

*Hardware Compilation and  
Resource Sharing Optimisations*

BEng Information Systems Engineering

**Final Year Project Report**

Summer 2001

Dong-U Lee

Imperial College

## Contents

Abstract .....	3
Acknowledgements .....	4
<i>Chapter 1: Introduction</i> .....	5
<i>Chapter 2: Background</i>	
2.1 Introduction .....	7
2.2 Field Programmable Gate Arrays (FPGAs).....	7
2.3 Types of FPGAs .....	8
2.4 Overview of the SONIC Board .....	9
2.5 Plug-In Processing Elements .....	11
2.6 RC1000.....	12
2.7 Handel-C .....	12
2.8 Electronic Design Interchange Format (EDIF).....	14
2.9 The BMP file format .....	14
2.10 Summary .....	15
<i>Chapter 3: Compile-time Support</i>	
3.1 Introduction .....	16
3.2 Motivations of using Handel-C .....	16
3.3 Extending the Handel-C language .....	16
3.4 Requirements of mapping Handel-C on SONIC .....	18
3.5 Overview of mapping Handel-C on SONIC.....	18
3.6 Memory Interfacing.....	21
3.7 Interfacing with the PIPE Bus .....	25
3.8 Interfacing with the PIPE Flow Bus .....	26
3.9 Summary .....	27
<i>Chapter 4: Run-time Support</i>	
4.1 Introduction .....	27
4.2 C++ application for Run-Time support .....	28
4.3 Example using the Memory interface and PIPE Bus.....	30
4.4 Example using PIPEflow Bus .....	32
4.5 Template for Handel-C programs .....	33
4.6 Summary .....	34
<i>Chapter 5: Partitioning in Space and Time</i>	
5.1 Introduction .....	35
5.2 Partitioning in Space .....	36
5.3 Partitioning in Time.....	37
5.4 Summary .....	41
<i>Chapter 6: Extending the Handel-C language</i>	
6.1 Introduction .....	42
6.2 Hardware Partitioning .....	42
6.3 Software Descriptions in EHC .....	46
6.4 Making EHC portable .....	49
6.5 Summary .....	52

<i>Chapter 7: Run-Time Reconfiguration</i>	
7.1	Introduction ..... 53
7.2	Example of reconfiguration: Edge Detector..... 53
7.3	Pipeline Morphing..... 57
7.4	Summary ..... 59
 <i>Chapter 8: Operator Sharing</i>	
8.1	Introduction ..... 60
8.2	Scope and Restrictions of Sharing in Handel-C ..... 61
8.3	Impact of Sharing on Speed and Area ..... 63
8.4	Inner Product Multiplication ..... 65
8.5	Elliptic Wave Filter ..... 68
8.7	Automatic Sharing..... 69
8.8	Summary ..... 72
 <i>Chapter 9: Conclusion</i> ..... 73	
References..... 76	
Appendix A ..... 77	
Appendix B..... 78	
Appendix C..... 82	
Appendix D ..... 84	
Appendix E..... 91	

## **Abstract**

The purpose of this project is to provide a high-level compilation and optimisation framework for reconfigurable engines, and to investigate the effect of sharing resources on FPGAs. The framework enables users to program reconfigurable engines with a high-level language, similar to C. It provides support for application-level optimisations such as space and time partitioning, and facilities for improving design abstraction and portability and for exploiting run-time reconfiguration of FPGAs. Device-level optimisations, such as operator sharing, are also supported. The SONIC system, a reconfigurable engine developed to accelerate imaging and graphics applications, is used as a vehicle to demonstrate our approach. The effectiveness of our approach is demonstrated by various applications.

## **Acknowledgements**

I would like to thank my supervisor Dr Luk for his interest and support, and of course Shay who has been helping me throughout the year.

## Chapter 1. Introduction

Traditionally, we either implemented computations in hardware (e.g. custom VLSI, Application-Specific Integrated Circuits, gate-arrays) or we implemented them in software running on processors (e.g. Digital signal Processors, microcontrollers, embedded or general-purpose microprocessors). More recently, however, Field-Programmable Gate Arrays (FPGAs) introduce a new alternative, which mixes and matches properties of the traditional hardware and software alternatives. Machines based on these FPGAs have achieved impressive performance [21],[31] — often achieving 100 times the performance of processor alternatives and 10-100 times the performance per unit of silicon area. The latest FPGAs have millions of reconfigurable gates, capable of running at hundreds of MHz. With falling prices, these devices are well-suited for graphics, video and image processing.

Using FPGAs for computing results in computer organisations is known as *reconfigurable computing* architectures. The key characteristics distinguishing these machines are that they can be customised to solve *any* problem after device fabrication, and they exploit a large degree of spatially customised calculations in order to perform the computations in parallel. Examples of a system containing FPGAs is the SONIC system and the RC1000 system. Such systems are often known as reconfigurable engines, because they support both software and hardware elements, both of which are programmable: generally, the software elements are implemented by microprocessors, and the hardware parts are implemented by FPGAs. Figure 1.1 shows the architecture of a typical reconfigurable engine.

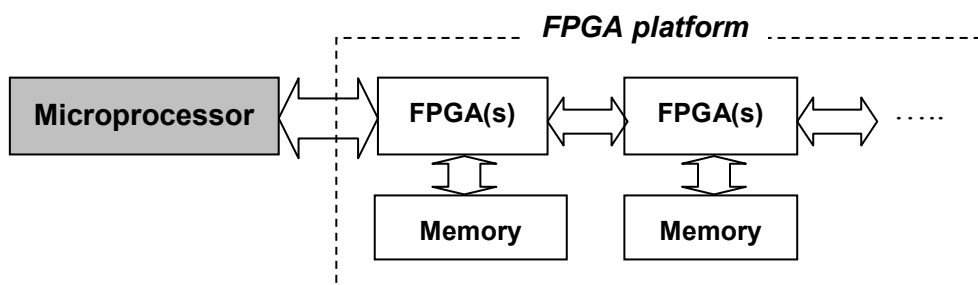


Figure 1.1 – A typical reconfigurable engine

This project is concerned with two major topics: hardware compilation and resource sharing optimizations.

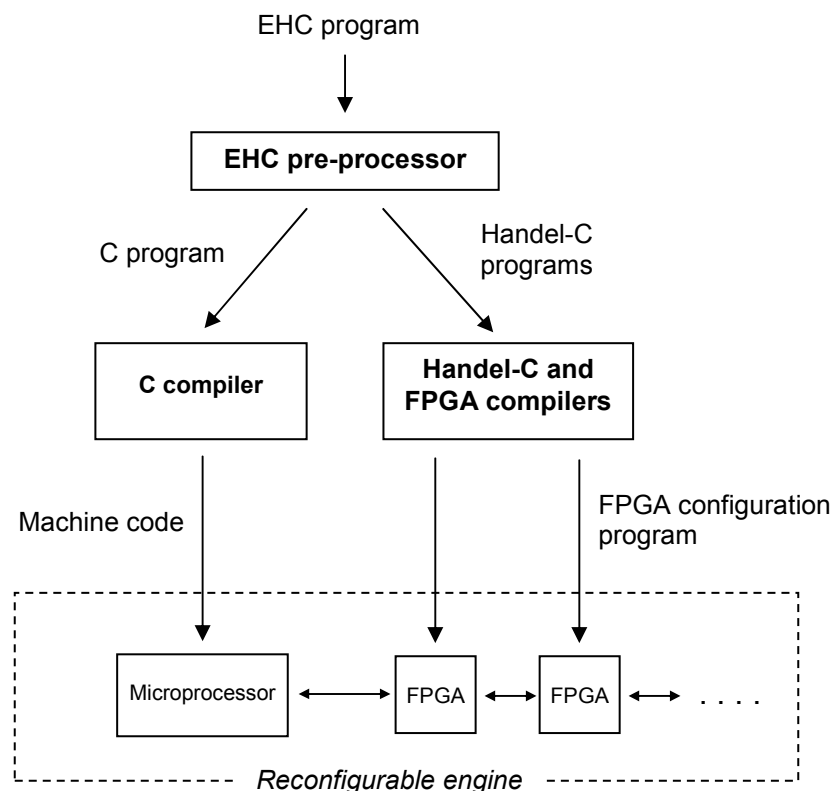
The hardware compilation part concentrates on the mapping of the Handel-C language onto the SONIC system. Handel-C is a high-level language that allows user control of sequential and parallel processing, and also operator size and widths. Compile time support is provided by designing various interfaces. These interfaces enable the Handel-C program to communicate with the different parts of the SONIC board, such as the

onboard memory. Run-time support is provided via a C++ application using the SONIC API [4]. This run-time support application allows the user to configure the SONIC board and to monitor the results it produces. Some simple examples such as image colour inversion are demonstrated to show the functionality of the interfaces.

The Handel-C language has been extended to include both hardware and software descriptions in a single framework, as shown in figure 1.2. The resulting language is called EHC (Extended Handel-C). EHC is used to investigate resource optimizations, which consist of three topics: space and time partitioning, run-time reconfiguration and operator sharing. EHC is designed to be portable between different platforms; it will be demonstrated that the same EHC code can be run on both SONIC and RC1000.

In space and time partitioning, we will see how the SONIC system can be used to exploit spatial and temporal parallelism. Run-time reconfiguration on SONIC is demonstrated with a simple example, where a PIPE is reconfigured to perform two tasks on an image.

Operator sharing involves the use of shared expressions in Handel-C. The effect of sharing multipliers on speed and area of FPGAs is discussed, and a simple model of implementing automatic sharing for Handel-C programs is suggested.



**Figure 1.2 - Mapping EHC onto reconfigurable engine. The local memory associated with each FPGA is not shown.**

## Chapter 2. Background

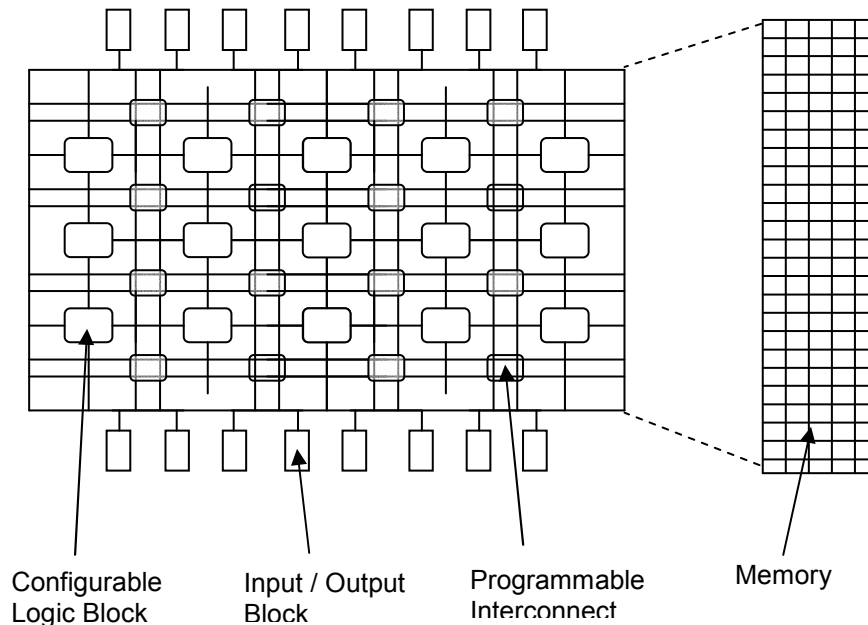
### 2.1 Introduction

This chapter introduces several basic but important concepts that are necessary to understand the content of this report. Emphasis is given on the SONIC architecture – a good understanding of SONIC is assumed throughout this report.

### 2.2 Field Programmable Gate Arrays (FPGAs)

Unlike the traditional fully customised VLSI circuits, Field Programmable Gate Arrays (FPGAs) represent a technical breakthrough in the corresponding industry. Before they were introduced, an electronic designer had only a few options for implementing digital logic. These options included discrete logic devices (VLSI or SSI); programmable devices (PALs or PLDs); and Masked Programmed Gate Arrays (MPGA) or Cell-Based ASICs. A discrete device can be used to implement a small amount of logic. A programmable device is a general-purpose device capable of implementing the logic of tens or hundreds of discrete devices. It is programmed by users at their site using programming hardware. The size of a PLD is limited by the power consumption and time delay. In order to implement designs with thousands or tens of thousands of gates on a single IC, MPGA can be used. An MPGA consists of a base of pre-designed transistors with customised wiring for each design. The wiring is built during the manufacturing process, so each design requires custom masks for the wiring. The cost of mask-making is expensive and the turnaround time is long (typically four to six weeks). The availability of FPGAs offer the benefits of both PLD and MPGA. FPGAs can implement thousands of gates of logic in a single IC and it can be programmed by users at their site in a few seconds or less depending on the type device used. The risk is low and the development time is short. These advantages have made FPGAs very popular for prototype development, custom computing, digital signal processing, and logic emulation. It should be noted that FPGAs improve faster than Moore's Law. For example, the Xilinx XC4085 FPGA contained 85K gates in 1996, now the new Xilinx XCV2000 FPGA contains 3 million gates; double capacity every year rather than 18 months.

## 2.3 Types of FPGAs



**Figure 2.1 – FPGA structure**

*Complex Programmable Logic Devices:* CPLDs are also known as Erasable Programmable Logic Devices (EPLDs). They consist of up to ten thousand macrocells interconnecting each other.

*Static RAM Field Programmable Gate Array:* In contrast to EPLDs, SRAM FPGAs have a different architecture. It consists of a large matrix of logic elements (LE) embedded in a configurable interconnection structure and surrounded by configurable I/O blocks. For instance, in the Altera's Flexible Logic Element Matrix (FLEX) 8000 family, each LE contains 4-input Look up tables (LUTs), a programmable flip-flop, a carry chain, and a cascade chain. The LUTs implement combinational and sequential logic functions and a user-programmable routing network, which provides connections among the LUTs. As the system powers up, the configuration data stored in EPROM, RAM or an external intelligent device will download to the dedicated device. The process time is less than 100 ms, real-time device configuration changes can be achieved during system operation.

## 2.4 Overview of the SONIC Board

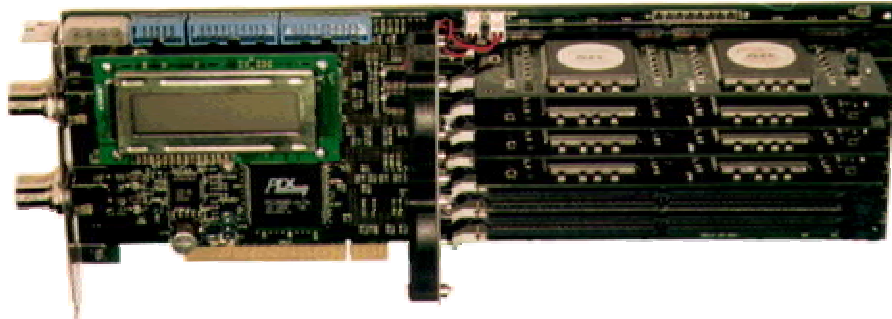


Figure 2.2 - The SONIC board

SONIC is a configurable computing system developed jointly by Imperial College and Sony. It was specifically designed to process (video) images using reconfigurable hardware. Focusing the application domain in this way also means that greater acceleration can be achieved than would be the case for a more general architecture. This also simplifies the Application Programmer's Interface (API). SONIC is a PCI board and has a Serial Digital Interface (SDI). This interface is widely used throughout the professional broadcasting industry. This allows for pipelined processing of video, using the SDI interface for I/O and using the PCI bus for control.

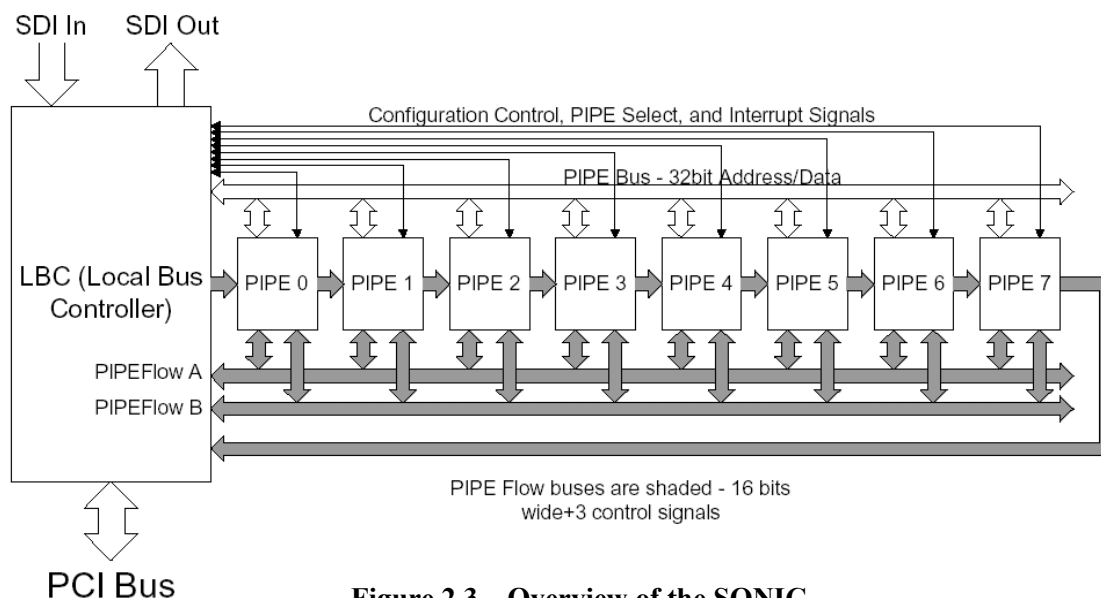


Figure 2.3 – Overview of the SONIC

Figure 2.2 gives an overview of the SONIC Board. The board consists of the following parts:

### *Plug In Processing Elements (PIPEs)*

There are 8 PIPEs on the SONIC board, which plug into the 200 pin DIMM sockets. The PIPEs process the images.

### *PCI Bus*

The PCI bus connects SONIC to the host PC. Any data going to or from the host PC will use the PCI bus.

### *Serial Digital Interface (SDI)*

The SONIC board also has a Serial Digital Interface (SDI) input and output, so that image data can be transferred to/from the SONIC board. The format of the SDI data is very similar to the PIPEFlow data stream that SONIC uses.

### *Local Bus Controller (LBC)*

This is responsible for interfacing the SONIC board to the external world. The LBC performs three tasks:

- Download Designs - The LBC will download the design to the appropriate PIPE to configure it.
- Image Transfer - All image transfer is handled by the LBC. The images can travel via the PCI bus or to/from the SDI.
- PIPEFlow Bus Routing - The LBC looks after all the PIPE routing.

### *PIPE Bus*

The PIPE bus is a global 32bit Multiplexed Address + 4 Control Signals Data bus. It is designed for:

- Fast Image Transfer - It will allow bursting to/from the PCI bus at the maximum burst rate (133Mbytes/sec) from/to the PIPE Memory.
- PIPE Register Access - Any run-time information required by the PIPE can be transferred using the PIPE bus. PIPE Status information can also be read.
- PR Switching - Controlling where the PR routes the PIPEFlow data.

### *PIPEFlow Buses*

The PIPEFlow buses are 16 bits in width with 3 single bit control signals. They are designed for pipelined operation of the SONIC board. Data is sent on these buses using a simple 'raster-scan' protocol. The PIPEFlow data can be routed from a number of sources, or automatically generated from an image stored in the memory on a PIPE. PIPEFlow buses A and B can be used to transfer images from any PIPE (or LBC) to any number of other PIPEs (or LBC) (for example, PIPE 0 could send image data to PIPEs 1,2,3 and 5 using the PIPEFlow A bus)

### *PIPE Control Signals*

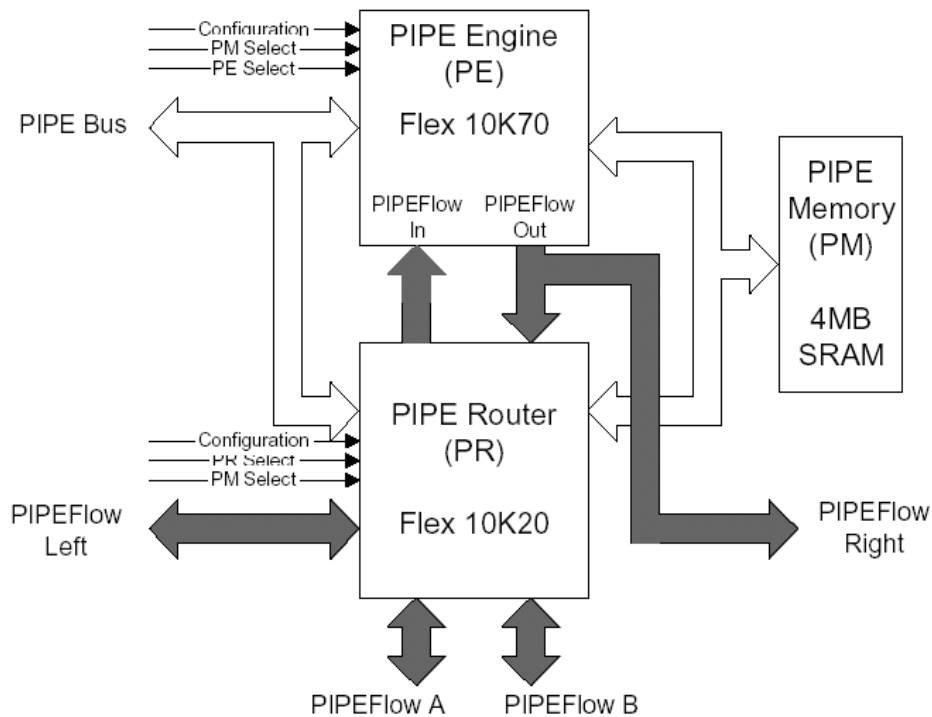
Each PIPE has a number of unique control signals coming to/from it. These are:

PIPE Select Signals - Used to indicate which PIPE and device on the PIPE the PIPE Bus is addressing.

Configuration - Used by the LBC to configure the PIPEs.

## 2.5 Plug-In Processing Elements (PIPEs)

The PIPEs are the core elements of the SONIC architecture, they perform the actual processing of the images. The PIPE Engine (PE) of the PIPEs are configured by an rbf file. Figure 2 shows an overview of the PIPE.



**Figure 2.4 – Overview of PIPE**

### *The PIPE Engine (PE)*

The PE is responsible for the processing of the image. On the current PIPEs, a FLEX10K70 is used to implement the PE. It is for this device (and only this device) that the designer must design hardware.

### *The PIPE Router (PR)*

The PR is implemented on a FLEX10K20 device, the configuration of which is pre-designed to implement all the required functions of the PR. The PR routes the PIPEFlow In & Out Buses to the PE. The PR can get the data for the PIPEFlow In bus from a number of places:

- PM - The PR has the capability of automatically generating the PIPEFlow data from an image held in the PM. This can only be done if the PM is not used by the PE.
- PIPEFlow Left - PIPEFlow data comes from the previous PIPE.
- PIPEFlow A - PIPEFlow data comes from the LBC or any other PIPE.
- PIPEFlow B - PIPEFlow data comes from the LBC or any other PIPE

The PR can also route the data on the PIPEFlow Out bus to a number of places:

- PM - The PR has the capability of automatically storing the PIPEFlow data as an image held in the PM. This can only be done if the PM is not used by the PE.
- PIPEFlow Right - The PIPEFlow out data is always routed to the next PIPE (it is actually hardwired this way), by the PIPEFlow Right bus.
- PIPEFlow A - The PIPEFlow out data is routed to bus 'A' for access by the LBC or any other PIPE. Only one PIPE at a time can drive the PIPEFlow A bus.
- PIPEFlow B - The PIPEFlow out data is routed to bus 'B' for access by the LBC or any other PIPE. Only one PIPE at a time can drive the PIPEFlow B bus.

The PR is also responsible for performing the burst transfers over the PIPE Bus, when an image is transferred to or from the PM.

#### *The PM*

Each PIPE has 4Mbytes of SRAM organised as 32bits by 1M (giving 20 bit address). This can be used for image storage and manipulation. The PM can be used in any way desired, or if it not required, the system can use the PM for image storage.

#### *Design Support for Sonic*

SONIC is designed to be programmed using AHDL (Altera Hardware Description Language) and compiled using Max+Plus II only.

## **2.6 RC1000**

The RC1000 is a standard PCI bus card equipped with a Xilinx Virtex family BG560 part with up to 2 million system gates. It has 8Mb of SRAM directly connected to the FPGA in four 32-bit wide memory banks. The memory is also visible to the host CPU across the PCI bus as if it were normal memory. Each of the 4 banks may be granted to either the host SRAM on the board. It is then accessible to the FPGA directly and to the host CPU either by DMA transfers across the PCI bus or simply as a virtual address. The board is equipped with two industry standard PMC connectors for directly connecting other processors and I/O devices to the FPGA; a PCI-PCI bridge chip also connects these interfaces to the host PCI bus, thereby protecting the available bandwidth from the PMC to the FPGA from host PCI bus traffic. A 50-pin unassigned header is provided for either interboard communication, allowing multiple RC1000s to be connected in parallel or for connecting custom interfaces.



**Figure 2.5 – The RC1000 board**

## 2.7 Handel-C

Handel-C is a programming language designed for compiling programs into hardware images of FPGAs or ASICs. It is basically a subset of C, extended with a few constructs for configuring the hardware device and to support generation of efficient hardware. It comprises all common expressions necessary to describe complex algorithms, but lacks processor-oriented features like pointers and floating point arithmetic. The programs are mapped into hardware at the netlist level, currently in xnf or edif format.

Different to other approaches of high-level hardware design, which actually use C to describe the behaviour and simply translate it into a netlist, Handel-C targets hardware directly, and provides a few hardware optimisation features. A big advantage, compared to the C-translators, is that variables and constants can be given a certain width, as small as one bit. When using C as the describing language, the smallest integer is possibly 8 bits wide, if not 16, which means that one wastes at least 7 1-bit registers when declaring a simple flag. Also, Handel-C provides bit manipulation operators and the possibility of parallel processing of single statements or whole modules. This can not be realised with other approaches based on a sequential language.

Comparing Handel-C with VHDL shows that the aims of these languages are quite different. VHDL is designed for hardware engineers who want to create sophisticated circuits. It provides all constructs necessary to craft complex, tailor made hardware designs. By choosing the right elements and language constructs in the right order, the specialist can specify every single gate or flip-flop built and manipulate the propagation delays of signals throughout the system. On the other hand, the use of VHDL expects the developer to have knowledge of low-level hardware and requires him to continuously think about the gate-level effects of every single code sequence. This ties up a lot of the designer's attention and can easily distract the designer from the actual algorithmic or functional subject.

In contrast to that, Handel-C is not designed to be a hardware description language, but a high-level programming language with hardware output. It doesn't provide highly specialised hardware features and allows only the design of digital, synchronous circuits. Instead of trying to cover all potentially possible design particularities, its focus is on fast prototyping and optimising at the algorithmic level. The low-level problems are hidden completely, all the gate-level decisions and optimisation are done by the compiler so that the programmer can focus his mind on the task he wants to implement.

As a consequence, hardware design using Handel-C shows a resemblance more close to programming than to hardware engineering, and in fact, this language is developed for programmers who have no hardware knowledge at all.

Handel-C is designed for fast prototyping can also be seen when working with the simulator. This small but efficient tool simulates the behaviour of the circuit at the algorithmic level, based on the semantics of the Handel-C program. Compared with hardware simulators, which usually analyse the gate-level function of the circuit, the simulation time is very short (seconds opposed to several minutes). Together with the very fast compilation, this encourages the designer to try out several implementations of the design.

To summarise, hardware design with Handel-C is very much like programming. Unlike with hardware description languages, the designer is not confronted with gate-level problems like fan-in and fan-out or choosing the appropriate type of gates or registers to be used. Apart from freeing the programmer's mind from those low-level decisions, it is much faster and more convenient to describe the system's desired behaviour at the algorithmic level. The fast compilation, combined with the high-level simulator, allows several implementation strategies to be tried out within a very short time.

## **2.8 Electronic Design Interchange Format (EDIF)**

The Electronic Design Interchange Format (EDIF) is a standardised representation of data, which is independent of specific manufactures and is designed to allow the transfer of information between otherwise incompatible systems.

In this project, EDIF acts as a link between Handel-C and SONIC. This is because the Handel-C compiler is able to generate EDIF files, which can be compiled using Max+Plus II.

## **2.9 The BMP file format**

BMP is a standard file format for computers running the Windows operating system. The format was developed by Microsoft for storing bitmap files in a device-independent bitmap (DIB) format that will allow Windows to display the bitmap on any type of display device. The term "device independent" means that the bitmap specifies pixel colour in a form independent of the method used by a display to represent colour.

Since BMP is a fairly simple file format, its structure is quite straightforward. Each bitmap file contains:

- a bitmap-file header: this contains information about the type, size, and layout of a device-independent bitmap file.

- a bitmap-information header which specifies the dimensions, compression type, and color format for the bitmap.
- a colour table, defined as an array of RGBQUAD structures, contains as many elements as there are colours in the bitmap. The colour table is not present for bitmaps with 24 color bits because each pixel is represented by 24-bit red-green-blue (RGB) values in the actual bitmap data area.
- an array of bytes which define the bitmap bits. These are the actual image data, represented by consecutive rows, or "scan lines," of the bitmap. Each scan line consists of consecutive bytes representing the pixels in the scan line, in left-to-right order.

BMP files always contain RGB data. The file can be:

- 1-bit: 2 colours (monochrome)
- 4-bit: 16 colours
- 8-bit: 256 colours.
- 24-bit: 16777216 colours, mixes 256 tints of Red with 256 tints of Green and Blue

Windows versions 3.0 and later versions support run-length encoded (RLE) formats for compressing bitmaps that use 4 bits per pixel and 8 bits per pixel.

The default filename extension of a Windows DIB file is .BMP.

## **2.10 Summary**

Several important terms and concepts have been introduced in this chapter. By now, the reader should have a good understanding of the SONIC architecture and Handel-C. The RC1000 board has been described briefly, which is also a reconfigurable platform similar to SONIC. Two different standards relevant to this project have been discussed: the EDIF and the BMP format.

## Chapter 3. Compile Time Support

### 3.1 Introduction

Although reconfigurable engines are powerful, they are difficult to program. The reasons for these difficulties are as follows. First, reconfigurable engines contain hardware and software elements running concurrently, but hardware and software are traditionally described by different languages; it is error prone to manage a parallel system using a variety of languages. Second, application developers often find hardware languages unfamiliar and difficult. Also existing languages do not support run-time hardware reconfiguration and operator sharing, techniques underpinning efficient FPGA implementations. Third, it is desirable for many graphics and imaging applications to be scalable: the same program should work, with minimum alteration, for different image resolutions or different data rates on various reconfigurable engines.

The aim of this chapter is to introduce a design framework based on the Handel-C and the EHC language, and to show how various interfaces are designed to enable Handel-C programs to work on SONIC. Three interfaces are developed:

- Memory Interface: Needed for the Handel-C program to read and write to the PIPE Memory.
- PIPEFlow interface: Used to transfer data to and from the PIPE Router.
- PIPE Bus interface: Needed for register access for the run-time facility. This is mainly used for synchronisation between the Handel-C program and the run-time facility.

The memory interface is the most complex of three interfaces.

This chapter also discusses briefly how Handel-C is extended to capture both hardware and software descriptions.

### 3.2 Motivations of using Handel-C

As explained in chapter 1, Handel-C is very much like normal programming. Unlike hardware description languages (HDL), the designer does not need to worry about low-level decisions. At the algorithmic level, which Handel-C is in, it is much faster and more convenient to describe the systems desired behaviour.

As an example, a 1D Filter written in AHDL consists of 4 sub-designs and over 100 lines of code. In Handel-C, the same design can be written with less than 50 lines of code. The only drawback of using Handel-C is that it is very hard to write efficient code, as the user has no access to gate-level decisions, which HDLs do have. For prototyping and for most situations, Handel-C is the preferred language. But for time critical applications, the user may decide to program using HDLs instead.

So far, AHDL was the only available option in programming for SONIC. The mapping of Handel-C would make the life of existing and new SONIC programmers easier.

### 3.3 Extending the Handel-C language

Although reconfigurable engines are powerful, they are difficult to program. The reasons for these difficulties are as follows. First, reconfigurable engines contain hardware and software elements running concurrently, but hardware and software are traditionally described by different languages; it is error prone to manage a parallel system using a variety of languages. Second, application developers often find hardware languages unfamiliar and difficult. Also existing languages do not support run-time hardware reconfiguration and operator sharing, techniques underpinning efficient FPGA implementations. Third, it is desirable for many graphics and imaging applications to be scalable: the same program should work, with minimum alteration, for different image resolutions or different data rates on various reconfigurable engines.

We have developed a language based on Handel-C, to include both hardware and software descriptions in a single framework. The language is called EHC (Extended Handel-C). EHC is designed in a generic manner, so that it is not restricted to a specific reconfigurable engine. EHC is designed to be portable across different reconfigurable engines, by using different platform-specific libraries. EHC includes two kinds of facilities: facilities for describing software operations, and facilities for describing hardware components called plug-ins. Software facilities include function calls for loading plug-ins onto FPGAs, and for storing and retrieving data to and from local memory. User-defined hardware functions can be developed to execute in a particular plug-in. A pre-processor for EHC has been developed for generating the appropriate files for existing software and hardware compilers.

Floating-point calculations can often be implemented more efficiently on microprocessors with a floating-point unit than on FPGAs. When partitioning a graphics pipeline between software and hardware using a reconfigurable engine, we usually execute floating-point geometric transformations on the microprocessor while implementing integer-intensive rasterisation and fill routines on the FPGAs. To pass data between software and hardware, the user can specify in EHC to use either synchronous or asynchronous communication. Synchronous communication is implemented in SONIC via one of its buses using a handshake mechanism, enabling data transfer when both communicating parties are ready. Asynchronous communication is implemented using the local memory on SONIC: for instance the hardware writes to appropriate memory locations and passes their addresses to the software which, when ready, then retrieves the data from the addresses supplied.

In EHC, two new constructs are added, the **plug** construct and the **software** construct. The plug construct is used for partitioning Handel-C programs across different processing elements of the reconfigurable engine, and the software construct is used to specify describe software operations. Detailed descriptions of these new constructs can be found in chapter 5. We shall first describe how Handel-C can be used in developing SONIC designs.

### 3.4 Requirements of mapping Handel-C on SONIC

In order to map Handel-C designs on SONIC, we should:

- Provide compile-time support for accessing the PIPE Memory, communication through the PIPE Bus and performing I/O via the PIPEFlow Bus.
- Provide run-time support by providing monitoring and control facilities.
- Utilisation of multiple PIPES through space and time partitioning and evaluate their efficiency.
- Show how reconfiguration can be used on the SONIC architecture.
- Provide a template for writing Handel-C programs for SONIC.

The extended Handel-C (EHC) should have support for software instructions such as configuring the reconfigurable engine, and accessing local memory of the FPGA board.

### 3.5 Overview of mapping Handel-C on SONIC

Figure 3.2 shows the three compile time interfaces involved in running Handel-C programs on SONIC.

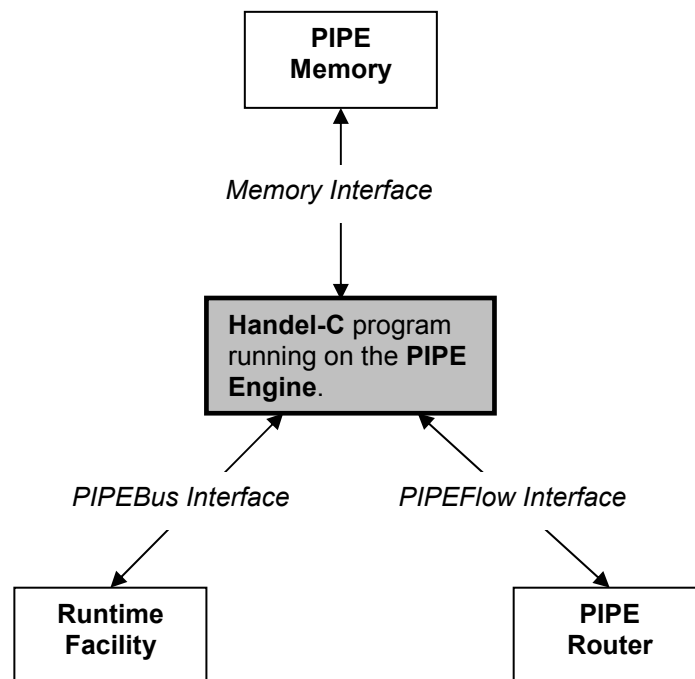
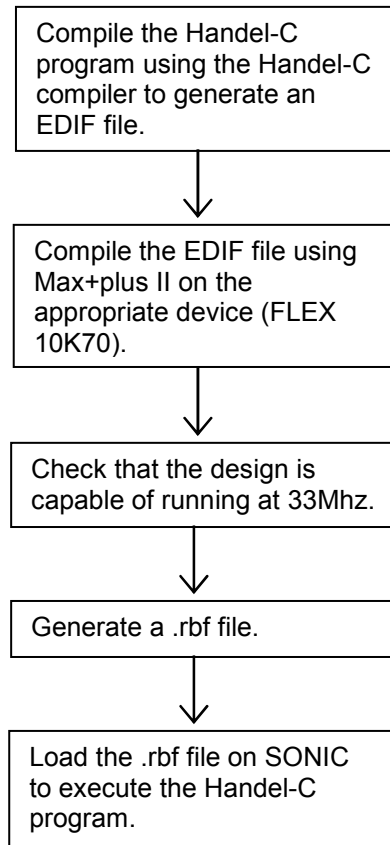


Figure 3.2 – Interfaces for the Handel-C program

Figure 3.3 shows the steps involved in producing a .rbf file (a file that can be loaded straight onto the SONIC board) from a Handel-C program.



**Figure 3.3 – Steps for mapping Handel-C on SONIC**

### 3.6 Memory Interfacing

The first objective was to write a memory interface, which should enable the PIPE Engine to read and write directly to the PIPE Memory. The PM consists of 4MB of SRAM and is arranged as 32bits x 1048576, giving 20 bit addresses.

The connections between the PE and the PM is shown in figure 3.3.

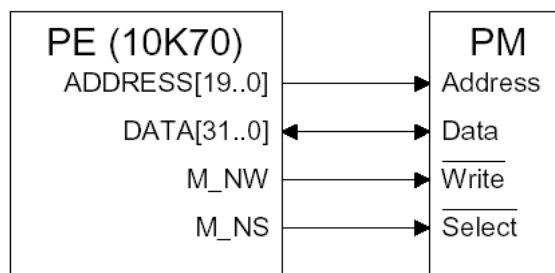


Figure 3.4 – Connections between PE and PM

The operations of the SRAM are given in table 1.

M_NS	M_NW	Mode	Data[31..0]
High	Don't care	High-Impedance	
Low	High	Write	Din
Low	High	Read	Dout

Table 3.1 – Operations of the SRAM

Figure 3.4, 3.5 and 3.6 show the timing diagrams of the SRAM, used on the PIPES (taken from the data sheet of the SRAM). These timing requirements need to be met for correct operation.

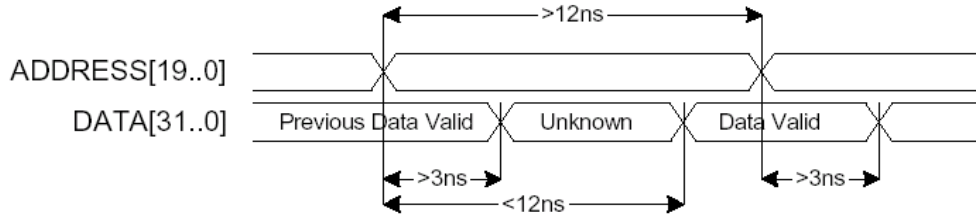


Figure 3.5 – Read Cycle: M\_NW = High, M\_NS=LOW

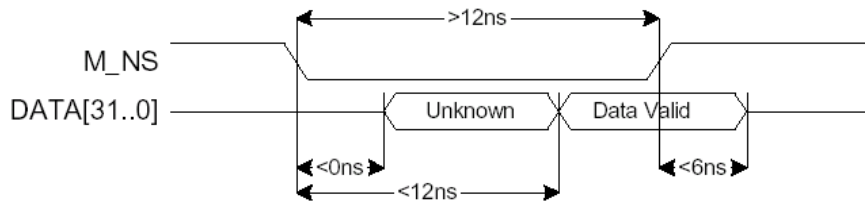


Figure 3.6 – Read Cycle: M\_NW = High

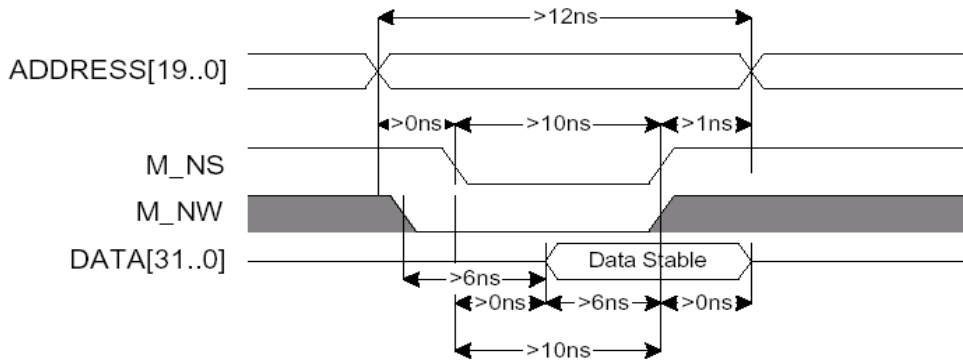


Figure 3.7 – Write Cycle

Off-chip interfacing facilities of Handel-C [16] were used to define the I/O pins between the PE and the PM. See appendix A for the PE pin allocations.

Handel-C provides the following off-chip interface types:

Type Identifier	Description
bus_in	Input bus from pins
bus_latch_in	Latched input bus from pins
bus_clock_in	Clocked input bus from pins
bus_out	Output bus to pins
bus_ts	Bi-directional tri-state bus
bus_ts_latch_in	Bi-directional tri-state bus with latched input
bus_ts_clock_in	Bi-directional tri-state bus with clocked input

**Table 3.2 – Interface Types**

The pins for ADDRESS[19..0], DATA[31..0], M\_NS, M\_NW are all declared as bi-directional tri-state busses with clocked inputs (bus\_ts\_clock\_in) as shown in Figure 3.7.

```

interface bus_ts_clock_in (unsigned int 32) m_data (m_data_out, m_data_ena == 1)
  with { data = {"171","201","97","181", "208","172","174","173",
                "6","99","119","148","9","116","8","175",
                "94","100","103","196","204","101","209","183",
                "11","113","98","195","193","7","107","95"}};

interface bus_ts_clock_in (unsigned int 20) m_addr (m_addr_out, m_addr_ena == 1)
  with { data = {"18","138","141","149","143","142","147","80",
                "146","218","23","207","88","151","139","39",
                "144","41","35","154"}};

interface bus_ts_clock_in (unsigned int 1) m_ns (m_ns_out,m_ns_ena == 1) with { data
= {"194"}};

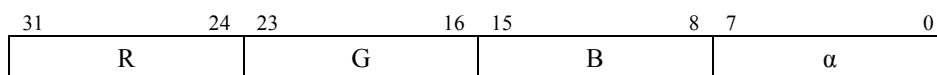
interface bus_ts_clock_in (unsigned int 1) m_nw (m_nw_out,m_nw_ena == 1) with { data
= {"152"}};

interface bus_ts_clock_in(unsigned int 1) stall(stall_out,stall_ena == 1) with { data
= {"105"}};

```

**Figure 3.8 – Interface for PM Connections**

Each 24-bit RGB pixel is stored in one 32-bit memory location as R,G,B, $\alpha$  as illustrated below.



The PM uses a little-endian architecture, since every memory access returns G,R, $\alpha$ ,B. This is very inconvenient to the Handel-C programmer, so the pin declarations of `m_data` is modified (by re-ordering the bytes) so that the PM returns R,G,B, $\alpha$ .

In order to meet the timing requirements of the memory, Simon Haynes, one of the developers of SONIC used LCELLs in AHDL [3] to give the correct asynchronous behaviour. His design gave a 3 cycle delay for reads and a 2 cycle delay for writes. However in Handel-C it is not possible to output signals asynchronously, which means that output signals can only change at the beginning of each clock cycle.

Two functions are written in the memory interface:

```
read (unsigned int 20 address, unsigned int 32 data)
for reading memory location address to data
```

```
write (unsigned int 20 address, unsigned int 32 data)
for writing data to the memory location address
```

Memory reads have a delay of 4 clock cycles and writes have a delay of 3 clock cycles. Figure 3.8 and 3.9 show the timing diagrams of a read and write respectively,

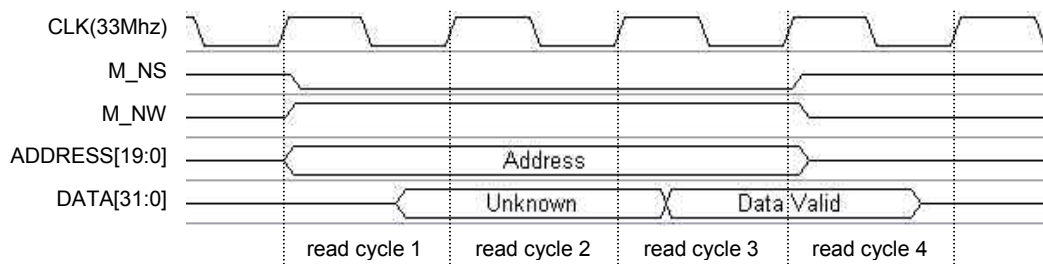


Figure 3.9 – read

Initially, all pins are tristated.

*read cycle 1:* `M_NS` is set to low and `M_NW` is set to high. The address of the memory location to be read is supplied to the memory. `DATA` is tristated.

*read cycle 2:* No action.

*read cycle 3:* No action. `DATA` becomes valid at the end of this cycle.

*read cycle 4:* Tristate all pins. Read in `DATA`.

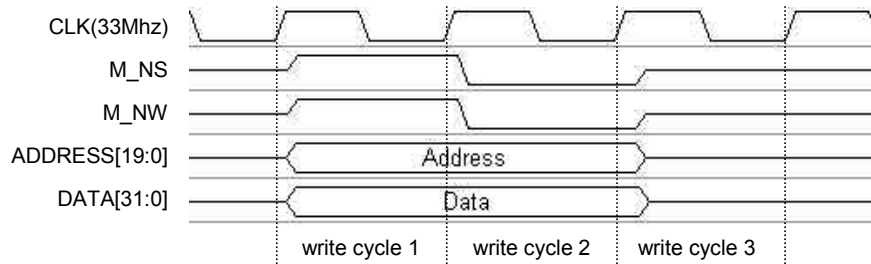


Figure 3.10 – write

Initially, all pins are tristated.

*write cycle 1:* M<sub>NS</sub> and M<sub>NW</sub> are set to high. The address of the memory location and the data to be written is supplied to the memory.

*write cycle 2:* M<sub>NS</sub> and M<sub>NW</sub> are set to low. DATA will be written to the memory during this cycle.

*write cycle 3:* Tristate all pins.

Please refer to Appendix B for the full source code of the memory interface and other interfaces.

### 3.7 Interfacing with the PIPE Bus

The PIPE bus is responsible for fast image transfer and PIPE register accesses. It is a 32 bit multiplexed address / data bus with 4 control signals.

The PIPE bus has two modes of operation: *Single Transaction Mode* which is designed for register access, and *Burst Mode* which is designed to enable the PIPE bus to operate at the maximum PCI Burst rate of 133MB/sec.

As the prime interest is register accesses (for synchronisation with the run-time facility), the single transaction mode is considered. Table 3.3 shows the signals of the PIPE Bus related to single transaction mode.

Signal	Direction w.r.t. PE	Description
PIPE_BUS_CS	IN	high when the PE is selected
PIPE_BUS_AS	IN	high when PIPE_BUS_AD[31..0] contains the address
PIPE_BUS_WRITE	IN	high when data is being written to the PIPE
PIPE_BUS_AD[31:0]	IN / OUT	32 bit Address / Data

Table 3.3 – PIPE Bus signals

Single transaction mode allows one 32 bit word to be transferred per address cycle. An address cycle immediately followed by a data cycle.

Figure 3.9 and 3.10 show the timing diagrams for a read and write respectively.

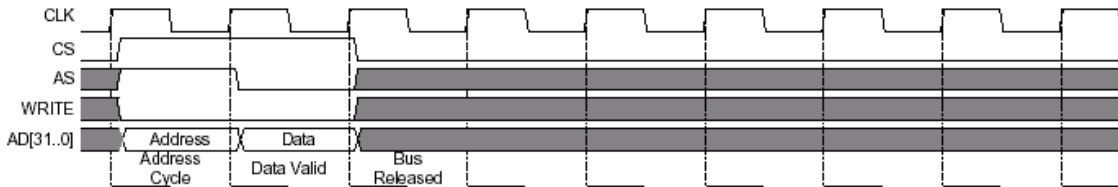


Figure 3.11 – read

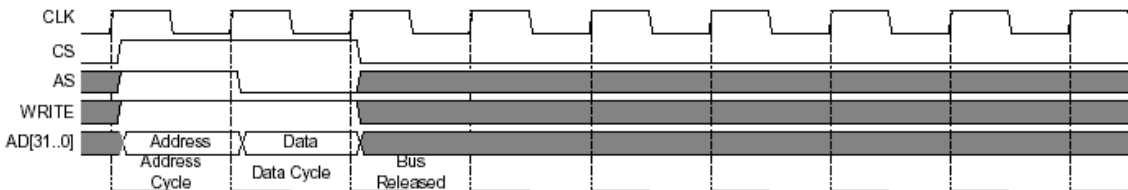


Figure 3.12 – write

Refer Appendix B for the PIPE Bus interface declarations.

### 3.8 Interfacing with the PIPEFlow Bus

In order to make use of the PR, it is necessary to implement an interface for the PIPEFlow Bus. PR enables us to transfer data between different PIPES very efficiently and even when using just one PIPE, it is much more efficient than using the memory interface alone. The PIPEFlow bus consists of 16 bits of data and 3 control bits. Its purpose is to transfer images in a raster scan fashion between PIPE Engines, and between the PIPE Memory and the PIPE Engine. The 16 bit data stream consists of a header, containing some information of the image, followed by the actual image data. Figure 3.12 shows a typical timing diagram for the PIPEFlow bus.

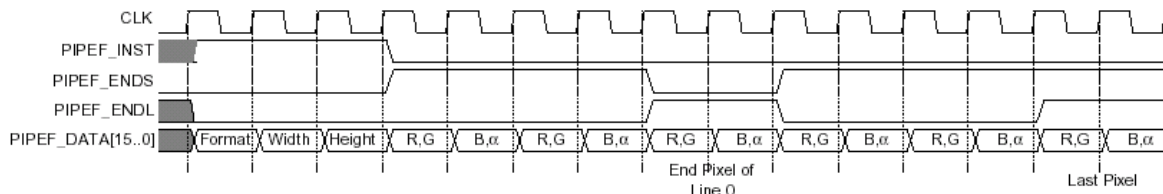


Figure 3.13 – Timing diagram for PIPE Flow Bus

PIPEF\_IN\_INST is high during the image header phase, PIPEF\_IN\_ENDS is high at the end of each strip and PIPEF\_ENDL is high at the end of each line. Figure 3.13 illustrates the PIPEFlow signals between the PE and PR.

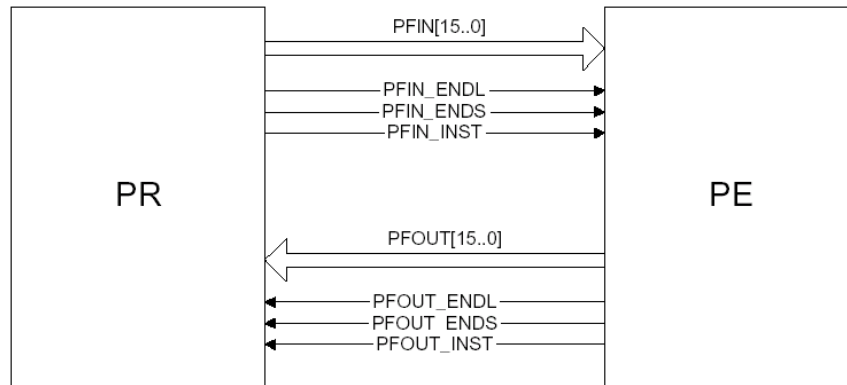


Figure 3.14 – Signals between PE and PR

The PIPEFlow input signals are declared as “bus\_in” and the PIPEFlow output signals are declared as “bus\_ts\_clock\_in”. Once again, the bytes of the data signals (PFIN[15..0], PFOUT[15..0]) are re-ordered because of the little-endian system the memory is using.

See Appendix B for the interface declarations.

### 3.9 Summary

The detailed implementations of three interfaces and the purpose of extending Handel-C have been discussed in this chapter. The memory interface shows how the timing requirements are met by increasing the latency of each memory access. We have also seen how the “little-endian, big-endian” problem is overcome by simply changing the orders of the pin declarations.

## **Chapter 4. Run-Time support**

### **4.1 Introduction**

This chapter explains a run-time facility that is developed to control and monitor the SONIC board, it also includes several examples of Handel-C running on SONIC. For run-time facility, we will see how the software communicates with the Handel-C program running on SONIC. To demonstrate the functionality of the three interfaces discussed in chapter 3, we will look at two examples implementing an image colour inverter.

## 4.2 C++ application for Run-Time support

A C++ application called SonicRun is written to enable the Handel-C / SONIC programmer to monitor and control the SONIC board. SonicRun is designed for running Handel-C programs on the SONIC board, using just one PIPE and the memory interface. The application uses the SONIC API [4] and performs following functions:

- Initialise and shutdown the SONIC board.
- Load the appropriate .rbf files on the PE and PR.
- Load a 24-bit bitmap image onto the PM.
- Save the contents of the PM to a 24-bit bitmap file.
- Load a file containing a stream of data onto the PM.
- Save the contents of the PM to an ASCII text file.
- Signal the Handel-C program on the PE to start execution.
- Signal the Handel-C program on the PE to stop execution.
- 

Figure 4.1 shows a screenshot of SonicRun.

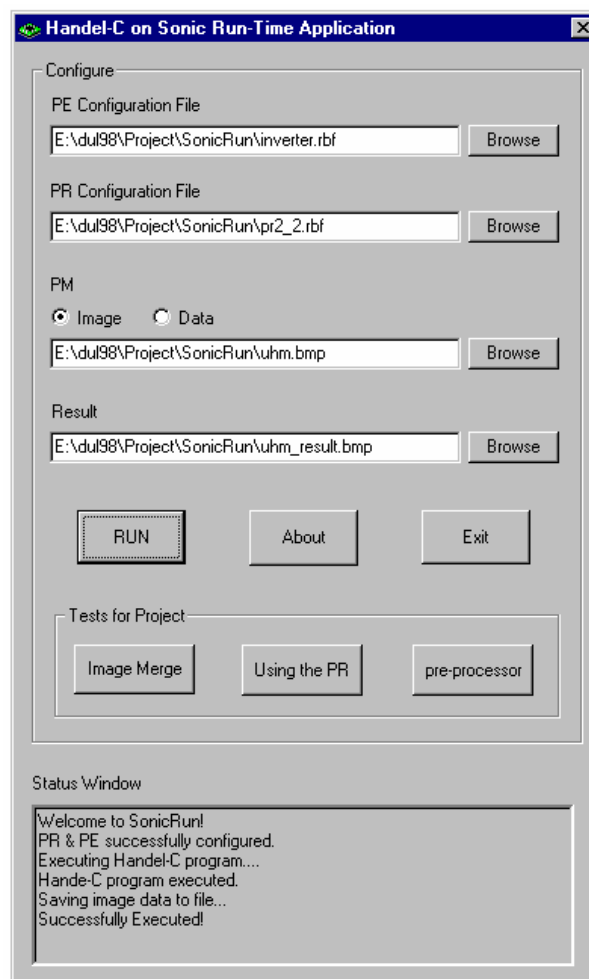
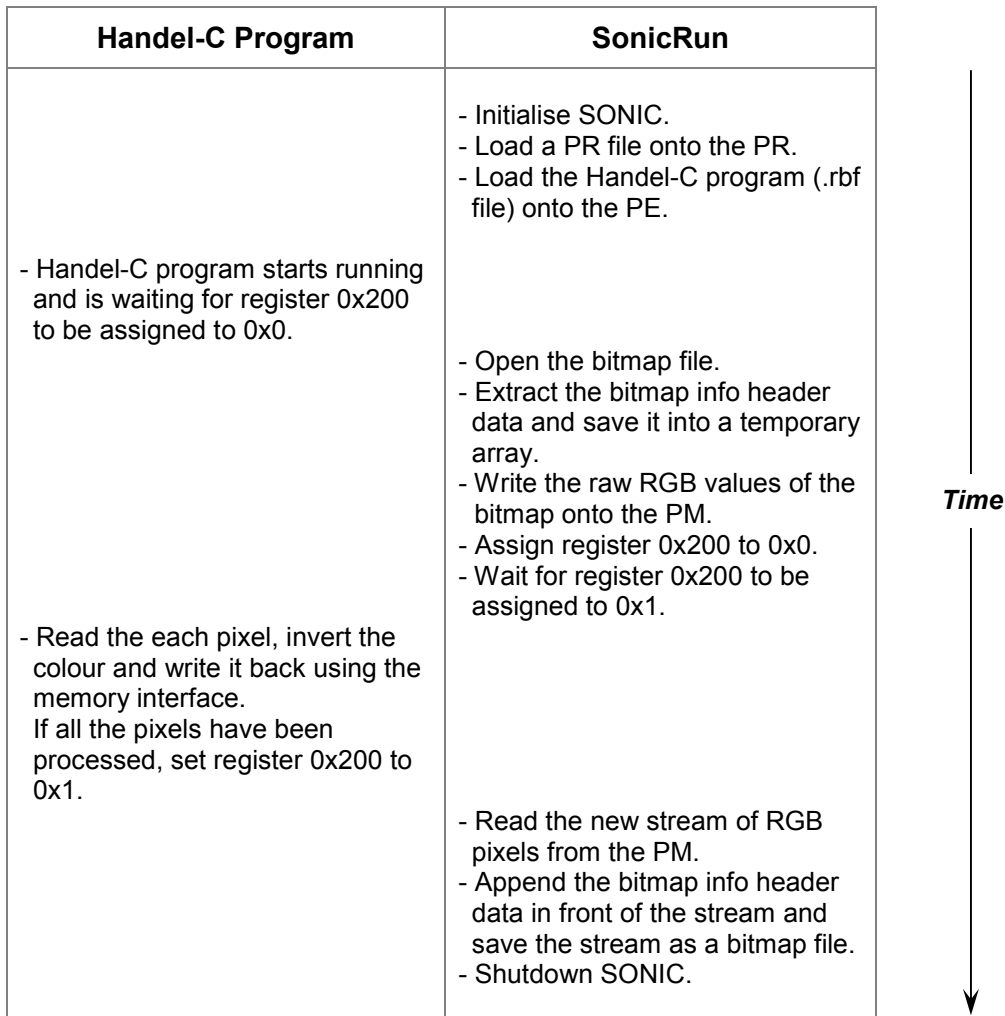


Figure 4.1 – SonicRun

The user needs to specify the .rbf files for the PE and the PR, the source file (a data stream file or a 24-bit bitmap file) and the file where the result should be stored.

Figure 4.2 illustrates the operation of the Handel-C program and SonicRun during a typical run.



**Figure 4.2 – Handel-C and SonicRun**

### 4.3 Example using the Memory Interface and PIPE Bus

An image colour inverter is implemented to demonstrate the functionality of the memory interface and the PIPE Bus.

The memory interface is used to read a pixel from the memory and to write it back. The PIPE Bus allows registers to be read and written to the PE using the SONIC API [4].

Each pixel is read from the memory using the **read** function and written back with the new RGB values using the **write** function. Figure 4.3 shows the Handel-C code of this process.

```

for (i=0; i<total_pixels; i++) {
    read(i,pixel);
    par {
        red   = 255-pixel[31:24];
        green = 255-pixel[23:16];
        blue  = 255-pixel[15:8];
        alpha = pixel[7:0];
    }
    write(i, red@green@blue@alpha);
}
    
```

Figure 4.3 – Colour inversion

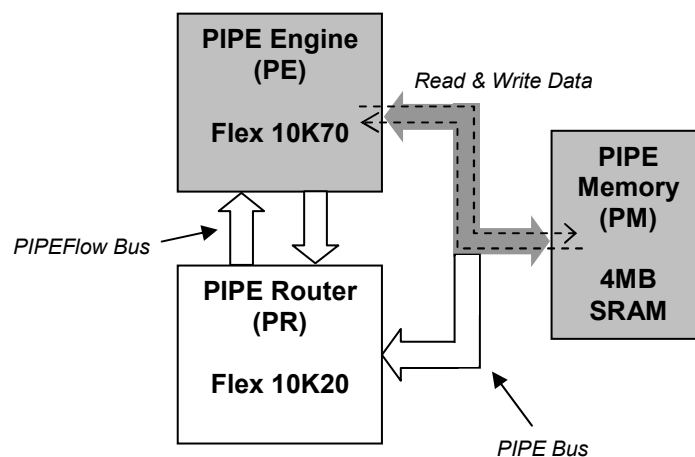


Figure 4.4 – Data Flow

As shown in the figure above, only the PIPE Bus is used to transfer data between the PE and PM, the PIPEFlow Bus and the PR is not used at all. This is known as direct access mode.

Figure 4.5 shows a 640x480 24-bit bitmap image before and after processing.



**Figure 4.5 (a) – Original Image**



**Figure 4.5 (b) – Colour inverted**

It takes 4 clock cycles to read a pixel from the memory, 3 clock cycles to write a pixel back to the memory, and 1 clock cycle to invert the colours of a pixel. In total, it takes  $8 \times 640 \times 480 = 2457600$  clock cycles to process the entire 640x480 bitmap image.

The full Handel-C source code of this colour inverter is shown in appendix C.

#### 4.4 Example using the PIPEFlow Bus

To demonstrate how useful the PIPEFlow mode is compared to direct access mode (i.e. using just the PE and PM), the colour inverter was implemented in PIPEFlow mode.

The PR routes data stored in the PM to PIPEFlow IN of the PE. The PE performs the colour inversion to the incoming pixels and outputs it to PIPEFlow OUT. The PR routes data from PIPEFlow OUT back to the PM. This is illustrated in figure 4.6.

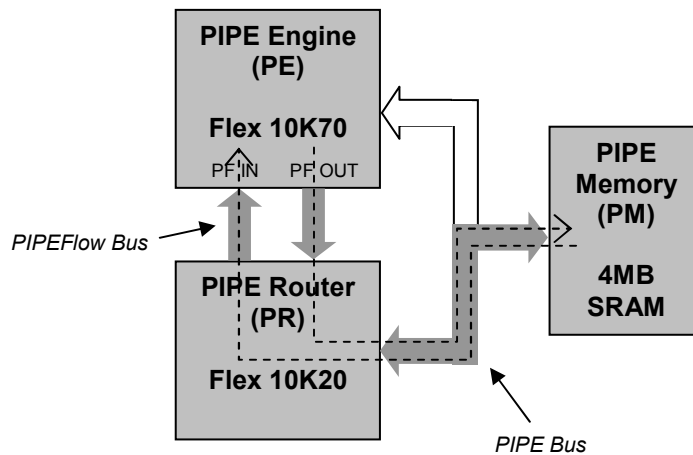


Figure 4.6 – Data Flow

As discussed in chapter 3.7, the PR transfers one whole pixel every 2 clock cycles. The colour inverter is fully pipelined. It has a latency of 3 clock cycles and a throughput of 1 clock cycle. Ignoring the latency which is negligible, the PIPE took  $640 \times 480 \times 2 = 614400$  clock cycles to process the same frame as the colour inverter using direct access mode. The direct access mode version took 2457600 clock cycles to process a frame, therefore there is a speed up of a factor of  $2457600 / 614400 = 4$ .

The SONIC board runs at 33Mhz, so the period of each clock cycle is 33ns.

Therefore it takes  $33\text{ns} \times 614400 = 20.2\text{ms}$  to invert the colours of a  $640 \times 480$  frame. Since, the time taken to transfer the image to and from the PM is 37.4ms, the total time to process the picture is  $20.2\text{ms} + 37.4\text{ms} = 57.6\text{ms}$ .

It should be noted that the fastest time that a frame can be processed is 37.4ms where the PCI bus becomes fully utilised.

If we process video sequences, we would be able to process  $640 \times 480$  frames about 17 frames/sec. The full Handel-C source code of this colour inverter is shown in appendix C.

## 4.5 Template for Handel-C programs

The template for Handel-C programs for execution on SonicRun is shown in figure 4.7. This illustration is based on the image colour inverter discussed in section 4.3. This template is used when using the memory interface and PIPE Bus.

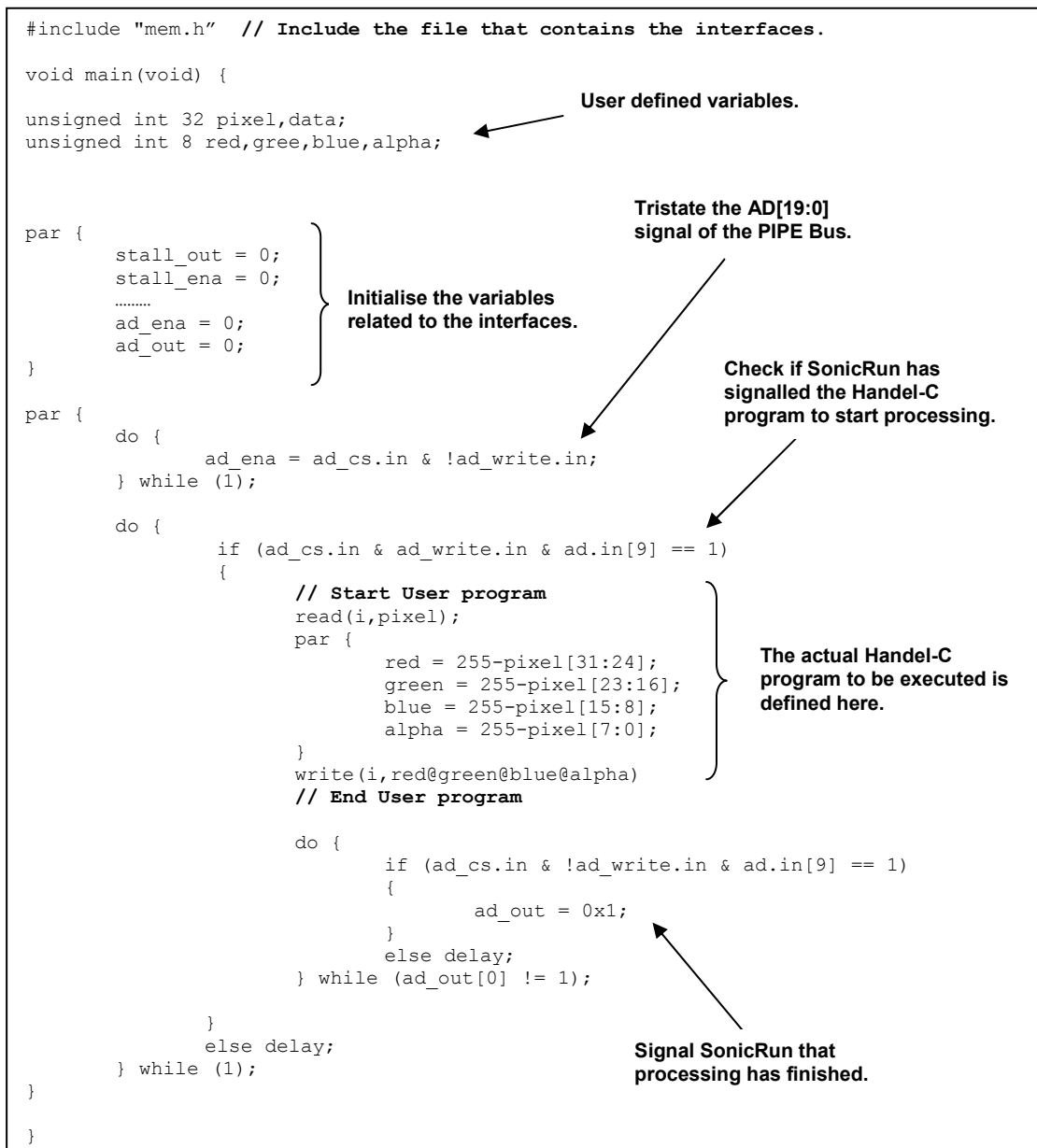


Figure 4.7 – Handel-C Code Template

## **4.6 Summary**

The functionality of SonicRun, which is the run-time facility was introduced and the way in which SonicRun synchronised with the Handel-C program, running on SONIC, was discussed.

In addition, two examples of image colour conversion were discussed. Both of them performed the same task in different ways. We have seen that reading many pixels from the PIPE Memory using the memory interface is not very efficient, because of the latencies involved of the memory accesses. When reading or writing pixels from the memory, it is much better to use the PIPE Router. The memory interface should only be used for reading in variables, such as coefficients for a filter.

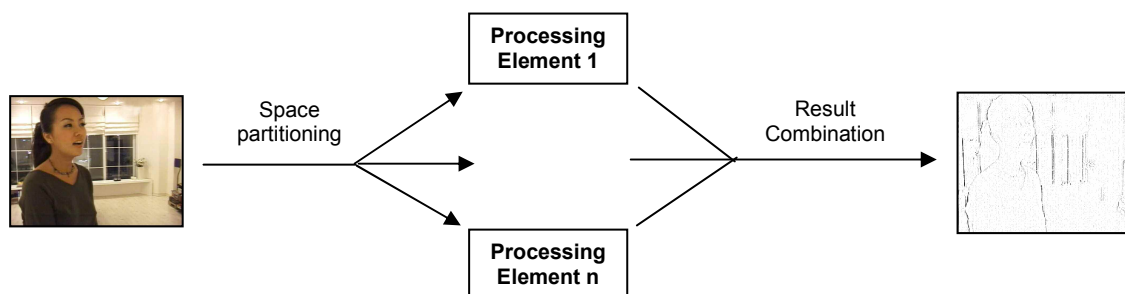
## Chapter 5. Partitioning in Space and Time

### 5.1 Introduction

Many image processing algorithms have inherent parallelism present in them, enabling processing to be distributed. There are two main ways for tasks to be distributed: in space and in time. Partitioning in space is done by dividing each frame into smaller parts and processing each part in parallel. Partitioning in time is achieved by processing many frames in parallel, often in the form of pipelined processing.

### 5.2 Partitioning in Space

Space partitioning exploits parallelism by distributing different partitions of an image onto different processing elements. It is useful when the image is very large, such as in high-definition television formats, or to accelerate processing speed. Figure 5.1 illustrates space partitioning an image for edge detection. All processing elements implement the same hardware, which performs Gaussian and Laplacian filtering.



**Figure 5.1 – Space partitioning on reconfigurable engines**

In order to see how the SONIC architecture can be used to implement spatially distributed processing, the colour inverter example from Chapter 3 is used. The two pipes shown in figure 5.1 are both configured with the colour inverter. PM0 is loaded with the upper part of the image and PM1 is loaded with the lower part. Each PIPE processes its part of the image and generates the appropriate part of the output image. The processing of the two pipes takes place in parallel. After processing, each part of the image is read from the appropriate PIPES to construct the entire output image.

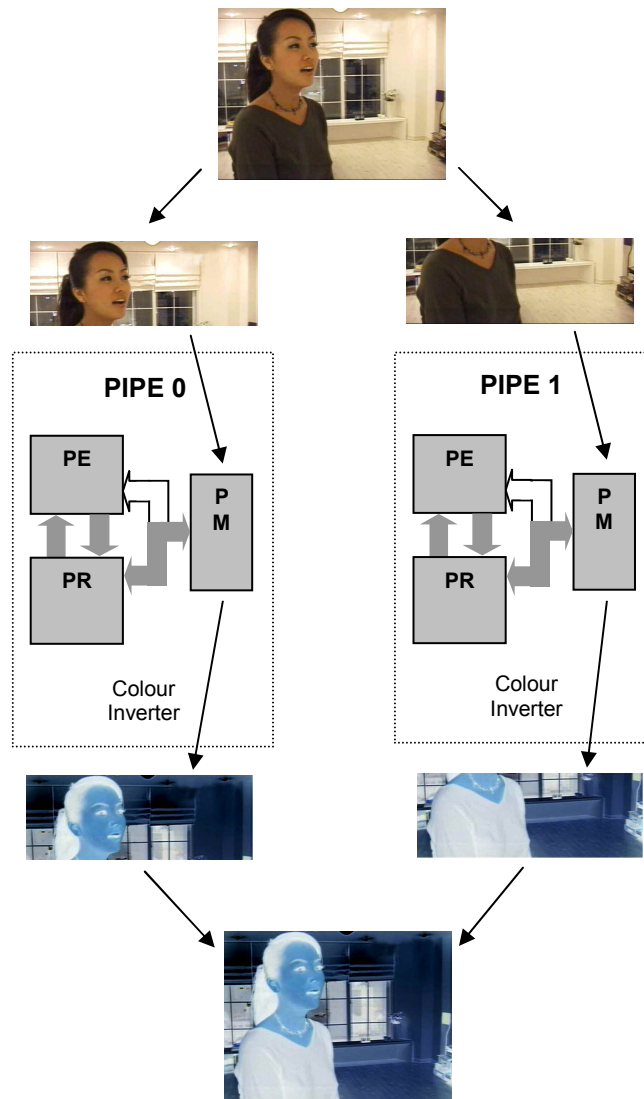


Figure 5.2 – Space Partitioning on SONIC

The time taken by this approach can be defined by:

$$Total\ Time = T + \frac{P}{n}$$

where  $T$  is the time taken to transfer the image to the PM,  $P$  is the processing time of a PIPE and  $n$  is the number of PIPEs used.

The configuration time for the PIPEs is excluded, since the PIPEs only need to be configured once, making the configuration time negligible when processing video sequences.

The speed up of using multiple PIPEs when compared to a single PIPE is given by:

$$SpeedUp = n \left( \frac{T + P}{T * n + P} \right)$$

Provided that the processing time  $P$  is significantly larger than the transfer time  $T$ , the speed up is approximately proportional to the number of PIPEs used,  $n$ .

Algorithms such as filtering and image transforms, whose output pixel depend on several input pixels, may require the whole image to be loaded on each PIPE. SONIC has a ‘broadcast’ mechanism, which allows the entire image to be written into the PMs of all the PIPEs. So the total image transfer time should take the same time as the colour inverter example above.

A software function is developed for parameterised space partitioning. The function takes in five parameters:

- the functions the PIPEs should perform
- the source image
- the destination image
- number of PIPEs to use
- whether to use automatic partitioning

For example, for non-automatic partitioning, if the user chooses 4 PIPEs, the function will split the source image into 4 horizontal parts, load them across 4 PIPEs and process simultaneously. After processing, all the parts are combined together and written to the destination.

For automatic sharing, the function will detect the size of the image and determine if it will fit into a single PIPE. So, if the image takes up less than 4MB, it is just processed on a single PIPE. However, high-resolution frames such as HDTV do not fit into a single PIPE. In this case, the partitioner will determine the number of PIPEs needed to accommodate the frame and process the image across multiple PIPEs.

The space partitioner was tested using simple algorithms, such as image colour inversion where the pixels are independent of one another. When dealing with filters, the pixels are dependent of its neighbouring pixels, therefore when dividing the image into different parts, overlaps need to be taken. Also, the implementation of 2D partitioning would need minor modifications in order use nested loops.

### 5.3 Partitioning in Time

Often in a video sequence the same algorithm is applied to many image frames in succession. In this case, a task can be partitioned in time to process multiple frames concurrently in a pipeline fashion for reconfigurable engines with multiple processing elements. A higher data rate can be supported by including more pipeline stages in the

processing elements. Edge detection is achieved by applying Gaussian filtering to the image first, which softens the edges and filter out noise. Then Laplacian is applied to the image to detect the edges. It is important to apply Gaussian to the image first, since Laplacian is very sensitive to noise. Figure 4 shows how the two-stage edge detection method can be optimised using time partitioning. Note that the two processing elements operate in parallel on two successive image frames.

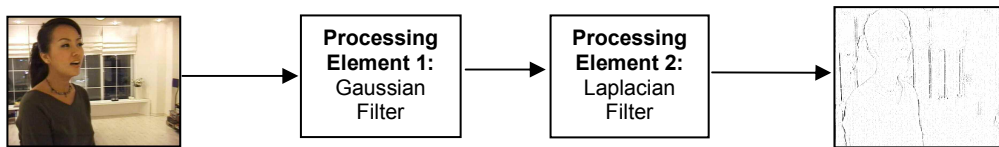


Figure 5.3 – Time partitioning on reconfigurable engines

The 3x3 masks used for Gaussian and Laplacian are shown below.

1	2	1
2	4	2
1	2	1

Figure 5.4 – Mask used for Gaussian

0	-1	0
-1	4	-1
0	-1	0

Figure 5.5 – Mask used for Laplacian

The 2D isotropic Gaussian is separable into x and y components. Thus the 2D convolution can be performed by first convolving with a 1D Gaussian in the x direction, and then convolving another 1D Gaussian in the y direction. The same principle applies to the Laplacian. Convolution in 1D is necessary since the PR transfers data in ‘raster-scan’ fashion.

Figure 5.5 and 5.6 show the 1D x component mask that would produce the full mask shown in Figure 5.3 and 5.4. The y component is exactly the same but is oriented vertically.

1	2	1
---	---	---

Figure 5.6 – Mask used for 1D horizontal Gaussian

-1	2	-1
----	---	----

Figure 5.7 – Mask used for 1D horizontal Laplacian

In total, the image will have to be processed 4 times. 2 passes for horizontal & vertical 1D Gaussian filtering and 2 passes for 1D Laplacian.

This principle can be applied to the SONIC architecture by configuring several PIPEs with algorithms required, and use the PIPEs to process frames in parallel. PIPE0 is configured for Gaussian filtering and PIPE1 for Laplacian.

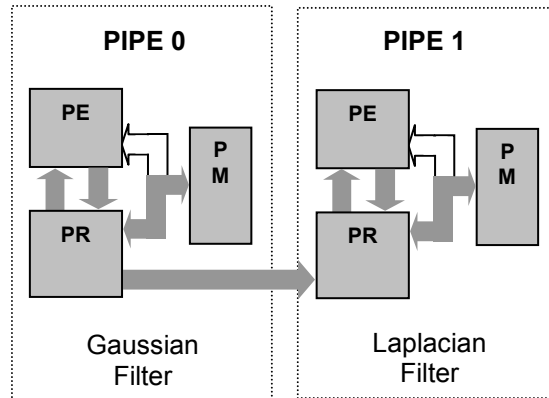


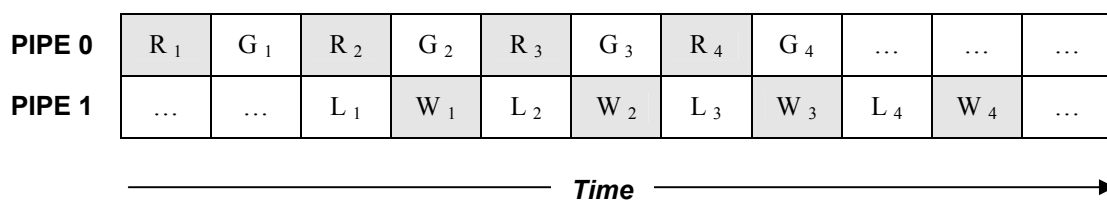
Figure 5.8 – Edge Detection on SONIC

The image is initially loaded to PIPE0 where Gaussian filtering is performed in 2 passes. PR0 routes the data to PIPE1 after the 2<sup>nd</sup> Gaussian pass, where Laplacian filtering is performed. The final edged detected image is stored in PM1.

Consider processing a video sequence. When the current frame has been Gaussian filtered and is passed to PIPE1, PIPE0 is idle which means that the next frame can be processed PIPE0. Therefore, frame  $t$  and frame  $t-1$  can be processed concurrently.

One limitation is that only one image transfer can be made at a time, since the PIPE Bus must be used sequentially.

The figure below shows how video sequences are processed, exploiting temporal parallelism.



R<sub>t</sub> = Read Frame t  
 W<sub>t</sub> = Write Frame t  
 G<sub>t</sub> = Gaussian Filter Frame t  
 L<sub>t</sub> = Laplacian Filter Frame t

Figure 5.9 – Tasks partitioned in Time

AVI TestBench [20], a C++ application written to benchmark the video processing performance of RC1000 [21] is modified to benchmark SONIC.

The performance of edge detector running SONIC is compared to the performance of a software implementation. The performance is measured in terms of frames per second. Figure 5.9 shows the original 320x240 video clip, and figure 10 and 11 show the edge detected image in SONIC and in software on a PentiumII running at 300Mhz. Filtering is done on the green component of each pixel, and this is duplicated across the red and blue components to produce a black & white image.



Figure 5.10 – Original Video Clip

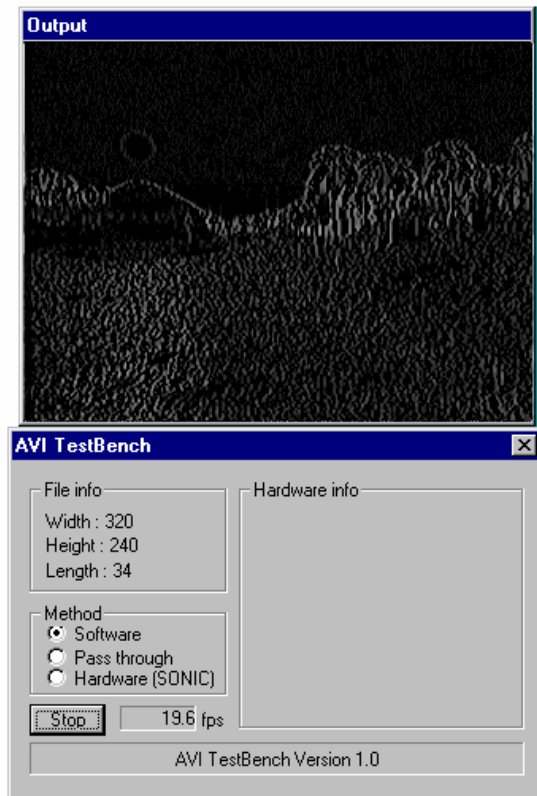


Figure 5.11 – Edge Detected in Software

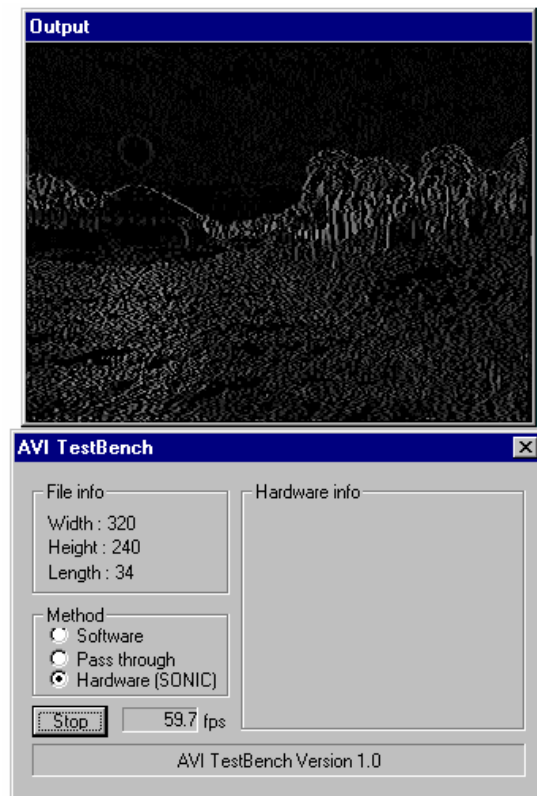


Figure 5.12 – Edge Detected in SONIC

The SONIC implementation is able to process the video clip at about 60 fps, whereas the software implementation only managed 20 fps.

This demonstrates how powerful FPGAs can be compared to a general-purpose processor. Even though the clock speed of the SONIC board is only about 1/10 of the PentiumII, the FPGA does a lot more tasks concurrently resulting in much faster performance.

## **5.4 Summary**

We have seen how Handel-C programs can be partitioned in space and time, resulting in improved performance. Especially when performing edge detection, the speed up of three times compared to the software implementation, which runs at a much higher clock speed is impressive.

## Chapter 6. Extending the Handel-C language

### 6.1 Introduction

The Handel-C language is extended to capture both software and hardware descriptions. In addition, the extension provides facilities for partitioning Handel-C programs onto different processing elements processing elements as described in Chapter 5, and for managing run-time reconfiguration as described in Chapter 7. These facilities improve design abstraction and design portability: for instance they can reduce the size of programs by up to 75% (see Table 6.3 in Section 6.3). The resulting language is called EHC (Extended Handel-C). A pre-processor to process EHC programs is written using a package called ANTLR [32], ANOther Tool for Language Recognition (formerly PCCTS), which is a computer language translation program.

### 6.2 Hardware Partitioning

In some cases, the SONIC programmer may wish to partition his Handel-C program into several PIPEs. Hardware partitioning may be desirable in several cases, for example if the design is too large to fit into one chip, there is too much routing congestion or to exploit temporal parallelism. The hardware partitioning considered here is done in terms of functions, i.e. each function performs a particular task. The functions may be all loaded onto one PIPE, or partitioned across several PIPEs.

Consider a situation where 3 tasks, shown below, are required to be performed on an image.

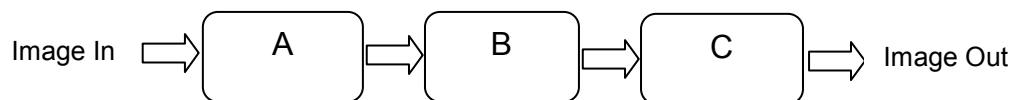


Figure 6.1

Each task is written as a function, and the input of each function is the PIPEFlow IN data stream and the output is PIPEFlow OUT. EHC provides a new syntax for partitioning functions across different PIPEs: the **plug** construct (plug as in plug-in). For example:

```

plug0 {
    A();
    B();
}

plug1 {
    C();
}
  
```

}

This indicates that functions A and B are included (A then B being the execution order) in plug0 and function C is included in plug1. Any of these plugs can be loaded onto any PIPE: this is controlled by the software. Such partitioning maybe useful if tasks A, B and C do not fit into one PIPE, or when temporally distributed processing is desired. We can see that the user requires the minimum knowledge: all the user needs to do is to specify which tasks are in which plug using the simple **plug** syntax. Figure 6.2 shows how such EHC program should be written.

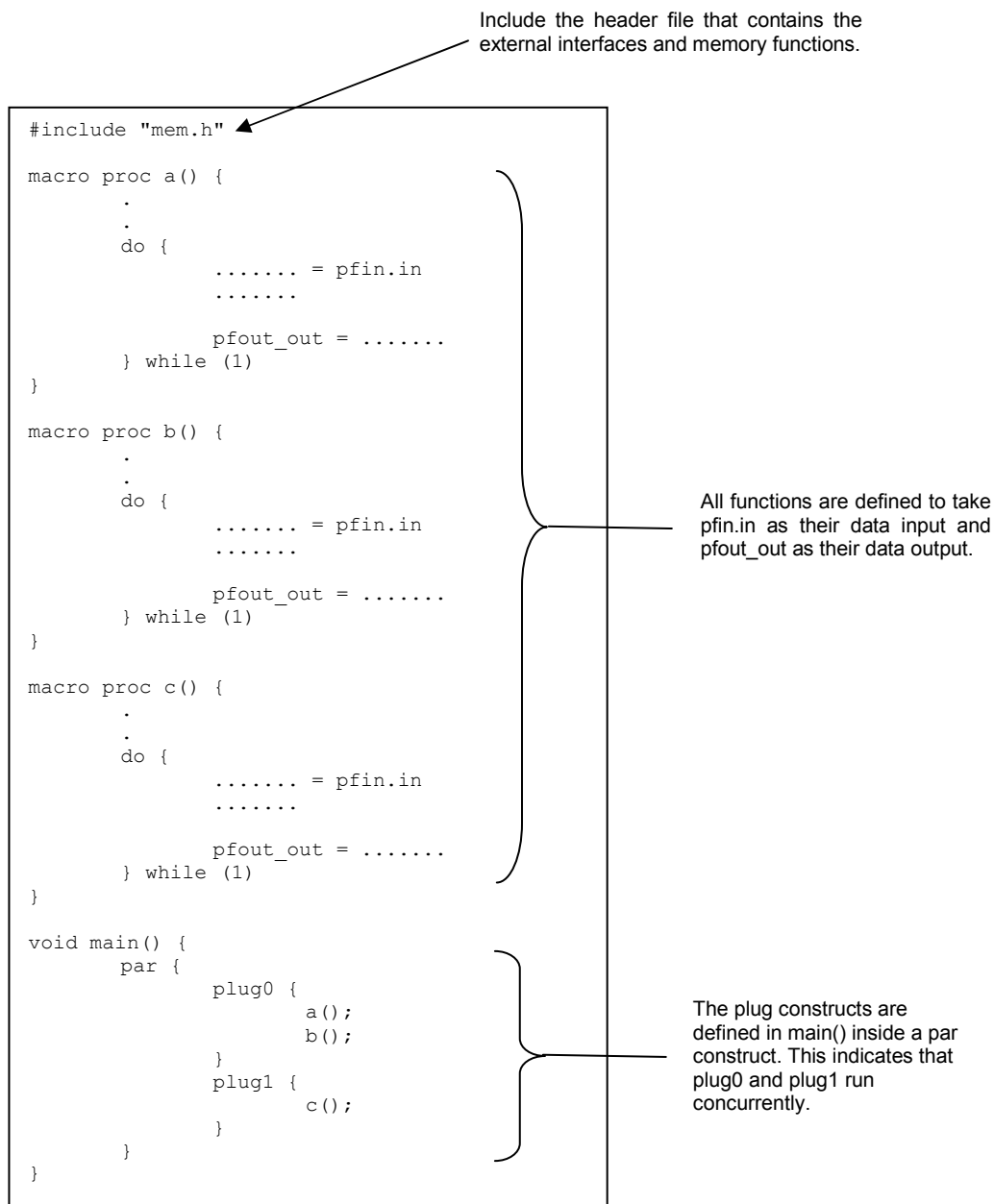


Figure 6.2 – Partitioning across PIPEs using EHC

When there are two or more functions to be executed in a single plug, the pre-processor renames the intermediate I/O pins to registers. For example, in the EHC program shown above, functions A and B are to be executed in plug0, and therefore on the same PIPE. The functions are defined to take the PIPEFlow IN pins as their input and PIPEFlow OUT pins as their output. We want A and B to be executed on a single PIPE, so the I/O pins that are defined between functions A and B will have to be renamed to registers. Figure 6.3 illustrates the effect of I/O renaming.

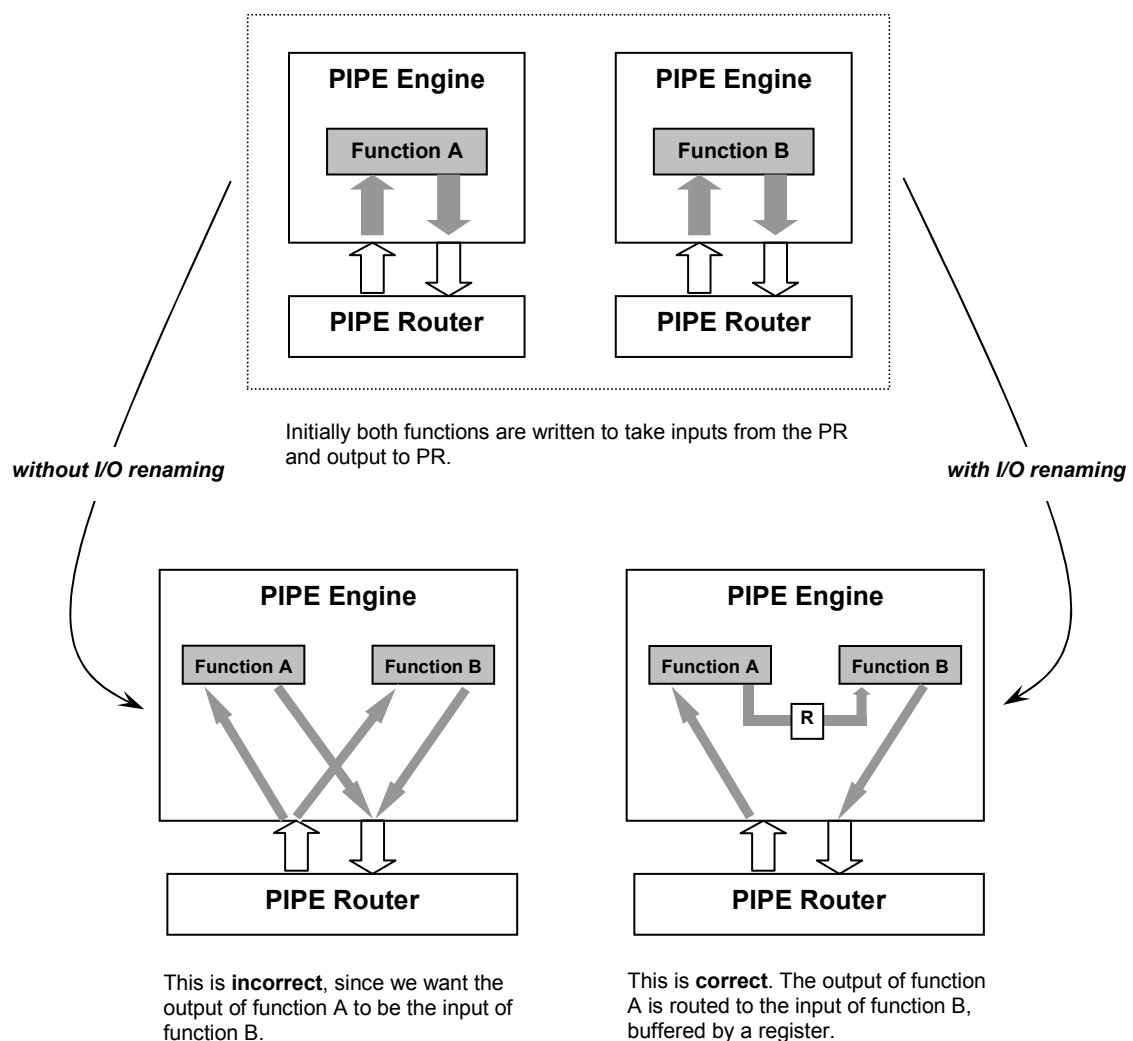
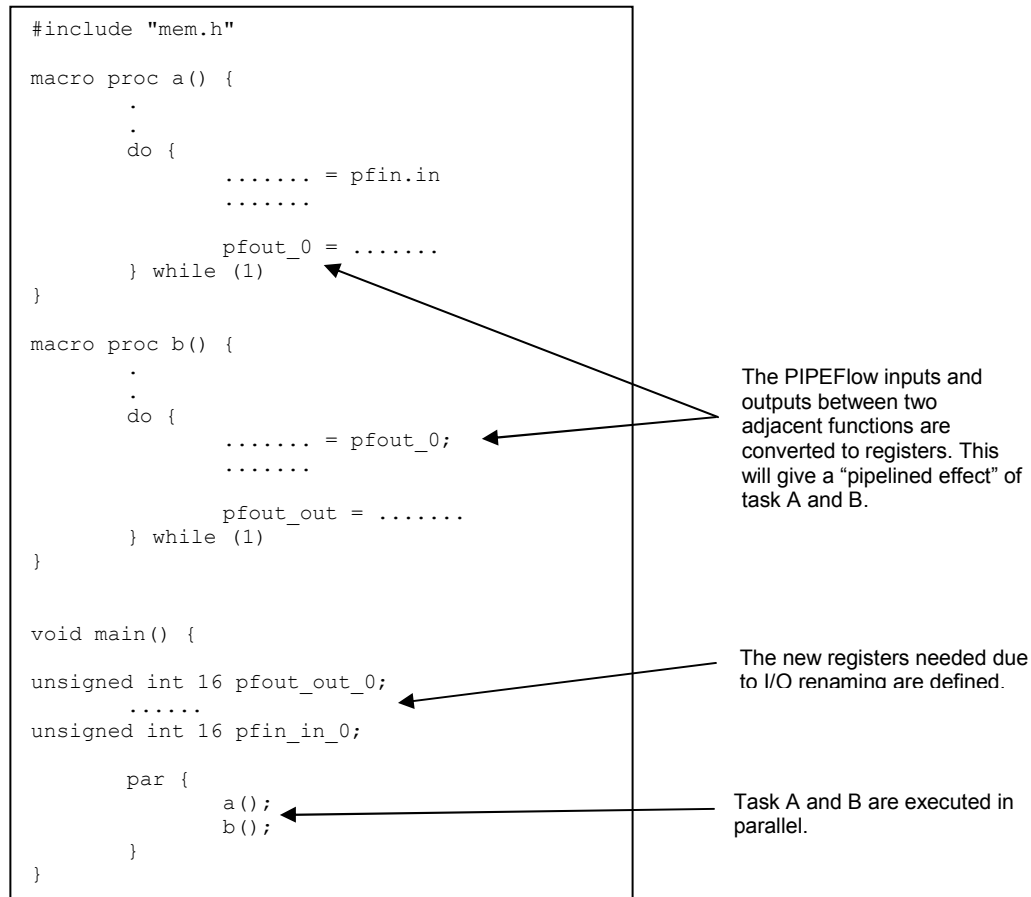


Figure 6.3 – I/O Renaming

After renaming, the pre-processor generates the variable declarations needed due to the I/O renaming. Two separate Handel-C files will be generated and compiled to EDIF by

the pre-processor: one file for plug0 and the other for plug1. These plugs can then be loaded into any PIPE the user desires.

The Handel-C file generated for plug0 is shown in figure 6.4.



**Figure 6.4 – Handel-C program generated by the pre-processor for PIPE0**

To test the functionality of the pre-processor, an EHC program with plug constructs as shown in figure 6.2 is written. The function `a()` is written to perform gamma correction, `b()` to perform 1-D Gaussian filtering and `c()` to perform colour to black and white conversion. The program was partitioned so that plug0 was to carry out gamma correction followed by 1-D Gaussian filtering, plug1 to carry out colour to black and white conversion.

After using the pre-processor to generate the EDIF files for each plug, plug0 is loaded onto PIPE0 and plug1 onto PIPE1, and the tasks performed successfully. Several partitioning configurations are tested, such as executing all three tasks on one PIPE, all of which run without any problems. It should be clear that this type of partitioning is well suited for partitioning tasks in time. Since EHC is designed to be portable across different reconfigurable systems, this partitioning idea could also be applied to partition tasks across 2 RC1000-PP boards [21] running in parallel over the PCI bus for example.

### 6.3 Software Descriptions in EHC

The pre-processor is extended to support a new construct: the **software** construct. This construct is used to define software descriptions within EHC. Software facilities include function calls for loading plug-ins onto FPGAs, and for storing and retrieving data to and from local memory. The reason for this implementation is to give the user a single programming environment to define both software and hardware descriptions. There is no need for the user to write a separate run-time application in C. The pre-processor translates the software instructions in EHC into the real SONIC API commands in C.

The table below shows the software instructions that can be defined in EHC and their translation in C by the pre-processor:

Software descriptions in EHC	Description	Equivalent C statements using SONIC API
<code>#setboard (board_ID);</code>	Set the reconfigurable platform. This is needed so that the correct platform specific libraries are loaded. If using SONIC, board_ID would be SONIC.	
<code>init();</code>	Initialise the reconfigurable engine.	<code>Sonic_Initialise();</code>
<code>fpga_conf(fpga_no, config_plug);</code>	Configure the FPGA fpga_no with config_plug.	<code>Sonic_Conf_PE (fpga_no, config_plug);</code>
<code>fpga_mode(fpga_no, args*);</code>	Set the mode of FPGA fpga_no with the arguments args*. <code>fpga_mode(pipe_no, _strip, _rcol, _ruppper, _wcol, _wupper);</code>	<pre> <b>if fpga_mode(pipe_no, _strip, _rcol, _ruppper, _wcol, _wupper):</b> Int strip = _strip; Int rcol = _rcol; Int ruppper = _ruppper; Int wcol = _wcol; Int wupper = _wupper; Sonic_PR_ImageMode_Write(pipe_no, &amp;strip, &amp;rcol, &amp;ruppper, &amp;wcol, &amp;wupper)  <b>if fpga_mode(pipe_no, _data):</b> DWORD data = _data; Sonic_PR_Route_Write (pipe_no, &amp;data) </pre>
<code>mem_write(fpga_no, data_file, offset);</code>	Write data_file to the local memory of fpga_no from offset.	<pre> //open bmp_file //extract all the header //fields and save to //a temporally storage //extract Iwidth and Iheight //save all the raw pixel data //into an array called pData Sonic_PR_ImageSize_Write(pipe_no, &amp;Iwidth, &amp;Iheight); </pre>

		Sonic_PM_Write(pipe_no, pData, (Iwith*Iheight*4), offset);
mem_read(pipe_no, data_file, offset);	Read data from local memory of fpga_no from offset to data_file.	Sonic_PM_Read(pipe_no, pData, (Iwith*Iheight*4), offset); //append the header fields to //the raw data from the //temporally storage //save the image to bmp_file
fpga_start(fpga_no);	Signal FPGA fpga_no to start processing.	DWORD Data = 1; Sonic_PR_Pipeflow_Write(fpga_no, &Data);
fpga_wait(fpga_no);	Stay idle till FPGA fpga_no has finished processing.	do Sonic_PR_Pipeflow_Read(fpga_no, &Data);while (Data != 0x2);
close();	Shutdown the reconfigurable engine.	Sonic_Close()

**Table 6.1 – Software descriptions in EHC**

When configuring the data routing of the PR using the **fpga\_mode(fpga\_no, arg)** function, the **arg** needs to be a two digit hexadecimal number, which needs to be worked out using a table in the SONIC specification. However for easier understanding, the hex digits used in this report have been replaced with the symbolic names as shown in table 6.2. Refer to figure 2.3 for the general architecture of the SONIC board.

Symbolic Name	Description	Hex number used in SONIC API
PM_PFIN_and_PFOUT_PM	Route data from PIPE Memory to PIPE Flow In and from PIPE Flow Out to PIPE Memory.	0x3F
PM_PFIN_and_PFOUT_PFRIGHT	Route data from PIPE Memory to PIPE Flow In and from PIPE Flow Out to PIPE Flow Right.	0x07
PFLEFT_PFIN_and_PFOUT_PM	Route data from PIPE Flow Left to PIPE Flow In and from PIPE Flow Out to PIPE Memory.	0x3C

**Table 6.2 – Symbolic names of the Routing Configurations**

The figure below shows a simple example of the software descriptions in EHC, where Gaussian filtering is performed using two PIPES: PIPE0 for horizontal pass and PIPE1 for vertical.

```

#setboard(SONIC); //set the reconfigurable engine to SONIC system
software {
  init(); //initialise sonic
  fpga_mode(0,PM_PFIN_and_PFOUT_PFRIGHT);
  //setup the routing for PIPE0
  fpga_mode(1,PFLEFT_PFIN_and_PFOUT_PM);
  //setup the routing for PIPE1
  fpga_conf(0,plug1); //configure PIPE0 for gaussian
  fpga_conf(1,plug1); //configure PIPE1 for gaussian
  fpga_mode(0,0,0,0,0,0); //horizontal scan for PIPE0
  fpga_mode(1,0,1,0,1,0); //vertical scan for PIPE1
  mem_write(0,"src.bmp",0); //load image to PIPE0
  fpga_start(0); //start processing in PIPE0, PIPE1
  fpga_wait(1); //wait until data ready at PE of PIPE1
  mem_read(0,"dst.bmp",0); //store image
  shutdown(); //shutdown
}

plug1 {
  gaussian();
}

```

Figure 6.5 – EHC program using software and plug constructs

Looking at the code, it is clearly compact and simple to use, and the equivalent statements in C requires a lot more statements. As mentioned earlier, the aim of the pre-processing facilities is to enable the user to specify all necessary hardware and software control for reconfigurable engines in a single piece of EHC code.

Writing programs in EHC proves to be much more convenient, since both hardware and software parts can be written in the same program. Also, because the software instructions are simplified, EHC results in much more compact code as shown in Table 6.3.

	Lines of EHC code	Lines of Handel-C and C code	Number of gates	Number of registers
<b>Gaussian Filtering</b>	70	310	127	87
<b>Image Merging</b>	60	260	214	187
<b>Elliptic Wave Filter</b>	321	455	11247	3912

Table 6.3 – Comparing EHC programs

## 6.4 Making EHC portable

EHC is designed to be platform independent, which means that the same EHC code should work for different kinds of reconfigurable engines. The previous section demonstrated the functionality of EHC on the SONIC platform. EHC is modified to support the RC1000 platform, as well. At the beginning of the EHC program, the user needs to specify the platform the user wishes to compile for using the **#setboard(boardID)** syntax, where the boardID is the platform.

There are some architectural differences between SONIC and RC1000. Firstly, whereas SONIC is a multi-FPGA platform, the RC1000 has a single FPGA to carry out all the tasks. Secondly, SONIC has a dedicated FPGA (PIPE Router) for routing data between or within processing elements. However, this is not the case with RC1000, which uses direct connection between the FPGA and the memory.

Considering these differences, EHC needs to be modified for portability between those two platforms: only one processing element will be used and the PIPE Router will not be used (i.e. the memory interface will be used). Table 6.4 shows the new modified EHC syntax. The **fpga\_mode** syntax has been removed since there is no need to configure the PIPE Router anymore. Because there is only one PIPE being used, the **fpga\_start** and **fpga\_wait** syntax have been simplified to **fpga\_send\_receive**, which signals the FPGA to start and waits for it to stop. Also because we are not using the PIPE Router anymore, there is no need to use the **plug** constructs.

EHC syntax	Description
<code>#setboard (board_ID);</code>	Set the reconfigurable platform. This is needed so that the correct platform specific libraries are loaded. If using SONIC, board_ID would be SONIC.
<code>init();</code>	Initialise the reconfigurable engine.
<code>fpga_conf(fpga_no, function);</code>	Configure the FPGA fpga_no with function.
<code>mem_write(fpga_no, data_file, offset);</code>	Write data_file to the local memory of fpga_no from offset.
<code>mem_read(pipe_no, data_file, offset);</code>	Read data from local memory of fpga_no from offset to data_file.
<code>fpga_send_receive(fpga_no);</code>	Signal FPGA fpga_no to start processing and wait for the FGPA to finish processing.
<code>close();</code>	Shutdown the reconfigurable engine.

Table 6.4 – Syntax for portable EHC

In addition to the modified EHC software statements, the **read(address, variable)** and the **write(address, value)** functions need to be translated to the platform specific commands. For SONIC, no alternation is needed,

however for RC1000, the pre-processor needs to modify the memory access functions to the following (only bank0 is used for testing purposes):

```
read(address, variable)  —→  PP1000ReadBank0(variable, address);  
write(address, value)   —→  PP1000WriteBank0(address, value);
```

**Figure 6.6 – Memory access function translation for RC1000**

For synchronisation purposes with the software, two new statements are added to the Handel-C part of EHC: **receive()** which waits for a signal from the software to start, and **send()** which signals the software that execution has finished.

To demonstrate the portability of EHC, a program which merges two images together is written. The formula for merging two images, A and B is given by:

$$C(x,y) = (1-\alpha) * A(x,y) + \alpha * B(x,y)$$

This means that when alpha is 0, only A is used to give C. As alpha increases, more and more of B becomes merged with A to produce C. When alpha reaches 1, C becomes entirely B. Figure 6.7 shows the EHC code for the image merging, which works on both SONIC and RC1000 simply by changing the **#setboard(boadID)** statement. Images A and B are stored in the upper and lower part of the memory respectively.

```
#setboard(RC1000);

macro proc image_merge() {
  unsigned int 32 pixel1,pixel2,data,pixel;
  unsigned int 8 red, green, blue, alpha;
  unsigned int 20 i;

  receive(); //wait for a signal from the software
  for (i=0; i[17] == 0; i++) {
    read(i,pixel1); //read pixel from image A
    read(i+524288,pixel2); //read pixel from image B
    par { //merge the two pixels
      red = (pixel1[31:24]>>1) + (pixel2[31:24]>>1);
      green = (pixel1[23:16]>>1) + (pixel2[23:16]>>1);
      blue = (pixel1[15:8]>>1) + (pixel2[15:8]>>1);
      alpha = (pixel1[7:0]>>1) + (pixel2[7:0]>>1);
    }
    write(i,red@green@blue@alpha); //write pixel to image C
  }
  send(); //signal the software that execution has finished
}

void main(void) {
software{
  init(); //initialise the board
  fpga_conf(0,image_merge); //configure the board with image_merge
  mem_write(0,image_a.bmp,0); //write image A to upper part of memory
  mem_write(0,image_b.bmp,524288); //write image B to lower part of memory
  fpga_send_receive(0); //signal the hardware to start and wait
  //for it to finish executing
  mem_read(0,image_c.bmp,0); //read the merged picture to image C
  close(); //shutdown the board
}
}
```

**Figure 6.7 – Image merging EHC program**

Figure 6.8 shows the source images A and B, and the merged image C.



Figure 6.8(a) – Image A



Figure 6.8(b) – Image B



Figure 6.8(c) – Image C

## 6.5 Summary

Handel-C is extended to include two new constructs; the plug and the software construct. The plug construct proves to be useful and simple to use for partitioning functions across multiple processing elements. The principle of “I/O renaming” is introduced which in effect, creates virtual PIPE Engines within physical PIPE Engines when two or more functions are loaded into a single PIPE. Using the software construct, the user can specify software and descriptions within the same framework. We have seen how the same EHC code can be ported onto different platforms, using the image merging as an example. Implementations can be produced more rapidly than those using existing methods, and their performance is often comparable to or faster than previous implementations.

## Chapter 7. Run-Time Reconfiguration

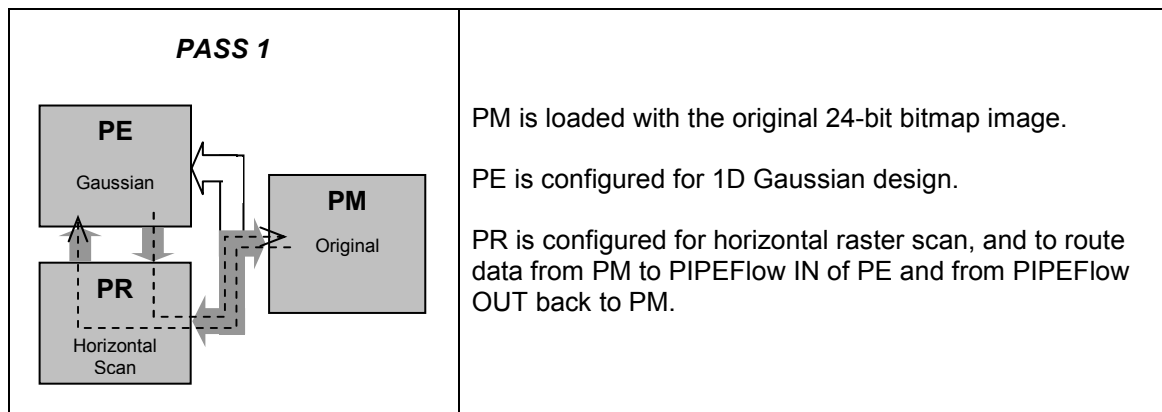
### 7.1 Introduction

FPGAs are capable of being reconfigured at run-time, they have significant potential for improved performance and resource usage for various applications. Reconfigurability provides them with an increasingly competitive edge over microprocessors, which tend to be flexible but slow, and over ASICs, which tend to be fast but inflexible. Some predict that even microprocessors will eventually be implemented using reconfigurable hardware [30]. In this chapter we will discuss the ways reconfiguration can be used on SONIC.

### 7.2 Example of reconfiguration: Edge Detector

This example will demonstrate how edge detection can be performed using just one PIPE exploiting the reconfigurability of the SONIC board.

As discussed in chapter 5.3, the image needs to be processed 4 times: 2 passes for horizontal & vertical 1D Gaussian filtering and 2 passes for 1D Laplacian. The diagram below shows what is happening on the PIPE during the 4 passes.



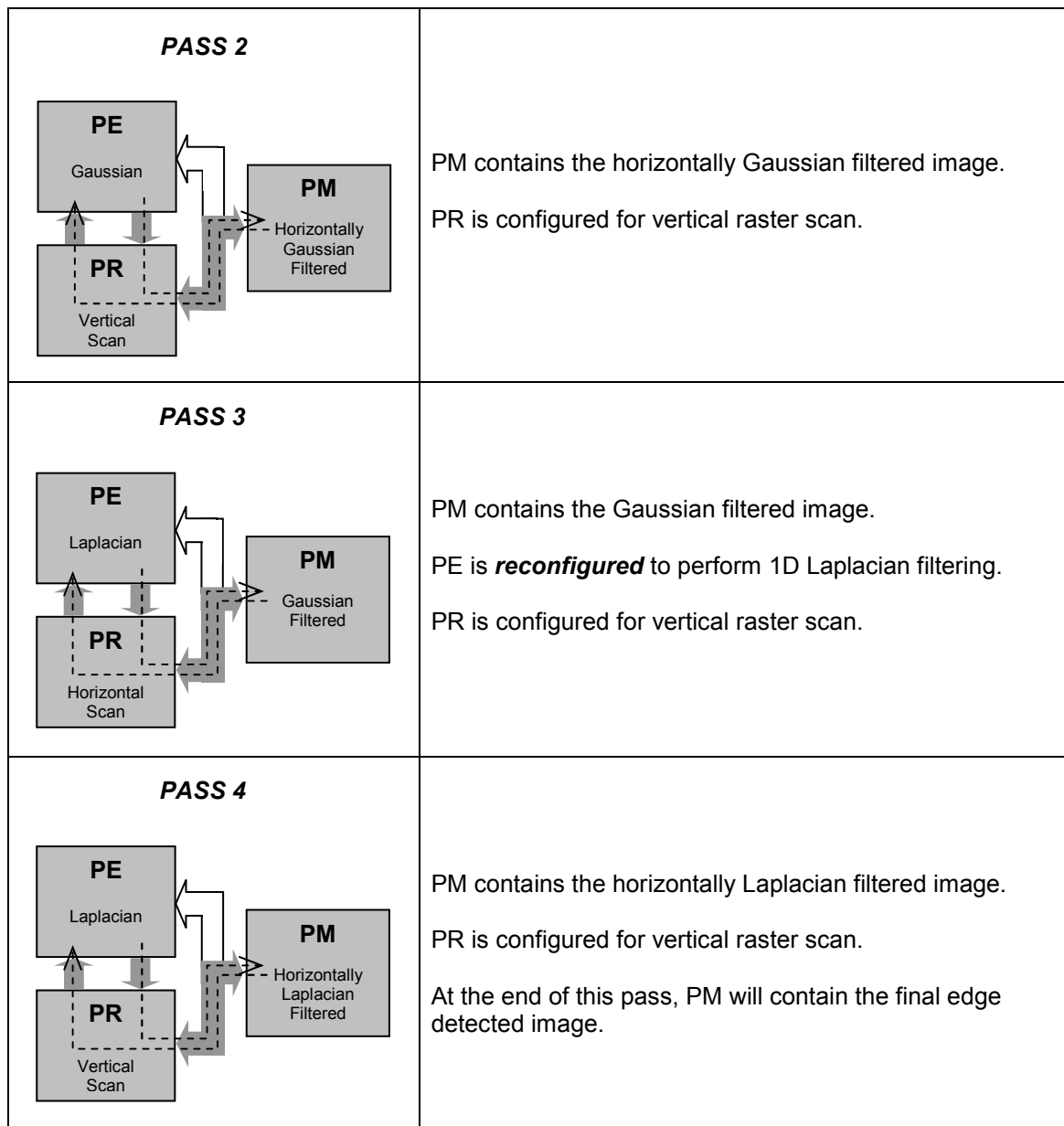


Figure 7.1 – Reconfiguration on SONIC

The original image (640x480, 24bit bitmap), Gaussian filtered image and Laplacian filtered image are shown below.



**Figure 7.2(a) – Original Image**



**Figure 7.2(b) – Image after Gaussian**



**Figure 7.2(c) – Edge detected Image**

This implementation is rather inefficient, due to the relatively long reconfiguration time of PE, which takes approximately 100ms. If we were to process video sequences, the PE would need to be reconfigured twice per frame.

It would be more efficient to use two PIPEs in series, one for Gaussian and one for Laplacian, such as the implementation that was considered in chapter 4.2. This eliminates the need for reconfiguration for every frame.

EHC explained in the previous chapter is used to perform the edge detection on a single PIPE, PIPE0. The software and plug constructs shown below are used for this process.

```
#setboard(SONIC); //set the reconfigurable engine to SONIC system
software {
  init(); //initialise sonic
  fpga_mode(0,PM_PFIN_and_PFOUT_PM);
  fpga_conf(0,plug1); //setup the routing for PIPE0
  mem_write(0,"src.bmp",0); //configure PIPE0 for gaussian
  fpga_mode(0,0,0,0,0,0); //load image onto PIPE0
  fpga_start(0); //horizontal scan for PIPE0
  fpga_wait(0); //start processing in PIPE0
  fpga_mode(0,0,1,0,1,0); //wait for the processing to finish in PIPE0
  fpga_start(0); //vertical scan for PIPE0
  fpga_wait(0); //start processing in PIPE0
  fpga_conf(0,plug2); //reconfigure PIPE0 for laplacian
  fpga_start(0); //start processing in PIPE0
  fpga_wait(0); //wait for the processing to finish in PIPE0
  fpga_mode(0,0,0,0,0,0); //horizontal scan for PIPE0
  fpga_start(0); //start processing in PIPE0
  fpga_wait(0); //wait for the processing to finish in PIPE0
  mem_read(0,"dst.bmp",0); //store image
  shutdown(); //shutdown
}

plug1 {
  gaussian();
}

plug2 {
  laplacian();
}
```

**Figure 7.3 – Reconfiguration using EHC**

The full Handel-C source code and the C program generated by the pre-processor are shown in appendix E.

### 7.3 Pipeline Morphing

Pipeline morphing [26] is a method used for reducing the latency involved in reconfiguring one pipeline to another. The basic idea is to overlap computation and reconfiguration: the first few pipeline stages are reconfigured to implement new functions, while the rest of the pipeline stages are completing the current computation. Instead of reconfiguring the entire pipeline at once, pipeline morphing involves morphing one pipeline to another. Figure 7.4 shows how a 3-stage pipeline X can be morphed into a pipeline Y in 3 steps.

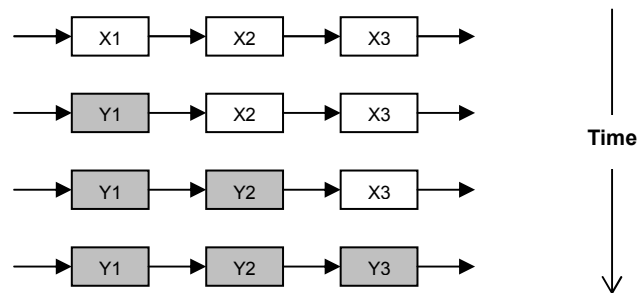


Figure 7.4 – Pipeline morphing

It should be clear from this example that during morphing, the flow of reconfiguration is synchronous with the flow of data, and hence the pipeline latency is eliminated. However, pipeline morphing can slow the pipeline down if the time of reconfiguration is longer than the pipeline processing time. Therefore, this is particularly suited for devices supporting rapid reconfiguration.

The SONIC architecture is very well suited for pipeline morphing, because of the PIPE architecture SONIC uses. Each PIPE can be considered as a pipeline stage. The interconnection between each pipeline would be very fast, due to the efficient PIPEFlow bus implementation.

Suppose we are processing a video stream using 2 PIPEs. PIPE0 performing task A and PIPE1 performing task B. We assume that the processing time of task A and B are longer than the reconfiguration time. If we choose PIPE0 to perform task C and PIPE1 to perform task D, we need to reconfigure the PIPEs. Without using pipeline morphing, the video stream will stop while the PIPEs are reconfiguring. However using pipeline morphing, a smooth transition to task C and D can be achieved, without having to stop the video stream. This is illustrated in figure 7.5.

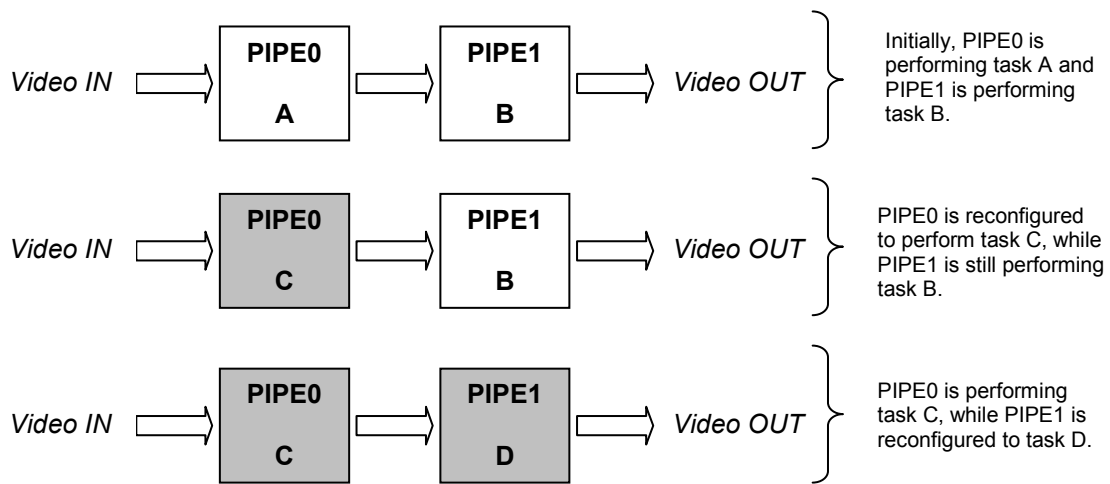


Figure 7.5 – Pipeline morphing on SONIC

Using EHC, pipeline morphing could be applied with the software construct such as the one shown below.

```
#setboard(SONIC);           //set the reconfigurable engine to SONIC system
software {
  init();                   //initialise sonic
  fpga_mode(0,PM_PFIN_and_PFOUT_PM);
                           //setup the routing for PIPE0
  fpga_mode(1,PM_PFIN_and_PFOUT_PM);
                           //setup the routing for PIPE1
  fpga_conf(0,plug_A);     //configure PIPE0 for A
  fpga_conf(1,plug_B);     //configure PIPE1 for B
  .
  .
  .
  fpga_start(0);           //start processing in PIPE0
  fpga_start(1);           //start processing in PIPE1
  fpga_wait(0);            //wait for the processing to finish in PIPE0
  fpga_wait(1);            //wait for the processing to finish in PIPE1

  fpga_start(1);           //start processing in PIPE1
  fpga_conf(0,plug_C);     //reconfigure PIPE0 to perform C, during this
                           //reconfiguration, PIPE1 is still processing B
  fpga_wait(1);            //wait for the processing to finish in PIPE1

  fpga_start(0);           //start processing in PIPE0
  fpga_conf(1,plug_D);     //reconfigure PIPE1 to perform D, during this
                           //reconfiguration, PIPE0 is still processing C
  fpga_wait(0);            //wait for the processing to finish in PIPE0

  fpga_start(0);           //start processing in PIPE0
  fpga_start(1);           //start processing in PIPE1
  fpga_wait(0);            //wait for the processing to finish in PIPE0
  fpga_wait(1);            //wait for the processing to finish in PIPE1
  .
  .
  .
  shutdown();              //shutdown
```

Figure 7.6 – Pipeline morphing using the pre-processor

## 6.4 Summary

In this chapter, we have seen how the SONIC board can be reconfigured at run-time to perform different tasks. The edge detector example pointed out that it is possible to perform 2 tasks on one PIPE using reconfiguration, but the implementation is inefficient due to the overhead involved in reconfiguration. The principles of pipeline morphing and its application on SONIC using EHC were also discussed.

## Chapter 8. Operator Sharing

### 8.1 Introduction

FPGA architectures are well suited to some operators such as binary addition, but other operators such as binary multiplication results in inefficient use of the available FPGA resources. In some applications, over 70% of the FPGA resources are used for multiplication. If there are many multiplications in a design, it would be a good idea to share the multipliers to reduce area. Because of the decrease in area, there is less congestion in the FPGA, resulting in improved speed.

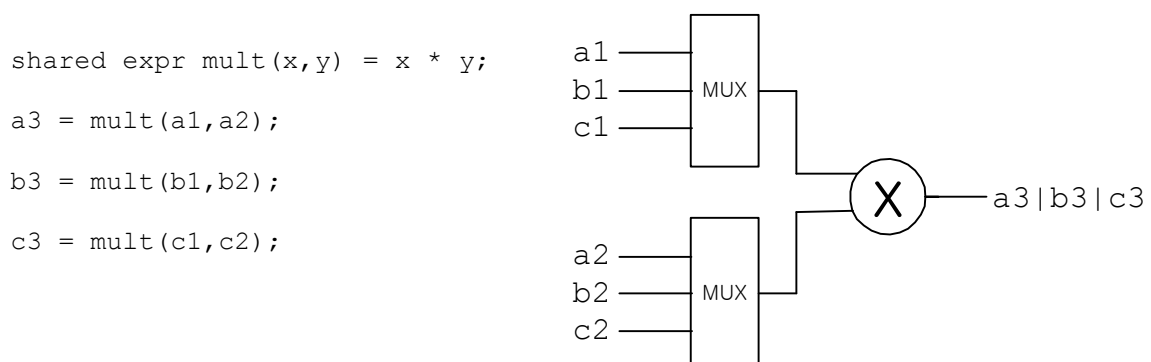
In Handel-C, each operator is built with separate circuitry. For example, every “\*” with non-computable operands supplied at run-time generates a new multiplier. This repetition of hardware increases the area of a design.

In contrast, with resource sharing, several “\*” operations can be implemented with a single multiplier, which can reduce the amount of hardware required.

Handel-C provides shared expressions, which allow hardware to be re-used between different parts of the program. This is best explained with the example shown below:



In the case above, three separate multipliers are generated. Hardware is wasted since each multiplier is only used once and none of them are executed in parallel. To optimise for speed, the designer can use the ‘**par**’ construct to indicate that the three multipliers are to operate concurrently. To optimise for space, the designer can use a ‘**shared**’ expression as used below:



In this example, only one multiplier is implemented and is used in every cycle which makes a better use of hardware. Multiplexers are used to supply the appropriate operands to the multiplier in each cycle.

Large amounts of resource sharing may cause routing problems in the FPGA since the amount of routing resources is fixed. Also, the maximum operating frequency of the FPGA may be reduced due to increased number and lengths of the interconnects.

## 8.2 Scope and Restrictions of sharing in Handel-C

Not all operations in a design can be shared. This section describes how to tell whether operations are candidates for sharing hardware.

*(1) Expressions cannot be shared by two different parts of the program in the same clock cycle.*

For example:

```
shared expr mult(x,y) = x * y;

par
{
    a3 = mult(a1,a2);
    b3 = mult(b1,b2);
}
```

This would not work since a single multiplier cannot be used twice in the same clock cycle.

*(2) Control Flow Conflicts*

Two operations can be shared only if no execution path exists from the start of the block to the end of the block that reaches both operations. For example, if two operations lie in separate branches of an 'if' or 'case' statement, they are not on the same path (and can be shared). The example below illustrates control flow conflicts for 'if' statements.

```
par
{
    a1 = a2 + a3;

    if (cond_1)
        b1 = b2+b3
    else {
        c1 = c2 + c3;
        if (cond_2)
            d1 = d2 + d3;
```

```

        else
            d1 = d4 + d5;
    }
    if (!cond_1)
        e1 = e2 + e3;
    else
        e1 = e4 + e5;
}

```

Table 8.1 summarizes the possible sharing operations in the example above. A ‘no’ indicates that sharing is not allowed because of the flow of control (execution path) through the block. A ‘yes’ means sharing is allowed.

	a2+a3	b2+b3	c2+c3	d2+d3	d4+e5	e2+e3	e4+e5
a2+a3	-	No	No	No	No	No	No
b2+b3	No	-	Yes	Yes	Yes	No	No
c2+c3	No	Yes	-	No	No	No	No
d2+d3	No	Yes	No	-	Yes	No	No
d4+e5	No	Yes	No	Yes	-	No	No
e2+e3	No	No	No	No	No	-	Yes
e4+e5	No	No	No	No	No	Yes	-

**Table 8.1 – Possible sharing operations**

Note that the C + D addition cannot be shared with the K + L addition, even though no set of input values causes both to execute. This assumes that the values of expressions that control ‘if’ statements are unrelated. The same rule applies to ‘case’ statements, as shown in the example below:

```

par
{
    a1 = a2 + a3;
    switch(cond)
    {
        case c1: b1 = b2 + b3;
        case c2: b1 = b4 + b5;
        case c3: b1 = b6 + b7;
        case c4: b1 = b8 + b9;
    }
}

```

Table 8.2 summarizes the possible sharing operations in the above example. A ‘no’

indicates that sharing is not allowed because of the flow of control (execution path) through the circuit. A yes means sharing is allowed.

	a2+a3	b2+b3	b4+b5	b6+b7	b8+b9
a2+a3	-	No	No	No	No
b2+b3	No	-	Yes	Yes	Yes
b4+b5	No	Yes	-	Yes	Yes
b6+b7	No	Yes	Yes	-	Yes
b8+b9	No	Yes	Yes	Yes	-

**Table 8.2 – Possible sharing operations**

Although operations in separate branches of an if statement can be shared, operations in separate branches of a ?: (conditional) construct cannot share the same hardware, even if they are on separate lines.

Consider the following line of code, where expression\_n, represents any expression.

```
z = expression_1 ? expression_2 : expression_3;
```

The Handel-C compiler interprets this code as

```
temp_1 = expression_1;
temp_2 = expression_2;
temp_3 = expression_3;
z = temp_1 ? temp_2 : temp_3;
```

The Handel-C compiler evaluates both expression\_2 and expression\_3, regardless of the value of the conditional. Therefore, operations in expression\_2 cannot share the same resource as operations in expression\_3.

### 8.3 Impact of Sharing on Speed and Area

In order to examine the effect of various degrees of sharing on the speed and area, the following two functions are examined:

- inner product multiplication
- fifth-order elliptic wave filter

They are written in Handel-C for execution on SONIC, making use of the memory interface explained in chapter 3.

Handel-C compiler version 2.1 is used to generate the EDIF and to obtain device independent results. Using the generated EDIF file, Max+plusII is used to obtain device dependent results. In both cases, the multipliers are shared.

Two different ways of sharing the operators are examined: “Adhoc” sharing and “Non-even” sharing.

Let us suppose there are 8 multiplications to be done in a program, and there are 3 shared multipliers available. A program like this is shown below.

```
shared expr mult1(x,y) = x * y;
shared expr mult2(x,y) = x * y;
shared expr mult3(x,y) = x * y;

a1 = a2 * a3;
b1 = a2 * a3;
c1 = a2 * a3;
d1 = a2 * a3;
e1 = a2 * a3;
f1 = a2 * a3;
g1 = a2 * a3;
h1 = h2 * h3;
```

For adhoc sharing you would use the three shared **mult** macros in turn. As shown below.

```
shared expr mult1(x,y) = x * y;
shared expr mult2(x,y) = x * y;
shared expr mult3(x,y) = x * y;

a1 = mult1(a2,a3);
b1 = mult2(b2,b3);
c1 = mult3(c2,c3);
d1 = mult1(d2,d3);
e1 = mult2(e2,e3);
f1 = mult3(f2,f3);
g1 = mult1(g2,g3);
h1 = mult2(h2,h3);
```

However, with non-even sharing we would use **mult2** and **mult3** only once at the end, and use **mult1** for the rest. As shown below.

```

shared expr mult1(x, y) = x * y;
shared expr mult2(x, y) = x * y;
shared expr mult3(x, y) = x * y;

a1 = mult1(a2, a3);
b1 = mult1(b2, b3);
c1 = mult1(c2, c3);
d1 = mult1(d2, d3);
e1 = mult1(e2, e3);
f1 = mult1(f2, f3);
g1 = mult2(g2, g3);
h1 = mult3(h2, h3);
    
```

### 8.4 Inner Product Multiplication

As the simplest example, inner product multiplication is investigated. The inner product of two sequences of two vectors  $\langle X_1, X_2, X_3, \dots, X_n \rangle$  and  $\langle Y_1, Y_2, Y_3, \dots, Y_n \rangle$  is defined to be the sum of products of corresponding elements. More precisely, the inner product is :

$$X_1Y_1 + X_2Y_2 + X_3Y_3 + \dots + X_nY_n$$

Each X and Y element is read from the memory and the result is written back to the memory.

The following 4 cases of inner product multiplication are examined:

Sharing Type	Size of the multiplicands	Vector size
Adhoc	16 bits	8
Non-even	16 bits	8
Adhoc	12 bits	16
Non-even	12 bits	16

In all 4 cases, with no sharing, the FPGA is almost full (about 90% of the logic cells were utilised).

Device independent area is calculated by adding the number of *gates* and *latches* from the output of the Handel-C compiler.

Device dependent area is calculated by adding the number of *logic cells* and *flip flops* from the report file in Max+plus II.

Device dependent speed is obtained using the *timing analyser* in Max+plus II. In order to compare the area and speed with different degrees of sharing, the values for area and speed are normalised.

Figure 8.1 shows the results of all 4 cases of inner product multiplication.

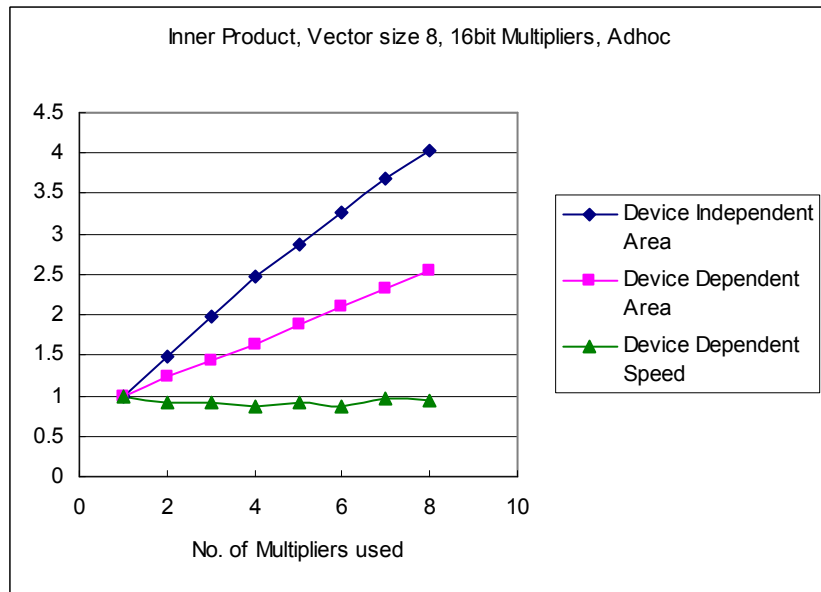


Figure 8.1(a) – Inner Product, 16bit Multipliers, Adhoc

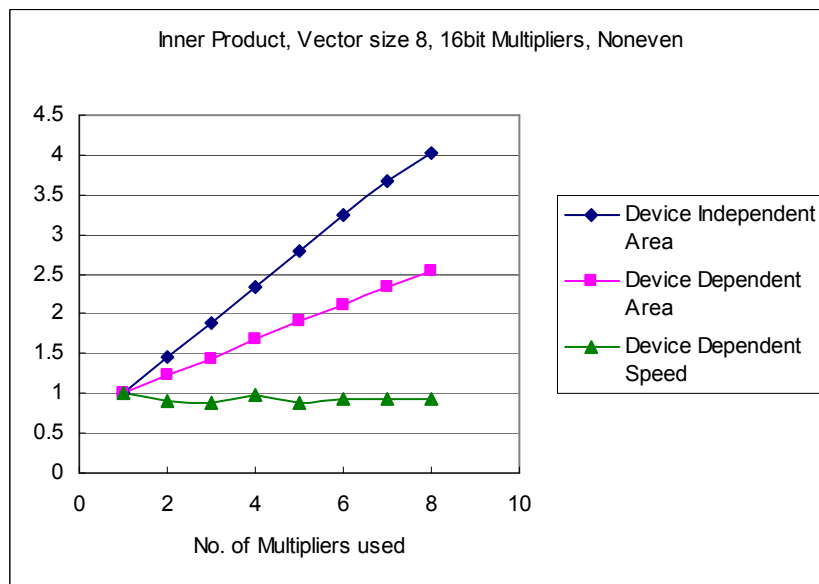


Figure 8.1(b) -- Inner Product, 16bit Multipliers, Noneven

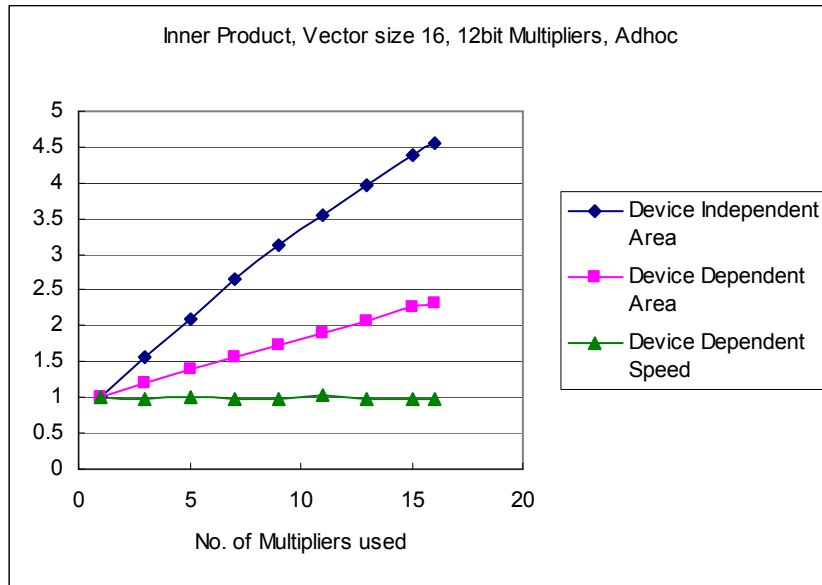


Figure 8.1 (c) – Inner Product, 12bit Multipliers, Adhoc

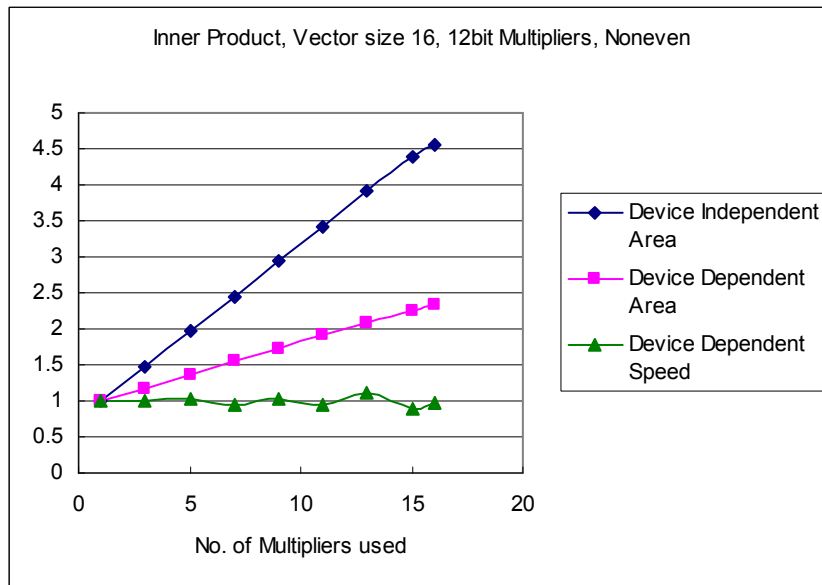


Figure 8.1 (d) – Inner Product, 12bit Multipliers, Noneven

All 4 graphs show that the area increases in a linear manner as more and more multipliers are used (i.e. less multipliers are shared), which is expected since multipliers take up considerable space in FPGAs.

Generally, it is believed that designs with more sharing are slower than designs with less sharing, because of the long routing delays caused by the extra multiplexers.

However, this is not the case. There was no real pattern with the device dependent speed and in some cases the designs with more sharing actually resulted in faster speed. It is suspected that with less sharing, the FPGA is more congested (88% of the LCs are utilised with minimum sharing) causing longer routing delays and thus slower speed. However with more sharing, although there are extra delays because of the multiplexers involved in sharing, the FPGA is less congested (27% of the LCs are utilised with maximum sharing). This is the reason some designs with more sharing performs better than designs with less sharing.

### 8.5 Elliptic Wave Filter

A fifth order elliptic wave filter is implemented as a more complex and real world example.

The following cases are examined:

Sharing Type	Size of the multiplicands
Adhoc	16 bits
Non-even	16 bits

There results are shown in figure 8.3.

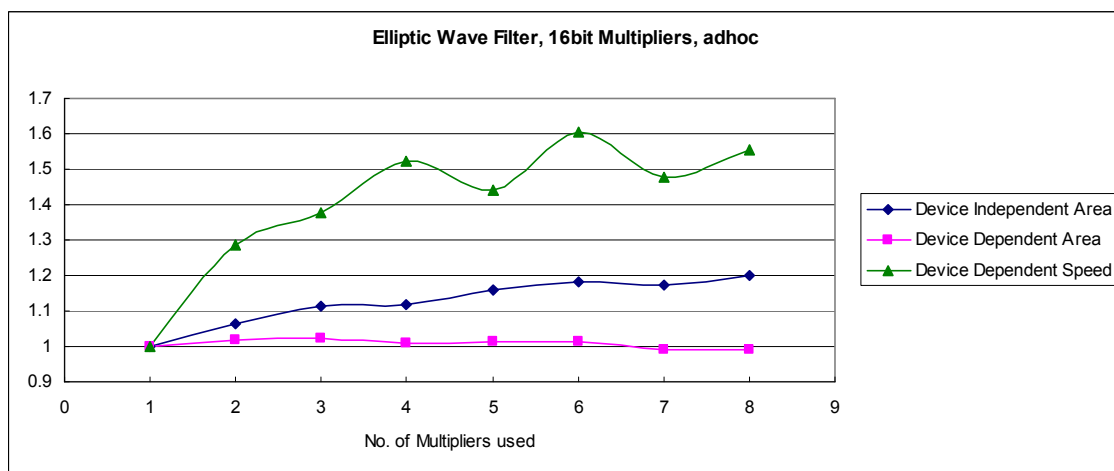


Figure 8.3(a) – Elliptic Wave Filter, 16bit Multipliers, Adhoc

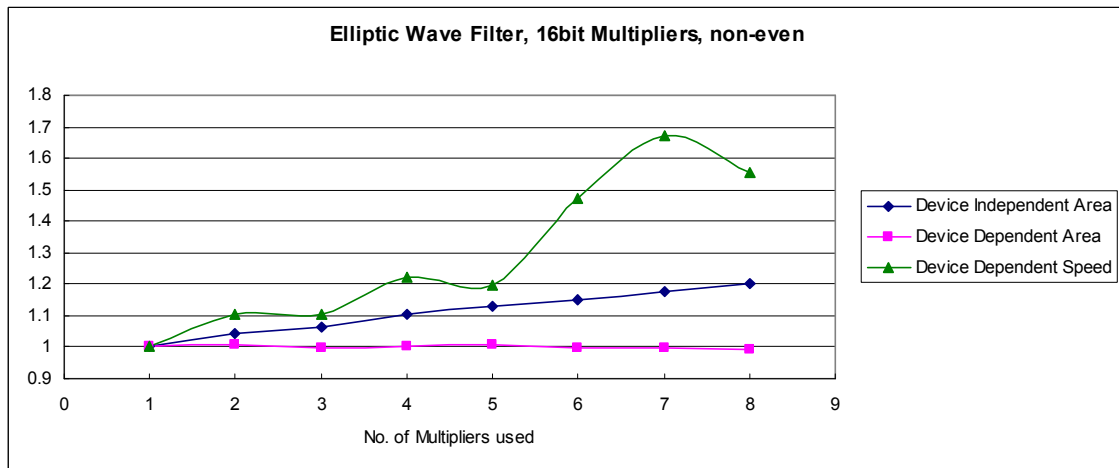


Figure 8.3(b) - Elliptic Wave Filter, 16bit Multipliers, Noneven

Just like the inner product example, the device independent area increases linearly as the degree of sharing is decreased. However, the device dependent area is almost constant. This is very interesting as there is no clear explanation for this kind of trend. It is speculated that possibly, in this particular example, the size of a multipliers is roughly the same as the multipliers and multiplexers required for one shared multiplication.

The trend of the device dependent speed generally increases, but in many cases more sharing results in significantly faster designs. For example, in figure 8.3 (a), using 6 multipliers is approximately 8% faster than using 7 multipliers. The argument from the inner product example is not valid here, since the device dependent area is actually higher with less sharing. It is suspected that possibly the routing congestion in the FPGA is less when multiplexers were used compared to multipliers with the same area. This example demonstrates that there is still a lot of research to be made on sharing operators and their effects.

## 8.6 Automatic Sharing

This part of the chapter discusses how automatic operator sharing could be implemented in Handel-C. By automatic sharing, we mean some software going through the code and sharing the shareable operators appropriately to meet the user's constraints. A pre-processor should identify which operators are shareable by considering the restrictions discussed in chapter 8.2. The constraint given by the user could be one of the following:

- Minimal area.
- Maximum speed.
- Specific area and speed.

There are two ways to meet the user's constraints: estimation using the *device independent* information and using *device dependent* information after "place and routing" onto the FPGA.

For estimation, the number of *gates*, *latches* and *inverters* from the output of the Handel-C compiler is used to estimate the area and speed of the target device. The assumption of more sharing resulting in smaller area and slower speed is used for estimation. In many cases, this assumption is correct and estimation may be good enough to meet the user's requirements.

However, as we have seen in the previous sections, this assumption is not always correct. Especially, when a program is quite complicated like the elliptic wave filter, estimation will not be able to give a satisfactory result. In this case, we have to compile the Handel-C code with the vendor specific tools and "place and route" on the target chip to get the *device dependent* results.

Figure 8.4 shows a data flow diagram illustrating how automatic sharing of Handel-C programs could be implemented. At first, the Handel-C program is passed through a pre-processor, in which all the sharable components are identified. Then the user should choose whether to use estimation or not. If estimation is chosen, the results of the Handel-C compiler is used to meet the user's speed and area constraints. If estimation is not chosen, the device specific compilers are used to meet the user's requirements. It is clear that estimation may be a quick way of sharing the components to meet the constraints, however it will not be very accurate. On the other hand with no estimation, it may take several hours or even days to meet the requirements, but will exactly meet the constraints.

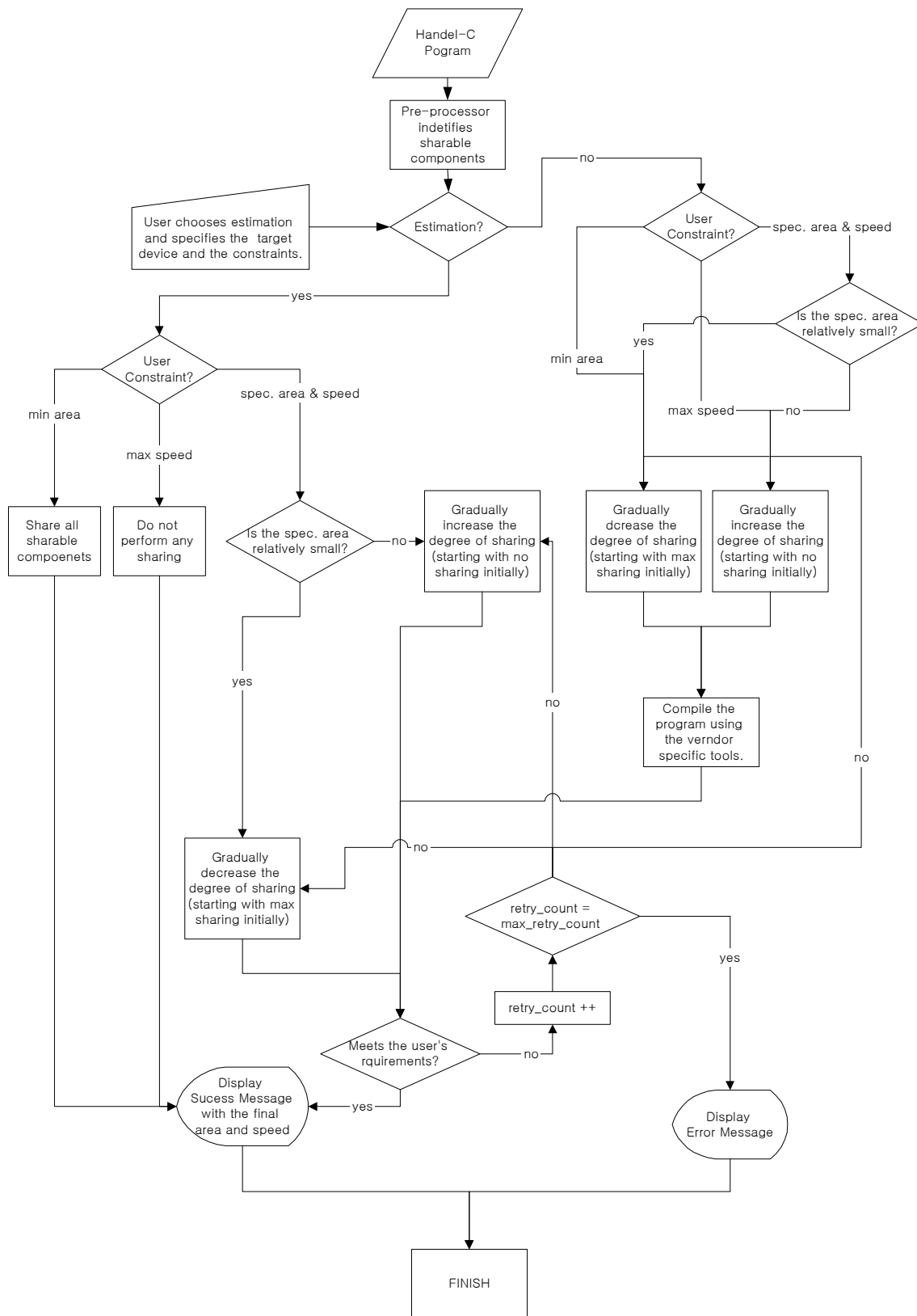


Figure 8.4 – A possible implementation of the automatic sharing of Handel-C programs

## **8.7 Summary**

In this chapter, we have discussed the restrictions of sharing, its impact on speed and area and a possible design to implement automatic sharing for Handel-C programs. Through the two examples, we saw that less sharing does not necessarily mean faster speed. A suggestion of automating sharing for optimal speed and area has also been discussed.

## Chapter 9. Conclusion

This report includes two main parts: hardware compilation and resource sharing optimisations.

The first part, hardware compilation involves implementing compile-time and run-time facilities for reconfigurable engines, where the SONIC system is used as an example. For the mapping of Handel-C programs on SONIC, three interfaces are implemented: a memory interface, a PIPE Bus interface and a PIPE Flow Bus interface. The memory interfacing part is the most challenging, since there are many obstacles involved such as meeting the setup and hold times. These three interfaces are all written into a header file, so that the user could simply include it in the Handel-C program, enabling the generated circuits to communicate with various parts of the SONIC board. A run-time facility is written in C++ with a graphical user interface, for the user to communicate with the Handel-C program running on SONIC. Its main function is to configure SONIC to the user's requirements, load and retrieve information from the PIPE Memory and to synchronise with the Handel-C program. In order to include both software and hardware descriptions in a single framework, Handel-C is extended. The plug construct provided an easy path to partition tasks across multiple processing elements, and the software construct can capture software descriptions.

The second part, resource sharing optimisations, involves demonstrating many well known principles of resource sharing on SONIC making use of EHC. We have seen how space partitioning is achieved simply by splitting the image into 2 halves and processing them using 2 parallel PIPEs. It is clear that we can split the image into more parts, possibly multi-dimensionally for parallel execution on more than 2 PIPEs. Time partitioning is demonstrated with an edge detector, where Gaussian filtering is performed in the first PIPE and Laplacian filtering in the second. In this configuration, the SONIC board has an impressive performance: being 3 times faster than the same implementation in software. The same edge detection technique is performed by using just a single PIPE and reconfiguring the FPGA to perform one of the two filters. Because of the high overhead of reconfiguration (approx. 100ms), this kind of implementation results in significantly slower performance. The idea of pipeline morphing fitted well into the SONIC architecture, because PIPEs were implemented similar to stages of a pipeline.

We have looked at the effects of sharing multipliers on speed and area of the FPGA. The general trend is that the area is increasing as less and less multipliers are shared, which is what we would expect. However, looking at the variations of the speed, the common conception of speed decreasing with more sharing is not always the case. In both inner product and elliptic wave filter examples, we have seen that in some cases the speed is actually faster with more sharing. It is believed that this is caused by the congestion in FPGAs when a lot of multipliers are implemented, causing long routing delays. This demonstrated that in some cases sharing operands can be beneficial in terms of area and speed. Finally, various limitations of sharing in Handel-C were examined. A data flow chart was shown using these limitations, for a possible implementation of automatic sharing of Handel-C programs.

Overall, a framework for programming both software and hardware of reconfigurable engines using a high-level language EHC, which proves to be a powerful compilation and optimisation tool is developed. We have seen that EHC is portable between different reconfigurable engines, using the SONIC and RC1000 system as an example. Using EHC, various graphics and imaging applications have been implemented in the SONIC system. Implementations can be produced more rapidly than those using existing methods, and their performance is often comparable to or faster than previous implementations. The scalable performance and flexibility of reconfigurable engines improve their performance for creating and processing images. This enables graphics designers to produce impressive high-resolution effects in real-time, and doctors to analyse complex medical image scans faster than traditional methods.

There are a number of future extensions that can be done on this project. They are:

- Improving the SonicRun program to support multiple PIPes.
- Extending EHC to cover programmable system-on-a-chip devices.
- Making use of the compile-time facilities by programming some benchmarks and compare the performance using different FPGA boards (such as the Pilchard, an FPGA system interfaced to the host computer through the fast memory bus).
- Implementing automatic sharing for Handel-C applying the ideas suggested.
- Examining the effect of sharing operators using different ways of sharing (e.g. sharing operators that are close in the data flow graph) with different examples.
- Investigating the sharing of blocks of operators instead of just single operators.

## References

1. S.D. Haynes, P. Cheung, W. Luk, John Stone, "Video Image Processing with the SONIC Architecture", IEEE Computer, April 2000, pages 50 - 57.
2. S.D. Haynes, P. Cheung, W. Luk, J. Stone, "SONIC: A Plug-in Architecture for Video Processing", in Field-Programmable Logic and Applications, P. Lysaght, J. Irvine and R.W. Hartenstein (editors), LNCS 1673, Springer, 1999, pages 21-30.
3. S.D. Haynes, "SONIC Board – PIPE Hardware Designers Guide Version 2.00", Imperial College.
4. S.D. Haynes, "SONIC API – Overview", Imperial College.
5. S.D. Haynes, "Design Description Document – 1D FIR Filter", Imperial College.
6. S.D. Haynes, "Design Description Document - Convert", Imperial College.
7. S.D. Haynes, "Design Description Document - Transform", Imperial College.
8. S.D. Haynes, "Registers for the PIPE Router (PR) Version 2.0", Imperial College.
9. S.D. Haynes, "Registers for the PIPE Router (PR) Version 2.0 & 2.1", Imperial College.
10. S.D. Haynes, "Image Merge Tutorial", Imperial College.
11. S.D. Haynes, "Registers for the PIPE Router (PR) Version 2.2", Imperial College.
12. "Sonic Homepage", Imperial College, URL: <http://infoeng.ee.ic.ac.uk/Sonic/>, Last Checked: June 2001.
13. "Virtual Computer Corporation FPGA Web-Page", Virtual Computer Corporation, URL: <http://www.vcc.com>, Last Checked: February 2001.
14. Babb, et al, "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators", Proc. FCCM '93, April 1993.
15. W. H. Mangione-Smith, et al, "Seeking solutions in configurable computing. IEEE Computer Magazine", pages 38-43, December 1997.
16. "Handel-C Language Reference Manual", Embedded Solutions.
17. "Handel-C Installation Guide and Compiler and Simulator Reference Manual", Embedded Solutions.
18. P. Dewilde, E. Deprettere, R. Nouta, "Parallel and pipe-lined VLSI implementations of signal processing algorithms", S.Y. Kung, H.J. Whitehouse, T. Kailath, eds., VLSI and Modern Signal Processing, Englewood Cliffs, NJ: Prentice-Hall, Chap. 15, pp. 257-275, 1985.
19. M. Leeser, R. Chapman, M Aagaard, M. Linderman, S. Meier, "High Level Synthesis and Generating FPGAs with the BEDROC System", Journal of VLSI Signal Processing, Vol 6, pp. 191-214, 1993.
20. Shay Ping Seng, "AVI Test Bench 1.0 Web-Page", URL: [http://www.doc.ic.ac.uk/~sps/esl/handelresources/aviTB\\_documentation.html](http://www.doc.ic.ac.uk/~sps/esl/handelresources/aviTB_documentation.html), Last Checked: June 2001.
21. "Reconfigurable Hardware Development Platform – RC1000", Celoxica, URL: <http://www.celoxica.com/>, Last Checked: September 2001.
22. P.I. Mackinlay, P. Cheung, W. Luk, R. Sandiford, "Riley-2: A Flexible Platform for Codesign and Dynamic Reconfigurable Computing Research", Imperial College.
23. R.C. Seals, G.F. Whapshott, "Programmable Logic: PLDs and FPGAs", School of Engineering, University of Greenwich, Antony Row Ltd, 1997.

24. W. Luk, “*Custom Computing Lecture Notes*”, Imperial College, 2000.
25. I. Page, W. Luk, “*Compiling occam into FPGAs*”, in FPGAs, W. Moore, W. Luk, Abingdon EE&CS Books, 1991, pp. 271-283.
26. W. Luk, N. Shirazi, S.R. Guo, P. Cheung, “*Pipeline Morphing and Virtual Pipelines*”, Imperial College.
27. H. Styles, W. Luk, “*Customising Graphics Applications: Techniques and Programming Interface*”, Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society Press, 2000.
28. W. Luk, T. Wu, I. Page, “*Hardware-Software Codesign of Multidimensional Programs*”, Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society Press, 1994.
29. M. Weinhardt, W. Luk, “*Task-Parallel Programming of Reconfigurable Systems*”, Field Programmable Logic and Applications, LNCS 2147, Springer, 2001.
30. F. Faggin, “The Future of Microprocessors”, ASAP Forbes, <http://www.forbes.com/asap>, 1996, Last checked: June 2001.
31. D. Buell, J. Arnold, W. Kleinfelder, “*Splash 2: FPGA in a Custom Computing Machine*”, IEEE Computer Society Press, 10662 Los Vasqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1264, 1996.
32. “*ANTLR Website*”, URL: <http://www.antlr.org>, Last Checked: September 2001.
33. J.M. Saul, “*Hardware/Software Codesign for FPGA-Based Systems*”, Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences, 1998.
34. S. Bakshi, “*Partitioning and Pipelining Performance-Constrained Hardware/Software systems*”, IEEE Transactions on VLSI systems, Vol 7, No 4, 1999

## Appendix A

The following PE pins are used by the PIPE. In order to work correctly, all the signals must be locked to the appropriate pines on the Flex 10K70 device.

Signal	Pin	Purpose	Signal	Pin	Purpose	
AD0	127	PIPE Bus Data/Address	ADDRESS1	35		
AD1	14		ADDRESS2	41		
AD2	129		ADDRESS3	144		
AD3	13		ADDRESS4	39		
AD4	15		ADDRESS5	139		
AD5	168		ADDRESS6	151		
AD6	167		ADDRESS7	88		
AD7	134		ADDRESS8	207		
AD8	166		ADDRESS9	23		
AD9	169		ADDRESS10	218		
AD10	159		ADDRESS11	146		
AD11	158		ADDRESS12	80		
AD12	157		ADDRESS13	147		
AD13	156		ADDRESS14	142		
AD14	28		ADDRESS15	143		
AD15	26		ADDRESS16	149		
AD16	25		ADDRESS17	141		
AD17	29		ADDRESS18	138		
AD18	190		ADDRESS19	18		
AD19	108	SRAM data	DATA0	173		
AD20	187		DATA1	174		
AD21	111		DATA2	172		
AD22	110		DATA3	208		
AD23	182		DATA4	181		
AD24	117		DATA5	97		
AD25	184		DATA6	201		
AD26	164		DATA7	171		
AD27	185		DATA8	175		
AD28	186		DATA9	8		
AD29	199		DATA10	116		
AD30	161		DATA11	9		
AD31	188		DATA12	148		
STALL	105		PIPE Control	DATA13	119	
TERM	212	DATA14		99		
WRITE	106		DATA15	6		
AS	210		DATA16	183		
CS	92	PE Select	DATA17	209		
CSM	90	Memory Select	DATA18	101		
PFIN0	217	PIPEFlow Bus (IN)	DATA19	204		
PFIN1	219		DATA20	196		
PFIN2	83		DATA21	103		
PFIN3	203		DATA22	100		
PFIN4	206		DATA23	94		
PFIN5	71		DATA24	95		
PFIN6	131		DATA25	107		
PFIN7	50		DATA26	7		
PFIN8	21		DATA27	193		
PFIN9	163		DATA28	195		
PFIN10	126		DATA29	98		
PFIN11	51		DATA30	113		
PFIN12	56		DATA31	11		
PFIN13	132	Connect Right Bus	CR0	31		
PFIN14	133		CR1	238		
PFIN15	49		CR2	17		
PFIN_ENDL	46		CR3	63		
PFIN_INST	48		CR4	67		
PFIN_ENDS	20		CR5	33		
PFOUT0	53		PIPEFlow Bus (OUT)	CR6	78	
PFOUT1	54			CR7	45	
PFOUT2	55	CR8		36		

PFOUT3	136		CR9	44	
PFOUT4	137		CR10	237	
PFOUT5	84		CR11	68	
PFOUT6	72		CR12	235	
PFOUT7	213		CR13	40	
PFOUT8	19		CR14	12	
PFOUT9	162		CR15	70	
PFOUT10	82		CR16	223	
PFOUT11	81		CR17	79	
PFOUT12	128		CR18	222	
PFOUT13	200		CR19	34	
PFOUT14	202		CR20	221	
PFOUT15	86		CR21	234	
PFOUT_ENDL	214		CL0	115	Connect Left Bus
PFOUT_INST	215		CL1	191	
PFOUT_ENDS	24		CL2	73	
CLK (GCLK1)	91	PLL Connections	CL3	74	
GCLK2	211		CL4	228	
GCLK2_OUT	38		CL5	233	
PLL_OUT	30		CL6	231	
PLL_FB	43		CL7	227	
PIPE_INT	153	Interrupt Pin	CL8	109	
M_NW	152	SRAM Write	CL9	230	
M_NS	194	SRAM Select	CL10	229	
ADDRESS0	154	SRAM Address	CL11	226	
			CL12	76	
			CL13	75	
			CL14	225	
			CL15	118	
			CL16	192	
			CL17	114	
			CL18	87	
			CL19	102	
			CL20	220	
			CL21	120	

## Appendix B

This is the header file that needs to be included with every Handel-C programs for execution on SONIC. It has external interface declarations and functions for memory accesses.

```

set family = Altera10K;
set part = "EPF10K70RC240-3";
set clock = external"91";

unsigned int 1 m_data_ena;
unsigned int 1 m_addr_ena;
unsigned int 1 ad_ena;
unsigned int 32 ad_out;
unsigned int 32 m_data_out;
unsigned int 20 m_addr_out;
unsigned int 1 m_ns_out;
unsigned int 1 m_nw_out;
unsigned int 1 m_ns_ena;
unsigned int 1 m_nw_ena;
unsigned int 1 stall_out;
unsigned int 1 stall_ena;
unsigned int 1 pfout_ena;
unsigned int 1 pfout_inst_ena;
unsigned int 1 pfout_ends_ena;
unsigned int 1 pfout_endl_ena;
unsigned int 1 cr_ena;
unsigned int 1 cl_ena;
unsigned int 16 pfout_out;
unsigned int 1 pfout_inst_out;
unsigned int 1 pfout_ends_out;
unsigned int 1 pfout_endl_out;
unsigned int 22 cr_out;
unsigned int 22 cl_out;

interface bus_ts_clock_in (unsigned int 32) m_data (m_data_out, m_data_ena == 1)
    with { data = {"171","201","97","181",
                  "208","172","174","173",
                  "6","99","119","148",
                  "9","116","8","175",
                  "94","100","103","196",
                  "204","101","209","183",
                  "11","113","98","195",
                  "193","7","107","95"}};

interface bus_ts_clock_in (unsigned int 20) m_addr (m_addr_out, m_addr_ena == 1)
    with { data = {"18","138","141","149",
                  "143","142","147","80",
                  "146","218","23","207",
                  "88","151","139","39",
                  "144","41","35","154"}};

interface bus_ts_clock_in (unsigned int 1) m_ns (m_ns_out,m_ns_ena == 1) with { data =
{"194"}};

interface bus_ts_clock_in (unsigned int 1) m_nw (m_nw_out,m_nw_ena == 1) with { data =
{"152"}};

interface bus_ts_clock_in(unsigned int 1) stall(stall_out,stall_ena == 1) with { data =
{"105"}};

interface bus_in(unsigned int 1) term() with { data = {"212"}};

```

```

interface bus_ts_clock_in(unsigned int 32) ad(ad_out, ad_ena == 1)
  with { data = { "188","161","199","186",
                  "185","164","184","117",
                  "182","110","111","187",
                  "108","190","29","25",
                  "26","28","156","157",
                  "158","159","169","166",
                  "134","167","168","15",
                  "13","129","14","127"};};

interface bus_in(unsigned int 1) ad_write() with { data = {"106"};};

interface bus_in(unsigned int 1) ad_cs() with { data = {"92"};};

interface bus_in(unsigned int 1) ad_as() with { data = {"210"};};

interface bus_in(unsigned int 1) csm() with { data = {"90"};};

interface bus_clock_in(unsigned int 16) pfin ()
  with { data = {
              "50","131","71","206",
              "203","83","219","217",
              "49","133","132","56",
              "51","126","163","21"};};
interface bus_clock_in(unsigned int 1) pfin_inst() with { data = {"48"};};

interface bus_clock_in(unsigned int 1) pfin_ends() with { data = {"20"};};
interface bus_clock_in(unsigned int 1) pfin_endl() with { data = {"46"};};

interface bus_ts_clock_in(unsigned int 16) pfout (pfout_out, pfout_ena == 1)
  with { data = {
              "213","72","84","137",
              "136","55","54","53",
              "86","202","200","128",
              "81","82","162","19"};};

interface bus_ts_clock_in(unsigned int 1) pfout_inst(pfout_inst_out,pfout_inst_ena == 1)
with { data = {"215"};};
interface bus_ts_clock_in(unsigned int 1) pfout_ends(pfout_ends_out,pfout_ends_ena == 1)
with { data = {"24"};};
interface bus_ts_clock_in(unsigned int 1) pfout_endl(pfout_endl_out,pfout_endl_ena == 1)
with { data = {"214"};};

interface bus_ts_clock_in(unsigned int 22) cr (cr_out, cr_ena == 1)
  with { data = {"234","221","34","222",
                "79","223","70","12",
                "40","235","68","237",
                "44","36","45","78",
                "33","67","63","17",
                "238","31"};};

interface bus_ts_clock_in(unsigned int 22) cl (cl_out, cl_ena == 1)
  with { data = {"120","220","102","87",
                "114","192","118","225",
                "75","76","226","229",
                "230","109","227","231",
                "233","228","74","73",
                "191","115"};};

macro proc read(address, value)
{
  par // read 1
  {
    m_nw_out = 1;
    m_ns_out = 0;
    m_addr_out = address;
    m_addr_ena = 1;
    m_ns_ena = 1;
    m_nw_ena = 1;
    m_data_ena = 0;
  }
}

```

```
    par // read 2
    {
        delay;
    }
    par // read 3
    {
        delay;
    }

    par // read 4
    {
        m_ns_out = 1;
        value = m_data.in;
        m_addr_ena = 0;
        m_nw_ena = 0;
        m_ns_ena = 0;
    }
}

macro proc write(address, value)
{
    par // write 1
    {
        m_nw_out = 1;
        m_ns_out = 1;
        m_data_out = value;
        m_addr_out = address;
        m_addr_ena = 1;
        m_data_ena = 1;
        m_ns_ena = 1;
        m_nw_ena = 1;
    }

    par // write 2
    {
        m_nw_out = 0;
        m_ns_out = 0;
    }

    par // write 3
    {
        m_data_ena = 0;
        m_addr_ena = 0;
        m_nw_ena = 0;
        m_ns_ena = 0;
        m_nw_out = 1;
        m_ns_out = 1;
    }
}
```

## Appendix C

Source code of the colour inverter in Handel-C versions and AHDL (written by Simon Haynes) version.

### Handel-C (using memory interface)

```
#include "mem.h"
#define WIDTH 32

void main(void) {

unsigned int 32 pixel,data;
unsigned int 8 red, green, blue, alpha;
unsigned int 20 i;

par {
    stall_out = 0;
    stall_ena = 0;
    m_data_ena = 0;
    m_addr_ena = 0;
    m_ns_ena = 0;
    m_nw_ena = 0;
    ad_ena = 0;
    ad_out = 0;
}
par {
do {
    ad_ena = ad_cs.in & !ad_write.in;
} while (1);

do {
    if (ad_cs.in & ad_write.in & ad.in[9] == 1)
    {
        // Start Main Program
        for (i=10; i[12] == 0; i++) {
            read(i,pixel);
            par {
                red = 255-pixel[31:24];
                green = 255-pixel[23:16];
                blue = 255-pixel[15:8];
                alpha = 255-pixel[7:0];
            }
            write(i,red@green@blue@alpha);
        }
        // End Main Program
        do {
            if (ad_cs.in & !ad_write.in & ad.in[9] == 1)
                ad_out = 0x1;
            else delay;
        } while (ad_out[0] != 1);
    }
    else delay;
} while (1);
}
```

## Handel-C (using PIPEFlow Bus)

```
#include "mem.h"

void main(void) {

    unsigned int 1 h_pfin_inst;
    unsigned int 1 h_pfin_endl;
    unsigned int 1 h_pfin_ends;
    unsigned int 16 h_pfin;

    unsigned int 1 p_pfout_inst;
    unsigned int 1 p_pfout_endl;
    unsigned int 1 p_pfout_ends;
    unsigned int 8 p_pfout_1;
    unsigned int 8 p_pfout_2;

    par {
        pfout_inst_ena = 1;
        pfout_endl_ena = 1;
        pfout_ends_ena = 1;
        pfout_ena = 1;
    }

    do {
        par {
            h_pfin_inst = pfin_inst.in;
            h_pfin_endl = pfin_endl.in;
            h_pfin_ends = pfin_ends.in;
            h_pfin      = pfin.in;

            p_pfout_inst = h_pfin_inst;
            p_pfout_endl = h_pfin_endl;
            p_pfout_ends = h_pfin_ends;

            p_pfout_1 = h_pfin_inst ? h_pfin[7:0] : 255-h_pfin[7:0];
            p_pfout_2 = h_pfin_inst ? h_pfin[15:8] : 255-h_pfin[15:8];

            pfout_inst_out = p_pfout_inst;
            pfout_endl_out = p_pfout_endl;
            pfout_ends_out = p_pfout_ends;
            pfout_out      = p_pfout_2@p_pfout_1;
        } while (1);
    }
}
```

## AHDL

```

SUBDESIGN pe_invert
(
  --PIPEFFlow IN
  pfin_inst : INPUT;
  pfin_endl : INPUT;
  pfin_ends : INPUT;
  pfin[15..0] : INPUT;
  --PIPEFFlow OUT
  pfout_inst : OUTPUT;
  pfout_endl : OUTPUT;
  pfout_ends : OUTPUT;
  pfout[15..0] : OUTPUT;
  %Clk %
  clk : INPUT;
)

VARIABLE
h_pfin_inst : dff; --Registered pfin_inst signal
h_pfin_endl : dff; --Registered pfin_endl signal
h_pfin_ends : dff; --Registered pfin_ends signal
h_pfin[15..0] : dff; --Registered pfin data
p_pfout_inst : dff; --pre-registered pfout_inst signal
p_pfout_endl : dff; --pre-registered pfout_endl signal
p_pfout_ends : dff; --pre-registered pfout_ends signal
p_pfout[15..0] : dff; --pre-registered pfout data
red_green : dff; --High when we are in the red/green cycle

BEGIN
--Register the incoming PIPEFFlow bus
h_pfin_inst.clk=clk; h_pfin_inst=dfin_inst;
h_pfin_endl.clk=clk; h_pfin_endl=dfin_endl;
h_pfin_ends.clk=clk; h_pfin_ends=dfin_ends;
h_pfin[].clk=clk; h_pfin[]=dfin[];

--Set the red/green cycle
red_green.clk=clk; red_green=pfin_inst # !red_green;

--Output register clocks
p_pfout_inst.clk=clk;
p_pfout_ends.clk=clk;
p_pfout_endl.clk=clk;
p_pfout[].clk=clk;

--Output register values
p_pfout_inst=h_pfin_inst;
p_pfout_endl=h_pfin_endl;
p_pfout_ends=h_pfin_ends;

--Only invert green not alpha, and pass data straight through if it is an instruction
--(inst=HIGH)
p_pfout[7..0]=(255-h_pfin[7..0]) & !(red_green # h_pfin_inst) #
h_pfin[7..0] & (red_green # h_pfin_inst);
p_pfout[15..8] = (255-h_pfin[15..8]) & !h_pfin_inst #
h_pfin[15..8] & h_pfin_inst;

--Set the block outputs;
pfout_inst=p_pfout_inst;
pfout_ends=p_pfout_ends;
pfout_endl=p_pfout_endl;
pfout[]=p_pfout[];

END;

```

## Appendix D

This user guide will show the steps involved to create an .rbf from a Handel-C file.

### Compiling the Handel-C program

Use the following command to create an EDIF file:

```
handelc -edif invert.c
```

You may want to specify the **-O** option to turn on maximum optimisations, or the **-lpm** option to tell the compiler to use LPMs for operations greater than a certain width. For example:

```
handelc -edif -O -lpm 8 invert.c
```

This tells the compiler to use LPMs on operations with data path wider than 8 bits. LPMs may help in reducing logic usage or improving the timing characteristics of the design. Refer to the Altera documentation for further details of LPM macros.

### Compiling the EDIF file in Max+PlusII

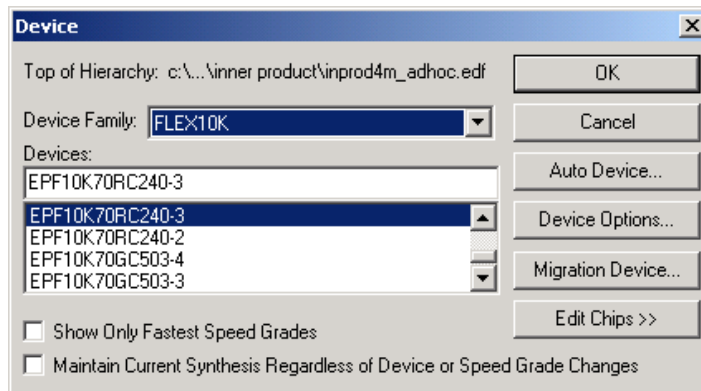
#### *Opening the file and setting it to the current project*

Open the EDIF file and set it to the current project by using the following menu command:

File → Project → Set Project to Current File

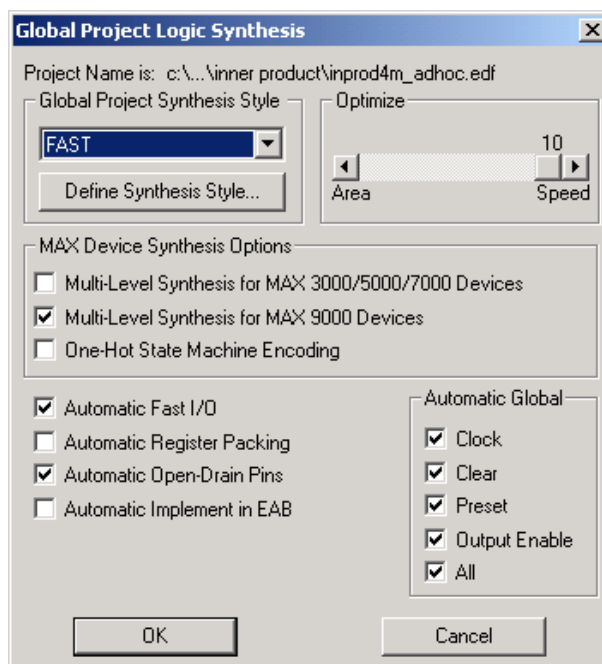
### Choosing the Correct Device

The next step is target the correct device. The PE is a 10K70RC240-3. This can be done using the device selection dialog box shown below. Use the following menu command:  
Assign → Device



### Setting Compiler Options

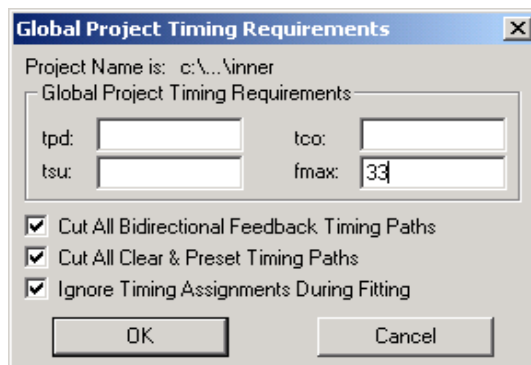
Bring up the Global Project Logic Synthesis dialog as shown below by using the following menu command:  
Assign → Global Project Logic Options



Set the Global Project Synthesis Style to 'FAST', which will use the 10K carry chain logic (amongst other things) which gives improved design speed and generally smaller designs. If your design is too slow or does not fit, you can try to change the various options in this dialog to improve the situation.

Next, bring up the Global Project Timing Requirements dialog as shown below by using the following menu command:

Assign → Global Project Timing Requirements



Set fmax which specifies the minimum acceptable clock frequency to 33, since the clock frequency the SONIC board uses is 33Mhz.

*Setting the LMF (library mapping file)*

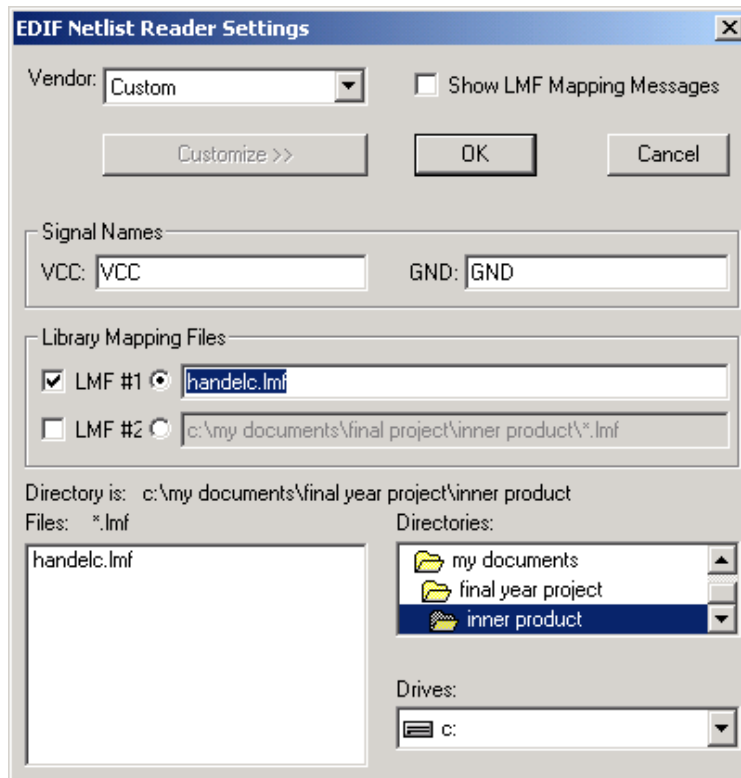
This step is necessary to tell the Altera tools how the Handel-C gate library maps to the Altera primitives.

First, bring up the Compiler window using the following menu command:

MAX+plus II → Compiler

Then the following to bring up the EDIF Netlist Reader Settings dialogue shown below.

Interface Menu → EDIF Netlist Reader Settings

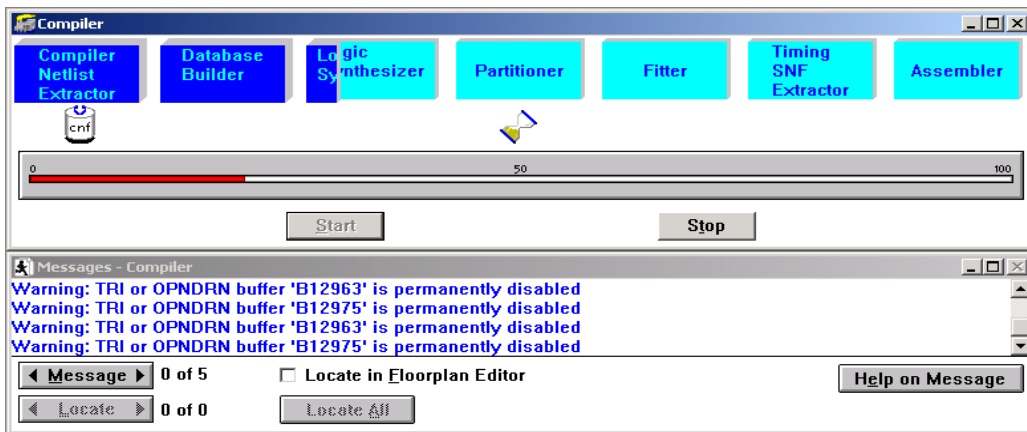


Set custom as the Vendor and hit the 'Customize' button. Type full pathname of the lmf\handelc.lmf file in the LMF #1 box

Select VCC and GND as the power symbols and click on OK.

### Compilation

Hit the Start button, and the compilation will begin as shown below. This will generate the .sof (SRAM Object File) for this project.

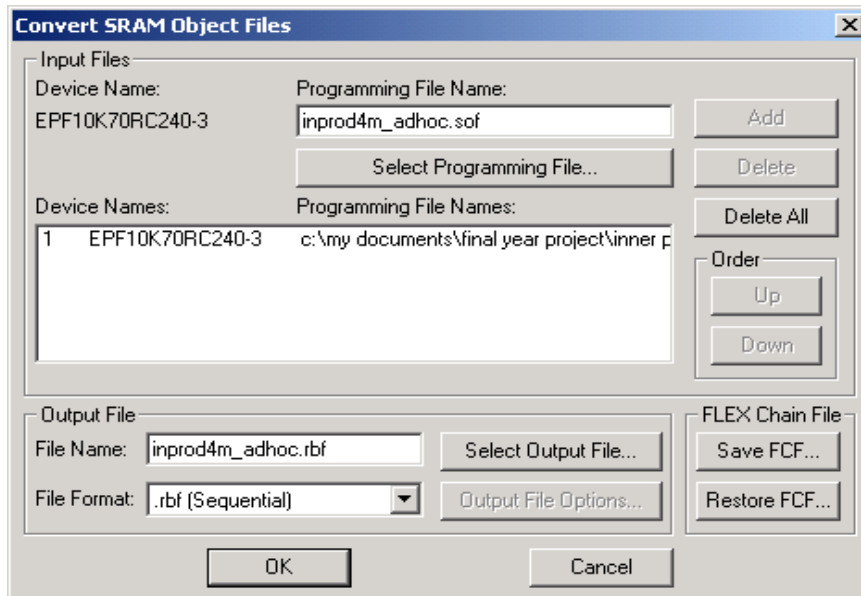


If you get a warning message indicating that your design cannot run at 33Mhz, you will have to optimize you Handel-C program in order to achieve a faster clock frequency.

### Conversion From .sof File to .rbf File

The SONIC platform requires the configuration data in .rbf format (the files are more compact this way). This conversion is simply done using the SRAM Object File Converter.

Bring up the dialog box shown below using the following menu command:  
File → Convert SRAM Object Files



Select the correct .sof file and click the 'Add' button. Ensure that the correct device is displayed (EPF10K70RC240-3). Select ".rbf (Sequential)" File Format and then enter the file name. Click on OK to generate the .rbf file.

## Appendix E

- **The Handel-C Code for edge detection using just one PIPE and the software construct:**

```
#include "mem.h"

macro proc mygauss() {
  unsigned int 1 inst_0, inst_1, endl_0, endl_1, ends_0, ends_1, red_green;
  unsigned int 16 data_0, data_1, data_2, data_3;
  unsigned int 8 green;

  ////////////////DO NOT EDIT//////////////////
  par {
    pfout_inst_ena = 1;
    pfout_endl_ena = 1;
    pfout_ends_ena = 1;
    pfout_ena = 1;
  }
  ///////////////////////////////////////////////////////////////////

  do {
    par {
      inst_0 = pfin_inst.in;
      inst_1 = inst_0;
      endl_0 = pfin_endl.in;
      endl_1 = endl_0;
      ends_0 = pfin_ends.in;
      ends_1 = ends_0;
      data_0 = pfin.in;
      data_1 = data_0;
      data_2 = data_1;
      data_3 = data_2;
      red_green = pfin_inst.in | !red_green;

      if (!inst_0) {
        if (red_green)
          par {
            pfout_out = (((unsigned int
10) (0@pfin.in[7:0])
+ ((unsigned int 10) ((0@data_1[7:0]) << 1)) +
(unsigned int 10) (0@data_3[7:0])) >> 2)) [7:0] @ (((unsigned int 10) (0@pfin.in[7:0])
+ ((unsigned int 10) ((0@data_1[7:0]) << 1)) +
(unsigned int 10) (0@data_3[7:0])) >> 2)) [7:0];

            green = (((unsigned int
10) (0@pfin.in[7:0])
+ ((unsigned int 10) ((0@data_1[7:0]) << 1)) +
(unsigned int 10) (0@data_3[7:0])) >> 2)) [7:0];
          }
        else
          pfout_out = green@data_1[7:0];
        }
      else
        pfout_out = data_1;
      pfout_inst_out = inst_1;
      pfout_endl_out = endl_1;
      pfout_ends_out = ends_1;
    }
  } while (1);
}

macro proc myedge() {
  unsigned int 1 inst_0, inst_1, endl_0, endl_1, ends_0, ends_1, ends_2, endl_2,
inst_2, red_green;

```

```

unsigned int 16 data_0, data_1, data_2, data_3, temp_data, pfout_temp;
unsigned int 8 green, temp_green;
unsigned int 1 temp;

//////////DO NOT EDIT//////////
par {
    pfout_inst_ena = 1;
    pfout_endl_ena = 1;
    pfout_ends_ena = 1;
    pfout_ena = 1;
}
//////////DO NOT EDIT//////////

do {
    par {
        inst_0 = pfin_inst.in;
        inst_1 = inst_0;
        inst_2 = inst_1;
        endl_0 = pfin_endl.in;
        endl_1 = endl_0;
        endl_2 = endl_1;
        ends_0 = pfin_ends.in;
        ends_1 = ends_0;
        ends_2 = ends_1;
        data_0 = pfin.in;
        data_1 = data_0;
        data_2 = data_1;
        data_3 = data_2;
        red_green = pfin_inst.in | !red_green;

        if (!inst_0) {
            if (red_green)
                par {
                    temp = (unsigned int 1) (((int
10) (0@(data_1[7:0])) << 1) - ((int 10) (0@pfin.in[7:0]))
                    - ((int 10) (0@data_3[7:0]))) [9];
                    green = ((unsigned int 10) ((int
10) (0@(data_1[7:0])) << 1) - ((int 10) (0@pfin.in[7:0]))
                    - ((int 10) (0@data_3[7:0]))) << -8) << 4;

                    if (temp)
                        par {
                            pfout_out = 0;
                            temp_green = 0;
                        }
                    else
                        par {
                            pfout_out = green@green;
                            temp_green = green;
                        }
                }
            else
                par {
                    pfout_temp = temp_green@temp_green;
                    pfout_out = temp_green@temp_green;
                }
        }
        else
            par {
                temp_data = data_1;
                pfout_out = temp_data;
            }
        pfout_inst_out = inst_2;
        pfout_endl_out = endl_2;
        pfout_ends_out = ends_2;
    }
} while (1);
}

macro proc invert() {
    unsigned int 1 h_pfin_inst;

```

```

unsigned int 1 h_pfin_endl;
unsigned int 1 h_pfin_ends;
unsigned int 16 h_pfin;

unsigned int 1 red_green;

unsigned int 1 p_pfout_inst;
unsigned int 1 p_pfout_endl;
unsigned int 1 p_pfout_ends;
unsigned int 8 p_pfout_1;
unsigned int 8 p_pfout_2;

//////////DO NOT EDIT//////////
par {
    pfout_inst_ena = 1;
    pfout_endl_ena = 1;
    pfout_ends_ena = 1;
    pfout_ena = 1;
    red_green = 0;
}
//////////DO NOT EDIT//////////

do {
    par {
        h_pfin_inst = pfin_inst.in;
        h_pfin_endl = pfin_endl.in;
        h_pfin_ends = pfin_ends.in;
        h_pfin = pfin.in;

        red_green = pfin_inst.in | !red_green;

        p_pfout_inst = h_pfin_inst;
        p_pfout_endl = h_pfin_endl;
        p_pfout_ends = h_pfin_ends;

        p_pfout_1 = (!red_green || h_pfin_inst) ? h_pfin[7:0] : (255-
h_pfin[7:0]);
        p_pfout_2 = h_pfin_inst ? h_pfin[15:8] : (255-h_pfin[15:8]);

        pfout_inst_out = p_pfout_inst;
        pfout_endl_out = p_pfout_endl;
        pfout_ends_out = p_pfout_ends;
        pfout_out = p_pfout_2 @ p_pfout_1;
    }
} while (1);
}

void main() {
    software {
        sonic_init(); //initialise sonic
        sonic_pr_route(0,0x3F); //setup the routing
        sonic_pe_conf(0,plug1); //confiure for gaussian
        sonic_pm_write(0,"uhm.bmp",0); //load image
        sonic_pr_mode(0,0,0,0,0,0); //horizontal scan
        sonic_pf_start(0); //horizontall gaussian pass
        sonic_pr_mode(0,0,1,0,1,0); //vertical scan
        sonic_pf_start(0); //vertical gaussian pass
        sonic_pe_conf(0,plug2); //reconfiguration to lapacian
        sonic_pf_start(0); //vertical lapacian
        sonic_pr_mode(0,0,0,0,0,0); //horizontal scan
        sonic_pf_start(0); // horizontal lapacian
        sonic_pm_read(0,"result.bmp",0); //store imamge
        sonic_close();
    }

    plug1 {
        mygauss();
    }

    plug2 {
        myedge();
    }
}

```

```
    plug3 {
        invert();
    }
}
```

- **The C code generated by the pre-processor:**

```
// HwSwDlg.cpp : implementation file
//

#include "stdafx.h"
#include "SonicRun.h"
#include "HwSwDlg.h"
#include "SonicAPI.h"
#include "message.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
SONIC_RETURN_CODE Code;

////////////////////////////////////
// HwSwDlg dialog

HwSwDlg::HwSwDlg(CWnd* pParent /*=NULL*/)
    : CDialog(HwSwDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(HwSwDlg)
    // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void HwSwDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(HwSwDlg)
    // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(HwSwDlg, CDialog)
    //{{AFX_MSG_MAP(HwSwDlg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// HwSwDlg message handlers

void HwSwDlg::OnOK()
{
    UINT i=0;
    int x=0;
    int y=0;
    int CF;
```

```
CFile File, _File;
CFileException Error;
BITMAPFILEHEADER bmfh;
BITMAPINFOHEADER bmih;
CSize Size;
BYTE Temp;
int remainder;
DWORD Data;
BOOL RCol;
BOOL RUpper;
BOOL WCol;
BOOL WUpper;
int Strip;
BYTE* m_pData;
CArchive stream(&File, CArchive::load);
CArchive _stream(&File, CArchive::store);
int Iwidth;
int Iheight;
ULONG m_lBytes;
m_pData = NULL;

do Sonic_Initialise(); while (Code==SONIC_API_FAILED);
do Sonic_Conf_PR (0,"pr2_2.rbf"); while (Code==SONIC_API_FAILED);
do Sonic_Conf_PR (1,"pr2_2.rbf"); while (Code==SONIC_API_FAILED);
Data = 0x3F;
do Sonic_PR_Route_Write (0, &Data); while (Code==SONIC_API_FAILED);
do Sonic_Conf_PE (0,"plug1.rbf"); while (Code==SONIC_API_FAILED);
do Sonic_PM_Read(0, (unsigned char *)m_pData, m_lBytes, 0); while
(Code==SONIC_API_FAILED);
if (!File.Open( "src.bmp", CFile::modeCreate | CFile::modeWrite , &Error))
return;

_stream << bmfh.bfType;
_stream << bmfh.bfSize;
_stream << bmfh.bfReserved1;
_stream << bmfh.bfReserved2;
_stream << bmfh.bfOffBits;
_stream << bmih.biSize;
_stream << bmih.biWidth;
_stream << bmih.biHeight;
_stream << bmih.biPlanes;
_stream << bmih.biBitCount;
_stream << bmih.biCompression;
_stream << bmih.biSizeImage;
_stream << bmih.biXPelsPerMeter;
_stream << bmih.biYPelsPerMeter;
_stream << bmih.biClrUsed;
_stream << bmih.biClrImportant;
for( y=0;y<(int)bmih.biHeight;y++)
{
    for(x=0;x<(int)bmih.biWidth;x++)
    {
        _stream<<m_pData[((y)*bmih.biWidth+x)*4]; //B
        _stream<<m_pData[((y)*bmih.biWidth+x)*4+1]; //G
        _stream<<m_pData[((y)*bmih.biWidth+x)*4+2]; //R
    }
    if (remainder!=0) for(int i=0;i<(4-remainder);i++) _stream<<Temp;
}
_stream.Flush();
File.Close();

Strip = 0;
RCol = 0;
RUpper = 0;
WCol = 0;
WUpper = 0;
do Sonic_PR_ImageMode_Write(0,&Strip, &RCol, &RUpper, &WCol, &WUpper); while
(Code==SONIC_API_FAILED);
Data = 1;
do Sonic_PR_Pipeline_Write (0, &Data); while (Code==SONIC_API_FAILED);
```

```

do Sonic_PR_Pipeflow_Read(0, &Data); while (Data != 0x2);
Strip = 0;
RCol = 1;
RUpper = 0;
WCol = 1;
WUpper = 0;
do Sonic_PR_ImageMode_Write(0,&Strip, &RCol, &RUpper, &WCol, &WUpper); while
(Code==SONIC_API_FAILED);
Data = 1;
do Sonic_PR_Pipeflow_Write (0, &Data); while (Code==SONIC_API_FAILED);
do Sonic_PR_Pipeflow_Read(0, &Data); while (Data != 0x2);
do Sonic_Conf_PE (0,"plug1.rbf"); while (Code==SONIC_API_FAILED);
Data = 1;
do Sonic_PR_Pipeflow_Write (0, &Data); while (Code==SONIC_API_FAILED);
do Sonic_PR_Pipeflow_Read(0, &Data); while (Data != 0x2);
Strip = 0;
RCol = 0;
RUpper = 0;
WCol = 0;
WUpper = 0;
do Sonic_PR_ImageMode_Write(0,&Strip, &RCol, &RUpper, &WCol, &WUpper); while
(Code==SONIC_API_FAILED);
Data = 1;
do Sonic_PR_Pipeflow_Write (0, &Data); while (Code==SONIC_API_FAILED);
do Sonic_PR_Pipeflow_Read(0, &Data); while (Data != 0x2);
do Sonic_PM_Read(0, (unsigned char *)m_pData, m_lBytes, 0); while
(Code==SONIC_API_FAILED);
if (!File.Open( "dst.bmp", CFile::modeCreate | CFile::modeWrite , &Error))
return;

```

```

_stream << bmfh.bfType;
_stream << bmfh.bfSize;
_stream << bmfh.bfReserved1;
_stream << bmfh.bfReserved2;
_stream << bmfh.bfOffBits;
_stream << bmih.biSize;
_stream << bmih.biWidth;
_stream << bmih.biHeight;
_stream << bmih.biPlanes;
_stream << bmih.biBitCount;
_stream << bmih.biCompression;
_stream << bmih.biSizeImage;
_stream << bmih.biXPelsPerMeter;
_stream << bmih.biYPelsPerMeter;
_stream << bmih.biClrUsed;
_stream << bmih.biClrImportant;
for( y=0;y<(int)bmih.biHeight;y++)
{
    for(x=0;x<(int)bmih.biWidth;x++)
    {
        _stream<<m_pData[((y)*bmih.biWidth+x)*4]; //B
        _stream<<m_pData[((y)*bmih.biWidth+x)*4+1]; //G
        _stream<<m_pData[((y)*bmih.biWidth+x)*4+2]; //R
    }
    if (remainder!=0) for(int i=0;i<(4-remainder);i++) _stream<<Temp;
}
_stream.Flush();
File.Close();

do Sonic_Close(); while (Code==SONIC_API_FAILED);
message dlg;dlg.DoModal();
}

```