M

O

N

A

S

H

FOURTH YEAR ELECTRICAL AND
COMPUTER SYSTEMS ENGINEERING
4th YEAR THESIS PROJECT ECE4912

CSIRAC II

Data-flow: a different approach

Brett Dickson

DEPARTMENT OF ELECTRICAL AND
COMPUTER SYSTEMS ENGINEERING
MONASH UNIVERSITY, CLAYTON,
VICTORIA 3168, AUSTRALIA

CSIRAC II

Data-flow:

a different approach.

Author: Brett Dickson

Supervisor: Prof. G.K. Egan

Department of Electrical and Computer Systems Engineering

Monash University

Executive Summary

Computer architectures have until recently been dominated by the von Neumann style architectures. The improvement in re-configurable hardware with the development of larger Field Programmable Gate Arrays (FPGAs) has allowed other styles of architectures to be implemented. One of these styles of architecture is data-flow. The data-flow architecture implemented on an FPGA, as described in this report, is a significant subset of the CSIRAC II data-flow architecture. The performance of this architecture was compared with a von Neumann style control-flow architecture, and comparable results were obtained with noticeable performance benefits with some programs.

Contents

Figures

1 Introduction

The study of computer architectures has been largely dominated by the von Neumann style architectures, which even von Neumann considered as being interim pending more advanced implementation technologies. However at the time it was proposed in the 1940's [1] the von Neumann architecture was the best solution for the available hardware. Despite the improvements in technology allowing other styles of architecture to be implemented, industry's conservative approach has meant that very few of these have been implemented in practice. One of these styles of architectures developed by research organisations was the data-flow architecture. Some of these designs were Monsoon, Manchester Data-flow Machine, and CSIRAC II [2]. With the recent advances in re-configurable hardware, caused by the development of large Field Programmable Gate Arrays (FPGAs), the implementation of these more advanced architectures has become viable.

2 Background

2.1 How do data-flow architectures work

Data-flow architectures only execute instructions when all of the required data is available. This is in contrast to control-flow architectures, based on the von Neumann design, where instructions are executed independent of whether the data is available. The advantage with data-flow is that all of the dependencies commonly existent in control-flow architectures are avoided as only instructions containing all their data are executed.

The programs executed by the data-flow machine are 'directed graph(s) consisting of named *nodes*, which represent instructions, and *arcs*, which represent data dependencies among nodes. Operands are propagated along the arcs in the form of data packets, called *tokens*.' [2] These directed graphs are normally written in a higher level language like SISAL, IF1 or i2 and compiled into a set of node descriptions and input tokens to be read into the processor. Both SISAL and IF1 compile into i2 which is a structural assembly language. When the processor is executing the program, the tokens are fired into the processor and then executed by the Execution Unit. This directed graph approach allows the nodes to be executed in any order, thus improving performance and in some cases simplifying programs that may be extremely complicated to represent as a list of instructions.

2.2 Different types of data-flow architectures

Data-flow architecture can be split into a few groups: Static (Dennis), Static Queued, Dynamic, and Hybrid. The Static architectures are the most restrictive as 'An enabled node is fired if there is no token on any of its output arcs (and when the resources are available)' [1]. This is a problem as more than one token could appear on an arc, thus control tokens are used to block the execution until the output arc is available. The Static Queued architecture permit tokens to be queued on arcs, thus eliminating the requirement of an interconnecting network for the control tokens. The Dynamic architectures also avoid the problem by altering the enabling and firing rule to be 'A node is enabled and fired as soon as tokens with identical tags are present on all input arcs (and when the resources are available)' [1]. These tags contain information about the destinations and an extra colour field to separate tokens on the same arc. In a normal application this could cause the number of tags to increase rapidly, thus a

hybrid of Static Queued and Dynamic was proposed. The Hybrid architecture uses static queuing for the inner loops while the dynamic loop unrolling is used for the outer loops, thus obtaining the benefits of a dynamic architecture while reducing the number of different tags with the static queuing [2].

2.3 The architecture being implemented

The CSIRAC II data-flow architecture was a Hybrid created in 1978 by Egan and was later implemented into hardware in the early 80's [1]. This implementation was extremely complicated to build and reasonably large. Figure 1 below shows the Execution Unit of this implementation and when compared to the FPGA, in the lower corner, it is easy to realise how much re-configurable hardware has evolved in the past few years. Some of the CSIRAC II architecture features include: a self loading Node Store, buffers on both the Matching Store and Execution Unit, most instructions found in a conventional processor and some extras, external network, 128-bit tokens, colours and ordered execution [2]. A few other data-flow architectures use a standard processor to load the node descriptions into the processor, however CSIRAC II does allow the nodes to be loaded through the Input List and also allows them to be changed while the processor is still executing. The external network allows the architecture to operate with multiple processors while also allowing input and output to peripheral devices to be separate from the processors executing the program. The tokens used in this architecture are 128-bit long. These tokens store information about which processor the token is going to be executed on, the ALU to be used on that processor, a colour field allowing loop unrolling and recursion, the type of node the token is for, and input side of the node, data type and the data required for that node.

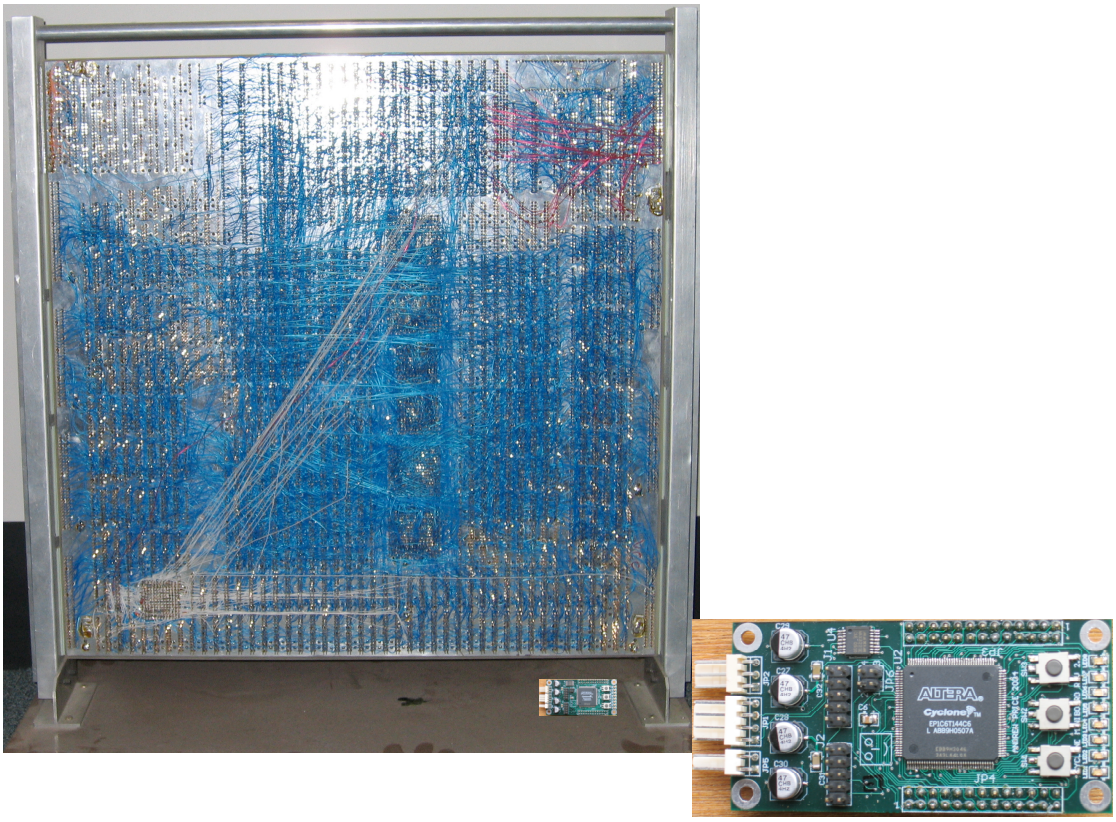

Figure 1. Execution Unit of original CSIRAC II implementation and Altera Cyclone FPGA.

Data-flow programs have two main types of nodes, monadic and dyadic, where monadic nodes have one input and dyadic nodes have two. These types of tokens are not be evenly fired into the processor, thus there has to be some elasticity in the processor to allow for these runs of monadic and failed dyadic tokens. [2] CSIRAC II accommodates this by having buffers before the Matching Store and Execution Unit. Some other architectures like the Manchester Data-flow Machine created by researches led by Watson and Gurd at Manchester University neglected this elasticity causing their processor to continually stall while the Matching Store collected tokens from the linked list [3].

3 Design

3.1 Architecture layout

The data-flow architecture being implemented for this project was a subset of the CSIRAC II data-flow processor. The processor was designed only to execute programs with graphs of less than one hundred nodes with only integer and bit operations being available. The reason for this was to reduce the amount of hardware required to implement it, thus enabling it to fit onto the FPGA being used.

The basic structure of the processor can be seen below in Figure 2.

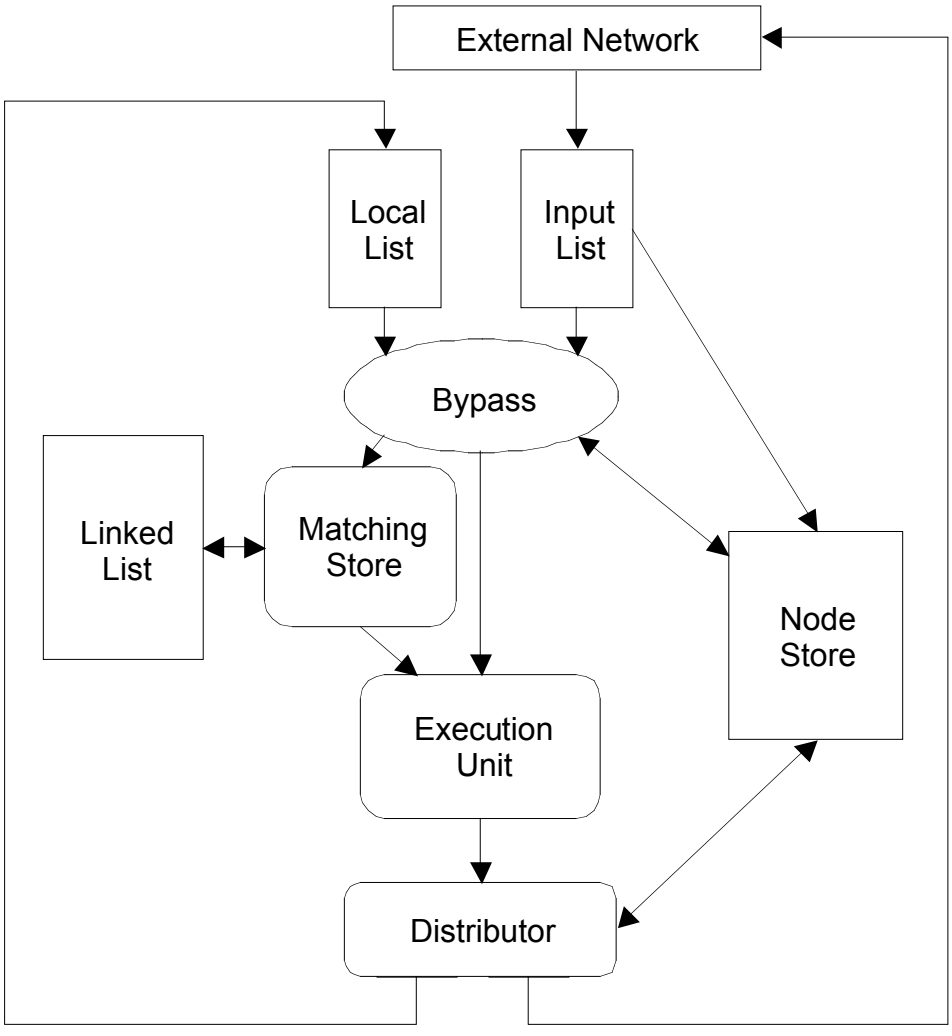

Figure 2. Block diagram of the CSIRAC II architecture.

The processor is split into six distinct sections: the Lists, Bypass, Matching Store, Node Store, Execution Unit, and Distributor. Because of this separation, each section was implemented as stand-alone function blocks with communication channels between them. The details of each section are as explained in the function block description section.

3.2 Function block descriptions

3.2.1 Lists

The lists hold the node descriptions and tokens before they enter the processing sections of the processor. There are two lists: one is a Local List, and the other is the Input List. The Local List is filled from the Distributor on that processor whereas the Input List is filled from an external source, which is normally another processor's Distributor, thus allowing the program and data to be entered into the processor.

3.2.2 Bypass

The Bypass selects between the two lists with a bias to the Local List, as filling this list will cause the processor to stall. The Bypass then adds the function and literal from the Node Store to the token, if required, before distributing the token to either the Matching Store or the Execution Unit. This causes dyadic tokens to pass through the Bypass, which is not necessary, however it does simplify the logic required in the Node Store to collect the function and literal.

3.2.3 Matching Store

The Matching Store is where tokens required for nodes with multiple inputs are matched together. This is accomplished by storing the tokens in an array indexed by the element number, which is the node id. When a token with the same element number is found it is sent to the Execution Unit Buffer with the data from the matching token if it is the opposite input, otherwise it is added to a Linked List connected to that element number. As the new token is added to the end of the Linked List, the order is preserved thus ensuring that the correct set of input data for each node is found.

3.2.4 Node Store

The Node Store is where the information for each node is recorded. The Node Store was split into two storage sections, one containing the function and literal and the other containing where the data is sent after executing the node. The reason for splitting the Node Store is that obtaining the function and literal when the tokens are being manipulated, to insert them into Execution Unit Buffer, reduces the delay before the Execution Unit can begin execution while not affecting the size of the buffers required. The other information is not required until after the execution and would require a larger buffer size if the information was obtained earlier. At the beginning of execution the Node Store contains no information about the program being executed, thus all of the node descriptions are entered through the Input List. Thus the operations of the Node Store consist of three sections: one for each of the storage sections to collect the information and one to insert the node descriptions into the Node Store.

3.2.5 Execution Unit

The Execution Unit uses an arithmetic logic unit (ALU) that is the same as in common architectures with the only difference being the reduced instruction set. The instruction set has been reduced as the processor was designed to only implement integers and a selection of the possible instructions specified for the architecture in The CSIRAC II Data-flow Computer Token and Node Definitions [4].

3.2.6 Distributor

The Distributor sends the newly created tokens to either another processor, or to the Local List depending on the distribution system being implemented.

3.3 The features of the subset being implemented

The subset of the CSIRAC II data-flow architecture implemented has the following instructions: Add, Subtract, Multiply, Shift Up, Shift Down, Equal to, Not Equal to, Greater or Equal to, Pass if True, Pass if False, Switch, Nop, and Replicate. These instructions can be either Bypass or Normal nodes, where the type of nodes indicates the number of inputs: Bypass has one input and Normal has two. The data types available on the processor are 8-bit, 16-bit and 32-bit integers, single bit, and node descriptions. All of these data types are contained in single word tokens except the node descriptions which in this implementation were restricted to the Input List and Node Store. The other restrictions are that nodes can have a maximum of two destinations, only Bypass nodes can have literals, and there is a maximum of 125 nodes in the directed graphs being implemented on the architecture. The processor it self is a single processor which accepts 128-bit input tokens and outputs 32-bit values if the destination is an output node.

3.4 Design considerations

The buffer and list sizes had to be carefully considered as filling one of these could cause the processor to deadlock. While it may be advisable to perform some probability tests in order to obtain the optimal size, in this implementation the FPGA resources determined the buffer and list sizes. For the buffers and lists using RAM, the size was dependent on the smallest number of RAM blocks required as there was a limitation on these blocks and once a block was required there was no reason not fully utilizing it. On the other hand the lists using an array were restricted to the absolute minimum size to enable the processor to fit on the FPGA. Therefore the buffers and lists using RAM were larger than required with no reason to reduce their size and those using arrays were as small as possible with major performance deterioration if they were increased. Even though the FPGA resources dictated the size of the buffers and lists, the comparative size required had to be considered. The Local List was made the largest, as filling this list will cause the processor to deadlock. Similarly the Execution Unit Buffer size was obtained by the maximum influx possible in the processor. The smallest in this implementation was the Input List, as there was no external network this list only contained triggering tokens, and a stall would not affect the rest of the processor.

As the processor was the only implemented part of the architecture, some of the data fields could be reduced while others could be excluded. The excluded fields are colour, process, and processor since the colours were not being implemented, there was only one ALU and only one processor. The fields reduced in size were the data

field, data type field, node id and the addresses to the Linked List. The data field was reduced to 32-bits since this was the largest data type the processor was capable of handling. The data type field was reduced to 4-bits so these two fields fit neatly into the 36-bit words of the RAM blocks, thus reducing the number of RAM blocks required for the Linked List while still being large enough to hold all of the data types. The node id and addresses to the Linked List were reduced to their minimum size because there is no reason for storing bits that are never going to be used.

As there are two lists feeding the Bypass a selection has to be made as to which list the next token will be taken from. The main requirement is that the Local List never becomes full, as this will deadlock the processor. Some approaches are to take the tokens from the fuller list, take the tokens from the Local List unless the Input List reaches a set level or take the tokens from the Local List unless the list is empty. Due to there being no external network, the only tokens in the Input List will be priming tokens thus the Local List will normally be empty when the Input List contains tokens. Therefore the third approach was used as there was no benefit in complicating the design to ensure the Input List never became full.

4 Design Tools

4.1 Compiler

There are many different compilers to program an FPGA, however only two were chosen as possible compilers. These were Impulse-C and Handel-C. The reason for choosing these was their similarity to standard C, thus there was familiarity with the code and only a few new operations had to be learnt as opposed to an entirely new language. One of the other possible languages was VHDL, however as C is a higher-level language the development of the processor will be simpler and clearer to follow, thus allowing the program to be easily extended or altered.

4.1.1 Impulse-C

Impulse-C offers a large library of functions to implement many of the more common operations, such that the program is able to implement these processes in the most optimal way [5]. However because the implementation is hidden the processes cannot be adjusted to suit implementations that are slightly different from normal. This becomes a problem when extra wide channels are required or queue jumping is desired for a possible extension to reduce delays in the Matching Store. The other problem with having processes in place to implement all of the channels as first in first out (FIFO) queue is that they require setting up, thus there is a reasonable amount of initiation code. The other major feature of this program is that each section is set up as an individual process with inter-communication through channels, thus giving the program a good structure, however this requires an extensive amount of initiation code to initialise.

4.1.2 Handel-C

Handel-C on the other hand only offers some special operations to allow variables to be manipulated and channels to allow handshaking transfer of data between different sections of the code, which is operating in parallel [6]. For this reason most of the operations like the queues have to be written in code thus increasing the amount of code written, however it does allow the width of the queue to be unrestricted. The width of the channels and variables can be preset or left for the compiler to set the

optimal width. This program also implements bit extraction better than Impulse-C as it allows any number of bits to be extracted by accessing them like bits in an array as opposed to having a function call which only allows bits to be extracted from 32 bit numbers.

4.1.3 Compiler Selection

The desired compiler was Handel-C as it does not have large amounts of initiation code and has no restrictions on variable size and bit extraction. The downside was the lack of pre-designed features i.e. the FIFO queues, however there was a possibility that the buffers and lists would require extra features to improve performance thus they would no longer be standard FIFO queues.

4.2 Loader Considerations

As the Handel-C package being used does not download the program directly onto the FPGA, Quartus was used to compile the EDIF file generated by Handel-C. This complicated the optimization of the processor as Quartus only gives line numbers of the variables causing the restrictions on the clock speed, thus the Handel-C code had to be cross referenced to find the section where the assignment occurred. This may have been a direct assignment or a sequence of control logic causing the assignment to restrict the clock speed. These could however be traced with the expected delays output files generated by Handel-C which listed the longest delays for each type of assignment and indicated the lines in the path.

4.2.1 Setting up Quartus

As the compiling of the program to the FPGA occurs in two different programs Handel-C generates a tcl script to correctly set the FPGA device and link the virtual pins created by the EDIF file to actual pins on the chip. As this linking is done automatically by running the tcl script, the pins in the Handel-C code had to be declared correctly. The pins were declared by typing only the pin number between talking marks i.e. "*pin number*". An example of this is in Appendix C. Architecture.hcc.

There are also some settings in Quartus that have to be set in order for the FPGA to operate correctly, these are all unused pins have to be set to tri-state inputs, and in the 'Design Entry/Synthesis' the 'Tool name' has to be set to custom with the 'Format' set to EDIF and the 'Library Mapping File' linked to the celeriox.mlf file in the Handel-C directory.

4.2.2 Initializing external RAM.

The external static RAM can be initialised in a few different ways, one is to load a program onto the FPGA to initialise the RAM before the actual processor is loaded, and another is to load the information into the flash RAM and have part of the program move this information to the static RAM. The loading of the program to initialise the RAM is the simplest method and works very well as long as the device has constant power. The other advantage with this method is that the processor can be loaded into the flash memory, which enables the processor to be reloaded by pressing the configure button on the board. Thus the program can be loaded onto the FPGA using the Joint Test Action Group (JTAH) port and executed by pressing the

configure button. The memory initialization program simply copies the program from initialized on-chip RAM to the static external RAM, thus only the memory initialization file (mif) has to be changed in Quartus to load a new program.

4.3 FPGA

The FPGA used was an Altera Cyclone EP1C6T144C6 FPGA, which has the following specifications:

- 20 M4K RAM blocks.
- 92160 RAM bits
- 5980 Logic Elements (LF)
- True dual-port RAM.

This chip was used with a development board designed and built by Dr. A Price of the Monash University Department of Electrical and Computer System Engineering (Appendix Q) containing.

- External Static RAM
- Flash RAM
- Eight LEDs

Some of these features, while being adequate for the project, placed restrictions on the complexity of the implementation and the way in which the processor was implemented. An example of this is the implementation of the Input and Local Lists. Due to the limited number of M4K RAM blocks and Les, the Local List was implemented using RAM, in the dual-port configuration, as it had to be larger to avoid the processor from deadlocking when the list became full. However the Input List was implemented as an array due to the restrictions on the number of RAM blocks and its size, as only a few tokens were stored in it at any one time. The number of LEDs available on the development board, while limiting displayable, information were adequate for this project.

4.3.1 Hardware design considerations

The storage of data was a major design consideration as the type of storage used and the size of the data has a major affect on the processor's performance. There are three types of storage: arrays and variables, on-chip RAM and external RAM. Each of these has their benefits and drawbacks, thus choice of storage for each application has to be carefully considered to ensure the best one or mixture is used.

The array does have a distinct advantage over RAM when it comes to accessing the data. The array allows unlimited accesses to the data during a clock cycle, compared to the on-chip RAM which restricts the accesses to a maximum of two per clock cycle, for the entire instance of RAM, when dual port RAM is used. Access speed is another area where the array delivers better performance than the RAM. The reason for this is that the array is created with a set of flip-flops joined together and therefore is mainly restricted by the control logic needed to locate the required set. RAM requires longer setup and hold times to access the element contained in the RAM block and is therefore restricted to a maximum of 200MHz [7], although the control logic is more efficient. The major advantage RAM has over the array is the area required on the chip. For an array each bit requires one logic element (LF) on the FPGA, which could otherwise be used for logic required to implement the processor whereas RAM is already allocated room on the chip, which can only be used for RAM. The down side to this is that RAM is segmented into blocks, which can only be

accessed a maximum of twice in a clock cycle and these accesses can only retrieve 36-bits at a time. Thus the standard data field size of 40-bits would be spread over two RAM blocks to allow it to be accessed every clock cycle. Due to this, a combination of both RAM and an array is sometimes desirable as the majority of the data can be stored in RAM with the extra bits stored in an array, thus reducing the number of RAM blocks and LEs used. However if a large amount of memory is required the external RAM on the development board is available. This RAM allows one access per a clock cycle and is limited by an 8-bit data bus, thus data requiring a larger word size requires multiple clock cycles to access.

Another design consideration is the depth of the logic being used and the fan out required for accessing variables multiple times. These affect the clock speed as they create long data paths, thus causing large propagation delays. One of the areas this becomes apparent is around RAM accesses, as they are initially restricted and any further delay causes them to further restrict the maximum clock speed.

5 Implementation of the Architecture

5.1 Overview of the Architecture

The architecture is split into five sections as specified in the design sections. In addition to these sections there is a function to retrieve the program, which is stored in external RAM, and load it into the Input List, and a function to operate the display.

During the execution of this program, values can be viewed on the leds located on the development board by using the appropriate instruction. The only drawback with this is that the speed of the processor may cause the leds to flash too quickly to be observed, therefore a delay section of code was added to the design stalling the processor while the value was being displayed.

Another feature of the processor is a memory led, which indicates when one of the memory sections has become full which will probably cause the processor to deadlock. The cause of this in most cases is the Local List becoming full due to excessive branching in the program, however allowing un-matched tokens to build up in the Matching Store has the same effect as the memory allocated for the Linked List becomes full. These problems can be avoided by writing the program so it does not excessively diverge before converging and ensure that no one side of a dyadic node enters the Matching Store excessively more often at any particular point in the execution.

5.1.1 Design considerations

The major design consideration was whether to use RAM or arrays. In general this decision was already made with the size restriction the FPGA placed on the project. Thus the majority of storage was placed in RAM, with a lesser amount implemented using arrays and the program executing on the chip stored in external RAM. The use of external RAM was avoided in the main section of the processor because of the effect it had on performance. Therefore the external RAM was restricted to storing the program, which would only be accessed once, thus avoiding the performance degradation in the main section of the processor while allow the processor to fit on the chip.

Another consideration was how Handel-C implemented some of its available features. A perfect example of this was the default division as it required the entire FPGA and reduced the maximum clock speed to 2MHz. The alternative to this was to implement a pipelined division, which would complicate the design. Altering the design will probable still restrict the processors performance while only prove that there are implementation of ALUs available with division, which can be implemented on the FPGA. Thus division was excluded for the instruction set and replaced with shift operations allowing the programmer to achieve the same results using different operations. Channels are designed for passing values between parallel sections of code using handshaking. While this is useful in the correct implementation, in many cases it was not practical to pass data this way, thus for some sections the data was passed using variables and signal.

5.1.2 Implementation restrictions

Due to the FPGA size restrictions and the desired performance requirements the following restrictions were placed on the processor.

The Linked List which stores the multiple tokens for a particular node in the Matching Store was placed in on-chip RAM, thus its size was limited to 256 locations which in most cases would be more than adequate as the probability of having more than three tokens for one node stored is extremely low.

The number of nodes allowed in the program was restricted to one hundred. This allows small programs to be executed on the processor, thus allowing the design to be tested while not creating problems caused by allocating excessive amounts of memory and therefore restricting the available space on the FPGA required for implementing the rest of the processor.

The instruction set and number of types were reduced. The reason for doing this was to simplify the ALU being used in the implementation as this part of the program could very easily be obtained from a conventional processor which implements a larger instructions set and more data types. Thus a select few instructions were chosen to allow more of the chip space to be dedicated to the data-flow sections of the processor while still allowing suitable programs to be written for the processor.

The branching factor for each node was restricted to two destinations. The reason for this is that the design is for a single processor, which could be joined to other processors later by using multiple FPGAs. Thus, having a larger branching factor would only fill the Local List with tokens faster than they can be removed causing the processor to deadlock. In a multiple processor design these extra tokens would be spread between the processors thus avoiding this problem.

5.1.3 On-chip specifications

The Quartus compilation results (Appendix R).

- 5,260 / 5,980 (87%) LE's
- 19 / 20 M4K RAM
- 62,090 / 92,160 (67%) bits of RAM
- 33 / 98 (33%) pins used
- 50 MHz clock.

Despite the fact that there were a limited number of M4K RAM blocks only 19 of the 20 were used, as altering the design to increase the storage used required two RAM blocks due to the way they had been configured. This configuring of the RAM in the RAM blocks also meant that most blocks were not fully utilized, thus only 67% of the total RAM bits where used. Not all of the LEs available on the FPGA were used, the reason for this is that increasing the percentage of used LEs increases the interconnections on the FPGA, thus reducing the clock speed. A larger FPGA would have reduced this limitation on the clock speed while allowing sections of the program to be written differently to obtain a higher clock speed.

5.2 Lists

The lists were FIFO queues with the head and tail recorded thus allowing the tokens to flow through the list with a minimum delay of one clock cycle. One less obvious feature is that the head and tail are only incremented with the size of the variable being used to wrap the list around, thus simplifying the calculation. The addition and removal of tokens was implemented in two independent parts of the procedure with a common array or RAM and variables thus allowing them to work independently of each other. To reduce the time required to remove tokens it is assumed the list always contains tokens, thus if the list would normally be empty in the next clock cycle an empty token is inserted. Similarly, the insertion of tokens does not check if there is room in the list, as another section of the program checks if the list is becoming full and stalls the pipeline. To eliminate pipeline stalls in the Distributor, the Local List is able to accept two tokens at a time, thus allowing the two possible destinations of each token to be inserted into the Local List at the same time. To enable this to occur multiple storage elements are used (figure 3) as this simplifies the incrementing of the tail and allows these tokens to be simultaneously inserted into the RAM blocks. When outputting the tokens, both lists send them to the Node Store and the Bypass, thus allowing the information from the Node Store to be collected and arrive at the Bypass at the same time thus reducing the logic required to implement this process. As the Bypass has to choose between the lists, only taking a token from the desired list, empty tokens are not sent thus allowing the Bypass to only choose between valid tokens.

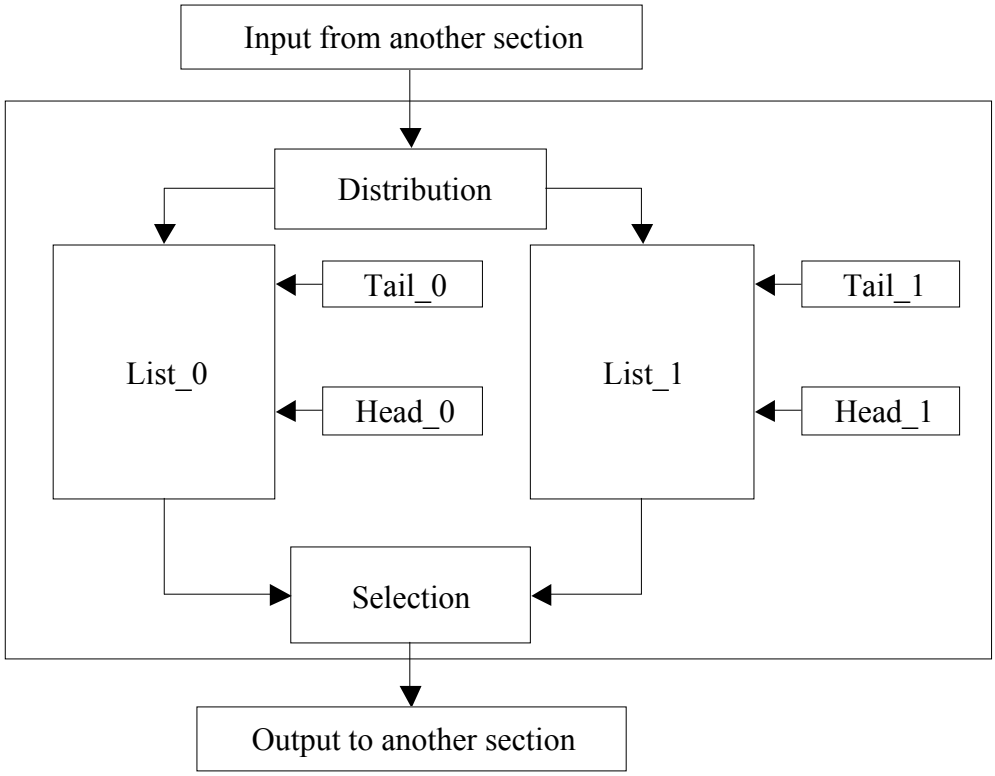

Figure 3. Layout of the Lists or buffers with multiple storage elements.

The Input List also receives the tokens into the processor and either places them in the list or sends them to the Node Store if they are node descriptions. In doing so the Input List also reduces the data stored by disregarding the extra bits in a standard token that are not required for this processor. The advantage with implementing the inputting into the processor in this way is that it allows the standard compilers for the language to be used while also removing unnecessary information to allow the processor to fit on the chip.

5.2.1 Design considerations

These lists while functioning in the same way, have very different size and input requirements affecting how they are implemented. These differences affect the preferred type of storage: RAM, an array or a combination of both. The main factor influencing the choice is size as any array requiring more than a few hundred bits uses up too many LEs. For this reason the Local List was placed in RAM while the Input List was placed in an array, as it could be restricted to a few hundred bits and there were not enough RAM blocks available. As the Input List is implemented as an array it does not have any accessing restrictions, whereas the Local List can only add and remove one token per clock cycle. This causes a problem as the nodes can have two destinations thus multiple lists are used for the Local List, this does however use more RAM blocks thus restricting the number of RAM blocks available to other parts of the processor.

5.2.2 Implementation restrictions

In order to use the wrapping of the head and tail, the lists were restricted to orders of twos. This had very little effect on the Local List, as the maximum number of elements available in the RAM block was 128. Thus this allowed the Local List to be 256 elements, which is larger than required, however no extra resources were required to have it this size. The Input List was made as small as possible to reduce the number of LEs as these will affect the resources required. Thus a size of four elements was chosen, as this was the smallest size that did not have problems with testing if the list was full.

5.2.3 On-chip specifications

Input List specifications

- 2 elements
- Implement as an array
- Stores 45-bit tokens
- FIFO list
- Minimum delay of one clock cycle

Local List specifications

- 256 elements
- 2 interleaved lists
- Implemented using on-chip RAM
- FIFO list
- Minimum delay of one clock cycle

5.3 Bypass

The function of the Bypass is to collect the token from the desired list with a bias for the Local List. This token is then attached to the functions obtained from the Node Store and literal if required, before being sent to the Execution Unit or Matching Store depending if the token is monadic or dyadic (figure 4).

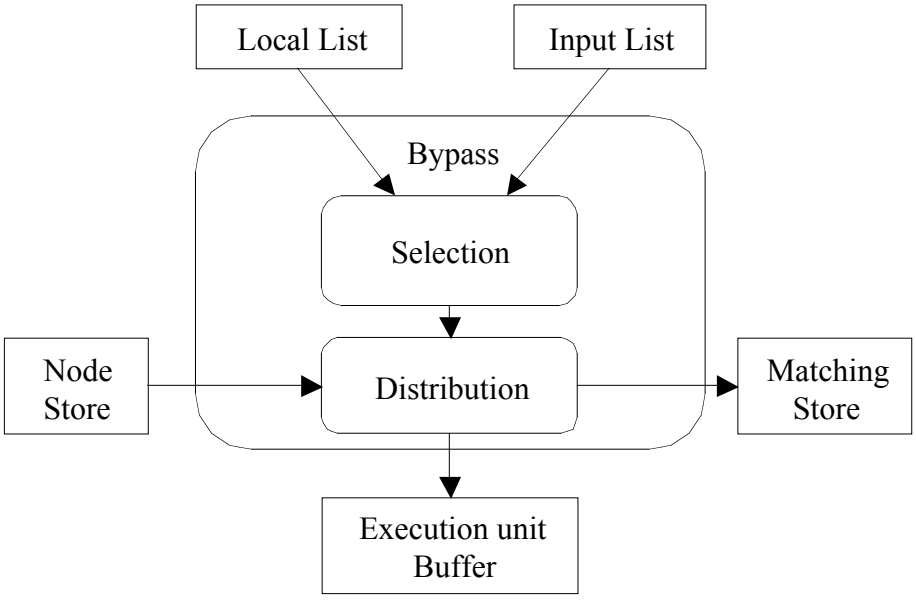

Figure 4. Layout of the Bypass.

5.3.1 Design considerations

The design considerations in the Bypass were how to send the data to the appropriate destinations and the biasing on the list selection. The data was sent using two different methods. Due to the implementations of the following sections of the processor these were a signal to the Execution Unit and channel to the Matching Store with empty tokens being sent to keep the Matching Store pipeline full. Selection was biased to the Local List as filling this list causes the processor to deadlock. There is no reason for using more complicated methods of choosing from the fuller list, as the Input List is mainly used at the start of execution to load the program and trigger tokens, as there is no network.

5.3.2 Implementation restrictions

A restriction of the design is that only one token can be retrieved from a list at a time thus causing a bottleneck. Most of the time this will not be a problem as the Input List will be empty due to the processor not being part of a network, thus there will only be one list to collect the tokens from anyway.

5.3.3 On-chip specifications

The features of the Bypass were:

- Blocking input channels
- Signal outputs
- Selects from the Local List first

5.4 Matching Store

The Matching Store is the heart of the data-flow architecture, where most of the tokens are matched together before being sent to the Execution Unit, thus an inefficient Matching Store will greatly affect the performance of the processor. The Matching Store can be spilt into four sections receiving a new token from the Bypass, collecting the information, updating and sending the information, and storing the tokens (figure 5). Each of these sections can be implemented in various different ways to ensure maximum performance is achieved.

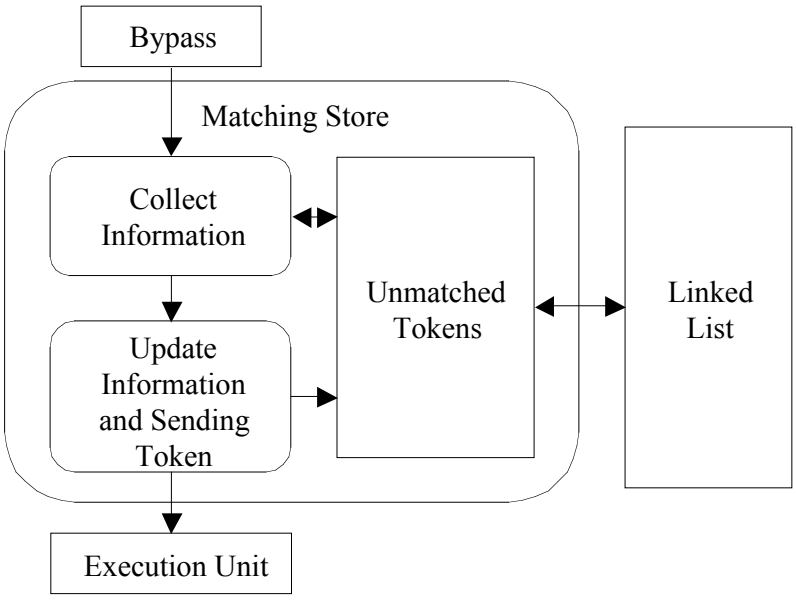

Figure 5. Layout of the Matching Store.

The receiving of tokens was implemented without a buffer, thus any delays in the Matching Store will propagate to the inputs into the processor and cause it to stall. This was done due to the limited number of resources available on the chip, thus these resources could be better utilised in another section of the processor.

Collecting of the information was implemented as a separate pipeline stage to enable the clock speed to be maintained while still allowing the required test to be carried out in the following stage. In this stage the information corresponding to the token in the Matching Store is obtained, however the removal of the buffer has made it crucial that there are no delays, therefore the values that have changed in the following pipeline stage are forwarded and used if the following token corresponds to the same node. This does create extra complication and control logic, however the performance achieved by utilising the memory allocated to the buffer in other sections of the processor outweighed any performance deterioration of the Matching Store.

The updating and sending of the tokens were combined into a single pipeline stage, as they required the same control logic and both could be completed in a single clock cycle. This was only possible as the programs were restricted to flat graphs as colours were not implemented, therefore any matched tokens would be removed from the front of the list, there is no searching requiring multiple clock cycles and a simple single linked list can be used. Knowing where the token will be removed from also simplifies the updating of the information as the updating section knows immediately if the token is going to be removed, as the sections are closely connected, and its location, thus the information can be updated by the completion of the second pipeline stage of the Matching Store. The only problem with this is that new values are unavailable until the next clock cycle, therefore these alterations and any other information that may have changed is forward to the previous stage. The reason for forwarding all of the information is that it reduces the logic required in the collecting stage to ensure the information is up to date.

Storing of the tokens was split into two sections, one for the actual Matching Store and a second for a Linked List to store multiple instances of the tokens. The Linked List was implemented using on-chip RAM. While this did restrict its size to 256 elements it was still large enough and only took one clock cycle to access the data, thus eliminating delays compared to five clock cycles if external RAM was used. To ensure none of the elements in the Linked List were overwritten a linked list of empty addresses was created, thus the next empty address could be quickly obtained and recently vacated addresses recorded. The Matching Store itself was also implemented using on-chip RAM for storing the control values, data and addresses to the Linked List. The technique for accessing this RAM was direct mapped, as the colours were not implemented, thus the first token stored will be the first to be matched. Therefore this first token was always stored in the Matching Store section of the memory and is immediately replaced by the first token in the Linked List when it is removed to reduce the logic required to locate this token. An alternative approach is to use the memory allocated to this token to store the addresses to the Linked List, when there is multiple tokens, however due to the number of bits required this would have no benefit in reducing the memory required as the same number of RAM blocks would be used.

5.4.1 Design considerations

The Linked List and Matching Store storage could be stored using arrays, on-chip RAM, external RAM or a combination of these with each having its benefits and restrictions. The array is the fastest of the three with the major drawback coming from the amount of chip space required for each bit. The second fastest is the on-chip RAM, which has an area on the chip allocated to it and thus it has minimal effect on the amount of the chip available for logic. However, there is only a set amount of RAM and RAM blocks. Each of these RAM blocks can only be accessed once in the clock cycle thus tokens requiring words larger than 36-bits require multiple tokens, however once a RAM block is being used there is minimal penalty in using all 128 elements. External RAM is the slowest of the three requiring multiple clock cycles to access, however it does have the advantage of being larger than any on-chip RAM. After considering the benefits and restrictions on the types of storage, on-chip RAM was chosen for the Linked List and the Matching Store storage, as they are both too large for arrays but not too large for the on-chip RAM thus there was no reason to incur the speed penalty of external RAM. The Matching Store was written so all of the stored values are only accessed once for reading and writing per clock cycle enabling dual ported RAM to be used. The size allocated to the Matching Store storage was dependent on utilising the minimal required amount of memory as it is only available in discrete amounts. The Linked List was slightly different with the size being restricted by the number of RAM blocks available after the rest of the processor was implemented.

Indexing of the memory could have been accomplished by either direct mapped, hashed, or using a 2-way set associative cache. Each of these methods has their advantages and purposes. Direct mapped is the easiest to implement and access as the index is used directly, thus ensuring fast accessing, although there is a possibility of having only a small portion of the memory used. Hashing has the advantage of spreading the values more evenly across the memory and possibly allowing the memory size to be reduced, however there is a potential to have to look in multiple locations if multiple tokens have the same hash values. The 2-way set associative cache uses two memory locations, the faster cache to store the more recently added tokens and slower directed mapped or hashed memory to store all of the tokens. The advantage with this is tokens will normally arrive at approximately the same time as their pair, thus these can be quickly accessed from the limited amount of fast expensive memory while the other tokens are still stored in the slower more abundant memory. This technique, of using an associative ‘token cache coupled with a secondary hash table in bulk memory’ [2], was used in the full implementation of the CSIRAC II architecture as the key field was too large for direct mapped memory. In this implementation a direct mapped approach was used due to its simplicity. Reducing the number of locations under one hundred would have no size advantage as the same number of RAM blocks would be required, likewise a cache would have no benefits as there is enough of this fast RAM available for this implementation, and as there were no colours there were no benefits in using hashing to store the tokens.

The Matching Store could be implemented in any number of pipeline stages. The advantage with increasing the number of stages is that the clock speed and through put can be increased, however there are other restrictions placed on the clock speed due to RAM accesses and control logic. For these reasons the Matching Store was implemented in two pipeline stages, thus allowing RAM indexed by RAM to be

spread over the two stages avoiding double RAM accesses, which restrict the clock speed. Increasing the pipeline further would have minimal benefit as the major restrictions on the clock speed were caused by RAM accesses and extra logic would have to be considered for forwarding results as any stalling of the pipeline would be unacceptable.

Consecutive tokens corresponding to the same node are a concern due to the multiple pipeline stages. There are two methods to overcome this problem, the first is to delay the pipeline until the values have been updated and the second is to forward the information. The first does have a major drawback as consecutive sets of these conditions could occur causing the tokens to bank up behind the Matching Store. The second ensures there are no pipeline delays, however requires more logic to forward the values. The other advantage with the second method is that the buffer can be excluded, which uses much more logic than that required to forward the values and has the added advantage of reducing the overall minimal number of pipeline stages for the Matching Store to two.

5.4.2 Implementation restrictions

Due to the on-chip RAM being used for the Linked List there is only 256 elements, however this should be sufficiently large to hold the Linked List required for any graph as long as it is written acceptably. In the event the Linked List becomes full the processor will deadlock with a signal being sent to an external pin, thus it is crucial that this does not occur.

In this implementation of the Matching Store only flat graphs can be implemented, as only the first token in the Linked List is matched with the incoming tokens, also colours have been excluded from this implementation. The reason for using colours is it allows different instances of a node to be distinguished instead of ensuring they are always executed in the same order. This enables recursive programs to be executed, however it complicates the Matching Store, possibly making it too large to fit onto the FPGA, thus it has been excluded from this implementation.

5.4.3 On-chip specifications

The features of the Matching Store were:

- 125 directed mapped elements
- 256 element linked list
- Single linked list
- 2 clock cycle execution
- Implemented using on-chip RAM

6 Node Store

The Node Store has three sections, these are inputting of the node descriptions, collecting the function and literal, and collecting the destinations (figure 6). Inputting of the node descriptions obtains the tokens from the top of the Input List and places them into the appropriate section of the Node Store. The storage of the information is split into two sections corresponding to the collecting sections of the Node Store. As RAM is used for the storing, some information is stored in temporary variables until the rest required for that section is obtained from a following token.

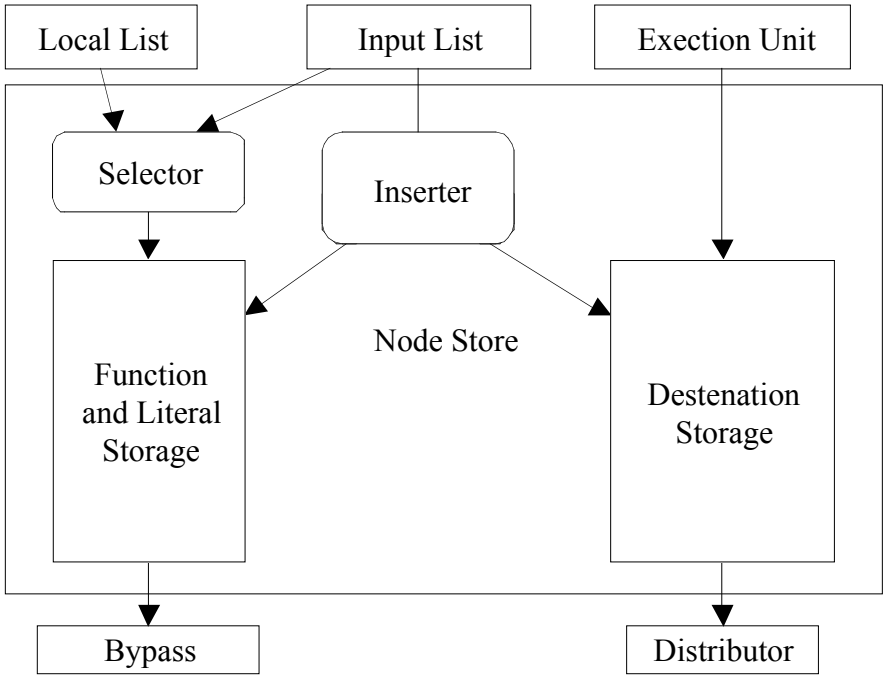

Figure 6. Layout of the Node Store

The collecting of the function and literal uses the read only part of this section of the storage, which is accessed using a signal. The advantage of doing this is that the information can be sent in the same clock cycle as it is requested, thus reducing the logic required to obtain the information at the start of the Node Store. As this is done in one clock cycle the literal value is sent regardless, so the Node Store does not affect the overall clock speed of the processor with the value being disregarded in the Bypass if it is not required.

Collecting of the destinations operates in the same way with the request sent in the Execution Unit to be received in the Distributor at the same time as the result. The reason these two sections are separate is that the destinations are not required until later in the processor than the function and literal, therefore the memory required can be reduced by collecting them at different stages in the processor. The function and literal could have been collected later however this would complicate the Execution Unit Buffer and possibly affect the maximum clock speed of the processor.

6.1.1 Design considerations

The type of storage used for this section had to be carefully considered as there is a large amount of data to be stored and the access speed of this section will greatly affect the processor as it is accessed for every token that enters the processor. For these reasons on-chip RAM was chosen, as the access speeds would not degrade the processor clock speed while still being an efficient way of storing the information.

The two collecting stages could be implemented with either one or two clock cycles. Both methods deliver the same results, however with the two-clock cycle design the requests have to be sent earlier creating the need for extra logic to ensure there are no pipeline delays waiting for the Node Store, while having no performance benefits to the overall processor.

6.1.2 Implementation restrictions

Due to the size restrictions of the FPGA, the Node Store is only able to hold one hundred nodes with the possibility of extending it to one hundred and twenty seven. Any larger than this would require more RAM blocks which are not available. This restriction on RAM blocks also restricts the Node Store to only allowing one collection of the function with the literal in one section and one collection of the destinations in the other section per clock cycle. This could be altered to allow two collections of each, however this would require the assurance that the Node Store was completely loaded before the processor begins or control logic to avoid one of the collections occurring when information was being stored.

6.1.3 On-chip specifications

The features of the Node Store were:

- Same clock cycle retrieval on of information
- Implemented using on-chip RAM
- Separate storage for the function and literal, and destinations
- Maximum storage of 125 directed mapped nodes

6.2 Execution Unit

The Execution Unit consists of a buffer and an ALU (figure 7). The ALU section was implemented with a switch statement to choose between the different instructions available to the programmer. All of these instructions are single assignment instructions on 32-bit integers with any smaller integers being sign extended to allow the same functions to be used. The Execution Unit Buffer was required as tokens can enter from the Bypass and the Matching Store, thus the buffer can handle the two tokens without delaying either section of the processor whereas the Execution Unit would have to stall the pipeline as it can only accept one token at a time. This buffer has 256 elements, which is more than enough as the maximum influx of tokens is four, as the Matching Store only has two pipeline stages. Any further increase will only occur if the Local List becomes full, which will deadlock the processor anyway. The reason there are so many elements is that the buffer is implemented similarly to the Local List with multiple storage sections (figure 3) to allow for multiple insertions in a clock cycle, thus the maximum number of elements using the minimum number of RAM blocks is 256.

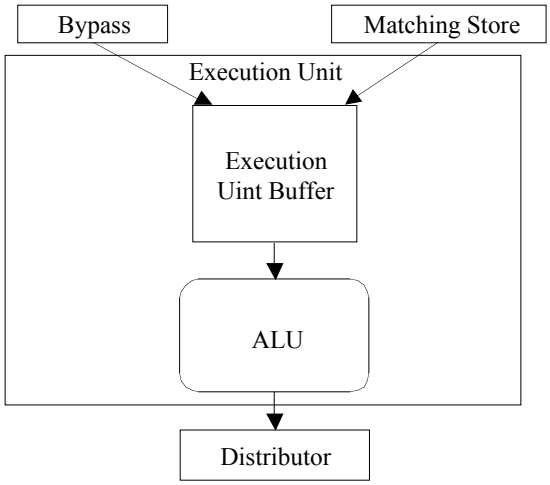

Figure 7. Layout of the Execution Unit.

6.2.1 Design considerations

The instructions could have been implemented as separate functions for each of the types, however as they all behave the same this would only increase the logic required while not delivering any performance benefits as the design would be limited by the 32-bit integer instructions. Also it is easier to sign extend everything to 32-bits, than to calculate what the data has to be extended to.

The size of the buffer was chosen to be 256 elements as this would be sufficient to handle the possible fluctuations in tokens while not being too large to require extra resources while fully utilizing the minimal number of RAM blocks.

There are a few different ways to receive and input the tokens into the Execution Unit Buffer. The first stores them in a temporary location allowing the nop instructions to be discarded and the others to be correctly added to the buffer, while the second method uses signals allowing the values to be checked and stored in the same clock cycle. The advantage with the first method is that the amount of logic in each clock cycle is reduced thus allowing the clock speed to increase, the down side is there is an extra pipeline stage which can be avoid by adding extra control logic to Bypass the buffer if it is empty. The second approach was chosen as it was simpler, thus requiring less control logic to obtain the same results, and as the speed penalty was not noticeable.

6.2.2 Implementation restrictions

There is a limited number of instructions available, this simplified the Execution Unit and avoided the clock speed form being limited by the ALU. For this reason a major exclusion was division as this reduced the clock speed by 90% and required more space than was available on the FPGA. However there is a suitable selection of instructions to enable most programs to be executed on this processor.

The number of data types was also limited to reduce the size of the ALU and the size of the data being stored on the FPGA, thus allowing the performance of the processor to be increased.

The Execution Unit contains a single ALU thus only one instruction can be implemented at a time, however the number of tokens entering the processor is basically equivalent to this, thus it should not create a bottleneck in the processor.

6.2.3 On-chip specifications

The features of the Execution Unit were:

- 13 instructions
- 32-bit sign extension
- Precision is maintained
- 256 element buffer
- Single clock cycle instructions
- Implemented using on-chip RAM

6.3 Distributor

The Distributor obtains the information from both the Execution Unit and the Node Store to enable it to send the appropriate data to the Local List, as there is no network (figure 8). One of the pieces of information obtained from the Execution Unit informs the Distributor if it should send the data to all of the destinations, one of the destinations or none of the destinations, the other is the data to be sent which is combined with the destinations obtained from the Node Store. As there is a maximum of two destinations the Distributor was enabled to insert two tokens at a time into the Input List, therefore ensuring there are no pipeline delays caused by this section of the processor.

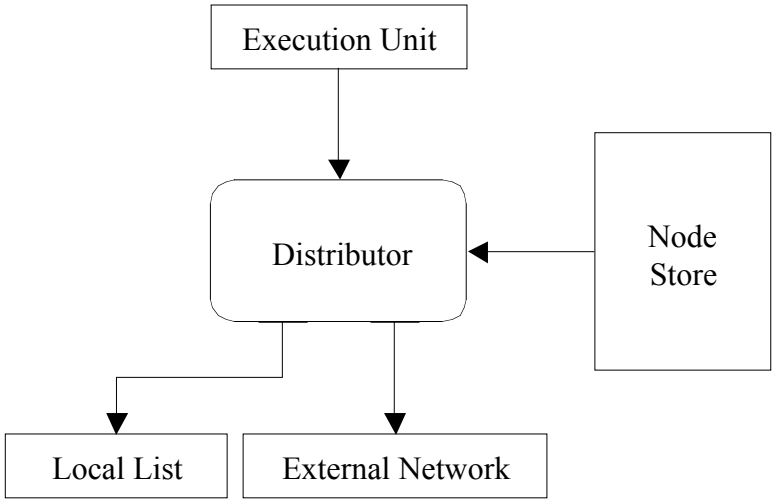

Figure 8. Layout of the Distributor.

6.3.1 Design considerations

As the Distributor only inserts tokens into the Local List the branching size had to be considered as too large a branching factor would fill the Local List, increase execution time, and cause possible pipeline delays while inserting tokens into the Local List. Thus a branching factor of two was chosen, as it was suitable for the size of the problems being implemented on the processor while not placing a large demand on FPGA resources to obtain an implementation without pipeline delays.

6.3.2 Implementation restrictions

The Distributor was restricted to only accepting one answer from the Execution Unit and generating two tokens at a time, thus ensuring that there are no pipeline delays while not over restricting the programs that can be implemented on this processor.

6.3.3 On-chip specifications

The features of the Distributor were:

- Maximum of two destinations
- Only one clock cycle to store the destinations

6.4 Tokens and Nodes

This architecture operates using tokens that contain the data and nodes that contain the instructions to perform on the data and where the result is to be sent. The tokens being inserted into the processor were implemented as specified in The CSIRAC II Data-flow Computer Token and Node Definitions [4] with the restriction that they were fixed length to handle a maximum of 32-bit integers and that vectors were not allowed. Once in the processor some of the fields are reduced or ignored, as the processor was not designed to handle these features. One such feature is the colours, in the specifications there is 40-bits allocated to the colour field, which is ignored as the processor does not implement colours, as this would increase the required control logic causing the processor to become to large for the FPGA and the physical space required on the chip to store the field is unavailable.

The Node descriptions were inserted into the processor using tokens and are implemented as specified in The CSIRAC II Data-flow Computer Token and Node Definitions [4]. Once in the processor they were inserted into the Node Store with the relative information being extracted, thus reducing the resources required to store the large tokens the node descriptions are stored in.

6.5 Problems encountered and solutions

Some of the problems encountered during this project were:

- Collecting the outputs from the Bypass and the Matching Store and inserting them into the Execution Unit Buffer. As it was decided to do this in one clock cycle signals had to be used, this caused some problems as the required control logic only works if the signals are assigned before they are being tested. As it could not be ensured that the signal containing the data to be stored would be assigned before the testing, variables were used. These had to be assigned in the previous clock cycle which was not too difficult, as knowing if something will be sent is simpler than knowing what will be sent.
- The Node Store had a similar problem with obtaining the function and argument, however the destinations could use a channel as there was no control logic due to there only being one input to chose from.
- Being able to stall the pipeline if the lists and buffers were becoming full caused some problems with the signal used in the pipeline stages. If the stages were stalled when the list or buffer became full some of the information was lost, however if the pipeline stages were fed Nop and continued to run the processor operated correctly.
- Timing of the hardware design was found to affect the output with minor adjustments in the design causing tokens to be gained or lost in different sections of the processor. The cause of these problems was very difficult to locate as the simulator in Handel-C was not affected by them and there were very few debugging features added to the implementation, which would have affected the design anyway. Another possibility was some of the data stored in the external ram was being corrupted depending on the design being loaded onto the FPGA.

7 Operations

7.1 Compiling the test programs

To operate the processor the data-flow graphs had to be compiled into machine code. This process was done in two steps to allow the program to be checked between the steps. The first step compiled an i2 representation of the directed graph into a list of node descriptions and tokens. The program was written in i2 as this higher-level language allows functions to be written and reused as well as being easier to follow. Normally the data-flow machine would execute this directly however due to the restraints on the implementation a simple parser was used to convert this into serial code.

7.2 Description of the test programs

There were two test programs implemented: a modeling of a small heated 2-d mesh (Appendix M), and a filter (Appendix O). These programs were executed on a control-flow processor as well as the data-flow processor in order to obtain a performance comparison.

The mesh contained 9-cells being heated from the top, cooled from the bottom and insulated on the sides. This mesh is continually heated until the center cell has a constant temperature for a set number of iterations. The reason for having multiple iterations at the same temperature was to ensure the cell had reached equilibrium as the increase in temperature becomes very small around this temperature. This mesh program should execute in approximately the same time for the two architectures as the programs have approximately the same number of instructions due to the limited branching factor on the data-flow program, the same clock speed, the data-flow processor has the blocking node primed to allow multiple iterations of the same value and branch predictions is enabled on the control-flow architecture. Taking all of these considerations into account the data-flow processor should perform as well as the control-flow processor. The reason for this is the mesh is very small thus allowing the values to be stored in registers in the control-flow processor, this alleviates the need to load and store data from cache therefore avoiding most of the data dependencies. Having the values in memory also reduces the number of instructions required to update the old values, thus the advantage of the data-flow processor keeping the values for each iteration separate will not be apparent. The control instructions to ensure the processors do the same number of iterations will only be slightly smaller in the data-flow design and the advantage of the data-flow processor continuing to execute nodes while the branching is being evaluated will probably not be obvious due to the branch prediction being used in the control-flow processor.

The filter was an integrator consisting of a multiplication node, an add node and a feedback path consisting of a shift down node. This graph was fed with a list of tokens generating a step response. As a directed graph this program only contained four instructions compared to the seven instructions for the control-flow implementation written in C, thus the data-flow program was assured of executing faster. The reason the data-flow machine had so few instructions as there was no branching required because the output would be stopped when there was no input, instead of checking if there is no input then breaking out of the loop.

7.3 Performance comparison of the two architectures

With the mesh the data-flow architecture was found to perform slightly slower than the control-flow architecture taking 125 clock cycles compared to 77 per iteration. The main reasons for this was that all of the advantages a data-flow architecture has over the control-flow architecture were not tested in this program, also about 50% of the nodes in the directed graph were dyadic and as this implementation of the CSIRAC II architecture sends all of the tokens through the Bypass, the Execution Unit was only executing two-thirds of the time. However if the control-flow architecture had to store and retrieve the values from memory the number of clock cycles would increase by 54 per iteration causing it to execute slower than the data-flow implementation, this is assuming each memory access only takes one clock cycle which is very generous.

The filter also had better performance on the control-flow architecture. However this was expected as the data-flow architecture reads the tokens from the external RAM which takes 16 clock cycles thus the processor would have computed the result before the next values was added to the Input List. As this did not give a fair comparison due to the hardware limitations dictating the processors performance, the loading section of the program was rewritten (Appendix L) so these tokens were fired multiple times instead of multiple instances of them being read from memory. With this alteration the data-flow architecture's performance was found to exceed that of the control-flow taking an average of five clock cycles compared to the seven clock cycles per input value. The reason the data-flow processor is able to output an almost continuous stream of results is the proceeding values beginning execution before the previous values has been fully executed.

8 Conclusions

A subset of the CSIRAC II data-flow architecture was successfully implemented on an Altera Cyclone FPGA. However there were some limitations on the design mainly due to the large amount of memory required and the configuration of the available RAM. One of these restrictions was that the program, due to its size, had to be stored in external RAM. This increased the number of clock cycles required to load the node definitions by a factor of 16 as the data bus was only 8-bits. This also starved the processor while executing the filter program, as the previous token would have completed execution before the processor had retrieved the next token from the external RAM. The effect of this could have been reduced by using another device to implement the external network thus allowing a larger data bus to be used. Alternatively the token size could have been reduced however this would cause the processor to differ from the specifications of the CSIRAC II architecture. The other limiting factor causing the clock speed to be restricted to 50MHz was the number LEs available on the FPGA. This caused compromises to be made in the design, thus limiting possible performance improvements and reducing the performance gain obtained from the scalability of a data-flow architecture.

When comparing the performance of the processor to a control-flow architecture it was found that the performance advantages of the architecture were only apparent in some of the programs being executed on the processor. This was due to the size restrictions on the directed graphs being executed, thus the control-flow architecture was able to use registers for all of its data storage. However if the program was scaled up the control-flow architecture would have to access memory, drastically decreasing

its performance, while the data-flow architecture would only be affected by the increase in the number of instructions being executed. The program that performed better on the data-flow architecture was the filter. This was an expected result as the filter requires a constant stream of data to be processed simultaneously which is possible in the data-flow architecture as another iteration of the graph can begin execution before the previous finishes. Also, unlike the control-flow architecture, no control logic is required, thus reducing the number of instructions being executed from seven down to four.

8.1 Further work

Some possible improvements to the processor to improve its performance would be:

- Move the retrieval of the function and literal closer to the Execution Unit so the dyadic tokens do not have to go through the Bypass thus increasing the average throughput of the Execution Unit if more tokens are entering from the Input List.
- Distribute the tokens to both the Local and Input List, thus allow a larger branching factor to be used and therefore reducing the number of replicates used in the programs.
- Allow the Local List to send tokens to the Bypass and the Matching Store simultaneously.
- Collect the initialize tokens from another source so a larger word size can be used or alternatively have a different clock for this section to ensure the maximum clock speed to collect the tokens.
- Implement the architecture on a larger FPGA thus allowing it to be scaled-up, therefore the performance will be comparable to the control-flow architecture. Also this would allow the implementation of the design alteration that were previously disregarded because of the size restrictions of the FPGA.

9 Acknowledgments

I would like to acknowledge Prof. G. Egan for his support and guidance in developing this implementation of the CSIRAC II architecture. Mr. A Sloan for the supplying of compilers, a parser and an emulator. Dr. A. Price, Mr. T. Cornall, and Mr. T. Ramdas for their technical assistance and suggestion in overcoming some of the problem encountered during the project. Miss. N. Brammer for her support and assistance with editing this report.

10 References

[1] J. Silc, B. Robic, T. Ungerer. Processor Architecture: From Dataflow to Superscalar and Beyond. Springer-Verlag Berlin Heidelbery, 1999.

[2] Gaudiot, J-L. & Bic, L., Advanced topics in data-flow computing, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[3] Egan, G.K. (2005), personal communication.

[4] Egan, G.K. (1990), *The CSIRAC II Data-flow Computer: Token and Node Definitions*.,Technical Report 31-001, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology.

[5] Impulse. Impulse-C User Guide. Impulse accelerated technologies.

[6] M. Bowen. Handle-C Language Reference Manual. Embedded Solutions

[7] Altera Corporation (2003), Section I. Cyclone FPGA Family Data Sheet, October 2003

Appendix A. Arch_marco.hcc

Appendix B. Archdataflow.h

Appendix C. Archdataflow.hcc

Appendix D. bypass.hcc

Appendix E. dist.hcc

Appendix F. execution_unit.hcc

Appendix G. input_list.hcc

Appendix H. local_list.hcc

Appendix I. matching_store.hcc

Appendix J. Node_store.hcc

Appendix K. program.h

Appendix L. archdataflow.hcc edited for filter program

Appendix M. mesh.i2

Appendix N. mesh.dfo

Appendix O. filter.i2

Appendix P. filter.dfo

Appendix Q. Schematic

Appendix R. Quartus compilation Summary

Appendix S. Conference paper