ClearSpeed™

Introductory Programming Manual

The ClearSpeed Software Development Kit

Document No. 06-UG-1117 **Revision:** 2.K

September 2010

ClearSpeed™

The ClearSpeed Software Development Kit Introductory Programming Manual

Overview

This document is intended to help new users get started with the ClearSpeed Software Development Kit (SDK). Some understanding of parallel processing in general, and the CSX processor architecture in particular, will be useful although the main concepts are introduced in this document.

This manual starts with an overview of the architecture. It then works through a series of examples which demonstrate various features of the C language extensions used to program the CSX processor.

The following chapters provide more detail of the tool chain and the **C*n*** programming language. This is primarily targeted at programmers with some previous C or C++ programming experience.

Note: *Please refer to the ClearSpeed Software Developer Kit Installation Guide for install instructions.*

Structure of the document

This chapter explains the background to the **C*n*** language and the CSX architecture.

Chapter 2: Simple Cn programs, presents a number of example programs to demonstrate the use of **C*n*** for real problems. There are a number of exercises that the reader can use to test and extend their understanding.

Chapter 3: Building and running Cn programs, gives a brief overview of the commands to run the main tools in the Software Development Kit (SDK). This should be enough to allow the reader to run the examples provided.

Chapter 4: Parallel programming in Cn, introduces the basics of data-parallel programming and the use of the debugger.

Chapter 5: Cn for the working C programmer, provides an informal discussion of the features of the **C*n*** programming language, focusing on the features specific to programming the CSX processor.

Chapter 6: More Cn programs, uses some more complex programs and a series of exercises to teach more about the **C*n*** language and the CSX architecture.

Chapter 7: Debugging Cn, explains how to use the debugger, a port of GDB, to debug **C*n*** programs.

Chapter 8: Programming host applications, describes how to build applications which run on both the host processor and a CSX coprocessor.

Chapter 9: Programming hints, provides some reminders of common mistakes made when learning **C*n***.

Bibliography, is a list of references and suggested further reading.

Table of contents

1 Overview of the architecture 7

1.1 Terminology 9

1.2 Programming model 9

2 Simple Cn programs 10

2.1 Aim of this document 10

2.2 Example 1: Hello world 11

2.3 Example 2: A first poly program 11

3 Building and running Cn programs 13

3.1 A brief description of the tool chain 13

3.2 Compiling program 13

3.2.1 File naming conventions 14

3.2.2 Compiling as C 14

3.3 Running your code 15

3.3.1 Running on a simulator 15

4 Parallel programming in Cn 17

4.1 Sequential code and parallel code 17

4.2 Using the debugger 18

4.3 Start the program 19

4.4 Exit the debugger 19

5 Cn for the working C programmer 20

5.1 Comments 20

5.2 Data types 20

5.3 Mono and poly specifiers 20

5.3.1 Basic types 21

5.3.2 Pointer types 21

5.3.3 Pointers to mono data 21

5.3.4 Pointers to poly data 23

5.3.5 Illegal casts 25

5.3.6 Array types 26

5.3.7 Struct and union types . . . 26
5.3.8 Typedefs . . . 27
5.4 Mixing mono and poly variables . . . 28
5.5 Flow control . . . 28
5.5.1 If statements . . . 28
5.5.2 For, while and do..while loops . . . 30
5.5.3 Goto statement . . . 30
5.5.4 Labeled breaks . . . 31
5.5.5 Switch statements . . . 31
5.6 Functions . . . 32
5.7 Other architectural features . . . 32
5.7.1 Data transfers between mono and poly . . . 33

6 More Cn programs . . . 35
6.1 Example 1: Mandelbrot set . . . 35
6.1.1 Standard (mono) implementation . . . 36
6.1.2 Parallel (poly) implementation . . . 38
6.2 Exercise 2: Input and output . . . 41
6.3 Example 3: Asynchronous I/O . . . 43

7 Debugging Cn . . . 47
7.1 Compiling for debug . . . 47
7.2 Starting csgdb . . . 47
7.3 Using csgdb to investigate mandelbrot_poly.cn . . . 48
7.3.1 Listing source code . . . 48
7.3.2 Connecting to the device . . . 49
7.3.3 Setting breakpoints . . . 49
7.3.4 Starting execution . . . 49
7.3.5 Examining variables . . . 50
7.3.6 Reading registers . . . 52
7.3.7 Stepping to a function call . . . 53
7.3.8 Viewing the poly enable state . . . 55
7.3.9 Attaching commands to breakpoints . . . 59
7.3.10 Returning from a function . . . 62
7.3.11 Viewing memory . . . 69
7.3.12 Terminating the program . . . 73

8 Programming host applications . . . 74

8.1 Initialization . . . 75

8.2 Synchronization and communication . . . 75

8.3 Sample code . . . 76

9 Programming hints . . . 77

9.1 Effects of function return type . . . 77

9.2 Use of mono variables inside poly conditionals . . . 77

9.3 Debugging tips . . . 78

9.3.1 Random errors occuring . . . 78

9.4 Mixing mono and poly conditions . . . 78

9.5 Dereferencing mono*poly pointers . . . 79

Bibliography . . . 80

Revision history . . . 81

1 Overview of the architecture

Figure 1 shows a high-level view of the CSX600 processor architecture. The CSX600 processor comprises a multi-threaded single-instruction multiple-data array processor core, external DRAM interface, high-speed interfaces and embedded SRAM integrated onto a single chip. All subsystems on the chip are interconnected using the ClearConnect bus on-chip network. The ClearConnect bus can be extended through the ClearConnect bridge ports to provide communication between two or more CSX600 processors or other devices such as FPGAs and through the host debug port to the host.

The array processor core contains 96 processing elements (PE). Each PE can execute simultaneous add and multiply operations. The CSX600 supports fully pipelined operation executing one instruction per cycle.

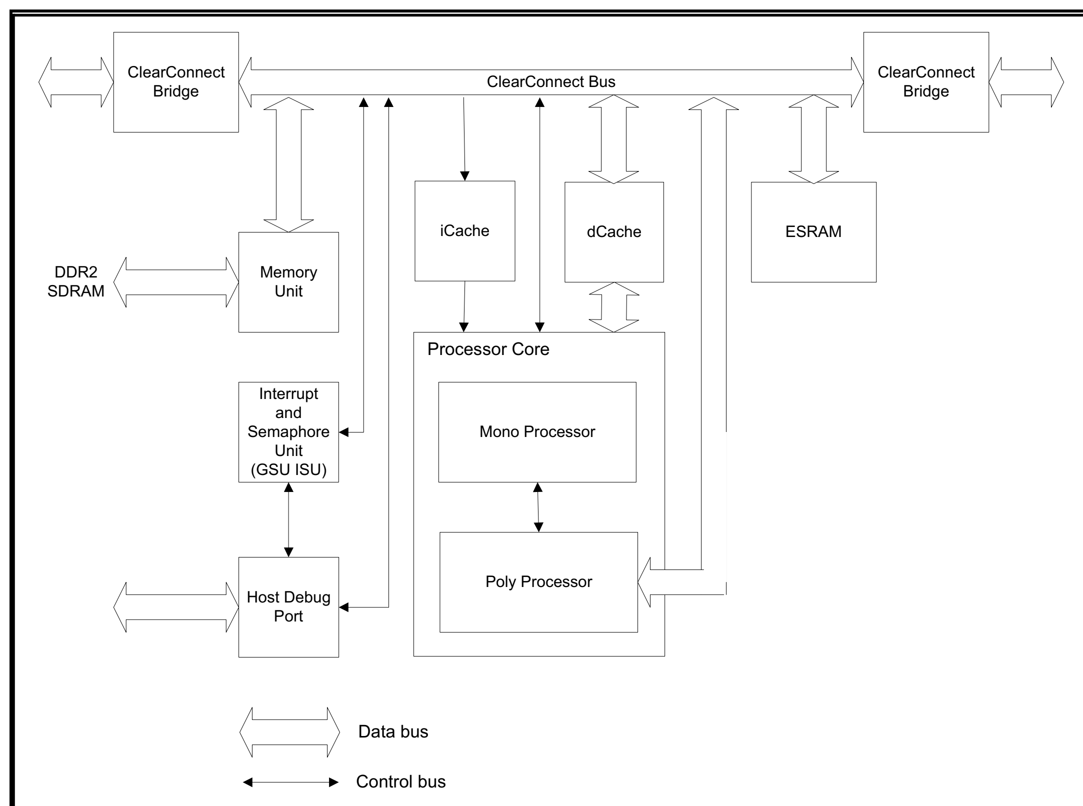

Figure 1. CSX600 processor

The CSX600 has 256 Kbytes of on chip SRAM (mono memory) and a DDR2 interface to SDRAM. The part that is most relevant to our discussion are the execution units. This consists of two main parts: the *mono* execution unit and an array of processing elements (PEs) which form the *poly* execution unit. Each instruction in the single instruction stream is executed by the mono or poly execution unit, as appropriate.

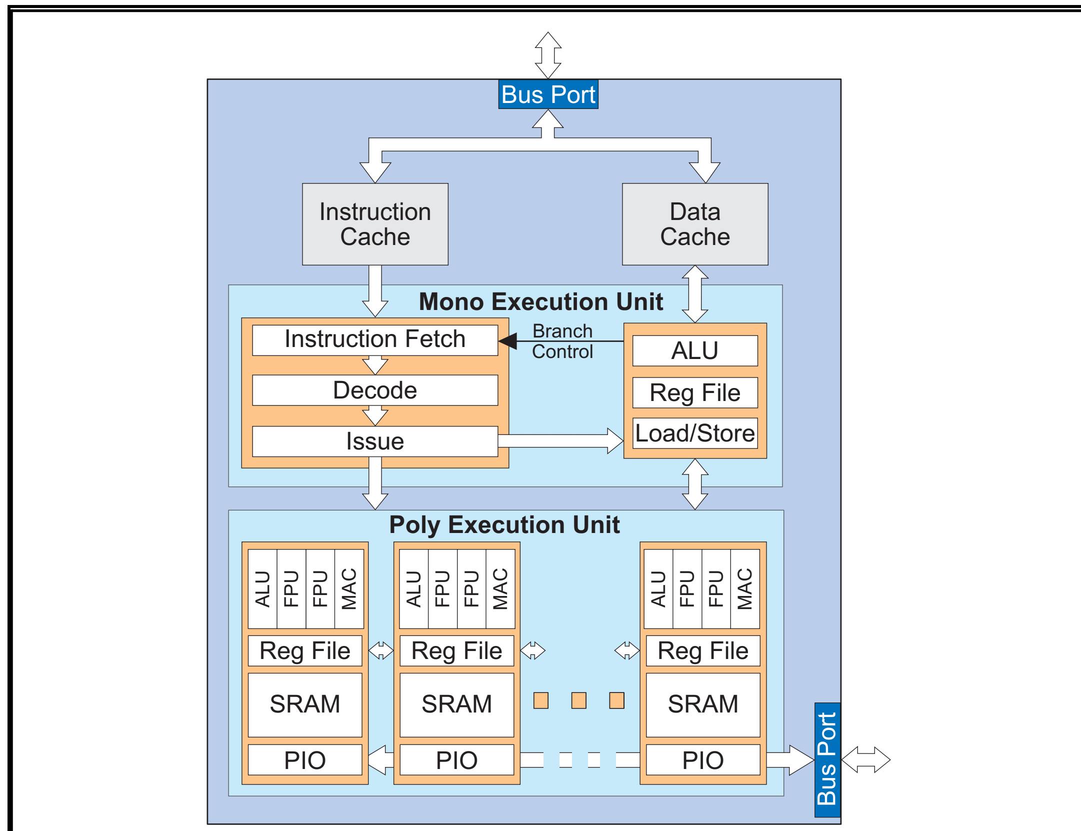

Figure 2. Execution units

The performance of the processor comes from the fact that data is processed in parallel by the poly execution unit.

The main features of the poly execution unit are:

- An SIMD (single instruction multiple data) array of 96 PEs. This means that each PE executes the same instruction but on different data.
- Each PE has an ALU (arithmetic logic unit), 32 + 64 bit FPU (floating point unit), a register file and 6 Kbytes of SRAM.
- The PEs are connected as a linear array via a connection path.

1.1 Terminology

To avoid any confusion a number of terms are defined here:

- **Basic type**: A variable which stores a single data object (for example, a char, an int, a float, and so on).
- **Aggregate type**: Types, such as arrays, structures, unions or pointers, derived from other types. These types may hold several data objects.
- **Mono variable**: A variable that has one instance. This can be of basic or aggregate type.
- **Poly variable**: A variable that has many instances with, typically, different data values on each poly Processing Element (PE). This can be of either a basic or aggregate type.
- **Mono memory**: Memory associated with mono data. There is one instance of this accessible by all PEs. Also referred to as *local memory*. This memory may be on chip and/or on the same PCB card as the CSX processor.
- **Poly memory** (also known as *PE memory*): Memory associated with poly data. Each PE has its own local block of poly memory; each instance of poly memory is only visible to the corresponding PE.

Note: Mono and poly memory are two physically distinct memory spaces, with their own memory maps.

1.2 Programming model

The main thing that is required in a programming language for the CSX architecture is a means of representing poly data. This is done by introducing the keyword `poly` and a corresponding `mono` keyword. These new keywords are called *multiplicity specifiers*. They allow the programmer to specify the domain in which the declaration will exist.

- `mono` – the object exists in the mono domain (that is, a single instance).
  For example, `mono int a;` is equivalent to `int a;`
- `poly` – the object exists in the poly domain (that is, many instances).
  For example, `poly int b;` the variable `b` has a separate value on each PE.

The default multiplicity is mono; if no multiplicity specifier is used, the variable will be mono.

All operations that can be performed on ‘normal’ (mono) variables can also be done with poly variables. In the case of poly variables, multiple values will be operated on simultaneously (one per PE).

Poly variables can also be used in conditional statements, so that code can be executed on some PEs and not others.

Because of the extra complexity of having both mono and poly conditional expressions, it is impractical to support the `goto` statement. As a common use of this is to jump out of deeply nested conditions or loops, another extension to the **C*n*** language is the use of labels with break or continue statements

Refer to the following **C*n*** chapters in this manual. Also see the appropriate chapter of the *SDK Reference Manual* for full details of the **C*n*** language.

2 Simple C^n programs

The purpose of this document is to teach the basics of **C^n** and to inspire confidence in using it. Programming exercises are provided to teach these basics, since it is believed that hands-on experience is the best teacher.

2.1 Aim of this document

Having completed the five programming exercises in *Chapter 2* and *Chapter 6* you should have code running which uses all of the following elements of **C^n**:

- **C^n** types
  - monos
  - polys
- Conditionals
  - conditionals on monos
  - conditionals on polys
- Pointers
  - mono pointers to monos
  - mono pointers to polys
  - poly pointers to monos
  - poly pointers to polys
- External memory access
  - semaphores
  - programmed I/O memory transfers

While it is possible to step through these elements sequentially in a programming example, it is always more interesting to learn something in a practical context.

Note: *These programs are tutorials intended to show how the features of the language are used in practice, the code is not optimized. There are a number of techniques which can be used to greatly improve the efficiency of code on this architecture. These vary from generic techniques to keep more of the PEs busy more of the time, greater overlapping of I/O and processing, through to novel algorithms designed specifically for the architecture.*

2.2 Example 1: Hello world

The canonical example for a new programming language is the *Hello world* program. Not wanting to break this ancient tradition, it is used here.

```
#include <stdio.h>      // Output support

int main() {
    printf("Hello world\n");

    return 0;
}
```

This can be compiled with the following command:

```
cscn hello.cn
```

This compiles the source code in the file `hello.cn` in to an executable file called `a.csx`. This can then be run with the following commands[(1)]:

```
csrun -r a.csx
```

This resets and then loads the executable on to the CSX processor. The program `csrun` will then handle communication from the running program: the message “Hello world” will be displayed and then the program will terminate.

Note that `csrun` will normally search for a CSX processor in your system to run the program on. If you are using a simulator, you will need to specify further command line options (see *Note 1*). See *Section 3.3.1: Running on a simulator* for details of how to use the simulator.

2.3 Example 2: A first poly program

We can also implement a poly variant of this simple program where each Processing Element (PE) outputs a different message. Rather than introduce poly strings and pointers at this point, we will stick with integers.

```
#include <stdiop.h>      // Output support
#include <lib_ext.h>     // Extra functions to support features of
hardware

int main() {
    poly int n;

    n = get_penum();      // individual PE number

    printfp("PE number: %d\n", n); // Output different message per
PE

    return 0;
}
```

---

1. Running code on a simulator

If running code on a simulator, then the `-s` option must be used with `csreset` and `csrun`. This tells it to connect to a simulator, rather than searching for hardware.

This example can be compiled and run in the same way as the previous one. This time we will use the `-o` option to specify the output filename.

```
cscn -o count.csx count.cn
```

This compiles the program, producing an executable file called `count.csx`. This can then be run with the command:

```
csrun count.csx
```

This will print a series of integers from 0 upwards; one per PE.

Note: It is not necessary to reset the processor if it has been reset before and if the previously run program terminated correctly.

3 Building and running C^n programs

This section provides a brief introduction to the main tools in the **C^n** Software Development Kit (SDK). Enough information is provided to get you started, that is, able to compile and run the examples provided.

3.1 A brief description of the tool chain

The SDK contains all the tools necessary to write, compile and run programs on ClearSpeed's CSX processors. The most commonly used tools are:

- `cscn`
  compiles source code to executable programs[1];
- `csreset`
  resets the processor prior to running programs;
- `csrun`
  runs an executable on a CSX processor;
- `csgdb`
  the source code debugger;
- `isim`
  simulator of CSX processor for executing programs in the absence of hardware.

Only basic use of the tools needed to compile the examples are described below. For more details, and for information on the other tools in the SDK, see the *SDK Reference Manual*.

3.2 Compiling program

Once you have written your **C^n** code, you need to run it through the **C^n** compiler. In most cases this is simply a matter of using the `cscn` command with a few basic options and a list of source code files. The most commonly used options are:

`-g`

This option enables debugging support. It generates extra information in the executable file for the debugger. This option *must* be used if you wish to use `csgdb` to debug your program.

`-o` *`filename`*

This option specifies the output filename. If this is omitted the code is written to a file called `a.csx`.

`-h`

This option will display information on all the command line parameters that can be used with `cscn`.

1. The program `cscn` actually invokes a series of other tools to compile your source code. These are: the C pre-processor (`cscpp`), the compiler (`cncc`), the macro assembler (`mass`) and the linker (`cld`).

For larger projects it can be more efficient to run these stages of processing individually, for example with a make file.

For example, if the program being compiled is split across two source files (`main.cn` and `functions.cn`) then the program can be compiled with debug support with the following command:

```
cscn -g -o example.csx main.cn functions.cn
```

The files specified on the command line can be **C^n** source, assembly code or object files. `cscn` will work out what needs to be done with each file based on the file name extension.

3.2.1 File naming conventions

The file name extensions used by the SDK are:

- `.cn` **C^n** language source file
- `.h` **C^n** include file
- `.csx` executable file
- `.csi` pre-processed **C^n** source file
- `.is` assembler source file
- `.inc` assembler include file
- `.s` assembler output from compiler
- `.cso` object file
- `.csa` library file

3.2.2 Compiling as C

Because **C^n** has been based closely on ANSI C, it is often possible to write code that can be compiled and run as either **C^n** or as C. Obviously, if compiled as C, none of the parallel processing or other specific features of the processor will be available, but this can still be a useful technique for debugging code and ensuring that, in 'sequential mode', the code does what is expected.

The predefined macro `__STDCN__` can be used to control a set of include statements and macro definitions to simplify this. For example, a program could have the following at the start:

```
#ifdef __STDCN__
/* Include standard Cn support functions */
#include <lib_ext.h>
#else
/* Macros to allow code to be compiled as C */
#define poly
#define mono
#endif
```

By defining empty macros for the **C^n** keywords mono and poly, these will not be seen by a standard C compiler. Similarly, any **C^n**-specific functions that are used, for example, poly variants of standard functions, could be mapped onto equivalent C functions; for example:

```
#define sinp sin     /* Map poly sine function to standard
equivalent */
```

3.3 Running your code

Once you have created an executable file, it can be run on a CSX processor (or on a simulation of the processor). This requires a program running on the host computer to load the executable code on to the CSX processor and then communicate with it as it runs.

In the simplest case where the program runs almost entirely on the CSX processor, then a program such as `csrun` or the debugger can be used. This loads the code on to the CSX processor and then waits for communication from it.

In the more general case, the whole application may be made up of a host component and a part running on one or more CSX processors—perhaps accelerating some specific function. This is illustrated in *Figure 3*.

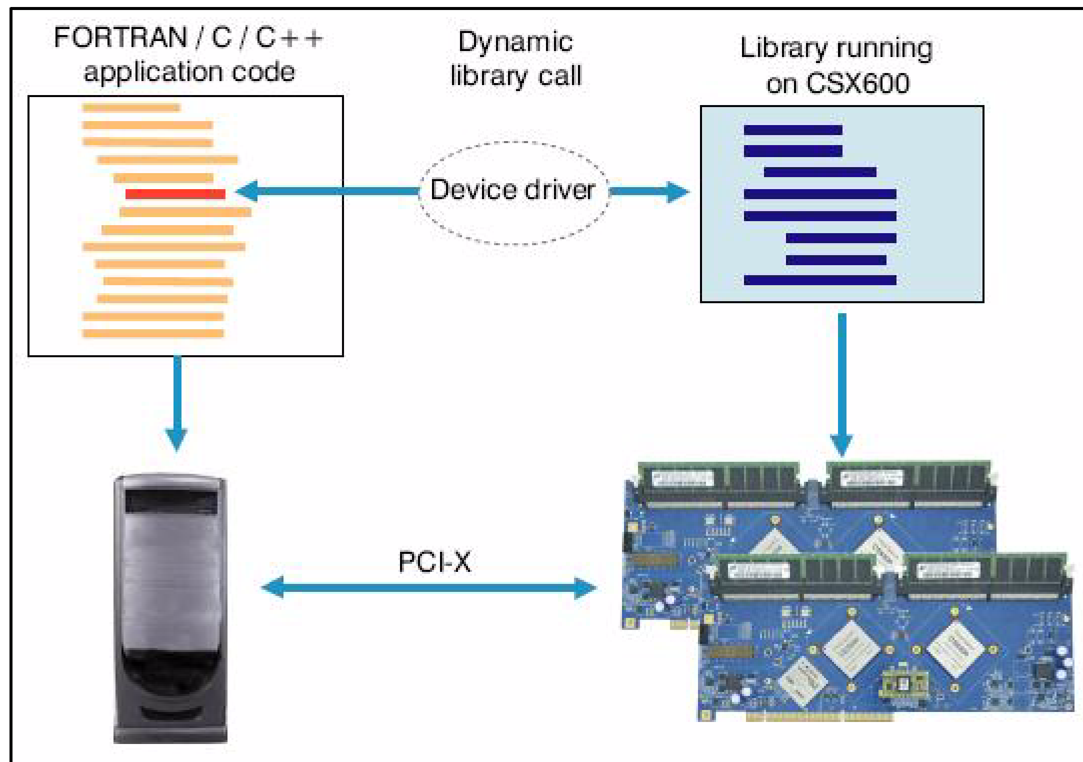

Figure 3. Application acceleration

In order to communicate between the CSX device (or simulator) and the host computer a *device driver* is used. This is a small software module that provides input and output (I/O) services between the host and the CSX processor. The device driver will have been installed as part of the software installation process.

3.3.1 Running on a simulator

If you wish to run your compiled CSX code on the simulator, for example because you do not have access to a CSX processor, then before the CSX code can be run, the simulator must be started in its own command window.

The simulator is run using the command[(1)]:

```
isim
```

When `isim` is running, it does not output any text to the screen until it is accessed by a program such as `csreset`, `csrun` or another host application.

Note: *After `isim` is started, `csreset` must be used to put the simulator into a known state ready for a program to be loaded and executed:*

```
csreset --sim
```

Note that the `--sim` (or `-s`) option is required to tell `csreset` to connect to the simulator rather than looking for hardware.

At this point the simulator is ready to run a program. The executable now needs to be loaded and run. There are two common cases here: the debugger, `csgdb`, or `csrun`.

To simply load and run a program using `csrun`, the command is:

```
csrun -sim filename.csx
```

To run the program with the debugger, the command would be:

```
csgdb filename.csx
```

1. If using Linux, then the simulator can be started in its own window using the command:

```
xterm -e isim &
```

If running Microsoft Windows, the same thing can be achieved with the command:

```
start isim
```

4 Parallel programming in C^n

This chapter explains how a loop in a standard C program can be replaced—or, at least, unrolled—by the use of poly variables. It also shows some basic capabilities of the debugger.

4.1 Sequential code and parallel code

The standard C code and the **C^n** code are shown side by side below. This example assumes a CSX processor with 96 (or more) processing elements.

Sequential C code

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


#define SAMPLES 96

int main() {
    float sine, angle;
    int i;

    // loop over values 0...n-1
    for (i = 0; i < SAMPLES; i++) {
        // convert to an angle in
        // the range 0 to Pi
        angle = i * M_PI / SAMPLES;
        // calculate sine of angle
        sine  = sin(angle);
        // print out values
        printf("%d: %0.3f\n",i,sine);
    }

    return 0;
}
```

Parallel C^n code

```
#include <stdiop.h>
#include <mathp.h>
#include <lib_ext.h>

#define SAMPLES 96

int main() {
    poly float sine, angle;
    poly int i;

    // get PE number: 0...n-1
    i = get_penum();
    // convert to an angle in
    // the range 0 to Pi
    angle = i * M_PI / SAMPLES;
    // calculate sine on each PE
    sine  = sinp(angle);
    // print out values
    printfp("%d: %0.3f\n", i, sine);

    return 0;
}
```

The sequential version of the code simply iterates over a loop and calculates 96 values.

The parallel version uses the PE number to calculate a different value from the series on each PE. Compiling and running each of these programs will produce identical output. However, the parallel version will be executed 96 times faster (ignoring the fact that the run time for this trivial example will actually be dominated by the display of the results).

This is an interesting example to demonstrate some basic capabilities of the debugger (the debugger is described in more detail in *Chapter 7: Debugging Cn*).

4.2 Using the debugger

Firstly, make sure the code is compiled with debug support; for example:

```
cscn -g -o sine_poly.csx sine_poly.cn
```

Then start the debugger:

```
csgdb sine_poly.csx
```

This prints a startup message from the debugger and then the command prompt '`(gdb)`'. Now use the `connect` command to connect to the hardware (or simulator). When successfully connected, this displays the current location of the program counter:

```
(gdb) connect
main() at sine_poly.cn:7
7 . int main(){
```

This output may vary depending upon the format the code was originally entered.

Now use the list command to display the program source. Type return at the prompt after the first group of lines to display the rest of the program:

```
(gdb) list
3       #include <lib_ext.h>
4
5       #define SAMPLES 96
6
7       int main() {
8           poly float sine, angle;
9           poly short i;
10
11          // get PE number: 0...n-1
12          i    = get_penum();
(gdb)
13          // convert to an angle in
14          // the range 0 to Pi
15          angle = i * M_PI / SAMPLES;
16          // calculate sine on each PE
17          sine  = sinp(angle);
18          // print out values
19          printfp("%d: %0.3f\n", i, sine);
20
21          return 0;
22      }
(gdb)
```

Now set a break point before the print statement:

```
(gdb) break 19
Breakpoint 1 at 0x80015180: file sine_poly.cn, line 19.
(gdb)
```

4.3 Start the program

Next start the program running:

```
(gdb) run
Starting program: C:\src\cn\hlpg/sine_poly.csx
Breakpoint 1, main () at sine_poly.cn:19
19          printfp("%d: %0.3f\n", i, sine);
(gdb)
```

You can now use the debugger to examine the state of some variables. Because these are poly variables, a number of values will be displayed, one per PE.

For example:

```
(gdb) print sine
$1 = {0, 0.0327190831, 0.0654031336, 0.0980171412, 0.1305262,
0.162895471,
  0.195090324, 0.227076262, 0.258819044, 0.290284663, 0.321439445,
0.35225004,
...
  0.16289553, 0.130526081, 0.0980170965, 0.0654031485,
0.0327191688}
(gdb)
```

Note: *Some variables may appear to be out of scope because of compiler optimizations, even if you would expect them to still be in scope according to the scoping rules of C.*

Now type `continue` to run to the end of the program. This will print out all the sine values from all PEs:

```
(gdb) next
0: 0.0
1: 0.00
2: 0.01
3: 0.01
.
.
90: 0.020
91: 0.016
92: 0.013
93: 0.01
94: 0.01
95: 0.0

Processor 0 has terminated.
Program exited normally.
(gdb)
```

4.4 Exit the debugger

Now you can use the quit commend to exit the debugger. A more complete tutorial on using the debugger can be found in *Chapter 7: Debugging Cn*.

5 C^n for the working C programmer

The **C^n** language is based very strongly on ANSI C. Any programmer familiar with ANSI C should have no difficulties with the syntax and fundamental concepts of **C^n**. It is recommended that you refer to *[8.] The C Programming Language*. If you are unfamiliar with ANSI C, it is suggested that you spend some time familiarizing yourself with it, since **C^n** introduces some extra concepts for programming parallel systems and that is the main focus of this chapter.

The following sections explain the features found in the **C^n** language.

5.1 Comments

The compiler accepts both standard C-style block comments (`/* ... */`) and the C++ syntax (`// ...`) for line comments.

5.2 Data types

The **C^n** language supports the following basic types:

- char, unsigned char, signed char
- short, unsigned short, signed short
- int, unsigned int, signed int
- long, unsigned long, signed long
- float, double

C^n also supports the following *aggregate* types:

- struct
- union
- pointers
- arrays

These are exactly the same as for ANSI C and should cause no issues for a C programmer.

For details of the sizes and representation of these types, see the *[1.] SDK Reference Manual*.

5.3 Mono and poly specifiers

The **C^n** language adds two extra keywords used in declarations. The new keywords are called *multiplicity specifiers.* The multiplicity specifier allows the programmer to specify the domain in which the declaration will exist.

- **mono**: the declaration exists in the mono domain (one instance), for example, mono variable/memory.
- **poly**: the declaration exists in the poly domain (many instances), for example, poly variable/memory.

The default multiplicity is mono: that is, there will be an implicit mono unless an explicit poly is used.

There are several situations in which the multiplicity specifier can be used. The next subsections will detail these and provide some examples.

5.3.1 Basic types

The basic types can always be used in conjunction with multiplicity specifiers. For instance:

```
poly int counter;
// A different instance of 'counter' exists on each PE
mono unsigned char initial;
// A single instance of 'initial' exists in mono memory
mono unsigned long tval;
poly unsigned long p_tval;
```

5.3.2 Pointer types

Pointers are more complicated than basic types. Pointer declarations consist of a base type and a pointer. The definition on the left-hand side of the `*` represents the base type (the object type that the pointer points to). The definition on the right-hand side of the `*` represents the pointer object itself. The possible combinations should be familiar to most C programmers as it is possible to make either of these entities constant. For instance:

```
const int * const foo;  /* const pointer to const int */
int * const bar;        /* const pointer to non-const int */
const int * bing;       /* non-const pointer to const int */
```

Multiplicity specifiers work in a similar way. Consider the following:

```
poly int * poly foo;   /* poly pointer to poly int */
int * poly bar;
/* poly pointer to mono int (equiv to mono int * poly bar) */
poly int * bing;
/* mono pointer to poly int (equiv to poly int * mono bing) */
```

The best way to understand this is to visualize the poly and mono memory spaces. Imagine the pointers as separate entities in these memory spaces which reference data objects which may themselves be in mono or poly memory space. The diagrams below should make this clearer.

It is also possible to create more complex multiple pointer types such as `mono int * poly * poly` but, as long as you follow the rules for decomposing and visualizing each component of these more complex declarations, they can be understood in the same way.

5.3.3 Pointers to mono data

For mono data, there are two types of pointers:

- *Mono pointer*, that is, a single instance of the pointer and the object pointed to, both in mono memory.
- *Poly pointer*, that is, a pointer on each PE that points to data in mono memory.

Mono pointer to mono data

This is the “normal” C pointer type, for example, `int *` (that is, `mono int * mono`) – there is a single instance of the pointer and of the object pointed to; both are in mono memory. *Figure 4* shows how this can be visualized.

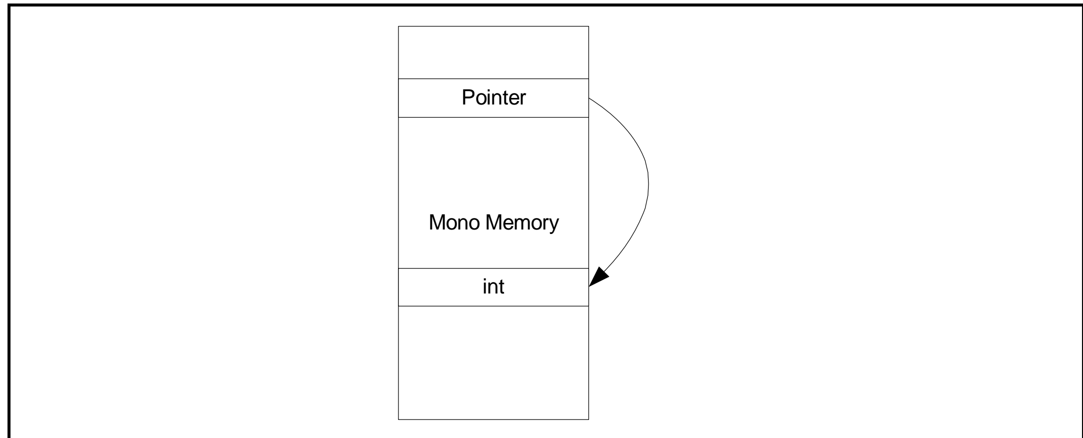

Figure 4. Representation of `mono int * mono`

This type of pointer has exactly the same uses, and behaves in exactly the same way, as in standard C. For example, taking the address of a mono variable will create a pointer of this type.

```
mono int n;
mono int * mono p;

p = &n;     // create a pointer to n
*p = 1;     // assign a value to  n
```

Poly pointer to mono data

It is also possible to have a pointer on each PE that points to data in mono memory. In this instance, the pointer value on each PE could contain a different address; each pointing to a different element of the same array, for example. This is shown in *Figure 5*.

Pointers of the type `mono type * poly` are not used very frequently. The compiler does not currently support de-referencing this type of pointer as this implies moving data from mono space into poly space. Because this data movement might have a significant performance impact, such data transfers must be done explicitly using functions from the standard library (see *Section 5.7.1: Data transfers between mono and poly*). Therefore, the only use for this type of pointer is as an argument to the library functions which perform transfer of data between poly and mono memory.

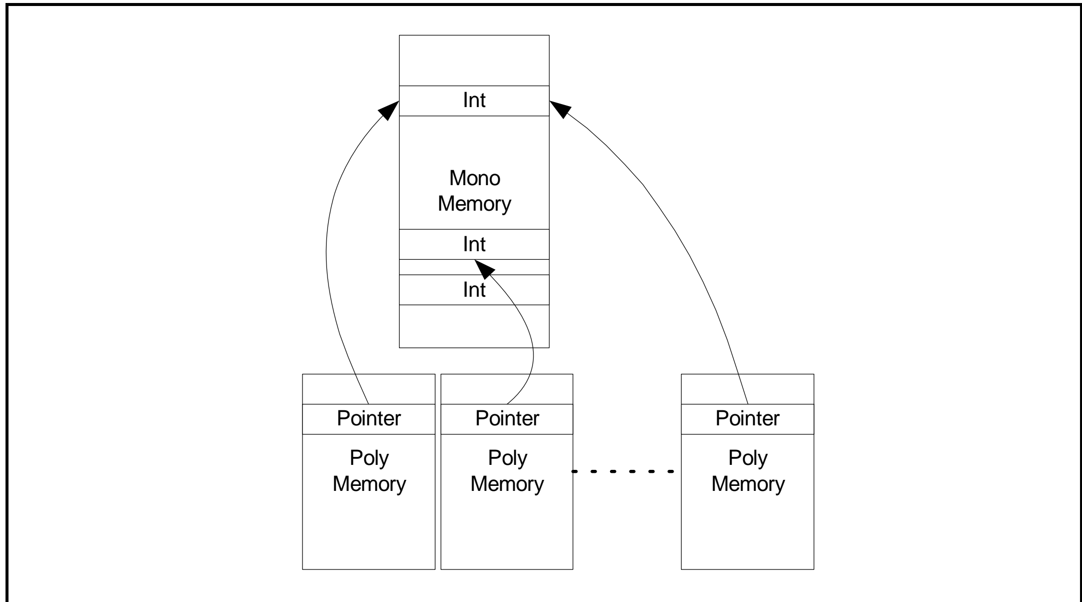

Figure 5. Representation of `mono int * poly`

```
mono int a[ARRAY_SIZE];
poly short i;
mono int * poly p;

i = get_penum()
p = a + i     // each PE points to a different element of the array p
```

5.3.4 Pointers to poly data

For poly data, the same two types of pointers exist:

- *mono pointer* points to the same address in every PE’s memory.
- *poly pointer* can hold a different address on each PE (pointing to data on the same PE).

Mono pointer to poly data

A mono pointer to poly memory (`poly * mono`) is shown in *Figure 6*; here a mono variable provides the address of the data in poly memory.

For example, taking the address of a poly array produces a `poly*mono` pointer: an instance of the array exists on each PE but, because they are all at the same address, a mono value can be used as a pointer to the array.

Code to count the number of bytes with the value zero in a buffer on each PE could be coded as follows:

```
poly char buffer[BUFFER_SIZE]
poly char * mono ptr;
poly int count;
int i;

// initialize
ptr = buffer;
count = 0;
// iterate over the buffer
for (i = 0; i < BUFFER_SIZE; i++) {
    // check value pointed to on each PE
    if (*ptr == 0) {
        count++;
       // increment counter on those PEs where it is zero
    }
    ptr++;    // move the pointer to the next byte
}
```

In this example, the mono variable `ptr` contains a single address—every PE uses this as the address of the next byte to be checked and counted.

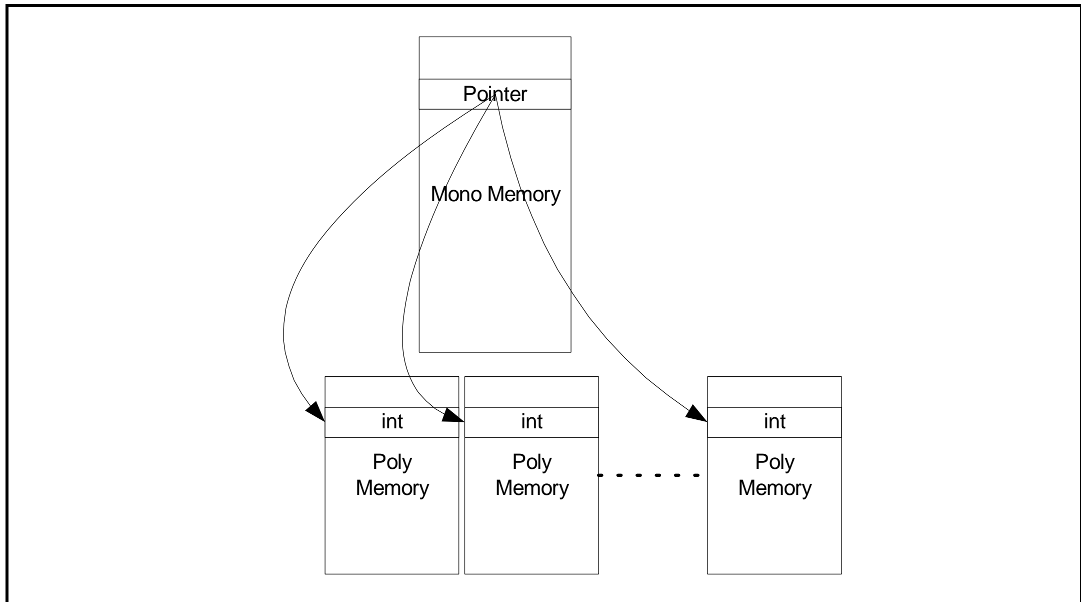

Figure 6. Representation of `mono int * mono`

Poly pointer to poly data

The last pointer type (`poly * poly`) provides even greater flexibility. Here, each PE can calculate a different address for accessing data in its poly memory. This is illustrated in *Figure 6*.

This is an important feature of the language (and of the architecture). It allows you to have each PE operate on different data in a very flexible way. Each PE can calculate the address of the data it is to use rather than having to allocate data to be processed in a fixed way.

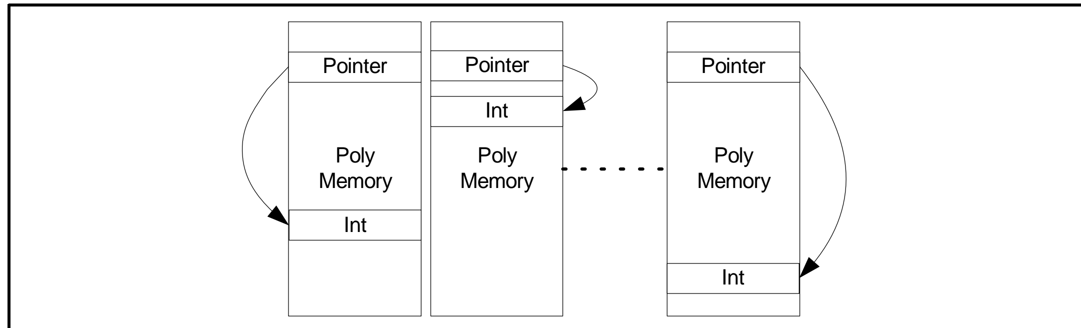

Figure 7. Representation of `poly int * poly`

An example of the use of these pointers is shown below. Here the code is searching for the first position of a character within a string. Each PE can process a different string and so the pointer to the character will be different on each PE.

```
// prototype for poly version of strchr() - from strings.h
poly char * poly strchrp(const poly char * mono s1, poly char c);
poly char str[256];
// the variable str is actually a poly * mono pointer
poly char * poly ptr;     // a pointer into the string

...     // initialize string on each PE
ptr = strchrp(str, 'z');
// search for first occurrence of 'z': different on each PE
```

5.3.5 Illegal casts

Note that, because mono and poly data are in completely separate memory spaces, it is not legal to cast or assign a `mono *` pointer to a `poly *` pointer, or vice-versa.

There would be two problems with attempting to do this sort of cast. Firstly, the mono and poly pointer sizes are not guaranteed to be the same; poly memory is typically quite small and may use 16-bit pointers, while mono pointers may be 32 or 64 bits. Secondly, a pointer to data in poly memory, for example, will not necessarily point to anything meaningful if cast to a pointer to mono memory; it could point to arbitrary data or even code.

5.3.6 Array types

The multiplicity specifier for an array type defines the domain for the base type. For instance, consider the following:

```
poly char buffer[20];
```

In this example the array base type is poly char. This declaration will create space for 20 characters on the poly stack frame. The address of each of these arrays in poly space will be the same. Therefore the type of the pointer that is implied by a reference to `buffer` is `poly char * mono` – a pointer existing in mono space that points to poly space.

Note: *It is not possible to create a `poly char * poly` pointer using array notation since an array declaration only specifies the base type multiplicity class (the implicit pointer multiplicity class will always be mono).*

Multi dimension arrays are supported in **C***n* in exactly the same way as for ANSI C.

5.3.7 Struct and union types

Structures and unions are exactly the same as for ANSI C, however, multiplicity specifiers have strict rules for use inside a struct or union construct. Objects inside a struct/union *definition* have no multiplicity class (it is undefined). It is only when a variable that has the type of the struct/union is *declared* that the multiplicity class is specified. This means that an object that is a struct/union can be declared as either mono or poly. For example:

```
struct _A {
    int a;        //
    char b;
     // Multiplicity class not defined in struct definition
    float c;      //
};

poly struct _A my_struct;
// All objects within the struct are poly
mono struct _A my_struct_2;
// All objects within the struct are mono

poly struct _B {
    int a;
     // Not allowed to declare multiplicity inside definition
    int b;
     // (but statment also declares a poly object)
} my_struct_3;

union _B {
    poly int a;   // Illegal use of multiplicity specifier
    mono char b;  // Illegal use of multiplicity specifier
    float c;
};
mono union _B my_union;
// Multiplicity of declaration would conflict with definition
```

The only situation where a poly or mono specifier can be used in a struct or union is when declaring a pointer. This is due to the fact that the member itself (the pointer) cannot have a multiplicity specifier, but the object pointed to can. This makes sense, since without this capability it would not be possible to have a pointer to a poly object as a member of a struct or union. For instance:

```
struct _C {
    mono int *a;            // pointer to a mono int
    poly char *b;           // pointer to a poly char
};
struct _C my_struct2;       // Note: this is an implicit mono object
                            //   a is mono pointer to mono int
                            //   b is mono pointer to poly char

struct _C poly my_struct3;  // Poly object of the same type
                            //   a is poly pointer to mono int
                            //   b is poly pointer to poly char
```

In the example just given, the first object `my_struct2` contains two members which are `mono int * mono a` and `poly char * mono b`. The second object `my_struct3` contains two members which are `mono int * poly a` and `poly char * poly b`.

5.3.8 Typedefs

As with ANSI C, **C^n** supports `typedef`. This allows the programmer to define their own types. The `typedef` statement cannot use multiplicity specifiers to define the multiplicity of the type. However, as with structs and unions, it can define *pointers* to mono or poly types. For instance:

```
typedef poly int p_int;      // illegal use of multiplicity specifier
typedef poly int * p_ptr;    // 'p_ptr' is a pointer to poly int
typedef mono int * m_ptr;    // 'm_ptr' is a pointer to mono int
p_ptr a;                     // poly int * mono a
poly p_ptr a;                // poly int * poly a
m_ptr a;                     // mono int * mono a
poly m_ptr a;                // mono int * poly a
```

5.4 Mixing mono and poly variables

It is generally legal to mix mono and poly variables in expressions. The basic rules are outlined here.

1. A mono value can be assigned to a poly variable:

   ```
   poly int x = 1;
   ```

   In this case, the mono value is copied to all instances of the poly variable.

2. An expression can mix mono and poly variables:

   ```
   poly int x;
   int y;
   x = x + y;
   ```

   In this case, the mono variable is promoted to a poly (that is, an instance is created on every PE—every instance having the same value) and then the expression is evaluated concurrently by all the PEs.

   The compiler will be able to optimize this so that the mono variable is not copied to poly memory every time it is used.

3. It is *not* legal (or even meaningful) to assign a poly to a mono variable as this means that multiple values would have to be written to a single variable.

5.5 Flow control

C^n supports the same basic flow control statements as ANSI C with some minor additions. The basic flow control constructs are:

- `if` statements,
- `for` loops,
- `while` loops,
- `do .. while` loops,
- `switch` statements (not supported for poly expressions).

Each of these statements uses expressions as loop or branch control. The difference with **C^n** is that these control expressions can be of mono or poly type. (Any expression has a resultant type which, in the case of **C^n**, can have a multiplicity).

5.5.1 If statements

As mentioned above, the expression used in any flow control statement can be either mono or poly. The **C^n** mono variant of these expressions is the same as standard C. Consider the following snippet:

```
int i;
...
if (i > 100) {
    ...    /* do some work */
}
else {
    ...    /* do some other work */
}
```

As a C programmer, you should be familiar with the mechanics of this code. If the condition is met, then the first branch of the `if` statement is executed, otherwise the `else` branch is executed. This is the same whether the statements in the two branches operate on mono or poly data.

Poly conditionals

Now consider the case with a poly expression for the condition:

```
poly short penum;
...
penum = get_penum();
/* Each PE now contains a different value in the penum variable */

if (penum < 32) {
     ... /* Do some work */
}
else {
     ... /* Do some different work */
}
```

In this case, some PEs will execute the first branch, and some will execute the second. Remember, however, that there is a single instruction stream executed by all PEs. So what actually happens is that each PE is *enabled* for the instructions where the condition is true, and *disabled* for the branch where the condition is false. However, because there is a separate mono execution unit (which is always enabled) any mono operations will **always** be executed, whichever branch they are in.

In general, if a poly expression is used for flow control, then the instructions for all alternatives will be issued. The instructions for each branch will be executed on a different subset of the PEs.

This has important implications for mono objects that are used inside poly flow control statements. Consider the following code snippet:

```
poly short penum = get_penum();
mono int i;

if (penum < 32) {
     ...      /* Do some work */
     i = 0;   /* Set mono variable */
}
else {
     ...      /* Do some other work */
     i = 1;   /* Set mono variable */
}
```

What is the value of `i` after this statement has finished? It will, in fact, be 1. Operations on poly data are controlled by the condition in the `if` statement, but mono operations are not because the code for both branches is executed, the variable `i` is initially set to 0 and then to 1.

This is not always intuitive and can cause unexpected results for the unwary programmer. Even if all the poly variables used in the `if` statement fulfil the first condition, the mono variables are still updated as if **both** branches had been executed.

Consider the following piece of code:

```
poly int value = 0;
// Even though value set to 0, both branches will still execute
mono int i;
if (value == 0) {
    i = 1;
}
else {
    i = 2;
}
```

In this case, even though all of the PEs fulfill the condition (and hence none will execute the `else` clause), the compiler will emit code to execute both branches and the processor will execute it. In fact, by writing code like this you are restricting the possibility of optimizing the code, since both branches must be executed. (If the branches contain only operations on poly data, then the compiler can, in principle, remove "dead code" that can never be executed.)

5.5.2 For, while and do..while loops

Similar rules to the `if` statement apply to the `for`, `while` and `do..while` loops. In the same way that both branches of a poly conditional are executed, a loop with a poly control value will be executed as long as the condition is true on *any* PE. Those PEs which evaluate the condition as false will be disabled for the remaining iterations. But, again, mono operations will be executed on *every* iteration.

Consider the following piece of code:

```
poly int i;
mono int loop_count = 0;

...
/* i is set in this piece of code, the value may be different on
/* each PE */
while (i > 100) {
    //...           /* Do some work */
    i -= 2;         /* Decrement poly loop control */
    loop_count++;   /* Increment mono loop count */
}
```

The value of `loop_count` will depend on the maximum value of `i` in the poly space (that is, the number of times the loop is executed on any PE). In this case, `loop_count` is a useful value since it tells the programmer the maximum number of iterations that the loop actually went through.

5.5.3 Goto statement

The `goto` statement is supported in **C*n*** with the restriction that a `goto` statement can not cross a poly boundary. For example, anything which changes enable state such as a `poly if` or a `poly while`.

5.5.4 Labeled breaks

To provide a similar level of control to the use of `goto`, the `break` and `continue` statements are extended in **C*n***. Labels are allowed in **C*n*** but only on loop constructs (`for`, `while`, `do..while`). Break and continue statements in **C*n*** can then specify a label which allows the program to break out of heavily nested loops.

For instance:

```
for_i:
    for (i = 0; i < 10000; i++) {
     // Label for_i is associated with the for loop
        while(j > 100) {
            do {
                // ...
                if (foo == bar) {
                    break for_i;
                }
                // ...
            } while (a != b);
        }
    }
```

In this example, the `break` will break out of the deepest point in the nested loop to the outermost level (breaking completely out of all the nested loops).

This gives most of the flexibility of `goto`, but in a more structured way.

5.5.5 Switch statements

Switch statements are supported in **C*n***. They provide the same functionality as for ANSI C. Switch statement expressions must be mono expressions.

```
int val;
...                 /* Some code which sets up the value in val */
switch (val) {      /* Only mono expressions are valid for switch */
case 0:
case 1:
    ...  /* Do some work */
    break;
case 2:
    ...  /* Do other work */
    break;
default:
    ...  /* etc. */
}
```

An equivalent for poly expressions can be constructed using if statements. Consider the following section of code performing the same functionality as a poly switch statement in **C*n***:

```
poly int val;
...                /* Some code which sets up the values in val */
if ((val == 0) || (val == 1)) {
/* Select operations to be done on each PE */
    ...  /* Do some work */
}
else if (val == 2) {
    ...  /* Do other work */
}
else {
    ...  /* etc. */
}
```

5.6 Functions

Functions are fundamentally the same in **C*n*** as they are in ANSI C.

C*n* supports function pointers.

Multiplicity specifiers can be used with the arguments and the return type of a function. The return type defines multiplicity of the function: that is, a function returning a poly type is referred to as a poly function.

The multiplicity of a function affects how returns are handled. This is analogous to the way conditional statements are handled.

- A **mono** function will return immediately a `return` is executed.
- A **poly** function will only return once all of the PEs have returned a value and all remaining mono code has executed. In effect, a poly function will execute to the end with the PEs that have executed the return statement disabled. If there is no mono code, and all PEs have returned, then a compiler may be able to optimize by returning early.

5.7 Other architectural features

The following sections describe some important features of the architecture which are not directly supported in **C*n***. These can be accessed through library functions or by programming in assembly language. Examples of using these can be found in *Chapter 6: More Cn programs*.

5.7.1 Data transfers between mono and poly

A set of input and output (I/O) operations are defined to support the movement of data between mono and poly memory spaces. There are a number of different modes of operation for these I/O transfers. The most commonly used variants are made available as library functions.

The library functions are extensions of the standard `memcpy` functions. These support mono to poly and poly to mono transfers using both address mode and strided mode. The functions are summarized below[1]:

`memcpym2p` Transfers data from mono space to poly space. Each PE can individually specify the source address (addressed mode). Every enabled PE transfers the same amount of data to the same location in poly memory. Disabled PEs do not take part in the transfer.

`memcpyp2m` As above, but transferring data from poly to mono memory.

`memcpym2p_strided` This function transfers data from mono to poly using strided mode: the starting address in mono memory is specified; this is then incremented by the specified *stride* value for each PE's data. Every enabled PE transfers the same amount of data to the same location in poly memory; disabled PEs do not take part in the transfer.

`memcpyp2m_strided` As above, but transferring data from poly to mono memory.

In addition, there are *asynchronous* versions of these functions which execute the I/O on a separate thread so that it can proceed concurrently with computation being performed on the processor. These functions also use semaphores, described below, to synchronize the completion of I/O with program flow.

Note: There are some restrictions on asynchronous functions. For example, size of transfer and alignment. See the Standard Library Reference Manual for further details.

Caching and I/O

Caution: It is important to be aware of the way that the cache is used for mono data when using the I/O functions.

Normally, accesses by a program to mono memory are cached to provide faster access to frequently used data. However, I/O transfers to and from the PEs do not go via the cache—this could lead to unexpected behavior unless efforts are made to keep the contents of the cache and external memory consistent. The `memcpy` functions described above do this automatically; however, the asynchronous versions *do not*.

The function `dcache_flush` can be used to ensure that the contents of the data cache are consistent with mono memory. This should be used if your program mixes normal accesses to mono memory with the I/O functions.

Semaphores

Semaphores are a standard method for synchronizing multiple concurrent operations. A semaphore is a status word used to control access to a shared resource or to control the execution of concurrent tasks.

1. Please refer to the *SDK Reference Manual*, section *Assignment*.

A semaphore is a non-negative integer value and two associated operations:

- Signal

  A signal is an *atomic*[(1)] operation that increments the semaphore value.
- Wait

  Wait is an atomic operation that decrements the semaphore value.

  The value cannot be negative and so, if the semaphore is zero when the wait is executed, the wait operation will not finish until the semaphore is signalled by another thread. The waiting thread may be descheduled allowing other threads to run.

Valid user semaphore numbers range between 0 and 92. Semaphores 93-127 are reserved for system use and should not be used.

As an example of the use of semaphores, consider the asynchronous I/O functions described above. These perform the I/O concurrently with the program that calls them: they use semaphores to indicate when the I/O has completed. This is shown diagrammatically in *Figure 8*. This shows a compute thread that performs a couple of asynchronous I/O operations.

The semaphore is initialized with the value 0 (**A**). When the first I/O completes, the semaphore is signalled (**B**). The semaphore value of 1 indicates that there is data available. When the compute thread wishes to use the data, it must first wait on the semaphore; in this case the data is available and the thread continues immediately (**C**). The program then initiates a second I/O operation (**D**) and is ready for the data before the operation has completed (**E**). The compute thread is suspended until the I/O operation completes and signals the semaphore (**F**).

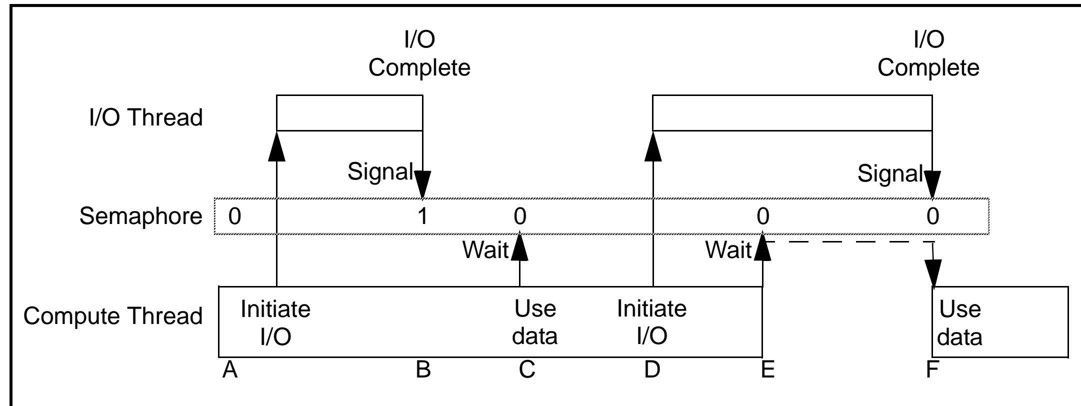

Figure 8. Synchronization of I/O and compute with semaphores

Note: *The program could have multiple data buffers so it can read multiple data items ahead of when they are required. In this case the semaphore value will be incremented for each completed I/O and so indicate the number of data items available to be processed.*

1. An atomic operation is one that cannot be interrupted: that is, in this case, the semaphore value is read, incremented and written back as a single operation. This ensures that behavior is consistent even when multiple threads access the same semaphore.

6 More C^n programs

This chapters gives several examples of how to use **C^n** programs and it also provides a few exercises for you to do.

6.1 Example 1: Mandelbrot set

The Mandelbrot set *The Mandelbrot Set Explorer* is an example of fractal geometry discovered by Benoit Mandelbrot in the 1970s. It is a simple but compute intensive algorithm that produces complex and beautiful images (see *Figure 9*).

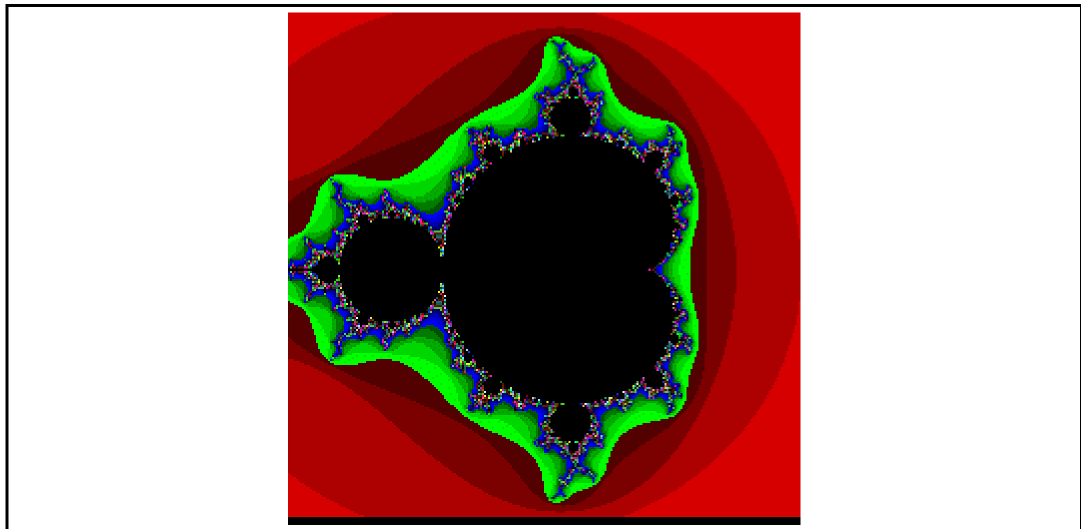

Figure 9. Example of a Mandelbrot set

The algorithm used to generate the Mandelbrot set can be represented by the following pseudo-code; for a given point x, y in the complex plane:

while $(x^2 + y^2) \leq 4$ **and** *number of iterations* < *threshold value*

result = *result* + 1

$x = x^2 + y^2 - x_{orig} \quad ; \quad y = 2xy + y_{orig}$

note: these two assignments are concurrent

The usual way of implementing this is to iterate over a series of values of x and y and run through this equation for each. This then gives a result value for each coordinate which is used to represent a color or grey-scale value. The image in *Figure 9* was generated by iterating over values of x from -1.5 to 1.0 and values of y from -1.5 to 1.25. The resolution of the image is 256 x 256. A false-color map has been applied to the image to make it more interesting.

6.1.1 Standard (mono) implementation[1]

Given this information, you can now devise a standard C program to generate a Mandelbrot set. This code can be compiled and run on any standard processor as well as a CSX processor; but obviously without exploiting the parallelism.

First, define macros for constant parameters used by the program:

```
#define NUMROWS 96      // Size of image to be generated
#define NUMCOLS 96

#define MINX   -1.5f    // Coordinates of fractal to be evaluated
#define MINY   -1.25f
#define MAXX   1.0f
#define MAXY   1.25f

// Increment for each screen coordinate
#define STEPX  ((MAXX - MINX) / NUMCOLS) #define STEPY  ((MAXY -
MINY) / NUMROWS)
#define RES    150       // Max number of iterations
```

Include the standard **C*n*** header files.

```
#include <dprint.h>

#include <lib_ext.h>
```

Define a function to test the termination condition for a given (x,y) coordinate.

```
/* Evaluate the termination condition */
int terminate(float x, float y) {
    return (x*x + y*y > 4.0f);
}
```

The main part of the work is done in the function `calcres()` which evaluates the result at position (x,y).

```
/* Calculate the result for a given x and y position */
char calcres(float x, float y, int res) {
    char result;
    int turnedon;
    int i;
    float tx;
    float xcalc, ycalc;

   /* Set the xcalc cumulative value to its initial value */
    xcalc    = x;
   /* Set the ycalc cumulative value to its initial value */
    ycalc    = y;

    result   = 0;
   /* Initialize flag to control iterations */
    turnedon = 1;

    /* Loop up to res times for each position (this will determine) the
```

1. Please refer to the *Cn Standard Library Reference Manual* for more information.

```
     quality of the final picture, the more iterations, the better
     but slower the solution will be) */
    for (i = 0; i < res; i++) {
        /* Only continue calculating if the result has not been
       determined yet */
        if (turnedon) {
          /* Check to see if the termination condition is met */
            if (terminate(xcalc, ycalc)) {
               /* Final result is the number of iterations required
              to terminate */
                result   = i + 1;
                turnedon = 0;
            }
            else {
                /* Set the values for the next iteration */
                tx    = xcalc * xcalc - ycalc * ycalc + x;
                ycalc = 2.0f * xcalc * ycalc + y;
                xcalc = tx;
            }
        }
    }
    return result;
}
```

The main body of the program: initializes variables, iterates calculating the Mandelbrot set one row at a time and printing the results.

```
int main() {
    float x, y;
    int col_count, row_count;

    /* Create some space on the stack for the results */
    /* We calculate a row at a time, hence we only need a buffer for
     one row */
    char buffer[NUMCOLS];

    /* Iterate over the rows */
    for (row_count = 0; row_count < NUMROWS; row_count++) {
        /* Iterate over the columns */
        for (col_count = 0; col_count < NUMCOLS; col_count++) {

            x = MINX + col_count * STEPX;
            // Calculate the x value for this position
            y = MAXY - row_count * STEPY;
            // Calculate the y value for this position

            /* Calculate the value for this position */
            buffer[col_count] = calcres(x, y, RES);
        }
        /* Print out the current row */
        dprint_mono_memory_raw( buffer, NUMCOLS);
    }
```

```
    return 0;
}
```

Exercise 6.1: Get this program to compile and run under a standard compiler and operating system. You will need to comment out the include at the top of the program and come up with a definition of the `dprint_mono_memory_raw()` function (*hint*: this function simply prints out a series of 4-byte values in hexadecimal, from a given memory location, to the console window).

Exercise 6.2: Get this program to compile and run under the **C***n* compiler. (*hint*: *Chapter 3: Building and running Cn programs*, contains instructions on how to do this. The library and include files used in this example are included with the SDK distribution).

Exercise 6.3: Can you see a way to optimize the performance of this code? (*hint:* The `calcres()` function will sometimes iterate unnecessarily around the loop when the termination condition has already been reached).

6.1.2 Parallel (poly) implementation

The next step is to make use of the poly (parallel data) capabilities of the processor. Obviously, you are iterating over each of the coordinates in a serial fashion. In the example as shown, there are 96x96 coordinates in the image. If you use a processor with 96 processing elements, it makes sense to exploit these to calculate an entire row or column in parallel.

The change that will be made is to calculate an entire column(1) of the image in one go. Each PE is allocated a row of the complete image. As the program iterates through the columns, every PE calculates the pixel value for its row simultaneously.

The `terminate()`and `calcres()` functions need to be modified to operate on poly variables. They now look like this:

```
/* Evaluate the termination condition in parallel */
poly int terminate(poly float x, poly float y) {
    return (x*x + y*y > 4.0f);
}

poly char calcres(mono float x, poly float y, mono int res) {
    poly char result;    // Different result on each PE
    poly int turnedon;
     // Each PE does a different number of iterations
    int i;               // but they all go round the same loop
    poly float tx;       // Intermediate values for each PE
    poly float xcalc, ycalc;
```

1. The only reason we generate *columns* in parallel (rather than rows) is simply because it makes it easier to print out the generated data.

```
    /* Set the xcalc cumulative value to its initial value */
    xcalc    = x;
    /* Set the ycalc cumulative value to its initial value */
    ycalc    = y;

    result   = 0;
    /* Initialize flag to control iterations */
    turnedon = 1;

    /* Loop up to 'res' times for each position (this will determine
     the quality of the final picture, the more iterations, the
     better but slower the solution will be) */
(i) for (i = 0; i < res; i++) {
        /* Only continue calculating if the result has not been
determined yet */
(ii)    if (turnedon) {
            /* Check to see if the termination condition is met */
(iii)       if (terminate(xcalc, ycalc)) {
                /* Final result is the number of iterations req. to
                 terminate */
                result   = i + 1;
                turnedon = 0;
            }
(iv)        else {
                /* Set the values for the next iteration */
                tx    = xcalc * xcalc - ycalc * ycalc + x;
                ycalc = 2.0f * xcalc * ycalc + y;
                xcalc = tx;
            }
        }
    }
    return result;
}
```

The main changes to be pointed out are as follows:

- The `terminate()` and `calcres()` functions now use poly arguments and a poly return type. This means that the results are being evaluated on all of the poly processing elements in parallel.
- The loop at (i) remains unchanged from the mono version of the code. The same number of iterations will be done each time.

Note: 1 *This means that the optimization can still be applied. Instead of iterating the maximum number of times, the loop can iterate just as many times as any of the PEs may need it to.*

2 *Although this may seem inefficient, the fact that some PEs may be idle, for a number of iterations is completely outweighed by the increase in performance obtained from the high level of parallelism.*

- The conditional statements (ii) and (iii) are now poly conditionals. This means that the result of these conditions may be different on each of the PEs.
- Both the branches of the inner `if` statement (iii) and (iv) will be executed. Some of the PEs will execute the first branch (iii) and some will execute the `else` clause (iv).

There are also some modifications to the `main()` function to set up the PEs so that they can perform in parallel.

```
int main() {
    mono float x;
    poly float y;
    int col_count;

    /* Create some space on the stack for the results */
    /* We iterate in x and evaluate a different y position on every
       PE simultaneously, so we need a one-row buffer on each PE:
       i.e. a poly buffer */
    poly char buffer[NUMCOLS];

    /* Iterate over the columns evaluating a complete column each
       time */
    for (col_count = 0; col_count < NUMCOLS; col_count++) {

        x = MINX + col_count   * STEPX;
       // Calculate the x value for this position
        y = MAXY - get_penum() * STEPY;
       // Calculate the y value for this PE

        /* Evaluate this column for 96 rows of the image in one step
*/
        buffer[col_count] = calcres(x, y, RES);
    }
    /* Print out all rows */
    dprint_poly_memory_raw(buffer, NUMCOLS);

    return 0;
}
```

Important points to note are:

- The use of a poly array for the rows of results. There is an instance of this on every poly processing element—every PE calculates a value for its position in the column simultaneously.
- The `get_penum()` library function returns the PE number (ranging in this case from 0 to 95). This allows the program to set up a unique y co-ordinate for each of the PEs.
- The `dprint_poly_memory_raw()` function prints out the values from the memory of each PE in turn.

Exercise 6.4: Compile and run the program using the **C*n*** compiler. (*hint*: this should be no different than the process you went through for the mono variant).

Exercise 6.5: Apply the equivalent optimization to the `calcres()` function. Think about what this implies for the behavior of the program (*hint*: there is a feedback mechanism to tell the control unit when all the PEs are disabled). See *The **C*n*** Standard Library Reference Manual* for a list of functions.

Exercise 6.6: Can you modify this code to cope with images larger than 96x96? (*hint*: You can use more than 96 bytes per poly processing elements to contain the results). Consider processing an 8x8 square of pixels.

6.2 Exercise 2: Input and output

It is frequently necessary to transfer data between mono and poly memory, for example, to distribute data across the PE array for processing. The processor has a number of input/output (I/O) modes[1] to support this sort of data movement. These are made available via functions based on the standard `memcpy()` library functions.

The following program provides a simple example of the use of the I/O capabilities of the processor. This uses the library functions to load array-sized *chunks* of data into PE memory from external (mono) memory. It does some trivial processing and then writes the results back to mono memory.

```
#include <string.h>
#include <dprint.h>
#include <lib_ext.h>

#define DATA_SIZE 150

int main () {
    // buffers to store the data being processed
    mono float input_data[DATA_SIZE], output_data[DATA_SIZE];

    poly float element;
     // the values being processed on each PE
    mono float scale;
     // the scaling factor applied to each value

    // pointers to input and output data
    mono int * poly src_addr, * poly dst_addr;

    // loop control variables
    mono int i, count, array_size;
    poly short penum;

    // initialize
    array_size = get_num_pes();
    penum      = get_penum();
    scale      = 3.0f;
    count      = DATA_SIZE;
    for (i = 0; i < DATA_SIZE; i++) {
        input_data[i] = (float) i;
    }
```

1. See the *CSX600 Hardware Programming Manual* for more details.

```
    for (i = 0; i < DATA_SIZE; i += array_size) {
        // process only remaining elements
        if (penum <= (count - 1)) {
            // set up external memory address to read from for each
              PE
            src_addr = input_data + i + penum;

            // copy the data to be updated into the array
            memcpym2p(&element, src_addr, sizeof(float));

            // do the processing on all data in parallel
            element *= scale;

            // calculate address to write each PE's results to
            dst_addr = output_data + i + penum;

            // copy the results back out to mono memory
            memcpyp2m(dst_addr, &element, sizeof(float));
        }
        // count how many data elements left
        count -= array_size;
    }

    // print out the results
    for (i = 0; i < DATA_SIZE; i++) {
        dprint_mono(FLOAT, output_data[i]);
    }

    return 0;
}
```

This is not an ideal implementation of the algorithm, however, because it does not overlap the I/O and compute operations. For example, when data is read from mono memory using the `memcpym2p` function, the PEs are idle while waiting for the data transfer to complete. When there are a large number of PEs or a large volume of data is being read, this can become a significant loss of efficiency.

One of the main reasons for supporting multithreading in the architecture is to allow compute and I/O to be efficiently overlapped. This is made available to programmers through the asynchronous I/O functions described next.

6.3 Example 3: Asynchronous I/O

The asynchronous I/O functions are similar to the functions used previously but execute on a separate thread, allowing the compute thread to continue running. To synchronize the completion of a read or write operation with the use of the data in a compute thread, the functions take an extra parameter which is the ID of a semaphore used to signal completion.

This example extends the code above to use asynchronous I/O. To do this you need to introduce two buffers for the data being processed on the PEs. One buffer (the *active* buffer) can be processed while the next buffer (the *background* buffer) is being filled, then the active buffer is output while the next is processed, and so on.

This means that the modified program will be as follows:

1. Read into the active buffer.
2. Wait for the background buffer to be empty then read the next chunk into it.
3. Wait for the active buffer to be full then process the contents.
4. Write out the active buffer.
5. Swap the active and background buffers.
6. Repeat from Step 2 until all chunks are processed.
7. Wait for the write of the last active buffer to finish then print the results to the console.

The important thing to remember here is that the reading and writing functions return immediately while the I/O operation continues in the background—in parallel with any subsequent processing. For simplicity, this algorithm assumes that you have a minimum of two chunks to process and that the total amount of data to process is a multiple of the array size.

You will also notice that there are several points where you wait for buffers to fill or empty. Semaphores are used to ensure that a read has completed and therefore the data can be processed, or that a write has completed and so the buffer is empty and can be used for something else. The semaphores are signalled, implicitly, by the asynchronous read and write functions. The code has to explicitly wait on the appropriate semaphore before using the buffers. Two semaphores are used: one to synchronize reading a buffer and processing that data; the second to synchronize writing the data with refilling the buffer.

Valid semaphore numbers range between 0 and 92. Semaphores 93-127 are reserved for system use and should not be used.

The following example illustrates proper usage with a `dcache_flush` call to force the write back from the 4K data cache associated with mono memory of the `input_data` and `output_data` arrays before being used by the `async_memcpym2p` and `async_memcpyp2m` routines respectively.

Caution: Omission of the `dcache_flush` call results in incorrect behavior. Unlike the synchronous `memcpy` transfers described in section *Section 5.7.1 on page 33*, asynchronous memory copies do not guarantee cache coherency with mono memory. To ensure cache coherency with mono memory, use the data cache write back routine (`dcache_flush`) as described in *[2.] The Cn Standard Libraries Reference Manual*.

```
#include <string.h>
#include <lib_ext.h>
#include <dprint.h>
#include <stdio.h>

#define DATA_SIZE          192

//arrays to store the input and output data
#pragma align 8
mono float input_data[DATA_SIZE];
#pragma align 8
mono float output_data[DATA_SIZE];

int main() {

  //the values being processed on each PE (double buffered)
  poly float element[2];

  //pointers to input and output data on each PE
  mono int * poly src_addr;
  mono int * poly dst_addr;



  //array index values for managing the double buffering
  mono short background = 0, active = 1;
  mono unsigned int iterate=1;

  //the scaling factor applied to each value
  mono float scale;

  //loop control variables
  mono int i, count, array_size;

  //semaphores
  mono short READ_SEMAPHORE_a =  1;
  mono short READ_SEMAPHORE_b =  2;
  mono short WRITE_SEMAPHORE  = 10;

  poly short penum;

  //initialize
  array_size = get_num_pes();
  penum      = get_penum();
  scale      = 2.0f;
  count      = DATA_SIZE;

  //set up addresses to copy to/from on each PE
  src_addr   = input_data + penum;
  dst_addr   = output_data + penum;
```

```
  //initialize input and output data
  for (i = 0; i < DATA_SIZE; i++) {
    input_data[i]  = (float)i;
    output_data[i] = -1.0f;
  }

  //ensure data is written out from cache
  dcache_flush();

  //read some data into the 'active' buffer
  async_memcpym2p(READ_SEMAPHORE_a, &element[active], src_addr,
sizeof(float));

  //use semaphore to flag the next input buffer as empty
  sem_sig(WRITE_SEMAPHORE);

  do {
    //count how many data elements left to process after
    //completion of current iteration
    count -= array_size;
    iterate = (count > 0);

    //wait for the previous 'active' buffer to empty
    sem_wait(WRITE_SEMAPHORE);

    //read the next set of data into the 'background' buffer
    if (iterate) {
      //move pointer to next data to process
      src_addr += array_size;
      async_memcpym2p(READ_SEMAPHORE_b, &element[background],
src_addr,sizeof(float));
    }

    //wait for the 'active' buffer to fill
    sem_wait(READ_SEMAPHORE_a);

    //do the processing on all data in parallel
    element[active] *= scale;

    //copy the results back out to mono memory
    async_memcpyp2m(WRITE_SEMAPHORE, dst_addr, &element[active],
sizeof(float));

    //move pointer to next output data position
    dst_addr += array_size;

    //swap the 'active' and 'background' buffer poitners over
    {
      short t,tmp_SEMAPHORE;
```

```
        tmp_SEMAPHORE     = READ_SEMAPHORE_a;
        READ_SEMAPHORE_a = READ_SEMAPHORE_b;
        READ_SEMAPHORE_b = tmp_SEMAPHORE;

        t = active;
        active = background;
        background = t;
      }
    } while (iterate);

    //wait for the last 'active' buffer to empty
    sem_wait(WRITE_SEMAPHORE);

    //print out the results
    for (i = 0; i < DATA_SIZE; i++) {
      dprint_mono(FLOAT, output_data[i]);
    }

    return 0;
  }
```

Exercise 6.7: Extend the program to work with any amount of data.

7 Debugging C^n

The CSX SDK includes a port of the standard GDB debugger.

7.1 Compiling for debug

To use the debugger the source code needs to be compiled with the `-g` option to generate debug information, which is stored in the executable. The compilation command line should be as follows:

```
cscn -o mandelbrot.csx -g mandelbrot_poly.cn
```

7.2 Starting csgdb

The debugger is started from the command line, passing it the executable name as a command line argument. This allows it to read the symbol table from the executable when it starts up. You start the debugger as follows:

```
csgdb mandelbrot.csx
```

The `csgdb` interface will start up and wait at the prompt for a command.

```
GNU gdb 6.5 (1.29.1.6 beta at 24-10-2007 16:11 on win32_i386)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "--host=i686-mingw32 --target=cs-
clearspeed-elf"...
(gdb)
```

7.3 Using csgdb to investigate mandelbrot_poly.cn

7.3.1 Listing source code

As you have started up `csgdb` with the executable name on the command line it will already have stored the information relating to source code file names and locations. This means that you can view the source code for the Mandelbrot application straight away.

You can now list the first 10 lines around the function `main` by using the command `list main`.

```
(gdb) list main
52        }
53      }
54      return result;
55    }
56
57    int main() {
58      mono float x;
59      poly float y;
60      int col_count;
61      /* Create some space on the stack for the results */
(gdb)
```

Pressing enter three more times displays the rest of the source code for the function `main`:

```
62    /* We iterate in x and evaluate a different y position on every
PE
63    simultaneously, so we need a one-row buffer on each PE: i.e. a
poly buffer */
64    poly char buffer[NUMCOLS];
65    /* Iterate over the columns evaluating a complete column each
time */
66    for (col_count = 0; col_count < NUMCOLS; col_count++) {
67      x = MINX + col_count   * STEPX;  // Calculate the x value for
this position
68      y = MAXY - get_penum() * STEPY;  // Calculate the y value for
this PE
69      /* Evaluate this column for 64 rows of the image in one step
*/
70      buffer[col_count] = calcres(x, y, RES);
71    }
(gdb)
72      /* Print out all rows */
73      dprint_poly_memory_raw(buffer, NUMCOLS);
74    }
(gdb)
```

The debugger can be configured to display more than 10 lines at a time using the `set listsize` command but for this example leave it at the default.

7.3.2 Connecting to the device

To start debugging the running code you have to connect to the processor running the code using the `connect` command:

```
(gdb) connect
main() at mandelbrot.cn:57
57 int main() {
(gdb)
```

The debugger is now connected to the target device and ready to start running code. The device is currently stopped at the reset address.

7.3.3 Setting breakpoints

Breakpoints can be set on a function name, source code line or an address in memory. Start by setting a breakpoint at `main` and a temporary breakpoint at line 109 of `mandelbrot.cn` using the `break` and `tbreak` commands:

```
(gdb) break main
Breakpoint 1 at 0x020055C8: file mandelbrot.cn, line 66.
(gdb) tbreak 67
Breakpoint 2 at 0x02005638: file mandelbrot.cn, line 67.
(gdb)
```

There are now two breakpoints set in the debugger and their status can be displayed using the `info break` command:

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x020055C8 in main at mandelbrot.cn:66
2   breakpoint     del  y   0x02005638 in main at mandelbrot.cn:67
(gdb)
```

As you can see both breakpoints are set and they are both enabled but breakpoint 2 will be deleted once it is hit by the executing code. There are many things you can do with breakpoints in `csgdb` and some more advanced features are demonstrated later.

7.3.4 Starting execution

The processor is currently stopped at the reset address and you now want to continue it to the point where it hits the breakpoint at `main`. This can be done using the `run` command. The `run` command is a combination of the `load` and `continue` commands.

Enter the `run` command now to start the processor executing:

```
(gdb) run
Starting program: C:\Program
Files\clearspeed\csx600_m512_le\examples
\csapi\mandelbrot1/mandelbrot.csx

Breakpoint 1, main () at mandelbrot_poly.cn:66
66   for (col_count = 0; col_count < NUMCOLS; col_count++) {
(gdb)
```

As you can see `csgdb` has reported that the breakpoint has been hit and that the processor has stopped at line 90. Using the `info break` command you can view the current breakpoint status.

Note: *Breakpoint 1 is now reported as having been hit once.*

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x020055C8 in main at mandelbrot.cn:66
        breakpoint already hit 1 time
2   breakpoint     del  y   0x02005638 in main at mandelbrot.cn:67
(gdb)
```

If you use the `continue` command to restart execution, you will hit the temporary breakpoint set at line 67 in `mandelbrot.cn`:

```
(gdb) continue
Continuing.
main () at mandelbrot_poly.cn:67
67   x = MINX + col_count   * STEPX;  // Calculate the x value for
this position
(gdb)
```

The temporary breakpoint has now been hit and issuing the `info break` command again will show the breakpoint state:

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x020055C8 in main at mandelbrot.cn:66
        breakpoint already hit 1 time
(gdb)
```

The temporary breakpoint has been removed and only the one at `main` remains.

Next you need to get to line 70 and this can be done using the `next` command. Entering the `next` command will run to line 68. Using it again will get us to line 70. Note that a command can be repeated simply by hitting enter:

```
(gdb) next
68   y = MINY + (get_penum() * STEPY); // Calculate the y value for
this PE
(gdb)
70   buffer[col_count] = calcres(x, y, RES);
(gdb)
```

7.3.5 Examining variables

At line 70 the code is just about to call the function `calcres`. The first two arguments to this function are a mono floating point value and a poly floating point value. You can now see how these differ when displayed in `csgdb`. The debugger understands the `poly` keyword and has been extended to allow you to see the values of symbolic variables across all of the processing elements.

First look at the mono value of variable `x`. This can be displayed using the `print` command. The `print` command will display values using the type of the object unless otherwise instructed:

```
(gdb) print x
```

```
$1 = -1.5
(gdb) print/d x
$2 = -1
(gdb)
```

The first output is the floating-point value of `x` and the second is the value cast to an integer.

The poly value `y` can be displayed using the same command. This differs from the mono display by showing the values across all of the processing elements:

```
(gdb) print y
$3 = {1.25, 1.22395837, 1.19791663, 1.171875, 1.14583337,
1.11979163,
  1.09375, 1.06770837, 1.04166663, 1.015625, 0.989583373,
0.963541687,
  0.9375, 0.911458373, 0.885416687, 0.859375, 0.833333373,
0.807291687,
  0.78125, 0.755208373, 0.729166687, 0.703125, 0.677083373,
0.651041687,
  0.625, 0.598958373, 0.572916687, 0.546875, 0.520833373,
0.494791687,
  0.46875, 0.442708373, 0.416666687, 0.390625, 0.364583373,
0.338541687,
  0.3125, 0.286458373, 0.260416687, 0.234375, 0.208333373,
0.182291746,
  0.15625, 0.130208373, 0.104166746, 0.078125, 0.0520833731,
0.0260417461, 0,
  -0.0260416269, -0.0520832539, -0.078125, -0.104166627, -
0.130208254,
  -0.15625, -0.182291627, -0.208333254, -0.234375, -0.260416627,
  -0.286458254, -0.3125, -0.338541627, -0.364583254, -0.390625, -
0.416666627,
  -0.442708254, -0.46875, -0.494791627, -0.520833254, -0.546875,
  -0.572916627, -0.598958254, -0.625, -0.651041627, -0.677083254, -
0.703125,
  -0.729166627, -0.755208254, -0.78125, -0.807291508, -0.833333254,
  -0.859375, -0.885416508, -0.911458254, -0.9375, -0.963541508, -
0.989583254,
  -1.015625, -1.04166651, -1.06770825, -1.09375, -1.11979151, -
1.14583325,
  -1.171875, -1.19791651, -1.22395825}
(gdb) print/d y
$4 = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, -1, -1, -1,
-1, -1}
(gdb)
```

As you can see the value of `y` as a float is different across every processing element; the value of `y` cast to an integer varies between -1 and 1 due to rounding.

You can also find out where the variables x and y are located by taking the address of each variable. This is done in the csgdb command language as in C, by the use of the & operator.

```
(gdb) print &x
Address requested for identifier "x" which is in register $16m4
(gdb) print &y
Address requested for identifier "y" which is in register $8p4
(gdb)
```

7.3.6 Reading registers

The mono variable x is present in register 16:m4 and the poly variable y is contained in register 8:p4. The $ prefix on the register names indicates that they are variables within the debugger command language.

All of the registers used by the source code are available as command language variables. These can be displayed in the same way as variables in the source code by using the print/f command.

The /f format is required when the print command is used with registers to print the values in floating point format as they default to hexadecimal.

```
(gdb) print/f $16m4
$6 = -1.5
(gdb) print/f $8p4
$7 = {-1.25, -1.22395837, -1.19791663, -1.171875, -1.14583337, -
1.11979163, -1.09375, -1.06770837, -1.04166663, -1.015625, -
0.989583373, - 0.963541687, -0.9375, -0.911458373, -0.885416687, -
0.859375, -0.833333373, - 0.807291687, -0.78125, -0.755208373, -
0.729166687, -0.703125, -0.677083373, - 0.651041687, -0.625, -
0.598958373, -0.572916687, -0.546875, -0.520833373, - 0.494791687,
-0.46875, -0.442708373, -0.416666687, -0.390625, -0.364583373, -
0.338541687, -0.3125, -0.286458373, -0.260416687, -0.234375, -
0.208333373, - 0.182291746, -0.15625, -0.130208373, -0.104166746, -
0.078125, -0.0520833731, -0.0260417461, 0, 0.0260416269,
0.0520832539, 0.078125, 0.104166627,
0.130208254, 0.15625, 0.182291627, 0.208333254, 0.234375,
0.260416627, 0.286458254, 0.3125, 0.338541627, 0.364583254,
0.390625, 0.416666627, 0.442708254, 0.46875, 0.494791627,
0.520833254, 0.546875, 0.572916627, 0.598958254, 0.625,
0.651041627, 0.677083254, 0.703125, 0.729166627, 0.755208254,
0.78125, 0.807291508, 0.833333254, 0.859375, 0.885416508,
0.911458254, 0.9375, 0.963541508, 0.989583254, 1.015625,
1.04166651, 1.06770825, 1.09375, 1.11979151, 1.14583325, 1.171875,
1.19791651, 1.22395825}
(gdb)
```

As you can see the poly registers are displayed in the same way as poly source code variables: there is a value for every PE. When operating on poly registers in the command language there is the added advantage of being able to specify which PE's registers you want to display:

```
(gdb) print/f $8p4[5]
$8 = -1.11979163
(gdb)
```

Specifying the PE number in square brackets after the poly register name will display only the value for that PE.

The whole mono and poly register set can also be viewed using the `regs` and `peregs` commands. These commands take an optional size argument and default to the natural register size which is 2 bytes for mono and 1 byte for poly.

Displaying the mono registers as 4-byte registers is done as follows:

```
(gdb) regs 4
pc   0x80015584
ret  0x8001551c
pred 0x0075
0m4  0x80000000
4m4  0x0
8m4  0x0
12m4 0xbfc00000
16m4 0xbfc00000
20m4 0x80016934
24m4 0x0
28m4 0x0
32m4 0x0
36m4 0x0
40m4 0x0
44m4 0x0
48m4 0x0
52m4 0x0
56m4 0x0
60m4 0x80015694
(gdb)
```

Displaying the poly registers is done in a similar way but using the `peregs` command.

7.3.7 Stepping to a function call

To find the current location of the program counter within the source code, you use the `where` command:

```
(gdb) where
#0  main () at mandelbrot.cn:70
(gdb)
```

This informs you that you are presently at line 70 of `mandelbrot.cn` inside the function `main`.

Listing line 70 will give you visibility of the source code around that line. Use the `list` command to display the source code around line 70.

```
(gdb) list
65   /* Iterate over the columns evaluating a complete column each
time */
66   for (col_count = 0; col_count < NUMCOLS; col_count++) {
67      x = MINX + col_count   * STEPX;  // Calculate the x value for
this position
68      y = MAXY - get_penum() * STEPY;  // Calculate the y value for
this PE
```

```
69      /* Evaluate this column for 64 rows of the image in one step
*/
70        buffer[col_count] = calcres(x, y, RES);
71      }
72      /* Print out all rows */
73      dprint_poly_memory_raw(buffer, NUMCOLS);
74   }
(gdb)
```

You are currently stopped at the call to the function `calcres`.

In order to progress the debugger to the function `calcres` you need to use the `step` command. This differs from `next` because it steps into function calls rather than stepping over them. Enter the command `step` at the prompt:

```
(gdb) step
calcres (x=-1.47395837, y=
      {1.25, 1.22395837, 1.19791663, 1.171875, 1.14583337,
1.11979163, 1.09375, 1.06770837, 1.04166663, 1.015625, 0.989583373,
0.963541687, 0.9375, 0.911458373, 0.885416687, 0.859375,
0.833333373, 0.807291687, 0.78125, 0.755208373, 0.729166687,
0.703125, 0.677083373, 0.651041687, 0.625, 0.598958373,
0.572916687, 0.546875, 0.520833373, 0.494791687, 0.46875,
0.442708373, 0.416666687, 0.390625, 0.364583373, 0.338541687,
0.3125, 0.286458373, 0.260416687, 0.234375, 0.208333373,
0.182291746, 0.15625, 0.130208373, 0.104166746, 0.078125,
0.0520833731, 0.0260417461, 0,
-0.0260416269, -0.0520832539, -0.078125, -0.104166627, -
0.130208254,
-0.15625, -0.182291627, -0.208333254, -0.234375, -0.260416627, -
0.286458254,
-0.3125, -0.338541627, -0.364583254, -0.390625, -0.416666627, -
0.442708254,
-0.46875, -0.494791627, -0.520833254, -0.546875, -0.572916627, -
0.598958254,
-0.625, -0.651041627, -0.677083254, -0.703125, -0.729166627, -
0.755208254, -0.78125, -0.807291508, -0.833333254, -0.859375, -
0.885416508, -0.911458254, -0.9375,
-0.963541508, -0.989583254, -1.015625, -1.04166651, -1.06770825, -
1.09375, -1.11979151, -1.14583325, -1.171875, -1.19791651, -
1.22395825}, res=150)
    at mandelbrot_poly.cn:27
27   xcalc    = x;
(gdb)
```

As you can see you have now stopped at the first valid line of code for the function `calcres` which is line 27 of `mandelbrot.cn`. The function `calcres` takes a poly float as its second argument and this is displayed in the output as a poly float.

To limit the output, you can use the `set print elements` command to limit the number of processing element values that are displayed.

Use the command `set print elements 4` to limit the display to the first 4 PEs, then enter the command `where` to display the current back-trace limited to the displayed PEs.

```
(gdb) set print elements 4
(gdb) where
```

```
#0  calcres (x=-1.47395837, y={1.25, 1.22395837, 1.19791663,
1.171875...},
    res=150) at mandelbrot_poly.cn:27
#1  0x02005740 in main () at mandelbrot_poly.cn:70
(gdb)
```

The debugger can show the function call trace from `main` and can traverse up or down the list of functions. As you can see you are now in the function `calcres` at line 27.

To move up the call stack and back to `main` use the `up` command.

Enter the command `up` at the prompt to move the debugger view back out of `calcres` and into `main`.

```
(gdb) up
#1  0x02005740 in main () at mandelbrot_poly.cn:70
70              buffer[col_count] = calcres(x, y, RES);
(gdb)
```

The program is still at line 52 of `calcres` but the debugger is able to reconstruct the source code location and variable values before the function was called.

Print the program counter (PC) at the current location using the `print` command:

```
(gdb) print/x $pc
$8 = 0x800155a0
(gdb)
```

Move back down the call stack into `calcres` using the `down` command and print the PC again:

```
(gdb) down

#0   calcres (x=-1.47395837, y={1.25, 1.22395837, 1.19791663,
1.171875...},
    res=150) at mandelbrot_poly.cn:27
27     xcalc    = x;
(gdb) print/x $pc
$5 = 0x20052ac
(gdb)
```

As you can see the debugger has the ability to track the registers of the device at different levels of the call stack.

7.3.8 Viewing the poly enable state

It is useful to be able to view the enable state of the PE array when you are debugging code in order to tell what PEs are enabled when the code is executing. The enable state is viewable in the debugger as a pseudo PE register and is viewed in the same way as the other poly registers.

Print the value of the enable register in the debugger:

```
(gdb) p/x $enable
$2 = {0xff <repeats 96 times>}
(gdb)
```

The enable state is an 8 level deep stack and so it is clearer to view the register in binary format to see each level of the current state.

Print the value of the enable state in binary:

```
(gdb) p/t $enable
$3 = {11111111 <repeats 96 times>}
(gdb)
```

This show that all 96 of the PEs are currently enabled at every level.

You now need to move to a location in the code where the enabling and disabling of the enable state becomes more apparent.

Change:

```
poly int terminate(poly float x, poly float y) {
    return (x*x + y*y > 4.0f);
}
```

into:

```
poly int terminate(poly float x, poly float y) {
  poly float result = (x*x +y*y);
  if (result > 4.0f){
    return 1;
  }
  else{
    return 0;
  }
}
```

and recompile `mandelbrot_poly.cn`.

Now start the csgdb and set a breakpoint in `main()` and a temporary breakpoint on the function terminate with the tbreak. Also limit the display to the first 4 PEs :

```
set print elements 4

(gdb) connect
0x02000000 in _MONO_DEBUG_AREA ()
(gdb) b main
Breakpoint 1 at 0x200568c: file mandelbrot_poly.cn, line 72.
(gdb) tbreak terminate
Breakpoint 2 at 0x2005118: file mandelbrot_poly.cn, line 18.
(gdb) set print elements 4
(gdb)
```

Then run till you hit the temporary breakpoint and continue execution:

```
(gdb) run
Starting program: C:\Program
Files\clearspeed\csx600_m512_le\examples\csapi\
mandelbrot1/mandelbrot.csx

Breakpoint 1, main () at mandelbrot_poly.cn:72
72          for (col_count = 0; col_count < NUMCOLS; col_count++) {
(gdb) c
Continuing.
terminate (x={-1.5 <repeats 96 times>}, y=
```

```
      {1.25, 1.22395837, 1.19791663, 1.171875...}) at
mandelbrot_poly.cn:18
18              poly float result = (x*x +y*y);
(gdb)
28      // Evaluate the termination condition
29      poly int terminate(poly float x, poly float y)
30      {
31        poly float result = (x*x + y*y);
32
33        if (result > 4.0f) {
34          return 1;
35        } else {
36          return 0;
37        }
(gdb)
```

As you can see, the function `terminate` uses a poly conditional when checking the value of `result`. This will allow you to view the changes in enable state of the processing elements.

Set breakpoints at lines 20 and 23 to stop on the return statements of the function:

```
(gdb) b 20
Breakpoint 4 at 0x02005174: file mandelbrot.cn, line 20.
(gdb) b 23
Breakpoint 5 at 0x020051FC: file mandelbrot.cn, line 23.
(gdb)
```

You also need to remove the limit on the number of PE values displayed by using the `set print elements` command:

```
(gdb) set print elements 0
(gdb)
```

This will reset the number of PE values displayed so that they are all now visible.

You can print the variable `result` to visually inspect the values contained on each PE.

```
(gdb) print result $1 = {3.8125, 3.74807405, 3.68500423, 3.62329102,
3.56293416, 3.50393343, 3.44628906, 3.3900013, 3.33506942,
3.28149414, 3.22927523, 3.17841244, 3.12890625, 3.08075643,
3.03396273, 2.98852539, 2.94444466, 2.90171981, 2.86035156,
2.82033968, 2.78168392, 2.74438477, 2.70844197, 2.6738553,
2.640625, 2.60875106, 2.57823348, 2.54907227, 2.52126741,
2.49481869, 2.46972656, 2.4459908, 2.42361116, 2.40258789,
2.38292098, 2.36461043, 2.34765625, 2.33205843, 2.31781673,
2.30493164, 2.29340291, 2.2832303, 2.27441406, 2.26695418,
2.26085067, 2.25610352, 2.25271273, 2.25067806, 2.25, 2.25067806,
2.25271273, 2.25610352, 2.26085067, 2.26695418, 2.27441406,
2.2832303, 2.29340267, 2.30493164, 2.31781673, 2.33205843,
2.34765625, 2.36461043, 2.38292098, 2.40258789, 2.42361116,
2.44599056, 2.46972656, 2.49481869, 2.52126718, 2.54907227,
2.57823348, 2.60875106, 2.640625, 2.6738553, 2.70844173,
2.74438477, 2.78168392, 2.82033944, 2.86035156, 2.90171957,
2.94444418, 2.98852539, 3.03396225, 3.08075619, 3.12890625,
3.1784122, 3.22927499, 3.28149414, 3.33506918, 3.39000082,
3.44628906, 3.50393295, 3.56293392, 3.62329102, 3.685004,
3.74807382}
(gdb)
```

From visual inspection it is clear that none of the PE values are greater than 4.0.

If the code contained a mono compare, you would not expect the first part of the `if` statement to be taken. As this is a *poly* comparison then the whole statement is executed on all PEs and the enable state is used to determine which PEs are active when the code is executing.

Use the `continue` command to move the debugger onto line 20:

```
(gdb) continue
Continuing.

Breakpoint 3, terminate () at mandelbrot.cn:20
20 return 1;
(gdb)
```

As you can see you have hit the breakpoint inside the `if` statement on line 20. The value of `result` is not greater than 4.0 on any of the PEs and so in this part of the code you would expect them all to be disabled.

Display the enable state with the `print` command to confirm that all the PEs are currently disabled:

```
(gdb) print/t $enable
$2 = {11111110 <repeats 96 times>}
(gdb)
```

As you can see, the first level of the enable state is disabled for all of the PEs at this point.

Continue the processor again using the `continue` command:

```
(gdb) c
```

```
Continuing.

Breakpoint 4, terminate () at mandelbrot.cn:23
23 return 0;
(gdb)
```

The debugger has now stopped at the second return statement of the `terminate` function. As all of the values of result are less than 4.0 all of the PEs should be enabled for this section of code.

Display the enable state using the `print` command:

```
(gdb) print/t $enable
$3 = {11111111 <repeats 96 times>}
(gdb)
```

All of the PEs are now enabled as the first level of the enable state contains a 1.

7.3.9 Attaching commands to breakpoints

If you continue the processor again using the continue command, you once again hit the breakpoint on line 20 within the function `terminate`. But first you are going to *attach* some commands to the breakpoints on lines 20 and 23 of `terminate`.

Use the `info break` command to list the currently active breakpoints:

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y  0x0200568c in main at
mandelbrot_poly.cn:72
        breakpoint already hit 1 time
3   breakpoint     keep y   0x02005174 in terminate at
mandelbrot_poly.cn:20
        breakpoint already hit 1 time
4   breakpoint     keep y   0x020051fc in terminate at
mandelbrot_poly.cn:23
        breakpoint already hit 1 time
```

The breakpoints you are interested in are numbers 3 and 4 as they are set at the return statements of the `terminate` function.

To attach commands to breakpoints you need to use `commands`. You are going to get `csgdb` to display the enable state every time it stops at breakpoints 3 and 4.

Attach the command to print the enable state to breakpoints 3 and 4:

```
(gdb) commands 3
Type commands for when breakpoint 3 is hit, one per line.
End with a line saying just "end".
>print/t $enable
>end
(gdb) commands 4
Type commands for when breakpoint 4 is hit, one per line.
End with a line saying just "end".
>print/t $enable
>end
(gdb)
```

Listing the breakpoints using the `info break` command will show that these commands have been attached:

```
(gdb) info break
Num Type         Disp Enb Address    What
1   breakpoint   keep y   0x0200568c in main at
mandelbrot_poly.cn:72
        breakpoint already hit 1 time
3   breakpoint     keep y   0x02005174 in terminate at
mandelbrot_poly.cn:20
        breakpoint already hit 1 time
        print/t $enable
4   breakpoint     keep y   0x020051fc in terminate at
mandelbrot_poly.cn:23
        breakpoint already hit 1 time
        print/t $enable
(gdb)
```

As you can see breakpoints 3 and 4 have the command `p/t $enable` attached.

Continue the processor so that it hits breakpoint 3 again using the `continue` command:

```
(gdb) c
Continuing.

Breakpoint 3, terminate () at mandelbrot_poly.cn:20
20   return 1;
$4 = {11111111, 11111111, 11111111, 11111111, 11111111, 11111111,
11111111, 11111111, 11111111, 11111111, 11111110 <repeats 77 times>,
11111111, 11111111, 11111111, 11111111, 11111111, 11111111,
11111111, 11111111, 11111111}
(gdb)
```

As you can see the first 10 PEs (0..9) and the last 9 PEs (87..95) are enabled and the middle 77 (10..86) are all disabled. If you continue again, you would expect to see the reverse of these values at breakpoint 4.

Continue execution using the `continue` command:

```
(gdb) c
Continuing.

Breakpoint 4, terminate () at mandelbrot.cn:23
23    return 0;
$5 = {11111110, 11111110, 11111110, 11111110, 11111110, 11111110,
11111110,
  11111110, 11111110, 11111110, 11111111 <repeats 77 times>,
11111110,
  11111110, 11111110, 11111110, 11111110, 11111110, 11111110,
11111110,
  11111110}
(gdb)
```

As you can see from the display the values of the enable state across the PEs now is the exact opposite of the values of the enable state at breakpoint 3.

You can now delete breakpoints 3 and 4 and return from the function `terminate`.

Delete breakpoints 3 and 4 using the delete command:

```
(gdb) delete 3 4
(gdb)
```

7.3.10 Returning from a function

The debugger can return from a function by using the `finish` command:

```
(gdb) finish
Run till exit from #0  terminate (x=
      {-0.8125, -0.748074055, -0.685004234, -0.623291016, -
0.56293416, -0.503933311, -0.446289063, -0.390001178, -0.335069418,
-0.281494141, -0.229275227, -0.178412557, -0.12890625, -
0.0807564259, -0.0339627266, 0.0114746094, 0.0555554628,
0.0982801914, 0.139648438, 0.17966032, 0.218315959, 0.255615234,
0.291558146, 0.326144695, 0.359375, 0.391248941, 0.42176652,
0.450927734, 0.478732586, 0.505181313, 0.530273438, 0.554009199,
0.576388836, 0.597412109, 0.61707902, 0.635389566, 0.65234375,
0.66794157, 0.682183266, 0.695068359, 0.70659709, 0.716769695,
0.725585938, 0.733045816, 0.739149332, 0.743896484, 0.747287273,
0.749321938, 0.75, 0.749321938, 0.747287273, 0.743896484,
0.739149332, 0.733045816, 0.725585938, 0.716769695, 0.706597328,
0.695068359, 0.682183266, 0.66794157, 0.65234375, 0.635389566,
0.61707902, 0.597412109, 0.576388836, 0.554009438, 0.530273438,
0.505181313, 0.478732705, 0.450927734, 0.42176652, 0.391248941,
0.359375, 0.326144814, 0.291558266, 0.255615234, 0.218316078,
0.179660559, 0.139648438, 0.0982804298, 0.0555557013, 0.0114746094,
-0.033962369, -0.0807561874, -0.12890625, -0.178412199, -
0.229274988, -0.281494141, -0.33506906, -0.390000939,
-0.446289063, -0.503933072, -0.562933803, -0.623291016, -
0.685003996, -0.748073816},
    y=
      {-2.5, -2.44791651, -2.39583349, -2.34375, -2.29166651, -
2.23958349, -2.1875,
-2.13541651, -2.08333349, -2.03125, -1.97916663, -1.92708325, -
1.875, -1.82291663,
-1.77083325, -1.71875, -1.66666663, -1.61458325, -1.5625, -
1.51041663, -1.45833325, -1.40625, -1.35416663, -1.30208325, -1.25,
-1.19791675, -1.14583325,-1.09375,
-1.04166675, -0.989583313, -0.9375, -0.885416746, -0.833333313, -
0.78125,
-0.729166746, -0.677083313, -0.625, -0.572916746, -0.520833373, -
0.46875,
-0.416666746, -0.364583492, -0.3125, -0.260416746, -0.208333492, -
0.15625,
-0.104166746, -0.0520834923, 0, 0.0520832539, 0.104166508, 0.15625,
0.208333254, 0.260416508, 0.3125, 0.364583254, 0.416666508,
0.46875, 0.520833254, 0.572916508, 0.625, 0.677083254, 0.729166508,
0.78125, 0.833333254, 0.885416508, 0.9375, 0.989583
254, 1.04166651, 1.09375, 1.14583325, 1.19791651, 1.25, 1.30208325,
1.35416651,1.40625, 1.45833337, 1.51041651, 1.5625, 1.61458302,
1.66666651, 1.71875, 1.77083302, 1.82291651, 1.875, 1.92708302,
1.97916651, 2.03125, 2.08333302, 2.13541651, 2.1875, 2.23958302,
2.29166651, 2.34375, 2.39583302, 2.44791651})
   at mandelbrot_poly.cn:23
```

```
0x020054b8 in calcres (x=-1.5, y= {1.25, 1.22395837, 1.19791663,
1.171875, 1.14583337, 1.11979163, 1.09375, 1.06770837, 1.04166663,
1.015625, 0.989583373, 0.963541687, 0.9375, 0.911458373,
0.885416687, 0.859375, 0.833333373, 0.807291687, 0.78125,
0.755208373, 0.729166687, 0.703125, 0.677083373, 0.651041687,
0.625, 0.598958373, 0.572916687, 0.546875, 0.520833373,
0.494791687, 0.46875, 0.442708373, 0.416666687, 0.390625,
0.364583373, 0.338541687, 0.3125, 0.286458373, 0.260416687,
0.234375, 0.208333373, 0.182291746, 0.15625, 0.130208373,
0.104166746, 0.078125, 0.0520833731, 0.0260417461, 0, -
0.0260416269, -0.0520832539, -0.078125,
-0.104166627, -0.130208254, -0.15625, -0.182291627, -0.208333254, -
0.234375,
-0.260416627, -0.286458254, -0.3125, -0.338541627, -0.364583254, -
0.390625,
-0.416666627, -0.442708254, -0.46875, -0.494791627, -0.520833254, -
0.546875,
-0.572916627, -0.598958254, -0.625, -0.651041627, -0.677083254, -
0.703125,
-0.729166627, -0.755208254, -0.78125, -0.807291508, -0.833333254, -
0.859375,
-0.885416508, -0.911458254, -0.9375, -0.963541508, -0.989583254, -
1.015625,
-1.04166651, -1.06770825, -1.09375, -1.11979151, -1.14583325, -
1.171875,
-1.19791651, -1.22395825}, res=150)
    at mandelbrot_poly.cn:47
47        if (terminate(xcalc, ycalc)) {
Value returned is $6 =
  {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0 <repeats 77 times>, 1, 1, 1, 1,
1, 1, 1, 1, 1}
(gdb)
```

You are now at line 47 of `calcres` after returning from the function `terminate`. You need to set a breakpoint at the return statement of `calcres`.

```
(gdb) list
42
43   for (i = 0; i < res; i++) {
44      /* Only continue calculating if the result has not been
determined yet */
45         if (turnedon) {
46            /* Check to see if the termination condition is met */
47               if (terminate(xcalc, ycalc)) {
48                  /* Final result is the number of iterations req. to
terminate */
49                  result   = i + 1;
50                  turnedon = 0;
51               }
(gdb)
52               else {
53                  /* Set the values for the next iteration */
54                  tx    = xcalc * xcalc - ycalc * ycalc + x;
55                  ycalc = 2.0f * xcalc * ycalc + y;
56                  xcalc = tx;
57               }
58            }
59         }
60         return result;
61      }
(gdb)
62
63      int main() {
64      mono float x;
65      poly float y;
```

Set a breakpoint at line 60 using the `break` command and continue the processor using the `continue` command:

```
(gdb) break 60
Breakpoint 5 at 0x0200541C: file mandelbrot.cn, line 60.
(gdb) continue
Continuing.

Breakpoint 5, calcres (x=<n/a>, y=<n/a>, res=<n/a>) at
mandelbrot.cn:60
60 return result;
(gdb)
```

Note: *The values of the arguments to `calcres` are now all out of scope and so no longer available to `csgdb`. This is why `x`, `y` and `res` all now have a value of `<n/a>`.*

The result from the `calcres` function is a line of the Mandelbrot program output. The type of the variable `result` can be displayed using the `whatis` command.

Use the `whatis` command on the variable `result`:

```
(gdb) whatis result
```

```
type = poly char <96 PEs>
(gdb)
```

The debugger displays that it is a `poly char` and shows that it is available across 96 processing elements.

Display the value of `result` using the `print` command.

```
(gdb) print result
$3 = "\002\002\002\002\002\002\002\002\002\002", '\003' <repeats 22
times>,
"\004\004\004\004\005\005\005\005\005\005\005\006\a\a\b\v\000\v\b\a
\a\006\
005\005\005\005\005\005\005\004\004\004\004", '\003' <repeats 22
times>, "\002\002\002\002\002\002\002\002\002"
(gdb)
```

As the variable is a `char` type the default output for `csgdb` is to attempt to display it as a `char` value.

Display the value of `result` as an integer using the `print/x` command:

```
(gdb) print/x result
$5 = {0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2,
  0x3 <repeats 22 times>, 0x4, 0x4, 0x4, 0x4, 0x5, 0x5, 0x5, 0x5,
0x5, 0x5,
  0x5, 0x6, 0x7, 0x7, 0x8, 0xb, 0x0, 0xb, 0x8, 0x7, 0x7, 0x6, 0x5,
0x5, 0x5,
  0x5, 0x5, 0x5, 0x5, 0x4, 0x4, 0x4, 0x4, 0x3 <repeats 22 times>,
0x2, 0x2,
  0x2, 0x2, 0x2, 0x2, 0x2, 0x2, 0x2}
(gdb)
```

The function `calcres` is called multiple times from `main` and it is possible to set a condition on the breakpoint at line 88 so that it will not report the breakpoint hit until a certain count is reached. To set a counted breakpoint use the `ignore` command on breakpoint number 5:

```
(gdb) ignore 5 20
```

Will ignore next 20 crossings of breakpoint 5.

You can now use the `info break` command to check what condition the `ignore` command has set up on the breakpoint:

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0200568c in main at
mandelbrot_poly.cn:72
        breakpoint already hit 1 time
5   breakpoint     keep y   0x0200541c in calcres at
mandelbrot_poly.cn:60
        breakpoint already hit 1 time
        ignore next 20 hits
(gdb)
```

Breakpoint number 5 now has the condition set that it will ignore the next 20 hits.

Continue the processor using the `continue` command.

```
(gdb) continue
Continuing.

Breakpoint 5, calcres (x=<n/a>, y=<n/a>, res=<n/a>) at mandelbrot.cn:60
60 return result;
```

```
(gdb)
```

The breakpoint at line 60 has now been hit and using the `info break` command will show that it has actually now been hit 22 times:

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0200568c in main at
mandelbrot_poly.cn:72
        breakpoint already hit 1 time
5   breakpoint     keep y   0x0200541c in calcres at
mandelbrot_poly.cn:60
        breakpoint already hit 22 times
(gdb)
```

Breakpoint 5 has been hit 22 times: once before you set the ignore condition, then ignored 20 times and then finally after the condition has expired.

Move up the call stack back into `main` by typing `up`:

```
(gdb) up
#1  0x02005804 in main () at mandelbrot_poly.cn:76
76              buffer[col_count] = calcres(x, y, RES);
(gdb)

Now print the variable col_count using the print command:

(gdb) print col_count
$1 = 21
(gdb)
```

The loop in `main` is on its 22nd iteration and this matches up with the number of breakpoint hits.

Move back down the call stack using the `down` command:

```
(gdb) down
#0  calcres (x=<n/a>, y=<n/a>, res=<n/a>) at mandelbrot.cn:60
60 return result;
(gdb)
```

Now print the value of `result` using the `print/x` command:

```
(gdb) p/x result
$2 = {0x3 <repeats 18 times>, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4,
0x5, 0x5,
  0x5, 0x5, 0x5, 0x5, 0x6, 0x6, 0x7, 0x8, 0x9, 0xb, 0xf,
  0x0 <repeats 21 times>, 0xf, 0xb, 0x9, 0x8, 0x7, 0x6, 0x6, 0x5,
0x5, 0x5,
  0x5, 0x5, 0x5, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x3 <repeats 17
times>}
(gdb)
```

Comparing this to the value displayed earlier, it is clear that this line of the Mandelbrot set output is different. You can now move the debugger on to a location where you can look at the final Mandelbrot image.

Delete breakpoint number 5 by using the `delete` command:

```
(gdb) delete 5
(gdb)
```

Move up the call stack by using the `up` command:

```
(gdb) up
#1  0x02005804 in main () at mandelbrot.cn:76
76    buffer[col_count] = calcres(x, y, RES);
(gdb)
```

Use the `list` command to list the lines around this location:

```
(gdb) list
71 /* Iterate over the columns evaluating a complete column each
time */
72   for (col_count = 0; col_count < NUMCOLS; col_count++) {
73      x = MINX + col_count   * STEPX;  // Calculate the x value for
this position
74      y = MAXY - get_penum() * STEPY;  // Calculate the y value for
this PE
75      /* Evaluate this column for 64 rows of the image in one step
*/
76      buffer[col_count] = calcres(x, y, RES);
77   }
78   /* Print out all rows */
79   dprint_poly_memory_raw(buffer, NUMCOLS);
80 }
(gdb)
```

Set a breakpoint at line 79 as this is where you can examine the completed Mandelbrot set image in the debugger:

```
(gdb) break 79
Breakpoint 6 at 0x800155f0: file mandelbrot.cn, line 79.
(gdb)
```

Continue the program using the `continue` command (it will take a little while as it computes the remainder of the image):

```
(gdb) continue
Continuing.

Breakpoint 6, main () at mandelbrot.cn:79
79 dprint_poly_memory_raw(buffer, NUMCOLS);
(gdb)
```

You can examine line 22 of the image and check that it matches the values that were displayed when the debugger was stopped earlier just after it calculated line 22.

Use the `print/x` command to display the 22nd element of the `buffer` array:

```
(gdb) print/x buffer[21]
$3 = {0x3 <repeats 18 times>, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4,
0x5, 0x5,
  0x5, 0x5, 0x5, 0x5, 0x6, 0x6, 0x7, 0x8, 0x9, 0xb, 0xf,
  0x0 <repeats 21 times>, 0xf, 0xb, 0x9, 0x8, 0x7, 0x6, 0x6, 0x5,
0x5, 0x5,
```

```
  0x5, 0x5, 0x5, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4, 0x3 <repeats 17
times>}
(gdb)
```

This clearly matches up with the value printed earlier, when you were inside the function `calcres`.

The variable `buffer` is a 96 element poly `char` array. The debugger will tell you this if you use the `whatis` command on `buffer`:

```
(gdb) whatis buffer
type = poly char [96]<96 PEs>
(gdb)
```

Each PE has its own copy of `buffer`; this variable is too large to fit into registers. You can find the memory address of `buffer` by using the `print/x` command and the `&` operator:

```
(gdb) print/x &buffer
$4 = 0x1c
```

The `buffer` array starts at location 0x1C on each processing element.

7.3.11 Viewing memory

Both mono and poly memory can be displayed in `csgdb` and there are commands for viewing each type. The command `x` is used to examine mono memory and the command `pex` is used for poly memory.

List 2 integers at the current PC location in mono memory using the `x` command:

```
(gdb) x/2x $pc
0x800155f0 <main+408>:  0x04000060      0x88280940
(gdb)
```

The argument “2” determines the number of values to print and the “x” specifies that the output is hexadecimal.

The `pex` command has identical type syntax.

List 2 hex integers at address location `0x0` in PE memory using the `pex` command:

```
(gdb) pex/2x 0x0
(PE 0)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(gdb)
```

The main difference is that the `pex` command allows you to see the memory across all the PEs. The default display is to show only PE 0.

Enter the same pex command again but add the argument 0..10 to the end.

```
(gdb) pex/2x 0x0 0..10
(PE 0)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 1)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 2)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 3)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 4)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 5)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 6)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 7)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 8)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 9)   0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(PE 10)  0x0 <__FRAME_BEGIN_POLY__>:    0xdead2222      0x00000000
(gdb)
```

You can now see 2 integer values for each of the first 11 processing elements.

This will now prove useful as you can restrict your view of the variable buffer so you can find the center of the Mandelbrot set.

You need to switch off the height checking of the command output for this section of the code. The debugger will apply a default page size on startup which will prompt you when the output reaches a certain size to print out the rest.

Switch it off by entering the command set height -1:

```
(gdb) set height -1
(gdb)
```

The variable buffer is very large and the amount of data that the debugger has to display equates to 96x96 characters. Try printing buffer for yourself using the print command. This is the whole Mandelbrot image that will be displayed by dprint_poly_memory_raw once the program has continued.

You can be a little more selective in your view of the data by using the pex command.

Use the pex command to display 3 hex integer values at the address of variable buffer on PEs 33..63.

```
(gdb) pex/4x &buffer 33..63
(PE 33)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x04040404
0x06060505      0x080a0b08      0x07070707
(PE 34)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050404
0x06060505      0x090b0c08      0x07070708
(PE 35)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050504
0x06060505      0x0b0d0b08      0x090a0a0a
(PE 36)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07060605      0x101a0908      0x0c0d1213
(PE 37)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07060605      0x0b0a0908      0x11132811
(PE 38)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07070605      0x0c090907      0x00003613
(PE 39)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07070606      0x100a0907      0x00000016
(PE 40)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07070706      0x0d0b0a08      0x00001c11
```

```
(PE 41)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x06050505
0x08080807      0x140f120a      0x0000003a
(PE 42)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0d070605
0x09090808      0x00260d0b      0x00000000
(PE 43)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0b080706
0x0a090a0d      0x00170f0b      0x00000000
(PE 44)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0a080707
0x100d0d0f      0x0000100d      0x00000000
(PE 45)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0a090807
0x1f131b0c      0x00001317      0x00000000
(PE 46)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x100a0a08
0x0000150d      0x00000000      0x00000000
(PE 47)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0e0c0d0b
0x00003c17      0x00000000      0x00000000
(PE 48)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x00000000
0x00000000      0x00000000      0x00000000
(PE 49)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0e0c0d0b
0x00003c17      0x00000000      0x00000000
(PE 50)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x100a0a08
0x0000150d      0x00000000      0x00000000
(PE 51)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0a090807
0x1f131b0c      0x00001317      0x00000000
(PE 52)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0a080707
0x100d0d0f      0x0000100d      0x00000000
(PE 53)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0b080706
0x0a090a0d      0x00170f0b      0x00000000
(PE 54)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x0d070605
0x09090808      0x00260d0b      0x00000000
(PE 55)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x06050505
0x08080807      0x140f120a      0x0000003a
(PE 56)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07070706      0x0d0b0a08      0x00001c11
(PE 57)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07070606      0x100a0907      0x00000016
(PE 58)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07070605      0x0c090907      0x00003613
(PE 59)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07060605      0x0b0a0908      0x11132811
(PE 60)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050505
0x07060605      0x101a0908      0x0c0d1213
(PE 61)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050504
0x06060505      0x0b0d0b08      0x090a0a0a
(PE 62)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x05050404
0x06060505      0x090b0c08      0x07070708
(PE 63)  0x1c <__FRAME_BEGIN_POLY__+28>:        0x04040404
0x06060505      0x080a0b08      0x07070707
(gdb)
```

The start of the Mandelbrot image can be seen on PE 48 with the value 0x00000000 in the first column.

If you press enter again, you will automatically get the next 4-integer values of the image without having to re-enter the `pex` command.

```
(gdb)
(PE 33)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x080d0807
0x07070707      0x08070707      0x0c0a0808
(PE 34)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x0a100908
0x07070808      0x08080707      0x1f0f0908
(PE 35)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x0b100b09
0x08090909      0x09080808      0x002c0a09
(PE 36)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x210f0c0b
0x0a1f0b0e      0x09090909      0x150f0b0a
(PE 37)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x29130e0e
0x0f160f19      0x0a0a0a0b      0x00380d0a
(PE 38)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x001b0014
0x19000022      0x0b0b0c19      0x00170d0c
(PE 39)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x3c000000      0x0c0e1100      0x00000f0d
(PE 40)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x0e132600      0x0000000e
(PE 41)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x27000000      0x00002610
(PE 42)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x27000000      0x00001813
(PE 43)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000018
(PE 44)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00003d00
(PE 45)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000000
(PE 46)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000000
(PE 47)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000000
(PE 48)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000000
(PE 49)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000000
(PE 50)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000000
(PE 51)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000000
(PE 52)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00003d00
(PE 53)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x00000000      0x00000018
(PE 54)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x27000000      0x00001813
(PE 55)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x27000000      0x00002610
(PE 56)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x00000000      0x0e132600      0x0000000e
```

```
(PE 57)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x00000000
0x3c000000      0x0c0e1100      0x00000f0d
(PE 58)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x001b0014
0x19000022      0x0b0b0c19      0x00170d0c
(PE 59)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x29130e0e
0x0f160f19      0x0a0a0a0b      0x00380d0a
(PE 60)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x210f0c0b
0x0a1f0b0e      0x09090909      0x150f0b0a
(PE 61)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x0b100b09
0x08090909      0x09080808      0x002c0a09
(PE 62)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x0a100908
0x07070808      0x08080707      0x1f0f0908
(PE 63)  0x2c <__FRAME_BEGIN_POLY__+44>:        0x080d0807
0x07070707      0x08070707      0x0c0a0808
(gdb)
```

You can see the number of 0x00000000 values getting larger as the Mandelbrot image spreads out.

All that is left is to continue the application until it exits and returns from main.

7.3.12 Terminating the program

Enter the `continue` command at the prompt.

```
(gdb) c
Continuing.
..... mandelbrot image from dprint_mono_memory_raw() will appear in
terminal .....

Processor 0 has terminated.
Program exited normally.
(gdb)
```

The debugger has reported that the program has exited normally and that the processor has terminated. You can now quit `csgdb` by using the `quit` command and entering `y` when prompted.

You will now be back at the command line prompt as the debugger has exited.

8 Programming host applications

This section provides a brief introduction to the ClearSpeed Application Programming Interface (CSAPI) used to communicate between a program running on the host and the code on the CSX processor. For reasons of brevity, an outline of the method is presented here with some example code fragments. For more detailed information, see the appropriate chapter of the *Runtime Software User Guide*.

The API being described is designed to assist end-users in the creation of host applications using multithreaded array processors for executing speed-critical code. The programming model is best illustrated by the chart in *Figure 10*.

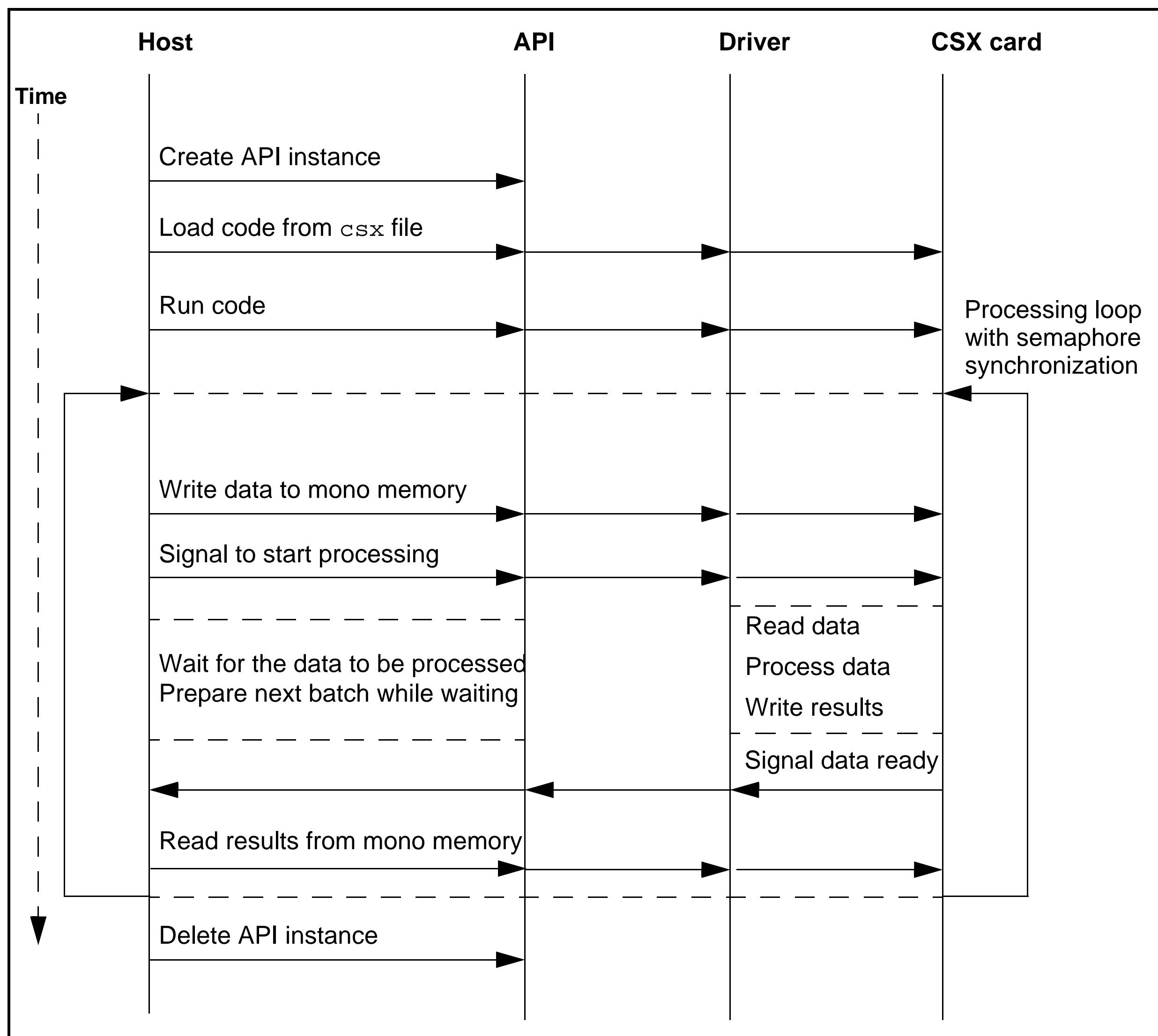

Figure 10. Timeline for driver interactions

There are a number of tasks to be performed when a program is split between the host and the array processor:

1. Initialization
2. Synchronization
3. Data transfer

8.1 Initialization

The first step is to initialize the state of the API interface. The host program can then connect to the driver and boot the program. The functions involved here are:

`CSAPI_new`
: Creates a new instance of the API state.

`CSAPI_connect`
: Connects to the driver on a specified host name or IP address.

`CSAPI_load`
: Loads the executable code from the specified object file onto the processor.

`CSAPI_run`
: Runs the code loaded onto the processor.

Once the driver and the processor have been initialized, the host program can start communicating with code on the array processor.

8.2 Synchronization and communication

`CSAPI_semaphore_signal`

`CSAPI_semaphore_wait`
: These two functions use semaphores to synchronize with the program on the ClearSpeed processor.

`CSAPI_get_symbol_value`
: This function is used to get the value of a symbol. This will typically be the address of a variable or of a label in the code.

`CSAPI_write_mono_memory`

`CSAPI_read_mono_memory`
: These functions are used to transfer data between the host and the memory of the array processor.

8.3 Sample code

Here is a simple example of a host program. It shows how to create a CSAPI state object and connect it to a card. It will then reset processor zero, load and run a program and wait for the program to terminate. Finally it will obtain the exit code.

```
void main()
{
   CSAPIErrno ret;
   struct CSAPIState *s;
   struct CSAPIProcess *process;
   int csx_exit_code;

   s = CSAPI_new();

   ret = CSAPI_connect( s, CSH_Private, CSC_Direct, “localhost”,
CSAPI_INSTANCE_ANY, 0 );

   ret = CSAPI_reset( s, 0, CSR_FullReset, CSAPI_NO_TIMEOUT );

   ret = CSAPI_load( s, 0, CSX_FILE_NAME, NULL, &process,
CSAPI_NO_TIMEOUT );

   ret = CSAPI_run( s, process, NULL );

   ret = CSAPI_wait_on_terminate( s, process, CSAPI_NO_TIMEOUT );

   ret = CSAPI_get_return_value( s, process, &csx_exit_code );

   CSAPI_delete( s );
}
```

The return code from each CSAPI call is not checked in this simple example. For more complete examples refer to the source code provided as part of the SDK installation.

9 Programming hints

This chapter contains a few tips for avoiding the common problems encountered during initial experience programming in **C**^{n}. These are things that may cause code to be less efficient than necessary or may cause unexpected behavior.

For more information on efficiently programming the CSX processors see the training courses on the ClearSpeed Develope web site:

http://developer.clearspeed.com/resources/training/

9.1 Poly flow control

Code with a poly expression for the condition (poly `if`, for example) uses *predicated* instructions that are executed based on the value of the conditional expression. This has a number of implications, some of which are covered in more detail below. However, it is important to note that this is not a branch, and so there is no 'jump' overhead. It is efficient if few additional or duplicate instructions are executed. As far as possible, remove common sub-expressions from poly `if` blocks which will reduce the amount of replicated work. In particular avoid mono computation within poly conditionals.

Be prepared to compute and ignore results if this leads to fewer poly conditional blocks. This can increase efficiency as it means that PEs are enabled and processing more of the time.

9.1.1 Mono code in poly conditionals

This is also discussed in some detail in *Section 5.5.1: If statements*.

The behavior in poly conditionals is not always intuitive and can cause unexpected results. Even if all the values of a poly expression in an `if` statement are the same (so all PEs execute the same branch) the compiler will still emit code to execute both branches and the processor will execute it. The PEs will be disabled for one branch but the mono execution unit will execute all mono instructions.

Consider the following piece of code:

```
poly int value = 0;
mono int i, j;
if (value == 0) {
    i = 1;
}
else {
    j = 2;
}
```

In this case, even though all of the PEs fulfil the condition (and hence none will execute the `else` clause), both the mono variables, `i` and `j`, are updated.

The same considerations apply to any conditional code, including loops and the operators `?:`, `&&` and `||`.

9.1.2 Mixing mono and poly conditions

There are also cases of conditional execution where the conditional behavior is not explicit. One example is the use of boolean operators, typically to combine conditional expressions.

The `&&` and the `||` operators are guaranteed to do *lazy evaluation*. They only evaluate the right hand subexpression if it is necessary to determine the truth or falsehood of the complete expression. This means that the evaluation of the second subexpression is conditional, with implications when the expression mixes mono and poly subexpressions.

In the following code fragment, the evaluation of the second part of the expression is conditional on the result of the first:

```
mono int m;
poly int p;
...
if ((m == 0) && (p == 0))
```

Here, if `m` is not zero, then it is known that the whole expression is false and the value of `p` is never tested. This is as expected, but now consider what happens when the conditional expression is rewritten as:

```
if ((p == 0) && (m == 0))
```

In this case, the value of `p` is tested first. However, if this is false it simply disables the execution of further *poly* code. The processor will continue executing mono code and, specifically, it will evaluate the second expression to determine the value of `m`. This means that the second expression is always evaluated and any side-effects of its evaluation will always occur.

Note: *Despite this, the overall condition is still a poly expression and so the behavior of the conditional statement is as described above.*

9.1.3 Effects of function return type

Care must be taken when using `return` inside a function. The behavior will change depending on the return type. *Section 5.6: Functions* provides more detail.

Essentially, a poly return type means that the function will not actually return until the end of the function, and all mono operations will be carried out.

A mono return type means that the function will return as soon as the return statement is encountered.

9.2 Dereferencing mono*poly pointers

A mono value can, in general, be assigned to a poly variable. One case that can cause confusion is dereferencing (implicitly or explicitly) a `mono*poly` pointer (see *Section 5.3.3: Pointers to mono data*).

In this case, each PE's instance of the pointer can point to a different location in mono memory. This means that dereferencing the pointer would require a complex operation where a different memory location is copied to each PE--this can be done, but requires the use of library functions such as `memcpym2p()`.

As an example, consider the code below:

```
mono int array[96];
```

```
mono int * poly p;
poly int x, n;
int i;

for (i = 0; i < 96; i++) {
   array[i] = i;    // initialise array contents
   x = i;           // a legal mono to poly assignment
}

n = get_penum();     // different value on each PE
p = array + n;       // calculate a different address on each PE
x = *p;              // illegal explicit dereference

x = array[n];        // illegal (implicit) dereference
```

This is illegal and will be reported as an error by the compiler.

9.3 Debugging “random” errors

Try running the code on the simulator. If, for instance, you access an address past the end of the 6 KB poly memory, the hardware will not report an error and eventually wraps the address (masking off irrelevant bits so 8 KB will become 0 KB). The simulator will tell you if you fall off the end of the memory map.

9.4 Optimizing code

For further information please refer to the *CSX600 Hardware Programming Manual*.

9.4.1 Poly computation

Remember that the poly ALU is 8 bits wide. Therefore, a 4-byte operation will take roughly twice as long as a 2-byte operation. Consider your code: do you need a 32-bit `int` when a 16-bit `short` will do? Good examples are array subscripts and pointers in the 6 KB PE memory (a signed short will cover 32 KB). The added benefit is that `short` variables take up less poly memory space which is a scarce resource.

Avoid mixing mono and poly values in an expressions. It is usually more efficient to copy a mono constant to a poly variable before operating on it.

Using vector data types

Use the vector intrinsics for maximum floating point performance but do not automatically assume you need to vectorize. This should only be done if you have a small set of working variables and you are not mono to poly bandwidth bound. Vectorization is not always effective in the case of sets of expressions with lots of variables as this can cause the compiler to run out of registers and hence cause more reads and writes to memory.

The Vector Math Library (VML)

VML functions are much faster than the equivalents in the standard math library. However, they do use more PE memory and so should be used sparingly.

9.4.2 Efficient use of embedded SRAM

The embedded SRAM (ESRAM) on CSX processors provides fast access to code. However, this is limited in size (128 KB on the CSX600) and so care needs to be taken in order to make the best use of it.

If your code is small enough to fit in the ESRAM then the linker will place it there by default.

However, if the code is larger than the available ESRAM then the code will *all* be placed in external DRAM. This may impact the performance of your application. This is partly due to the fact that DRAM is inherently slower than the ESRAM. In addition, frequent instruction fetches from DRAM may cause conflicts with other memory accesses (for example, host or PIO transfers) which will degrade performance further.

You can use the `hot` pragma in your Cn source code to identify functions which should be placed in ESRAM. All the functions marked as 'hot' will be placed in ESRAM first. The rest of the code will be also be put in ESRAM if it fits, otherwise it will be put in DRAM. If the code for the functions marked as hot is larger than the size of the ESRAM then an error will be reported when the code is loaded.

See the Cn language chapter of the *SDK Reference Manual* and the host interface library (CSAPI) chapter of the *Runtime Software User Guide* for more details.

9.4.3 Maximizing DRAM performance

There are a number of issues to consider to get the best performance out of the DRAM connected to the CSX processor.

- Sizes of data transfers
- Alignments of data transfers
- Concurrent or interleaved access to DRAM

In summary, it is best to make data accesses of at least 32 bytes, aligned to a 32-byte boundary. When performing PIO transfers between PE memory and DRAM, then a minimum transfer size of 64 bytes is most efficient.

Data transfer sizes

DRAM with error correction (ECC) always reads and writes words of 8 bytes. Therefore transfers should be a multiple of 8 bytes. DRAM has a *burst length* of 4 so the most efficient transfers are 4 x bytes: accessing 1 byte takes as long as accessing 32 bytes.

Data alignment

PIO accesses must be aligned to 8 byte boundaries. DMA transfers are most efficient when aligned to 32 bytes.

data alignment is supported in Cn using `#pragma align`. Assembly code can use the `.align` directive.

Multiple DRAM streams

The underlying issue here is switching between banks of DRAM. The memory can access locations within a single bank very quickly. There is an overhead for switching between banks. Accesses from a single source tend to be coherent and make multiple access to the same bank. Interleaving accesses from different sources (for example instruction fetch, PIO

and host accesses) can cause performance to drop as the memory switches between banks.

Similarly, changing between read and write also incurs a small penalty. Therefore it is more efficient to do as many accesses in the same direction as possible.

9.4.4 Swazzle path

Remember the swazzle path has a 160 Gbyte/s aggregate bandwidth available.

The swazzle path is 8 bytes wide. Swazzling 1 byte takes as long as 8 bytes. It is best to use a multiple of 8 bytes.

Do not use the `swazzle_up_zero` function. It is more efficient to use the `set_swazzle_ends` function followed by `swazzle_up`. You do not need to call `set_swazzle_ends` multiple times if, for example, you always want 0 in PE number 0.

9.4.5 Memory copying functions

Avoid using the `memcpym2p` and `memcpyp2m` functions. Instead, use the `async_` versions, as these are more efficient and allow data transfer to be overlapped with computation.

Remember that `async_memcpy` requires 4 byte alignment on data in poly memory. Use `#pragma align 4`, or use an integer variable, as this will always be 4 byte aligned.

9.4.6 Should you double-buffer mono to poly access?

Don’t automatically assume that you should double-buffer mono to poly transfers. Only do this if you are memory bandwidth bound.

Consider the 6 KB PE memory, if you use double buffering, you may consume significant memory resources. If you don't double buffer, you will get better PE memory use so you may actually be more efficient by being careful with memory and single buffering.

Bibliography

1. SDK Reference Manual
   Document Number: 06-UG-1136
   ClearSpeed Technology, 2004
2. The **C^n** Standard Libraries Reference Manual
   Document Number: 06-RM-1139
   ClearSpeed Technology, 2004
3. CSX Processor Architecture
   White Paper 02-WP-1110
   ClearSpeed Technology
4. Instruction Set Reference Manual
   Document Number: 06-RM-1137
   ClearSpeed Technology
5. Runtime Software User Guide
   Document Number: 06-UG-1345
   ClearSpeed Technology
6. CSX600 Hardware Programming Manual
   Document Number: 06-RM-1305
   ClearSpeed Technology
7. CSX600 Architecture Manual
   Document Number: 06-RM-1304
   ClearSpeed Technology
8. The C Programming Language
   Brian W. Kernighan & Dennis M. Ritchie
   ISBN: 0131103628
   Prentice Hall, 1988
9. ISO/IEC 9899: Programming Languages – C
   Reference number: ISO/IEC 9899 : 1990 (E)
   ISO/IEC Copyright Office
   Case Postale 56
   CH-1211 Genève 20
   Switzerland
10. The Mandelbrot Set Explorer
    http://math.bu.edu/DYSYS/explorer/index.html

Revision history

Date	Revision	Changes
Jul 2007	2.4	Initial version into the corporate standard. Change of title to reflect document contents.
Jan 2008	2.E	New corporate standard and mulitple amendments to contents.

Table 1. Document revision history

ClearSpeed Technology Ltd
130 Aztec West
Park Avenue
Bristol BS32 4UB
United Kingdom

Tel: +44 (0)1454 629 623
Fax: +44 (0)1454 629 624

Email: info@clearspeed.com

Web: http://www.clearspeed.com

Support: http://support.clearspeed.com

1. Information and data contained in this document, together with the information contained in any and all associated ClearSpeed documents including without limitation, data sheets, application notes and the like ('Information') is provided in connection with ClearSpeed products and is provided for information only. Quoted figures in the Information, which may be performance, size, cost, power and the like are estimates based upon analysis and simulations of current designs and are liable to change.
2. Such Information does not constitute an offer of, or an invitation by or on behalf of ClearSpeed, or any ClearSpeed affiliate to supply any product or provide any service to any party having access to this Information. Except as provided in ClearSpeed Terms and Conditions of Sale for ClearSpeed products, ClearSpeed assumes no liability whatsoever.
3. ClearSpeed products are not intended for use, whether directly or indirectly, in any medical, life saving and/ or life sustaining systems or applications.
4. The worldwide intellectual property rights in the Information and data contained therein is owned by ClearSpeed. No license whether express or implied either by estoppel or otherwise to any intellectual property rights is granted by this document or otherwise. You may not download, copy, adapt or distribute this Information except with the consent in writing of ClearSpeed.
5. The system vendor remains solely responsible for any and all design, functionality and terms of sale of any product which incorporates a ClearSpeed product including without limitation, product liability, intellectual property infringement, warranty including conformance to specification and or performance.
6. Any condition, warranty or other term which might but for this paragraph have effect between ClearSpeed and you or which would otherwise be implied into or incorporated into the Information (including without limitation, the implied terms of satisfactory quality, merchantability or fitness for purpose), whether by statute, common law or otherwise are hereby excluded.
7. ClearSpeed reserves the right to make changes to the Information or the data contained therein at any time without notice.

© Copyright ClearSpeed Technology Ltd 2010. All rights reserved.

Advance is a registered trademark of ClearSpeed Technology Ltd

ClearSpeed, ClearConnect, Advance and the ClearSpeed logo are trade marks or registered trade marks of ClearSpeed Technology Ltd. All other brands and names are the property of their respective owners.