`Handel-C` Language Examples

Matthew Aubury
Ian Page
Geoff Randall
Jonathan Saul
Robin Watts

Oxford University Computing Laboratory

August 28, 1996

Contents

1 Short Examples **2**
1.1 Accumulator (`accu.c`) 2
1.2 Integer Division (`divide.c`) 4
1.3 Square Root (`root.c`) 5
1.4 Queue (`queue.c`) 5
1.5 Microprocessor (`proc.c`) 6
1.6 Edge Detector (`edge.c`) 7
1.7 Merge Sort (`merge.c`) 9
1.8 Heap Sort (`heapsort.c`) 10

2 Developmental Example – Histogram Equalisation **11**
2.1 First Attempt 11
2.2 Making it Parallel 14
2.3 Making it smaller 14

```
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 1: Program `accu.c`

In this document 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.

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.

1.1 Accumulator (`accu.c`)

The `accu.c` program, shown in figure 1 is extremely simple, but demonstrates several important points. The compiler is called 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. This is the default behaviour of the `Handel-C` compiler: compile the program, perform some simple optimizations, and then enter the simulator.

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.

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 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

```
/* 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 2: Program `divide.c`

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.

1.2 Integer Division (`divide.c`)

This program (shown in figure 2) 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

```
/* 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 3: Program `root.c`

task, giving `Handel-C` programs access to division. In this case the channel communications would be internal to the body of `main`.

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!

1.4 Queue (`queue.c`)

The program `queue.c`, shown in figure 4, 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)

```
/* 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: Program `queue.c`

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.

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.

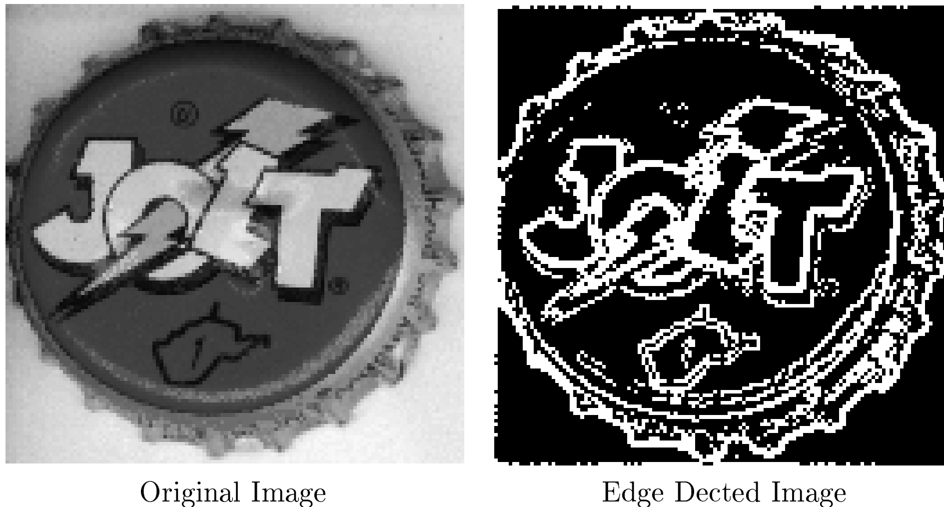

Figure 5: Result of program `edge.c`

1.6 Edge Detector (`edge.c`)

The program shown in figure 6 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 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 5.

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,

```
#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 6: Program `edge.c`

```
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 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.

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.

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 $\mathrm{O}(n)$ walks over the heap, the algorithm has the same asymptotic complexity, $\mathrm{O}(n \log n)$, 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).

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.

2.1 First Attempt

The first attempt is program `equal_v1.c`, shown in figure 7. 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, `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).

```
/* 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 7: Program `equal_v1.c`

```
/* 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 8: Program equal_v2.c

2.2 Making it Parallel

The next attempt is program `equal_v2.c`, which is shown in figure 8. 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.

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 10. The first is to denote common subexpressions (in particular `addr != 0`), and common substatements (such as `addr = addr + 1`) into `ints` 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:

```
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

Original Image Histogram Equalised Image

Figure 9: Result of Histogram Equalisation

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
```

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 9.

```
#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 10: Program `equal_v4.c`

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, "hcc: A Handel-C Compiler", 1996.