hcc: A `Handel-C` Compiler

Matthew Aubury
Ian Page
Geoff Randall
Jonathan Saul
Robin Watts

Oxford University Computing Laboratory

August 28, 1996

Contents

Foreword

`Handel-C` is a simple programming language designed to enable the compilation of programs into synchronous, usually FPGA based, hardware implementations. Handel is not a hardware description language though; rather it is a product of a long term research programme at Oxford investigating 'system codesign' – the creation of systems comprised of both hardware and software components – from a single program.

This research is underpinned by the belief that the engineering of any system should be based on sound mathematical principles, and that this is especially true of tomorrow's large-scale and highly parallel systems. So while the syntax of `Handel-C` reminiscent of that of Kernigan and Ritchie's C programming language, Handel is founded on the semantics of Hoare's CSP algebra.

It should be emphasised that `Handel-C` and the associated software are very much part of developing research projects, and are subject to considerable changes.

This document describes the operation of the hcc compiler. It does not concern the `Handel-C` language or the Harp reconfigurable computing platforms which are often used with it. These are covered in [8] and [5, 6, 7] respectively. Some examples of `Handel-C` programs are contained in [9].

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.

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.

For each clock cycle the simulator prints the number of clock cycles since the program was started, and the state of all variables at the start of the cycle. Variables are shown in the order that they were declared, and their values are shown as unsigned base 10 integers. If variables are required from an external channel then the user is prompted for a value. Pressing *return* represents a channel that is not ready. If variables are output to an external channel then the user is asked whether the channel is ready for output on each cycle until the answer *y* is received, when the output value is printed. Output values are shown as both signed and unsigned base 10 integers.

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 described in more detail in the following sections.

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.

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.

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

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.

6 Notes on the Compiler

6.1 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.

6.2 Bit Operators

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.

6.3 The STOP Statement

In simulation execution of the entire program will stop as soon as a `stop` statement is executed, whereas in the hardware any process running in parallel with `stop` will continue to run.

6.4 Renaming

The compiler 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 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.

6.5 Preprocessing

Preprocessor directives such as `#include` and `#define` can be used, but since the compiler 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`. The -cpp option calls `cpp` to do this. The compiler 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.

6.6 Simulator Behaviour

The `Handel-C` compiler contains an inbuilt gate level simulator to allow testing of circuits after compilation and optimisation.

Due to the way in which the simulator works, it is limited to simulating circuits which do not contain any combinational loops. Most `Handel-C` programs will conform to this without any problems, but it is possible to generate circuits that have combinational loops within them.

Generally these loops can arise in one of 3 ways:

- 1. PriAlts with external channels for guards can result in problems. Essentailly the simulator considers the "user" that drives the handshake lines for the channel communications by replying to its prompts as a component of the circuit. It therefore flags prialts on external channels as combinational cycles that involve the user.
- 2. Reading the transmission states of external channels (via rxrdy or txrdy), and then using these states to decide whether to do an input/output on the same channel, again produces a combinational cycle involving the user.
- 3. Accessing a location in an external RAM determined by the contents of another external RAM, and doing the reverse elsewhere in the program produces combinational cycles.

Whether or not combinational cycles involving the user are actually a problem comes down to precisely what the user is doing. To convince the simulator that there is no problem, an extra buffer process can be employed so as to avoid prialting on external channels.

To avoid the problem with mutually indexing RAMs, it suffices to load the value from one of the RAMs into a register, and then to operate from this register in the next cycle.

7 Interfaces

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

7.1 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.

7.1.1 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. Input and output channels are declared using the following syntax:

```
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.

7.1.2 Simple Port CPCs

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

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

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

where `<direction> = in | out`.

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

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

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

When both handshake lines are high on the rising edge of the clock, synchronisation takes place and the communication is scheduled. The external handshake signal must be removed before the rising edge of the clock pulse after the communication is scheduled or it will be taken as a request for a further communication. Behaviour is not defined for requests for communication which are offered and withdrawn before synchronisation takes place.

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 full ports are completely supported. However, some degenerate cases of port 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 port 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 port which transmits on every clock cycle so that no 'data valid' signal is needed.

7.2 Target Specifications

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

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.

8 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`.

8.1 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.

8.1.1 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.

8.1.2 Embedded Placement and Routing

The '.mac' lines give some commands that can be fed to the Xilinx xact tools to generate pre-placement 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 meta-stability and data slew.

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, the system writes 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.

Detailed instructions for interacting with the Xilinx software are contained in [5]

9 Error Messages

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

9.1 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.

9.2 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.

9.3 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.

9.4 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.

9.5 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.

9.6 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.

9.7 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.

9.8 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.

9.9 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.

9.10 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.

References

[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.

[7] A E Lawrence, "The HARP software library and utility package", 1996.

[8] M Aubury, I Page, G Randall, J Saul, R Watts, "Handel-C Language Reference Guide", 1996.

[9] M Aubury, I Page, G Randall, J Saul, R Watts, "Handel-C Program Examples", 1996.