`Handel-C` Language Reference Guide

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 fundamentals of the `Handel-C` language. It does not concern the hcc compiler 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 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* (see Section 2.3) deals with setting up global constants (such as the widths of data buses) and of defining specification fields to describe external interfaces.

The section *target declaration* (see Section 2.4) is optional and allows the user to define the target technology for the circuit.

The section *interface declarations* (see Section 2.5 declares the communication channels between the Handel-C program and the outside world.

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 *statements* (see Section 3). This is much like conventional C, except for 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 }.

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.

2.1 Values

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

2.1.1 Integer numbers

Integer constants in Handel-C may range over any values. 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 4.5).

2.1.2 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.

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.

2.2.1 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.

2.2.2 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 4.5). 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.

2.3 External Declarations

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

2.3.1 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. For example 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 whose width is specified.

2.3.2 Specifications

Specifications are free-form constants used to describe interfaces and the target architecture. 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"}
};
```

2.4 Target Declaration

The target declaration is placed in the argument to `main(...)`. It allows the specification of technology specific information. Target declarations follow the form:

```
target = spec
```

where the *spec* field is either a a braced specification or a `const spec` declared externally.

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) { ...
}
```

2.5 Interface Declarations

Interface declarations are placed in the argument to `main(...)`. Allowable declarations are channels, ports and external RAMs. Channel declarations are written as `chan(in)` *name* `:` *width* and `chan(out)` *name* `:` *width* for transputer 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 externally, or may be a full *spec* field, enclosed in braces. An example is as follows:

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

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

2.6 Internal Declarations

Internal declarations can be made immediately within the `main` block or within any braced block within the body of `main` (including procedures). Normal scoping rules are used. 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.

2.6.1 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.

2.6.2 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 process will use the same channel for input.

2.6.3 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 dependent 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.

2.6.4 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 to the language in future.

2.6.5 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 the 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 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: parallel assignment and parallel composition. Parallel assignment allows values to be assigned to a number of variables in one clock cycle. Parallel composition allows statements in a block to be executed in parallel. The semicolon, sometimes regarded as a sequential composition operator, instead has the meaning of statement terminator in `Handel-C`.

3.1 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 helpful:

```
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 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.

3.2 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.

3.3 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 may be possible but are not supported; they should only be used where the timing of the communication is precisely known so that both readers can use exactly the same clock cycle to do their simultaneous read, otherwise it is possible that two reads may be scheduled when only one was intended.

There are variants of the input and output commands with which the programmer can specify that a partner to a communication definitely will be ready to communicate whenever the command is scheduled. The compiler can use this guarantee to save some synchronisation hardware. This may be useful when communicating with external channels where the environment is always receptive. Examples might be reading or writing an external device register, or sampling a data line. The single-tick input and output commands are:

```
channel ?´ variable  channel !´ expression
```

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 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).

3.5 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.

3.6 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).

3.7 For loops

The `for` statement 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.

3.8 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.

3.9 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. Only 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.

3.10 Stop

The program statement `stop` in `Handel-C` terminates execution of the process immediately. Any process running in parallel with `stop` will continue to run.

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

3.11 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.

3.12 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 `prialt`s, 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, automatically 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.

3.13 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 states of any local variables in a procedure are 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.

4 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. Procedure calls and assignments are prohibited within expressions. Consequently the familiar C constructs of `++` and `--` are not supported by `Handel-C`.

4.1 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 most 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.

4.2 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)`	$\sqcup(\log_2 a)$

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);
```

4.3 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.

4.4 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.*

4.5 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). 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].

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.

4.6 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);
```

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

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 default expression `d`. If the `ci` exhaustively cover all of the possible values of `e` then the default clause must be omitted, otherwise it is mandatory.

4.7 Operator Precedence

The precedence and associativity of operators in `Handel-C` is shown in the following list. Operators higher in the list are given higher precedence than operators lower in the list. Operators on the same line have the same precedence.

right	`=`
right	`:`
left	`|`
left	`&`
left	`^`
left	`== !=`
left	`< > <= >= .<. .>. .<=. .>=.`
left	`@`
left	`<- \\`
left	`<< >>`
left	`+ -`
left	`* .*.` mod div
right	`~`
left	`. ..`

4.8 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	$2^{\#e}$

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 $[\ldots]$ denote optional components. Braces $\{\ldots\}$ denotes zero, one or several repetitions of the enclosed components. Braces with a trailing plus sign $\{\ldots\}^+$ denote one or several repetitions of the enclosed components. Parentheses $(\ldots)$ denote grouping.

5.1 Lexical Conventions

5.1.1 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.

5.1.2 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 */
```

5.1.3 Identifiers

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

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.

5.1.4 Integer literals

$$\begin{aligned} \textit{integer-literal} &::= [-]\{\texttt{0}\ldots\texttt{9}\}^+ \\ &\mid\ [-](\texttt{0x} \mid \texttt{0X})\{\texttt{0}\ldots\texttt{9} \mid \texttt{A}\ldots\texttt{F} \mid \texttt{a}\ldots\texttt{f}\}^+ \end{aligned}$$

| [−](0o | 0O){0...7}$^+$
| [−](0b | 0B){0...1}$^+$

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.)

5.1.5 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:

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

5.1.6 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.

6 Handel-C Vs C

While Handel-C was designed to look and feel as much as possible like a variant of C, the nature of the target architecture is such that certain C language features are inappropriate. These have therefore been removed in Handel-C.

Similarly, `C` was found not to contain some constructors vital to the expressiblilty of concepts like parallelism. Accordingly several new operators have been added. The tables below summarise these operators available.

It is worth noting that assignment is not part of the of the expression language in `Handel-C`, so the following code, which is perfectly legal (though arguably bad style) in `C` is not permitted.

```
void main(void)
{
    int a, b;
    if (a = 1) b = 2
}
```

As a side effect of this the commonly used `++` and `--` operators are not available.

Expressions					
In Both		In C Only		In Handel-C only	
	(negation)	->	(no structs)	?	(Channel input)
+		.		!	(Channel output)
-		!	(logical negation)	<-	(take)
*	(multiply)	++		\\	(drop)
<<		--		.<.	(unsigned ops)
>>		*	(indirect)	.>.	
<		&		.<=.	
>		sizeof		.>=.	
<=		/	(divide)	@	(concat)
>=		%	(modulo)		
==		&&	(logical and)		
!=		||	(logical or)		
&	(boolean and)				
^	(boolean exor)				
|	(boolean or)				
? :	(conditional)				
[]	(arrays)				

Statements		
In Both	In C Only	In Handel-C only
{;}	break	par {}
case/switch	continue	prialt {}
do while	return	stop
while	goto	skip
if else	typedef	
for (;;)		

Types and type operators		
In Both	In C Only	In Handel-C only
int	char	chan
const	double	bool
void	enum	spec
	float	eram
	long	ram
	short	rom
	register	
	static	
	extern	
	struct	
	volatile	
	unsigned	

References

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", 1995.

[6] A E Lawrence, "Macro support for the Xilinx Architecture", 1996.

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

[8] M Aubury, I Page, G Randall, J Saul, R Watts, "hcc: A Handel-C Compiler", 1996.

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