(unsigned char), WIDTH*HEIGHT, FilePtr);
    fclose(FilePtr);

    edge_detect(Source, Dest);

    FilePtr = fopen("Dest.raw", "wb");
    fwrite(Dest, sizeof(unsigned char), WIDTH*HEIGHT, FilePtr);
    fclose(FilePtr);

    return 0;
}
```

The file reads data from a raw data file into a buffer. The macro procedure **`edge_detect`** then performs a simple edge detection and stores the results in a second buffer which is stored in a second file.

The edge detection is performed by simply subtracting the pixel values for adjacent horizontal and vertical pixels, taking the absolute values and thresholding the result. The source and destination images are both 8 bit per pixel greyscale images.

The C source file and a compiled version are provided on the Handel-C compiler disk along with an example image. You should run the program now to see the results of the program. This is done by:

1. Converting the example BMP file to raw data with the **`bmp2raw`** utility. See the Handel-C Compiler Reference Manual for details of this utility but you can convert the example image by typing:

   ```
   bmp2raw -b source.bmp source.raw 8bppdest.rgb
   ```

2. Executing the conventional C edge detector by typing:

   ```
   edge_c
   ```

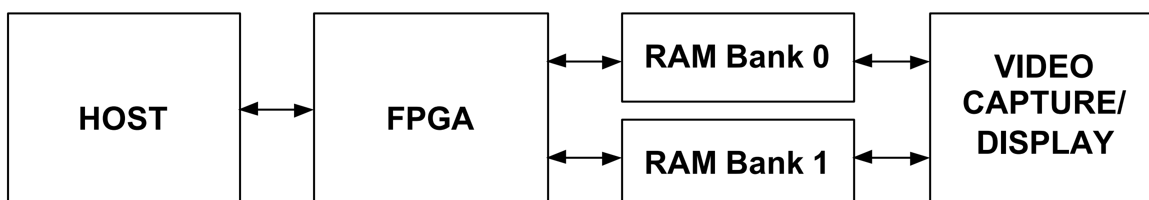
3. Converting the output from the edge detector back to a BMP file using the **`raw2bmp`** utility. See the Handel-C Compiler Reference Manual for details of this utility but you can convert the example image by typing:

   ```
   raw2bmp -b 256 dest.raw dest_c.bmp 8bppsrc.rgb
   ```

You can use the standard Windows 95 and NT paint utility to display the source and destination BMP files to compare results.

8.4.2 The Target Architecture

This example targets a simple architecture outlined below.



The FPGA has two banks of external synchronous SRAM and an interface to a host microprocessor connected directly to its pins. The SRAM conforms to the standard Handel-C model outlined in chapter 6 and the host interface has the same timings as the example interface given in section 6.6.

The video capture and display module fills RAM bank 0 with the original image at address 0 and displays the results of the edge detection from RAM bank 1 starting at address 0. Thus, the FPGA must read from one RAM bank and write to the second RAM bank.

The host is used to send the frame synchronisation to the video display module.

8.4.3 Mapping to the Target Architecture

The mapping for this example is fairly obvious. The two buffers for the source and destination image map onto the banks of RAM and the edge detection processing in the `edge_detect` macro procedure maps onto the FPGA.

The hardware implementation will require extra lines to read a threshold once at the start of processing and synchronise with the capture and display. The synchronisation takes the form of one word sent from the host to indicate that a new frame is ready for processing and one word sent to the host when the processing is complete.

8.4.4 First Attempt Handel-C Program

The first step is to port the conventional C to Handel-C with as few changes as possible to ensure that the resulting program works correctly. The file handling sections of the original program must be modified to read data from a file and write data back to a file using the Handel-C simulator as described in chapter 6. The resulting program is given below.

The following points should be noted about the port:

1. The **`Source`** and **`Dest`** buffers have been replaced with two RAMs.
2. An **`abs()`** macro expression has been used to replace the standard C function.
3. The **`x`** and **`y`** variables have been given widths equal to the number of address lines required for the RAMs to simplify the index of the RAM. Without this, each variable would have to be padded with zeros in its MSBs to avoid a width conflict when accessing the RAM.
4. Temporary variables have been used for the three pixels read from RAM to avoid the restriction on only one access per RAM per clock cycle. Without these variables, the condition for the **`if`** statement would require multiple accesses to the **`Source`** RAM.
5. The pixel values must be extended by one bit to ensure the subtract does not underflow.
6. The **`Input`** and **`Output`** channels are declared to read from and write to files. Refer to chapter 6 for details of the format of these files.

To execute the Handel-C code:

1. Convert the example BMP file to text data with the **`bmp2raw`** utility by typing:

   ```
   bmp2raw source.bmp source.dat 8bppdest.rgb
   ```

2. Simulate the Handel-C edge detector by typing:

   ```
   handelc -s edge_v1.c -ss 1000
   ```

3. Convert the output from the edge detector back to a BMP file using the **`raw2bmp`** utility by typing:

   ```
   raw2bmp 256 dest.dat dest_v1.bmp 8bppsrc.rgb
   ```

```
#define LOG2_WIDTH 8
#define WIDTH 256
#define LOG2_HEIGHT 8
#define HEIGHT 256
#define THRESHOLD 16

ram unsigned char Source[WIDTH*HEIGHT];
ram unsigned char Dest[WIDTH*HEIGHT];

macro expr abs(a) = (a<0 ? -a : a);

macro proc edge_detect()
{
    unsigned (LOG2_WIDTH+LOG2_HEIGHT) x;
    unsigned (LOG2_WIDTH+LOG2_HEIGHT) y;
    int 9 Pixel1, Pixel2, Pixel3;

    for (y=1; y<HEIGHT; y++)
    {
        for (x=1; x<WIDTH; x++)
        {
            Pixel1=(int)(0 @ Source[x + y*WIDTH]);
            Pixel2=(int)(0 @ Source[x-1 + y*WIDTH]);
            Pixel3=(int)(0 @ Source[x + (y-1)*WIDTH]);
            if (abs(Pixel1 - Pixel2) > THRESHOLD  ||
                abs(Pixel1 - Pixel3) > THRESHOLD)
            {
                Dest[x + y*WIDTH]=0xFF;
            }
            else
            {
                Dest[x + y*WIDTH]=0;
            }
        }
    }
}

void main(void)
{
    chanin unsigned Input with {infile = "Source.dat"};
    chanout unsigned Output with {outfile = "Dest.dat"};

    unsigned (LOG2_WIDTH+LOG2_HEIGHT) i;
    unsigned (LOG2_WIDTH+LOG2_HEIGHT) j;

    for (i=0; i<HEIGHT; i++)
        for (j=0; j<WIDTH; j++)
            Input ? Source[j + i*WIDTH];

    edge_detect();

    for (i=0; i<HEIGHT; i++)
        for (j=0; j<WIDTH; j++)
            Output ! Dest[j + i*WIDTH];
}
```

8.4.5 First Optimisations of the Handel-C Program

The next development stage is to change some of the operators familiar in C to operators more suitable for Handel-C.

In the above example, every time the `Source` or `Dest` RAM is accessed, a multiplication is made by the constant `WIDTH`. The Handel-C optimiser simplifies this to a shift left by 8 bits but we could easily do this by hand to reflect the hardware more accurately and reduce compilation times. We can also introduce new macros to access the RAMs given x and y co-ordinates:

```
macro expr ReadRAM(a, b) =
            ((unsigned 1)0) @
                Source[(0@a) + ((0@b) << 8)];
macro proc WriteRAM(a, b, c)
            Dest[(0@a) + ((0@b)<<8)] = c;
```

Notice how the macros pad both the result and the co-ordinate expressions with zeros. This allows us to reduce the width of the `x` and `y` counters to 8 bits each and reduces clutter in the rest of the program. This width reduction does mean that the loop conditions must be altered because `x` and `y` are no longer wide enough to hold the constant 256. Instead, we test against zero since the counters will wrap round to zero after 255.

The modified `edge_detect` macro procedure is:

```
macro proc edge_detect()
{
    unsigned LOG2_WIDTH x;
    unsigned LOG2_HEIGHT y;
    int 9 Pixel1, Pixel2, Pixel3;

    for (y=1; y!=0; y++)
    {
        for (x=1; x!=0; x++)
        {
            Pixel1=(int)ReadRAM(x, y);
            Pixel2=(int)ReadRAM(x-1, y);
            Pixel3=(int)ReadRAM(x, y-1);
            if (abs(Pixel1 - Pixel2) > THRESHOLD  ||
                abs(Pixel1 - Pixel3) > THRESHOLD)
               WriteRAM(x, y, 0xFF);
            else
               WriteRAM(x, y, 0);
        }
    }
}
```

To execute this version of the Handel-C code:

1. Simulate the Handel-C edge detector by typing:

```
handelc -s edge_v2.c -ss 1000
```

2. Convert the output from the edge detector back to a BMP file using the **raw2bmp** utility by typing:

```
raw2bmp 256 dest.dat dest_v2.bmp 8bppsrc.rgb
```

8.4.6 Adding Fine Grain Parallelism

There are two areas in this program that can be modified to improve performance. The first is to replace **for** loops with **while** loops and the second solves the problem of multiple accesses to external RAM in single clock cycles.

As described in chapter 2, the **for** loop expands into a **while** loop inside the compiler in the following way:

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

becomes:

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

This is normally not efficient for hardware implementation because the *Inc* statement is executed sequentially after the loop body when in most cases it could be executed in parallel. The solution is to expand the **for** loops by hand and use the **par** statement to execute the increment in parallel with one of the statements in the loop body.

The second optimisation concerns the three statements required to read the three pixels from external RAM. Without the restriction on multiple accesses to RAMs the loop body of the edge detector could be executed in a single cycle whereas our current program

requires four cycles, three of which access the RAM. What is needed is a modification to eliminate these RAM accesses.

By understanding that it is not possible to access the external RAM more than once in one clock cycle, we realise that the only way to improve this program is to access multiple RAMs in parallel. It should also be clear that the current program accesses most locations in the external RAM 3 times. For example, when **x** is 34 and **y** is 56 the three pixels read are at co-ordinates (34,55) , (33,56) and (34,56). The first of these is also read when **x** is 34 and **y** is 55 and when **x** is 35 and **y** is 55 whereas the second is also read when **x** is 33 and **y** is 56 and when **x** is 33 and **y** is 57. If we can devise a scheme whereby pixels are stored in two extra RAMs when they are read from the main external RAM for the first time then we could simply access these additional RAMs to get pixel values in the main loop.

The first step is to store the previous line of the image in an internal RAM on the FPGA. This allows the pixel above the current location to be read at the same time as the external RAM is accessed. The second step is to store the pixel to the left of the current location in a register. The loop body then looks something like this:

```
Pixel1 = ReadRAM(x, y);
Pixel2 = PixelLeft;
Pixel3 = LineAbove[x];

LineAbove[x] = Pixel1;
PixelLeft = Pixel1;
```

At first glance, it looks like we've made things worse by increasing the number of clock cycles but we can now add parallelism to make it look like this:

```
par
{
    Pixel1 = ReadRAM(x, y);
    Pixel2 = PixelLeft;
    Pixel3 = LineAbove[x];
}

par
{
    LineAbove[x] = Pixel1;
    PixelLeft = Pixel1;
}
```

Note the **LineAbove** RAM must be initialised at the start of the image to contain the first line of the image and the **PixelLeft** variable must be initialised at the start of each line with the left hand pixel on that line.

Since the second of these **par** statements and the **if** statement are not dependant on each other they can be executed in parallel. Putting all these modifications together gives an **edge_detect** procedure shown below.

Notice that the increment of **y** has been moved from the end of the loop to the start and the start and end values have been adjusted accordingly. This allows the increment to be executed without additional clock cycles which would have been required if it were placed at the end of the loop.

To execute this version of the Handel-C code:

1. Simulate the Handel-C edge detector by typing:

   ```
   handelc -s edge_v3.c -ss 1000
   ```

2. Convert the output from the edge detector back to a BMP file using the **raw2bmp** utility by typing:

   ```
   raw2bmp 256 dest.dat dest_v3.bmp 8bppsrc.rgb
   ```

```
macro proc edge_detect()
{
    unsigned LOG2_WIDTH x;
    unsigned LOG2_HEIGHT y;
    int 9 Pixel1, Pixel2, Pixel3, PixelLeft;
    ram LineAbove[];

    x = 1;
    while (x!=0)
    {
        par
        {
            LineAbove[x] = ReadRAM(x, (unsigned LOG2_HEIGHT)0);
            x++;
        }
    }

    y = 0;
    while (y!=255)
    {
        par
        {
            x = 1;
            PixelLeft = (int)ReadRAM((unsigned LOG2_WIDTH)0, y+1);
            y++;
        }

        while (x != 0)
        {
            par
            {
                Pixel1 = (int)ReadRAM(x, y);
                Pixel2 = PixelLeft;
                Pixel3 = (int)LineAbove[x];
            }

            par
            {
                LineAbove[x] = (unsigned)Pixel1;
                PixelLeft = Pixel1;

                if (abs(Pixel1 - Pixel2) > THRESHOLD  ||
                    abs(Pixel1 - Pixel3) > THRESHOLD)
                   WriteRAM(x, y, 0xFF);
                else
                   WriteRAM(x, y, 0);

                x++;
            }
        }
    }
}
```

8.4.7 Further Fine Grain Parallelism

We have now reduced the core loop body from five clock cycles (including the loop increment) to 2 clock cycles. Can we do any better? The answer is yes because we should be able to access the two hardware banks of RAM in parallel. Thus, the two parallel statements in the loop body could be executed simultaneously if we could organise the data flow correctly.

Now that we have realised that the two external RAM accesses can be made in parallel, we must modify the program again because the **`LineAbove`** internal RAM is accessed in both clock cycles. As it stands, parallelisation of the two statements is not permitted because it would involve two accesses to the same internal RAM in a single clock cycle. The solution is to increase the number of RAMs as before. The current line must be copied into one internal RAM while the previous line is read from a second internal RAM. The two internal RAM banks can then be swapped for the next line.

By also removing the **`Pixel1`**, **`Pixel2`** and **`Pixel3`** intermediate variables, the two statements in the loop body can be rolled into one. We use the LSB of the **`y`** coordinate to determine which line buffer to read from and which line buffer to write to. The external RAM read is done using a shared expression (**`RAMPixel`**) since we need the value from the RAM in multiple places but only want to perform the actual read once.

The new version of the edge detector is shown below. The core loop is now only one clock cycle long and is executed 255 times per line. One extra clock cycle is required per line for the initialisation of variables and 255 lines are processed. In addition, 255 cycles are required to initialise the on-chip RAM and one extra clock cycle per frame is required for variable initialisation. This gives a grand total of 65536 clock cycles per frame or an average of exactly one pixel per clock cycle. Since there is no way of getting the image into or the results out from the FPGA any faster than this we conclude that we have reached the fastest possible solution to our problem.

```
macro proc edge_detect()
{
    unsigned LOG2_WIDTH x;
    unsigned LOG2_HEIGHT y;
    int 9 PixelLeft;
    ram unsigned char LineAbove0[], LineAbove1[];

    par
    {
        x = 1;
        y = 0;
    }
    while (x!=0)
    {
        par
        {
            LineAbove0[x] = ReadRAM(x, (unsigned LOG2_HEIGHT)0)<-8;
            x++;
        }
    }
    while (y!=255)
    {
        par
        {
            x = 1;
            PixelLeft = (int)ReadRAM((unsigned LOG2_WIDTH)0, y+1);
            y++;
        }
        while (x != 0)
        {
            par
            {
                shared expr RAMPixel = (int)ReadRAM(x, y);
                shared expr PixelAbove = (int)(y[0]==0 ?
                                          0@LineAbove0[x] :
                                          0@LineAbove1[x]);
                macro expr abs(a) = (a<0 ? -a : a);

                if (y[0]==1)
                    LineAbove0[x] = (unsigned)(RAMPixel<-8);
                else
                    LineAbove1[x] = (unsigned)(RAMPixel<-8);

                PixelLeft = RAMPixel;

                if (abs(RAMPixel - PixelLeft) > THRESHOLD  ||
                    abs(RAMPixel - PixelAbove) > THRESHOLD)
                    WriteRAM(x, y, 0xFF);
                else
                    WriteRAM(x, y, 0);

                x++;
            }
        }
    }
}
```

To execute this version of the Handel-C code:

1. Simulate the Handel-C edge detector by typing:

   ```
   handelc -s edge_v4.c -ss 1000
   ```

2. Convert the output from the edge detector back to a BMP file using the **raw2bmp** utility by typing:

   ```
   raw2bmp 256 dest.dat dest_v4.bmp 8bppsrc.rgb
   ```

8.4.8 Adding the Hardware Interfaces

All that remains now that the program has been simulated correctly is to add the necessary hardware interfaces. As outlined above, the interface with the host requires the same signals and timings as the example set out in chapter 6. We now take the code from that example and produce two macro procedures - one to read a word from the host and one to write a word to the host. The suitably modified code looks like this:

```
// Read word from host
macro proc ReadWord(Reg)
{
    while (ReadReady == 0);
    Read = 1;      // Set the read strobe
    par
    {
        Reg = dataB.in; // Read the bus
        Read = 0;  // Clear the read strobe
    }
}

// Write one word back to host
macro proc WriteWord(Expr)
{
    par
    {
        while (WriteReady == 0);
        dataBOut = Expr;
    }
    par
    {
        En = 1;    // Drive the bus
        Write = 1; // Set the write strobe
    }
    Write = 0;     // Clear the write strobe
    En = 0;        // Stop driving the bus
}
```

We also need to define the pins for the external RAMs as detailed in chapter 6 and remove the RAM declarations we added to simulate the RAMs.

The main program also needs to be modified to include the code to synchronise the frame grabber with the edge detector.

The code excluding the edge detection and host interface macros is given below. Note that the pin definitions given are examples only and do not reflect the actual pins available on any particular device. In particular, even though the part is listed as a Xilinx 4010E, the pins given do not correspond to real I/O pins for that device.

```
#define LOG2_WIDTH 8
#define WIDTH 256
#define LOG2_HEIGHT 8
#define HEIGHT 256

set family = Xilinx4000E;
set clock = external "P63";
set part = "4010EPC84-3";

unsigned 8 Threshold;

// External RAM definitions/declarations
ram unsigned 8 Source[65536] 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", "P23", "P24"},
            we = {"P25"}, oe = {"P26"}, cs = {"P27"}};
ram unsigned 8 Dest[65536] with {
            offchip = 1,
            data = {"P28", "P29", "P30", "P31",
                    "P32", "P33", "P34", "P35"},
            addr = {"P36", "P37", "P38", "P39",
                    "P40", "P41", "P41", "P43",
                    "P44", "P45", "P46", "P47",
                    "P48", "P49", "P50", "P51"},
            we = {"P52"}, oe = {"P53"}, cs = {"54"}};

macro expr ReadRAM(a, b) =
            ((unsigned 1)0) @ Source[(0@a) + ((0@b) << 8)];
macro proc WriteRAM(a, b, c) Dest[(0@a) + ((0@b)<<8)] = c;

// Host bus definitions/declarations
unsigned 8 dataBOut;

int 1 En = 0;
interface bus_ts_clock_in(int 4) dataB(dataBOut, En==1) with
              {data = {"P55", "P56", "P57", "P58"}};

int 1 Write = 0;
interface bus_out() writeB(Write) with
              {data = {"P59"}};

int 1 Read = 0;
interface bus_out() readB(Read) with
              {data = {"P60"}};

interface bus_clock_in(int 1) WriteReady() with
              {data = {"P61"}};

interface bus_clock_in(int 1) ReadReady() with
              {data = {"P62"}};
```

Insert **`edge_detect`**, **`ReadWord`** *and* **`WriteWord`** *macro definitions here*

```
void main(void)
{
    ReadWord(Threshold);

    while(1)
    {
        unsigned Dummy;

        ReadWord(Dummy);
        edge_detect();
        WriteWord(Dummy);
    }
}
```

8.5 Summary

The aim of this chapter has been to show the development of a real Handel-C program from conventional C to a full program targeted at hardware. Is has also shown the performance benefits of the Handel-C approach by demonstrating a real time application executing with a great deal of parallelism.

9. Complete Language Syntax

9.1 Introduction

In this chapter, the complete Handel-C language syntax will be given in BNF-like notation.

9.2 Keywords

The identifiers below are reserved as keywords and cannot be used for any other purposes.

```
int              unsigned         undefined
while            do               ram
rom              interface        delay
for              main             switch
case             if               chan
chanin           chanout          with
else             default          break
par              void             char
short            long             set
intwidth         clock            external
internal         external_divide  internal_divide
macro            shared           expr
proc             family           part
bus_in           bus_out          bus_ts
bus_clock_in     bus_latch_in     bus_ts_clock_in
bus_ts_latch_in  width            select
prialt
```

The following character sequences are also reserved.

```
+       /*      */      //      =
;       {       }       [       ]
(       )       !       ?       :
-       ==      ++      --      <<
>>      <-      \\      @       *
!=      <       >       <=      >=
&&      ||      &       |       ^
~       +=      -=      *=      <<=
>>=     &=      |=      ^=      .
```

9.3 Complete Language Syntax

The complete language syntax is given in this section. 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.

9.3.1 Identifiers

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`**

9.3.2 Integer Literals

integer_literal ::= [**`-`**]$\{$**`1`**...**`9`**$\}^+${**`0`**...**`9`**}
| [**`-`**](**`0x`** | **`0X`**)$\{$**`0`**...**`9`** | **`A`**...**`F`** | **`a`**...**`f`**$\}^+$
| [**`-`**](**`0`**){**`0`**...**`7`**}
| [**`-`**](**`0b`** | **`0B`**)$\{$**`0`**...**`1`**$\}^+$

9.3.3 Strings

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

Here, *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	Formfeed
`\t`	9	Horizontal tab
`\n`	10	Newline
`\v`	11	Vertical tab
`\r`	13	Carriage return
`\"`	-	Double quote mark
`\0`	0	String terminator
`\\`	-	Backslash
`\'`	-	Single quote mark

9.3.4 Types

type ::= *basic_type* [*width*] | *c_type*

basic_type ::= **int** | **unsigned** [**int**]

width ::= **undefined**
| *integer_literal*
| **(***constant_expression***)**

c_type ::= **char**
| **unsigned char**
| **short**
| **unsigned short**
| **long**
| **unsigned long**

9.3.5 Hardware Control

```
hw_control ::= set clock = (internal_clock |
                            external_clock);
        |set part = string;
        |set family = family_identifier;
        |set intwidth = const_expression;

internal_clock ::= internal string
        |internal_divide  string integer_literal

external_clock ::= external pin_string
        |external_divide  string integer_literal

family_identifier ::= Xilinx3000
        |Xilinx4000
        |Xilinx4000A
        |Xilinx4000D
        |Xilinx4000H
        |Xilinx4000E
        |Xilinx4000EX
        |Xilinx4000L
        |Xilinx4000XL
        |Xilinx4000XV
        |Altera6K
        |Altera8K
        |Altera10K
```

9.3.6 Declarations

global_declaration ::= *type identifier* **=** *const_expression* **;**
  | *type identifier* {**[**[*const_expression*]**]**}+ **=**
    *array_initialiser***;**
  | *declaration*

array_initialiser ::= *array_init*
  | **{***array_initialiser* {**,** *array_initialiser* }**}**
array_init ::= **{***const_expression* {**,** *const_expression*}**}**

declaration ::= *var_declare*
  | *hw_control*
  | *array_declare*
  | *chan_declare*
  | *interface_declare*
  | *ram_rom_declare*
  | *macro_expr_declare*
  | *shared_expr_declare*
  | *macro_proc_declare*

9.3.7 Variable Declarations

var_declare ::= *type* {*identifier*} [{ **,** *identifier*}+] [*var_spec*]**;**

array_declare ::= *type* {*identifier* {**[** *const_expression* **]**} + }
  [{**,** *identifier* {**[** *const_expression* **]**}+}+]
    [*array_spec*]**;**

var_spec ::= **with** **{** *v_spec* { **,** *v_spec*} **}**
v_spec ::= {*show_spec* | *base_spec* | *file_spec* | *warn_spec* }

array_spec ::= **with** **{** *a_spec* { **,** *a_spec*} **}**
a_spec ::= {*show_spec* | *base_spec* | *warn_spec* }

9.3.8 Channel Declarations

chan_declare ::= **`chan`** *type* {*identifier*}$^+$ **`;`**
| **`chanin`** *type identifier* [*chanin_spec*]**`;`**
| **`chanout`** *type identifier* [*chanout_spec*]**`;`**

chanin_spec_file ::= **`with`** { *infile_spec* }

chanout_spec_file ::= **`with`** { *cout_spec* { **`,`** *cout_spec* } }

cout_spec ::= *show_spec* | *outfile_spec*
| *base_spec* | *warn_spec*

9.3.9 Interface Declarations

interface_declare ::= **interface** (*busin_declare*
| *lbusin_declare*
| *cbusin_declare*
| *busout_declare*
| *busts_declare*
| *lbusts_declare*
| *cbusts_declare*)**;**

busin_declare ::= **bus_in(***type***)** *identifier***()** **with** *inbus_spec*
lbusin_declare ::= **bus_latch_in(***type***)** *identifier***(***expression***)** **with** *inbus_spec*
cbusin_declare ::= **bus_clock_in(***type***)** *identifier***()** **with** *inbus_spec*

inbus_spec ::= { *data_spec* {**,** *in_spec* } }
in_spec ::= {*speed_spec* | *infile_spec* | *pull_spec*}

busout_declare ::= **bus_out()** *identifier***(***expression***)** **with** *outbus_spec*

outbus_spec ::= { *data_spec* {**,** *out_spec* } }
out_spec ::= {*speed_spec* | *show_spec* | *warn_spec* | *base_spec* | *outfile_spec* | *pull_spec*}

busts_declare ::= **bus_ts(***type***)** *identifier***(***expression***,** *expression***)** **with** *tsbus_spec*
lbusts_declare ::= **bus_ts_latch_in(***type***)** *identifier***(***expression***,** *expression, expression***)** **with** *tsbus_spec*
cbusts_declare ::= **bus_ts_clock_in(***type***)** *identifier***(***expression***,** *expression***)** **with** *tsbus_spec*

tsbus_spec ::= { *data_spec* { **,** *ts_spec*} }
ts_spec ::= *speed_spec* | *show_spec* | *base_spec* | *infile_spec* | *outfile_spec* | *pull_spec* | *warn_spec*

pin_list ::= {{*pin_string*} {**,***pin_string*}}

pin_string ::= **"P**{**1**...**9**}$^+${**0**..**9**}**"**

9.3.10 RAM and ROM Declarations

ram_rom_declare ::= *internal_ram_declare*
| *external_ram_declare*
| *rom_declare*

internal_ram_declare ::= **ram** *type ram_ident*
[**=** *ram_rom_init*]**;**
| *mult_ram_declare***;**
external_ram_declare ::= **ram** *type ram_ident* **with**
*ram_spec***;**
mult_ram_declare ::= **ram** *type* (*ram_ident* {**,***ram_ident*})

rom_declare ::= **rom** *type ram_ident* **=** *ram_rom_init*
[**with** *rom_spec*]**;**

ram_ident ::= *ident***[**[*const_expression*]**]**

ram_rom_init ::= **{***const_expression* {**,** *const_expression*}+**}**

9.3.11 Object Specifications

show_spec ::= **show** = (**1** | **0**)

base_spec ::= **base** = (**2** | **4** | **8** | **16**)

warn_spec ::= **warn** = (**1** | **0**)

speed_spec ::= **speed** **=** (**0** | **1** | **2** | **3**)

data_spec ::= **data** **=** *pin_lis*t

pull_spec ::= **pull** = (**1** | **0**)

infile_spec ::= **infile** **=** *string*

outfile_spec ::= **outfile** **=** *string*

ram_spec ::= { **offchip** **=** (**1** | **0**)**,**
[**wegate** **=** (**1** | **0**)**,**
| (**westart** **=** *const_expression***,**
welength **=** *const_expression***,**)]
data **=** *pin_list***,**
addr **=** *pin_list***,**
we **=** *pin_list***,**
oe **=** *pin_list***,**
cs **=** *pin_list* }

rom_spec ::= { **offchip** **=** (**1** | **0**)**,**
data **=** *pin_list***,**
addr **=** *pin_list***,**
we **=** *pin_list***,**
oe **=** *pin_list***,**
cs **=** *pin_list* }

9.3.12 Macro Expression Declarations

macro_expr_declare ::=
macro expr *identifier*[**(**{*identifier*}{**,***identifier*}**)**] **=**
*expression***;**

9.3.13 Shared Expression Declarations

shared_expr_declare ::=
shared expr *identifier*[({*identifier*}{ , *identifier*})] = *expression* ;

9.3.14 Macro Procedure Declarations

macro_proc_declare ::=
macro proc *identifier*[({*identifier*})] *statement*

9.3.15 Expressions

expression ::= (*expression*)
| *integer_literal*
| *variable*
| *macro_expression_ident* [({*expression*})]
| *shared_expression_ident* [({*expression*})]
| *rom_ram_entry*
| *bus_field*
| *const_expression*
| *expression* ? *expression* : *expression*
| **select** (*const_expression* , *expression* , *expression*)
| *prefix_op expression*
| **width** (*expression*)
| *expression postfix_op*
| *expression binary_op expression*
| *expression bin_const_op const_expression*

prefix_op ::= - | ! | ~ | (*type*)

postfix_op ::= [*const_expression*]
| [*const_expression* : *const_expression*]

binary_op ::= @ | + | - | * | == | != | < | >
| <= | >= | && | || | & | | | ^

bin_const_op ::= << | >> | <- | \\

const_const_op ::= `/` | `%`

variable ::= *var_identifier*
| *array_identifier* {`[`*const_expression*`]`}+

rom_ram_entry ::= *rom_ram_identifier*`[`*expression*`]`

bus_field ::= *busin_ts_ident*`.in`

const_expression ::= `(`*const_expression*`)`
| *integer_literal*
| **`width(`***expression*`)`
| **`select(`***const_expression* **`,`** *const_expression* **`,`** *const_expression*`)`
| *prefix_op const_expression*
| *const_expression postfix_op*
| *const_expression binary_op const_expression*
| *const_expression constconst_op const_expression*
| *const_expression bin_const_op const_expression*

Here, *macro_expr_ident* is an identifier of a macro expression, *shared_expr_ident* is an identifier of a shared expression, *var_identifier* is an identifier of a variable, *array_identifier* is an identifier of an array, *rom_ram_identifier* is an identifier of a ROM or RAM and *busin_ts_ident* is an identifier of an input bus or tri-state bus.

9.3.16 Statements

statement ::= {{*declaration*} {*statement*}$^+$}
| **par** {{*declaration*} *statement* {*statement*}$^+$}
| *macro_proc_ident***(**{*expression*}**)**
| *assignment*
| *channel_comms*
| *if_statement*
| *while_statement*
| *do_while_statement*
| *for_statement*
| *switch_statement*
| *prialt_statement*
| **break;**
| **delay;**

assignment ::= *variable unary_assign***;**
| (*variable* | *ram_entry)* = *expression***;**
| *variable binary_assign expression***;**

ram_entry ::= *ram_identifier***[** *expression* **]**

unary_assign ::= **++** | **--**

binary_assign ::= **+=** | **-=** | ***=** | **<<=** | **>>=** | **&=** | **|=** | **^=**

channel_comms ::= *channel_ident* **?** *variable***;**
| *channel_ident* **!** *expression***;**

if_statement ::= **if (***expression***)** *statement* [**else** *statement*]

while_statement ::= **while (***expression***)** *statement*

do_while_statement ::= **do** *statement* **while (***expression***);**

for_statement ::= **for (***statement* **;** *expression* **;**
*statement***)** *statement*

switch_statement ::= **switch (***expression***)** {
{{*switch_case*}+ *statement* [**break;**] }}

switch_case ::= **case** *const_expression* {**,***const_expression*}**:**
|**default :**

prialt_statement ::= **prialt** {{*prialt_case*}**+**}

prialt_case ::=
    case *channel_ident* **?** *variable* **:** *statement* **break;**
    | **case** *channel_ident* **!** *expression* **:** *statement* **break;**
    | **default :** *statement* **break;**

Here, *ram_identifier* is an identifier of a RAM.

9.3.17 Program

The overall syntax for the program is:

program ::= {*global_declaration*}

```
void main(void) {
        {declaration}
        {statement}+
}
```

Index

- ... 37, 47, 163, 185, 194
-- ... 25, 29, 33, 46, 85, 163, 185, 196
! ... 25, 31, 40, 46, 47, 70, 85, 162, 163, 185, 194, 196, 197
!= ... 39, 47, 163, 185, 194
#define ... 14, 68, 69, 71, 121
#include ... 14
% ... 44, 47, 163, 195
%= ... 163
& ... 41, 47, 163, 185, 194
&& ... 40, 47, 163, 185, 194
&= ... 25, 46, 85, 163, 185, 196
* ... 37, 47, 163, 185, 194
*= ... 25, 46, 85, 163, 185, 196
/ ... 44, 47, 163, 195
/*...*/ ... 14, 185
/= ... 163
// ... 14, 185
: ... 42, 47, 70, 163, 185, 194
? ... 25, 31, 42, 46, 47, 85, 162, 163, 185, 194, 196, 197
@ ... 35, 36, 38, 43, 47, 163, 185, 194
[] ... 19, 20, 21, 35, 36, 47, 163, 185, 189, 192, 194, 196
^ ... 41, 47, 163, 185, 194
^= ... 25, 46, 85, 163, 185, 196
| ... 41, 47, 163, 185, 194
|| ... 40, 47, 163, 185, 194
|= ... 25, 46, 85, 163, 185, 196
~ ... 41, 47, 163, 185, 194
+ ... 37, 47, 163, 185, 194
++ ... 25, 29, 33, 46, 85, 163, 185, 196
+= ... 25, 46, 85, 163, 185, 196
< ... 39, 47, 163, 185, 194
<- ... 35, 36, 37, 47, 163, 185, 194
<< ... 35, 47, 163, 185, 194
<<= ... 25, 46, 85, 163, 185, 196
<= ... 39, 47, 163, 185, 194
-= ... 25, 46, 85, 163, 185, 196
== ... 39, 47, 163, 185, 194
> ... 39, 47, 163, 185, 194
>= ... 39, 47, 163, 185, 194
>> ... 35, 47, 163, 185, 194
>>= ... 25, 46, 85, 163, 185, 196

abs ... 147
addition ... 37, 47
addr ... 126, 130, 193
addsat ... 148
adjs ... 138
adju ... 139
and ... 40, 41, 46, 47
argc ... 13
argv ... 13
arrays ... 19, 20, 21, 22, 34, 37, 47, 60
assignment ... 18, 24, 46, 87

base ... 126, 127, 193
binary ... 15
bit selection ... 35, 47
bitwise and ... 41, 46, 47
bitwise exclusive or ... 41, 46, 47
bitwise not ... 41, 47
bitwise or ... 41, 46, 47
block ... 7, 13, 23, 76, 82, 85
break ... 29, 30, 46, 162, 185, 196, 197
bus_clock_in ... 115, 117, 123, 125, 185, 191
bus_in ... 115, 121, 185, 191
bus_latch_in ... 115, 116, 123, 185, 191
bus_out ... 115, 117, 185, 191
bus_ts ... 115, 118, 185, 191
bus_ts_clock_in ... 115, 120, 185, 191
bus_ts_latch_in ... 115, 119, 185, 191

case ... 29, 30, 185, 196, 197
casting ... 16, 42, 43, 47
chan ... 19, 45, 53, 101, 162, 185, 190
chanin ... 45, 51, 53, 55, 58, 60, 101, 162, 185, 190
channel ... 6, 19, 20, 25, 30, 45, 46, 50, 53, 58, 60, 81, 83, 85, 101
chanout ... 45, 51, 53, 55, 58, 60, 101, 162, 185, 190
char ... 17, 45, 162, 185, 187
clock rate ... 33, 80, 91, 95, 106
combinatorial loops ... 32, 85
comments ... 14
compiling ... 52, 56, 59, 63
concatenate ... 35, 36, 47
conditional ... 26, 42, 46, 47, 70
constants ... 15, 102
continue ... 162
copy ... 140
cs ... 126, 130, 193

data ... 126, 130, 193
declarations ... 15
decode ... 149
default ... 29, 30, 185, 196, 197
delay ... 32, 46, 81, 85, 86, 162, 185, 196
design flow ... 9
div ... 150
division ... 37, 44, 47, 50, 55, 57

double 162
do...while ... 27, 46, 54, 85, 162, 185, 196
do...while loops 27, 46
drop 35, 47

efficiency 15, 33, 91
else 26, 46, 85, 162, 185, 196
enum 162
envp 13
equal 39, 47
exclusive or 41, 46, 47
exp2 151
expression 12, 24, 29, 33, 35, 37, 47, 69
extern 162
external 105, 185, 188
external_divide 105, 107, 129, 185, 188

family 104
Fibonacci 61
float 162
for 28, 29, 46, 85, 162, 185, 196
for loops 28, 46

global variables 18
goto 162
greater than 39, 47
greater than or equal 39, 47

hexadecimal 15

if 26, 46, 85, 162, 185, 196
incwrap 152
infile 102, 126, 127, 193
initialisation 18
int 45, 162, 185, 187
interface 115, 116, 117, 119, 120, 122, 125, 162, 185, 191
internal 105, 185, 188
internal_divide 105, 107, 129, 185, 188

latency 94
less than 39, 47
less than or equal 39, 47
lmo 141
lmz 142
log2ceil 153
log2floor 154
logical and 40, 47
logical not 40, 47
logical or 40, 47
long 17, 45, 162, 185, 187

macro expr 69, 71, 74, 185, 193
macro proc 76, 185, 194
macros 14, 68, 69, 71, 76
main 13, 18, 51, 53, 55, 58, 185, 197
metastability 124
mod 155
modulo arithmetic 44, 47
multi-dimensional arrays 19, 20
multiplication 37, 47, 73, 95, 123

not 40, 41, 47
not equal 39, 47

octal 15
oe 126, 130, 193
offchip 109, 126, 129, 193
or 40, 41, 46, 47
outfile 102, 126, 127, 193
overflow 15, 38

par 46, 85, 162, 185, 196
parallel 5, 13, 23, 46, 58, 87
pipelining 94
pointers 19
population 143
precedence 39
pre-processor 14, 64, 68, 71, 76, 121
prialt 30, 46, 85, 162, 185, 197
pull 126, 128, 193

queue 58, 87

ram 21, 34, 37, 45, 107, 109, 162, 185, 192
RAM 21, 32, 34, 37, 45, 46, 64, 107, 110
recursive macros 68, 71
register 162
resource conflicts 8, 32, 87
return 162
rmo 144
rmz 145
rom 21, 45, 107, 162, 185, 192
ROM 21, 34, 37, 45, 46, 61, 64, 107

scope 7
select 47, 70, 71, 163, 185, 194, 195
sequential 5, 23, 29
set clock 105, 185, 188
set family 104, 185, 188
set intwidth 18, 45, 185, 188
set part 104, 185, 188
shared expr 74, 75, 185, 194
shared macros 68, 73, 74, 75

shift left 35, 46, 47
shift right 35, 46, 47
short........................ 17, 45, 162, 185, 187
show.................................... 126, 127, 193
side effects................................. 29, 33, 46
sign... 156
simulator 45, 50, 51, 52, 53, 54, 56, 57, 59, 60, 63, 100, 101, 103, 115, 121
sizeof... 163
specifications....................................... 126
speed.................................. 126, 128, 193
statements 5, 12, 13, 23, 29, 30, 33, 46
static... 162
struct... 162
subsat... 157
subtraction..37, 47
switch........... 29, 46, 65, 85, 162, 185, 196
switch statements 29
synchronisation...6

take 35, 47
throughput.. 94
time sliced.. 6, 23
top.. 146
tri-state bus 115, 118, 119, 120
typedef.. 162

undefined.............. 16, 17, 18, 20, 25, 45, 71, 162, 185, 187
union... 162
unsigned............................ 162, 185, 187
unsigned char............................45, 187
unsigned int...................................... 45
unsigned long.............................45, 187
unsigned short...........................45, 187

variable 16, 87
void..162, 185
volatile.. 162

warn.................................... 126, 128, 193
we.. 126, 130, 193
wegate........... 108, 109, 111, 126, 129, 193
welength....... 107, 109, 111, 126, 129, 193
westart......... 107, 109, 111, 126, 129, 193
while............. 27, 28, 46, 85, 162, 185, 196
while loops 27, 46
width.................. 15, 16, 17, 24, 25, 35, 36, 37, 38, 39, 41, 43, 45, 47, 70, 71, 163, 185, 187, 195
with.............. 126, 185, 189, 190, 191, 192