`Handel-C` compiler, Release 1
Documentation and User's Manual

Matthew Aubury
Robin Watts
Ian Page

Oxford University Computing Laboratory

June 22, 1996

Contents

Foreword

This manual documents release 1 of the `Handel-C` compiler. It is organised as follows:

- Chapter 1, "Getting Started", explains how to install `Handel-C` on your machine, and how to run a simple example.
- Chapter 2, "The `Handel-C` Compiler", describes the compiler itself, and the options it takes.
- Chapter 3, "The `Handel-C` Language", describes the input language accepted by the compiler.
- Chapter 4, "Examples", describes the set of examples that come with the compiler, and details some of the important practical considerations needed when writing `Handel-C` programs.
- Chapter 5, "`Handel-C` Interfaces", describes some of the interfacing and technology specific parts of the `Handel-C` system.
- The appendices contain a language quick-reference guide, and descriptions of the error messages that can be produced by the compiler.

The `Handel-C`/ language, compiler, and associated software have been specifically developed to support the HARP reconfigurable computer system. It has found application in other contexts, in particular in embedded systems. The software has a number of features that various users have found beneficial in producing such embedded systems. However, some of the features in the current version of `Handel-C`/ are explicitly marked as being "unsupported". This means that these features will be largely undocumented, may fail to work as desired or intended, change arbitrarily, or be withdrawn completely, at any time.

Acknowledgements

We would like to acknowledge the work of everyone in the Oxford Hardware Compilation Group, and all those involved in the development of Standard ML of New Jersey, Caml light and MOSML.

Chapter 1

Getting Started

1.1 Installation instructions

This section explains how to install the `Handel-C` compiler on your machine.

1.1.1 `DOS` Version

Requirements

A PC equipped with a 80386 or better processor, at least 4MB of RAM, and running MS-DOS 3.3 or greater.

Installation

The `Handel-C` compiler is supplied as a "zip" file. Unpack this file using a recent version of the program PKUNZIP (a suitable copy is provided with the software release) into the directory you wish to use. For example,

```
C:\> a:pkunzip -d a:handelc.zip
```

The "-d" option is required for PKUNZIP to create the directory structure. The following lines must then be added to the bottom of your `autoexec.bat` file:

This section to be completed.

1.1.2 Unix version

Since there are many flavours of Unix, the installation instructions for the binary distribution may vary. The following is a general guide.

Requirements

Any Unix machine capable of running Caml Light (and hence MOSML), with at least 4Mb of real RAM, and at least 4Mb more swap.

Installation

The distribution is supplied as a tarred, gzipped file. The `tar` program should be supplied as standard with any Unix, whereas `gzip` is freeware and may be obtained from most general ftp sites. Having copied the `handelc.tgz` file to local storage, unpack the distribution as follows:

```
% zcat handelc.tgz | tar -xf -
```

This will extract the `Handel-C` directory tree to the working directory. The path to the `hcc` binary should be added to your `PATH` environment variable.

1.2 A Simple Example

To check that the compiler has be installed successfully (it will probably be necessary to reboot a DOS machine before this stage, in order to update the system variables), try typing

```
hcc -h
```

The compiler should print out a usage message and list of command line options. In order to run a simple example program, type the following at the command line:

```
hcc EXAMPLES/accu.c
```

The `Handel-C` compiler will read the Handel-C example program, compile it into hardware, optimise the hardware, and output a netlist documentation file `accu.hwp`). It then immediately enters the simulator, thus:

```
-- Handel-C Hardware Compiler, Version 161HC2
-- (c) Ian Page et al, OUCL, 1991-96

After compilation           :  35 FFs,  210 gates,   5 inverters; size  296
After netlist optimisations :  34 FFs,   97 gates,   5 inverters; size  148

Clock   State (at start of cycle)
   0    data=  0  accumulator=   0  counter= 0
   1    data=  0  accumulator=   0  counter= 0
   2  Input to STDIN (base 10):
```

After entering sixteen decimal numbers, it will ask if you are ready to accept output from the circuit on channel STDOUT. Answer with a carriage return, “y” or “yes”, and the simulator will return the sum of the sixteen numbers entered, and exit.

To get a quick start with more interesting examples, look at the documentation for the programs in the EXAMPLES directory given in chapter **??**, and use the language quick reference guide in appendix **??**.

Chapter 2

The Handel-C Compiler

2.1 Invoking the compiler

The `Handel-C` compiler can be invoked by the `hcc` command. The most basic use is simply to supply it with a `Handel-C` filename:

```
% hcc example.c
```

This will read the `Handel-C` program, compile it to hardware, perform hardware optimisations, output a `.hwp` file (and `.xnf` file or other target-specific netlist file if a hardware target is specified in the Handel-C program), and run the hardware simulator, directing input and output to the user's terminal. The system thus appears rather like a conventional compile-and-run system for any other programming language.

The compiler accepts several flags to change this default behaviour, and these can be listed by invoking the compiler with the `-h` flag:

```
% hcc -h
-- Handel-C Hardware Compiler, Beta release: H163.11
-- (c) Ian Page et al, OUCL, 1991-96

Usage: hcc [option list] filename, where an option is one of:

   -q|--quiet                 Quiet output
   -v|--verbose               Verbose output
   -vv|--very-verbose         Very verbose output
   -v[0-9]                    Set verbosity level explicitly
   -cpp|--c-preprocess        Run external C-preprocessor first
   -nr|--no-renaming          Turn off automatic renaming (aliasing)
   -ast|--add-statement-time  Add statement time information
   -ass|--add-statement-space Add statement space information
   -ald|--add-logic-depth     Add combinational logic depth information
   -nfd|--no-force-delays     Don't force minimal delays in loops
   -nwi|--no-width-inference  Turn off constant width inferencer
   -nhc|--no-health-checks    Turn off source code health checks
   -nsn|--no-sanitise         Turn off source code "sanitisation"
   -pp|--pretty-print         Pretty print source program after parsing
   -ppa|--pretty-print-after  Pretty print source program after compiling
   -O|--optimise              Turn on all major optimisations
   -O[0-9]                    Set optimisation level explicitly
```

```
-s|--simulate             Simulate after compilation
-ns|--no-simulate         Halt after compilation
-ss|--simulate-steps n    Set cycles between display for simulator
-su|--simulate-until n    Simulate for n step
-se|--simulate-extra n    Simulate for n steps after exhausting input
-if|--input-file fname    Specify an input for simulation data
-of|--output-file fname   Specify an output for simulation data
-?|-h|--version|--help    Print this message
```

These are decribed in more detail in the following sections.

2.2 General Controls

These flags control verbosity level (i.e. how much information is reported to the user at each stage of compilation).

- `-q` or `--quiet` **Reduce terminal output to a minimum**

  The quiet option causes the compiler only to output fatal error messages, or other vital user information.

- `-v` or `--verbose` **Increase amount of terminal output**

  The verbose option causes the compiler to output additional warnings and information about which stage of compilation it is involved in.

- `-vv` or `--very-verbose` **Greatly increase amount of terminal output**

  The very verbose option causes the compiler to output a great deal of detailed information about the progress of the compilation. It is particularly useful in conjunction with high optimisation levels (see next section), to view the progress of the optimisation.

- `-v` *number* **Set verbosity level explicitly**

  This option sets the verbosity explicitly. The number 0 is equivalent to the quiet option, 1 is equivalent to the default setting, 2 is equivalent to the verbose option, and 3 is equivalent to the very verbose option.

2.3 Compiler Controls

These flags control the behaviour of the compiler.

- `-cpp` or `--c-preprocess` **Run external C-preprocessor**

  On both unix and DOS, normal path searching is used to find a program called `cpp`. If found, it is run with the source file as input and generates a temporary file for the preprocessed data, which is deleted after use. If the preprocessor is not found, the compiler attempts to compile the raw source program.

- `-nr` or `--no-renaming` **Turn off automatic renaming (aliasing)**

  By default, the compiler renames variables and channels with the same name by prefixing one of them with an underscore (as many times as necessary to avoid conflicts). Clashes of this sort cause problems in generating the netlist, since signal names in the netlist are formed from variable names. This option turns off the renaming procedure.

- `-ast` or `--add-statement-time` **Add comments showing clock ticks taken by each statement**

  The pretty-printed program contains comments which show how many clock ticks will be taken by each statement in the program. When an exact number can't be determined statically, both a lower and an upper bound are given.

  This option automatically forces the `-ppa` option.

- `-ass` or `--add-statement-space` **Add space estimate comments to each statement**

  The pretty-printed program contains comments which show approximately how much hardware is generated by each statement in the program. There is a separate estimate for the number of flip-flops and the number of primitive gates. The estimate is made prior to any hardware optimisations since these optimisations explicitly do not respect statement boundaries.

  This option automatically forces the `-ppa` option.

- `-ald` or `--add-logic-depth` **Add logic-depth estimate comments to each statement**

  The pretty-printed program contains comments which show approximately the worst-case logic depth introduced by each statement. Usually, the logic depth will be determined by the expressions contained in a statement, and in particular by any multiplication, addition or subtraction operations. Note that hardware optimisation may reduce logic depth significantly.

  This option automatically forces the `-ppa` option.

- `-nfd` or `--no-force-delays` **Inhibit automatic introduction of delays in loops etc.**

  This option causes the compiler to inhibit its normal action which is to transform the program so as to introduce delays into loops and other places which might otherwise cause combinational cycles to appear in the generated hardware. Its only reasonable use is to allow you to view the pretty-printed program before these delays are added.

- `-nwi` or `--no-width-inference` **Inhibit automatic width inference phase**

  This option causes the compiler to inhibit its normal action of inferring the width of constants and variables which have not been given an explicit width (in bits) in the program. The compiler can only compile a program which has all widths made explicit, either by this automatic process or by the user.

- `-nhc` or `--no-health-checks` **Inhibit automatic source 'health checking' phase**

  This option causes the compiler to inhibit its normal action of checking scope, usage and other rules on the program. Using it will allow certain illegal programs to compile. The option has no reasonable use in normal circumstances.

- `-nsn` or `--no-sanitise` **Inhibit automatic source sanitisation**

  Performs some simple source code re-arrangement to produce more readable pretty-printed output.

- `-pp` or `--pretty-print` **Pretty print source program after parsing**

This option causes the compiler to print the source program immediately after parsing. This shows the internal representation of the program being compiled. It can sometimes be useful to determine if the compiler is interpreting your source program in the way you expected.

- `-ppa` or `--pretty-print-after` **Pretty print source program after compilation**

  This option causes the compiler to pretty-print the source program after the compilation process. This shows the final internal representation of the program after optimisation and other transformations. It is also contains comments added by the compiler to record warnings, errors and information requested by the user with the `-ast` and other compiler flags.

2.4 Optimiser Controls

These flags control the behaviour of the optimiser, in particular setting the tradeoff between the effectiveness of the optimiser and the time taken to optimise.

- `-O` or `--optimise` **Turn on all major optimisation features**

  This is a lot slower than the standard optimisations, and should only be used in the final stages of development. There is typically a 10-20% improvement in the size of circuits optimised this way over default optimisations.

- `-O` *number* **Set optimisation level explicitly**

  The default optimisation level is set at 6, all major optimisations are turned on by level 7. The features at each level are:

Optimise Level	Action
0	No optimisation
1	Peephole optimisations only
2	+ Common Subexpression Elimination
3	+ Inductive CSE for latches
7	+ Conditional Rewriting

2.5 Simulation Controls

- `-ns` or `--no-simulate` **Turn off automatic simulation**

  This option causes the compiler to terminate immediately after the compilation and optimisation stages, rather than to enter the simulator. It is useful when checking the syntax of the program, or when doing a final optimisation prior to hardware mapping.

- `-s` or `--simulate` **Simulate after compilation**

  This option causes the compiler to simulate the circuit after the compilation and optimisation stages. This is turned on by default, and thus this option is only useful in overriding a preceding `-ns` option.

- `-ss` *number* or `--simulate-steps` *number* **Set no. of cycles between display**

  This option determines the number of cycles the simulator will run for before echoing the state of each of the program variables. By default it is set to one, but this will generate a great deal of output and slow down the simulation during long runs.

- `-su` *number* or `--simulate-until` *number* **Set no. of simulation cycles**

  This option puts a hard limit on the number of cycles that the simulator will run for. By default, the simulator will run until either the program finishes or the "Stop" signal is asserted (by a `stop` statement).

- `-se` *number* or `--simulate-extra` *number* **Set number of extra cycles**

  This option causes the simulator to terminate *number* cycles after the input file (specified by the `-if` option) is exhausted. Set to zero for termination immediately after the input file is empty.

- `-if` *filename* or `--input-file` *filename* **Set input filename for simulator**

  Using this option will cause the simulator to take input from a file instead of the terminal. It behaves exactly as though the numbers were being typed, and as a result the file format is decimal numbers separated by carriage returns. An empty line is equivalent to refusing input for a cycle. Multiple input channels are not currently (correctly) supported.

- `-of` *filename* or `--output-file` *filename* **Set output filename for simulator**

  Using this option will cause the simulator to output data written to `(out)` channels to a file, rather than the terminal. It will never refuse output. The file format is again carriage return separated decimal numbers.

Chapter 3

The `Handel-C` Language

3.1 Structure of a `Handel-C` program

A `Handel-C` program is structured as follows:

```
external declarations

void main(target declaration, interface declarations)
{
     internal declarations
     statements
}
```

The section *external declarations* deals with setting up global constants (such as the widths of data buses) and of defining specification fields to describe interfaces with external RAMs etc. (e.g. specifying the pins of an FPGA chip on which a RAM is attached). The section *target declaration* is optional and allows the user to define the target technology for the circuit. This target includes information about the FPGA, the netlist format required, and board specific features such as clock pins. This section is described in chapter **??**.

The section *interface declarations* declares the communication channels between the `Handel-C` program and the outside world, and the way in which external RAMs are to be used – in the case of the `HARP` board these are the Transputer to FPGA communication channels, the two static RAM banks, and others such as the on-board LED and external interface.

Inside the body of the program, there are a set of internal declarations (which can include constants, variables, arrays, channels, on-chip RAMs and ROMs, subexpressions and procedures), followed by a sequence of statments. This is much like conventional `C`, except for the obvious difference in the set of objects, and the way procedures are declared *within* the body of `main`. Further declarations can be made inside procedure declarations or blocks of code denoted by `{` and `}`.

`Handel-C` includes a renaming scheme, so that a name may be used more than once. This is because names of certain nets in the netlist are derived directly from the names of variables and channels in the program. This is in order to aid traceability and hardware debugging. If the `Handel-C` parser finds the name of a variable, channel etc. declared for a second or subsequent time, its internal name is prepended with enough underscore characters to differentiate it from earlier incarnations.

Additionally, preprocessor directives such as `#include` and `#define` can be used, but since `Handel-C` does not incorporate a C-preprocessor of its own, such source files must be pre-processed by some third party program, such as UNIX or GNU `cpp`. `Handel-C` does, however, understand the directives introduced by any standard pre-processor and can therefore report back the location of errors in the original source files.

3.2 Declarations

This section details the types of declarations that can be made, as described in the previous section. It is important to understand how the type system differs from that of conventional `C`, and therefore this is dealt with first.

3.2.1 Values

This section describes the kinds of values that are manipulated by `Handel-C` programs.

Integer numbers

Integer constants in `Handel-C` may range over any values. However the `div` and `mod` operators can not at present work on integers outside the range -2^{30} to $2^{30}-1$, or -1073741824 to 1073741823. Note that constants may be formed directly by using large denotations, or they can also be formed by the bitwise concatenation of two or more such constants together (see section **??**).

Booleans

For convenience, the synonym `bool` is provided for single bit integers. The compiler treats objects of these types identically, so they may be used interchangably. Additionally the keywords `false` and `true` are synonyms for the constants 0 and 1, both of bit width 1.

3.2.2 Type expressions

As the compiler target is custom hardware, the conventional notion of types is extended slightly in `Handel-C`. Where in conventional programming languages we would have a single type, `Handel-C` provides a class of types each distinguished by the number of bits used to represent values. This allows the programmer to control the amount of hardware used to represent data.

Integers

Rather than just defining values to be of type `int`, `Handel-C` encourages programmers to define values which have an explicit bit-width as well. For example:

```
int x, y :  16;
int z;
```

defines `x` and `y` to be integer variables, each 16 bits wide. Integer variables can still be defined unwidthed (as in the case of `z` above), but the widths of such variables will be inferred from the context of the rest of the program at compile time. If the compiler fails to infer the width of any such construct, an error is reported.

Expressions

An expression may also be given an explicit width in order to help the width inference engine to correctly assign a width to every expression in the program. The width cast binds tightly so that brackets are usually necessary, as in the following example which says that the result of the expression should be exactly 16 bits:

```
(a+b-c) :  16
```

Note that this width cast does *not* allow an expression of a particular width to be treated as if it were another width. That job is done by the bit operations (see section ??). In fact, the example above has the effect of giving the width 16 to each one of the variables `a, b` and `c` (since they must all be equal in width) and hence the declarations of these variables need not contain this information.

3.2.3 External Declarations

Only constants and specification fields may be given outside `main`.

Constants

Constant integers or booleans may be declared with or without a width, as follows:

```
const x = 1, y = 2, z = 3;
const a = 64 :  8;
```

Giving widths to constants is useful in padding out integers and fixing problems with uninferrable widths. An expression like `x @ 0 < y @ 0` is uninferrable because no information about the width of the constants can be deduced. The fix is to replace at least one of the zeroes with a constant 0 of width one, or using the keywords `true` and `false`.

Specifications

Specifications are free-form constants used to describe interfaces. Currently in `Handel-C` they are used only to describe interfaces to RAMs, but in the future they will be used to describe the transputer interfaces, the external IO inteface, and so forth. The format is as follows:

```
const spec name = {
    name = { name, name ...  name },
    name = {},
    name = name
}
```

An example is the declaration of the “high” static RAM bank on the `HARP` board, which is written as follows:

```
const spec harp1hram =
{
   addr = { "P6", "P3", "P8", "P4", "P10", "P9", "P12", "P11",
            "P14", "P13", "P21", "P15", "P18", "P17", "P23" },
   data = { "P31", "P27", "P30", "P24", "P25", "P26", "P29", "P28",
            "P38", "P39", "P37", "P32", "P33", "P34", "P36", "P35"},
   ce   = "P22",
   wb   = {"P5", "P7"},
   en   = {"P7", "P16"}
};
```

3.2.4 Interface Declarations

Interface declarations are placed in the argument to `main(...)`. Allowable declarations are channels and external RAMs. Channel declarations are written as `chan(in)` ***name*** `:` ***width*** and `chan(out)` ***name*** `:` ***width*** for communications to and from the FPGA respectively. External RAM declarations follow the convention:

```
eram name[number of elements] = spec :  width
```

Once declared, an external RAM has an address width of $\lceil log_2 n \rceil$, where n is the number of elements declared. The *spec* field may be either the name of a `const spec` declared outside `main` (as shown in the previous section), or may be a full *spec* field, enclosed in curly braces. An example is as follows:

```
/* Include the definition of harp1l */
#include "harp.h"
const dw = 8;

void main (chan (out) stdout :  dw,
           chan (in)  stdin  :  dw,
           eram image[16384] = harp1l :  dw)
{
...  body ...
}
```

3.2.5 Internal Declarations

Internal declarations can be made immediately within the `main` block or within any curly bracketed block within the body of `main` (including procedures). Note, however, that although the compiler is capable of respecting normal scoping rules, the fact that variable names are used in the construction of signal names within the output netlist means that all variable names that are not disjoint out (but of scope) will be renamed (at present by prepending an underscore). This does not present any problem to the user, but may make the output of the simulator confusing. Objects that may be declared within blocks are constants, variables, channels, on-chip RAMs and ROMs, expressions and procedures. Constants are declared as before (in external declarations), and the remainder are declared as follows.

Variables

Ordinary variables are declared exactly as in conventional C, except for the optional width constraint. The following are all acceptable declarations:

```
int x, y;
int a, b :  8;
int p, q :  dw;
bool p;
```

The first two declarations declare two integers, whose width must be inferred from other statements in the program in order to avoid an error. The constant `dw` must have already been declared for the third declaration to be valid. Initial values may be attatched to registers, as in conventional `C`:

```
int x = 9, y = 6 :  dw;
```

This will cause the compiler to insert initialisation statements in the relevant place (and thus will cause the block to take an extra cycle to execute). The exception to this rule is at the beginning of `main`:

```
void main() {
   int x = 0 :  8;
   ...
```

The initialisation to zero will not generate an extra statement since all registers are reset to zero at the entry point by a global reset.

Channels

Channels are declared similarly to those in the interface declarations. The following are all valid channel declarations:

```
chan x, y;
chan p, q :  8;
chan int z :  dw;
chan bool a, b;
```

Unlike external channel declarations, internal channel declarations do not have directions attached, since both output and input statements must be within the scope of the declaration. Note that there may be multiple output and multiple input statements for any one channel. The compiler warns if any single process uses the same channel for both input and output, or if more than one parallel process uses the same channel for either input or output. In typical usage, exactly one parallel process will use an internal channel for output, and exactly one pocess will use the same channel for input.

On-chip RAMs and ROMs

On-chip RAMs and ROMs are declared as follows:

```
ram x[n] : dw;
ram int z[4] : 16;
ram bool q[20];
rom a = { 3, 4, 5, 6 } : 8;
rom bool b = { 0, 1, 1, 0 };
```

Rom declarations contain the information to be held within the ROM as a comma separated list: the size of the address bus of the rom will be $\lceil \log_2 n \rceil$, where n is the number of elements declared as being in the ROM. If a larger value is required the ROM must be padded with null elements (e.g. zero).

The cost of implementation of RAMs and ROMs is highly dependant on the target architecture: the Xilinx 3000 series requires latches to implement memories, whereas the Xilinx 4000 has dedicated on-chip RAMs available. The most important aspect of ROMs and RAMs is that only one element may be accessed in any clock cycle.

Expressions

Expression declarations allow the construction of explicit subexpressions. Since hardware is generated for every expression, if an expression (or any subexpression) is to be used more than once it is advisable to declare it separately in order to save hardware (the optimiser does a certain amount of common subexpression elimination, but this may not always be effective). The following are valid subexpression declarations:

```
int p() = q + r : 8;
int bob() = ((x <- 3) + (y >> 1)) @ multiplier;
```

In an expression declaration every term must already have been declared. The name given to the expression can be used in any further expression declaration or expression in the body of the program, and will have the value of the expression declared on the right hand side. Widths may be given to expression declarations as shown in the first example. Parameters may not be passed to expressions, but the syntax (the brackets are not optional) leaves room for this feature to be added in future.

Procedures

Procedures are quite unlike those in conventional C, but in many circumstances they may be used in a similar fashion. At present, procedures are parameterless, and have no return value, so values may only be communicated by use of global variables or channels. Only one invocation of a particular procedure can be in execution at any one time, thus prohibiting recursion, mutual recursion, and multiple calls from parts of the program operating in parallel. They are declared within they body of `main` (indeed, they may even be declared within procedures themselves), as follows:

```
void name() {
    ...body of the procedure...
}
```

Neither the `void` nor the brackets are optional. Procedures are called with a statements like `foo();`, as in conventional C, but, since procedures return no values, they can not be used in expressions.

3.3 Expressions

Expressions in `Handel-C` take no time to be evaluated, and so have no bearing on the number of clock cycles a given program takes to execute. They do however affect the maximum possible clock-speed for a program. The more complex an expression, the more hardware will be involved in its evaluation and the longer it is likely to take. The clock cycle time for the entire hardware program will be set by the longest such evaluation in the whole program. Also, since they take zero clock cycles to evaluate, procedure calls and assignments are prohibited within expressions. This has the side effect that the familiar C constructs of `++` and `--` are not supported by `Handel-C`.

3.3.1 Values

There are several types of value that can be used in an expression, and constants and variables may be used without restriction.

However, there are some unusual restrictions on the use of RAMs and ROMs. Because of their architecture, RAMs and ROMs are restricted to performing operations sequentially; this doesn't fit well with the highly parallel nature of hardware.

Only one element of an on-chip RAM or ROM may be addressed in any given clock cycle, and, as a result, familiar looking statements are often disallowed, for example:

```
ram x[4];
x[1] = x[3] + 1; /* This is illegal */
```

The following statement is also disallowed:

```
ram x[4];
if (x[0] == 0) x[1] = 1; /* This is illegal */
```

This is because the expression evaluation and the assignment take place in the same clock period. This problem can be avoided by placing a `delay` statement before the assignment (and in some cases, putting a delay in the else part). This problem will also crop up with other constructs such as `while`.

The same problems occurs when using external RAMs. Clearly only one external RAM access may be made per cycle (per external RAM), and these may take place on either the left or right hand side of an assignment, in channel communications, in `switch` statements and in `if` statements. They may not occur in loop controlling conditions, or in `prialts`.

The final type of value is `any`. This can be used in channel communications to indicate that a channel read should occur but that the value should be discarded, or that a channel write should occur but its value will be undefined. Its value in expressions is also undefined.

3.3.2 Arithmetic operators

The following arithmetic operators are defined in `Handel-C`:

`+ - * .*.`

The first three have the usual meanings of addition, subtraction and multiplication. The `.*.` operator designates unsigned multiplication. `+` and `-` require both operand expressions to deliver results of the same width. Any attempt to add two expressions with differing widths will cause a compilation error. The expression returned has the same number of bits as the original two expressions. Any bits that overflow during the calculation are lost.

For instance:

```
const x = 10 :  4;
const y = 8 :  4;

z = x + y;
```

will put 2 into `z` because of overflow of the 4-bit result.

Both `*` and `.*.` will accept expressions of differing widths, say m and n bits respectively, and return an expression $n + m$ bits wide. For instance, in the following code fragment, `z` must be 7 bits wide.

```
const x = 8 :  4;
const y = 5 :  3;

z = x .*.  y;
```

There is no runtime division operator provided by `Handel-C`, but `div` and several other operators are provided for use only when describing constants.

`a mod b`	a modulo b
`a div b`	$\sqcup(a/b)$
`log2 (a)`	$\sqcup(\log_2 a)$
`exp2 (a)`	2^a

These operators will raise an error if used in non-constant expressions. Some examples of use are:

```
const x = 10 :  4;
const y = 7 :  3;
const z = x div y;
const w = 10 mod 7 :  log2(z);
```

3.3.3 Relational operators

The following relational operators are defined in `Handel-C` for signed and unsigned numbers:

Operator	Signed	Unsigned
Equal	`==`	`==`
Not Equal	`!=`	`!=`
Less Than	`<`	`.<.`
Greater Than	`>`	`.>.`
Less Than or Equal	`<=`	`.<=.`
Greater Than or Equal	`>=`	`.>=.`

In all cases, both sides must have the same bit width. The result of any of these comparisons is a single bit with 0 representing false, and 1 representing true, as usual. `Handel-C` does not possess the conventional `&&` and `||` operators, instead, results of relational comparisons may be composed using the bitwise `&` and `|` operators.

3.3.4 Logical Operators

The following logical operators are defined in `Handel-C`:

`& | ^ ~`

These have the meanings of and, or, exclusive-or and not respectively. Note that "not" is written with `~` rather than `!` to differentiate it from channel communication. All of the binary operators require both the left and right hand side of the expression to be of the same bit width, otherwise an error is generated.

3.3.5 Bit Operators

The following bit operators are defined in `Handel-C`:

`<< >> <- \\ @ .`

Left and right logical shifts are implemented in `Handel-C` using `<<` and `>>` respectively. The bit width of the result is the same as the width of the number being shifted, the additional spaces being padded with zeroes.

The amount that a value is shifted by must be a constant, since shifts by variable amounts are not implemented (constant shifts require no hardware, but variable shifts would require a great deal). The compiler cannot infer widths through bit shifts, and so a reasonable program may give an "uninferable width" error. Correct this by declaring the width of each identifier and constant in the expression being shifted.

`Handel-C` contains special constructors for taking or dropping numbers of bits from an expression. For an n bit expression `e`, `(e <-` m`)` will return an expression formed from the least significant m bits of `e`. Similarly, `(e \\`m`)` will return an $n - m$ bit expression formed from all but the m least significant bits of `e`. m must be a constant expression.

Two expressions can be bitwise concatenated to form a wider expression. For an m bit expression `e` and an n bit expression `f`, `(e @ f)` will return an $m + n$ bit expression which has `e` as the most significant bits, and `f` as the least significant bits[1].

---

[1] Note that this is the opposite sense from `Handel-AS` and from earlier versions of `Handel-C`

An expression can be subscripted by constants to extract one or more bits from its representation:

```
e.5
e.(5..7)
```

In the first example, the value denoted is that of bit 5 in the representation of the value of `e` (bit 0 being the least significant). In the second example, it is the 3-bit field consisting of bits 5 through 7 of `e`.

3.3.6 Conditional Operators

Two conditional operators exist in `Handel-C`. One is very similar to the C conditional except that all its elements must be expressions:

```
max = (a > b ?  a :  b);
```

The first expression must always be a boolean. In this example, the boolean expression (`a > b`) is evaluated and the value of the whole conditional is the value of the second expression (here `a`) if the boolean evaluates true, otherwise it is the value of the third expression (here `b`). As with all expressions in `Handel-C`, the whole conditional expression evaluates within one clock cycle. Both of the selectable expressions must have the same width.

The second form of conditional expression is a generalisation of the first and can have as many alternative as needed. It necessarily has a different syntax from the first form in order to remove parsing ambiguities:

```
cond (e, c1->e1, c2->e2, c3->e3, ..., default->d);
```

In this form, each of the `ci` are constant expressions which must each be distinct, and the expressions `ei` are width compatible expressions. The expression `e` is evaluated and if its value matches any one of the `ci`, then the value of the whole conditional is the value of the corresponding `ei`. If no value matches, then the value of the conditional is given by the mandatory default expression `d`.

3.3.7 Built-in Functions

There are a number of built-in functions in `Handel-C`.In common with all `Handel-C` functions they actually designate expressions, so that they all evaluate within one clock cycle. In the following, $\#e$ designates the width of expression e.

Function	Returns	Width
`abs (e)`	absolute value of expression e	$\#e$
`rxrdy(c)`	true if channel `c` is ready to receive	1
`txrdy(c)`	true if channel `c` is ready to transmit	1
`exp2 (e)`	2^e	$2^{\#e}$

3.4 Statements

As `Handel-C` is targetted onto custom hardware, the programmer will be highly aware of both the time and space requirements of the program. Consequently, it makes sense to discuss the timings of statements in the language. Each statement takes a number of clock cycles to execute. Exactly how many clock cycles each instruction requires to execute can depend on the values of variables, or, in the case of channel communication, on when another process is prepared to cooperate.

In addition, it is extremely important to make as much use as possible of parallelism, since it is essentially free in hardware. There are two statement constructors available to this end; namely parallel assignment and parallel composition. Parallel assignment has the same behaviour as in other languages where it is supported. Parallel composition is similar to `occam`. By default, a curly bracked block of statements is composed in sequence. By prefixing the block with `par` the statements in the block will be executed in parallel. The semicolon, sometimes regarded as a sequential composition operator, instead has the meaning of statement terminator in `Handel-C`.

3.4.1 Skip

The program statement `skip` in `Handel-C` corresponds to a statement to do nothing, and to take no time doing it. It may be useful for statements such as `prialts`, to denote an empty input guard.

Timings

A `skip` statement does nothing and takes no time to execute.

However there are a (very) few places in the language where a statement *must* take at least one clock cycle to execute, such as the body of a loop. This is a consequence of the design decision that the `while` loop should execute in exactly the time taken by the sum of the actual execution times of the loop body (even if there are zero of them).

If it were permitted for the body of the loop also to execute in zero cycles, that would imply that the timing semantics of `Handel-C` were unimplementable (and the compiler would end up producing a combinational loop in the control logic). Hence such programs as

```
while (1)
   if (e) skip; else a=b;
```

are syntactically banned in `Handel-C` because there is the potential for the loop body to execute in zero cycles.

However, the compiler will, by default, automagically convert such `skip` statements to `delay` statements in order that the program can be compiled. The user is always warned of these conversions, since it may, in a few cases, be better to convert them by hand rather than to allow the compiler to judge which particular conversions to make.

3.4.2 Stop

The program statement `stop` in `Handel-C` terminates execution of the program immediately. Execution of the entire program will stop as soon as this statement is executed (this is true only in simulation – in hardware any process running in parallel with `stop` will continue to run).

Timings

A `stop` statement takes essentially infinite time to execute.

3.4.3 Delay

The program statement `delay` in `Handel-C` corresponds to a statement to do nothing, but to take exactly one cycle to do it. This is useful to avoid resource conflicts (for instance, when using RAMs and ROMs), amongst other things. The statement `delay n` is also supported where `n` is an non-negative integer constant and the corresponding delay is `n` clock cycles.

Timings

A `delay` statement takes 1 clock cycle.

3.4.4 Parallel Composition

The `par` statement composes a series of statments in parallel. All the statements will commence execution at the same instant, and the `par` statement will terminate execution when all branches have terminated. The format is as follows:

```
par {
     statement;
     ⋮
     statement;
}
```

To understand exactly what gets done in parallel, an example is necessary:

```
int x, y;

par {
    x = 1;
    { y = 1; x = x + 1; }
}
```

In this program, on the first cycle the statments `x = 1;` and `y = 1;` will be executed in parallel. The first branch will then terminate. On the second cycle, the statement `x = x + 1;` will be executed, and this branch will then terminate, and as a result the `par` will terminate. On the next cycle, any statements following the `par` will then be executed.

This sort of program will generate a warning, since the variable `x` is used in both branches. However, its usage is safe since it is never assigned to by both branches simulatanously.

Timings

A par statement takes exactly as many cycles to execute as its longest branch. There is no overhead in its execution.

3.4.5 Assignment

Assignments in `Handel-C` take the form:

variable {`,` *variable*} `=` *expr* {`,` *expr*}

When this statement is executed, all the expressions on the right hand side are evaluated, and then all the assignments are performed. Thus the final value of `x` in the following code fragment is 4, not 5.

```
y = 2;
y, x = y + 1, y + 2;
```

Assigments involving the same variable more than once on the left hand side are in error, as are assignments from/to objects of differing widths.

Timings

Every assignment statement takes one clock cycle to execute.

3.4.6 Channel Communication

Channel communication is identical for internal channels and communication across the FPGA interface. Reading from a channel is as follows:

channel `?` *variable*

Where the variable may also be an array element or memory element. Writing is as follows:

channel `!` *expression*

where there are no restrictions on the expression. In both cases the width of the channel must be the same as the width of the variable or expression.

No two statements may simultaneously write to a channel. Simultaneous reads are allowed, but should only be used where the timing of the communication is precisely known, otherwise it is possible that two reads may occur when only one was intended.

Timings

Both reads and writes enter a waiting state until the converse operation is performed elsewhere on the channel. This will continue for as many cycles as necessary. If both reader and writer are ready, the communication occurs in precisely one cycle (the time taken to update the variable).

3.4.7 Conditional

`Handel-C` provides the standard `C` conditional execution constructor, `if`, as below:

```
if (bool)
    statement
else
    statement
```

As usual in `C` the `else` clause can be omitted if not required. For example:

```
if (x == 1) {
   x = x + 1;
   x = x * 2;
}
```

Timings

The `if` statement takes as long to execute as whichever of its component statements does (i.e. there is no overhead for the `if` itself).

3.4.8 While loops

While loops are provided exactly as in `C`.

```
while (expr)
      statement
```

The contents of a while loop may be executed zero or more times.

Timings

A `while` takes no time over and above the time taken to execute the loop body as many times as required.

3.4.9 Do . . . While loops

Do . . . While loops are provided exactly as in `C`.

```
do
    statement
while (expr);
```

Unlike `while` and `for` loops, the bodies of `do ...while` loops are always executed at least once.

Timings

`do ...while` loops incur no overhead, and so the execution time is a product of the time taken to execute the statement and the number of interations (which is a minimum of 1).

3.4.10 `for` loops

The `for` statment is supplied similar to conventional `C`.

```
for (initialisation ; test ; iteration)
    loop body
```

where *initialisation* is a simultaneous assignment or other statement which sets up the variables before the first execution of the loop, *test* is a boolean expression which must evaluate to `true` for execution of the loop to continue, and *iteration* is a simultaneous assignment or other statement which adjusts these variables for the next iteration of the loop. The only difference to conventional `C` is that multiple comma separated statements are not allowed in the *initialisation* and *iteration* positions and the *test* can only be a boolean expression.

A typical code fragment might look like this:

```
for (x = 0 ; x < y ; x = x + 1) {
    out ! x;
    in ? z;
}
```

A `for` loop may be executed zero or more times according to the results of the test. Each of the *initialisation*, *test* and *iteration* statements are optional, and can be omitted if not required. There is a direct correspondence between `for` and `while` loops.

```
for (init; test; inc)
    body
```

is equivalent to:

```
{
    init;
    while (test) {
          body; inc;
    }
}
```

Timings

The exact number of cycles a `for` command executes in is determined by the behaviour of its component statements. Initialisation takes as long as the statement itself does, and there is no overhead in clock cycles for the test. The body of the loop takes as many clock cycles as the statements in the body plus the increment statement.

Thus, a `for` statement with initialisation, iteration, and increment statements all taking constant time, will take $i + n * (m + c)$ cycles to complete, where i is the number of cycles required for initialisation, n is the number of times the loop body is executed, m is the number of cycles to execute the loop body, and c is the number of cycles needed to perform the increment. Any omitted statements default to skip, and hence take zero cycles to execute.

3.4.11 Case

The `case` constructor can be used as in `C` to choose actions according to the value of a variable. The syntax, however, differs significantly from that of conventional `C`.

```
case (expr) {
      int {, int ...}:
          statement;
      int {, int ...}:
          statement;
      ...
      default:
          statement;
}
```

The case expression is evaluated and checked against each of the integer constants used to label the guarded statements. The integer constants should be non-overlapping. The statement guarded by a matching constant is selected for execution. If no matches are found, the default statement is executed. If no default option is provided, `stop` is assumed.

Timings

A `switch` takes no clock cycles to determine which branch to choose, and thus terminates in as many clock cycles as the statement in the selected branch takes to terminate.

3.4.12 Prioritised Alternations

A prioritised alternation, is a program construct that allows the direction of control flow to be chosen between a set of alternatives, based upon a combination of boolean guards, and input statements.

```
prialt {
        expr $ channel input :
              statement
        expr :
              statement
        channel input :
              statement
        ...
}
```

When executed, each branch of the `prialt` is examined in turn. The first branch that has both a true boolean guard, and a channel input that can complete immediately (i.e. another process is already trying to output down the channel), will be executed. At most one branch of the `prialt` will be executed each time the `prialt` is run. If no branch has both a true boolean guard and a channel input request which can be immediately satisfied, then the `prialt` statement will wait until one does.

Either the boolean guard or the channel input (but not both) may be omitted. In the case of an omitted boolean guard, `true` is assumed, and in the case of an omitted channel input, no synchronisation with a parallel process is performed.

For example:

```
prialt {
        reading $ read ?  x :
                 write !  x;
        stalled :
                 write !  0;
        read ?  x :
                 stop;
        true :
                 delay;
}
```

Here, whenever data is being offered on the channel `read`, and `reading` evaluates to true, the input will take place, the first branch will be executed and the `prialt` will terminate. Otherwise, the construct continues to consider later branches.

Timings

A `prialt` command takes just as long as the branch it executes to complete plus the number of clock cycles initially taken for that branch to become ready for execution.

3.4.13 Procedure Calls

Procedure calls are made in the following way:

```
name();
```

No parameters may be passed, and no return value is given. Procedure calls are *statements* and therefore may not appear in expressions. Recursive and mutually recursive procedures are not allowed. The state of any local variables in a procedure is maintained between calls. It is important to avoid overlapping calls when calling the same procedure twice in parallel, since unusual results can arise (a warning will be generated by the compiler).

Timings

The time taken for a procedure call to execute is exactly as long as for the body of the procedure itself to execute, i.e there is no overhead in the call mechanism.

3.5 Language reference

The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (`like this`). Non-terminal symbols are set in italic font (*like that*). Square brackets [...] denote optional components. Curly brackets {...} denotes zero, one or several repetitions of the enclosed components. Curly brackets with a trailing plus sign $\{\ldots\}^+$ denote one or several repetitions of the enclosed components. Parentheses (...) denote grouping.

3.5.1 Lexical Conventions

Blanks

The following characters are considered as blanks: space, newline, horizontal tabulation, carriage returns, line feed and form feed. Blanks are ignored, but they separate adjacent identifiers, literals and keywords that would otherwise be confused as one single identifier, literal or keyword.

Comments

Comments are introduced by the two characters `/*`, with no intervening blanks, and terminated by the characters `*/`, with no intervening blanks. Comments are treated as blank characters. Nested comments are not handled; a lexer error will be raised if any are encountered during parsing.

```
/* This is a legal comment. */
/* This is /* NOT */ a legal comment */
```

Identifiers

$$\begin{array}{rcl} \textit{ident} & ::= & \textit{letter}\ \{\textit{letter} \mid \texttt{0}\ldots\texttt{9} \mid \texttt{\_}\} \\ \textit{letter} & ::= & \texttt{A}\ldots\texttt{Z} \mid \texttt{a}\ldots\texttt{z} \end{array}$$

Identifiers are sequences of letter, digits, and _ (the underscore character), starting with a letter. All characters in an identifier are meaningful and all identifiers are case-sensitive.

Integer literals

$$\begin{array}{rcl} \textit{integer} - \textit{literal} & ::= & [-]\{\texttt{0}\ldots\texttt{9}\}^+ \\ & \mid & [-](\texttt{0x} \mid \texttt{0X})\{\texttt{0}\ldots\texttt{9} \mid \texttt{A}\ldots\texttt{F} \mid \texttt{a}\ldots\texttt{f}\}^+ \\ & \mid & [-](\texttt{0o} \mid \texttt{0O})\{\texttt{0}\ldots\texttt{7}\}^+ \\ & \mid & [-](\texttt{0b} \mid \texttt{0B})\{\texttt{0}\ldots\texttt{1}\}^+ \end{array}$$

An integer literal is a sequence of one or more digits, optionally preceded by a minus sign. By default, integer literals are in decimal (radix 10). The following prefixes select a different radix:

`0x`, `0X`	hexadecimal (radix 16)
`0o`, `0O`	octal (radix 8)
`0b`, `0B`	binary (radix 2).

(The initial `0` is the digit zero; the `O` for octal is the letter O.)

Keywords

The identifiers below are reserved as keywords, and cannot be employed otherwise:

```
any    bool    bus    case chan   cond
const  default delay  div  do     else
eram   false   for    if   in     inout
int    led     main   mod  out    par
port   prialt  ram    rom  seq    skip
spec   stop    target true tsport void
while
```

The following character sequences are also keywords:

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

Ambiguities

Lexical ambiguities are resolved according to the “longest match” rule: when a character sequence can be decomposed into two tokens in several different ways, the decomposition retained is the one with the longest first token.

Chapter 4

Examples

In this section we detail some of the example `Handel-C` programs provided with the compiler. The first examples are demonstrations of various constructs, whilst the later ones are more realistic examples of possible programs, including a program showing the evolution from a slow, large design to a faster, smaller finished design.

4.1 Short Examples

This section contains descriptions of some of the more simple example programs provided with the `Handel-C` compiler. This are intended to demonstrate the use of most of the `Handel-C` constructs, and to be a guide to the style of programming required to build efficient hardware implementations.

4.1.1 Accumulator (`accu.c`)

The `accu.c` program, shown in figure **??** is extremely simple, but demonstrates several important points. Invoke the compiler on the program with the line:

```
% hcc EXAMPLES/accu.c
```

(Note that the slash goes the other way for DOS). The compiler should report back something like:

```
-- Handel-C Hardware Compiler, Beta release: H163.11
-- (c) Ian Page et al, OUCL, 1991-96

After compilation           :  35 FFs,  210 gates,   5 inverters; size  296
After netlist optimisations :  34 FFs,   97 gates,   5 inverters; size  148
```

And will then immediately enter the simulator, and ask for input. After entering sixteen numbers the compiler should report back the sum of those numbers. The compilation/simple optimisations/simulation sequence is the default behaviour of the `Handel-C` compiler.

Looking at the program, there are some important differences to the way we would probably write this program in conventional `C`. The first is the external parameterisation of the program by use of `const` declarations — in this case to specify the width of the input data and the number of data elements (in $\log_2$ format for reasons we shall see shortly). The size of the result can be derived as being the sum of these two values.

```
const dw = 8;          /* Width of incoming data    */
const nw = 4;          /* 2 ^ nw = Number of inputs */
const rw = dw + nw;    /* Width of the result       */

void main(chan (in)  STDIN  : dw,
          chan (out) STDOUT : rw)
{
    int data        : dw;
    int accumulator : rw;
    int counter     : nw;

    accumulator, counter = 0, 0;

    do {
        STDIN ? data;
        accumulator = accumulator + (0 @ data);
        counter = counter + 1;
    } while (counter != 0);

    STDOUT ! accumulator;
}
```

Figure 4.1: Program `accu.c`

Next we have the declaration of the body of `main`, including definitions of the input and output channels (which, for our purposes, are “connected” to the simulator). The names of the channels are unimportant, we have used `STDIN` and `STDOUT` consistently. They must be given widths (these cannot be inferred), and directions.

Next come the program variable declarations. We have three integers, all of which are given their relevant widths. This is, in fact, unnecessary since the widths of `data` and `accumulator` could be inferred from the external channels. The width of `counter`, however, could not and so this width must be specified to avoid an “Incomplete Inference” error message.

The first line of the program initialises the accumulator and counter to zero, using the new construct of parallel assignment. In this circumstance the initialisation is unnecessary since all registers are reset to zero at the beginning of execution, however this is an hand optimisation which should be deferred to the later stages of development to avoid confusion.

In a traditional `C` program, the loop body would probably have been a `for` loop, but these map relatively poorly into hardware. Instead, we have used a `do ...while` construct, rarely seen in most `C` programs. There are two good reasons for this. Firstly, in `Handel-C` loops, it is normally possible to increment the counter variable in parallel with some other update in the cycle, whereas by default the `for` construct of `Handel-C` places the increment statement sequentially after the body of the loop. Secondly, in traditional `C` we would have written our test something like `counter < 16` (or whatever 2^{nw} is in a particular instance). But we can see that the constant 16 is in fact a 5-bit unsigned constant (a 6-bit signed constant). Because the comparison operators expect equal widths on each side, this would imply that we had to make the counter wider than we actually require. In fact, the method shown is the *only* way of looping over all 2^n cases using an n-bit counter.

In the body of the loop we read data from the input channel using communications of the form (*channel* ? *variable*), then add this data into the accumulator. Note that because the accumulator is wider than the data variable we “pad” the data variable with zeroes (`data @ 0`). This will only

```
/* Integer division by long-division method */

const dw = 16;

void main(chan (in)  STDIN  : dw,
          chan (out) STDOUT : dw)
{
    int a, b, c : dw;
    int bits : 5;

    while (1) {

        STDIN ? a;
        STDIN ? b;

        c, bits = 0, 1;
        while (b .<=. a)
            b, bits = b << 1, bits + 1;

        do par {
            if (a .>=. b)
                a, c = a - b, (c << 1) ^ 1;
            else
                c = c << 1;
            b, bits = b >> 1, bits - 1;
        } while (bits != 0);

        STDOUT ! c;
    }
}
```

Figure 4.2: Program `divide.c`

work for unsigned input – signed input must be "sign extended" by repeatedly copying the most significant bit as many times as required. Finally in the loop, the counter variable is incremented – not with the traditional `++` operator, which is not provided in `Handel-C`, but with a straight incrementing assignment. This increment could be done in parallel with the accumulation, but that has not been done for clarity. The last statement in the program simply outputs the result.

Although this explanation may seem excessive for such a simple and obvious program, it covers most of the important points required for writing good `Handel-C` programs, namely parameterisation, attention to bit widths, efficient use of resources, and parallelism. The following examples will include description only of the basic algorithm and any more interesting design choices made in writing the `Handel-C` program.

4.1.2 Integer Division (`divide.c`)

This program (shown in figure **??**) 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 $\lfloor a/b \rfloor$. It works with sixteen-bit (parameterisble), unsigned values. Although not very useful by itself, the body of the program could be run in parallel with some other task, giving `Handel-C` programs access to division. In this case the channel communications would be internal to the body of `main`.

```
/* Square root without multiplication or division */

const half_dw = 8 : 4;
const dw = half_dw << 1;

const sq = (1 << (dw - 2)) : dw;

void main(chan (in)  STDIN  : dw,
          chan (out) STDOUT : dw)
{
  int a, p, q, r;
  int i;

  while (1) {

      STDIN ? a;
      p, q, r = 0, 0, sq;
      i = half_dw;

      do {
        if (p + q + r .<=. a)
          i, r, q, p = i - 1, r >> 2, (q >> 1) + r, p + q + r;
        else
          i, r, q    = i - 1, r >> 2, (q >> 1);
      } while (i != 0);

      STDOUT ! q;
  }
}
```

Figure 4.3: Program `root.c`

4.1.3 Square Root (`root.c`)

This program is very similar to the divide example, and can be used as a module in a similar way. How it manages square root without division or multiplication is left as an exercise for the reader!

4.1.4 Queue (`queue.c`)

The program `queue.c`, shown in figure **??**, implements a small (four place) queue using internal communications. This is a good example of parallel programming in `Handel-C`. There are four processes (e0 to e3), each of which engages in an infinite loop of reading data from the channel to their right, and outputting that data to the channel on their left. There are three internal channels, c1 to c3, and two external channels c0 and c4 connected to either end of the pipeline. The program can be seen to be correct (that is, to not lose or duplicate any data) because each process can hold at most one piece of data, will never accept more data until it is empty, and can never output more times than it inputs.

To observe the program's behaviour, during simulation answer 'n' to stop the program from outputing, and press return to refuse input. If no data is in the queue it will only attempt to input, else if the queue is full it will only attempt to output. Otherwise it will attempt both.

```
/* Four place queue using internal communications */

const dw = 8;

void main(chan (in)  c4 : dw,
          chan (out) c0 : dw)
{
    int d0, d1, d2, d3;
    chan c1, c2, c3;

    void e0() { while (1) { c1 ? d0; c0 ! d0; } }
    void e1() { while (1) { c2 ? d1; c1 ! d1; } }
    void e2() { while (1) { c3 ? d2; c2 ! d2; } }
    void e3() { while (1) { c4 ? d3; c3 ! d3; } }

    par {
        e0(); e1(); e2(); e3();
    }
}
```

Figure 4.4: Program `queue.c`

4.1.5 Microprocessor (`proc.c`)

When compiled this program builds a small microprocessor in hardware. This is an 8-bit microprocessor, featuring internal 16 location RAM and 16 location ROM. The program as presented fills the ROM with a fibonacci number generating program. When run, it requests a number, n, and then outputs $2n$ fibonacci numbers (this slightly odd behaviour is due to the way the assembly language program is written to fit into 16 basic instructions!).

This program introduces several features we have not used in previous examples. The first is the use of the `C` pre-processor to expand the `_asm_` macro, for assembling instructions. We also use internal RAM's and ROM's (parameterised by external constants), and explicit sub-expressions to separate the opcode and operand from an instruction. There are 10 instructions defined by default – this is a good program to experiment with since there are many possible variations on this simple set. There are two internal registers, the program counter and the instruction register, and one general purpose data register. The operation of the program should be clear – an instruction is read and the program counter incremented simultaneously, and then the instruction is decoded and the appropriate action taken. This is done using a `case` statement.

4.1.6 Edge Detector (`edge.c`)

The program shown in figure **??** is a simple grey-scale image edge detector. This program is the first we have examined that uses external RAMs. The RAMs used are nominally those of the `HARP` board, and their definitions (`harp1lram` and `harp1hram`) are taken from the included file `harp1.h`. These two 32k × 16-bit RAMs are parametrised by the definitions to become one 16k × 16-bit and one 16k × 1-bit RAMs(`img1` and `img2` respectively).

The program works as follows. The data is read in line-by-line format, and put into a pipeline of registers, `p[0]` to `p[2]`. The differential of this data is calculated and stored alongside the unprocessed image in the 16-bit RAMs. Next, the data is processed in column by column format by incrementing the address counters in the opposite order (note how the full address is generated by concatenating the x and y coordinate bit strings). This data is put through the pipeline and

Original Image Edge Dected Image

Figure 4.5: Result of program `edge.c`

differentiated in the vertical direction. The absolutes of these two differentials are added, and compared against a threshold value. If the sum of the absolutes (which is an approximation to the absolute edge intensity at a point) is higher than the threshold a single bit one is placed in the second RAM. Finally, the data is read back from the second RAM and the ones and zeroes are replaced with black (0) and white (255) and output. An example of the result of the edge detector is shown in figure **??**.

On a UNIX system, running X with the PBM toolkit installed, creating and processing and viewing images is quite easy. In order to generate the intial data it is necessary to build a 128×128 8-bit grey-scale image, and output it in 8-bit RAW PBM/PGM format (the *xv* program is most suitable for this job). An example of such an image is given in `DATA/bottle.pgm`. Having obtained this image, in needs to be converted to "dat" format, suitable for direct input to the `Handel-C` simulator. This format is simply a list of carriage return separated numbers. A utility for doing this is supplied in the `UTILS/` directory as `raw8todat`. Thus,

```
tail -c 16384 DATA/bottle.pgm >DATA/bottle.raw
UTILS/raw8todat DATA/bottle.raw DATA/bottle.dat
```

cuts the image data from the PGM file, and converts this raw data to "dat" format. The compiler can then be invoked to input this data and output the edge detected to another file:

```
cpp EXAMPLES/edge.c >EXAMPLES/edge.cpp
hcc EXAMPLES/edge.cpp -ss 1000 -if DATA/bottle.dat -of DATA/bottle\_p.dat
```

The `-if` file specifies the *input* data, `-of` file specifies the *output* data, and the `-ss` flag specifies "simulation steps" – i.e. how many cycles go by before the simulator reports the status of the variables in the program. This is useful because this program takes many cycles to run (around 50,000), and it would be tedious to read through all of these results! The compilation will generate a considerable number of warnings. This is because we are violating a number of laws about using variables in parallel, i.e. we are both using updating and using a variable in separate, parallel

```
#include "harp1.h"

const dw = 8;         /* Data width */
const iw = dw << 1;   /* Internal data width */
const xw = 7;         /* Log horizontal size */
const yw = 7;         /* Log vertical size */
const aw = xw + yw;   /* Address width */
const ae = 1 << aw;   /* Number of elements in ram */

const null = 0 : dw;  /* Null element */

const threshold = 48;
const white = (1 << dw) - 1;
const black = 0;

#define LO(x) ((x) <- dw)  /* Lo word */
#define HI(x) ((x) \\ dw)  /* Hi word */

void main (chan (in)  STDIN          : dw,
           chan (out) STDOUT         : dw,
           eram img1[ae] = harp1lram : iw,
           eram img2[ae] = harp1lram : 1)
{
    int x : xw;
    int y : yw;
    int p_0, p_1, p_2 : iw;

    int abs_diff() = abs(LO(p_2) - LO(p_0));
    int sum_abs()  = (false @ abs_diff() ) + (false @ HI(p_1));

    /* Differentiate in x direction */
    do {
        do par {
            STDIN ? any;
            p_0, p_1, p_2 = p_1, p_2, null @ STDIN;
            img1[y @ x] = abs_diff() @ LO(p_1);
            x = x + 1;
        } while (x != 0);
        y = y + 1;
    } while (y != 0);

    /* Differentiate and threshold in y direction */
    do {
        do par {
            p_0, p_1, p_2 = p_1, p_2, img1[y @ x];
            img2[y @ x] = (sum_abs() >= threshold ? 1 : 0);
            y = y + 1;
        } while (y != 0);
        x = x + 1;
    } while (x != 0);

    /* Output */
    y = 2;
    do {
        x = 2;
        do par {
            STDOUT ! (img2[y @ x] ? white : black);
            x = x + 1;
        } while (x != 2);
        y = y + 1;
    } while (y != 2);
}
```

Figure 4.6: Program `edge.c`

statements. However, because we know that each assignment takes exactly one cycle to complete, we can see that the program will produces the result we require. This issue is an important one, and one we will address more thoroughly in the developmental example.

On completion, the program will terminate having generated the new file `DATA/bottle_p.dat`. A suitable way of viewing this image is:

```
UTILS/dattoraw8 DATA/bottle\_p.dat | rawtopgm 128 128 | xv -
```

This line avoids the need for intermediate files by use of UNIX pipes. DOS equivalents for these programs are under development.

4.1.7 Merge Sort (`merge.c`)

The program `merge.c` is a `Handel-C` version of the familiar merge sort algorithm. The program is too long to list here, but a copy exists in the `EXAMPLES` directory. By default, it sorts 256 9-bit signed values. To run the program on some random data:

```
cpp EXAMPLES/merge.c >EXAMPLES/merge.cpp
hcc EXAMPLES/merge.cpp -if DATA/random.dat -of DATA/sorted.dat -ss 100
```

(the preprocessed version is also supplied in case you don't have "cpp" to hand). The sorted data is written to a file `sorted.dat` – the UNIX command `sort -n` can be used to sort the random data for comparison.

The program uses a lot of features we have seen so far, and in addition makes use of procedures. The procedure `inc_addr()` is used to increment the address counter – this is an example of using a procedure to save hardware rather than to organise the program more efficiently. The other procedures are `merge()`, which does the real work of merging sorted lists, `input_data` and `output_data` which have the obvious uses. The program starts off by treating the data as 256 lists of length one. Adjacent lists are merged together to produce 128 lists of length two, and so on, until we have one list of length 256. Two RAMs are used, with the second one acting as a temporary area for the first. This could be arrange more efficiently to save the coping process, but we have left this program in the clearer form.

As a matter of comparison to the heap sort shown next, the 256 element 9-bit merge sorted compiles to:

```
After compilation          : 140 LATCHES, 1152 GATES, 59 INVERTERS; SIZE 1703
After netlist optimisations :  98 LATCHES,  703 GATES, 41 INVERTERS; SIZE 1028
After fan-in adjustment     :  98 LATCHES,  728 GATES, 41 INVERTERS; SIZE 1028
```

And takes around 5600 cycles to complete.

4.1.8 Heap Sort (`heapsort.c`)

The program `heapsort.c` is a `Handel-C` version of the heap sort algorithm. The program is too long to list here, but a copy exists in the `EXAMPLES` directory. By default, it sorts 255 9-bit signed values. To run the program on some random data:

```
cpp EXAMPLES/heapsort.c >EXAMPLES/heapsort.cpp
hcc EXAMPLES/heapsort.cpp -if DATA/random.dat -of DATA/sorted.dat -ss 100
```

The program is similar in structure to the merge sorter. The data is read into RAM by the procedure `input_data`, from memory locations 1 to 255 (this explains why 255 not 256 elements are sorted – it makes navigating through a heap much easier). This forms a binary tree with a number at every node, the root node being element 1. This binary tree is then "heap-ordered" by the procedure `build_heap` which swaps elements to make the parent node smaller than both of its children. The two procedures for doing this are `bubble_down` and `swap_with_smaller_child`. The second of these procedures takes parameters `swsc_i` and `n` by means of global variables. These indicate the position of a parent and the number of elements in the heap respectively. Once the heap ordering is complete, the first element of the heap is the minimum. This is output, and replaced with an element taken (arbitrarily) from the end of the heap. This shortens the heap by one, and upsets the ordering. Thus the ordering is restore by another call to `bubble_down`. Now we have the second smallest element in the original list, and we repeat the process. This cycle is implemented by the procedure `output_sorted`. Since the heap is of $\log n$ depth, and we do $O(n)$ walks over the heap, the algorithm has the same asymptotic complexity, $O(n \log n)$, as merge sort.

The trade-offs between the two algorithms are apparent. Here are the results from compiling the default version:

```
After compilation           : 134 LATCHES, 950 GATES, 109 INVERTERS; SIZE 1487
After netlist optimisations :  93 LATCHES, 567 GATES,  81 INVERTERS; SIZE  866
After fan-in adjustment     :  93 LATCHES, 596 GATES,  81 INVERTERS; SIZE  866
```

With completion in around 12300 cycles. Thus heap sort uses (perhaps suprisingly) less hardware than merge sort, and also requires the use of only one RAM (being an "in-place" sorter), but takes around twice as many cycles to complete (although in both examples cycles could be saved by use of more parallelism and pipelining).

4.2 Developmental Example – Histogram Equalisation

In this section, we show how a simple example program (namely histogram equalisation) can be taken from a "first-attempt" version (slow, big, but working) to a version "hand optimisied" for speed and size. This process is important in developing `Handel-C` programs to actual hardware, since inevitably your program will compile to something just a bit too large to fit on the chip (trust me).

Histogram equalisation is a well known technique in image processing for correcting contrast and intensity variations in images. It attempts to make the cumulative histogram of intensity a triangular shape – the upshot of this is that there will be roughly the same number of any given shade of grey as any other, meaning the image had a wide spread of intensity levels. It doesn't work well when large parts of the image are intentionally dominated by a single shade.

The major processes are: generating a histogram of the image; generating a cumulative histogram from this; using the normalised, cumulative histogram as a "look up table" to transform the intensities in the image. Our first attempt follows this course strictly.

4.2.1 First Attempt

The first attempt is program `equal_v1.c`, shown in figure **??**. It contains three procedures, called in sequence. The first, `input_and_histogram`, reads the data from the input channel, stores it in the `data` RAM, and builds the histogram in the `hist` RAM. Note that a statement such as `hist[x] = hist[x] + 1` isn't allowed because it implies two memory accesses in one cycle. Next,

```
/* Histogram equalisation - version 1 */

#include "harp1.h"

const dw = 8, aw = 14;
const de = (1 << dw), ae = (1 << aw);

void main(chan(in)  STDIN           : dw,
          chan(out) STDOUT          : dw,
          eram data[ae] = harp1lram : dw,
          eram hist[de] = harp1hram : aw)
{
    int addr, accu, x, temp;

    void input_and_histogram() {
        addr = 0;
        do {
            STDIN ? x;
            data[addr] = x;
            temp = hist[x];
            hist[x] = temp + 1;
            addr = addr + 1;
        } while (addr != 0);
    }

    void accumulate_histogram() {
        x = 0;
        accu = 0;
        do {
            temp = hist[x];
            hist[x] = accu;
            accu = accu + temp;
            x = x + 1;
        } while (x != 0);
    }

    void equalise_and_output() {
        addr = 0;
        do {
            x = data[addr];
            temp = hist[x];
            STDOUT ! temp \\ (aw - dw);
            addr = addr + 1;
        } while (addr != 0);
    }

    /* Main program */
    input_and_histogram();
    accumulate_histogram();
    equalise_and_output();
}
```

Figure 4.7: Program `equal_v1.c`

`accumulate_histogram` builds the cumulative histogram, and finally `equalise_and_output` uses this as a look-up-table to transform the stored image data and output.

The results of compiling this are:

```
% cpp EXAMPLES/equal_v1.c >EXAMPLES/equal_v1.cpp
% hcc EXAMPLES/equal_v1.cpp -if DATA/rose.dat -of DATA/rose_p.dat -ss 1000

After compilation           : 102 LATCHES, 897 GATES, 26 INVERTERS; SIZE 1342
After netlist optimisations :  75 LATCHES, 536 GATES, 19 INVERTERS; SIZE  769
After fan-in adjustment     :  75 LATCHES, 555 GATES, 19 INVERTERS; SIZE  769
```

And the simulation runs for 148484 cycles. This is not suprising, since we have not used *any* parallel operators yet. However, the important point is that it produces the correct result. Now we can set about transforming the program to make it go faster (or at least, run in fewer cycles).

4.2.2 Making it Parallel

The next attempt is program `equal_v2.c`, which is shown in figure **??**. The first procedure, `input_and_histogram`, has been rearranged into two parallel sections. On the first cycle, data is read in from the input. On the second cycle, this data is simultaneously stored in the `data` RAM, whilst recalling the histogram count for the relevant level. On the third cycle, the updated histogram count is written back and the address counter is incremented.

In the procedure `accumulate_histogram` three of the assignments have been rolled into one, saving two cycles. Finally, in the `equalise_and_output` procedure, we have used indirect addressing and a `par` loop to compress the four cycles into one. Note that this indirect addressing is probably not such a good idea in practice – it will indoubtedly lead to an extremely long critical path, and so it would be more advisable to pipeline these statements.

So, using parallelism but without pipelining, the new results are:

```
% cpp EXAMPLES/equal_v2.c >EXAMPLES/equal_v2.cpp
% hcc EXAMPLES/equal_v2.cpp -if DATA/rose.dat -of DATA/rose_p.dat -ss 1000
```

(several warnings)

```
After compilation           : 103 LATCHES, 881 GATES, 27 INVERTERS; SIZE 1305
After netlist optimisations :  75 LATCHES, 525 GATES, 20 INVERTERS; SIZE  737
After fan-in adjustment     :  75 LATCHES, 545 GATES, 20 INVERTERS; SIZE  737
```

And the simulation runs for 66052 cycles. It is clear, though, that the first procedure takes one more cycle than necessary. Pipelining this gives us program `equal_v3.c`, which takes 49668 cycles to complete. We won't look at this program in any more detail since it differs only slightly. This program has one problem, however, because it reads one more value than it should (and discards it). This is a problem which does not affect simulation, but is vitally important in practice.

4.2.3 Making it smaller

There are several ways in which we can make the program smaller, and many of these have been done to give the program `equal_v4.c`, shown in figure **??**. The first is to denote common subexpressions (in particular `addr != 0`), and common substatements (such as `addr = addr + 1`) into `int`s and procedures respectively. Secondly, we can remove initialisation statements which have no effect – any "reset to zero" initialisations at the beginning of the program or following a `do ...while (...!= 0)` loop. Having done all this we get the new result:

```
/* Histogram equalisation - version 2 */

#include "harp1.h"

const dw = 8, aw = 14;
const de = (1 << dw), ae = (1 << aw);

void main(chan(in)  STDIN           : dw,
          chan(out) STDOUT          : dw,
          eram data[ae] = harp1lram : dw,
          eram hist[de] = harp1hram : aw)
{
    int addr, accu, x, temp;

    void input_and_histogram() {
        addr = 0;
        do par {
            {
                STDIN ? x;
                data[addr] = x;
                addr = addr + 1;
            }
            {
                delay;
                temp = hist[x];
                hist[x] = temp + 1;
            }
        } while (addr != 0);
    }

    void accumulate_histogram() {
        x = 0;
        accu = 0;
        do {
            temp = hist[x];
            hist[x], accu, x = accu, accu + temp, x + 1;
        } while (x != 0);
    }

    void equalise_and_output() {
        addr = 0;
        do par {
            STDOUT ! hist[data[addr]] \\ (aw - dw);
            addr = addr + 1;
        } while (addr != 0);
    }

    /* Main program */
    input_and_histogram();
    accumulate_histogram();
    equalise_and_output();
}
```

Figure 4.8: Program `equal_v2.c`

Original Image Histogram Equalised Image

Figure 4.9: Result of Histogram Equalisation

```
After compilation           : 102 LATCHES, 735 GATES, 32 INVERTERS; SIZE 1033
After netlist optimisations :  79 LATCHES, 438 GATES, 22 INVERTERS; SIZE  619
After fan-in adjustment     :  79 LATCHES, 445 GATES, 22 INVERTERS; SIZE  619
```

This saves 100 gates (with normal optimisations) – the saving would probably be less with full optimisation. This may seem to be just about as much as we can do with this program, but there will always (with any program) be many more areas in which savings can be made. It is important, when trying to reduce the size as much as possible, to consider just about every statement and expression. We can actually make more savings here, by noting that the test `addr != 0` is actually quite expensive. If instead, we replace `addr` with a register that is one bit wider, we can use the single additional bit to test for loop termination. We have to rewrite the `increment_address` statement also:

```
int extended_addr :  aw + 1;

int addr = extended_addr <- aw;

int addr_nz = (extended_addr
aw) != 1;

void increment_address() {
    extended_addr = (addr @ 0) + 1;
}
```

This ensures that the single bit is only set for one cycle. This change is made for program `equal_v5.c` (not shown here), and a corresponding change to the `x` variable gives a result of:

```
After compilation           : 105 LATCHES, 731 GATES, 32 INVERTERS; SIZE 1018
After netlist optimisations :  81 LATCHES, 417 GATES, 22 INVERTERS; SIZE  591
After fan-in adjustment     :  81 LATCHES, 418 GATES, 22 INVERTERS; SIZE  591
```

```
#include "harp1.h"

const dw = 8, aw = 12;
const de = (1 << dw), ae = (1 << aw);

void main(chan(in)  STDIN           : dw, chan(out) STDOUT          : dw,
          eram data[ae] = harp1lram : dw, eram hist[de] = harp1hram : aw)
{
    int addr, accu, x, temp;
    int addr_nz() = (addr != 0) : 1;

    void increment_address() { addr = addr + 1; }
    void input_to_x()        { STDIN ? x;       }
    void hist_x_to_temp()    { temp = hist[x];  }

    void input_and_histogram() {
        input_to_x();
        do par {
            { delay;            input_to_x();         }
            { data[addr] = x;   increment_address();  }
            { hist_x_to_temp(); hist[x] = temp + 1;   }
        } while (addr_nz());
    }

    void accumulate_histogram() {
        x = 0;
        do {
            hist_x_to_temp();
            hist[x], accu, x = accu, accu + temp, x + 1;
        } while (x != 0);
    }

    void equalise_and_output() {
        do par {
            STDOUT ! hist[data[addr]] \\ (aw - dw);
            increment_address();
        } while (addr_nz());
    }

    /* Main program */
    input_and_histogram();
    accumulate_histogram();
    equalise_and_output();
}
```

Figure 4.10: Program `equal_v4.c`

Though only a little smaller, this change may still be worthwhile. It is impossible to give all of the possible source level optimisations here, but, with thought and practice, writing efficient `Handel-C` programs becomes second nature. To conclude, and in case anyone is not aware of the effect, the before and after images for histogram equalisation are shown in figure **??**.

Chapter 5

`Handel-C` Interfaces

This chapter describes some of the interfacing and technology specific parts of the `Handel-C` system.

5.1 Channel Protocol Converters

In this section the specification of Channel Protocol Converters (CPCs) for `Handel-C` programs are described. A CPC is a mechanism for mediating between the internal channel communication protocols used by `Handel-C` programs and the communication protocols used by the environment of the program. They are thus used to build all the interfaces between a `Handel-C` program and its environment.

In the hardware, a `Handel-C` channel is implemented as a shared data bus which has as many wires as the width of the channel. The only communication model which is supported over such channels is that which is implemented by the input and output commands (`?` and `!`). Non-standard uses of channels may be possible, but are unsupported.

An arbitrary number of `Handel-C` processes may share access to a channel. The communication is synchronised, point-to-point, and unbuffered. The implementation only supports communications which are characterised by having at most a single reader and a single writer ready to cummicate over any one channel on any one clock cycle. Other uses may be possible but are unsupported. These conditions can be guaranteed by enforcing occam-style scope and usage rules on channels.

If a channel is used as a parameter to `main`, then the external environment may use that channel to communicate with the `Handel-C` program. In this case, and for each channel, the external environment must be modelled either as a reader or a writer, but not both. Accordingly these channels are tagged as either `input` or `output` channels (with respect to the `Handel-C` program). Any use of a channel designated as an input channel must involve only input commands, and similarly for output channels. Any other uses are unsupported.

5.1.1 Simulation CPCs

For ease of simulation during early phases of development of `Handel-C` programs, the compiler supports some simple, technology-independent CPCs. These simply connect the input/output channels of a `Handel-C` program to the built-in simulator. Many instances of these CPCs have been seen in earlier parts of this document. Input and output channels are declared using the syntax exemplified here:

```
void main(chan (in) STDIN : 8,
          chan (out) STDOUT : 32,
          chan (out) ERROR)
{ ...
}
```

The keyword `chan` introduces a channel parameter. The qualifiers `(in)` and `(out)` specify that all communications within the `Handel-C` program using that channel are generated by either input or output commands respectively. The user must also supply a local name for the channel. The width qualifier is optional, and can be omitted with the usual proviso that the width inferencer must be able to infer the width of the channel from the rest of the program. The declared names of channels and variables are used to form names of signals in the netlist associated with those structures. This can help the designer who must interact with tools which are 'downstream' of the `Handel-C` system.

5.1.2 Simple Channel CPCs

The declaration of simple channel CPCs is very similar to the use of external rams, in that it uses the 'spec' mechanism. A spec block for a simple channel or port looks something like this:

```
const spec <spec name> = {
     data   = { "P1", "P2", "P3", "P4", ...  },
     txrdy  = "Pt",
     rxrdy  = "Pr"
};

void main(port (<direction>) <channel name> = <spec name | spec>)
{ ...
}
```

where ¡direction¿ = in — out.

This named spec gives the pin names for the signals associated with the channel. Note that the pins for the data channel are given with the least significant bit first, and that the number of pins specified should be equal to the width of the channel. For an input channel the I/O blocks will be configured as input pads, and for an output channel as output pads.

The `txrdy` signal is asserted when a writer wishes to output to the channel. This will be asserted by the CPC in the case of an output channel, and by the environment for an input channel.

The `rxrdy` signal is asserted when a reader wishes to input from the channel. This will be asserted by the CPC in the case of an input channel, and by the environment for an output channel.

For any control or data signal asserted by a CPC, the assertion is coincident with the rising edge of the `Handel-C` clock. How long such signals take to propagate to the FPGA pins will depend on the complexity, placement, and routing of the internal circuitry, and the `Handel-C` compiler can thus give no guarantees on these delays.

For any control or data signal asserted by the environment, the CPC will sample that signal on the rising edge of the `Handel-C` clock. For the reasons given above, the `Handel-C` compiler has no control over delays, so it is up to the user to ensure that propagation delays, setup times for registers etc. are met.

Only channels are fully supported. However, some degenerate cases of channel CPCs may work by leaving out one or both of the `rxrdy` and `txrdy` signals from the spec. For instance, leaving out the `rxrdy` signal from an output channel results in the communication always succeeding immediately. The environment can then use the `txrdy` signal as a 'data valid' signal. Additionally the `txrdy` signal may also be suppressed, as might be required on a channel which transmits on every clock cycle.

5.2 Target Specifications

In this section the specification of a target hardware environment for a `Handel-C` program is described.

5.2.1 The 'target' data structure

The compiler requires a number of miscellaneous pieces of information in order to complete the output of a technology-specific netlist. Some of these relate the program to the target hardware platform, and others are specific to each CPC.

The items related to the program as a whole are gathered together into a data structure which is passed to the compiler via a `target=` construction in the parameter list to `main`. This is an example of such a data structure for the Harp1 card.

```
const spec harp = {
    fpga_type       = "Xilinx3000",
    fpga_chip       = "3195PQ160-3",
    clock_pad       = "P160",
    not_error_pad   = "P55",
    finish_pad      = "P44",
    clock_divider   = "1",
    carry_weight    = "50",
    critical_weight = "100"
};

void main(target=harp) { ...
}
```

The `fpga_type` field carries the definition of the target FPGA device family. For the HARP card, the family must be `Xilinx3000`. There are similar, but unsupported, options for the 2000, 4000, and 6000 families of Xilinx parts.

The `fpga_chip` field is a string which gets inserted into the xnf output file as required by the xilinx tools. The `clock_pad` field specifies which pin of the FPGA carries the clock for the `Handel-C` program. The `not_error_pad` field specifies the pin which is activated (until program restart) if a `Handel-C stop` command is executed and `finish_pad` specifies the pin which is activated (until program restart) when the `Handel-C` program terminates.

The `clock_divider` string contains a positive integer which specifies how many clock cycles of the supplied clock (on `clock_pad`) are used to generate a single clock cycle for the `Handel-C` program. All integers except "1" are unsupported.

The `carry_weight` and `critical_weight` fields specify the xnf criticality weighting of certain nets. Most ripple carry lines are tagged with the former weighting, while the latter is an unsupported feature.

5.3 Compiler Output

In this section the output formats supported by the compiler are described. Currently, only support for Xilinx XNF format output files has been implemented in `Handel-C`.

5.3.1 XNF netlist output

The default mode of the `Handel-C` compiler is to output XNF format files. This netlist format is proprietary to Xilinx, but is supported by a wide range of different tools from different vendors.

The XNF produced by the compiler can be fed directly to Xilinx tools, but encoded within the XNF is extra information that may be useful when placing and routing, or when interfacing software to the design. Lines in the XNF file which begin

```
USER,HDR,
```

contain information which documents the date and time of compilation, the version of the Handel compiler used, and the source file concerned.

Lines in the XNF file which begin

```
USER,SRC,
```

contain a pretty-printed version of the program which was compiled by the (core) Handel compiler. This may differ in some ways from the Handel-C program submitted by the user as the compiler may have performed certain source level transformations before the final step of compilation into hardware.

Lines in the XNF file which begin with either of

```
USER,NET,
USER,xmacros,
```

contain comments which relate parts of the XNF file to the fragments of the program that generated them. These comments are not intended to be exhaustive or even very useful.

Certain language constructions of a `Handel-C` program (such as channels or ports connected to the outside world) may output extra information into the XNF file that may be of use in later stages of the design process.

This extra information is encoded by lines of the XNF source that begin:

```
USER,HANDEL,.xxx,
```

where xxx is replaced by a string describing the type of information contained within those lines. For instance, external channels defined in `Handel-C` will output lines beginning with either of the following lines

```
USER,HANDEL,.mac,
USER,HANDEL,.occ,
```

These lines can be extracted from the file manually with an editor or by using a suitable perl script and used in various different ways as described below.

Embedded OCCAM

The '.occ' lines give some OCCAM source code that may be useful for interfacing between OCCAM channels used by programs running on the transuter and `Handel-C` channels in the hardware when used with the Harp board.

This OCCAM includes 2 versions of an event handler for each of the TDS and the Toolset and the required "PLACE"ments of variables in the address space. This OCCAM is tied to the particular configuration of channels used in each design and so may change between runs of the compiler.

The Event Handlers are written to be general and can often be simplified in the light of knowledge about the way the program will behave (for instance it may be possible to shut the handler down by closing just one channel as opposed to closing them all individually). Such customisation is essential to ensure efficiency.

Embedded Placement and Routing

The '.mac' lines give some commands that can be fed to the Xilinx xact tools to generate preplacement and routing information for the external components of a circuit.

Typically sections of the circuit that communicate with the outside world are very timing dependent; in particular, parts of any circuit that communicate between separately clocked systems (e.g. between the FPGA and transputer on the HARP board) must be very carefully designed to avoid problems of metastability and data slew. It is an unfortunate feature of the universe we live in that there can be no absolute guarantee of error-free communication between separately clocked digital systems.

To make them adequately reliable, the design of such interfaces necessarily encompasses much more than just the gate-level design of the circuit itself; for example, gate, wire and pin delays must all be taken into account in order to make the interfaces work successfully. Thus it is not reasonable to expect that such carefully designed circuits can be exposed to a general-purpose place and route system such as the Xilinx `ppr` tools.

The `Handel-C` system interacts with the xilinx software via xnf-format files. Unfortunately the xnf language does not support the transfer of pre-routed designs, although pre-placed designs can be specified. Since some of the CPCs necessarily have to be pre-routed, we have built a system which outputs a set of text lines which are actually Xilinx LCA editor commands that wire up these critical CPC circuits. To achieve the level of reliability of communication that the HARP and `Handel-C` systems were designed for, it is necessary to use this information to force the Xilinx software to implement the CPCs in precisely the intended way, placement and routing included.

To use placement and routing for a design called mydesign.xnf the following steps should be followed:

- Extract the '.mac' lines from mydesign.xnf creating mydesign.mac.
- Start the Xilinx xact editor software.
- From the Profile menu select Mode and choose "Expert".
- From the Designs menu select Part and choose the appropriate part number (typically 3195PQ160 for Harp-1).
- From the Designs menu select Speed and choose the appropriate speed grade for the part (typically 3 for Harp-1).

- Ensure there is no previous “guide.lca” in the current directory.
- From the Designs menu select Design and choose new. Type “guide.lca” in the top window.
- From the Programs menu select EditLCA; EditLCA will now load and the menu change.
- In the Profile menu ensure that “AutoDRC”, “AutoTime” and “AutoRoute” are all turned off.
- Enter “execute”, then when prompted for a filename “mydesign.mac”. The macros are then expanded into the design, and this may take a few minutes.
- Next, if you wish to take advantage of the pre-routing, type “route *”. Some errors may be thrown up at this stage - ignore them.
- Enter “save” then hit return to save the guide file.
- Entering exit twice will leave XACT.

Once you have generated a guide file, how you use it depends on the synthesis tool you then use. Consult the appropriate user manual for details. For the Xilinx apr tool add “-g guide.lca” to the command line options.

Appendix A

Quick Reference

A.1 Compiler Options

A.2 Expression Syntax

A.3 Statement Syntax

A.4 Differences from `C`

Appendix B

Error Messages

The following error messages can be caused at different stages in compilation.

B.1 Parse Errors

Illegal character
A character or character sequence in the input sequence stream was not recognised. Check the source file.

Unmatched closing comment
A closing comment was found when the compiler was not parsing a comment. Remove the closing comment.

Unterminated comment
The compiler read to the end of a file whilst still parsing a comment. Insert another closing comment.

Nested comment
An opening comment was found whilst already parsing a comment. Remove the nested comment.

Syntax error
The compiler did not recognised the sequence of tokens given. Check the form of the statement shown, and that the previous statement was terminated by a semicolon.

B.2 Semantic Errors

Procedure . . . not declared
A procedure call was made to an undeclared procedure. Check that the declaration name exactly matches the calling name.

Redefinition of parameter . . .
A parameter given to a `spec` format argument was declared more than once. Remove the repeated declaration.

Parameter . . . not declared

A parameter required in a `spec` definition was not given (this is also caused when no `spec` definition is given to an external RAM). Insert the required definition.

eram: insufficient address pins

The number of elements required in the interface declaration of an external RAM exceeds the maximum address bus width of the RAM. Use a larger RAM or a smaller number of elements.

eram: insufficient data pins

The width of data bus required by the interface declaration cannot be satisfied by the number of data pins given in the `spec` declaration. Use a narrower bus or a wider RAM.

eram: no clock enable pin

No CE pin was provided to the external RAM. Add one.

eram: unknown parameters in definition

There were additional unexpected parameters in the `spec` definition for an external RAM. Remove the offending parameters.

Multiple definitions of ...

A certain identifier was declared twice. Change the name of one of the declarations.

... not declared

The given identifier was used but not declared. Declare it.

Expected ... to be an ...

A name was found matching an identifer, but the object was not of the expected type. Check the allowed usage of the object.

... cannot be used without a subscript

A name of a RAM or ROM was given but without a subscript (pointers are not supported). Subscript the identifier.

cannot subscript ... with an expression

An object (perhaps an array) is being subscripted by a non-constant, which is not allowed. Unsubscript the identifier.

B.3 Health Check Errors

Not declared (warning)

An identifier was used but not declared. This error should not occur with `Handel-C`.

Never read from ... (warning)

A channel or variable is never read from. This variable or channel is likely to be removed during optimisation.

Never written to ... (warning)

A channel or variable is never written to. This variable or channel is likely to be replaced by a constant zero during optimisation.

Duplicated declaration

An declaration has occured twice. Remove one of the declarations.

Duplicated name

A name has been used twice. Remove one of the names.

Bad param
A value that is not a channel is written to by a channel communication.

Illegal channel input statement involving . . .
A value that is not a channel is read from by a channel communication.

Loop may have zero-time body (warning)
The given loop may take zero cycles to execute. If this is the case, a combinatorial cycle will be generated. This will be broken automatically by the compiler.

Duplicated LHS in Assignment
In a parallel assignment the same value is written to twice.

Check variable/channel usage in par (warning)
A variable or channel is written to in more than one branch of a `par`. This may not be a problem if the program is correctly structured.

Check procedure usage in par (warning)
A procedure is used in more than one branch of a `par`. This may not be a problem if the program is correctly structured.

B.4 Width Inferencer Errors

Invalid Inference
Widths of objects have been specified in a contradictory way. The expression in which the contradiction was found is printed.

Incomplete Inference
There were insufficient widths specified in the source program for the compiler to infer as much information as required. Specify more information.

Inference Internal Error
The width inferencer has detected an invalid state. This error should never occur, please report.

Unexpected EXPR in Declare
An expression was found in a declaration. This error should never occur, please report.

Unexpected EXPR in IO list
An expression was found in an interface specification. This error should never occur, please report.

Unexpected EXPR in MonList
An expression was found in a simulator monitor list. This error should never occur, please report.

B.5 Compiler Errors

EXCEPTION: BAD_ASSIGNMENT
An attempt was made to assign to something other than a register, RAM, channel or bus.

EXCEPTION: BAD_BUS_MATCH

An internal error occured in the compiler. This error should never occur, please report.

EXCEPTION: BAD_CONST
A constant of undefined width or of width zero has been found.

EXCEPTION: BAD_DECL
An invalid type of declaration has been made. This error should never occur, please report.

EXCEPTION: BAD_EXPR
An attempt was made to compile an invalid expression. This error should never occur, please report.

EXCEPTION: BAD_STAT
A bad statement was found, probably reading from a non channel. This error should never occur, please report.

EXCEPTION: INCONSISTENT_ASSERTION
The compiler could not meet timing constraints given in the source program.

EXCEPTION: OP_BUS_OF
An attempt was made to make a bus from an invalid object. This error should never occur, please report.

EXCEPTION: SYMBOL_NAME_CLASH
An identifier name has been used twice, and has lead to a conflict in the netlist. Change one of the names.

EXCEPTION: Uninferable_Width
A bit width could not be inferred. This may be due to an underspecified left or right shift in `Handel-C`.

EXCEPTION: WIDTHS_DONT_MATCH
An attempt was made to combine two expressions of differing widths. This error should never occur, please report.

B.6 Block Checking Errors

Fatal Error: AND gate with empty input list

Fatal Error: OR gate with empty input list

Fatal Error: XOR gate with empty input list

Fatal Error: TSC gate with empty input list

Fatal Error: Inverter with empty input list

Fatal Error: Inverter with multiple inputs

Fatal Error: Distributed gates still present in block list

Fatal Error: Wires still present in block list

Fatal Error: removing distributed gates

Fatal Error: Different types of distributed gate with same output
An internal compiler error has occured. This error should never occur, please report.

B.7 Optimiser Errors

Fatal Error: le_gate: Floating input found

Fatal Error: This shouldn't have happened! (1)

Fatal Error: This shouldn't have happened! (2)

Fatal Error: gengraph: Distributed gate found in blocklist!

Fatal Error: set_essential: Floating input found!

Fatal Error: gengraph2: Inverter found with ¿1 input!

Fatal Error: set_behaviour: Dummy Node!

Fatal Error: fix_drains: Floating input found

Fatal Error: set_all_needs_opt: Floating input found!

Fatal Error: dispnode: Floaty input!

Fatal Error: rewrite: Floaty input!

Fatal Error: Floaty input! (doscan)

Fatal Error: Floaty input! (simplify)

Fatal Error: Floaty input! (dorewrite)

Fatal Error: Floaty input! (dopass)
An internal error has occurred. This error should never occur, please report.

B.8 Netlist Output Errors

EXCEPTION: BAD_BLOCK
An attempt was made to output a block in a format in which it is not valid.

Fatal Error: extra_output_function has not been implemented for BLIF format
BLIF output format cannot support the combined netlist format used with XNF.

Fatal Error: BLIF format does not support bidirectional pads
BLIF output format does not support features compiled into the netlist.

Fatal Error: ClockConnections: Clock Dividers are Xnf specific
The netlist generated contained a clock divider, but the desired output format was not XNF.

Fatal Error: ClockGenerator: Internal crystal oscillators are Xnf Specific
The netlist generated contained an internal crystal oscillator, but the desired output format was not XNF.

Fatal Error: CPC_TPChanOut can only output Xilinx ...

Fatal Error: CPC_TPChanOut can only output Xilinx ...

Fatal Error: CPC_TPChanInOut can only output Xilinx ...

Fatal Error: CPC_TPPortInOut can only output Xilinx ...
An attempt was made to use a CPC with non-Xilinx output.

B.9 Simulator Errors

EXCEPTION: SIG_VAL

An unusual signal value was discovered during simulation. This error should never occur, please report.

Fatal Error: Cannot simulate BlackBox called ...

An unusual "blackbox" was found in the netlist, rendering it unsimulatable. This error should never occur, please report.

Fatal Error: Simulator can't proceed

An error has occurred rendering the netlist unsimulatable.

Fatal Error: Program has errors which make the output unsimulatable

An error has occurred rendering the netlist unsimulatable.

B.10 General Errors

Out of memory

The compiler ran out of memory. This is normally indicative of an infinite loop. This error should never occur, please report.

Evaluation failed ...

An internal error occurred. This error should never occur, please report.

Invalid argument ...

An internal error occurred. This error should never occur, please report.

I/O failure ...

Some I/O failure occurred, such as not being able to read the input file, or write the output netlist.

Uncaught exception

Some unspecified error occurred. This error should never occur, please report.

Bibliography

[1] Michael Spivey and Ian Page. “How to design hardware with Handel”, Technical Report, Oxford University Computing Lab, 1993.

[2] Ian Page and Wayne Luk, “Compiling OCCAM into FPGAs” in FPGAs, Eds Will Moore and Wayne Luk, 271-283, Abingdon EE & CS books, 1991.

[3] Geraint Jones, “Programming in OCCAM”, Prentice-Hall International, 1987.

[4] INMOS Ltd, “The OCCAM2 Programming Manual”, Prentice-Hall International, 1988.

[5] A E Lawrence, “HARP (TRAMple) Manual, Volume 1, User Manual for HARP 1 and HARP 2”.

[6] A E Lawrence, “Macro support for the Xilinx Architecture”, 1995.