

MASTER

Optimization of inter processor buffers using regular location assignment.

van Bladel, F.M.A.M.

Award date:
1993

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

**Optimization of inter processor buffers
using regular location assignment**

By : F.M.A.M van Bladel

Master thesis on a project at the Philips Research Laboratories in Eindhoven under supervision of:

Prof. dr. ing J.A.G. Jess

Coach (university) : Dr. ir. J. van Eijndhoven
Coach (Philips) : Dr. ir. J.L. van Meerbergen
Coach (Philips) : Ir. P. Lippens

The department of electrical engineering of the Eindhoven university of technology does not accept any responsibility for the contents of student reports and master theses.

Abstract

Looking at a VLSI chip one can distinguish different kinds of recourses. There are arithmetic- buffer- and control units and interconnection. The arithmetic units can be designed in a very short time with the use of a silicon compiler like PIRAMID or PHIDEO. During the design of these units little attention is payed to the communication problems that may arise between different arithmetic units. To solve these problems inter processor buffers (IPBs) are used. These IPBs can be designed with tools like ESPA or MATCHBOX. This report focuses on the techniques used by the tool MATCHBOX.

The techniques used by MATCHBOX are presented and compared with a new technique called regular placement. This regular placement technique aims at reducing the area used for address generation in the IPB. This is done by assigning locations to samples in such a way that the address sequences become regular.

For different kind of applications IPBs are generated with regular placement and the techniques from MATCHBOX. It appeared that regular placement did not result in a significant reduction in area costs, when the IPB was designed for only one application. For these IPBs the techniques and target architectures of MATCHBOX will cover the design space very well. But when IPBs are designed to handle more than one application regular placement is a good technique for reducing the size of this merged IPB.

Contents

1	Introduction	3
2	MATCHBOX	7
2.1	Absolute location assignment	9
2.2	Counter addressing	12
2.3	Relative location assignment	14
2.4	Target architectures	16
2.4.1	The counter architecture	16
2.4.2	The address table architecture	17
2.4.3	The delta table architecture	18
2.4.4	The run length delta table architecture	20
3	Regular placement	23
3.1	The solution space of regular placement	25
3.1.1	Solution space exploration	27
4	Test applications	35
4.1	Matrix transposition	35
4.2	Zig-zag transformation	36
4.3	Spiral left turning	37
4.4	Radix 2 Fast Fourier Transformation	37
4.5	Radix 4 Fast Fourier Transformation	38
5	Resulting IPBs	39
5.1	Results Matchbox	39
5.1.1	The memory sizes	39
5.1.2	The address generators	39
5.1.3	The total cost of the inter processor buffers	40
5.2	Results regular placement	41
6	Merging of IPBs	45
6.1	Results MATCHBOX and regular placement	45
6.2	Merging of N applications	47
7	Conclusions and recommendations	51

A	PIF	53
A.1	PIF description of matrix transposition	53
A.2	PIF description of 8×8 zig-zag transformation	54
A.3	PIF description of 4×4 left turning spiral	56
A.4	PIF description of radix 2 FFT	57
A.5	PIF description of radix 4 FFT	58
B	Regular placement program	59
C	Matchbox memory sizes	63
D	Matchbox address generator sizes	65
E	Result regular placement	69
F	architecture merged IPB	71
G	Unix script	73

Chapter 1

Introduction

Looking at a VLSI chip different units which perform different tasks can be distinguished.

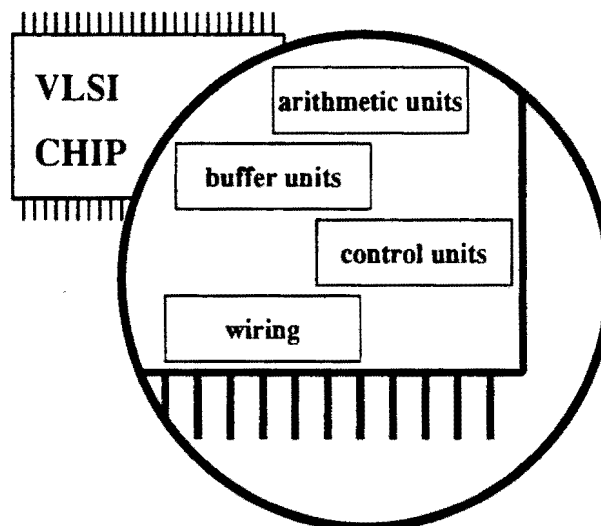


Figure 1.1: The different units on a VLSI chip

The arithmetic units together with the buffer units make up the data path which is controlled by the control units. The wiring is used to connect the different units together. The arithmetic units are the producers and consumers of the data. With the use of a silicon compiler arithmetic units can be designed in a very short time. For applications with a clock to sample rate ratio between 1 and 20 (high throughput) the silicon compiler PHIDEO [2] is used. For applications with such a ratio of 1000 or more (low throughput) PIRAMID [10] is used. Complex systems however need more than one of these arithmetic units on one chip. During the design of these units, PHIDEO and PIRAMID pay little or no attention to the communication problems that may arise between these different arithmetic units. To make the communication between two arithmetic units or between arithmetic units and outside world possible, inter processor buffers (IPBs) are used. These

IPBs can be designed with tools like **ESPA** and **MATCHBOX**. **ESPA** is used to design IPBs for arithmetic units which are designed with **PIRAMID**. **MATCHBOX** is used for the design of IPBs which have to solve the communication problems between arithmetic units of **PHIDEO**.

Because the arithmetic units are designed before the IPBs are designed it is known in what order and on which timepoints data is produced or consumed. An IPB can be seen as a black box between two arithmetic units which input is a certain stream of data (samples) and which output is delayed version of that input. Where each sample can have a different delay.

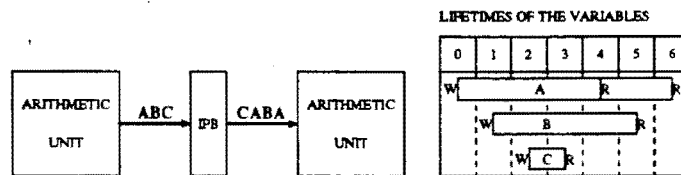


Figure 1.2: The inter processor buffer with its in and outputs

A sample can only be produced once but can be consumed multiple times. The lifetime of a sample is the time between its production and its last consumption. During this lifetime it has to be stored in the memory of the inter processor buffer.

Figure 1.3 shows an example of an architecture at the chip level.

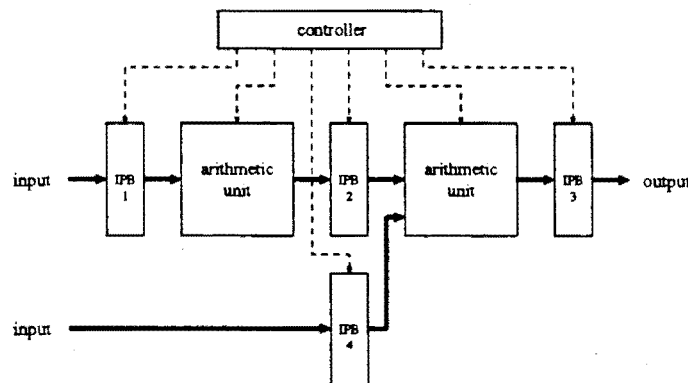


Figure 1.3: multi arithmetic unit overview

As can be seen in Figure 1.3, IPBs can be needed quite often on a chip. That is why it is relevant to reduce the size of these individual IPBs [3] [4] [5] [8] [9]. When an arithmetic unit is capable of executing different applications, a different IPB is necessary for each application. Therefore merging different IPBs to one IPB which is capable of coping with different applications is also an issue which needs attention.

An inter processor buffer consists of a memory block and address generators (Figure 1.4). In most of the cases two address generators are needed, one for

the read accesses and one for write accesses. But in some cases these two address generators can be merged together to one address generator which generates the addresses both for the read and write accesses.

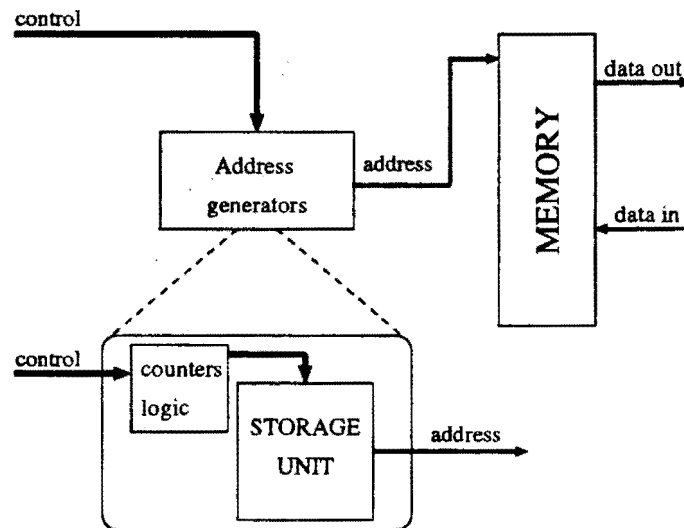


Figure 1.4: The inter processor buffer

The address generators consist of counters, some logic and a storage unit. This storage unit can be a PLA a ROM or a standard cell implementation. In this storage unit the information of which address has to be generated is stored. Several different strategies exist to store this information. One can store the actual addresses, the differences between consecutive addresses, or the runlength of rows of the same deltas and the delta size. Which way of storing the addresses, and what kind of storage unit is used, will be determined by the resulting size of the address generator. The smallest solution will be selected.

But before the problem of address generation is under discussion we first have to determine a place for all the samples in the memory in such a way that no samples with overlapping lifetimes are stored in the same memory location.

The research has been focused primary on MATCHBOX. All the techniques of MATCHBOX optimize either the memory cost or the addressing hardware cost. But as a result of optimizing the one, the other has unpredictable cost. These unpredictable cost turns out to be very high in some cases. The goal is, to find a technique which makes a good trade off between memory cost and addressing cost. An address generator will be relatively small when the addresses it has to generate are in some way regular. To achieve that the samples have to be placed in the memory in a regular way. This has to be done without to much extra memory cost, otherwise the cost of the total IPB will be too high.

The search for regularity in a address sequence can be done at different levels. For example one can try to find regularity at the bit level of the address words and aim to generate these bits with a counter, and try to re-use the bits of this counter as much as possible [6]. Or one can try to find regularity at the word

level and try to generate the entire address word with a counter or an other architecture. Both techniques are a subject of research at the Philips Nat. Lab. This paper will deal with the search for regularity at the word level, and with the consequences for memory and address generator size.

Another goal of the research was to develop a flexible address generator. The ultimate flexible solution is a programmable solution. In this way the behaviour of the inter processor buffer can be changed by changing the program. The other extreme in this view is the dedicated single application IPB. In between these two extremes lies merging the IPB for a few applications. By making it possible to use an inter processor buffer for multiple problems the chip area that is used by these buffers can decrease because of the reuse of hardware resources. All these different IPB have been subject of investigation.

In chapter 2 the techniques of MATCHBOX will be discussed. In chapter 3 the technique of regular placement will be explained and the different techniques which are used to increase the quality of the results are discussed. Also some techniques to reduce the computation time are presented here. In chapter 4 the applications which are use in the tests are presented. In chapter 5 the results of MATCHBOX and regular placement will be presented and the solutions will be compared. In chapter 6 the merging of different application into one IPB will be discussed. In chapter 7 conclusions will be drawn and some recommendations will be provided.

Chapter 2

MATCHBOX

PHIDEO is a silicon compiler targeted at the design of high performance real time systems with high sampling frequencies such as HDTV. It supports the complete design trajectory starting from a high level specification all the way down to layout. At a certain point in this design trajectory the production and consumption time points of samples and their source and destination arithmetic units are known. The problem is now to synthesize an architecture so that the area of memories, address generators and interconnection hardware is minimized. This task can be divided into two sub-tasks.

1. memory allocation \Rightarrow MEDEA.
2. location assignment and address generation \Rightarrow MATCHBOX.

The first step is memory allocation. Here the decision is made which samples share the same memory. After this step it is exactly known how many memories will be used and how the memories are connected to the arithmetic units. The remaining problem is now to store the samples in the memories in such a way that the size of the IPBs is minimized. The place of the samples in the memory and the timepoints at which they are produced and consumed determine the sequence of addresses that has to be generated. The list of timepoints and addresses is called an address schedule. To determine an address schedule the data schedule is used. A data schedule is a list of read and write timepoints of the different samples.

Framelength = 5		
sample	Write timepoint	Read timepoint
A	0	6
B	1	5
C	2	9
D	3	8
E	4	7

Table 2.1: A data schedule

All applications which are used during Digital Signal Processing (DSP) have a repetitive nature. That means that the data schedule repeats after a certain time. One repetition of the data schedule is called a *frame*. The time one repetition takes is called the *frame length*. It is possible that a sample is read in a later frame than the one in which it is written. Then it is said that the sample crosses the *frame boundary*. With the use of the information from the data schedule a lifetime diagram of the samples can be constructed. The lifetime diagram of the samples for one repetition derived from the data schedule in Table 2.1 is shown in Figure 2.1.

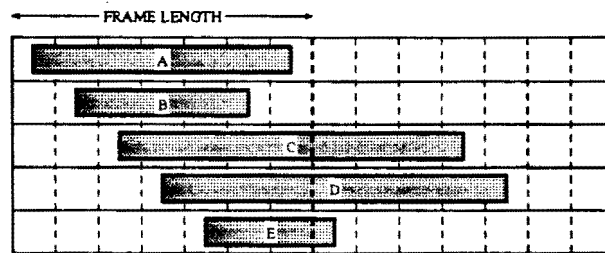


Figure 2.1: Lifetime diagram for one repetition

But the data schedule in Table 2.1 is repetitive and it also contains samples which cross the frame boundary. That means that during one frame not only the samples of the current frame can be alive but also samples from previous frames. The samples from previous frames are called *delayed versions* of the current samples. This is indicated with a '@1' suffix. The number in this suffix indicates how many frames ago the sample started living. Figure 2.2 shows the lifetimes during several frames.

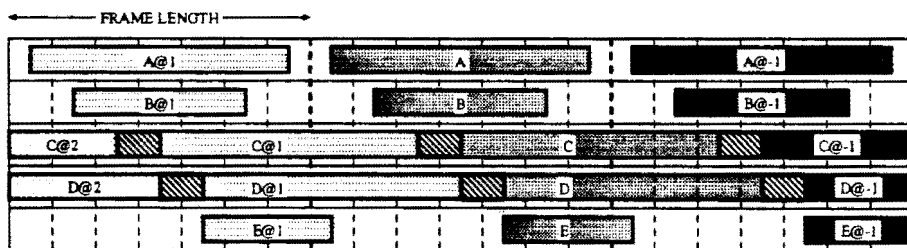


Figure 2.2: Lifetimes of the samples during several frames

It can be seen that the pattern of lifetimes is the same every frame. So to characterise the application it is enough to know the lifetimes during one frame. The lifetimes during one frame can be seen in Figure 2.3.

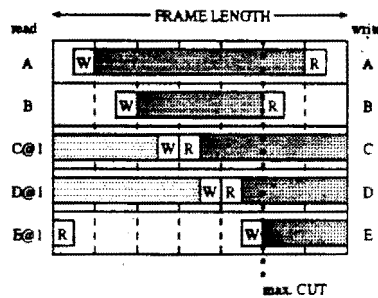


Figure 2.3: Lifetimes of the samples during one frame

This lifetime diagram is used to determine the address schedule. When two samples are never alive at the same time they can share the same memory location. For the lifetime diagram that means that samples with non-overlapping lifetimes can share the same location. When it is tried to store two samples with overlapping lifetimes in the same location a *lifetime clash* occurs. An address schedule which contains lifetime clashes is called 'not valid'. From the lifetime diagram the minimum memory size which is necessary to store all the samples can be derived. The minimum memory size equals the maximum cut of alive samples in the life time diagram. This can be explained as follows. At the timepoint where the cut is maximal, there are for example 'X' samples alive. Because all are alive at this timepoint they can not share the same memory location. So at this point in time 'X' memory locations are necessary. This lower bound of the memory size will be called M_{lowb} . For the lifetime diagram from Figure 2.3 the maximum number of samples which are alive simultaneously is five. A maximum cut is indicated with the dotted line.

In the search for a valid address schedule which is cheap in area costs PHIDEO evaluates three different techniques. These three techniques will be explained in the following sections.

2.1 Absolute location assignment

Absolute location assignment is an assignment of memory locations to the samples of a data schedule in such a way that each version of a sample is written into the same memory location every frame. So after applying absolute location assignment every sample has its own absolute address which stays the same every frame. That is the reason why this technique is called absolute location assignment. The location of a sample is independent of the frame. This is indicated in Equation 2.1 where $location(s, f)$ represents a function which assigns a location to sample s during frame f . $L(s)$ represents a location assignment function which only depends on s .

$$location(s, f) = L(s) \quad (2.1)$$

Problems occur when there are samples which lifetime is longer than the framelength. For example samples 'C', 'D' in Figure 2.1. In this case it is impossible to come up with a valid schedule. This is because of the restriction that a sample has to be written to the same memory location every frame. Using this restriction, sample C and its delayed version C@1 have to be stored in the same location, the same yields for sample D. But sample 'C@1' and 'D@1' are not read yet when 'C' and 'D' have to be stored again. A lifetime clash exists between 'C' and 'C@1' and between 'D' and 'D@1'. So it is not possible to store 'C@1' and 'C' or 'D@1' and 'D' in the same location. To solve this problem the frame length is expanded. By expanding the frame length the new framelength becomes an integer times the old framelength. The samples which are starting to live during this new frame will be treated as different samples. So before expansion 'A' and 'A@1' where the same samples only A@1 is a delayed version of A. After expansion they are treated as different samples. This is indicated by changing the names. For sample A this is shown in Table 2.2 for an expansion of two.

old sample name	new sample name
A	A'
A@1	A''
A@2	A'''@1
A@3	A''''@1

Table 2.2: Sample names before and after expansion

The number of expansions necessary equals the number of frames in which the longest living sample is alive. This is shown in Equation 2.2.

$$\text{Number of expansions} = \left\lceil \frac{\text{longest lifetime}}{\text{framelength}} \right\rceil \quad (2.2)$$

For our example from Figure 2.3 an expansion of two will be enough. The resulting data schedule after expansion can be seen in fig 2.4.

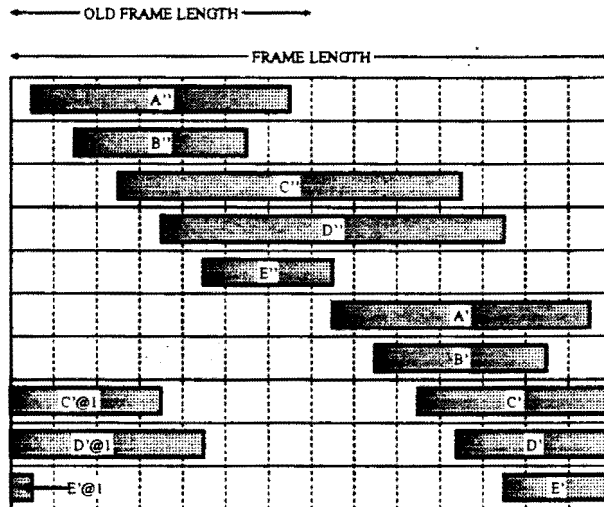


Figure 2.4: Frame expansion for absolute location assignment

By expanding the frame length the characteristics of the application are not changed. Only the constraint that the address sequence has to be repetitive after one frame is relaxed by increasing the framelength. Now the address sequence only has to be repetitive after N old frames, with N the number of expansions. With this new lifetime diagram it is possible to find a valid schedule. The problem that samples clash with their delayed versions is solved now, because there are no samples left which lifetime is longer than the framelength. PHIDEO uses graph colouring techniques to solve the problem of placing the samples without life time clashes in an as small as possible memory. The graph that has to be coloured is called a conflict graph. In this graph the different vertices are the samples. An edge between two vertices exists, when the two samples have overlapping life times. When the resulting graph is an interval graph, than the left edge algorithm will be used, which runs in $O(n \log n)$ time [7] for n samples. This will result in a minimal number of colours thus a memory size equal to M_{lowb} . When the graph is not an interval graph but a circular arc graph left edge cannot be used. Colouring a circular arc graph is known to be a NP-complete problem [11]. Consequently no polynomial time algorithm is known that solves this problem. Furthermore Tucker [12] proved that an optimal result M_{circ} , found by an exhaustive algorithm for example can be far from the lower bound M_{lowb} . In [12] Tucker shows circular arc graphs for which

$$M_{circ} = 2 \times M_{lowb} - 1 \quad (2.3)$$

From the lifetime diagram can be derived whether the conflict graph will be an interval graph or a circular arc graph. If there is a timepoint during a frame at which no sample is alive than the resulting graph will be an interval graph and the left edge algorithm will be used to colour the graph. In the life time diagram of Figure 2.4 there is no timepoint on which no sample is alive so graph colouring with heuristics is used. This leads to a placement like in Figure 2.5.

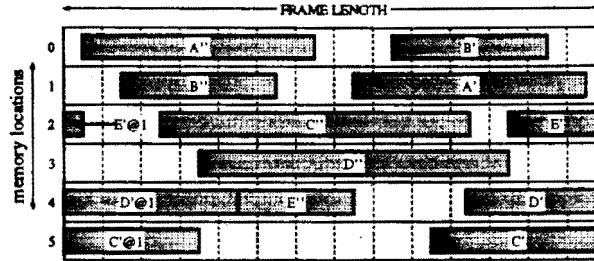


Figure 2.5: absolute location assignment resulting from graph colouring

The schedule is valid now but the complexity of the address sequence is not taken into account. The resulting addressing sequences can be seen in Table 2.3

Time (clock cycle)	Write location	Read location
0	0	2
1	1	X
2	2	X
3	3	5
4	4	4
5	X	1
6	X	0
7	1	4
8	0	X
9	5	X
10	4	2
11	2	3
12	X	0
13	X	1

Table 2.3: Addressing sequences absolute location assignment

In Table 2.3 the address sequences for one frame are shown. These are the only sequences which have to be generated. The address sequences during the other frames are a repetition of these sequences. The memory needed in this case is larger than the minimum. When the frame has to be expanded the addressing sequences are longer than the number of samples. Also by using graph colouring no attention is paid to how difficult it will be to generate these sequences. So although absolute location assignment aims at an as small as possible memory the cost of the total inter processor buffer can turn out to be high because of excessive cost of address generation.

2.2 Counter addressing

Counter addressing is a kind of absolute location assignment. So the restriction that the delayed versions of a sample are stored in the same memory location every frame still holds. The difference is that the samples are not placed with graph colouring techniques but in such a way that the write address sequence can

be generated with a counter. Because counter addressing is based on absolute addressing it is possible that the frame has to be expanded. Aiming at a counter solution can sometimes result in a large memory, as shown in Figure 2.6 for the application from Figure 2.2.

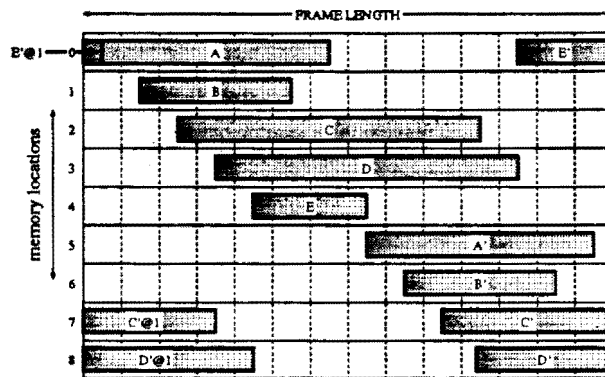


Figure 2.6: absolute location assignment for counter addressing

From Table 2.4 it follows that the write addressing sequence can be generated with a counter which is very cheap. But the read sequence is still as hard to generate as with absolute location assignment.

Time (clock. cycle)	Write location	Read location
0	0	0
1	1	X
2	2	X
3	3	7
4	4	8
5	X	1
6	X	0
7	5	4
8	6	X
9	7	X
10	8	2
11	0	3
12	X	6
13	X	5

Table 2.4: Addressing sequences counter addressing

Because the lifetime diagram consist of a two times expanded frame, the address sequence will still be twice as long as the number of samples. For the write address generator this is not really a problem because the counter address generator will stay quite cheap. But for the read address generator holds the same as for the address generators following from absolute location assignment. The addressing hardware of the inter processor buffer generated with counter addressing will consist of a cheap write address generator and a read address generator

which has cost comparable to the one generated with absolute addressing. But because the memory needed for this counter technique is large in a lot of cases, the total cost of the inter processor buffer can be high due to large memory cost.

2.3 Relative location assignment

In this case a pointer technique is used. This pointer is incremented every frame. Relative to this pointer position the location of the different samples is the same. This means that the absolute location of a sample in the memory is dependent on the frame. This technique is only possible if the location calculations are executed using modulo arithmetic. The absolute location of a sample s in frame f is defined as :

$$location(s, f) = (P(f) + R(s)) \bmod M \quad (2.4)$$

In this equation $P(f)$ stands for the base location (pointer) which is updated every frame. $R(s)$ stands for the relative location of the sample. This location is relative to the base location and is independent of the frame. M is the size of the memory. The pointer mechanism solves the problem that samples clash with their own delayed versions, because in every frame the samples will be stored in different absolute locations. In Figure 2.7 the result of relative location assignment for the application of Figure 2.3 can be seen. To illustrate the principle three frames are shown. One can see in Figure 2.7 that because of the changing base pointer the samples rotate through the memory.

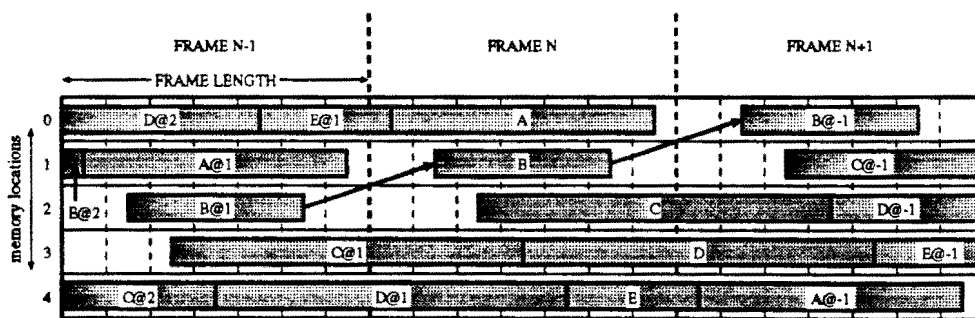


Figure 2.7: Result of relative location assignment

An interesting property is that it can be proven that a memory size of at most $M_{lowb} + 1$ is necessary. To see this, one has to realize that there are two equivalent ways to look at the problem. Assuming that M memory locations are available.

- Select one frame (e.g frame 0) and discuss the location that is assigned to every sample.

- Select on location (e.g location 0) and look over M frames to discuss the frame at which every sample is stored.

The role of time and place in the two cases is interchanged. These two points of view contain the same information because the base location is updated (decremented) every frame. When it is determined in which frame a sample is assigned to location zero than the location of that same sample in frame zero can be calculated. To demonstrate this the place of the samples in the memory during several frames is shown in Figure 2.8.

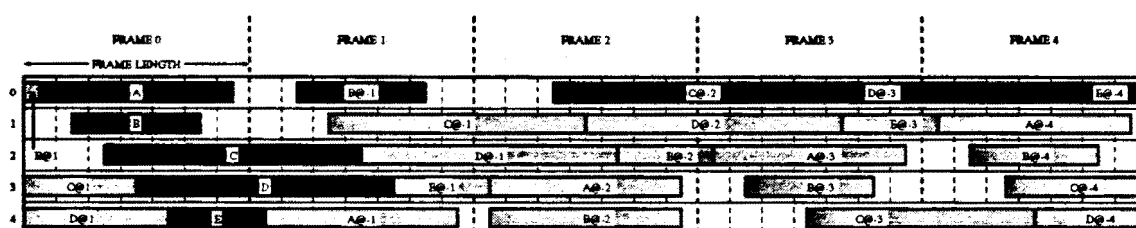


Figure 2.8: Result of relative location assignment for several frames

From Figure 2.8 it can be concluded that the two viewpoints are indeed identical. So it is enough to concentrate on location zero and schedule the samples in a efficient way in this location. This can be done by using the first fit algorithm. The first fit technique tries to place the samples as close as possible together in memory location zero. So the minimum amount of frames (memory locations) is needed to store the samples. To come up with a placement with the variables as close together as possible the algorithm starts with the sample with the earliest write timepoint (sample A) followed by the sample with a write timepoint the closest to the read timepoint of sample A in this case sample B and so on until all samples are placed. More information about relative location assignment can be found in [1]. The advantage of relative location assignment is that the memory cost will be low, as the memory needed is at most $M_{lowb} + 1$. Also the address sequences that have to be generated during one frame are never longer than the number of variables as is shown in Table 2.5. The address sequences during other frames can easily be derived from these sequences, because they are just shifted one or more places.

Time clock cycle	Write location	Read location
0	0	0
1	1	X
2	2	X
3	3	3
4	4	4
5	X	1
6	X	0

Table 2.5: Addressing sequences relative location assignment for frame 0

On the other hand it is possible that the address sequence is so complex and irregular that it leads to expensive address generators. Also, extra cost (in comparison to absolute and counter addressing) will be introduced by the modulo hardware. Due to this, the overall cost of the IPB with relative location assignment can be higher than when using counter or absolute location assignment.

The kind of architecture which is used for address generation can also influence the size of the address generators. Therefore the target architectures of MATCHBOX will be discussed.

2.4 Target architectures

To generate the different address sequences MATCHBOX has certain target architectures at its disposal. These architectures are :

- counter
- address table
- delta table
- run-length delta table

Each of these architectures has its own way of storing the information which is necessary to generate the address sequence. Only the counter architecture has no storage, it can only generate consecutive addresses like 0,1,2,3,4,5 etc. The address table architecture has all the actual addresses stored in its storage unit. This storage unit is addressed with a counter. The Delta table architecture has only the differences between the different addresses stored in its storage unit. The actual addresses are calculated from the previous address and the delta. In the run length delta architecture the storage unit is used to store the number of the same consecutive deltas (*the run length*) and the size of the delta. The actual addresses are again calculated from the previous addresses and the deltas. In the following sections the different architectures will be discussed in more detail.

2.4.1 The counter architecture

The counter architecture can only be used when the samples are placed in the memory in such a way that they can be addressed with a counter. So when the goal is to use a counter architecture as an address generator, it should be taken into account during the assigning of locations to the different samples. It is always possible to place the samples in such a way in the memory that the write address generator can be realized by a counter. The counter architecture is not complex and therefore very cheap and can be seen in Figure 2.9

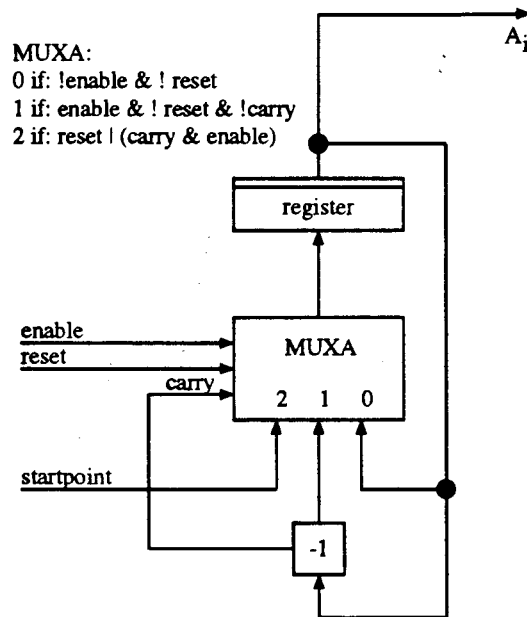


Figure 2.9: counter architecture

The architecture shown here is a down counter. it counts like:

$M - 1, M - 2, \dots, 0, M - 1, M - 2 \dots$

The architecture consists of a decrementor, a block with switch logic and a register. The quantity *start point*, which is equal to the memory size minus one, is known at compile time and can thus be stored locally. So the only control signals which have to be generated by the controller, are *reset* and *enable*. The down counter is preset every time address zero is generated. This can be easily implemented using the carry flag of the decrementor. The enable signal is used to obtain the next address. If the enable signal is low the register will hold.

2.4.2 The address table architecture

When the samples are placed in the memory and the address sequences show no regularity at the word level, than MATCHBOX will choose for an address look up table. This method also uses a counter, but the output of this counter is now used as the entry of a table in which the addresses are stored. The hardware is shown in Figure 2.10. The table can be implemented with a ROM but when the addresses show some modulo two regularity, at the bit level, it may be cheaper to replace the ROM with a PLA. Because it than possible that the PLA can be reduced at lot. The counter is the same as used with counter addressing. The output of the table is latched in a register. The hardwired signal *start point* determines the address of the first table entry. This architecture can only be used for absolute location assignment because for relative location assignment the actual addresses

will be different every frame. For small examples the address table may be a cheap solution but when the problems become larger the address look up table can become large and expensive. In that case the other architectures may offer a cheaper solution.

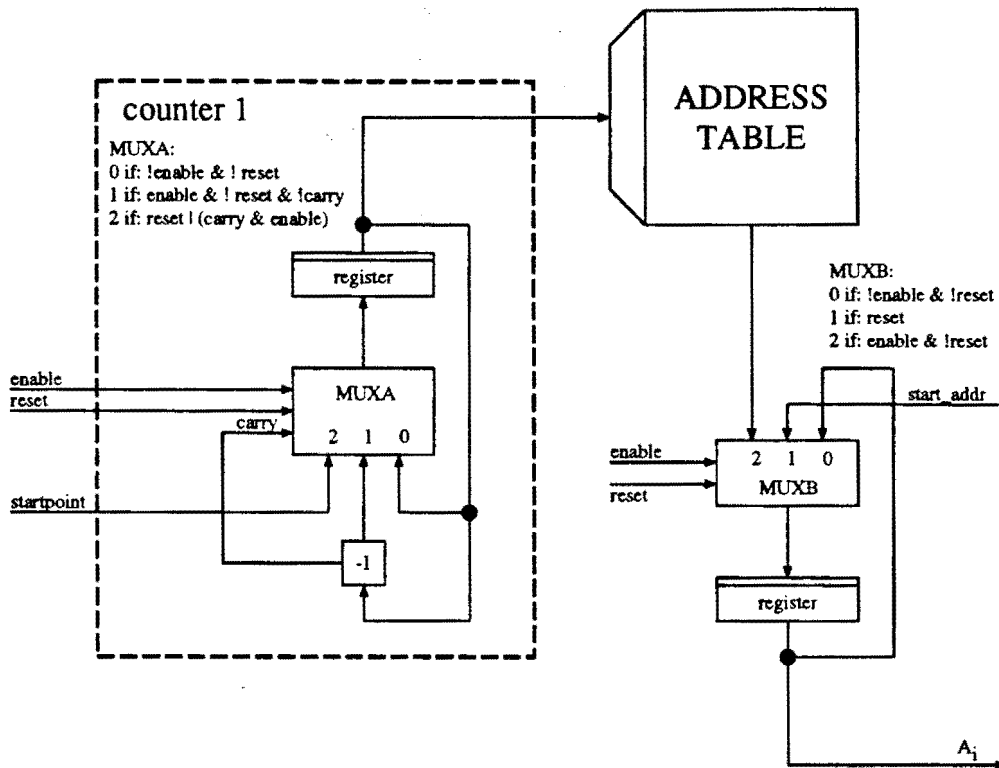


Figure 2.10: Address table architecture

2.4.3 The delta table architecture

Another way to generate addresses is to store only the differences between two subsequent addresses. The advantage of this is that the word width of the largest delta will be smaller than the word width of the largest address. So the storage unit requires a smaller word with. The number of entries in the PLA will be one less than in an address table. On the other hand the hardware that is needed to calculate the addresses will introduce extra costs.

The architecture in case no modulo hardware is necessary (absolute location assignment) is shown in Figure 2.11.

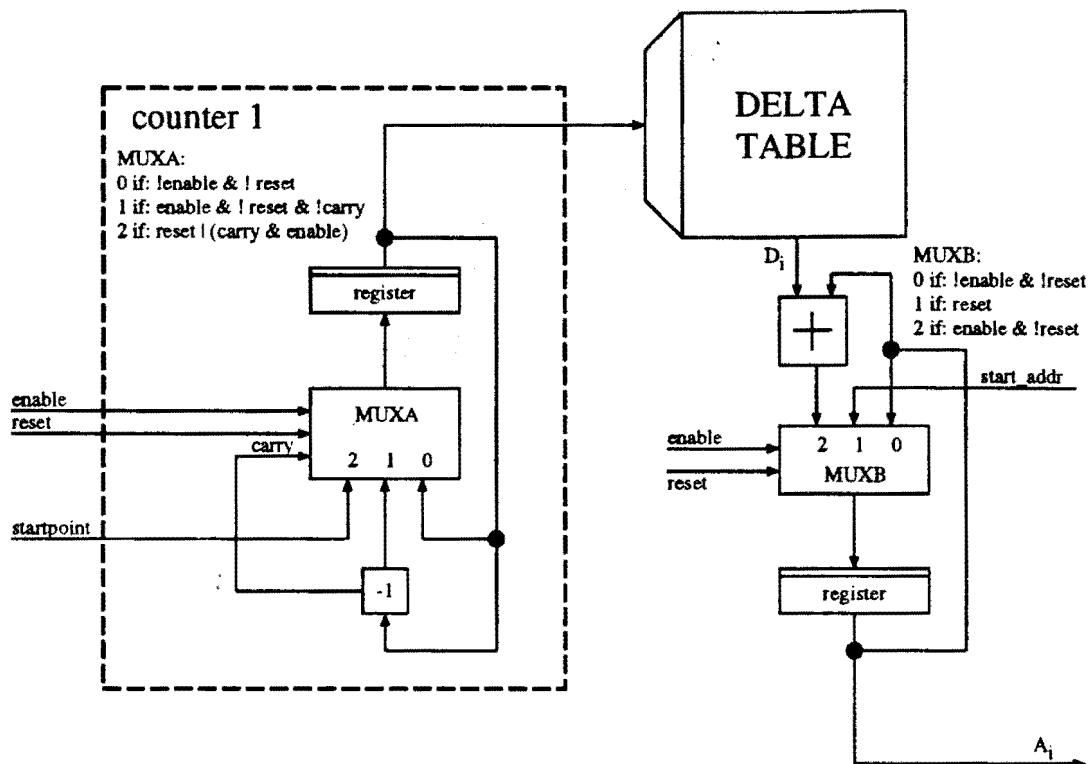


Figure 2.11: Delta table address generator without modulo hardware

An adder is needed to add the difference to the previous address. This result is stored in a register. The enable signal is used to step to the next address. A reset is given at least once to synchronize the read and write address generator. The delta table architecture can also be used to generate the addresses which result from relative location assignment. The architecture has to be modified slightly for it. The differences (D_i) between the subsequent addresses can be calculated as follows:

$$D_i = (loc(s_i, f) - loc(s_{i-1}, f)) \bmod M \quad (2.5)$$

In this equation f is the frame number, s_i is a sample that has to be written or read at timepoint i , s_{i-1} is a sample that has to be written or read on timepoint $i-1$. loc is a function which assigns a location to a sample in a particular frame. So $loc(s_i, f)$ and $loc(s_{i-1}, f)$ are two consecutive addresses in time. The value M equals the memory size, this modulo memory size operation is needed to exclude negative deltas.

By using Equation 2.3 it follows

$$D_i = ((P(f) + R(v_i)) \bmod M - (P(f) + R(v_{i-1})) \bmod M) \bmod M \quad (2.6)$$

$$D_i = (R(v_i) - R(v_{i-1})) \bmod M \quad (2.7)$$

In Equation 2.7 can be seen that because of the pointer mechanism D_i is independent of the frame number. From equation 2.7 it also follows that modulo hardware is needed to calculate the differences. So to calculate an address from a delta and a previous address it will also be necessary to use a modulo operation. This modulo architecture can be seen in fig 2.12. The cost of this modulo hardware is an extra cost which can be saved by using absolute location assignment. It depends on the problem what turns out to be cheaper, the modulo hardware or the frame expansion.

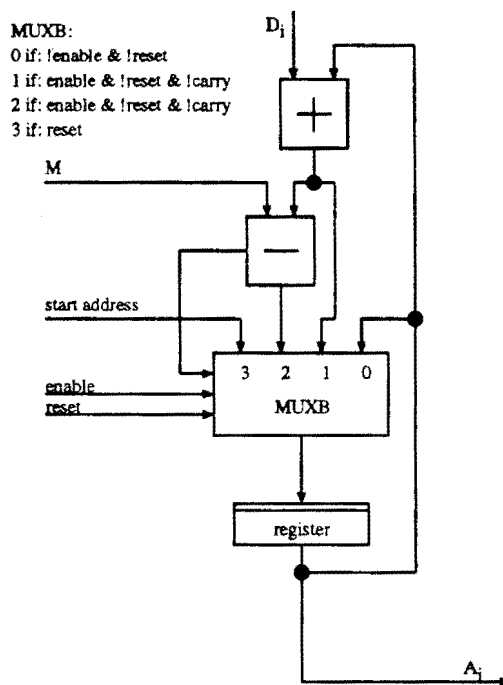


Figure 2.12: Modulo hardware for relative location assignment

The implementation of the pointer mechanism can be done in two ways. The first is to implement the last delta one smaller than it actually should be. In this case it not allowed to give more than one reset. The reset is only used to synchronize the read and write address generator. The other way is to change the start address every frame. In this case every frame a reset should be given. Changing the start address every frame will be expensive in hardware, but implementing a smaller last delta can lead to synchronisation problems between read and write address generator.

2.4.4 The run length delta table architecture

When the delta sequence contains a long series of constant delta's the possibility exists to store the runlength and the delta. This can be achieved with the architecture shown in Figure 2.13. For this architecture holds the same as for the

delta table address generator. It is shown for absolute location assignment but it can also be used for relative location assignment by adding the modulo hardware.

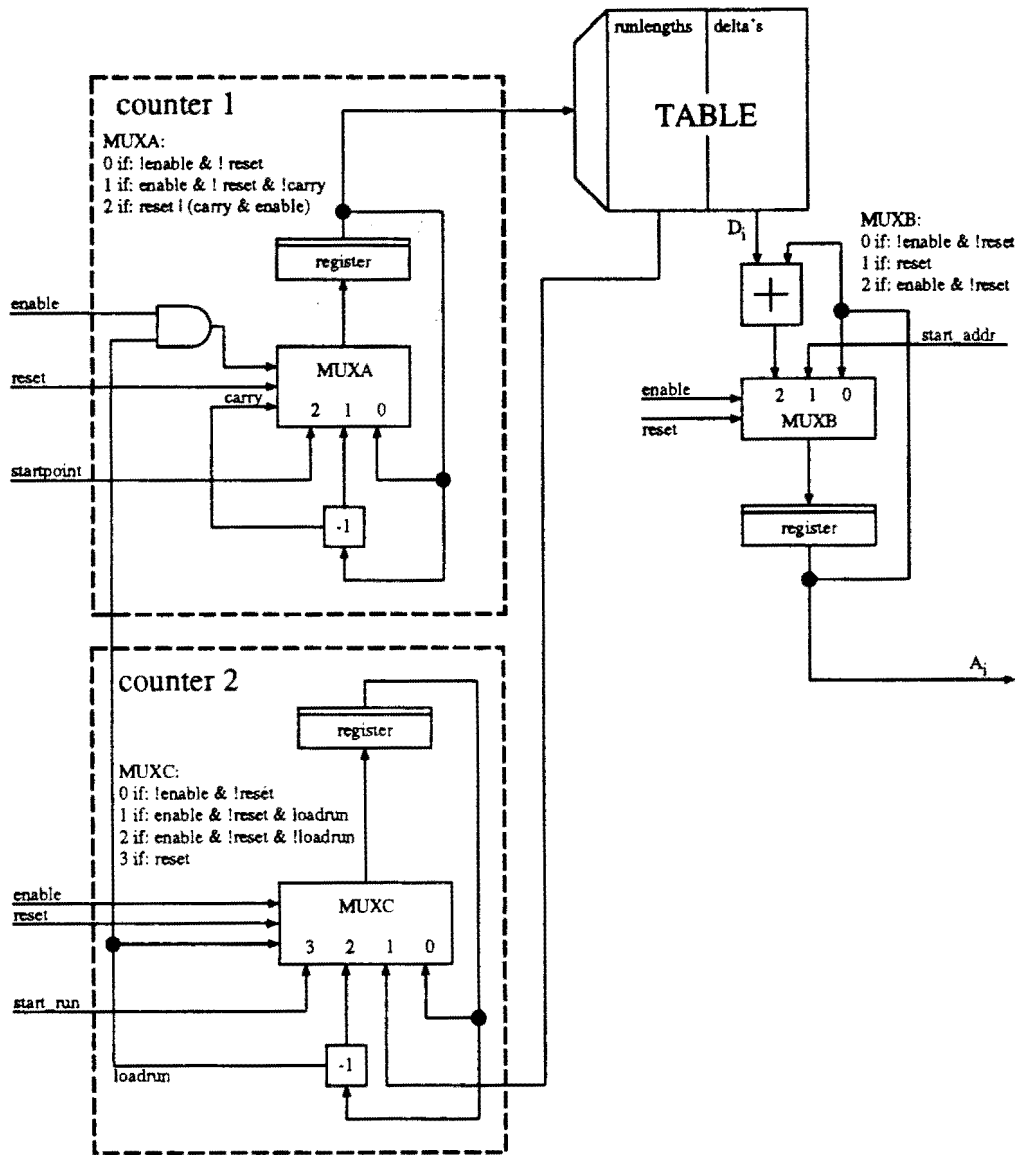


Figure 2.13: The run length delta architecture without modulo hardware

This architecture needs some explanation because its operation is not as obvious as it seems. Counter two counts down the run length for every difference. When the carry out of this counter becomes active counter one is decremented and the next run length is loaded into counter two. *Start_run* is a hard wired signal which determines the first run length. This signal is necessary because the first run length from the table is loaded into counter two only after the first difference has been counted down. So the first run length in the table belongs to the second difference. This means that run length and difference are shifted one place in the table. The carry from counter two is generated with a delay of one

runlength. That is why all the run lengths in the table are one smaller than their actual value. Further should be mentioned that this architecture has a very large overhead of hardware. So will this run length delta architecture be the smallest and thus the cheapest solution the storage unit must be very small in comparison to the storage units in the other architectures.

Chapter 3

Regular placement

Looking at the methods MATCHBOX uses for location assignment, one can see that the used techniques aim at minimizing either the memory cost or the addressing cost. The counter technique which tries to minimize the addressing cost, sometimes turns out to be very expensive in memory cost. The other techniques, absolute location assignment and relative location assignment, which aim at small memory costs have unpredictable address generator costs. The total costs of an inter processor buffer can be separated into three parts, Equation 3.1.

$$\begin{array}{r} \textit{Read Address generator cost} \\ \textit{Write Address generator cost} \\ + \\ \hline \textit{Memory cost} \\ \hline \textit{Total cost} \end{array} \quad (3.1)$$

Predicting the total costs of the inter processor buffers resulting from the different techniques appeared to be very hard. There are only a few things which are predictable and which are not problem dependent. For relative location assignment it is known that the memory cost will be close to the minimum (M_{lowb} or $M_{lowb} + 1$). On the other hand the costs of the address generators can not be estimated accurately a priori. This addressing cost is dependent on the application for which the IPB is designed, but is also strongly influenced by the unpredictable effect of the minimization of the storage unit. It is known that the cost of the address generators will be larger than the cost of counter architecture. By using the counter technique there is chosen for minimal cost for one address generator (the write address generator). The memory size which is necessary for realizing counter addressing is dependent of the application for which the IPB is designed. But even when the application is known it is not possible to make a good estimation of the memory size that is needed to realize counter addressing. Although only a few things are known about the design space it is still possible to draw an address generator- memory- cost graph. This graph contains the cost of the write address generator, on the X-axis, and the cost of the memory size, on the Y-axis. The reason why only one address generator is taken into account for the design space cost graph is that when one address generator is defined,

the other address generator will be the logical consequence of the defined address generator and the application. So when there is something to gain in total cost it has to be done by decreasing the cost one of the two address generators while keeping the memory as small as possible. Minimizing both address generators is impossible. The graph shown in Figure 3.1 is a representation of the costs of one address generator and the required memory cost for one application.

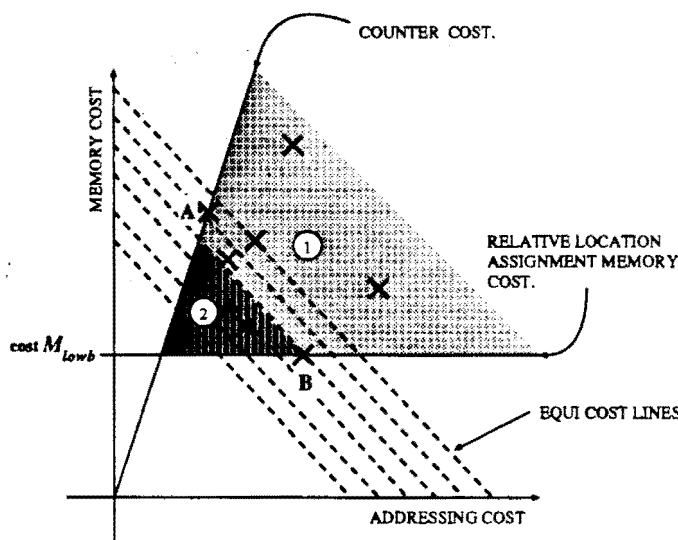


Figure 3.1: Design cost space

The line which represents the cost of a counter in Figure 3.1 is slanted, this is because a larger memory needs a larger counter to address it. By increasing the size of the counter the area cost will also increase. The "counter addressing line" also represents the absolute minimum costs for which it is possible to realize an address generator. So on the left hand side of this line there exist no address generator. The minimum memory line is constant in the entire solution space. This minimum is application dependent, and can be calculated when the lifetimes of the samples are known. Below this line there exist no memory size in which it is possible to store all the samples. The marks represent the cost of the address generator and the memory as a result of using different kind of techniques for memory allocation. Mark *A* represents the cost of a counter solution and mark *B* represents the cost of a relative solution. The memory cost of relative addressing is placed on the cost of M_{lowb} but in reality the memory needed with relative location assignment can be one more than M_{lowb} so the cost can be a little higher. Absolute location assignment with graph colouring can be represented by one of the other marks somewhere in the two grey areas. The question whether absolute location assignment is cheaper than the cheapest of one of the two extremes (in area two in Figure 3.1) or more expensive (in area one in Figure 3.1) is impossible to answer a priori because this is dependent on many factors, like the heuristic of the graph colouring algorithm, the application for which the IPB has to be

designed and the unpredictable effect of the minimisation of the storage unit of the address generator. In developing a technique for location assignment the goal was to find another technique which results in a inter processor buffer with low costs. So the cost should be somewhere in area two of Figure 3.1. On the other hand the flexibility should be kept in mind. Knowing all this it is tried to optimize the address generator size within such limits that the memory would not grow too much. Minimizing the address generator costs is done by assigning memory locations to samples in such a way that the addressing sequence will become regular. This is called regular placement.

The architecture which is best suited to generate these regular sequences is the run length delta architecture. For the address table and the delta table architecture, the storage pattern of the samples will not have a predictable influence on the size of the address generator. The number of entries in the storage unit, in which the addresses or the deltas are stored, stays the same, so a big difference was not expected. In practice it appeared that the size of these address generators did differ a lot between one storage pattern and the other. This was due to the fact that with one storage pattern the storage unit can be reduced more than with another, but the amount of reduction is unpredictable. On the other hand the number of entries in the storage unit of the run length delta architecture will decrease when the sequence becomes more regular. So the address generator may become smaller when the address sequence becomes more regular. But the costs of an inter processor buffer consist of more than the cost of one address generator. The cost of the other address generator is unpredictable and the cost of the memory is an uncertain factor too. The question now is, will this method result in a inter processor buffer which is smaller than the other techniques and is it possible to turn the resulting IPB into a flexible one. Before answering these questions let us first look into the method of regular placement.

3.1 The solution space of regular placement

The definition of a regular addressing sequence is an important issue. Because regularity can have many different forms. In the previous section it is already mentioned that the size of the run length delta architecture will decrease when the address schedule is more regular. That is why this architecture is used to determine a measure for regularity. There are numerous solutions which can have the label regular solution. Three examples of a regular solution for 25 samples can be seen in Figure 3.2.

SOLUTION 1 #mem=23 (rel.)		SOLUTION 2 #mem=28 (rel.)		SOLUTION 3 #mem=33 (rel.)	
ADDRESS	DELTA	ADDRESS	DELTA	ADDRESS	DELTA
0	3	0	5	0	29
3	3	5	5	29	29
6	3	10	5	25	29
9	3	15	5	21	29
12	3	20	5	17	29
15	3	25	5	13	29
18	3	2	5	9	29
21	3	7	5	5	29
1	3	12	5	1	29
4	3	17	5	30	29
7	3	12	5	25	29
10	3	6	5	22	29
13	3	11	5	18	29
16	3	16	5	14	29
19	3	21	5	10	29
22	3	26	5	6	29
2	21	3	5	2	29
0	15	8	5	31	29
15	15	13	5	27	29
7	15	18	5	23	29
22	15	23	5	19	29
14	15	0	5	15	29
3	12	5	5	11	29
18	15	25	5	7	29
10	15	20	5	3	29
	11	22	5		28



 = Regular delta sequence
 = Start address

Figure 3.2: Several regular solutions

As can be seen in Figure 3.2 a sequence consists of a start address followed by a part with a constant delta. The number of consecutive and identical deltas plus one is called the length of a sequence. The "plus one" is due to the delta to go to the start address of the sequence. In the run length delta storage unit one regular sequence will result in two entries, one with run length of one, for the delta to go to the start address of the sequence and one with a runlength equal to the number of identical deltas. A solution with maximum regularity will be the solution with one delta to go to start address and a list of the same deltas to go to the other addresses, like solution three in Figure 3.2. These kind of solutions usually tend to need a large memory. To be able to reduce the memory size one can try to find a solution with less regularity which needs a smaller memory. That is done by allowing more shorter regular sequences. This will result in solutions like one and two in Figure 3.2. These solutions are generated by setting a maximum and minimum allowed sequence length. But there are restrictions on these maximum and minimum sequence lengths and on the delta sizes. For the sequence length it is obvious that there exist no sequence length of zero and no sequence length longer than the number of samples. For the delta there is only a maximum, namely the memory size. A delta larger than the memory size or a negative delta can, with a modulo operation, always be represented by a delta in between zero and the memory size - 1. When we don't take the lifetime clashes into account there are a lot of solutions possible. All combinations of sequence lengths and deltas are allowed as long as Equation 3.2 holds.

$$\sum_{i=0}^{\#seq-1} \text{sequence length}_i = \text{number of samples} \quad (3.2)$$

When we do take the lifetimes of the samples into account some of solutions for which Equation 3.2 holds will not be valid schedules because of lifetime clashes.

But still a lot of possibilities remain.

3.1.1 Solution space exploration

Now it is known what the solution space looks like the next problem is to find an efficient way to explore this space. The groups of sequence length delta combinations which result in a valid schedule are a sub set of the solution space defined by Equation 3.2. To find a group which belongs to this sub set a few parameters can be manipulated. These parameters are:

- The memory size.

This can be varied between M_{lowb} , the size needed for relative addressing, and a maximum value on which there is no restriction. But for the maximum memory size one has to keep in mind that the cost of the inter processor buffer can become unacceptably high when the memory size is too large. That is why before increasing the memory size, all possible sequence length and delta size combinations are tested in order to find a solution which fits in a small memory.

- The sequence length.

This parameter can change from the number of samples, which is the maximum, till a minimum value below which the sequence is not called regular any more. This minimum value is three. This minimum needs some explanation. As mentioned a sequence length consists of a delta to go to the start address followed by a number of equal deltas. Because the delta to go to the start address is usually not the same as the deltas of the regular sequence the number of the same deltas in the regular sequence should at least be two, otherwise one cannot speak of regularity. So the minimal number of consecutive and identical deltas is two. For the sequence length we add one, from the delta to go to the start address. So the minimal sequence length is three.

- Delta size

This parameter can change from zero, which means that two consecutive samples are stored in the same memory location, till the memory size - 1.

- The start position.

This parameter determines the place where the regular sequence starts. This can be any place in the memory. therefore the maximum is the memory size - 1 and the minimum is zero.

To find an optimal solution the parameters are changed in a particular order. Starting with a fixed memory size, it is tried to place the samples without lifetime clashes by changing the delta size and the sequence lengths for the different regular

sequences. When all the different possibilities of delta size and sequence lengths are tested and none of them led to a valid address schedule then the memory size is increased.

Optimizing regularity for a given memory size

Optimizing regularity is actually finding a solution with as few as possible changes in its deltas. The optimal regular solution is of course the one with no changes in the deltas. But that is not always possible to realize in a given memory size. When its not possible to place all the samples in one sequence with a certain delta there two possible ways to solve this problem. The first one is, keep trying to place the samples in one sequence but with another delta. The second one is allow more regular sequences. This is done by first placing an as long as possible sequence. Then try to place the remaining unplaced samples in a regular way. All these shorter sequences are tested again for different delta sizes. An intermediated multi sequence solution is shown in Table 3.1. In this table N stands for the total number of samples.

	Placed samples			# Unplaced samples
Seq. number	seq. 1	seq. 2	seq. 3	N-L1-L2-L3
Δ	$\Delta 1$	$\Delta 2$	$\Delta 3$	
Seq. length	L 1	L 2	L 3	

Table 3.1: intermediate multi- sequence solution

Sequence one is placed first and contains as many as possible samples for that delta. When the length is larger than the minimum the sequence is accepted. The length of the sequences is limited by the lifetime clashes. For the example from Table 3.1, at a certain point three sequences are placed and the remaining samples can not be placed regular without lifetime clashes, for any sequence length delta combination. Two strategies are possible to solve this problem.

The first one is, decrease the length of sequence three and try to place the remaining samples. When the minimum sequence length of sequence three is reached the delta is increased. When no valid placement is found for any sequence length delta combination of sequence three the sequence length of sequence two is decreased and so until a valid placement is found. When the maximum delta of sequence one is reached it is impossible to place the samples in a regular way in the given memory size. Then the memory size is increased. How this scheme works for one regular sequence is shown in Figure 3.3.

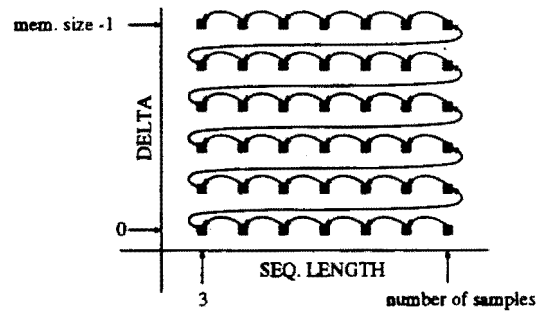


Figure 3.3: Search scheme : decrease sequence length first

In reality the sequence lengths are not always of maximum length. This is because it is possible that a certain sequence length delta combination does not exist because of lifetime clashes. When a sequence has to be checked for validity the samples of that sequence are placed in a imaginary memory. Then the schedule is checked on lifetime clashes. As an example we take five samples which have to be placed in a memory with three places, and we determine how many place and remove operations have to be performed when all the sequence length delta combinations have to be checked for the first sequence. This can be seen in Table 3.2.

test no.	seq. length	delta size	# place operations	# remove operations
1	5	0	5	0
2	4	0	0	1
3	3	0	0	1
4	5	1	5	0
5	4	1	0	1
6	3	1	0	1
7	5	2	5	0
8	4	2	0	1
9	3	2	0	1
		total	15	6

Table 3.2: indication of number of place and remove operations

This amount of place and remove operations is used to compare this search scheme with the second possible search scheme, which will be discussed next.

The other possible search scheme is that the sequence length is kept constant while the delta is changed first. This is also done in order to make it possible to place the remaining samples. When no valid placement is found, all sequences with maximum length minus one are tested for all different deltas and so on until the minimum sequence lengths are reached or a valid placement is found. This search scheme for one regular sequence can be seen in Figure 3.4.

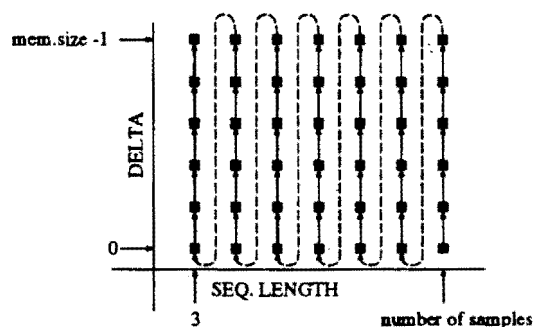


Figure 3.4: Search scheme : change delta first

The major difference between these schemes is the computation time they need. In the search scheme from Figure 3.4 the old sequence has to be removed every time a new delta has to be checked. For the example with five samples and three memory place this can be seen in Table 3.3

test no.	seq. length	delta size	# place operations	# remove operations
1	5	0	5	0
2	5	1	5	5
3	5	2	5	5
4	4	0	4	5
5	4	1	4	4
6	4	2	4	4
7	3	0	3	4
8	3	1	3	3
9	3	2	3	3
		total	36	33

Table 3.3: indication of number of place and remove operations

When a sequence does not result in a valid addressing schedule, the sequence has to be removed and a new sequence with a different delta has to be placed again. This leads to a lot more place and remove operations than in the scheme of Figure 3.3 as can be seen in Table 3.3. So the scheme from Figure 3.4 need much more computation time than the one from Figure 3.3. The difference appeared to be so big that only the search scheme from Figure 3.3 is implemented in the program.

The maximum and minimum sequence length are not fixed numbers. They can be varied by the user in order to find a regular solution which fits the problem. So the maximum allowed sequence length does not have to be the number of samples and the minimum does not have to be three. Setting the minimum to three is even not advisable because computation times will become very long. The smaller the gap between maximum and minimum sequence length is, the faster the program will come up with a valid address schedule. By manipulating these numbers the user can determine the amount of regularity in the addressing sequence. By increasing the values, regularity will increase but also the needed

memory will increase. By decreasing the values, computation times will increase and the memory needed will decrease but also the amount of regularity in the addressing sequence will decrease. It is also possible to give a minimum sequence length which is not fixed during the execution the algorithm. The first value of the minimum is given by the user. After this it is checked how many samples are left unplaced. For these remaining samples the maximum sequence length is set to the number of remaining samples and the minimum is set to the half of the number of remaining samples. So during every attempt of placing samples with a certain delta at least half of the amount of samples must be placed before this delta sequence length combination is accepted.

The search for sequence length and delta combinations should be rather fast because such a search has to be done for a lot of different memory sizes. Although the program used the fastest search scheme for determining sequence length delta combinations, it can still be time consuming. Because of the long computation times the third parameter, the start position of the sequence, is kept constant. For the start position the first fitting position is taken. This means that the start address of a regular sequence is determined by the first position on which the first sample of the sequence fits. The other measure that is taken to increase computation times is that it is tried to minimize the number of memory sizes which have to be tested.

Optimizing the memory size

As mentioned, the buffer sizes which have to be tested lay in between the memory size from relative addressing, which is the absolute minimum, and infinity. Infinity is not a realistic memory size and that is why the maximum memory size has to be given by the user. A very rough estimate for this maximum can be derived by checking how many frames the longest living sample is alive and multiply this value with the number of samples, or one can take the memory size that counter addressing needs. For a certain memory size there are two possibilities. It is possible to find a valid address schedule with a certain sequence length and delta combination or it is not. So the solution space can graphically be represented like in Figure 3.5.

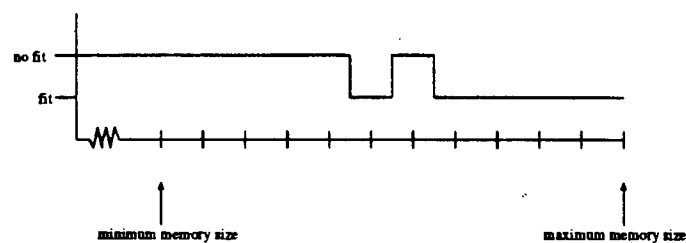


Figure 3.5: Solution space memory size

This solution space can be tested linearly from the minimum memory size

until a fitting memory size is found. But when the distance between the minimum memory size until the solution memory size is large, computation times will be very long. One big advantage of this way of searching is that it will always result in the smallest possible solution, as can be seen in Figure 3.6.

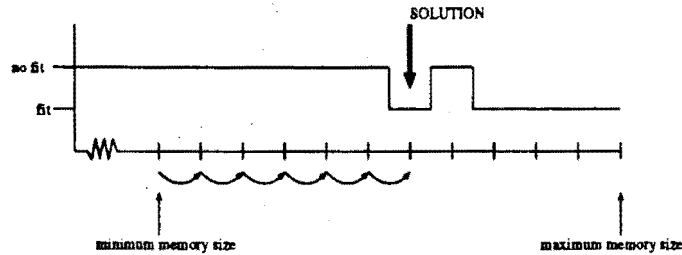


Figure 3.6: linear search through memory sizes

An other way of searching is the binary search. It starts with a maximum and a minimum value for the memory size. It checks if the maximum fits, if not, it stops and assumes there is no solution. Then it checks the minimum value, if it fits, then that is the solution. In the case that the minimum doesn't fit and the maximum does, the binary search starts. The next memory size to test follows from Equation 3.3

$$\text{new memory size} = \lfloor \frac{1}{2}(\text{max. memory size} - \text{min. memory size}) \rfloor \quad (3.3)$$

If there exist a valid solution for this memory size, than this memory size will become the new maximum. If there exist no valid solution than this memory size will become the new minimum. This will continue until the maximum value equals the minimum. A draw back of this way of searching is that it will not always ,like in Figure 3.7, result in the smallest memory size.

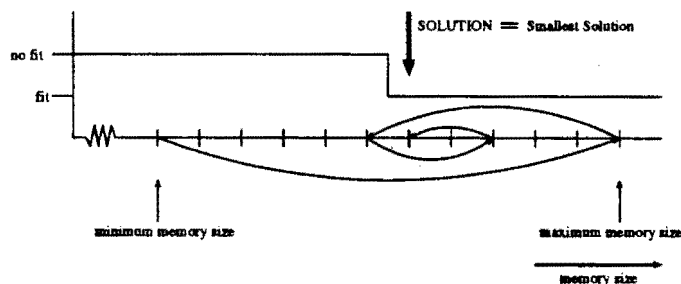


Figure 3.7: binary search through memory sizes resulting in a minimal memory size

The outcome is depending on how the function of the solution space look like. When it is like in Figure 3.8 the result will not be the minimal memory size.

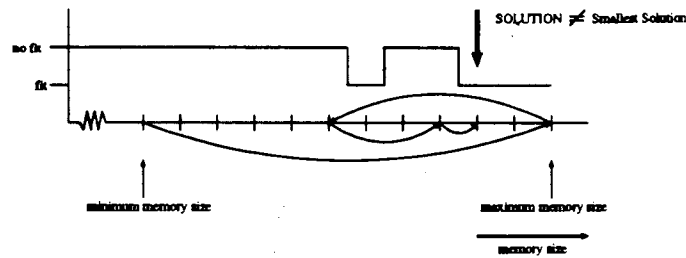


Figure 3.8: Binary search through memory sizes not resulting in minimal memory size

From the numerous tests that have been performed it appeared that the solution found with binary search didn't differ that much from the one found with linearly search. The search for the minimal memory should be done as follows:

First find an indication of the memory size with binary search, than try to determine whether it is an absolute minimum with linear search.

An overview of all the possible settings, the syntax of the input file and the several output files of the regular placement program can be found in appendix B.

Chapter 4

Test applications

To be able to check whether regular placement results in smaller solutions than other methods of placement a few test problems are selected. The selected applications are data format conversions which can occur in many different sizes. The main characteristics of these applications are treated shortly in the following sections.

In these sections only one size of the problems is presented in order to clarify the read and write orders. During the actual tests a lot of different sizes are used. The high level descriptions of the applications in *Phideo Input Format* (PIF) can be seen in appendix A.

4.1 Matrix transposition

The problem of matrix transposition is defined as follows. The samples are produced row by row so the inter processor buffer has to store these samples in a row by row order. The samples have to be consumed column by column so the read address generator has to produce addresses which achieve this. This kind of transformation occurs for example in video compression applications using 2D transforms such as the 2D discrete cosine transform. When the memory is seen as a matrix like structure the read and write order look like in Figure 4.1 The arrows indicate the order in which the samples, which are numbered from zero till 25, have to be read and written.

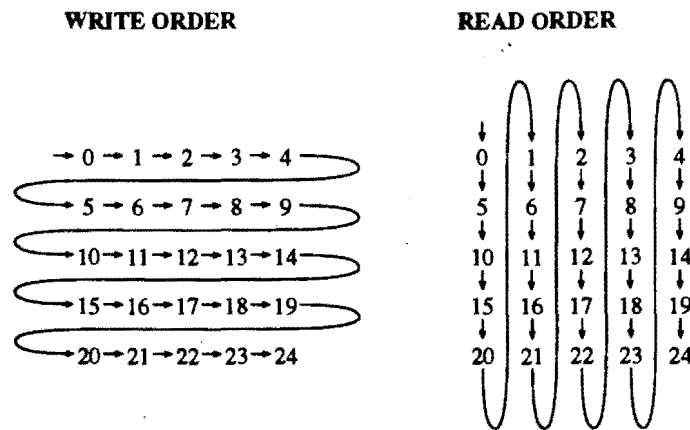


Figure 4.1: Matrix transposition

From the read and write order follows the earliest timepoint on which the read action can start, assuming that every clock cycle a read or write action occurs. In this case that is clock cycle 16, so the skew between write and read is 16 clock cycles. When the read action is delayed 16 clock cycles than sample 20 is written, and in the same clock cycle read again.

4.2 Zig-zag transformation

In this section we treat the conversion of a line by line scanned input frame to a zig - zag scanned frame. The zig - zag scanned sequence is useful in image transmission after Huffman coding [8] or after Discrete cosine transformation. The samples are produced row by row and have to be read in a zig - zag pattern. How the read and write orders look like can be seen in Figure 4.2.

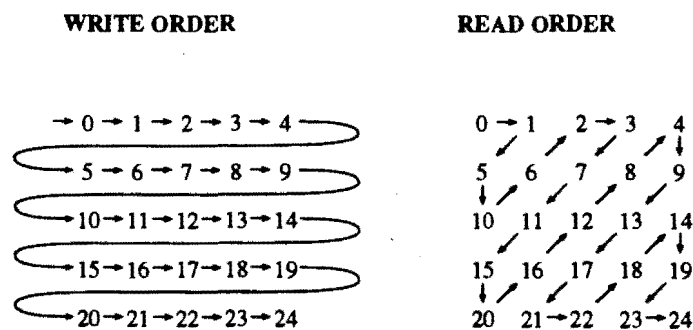


Figure 4.2: Zig - zag transformation

The minimal skew for this problem is nine clock cycles, again with the assumption that every clock cycle a read and write action occurs. For this kind of transformations only square examples are used because that what is typically needed in practice.

4.3 Spiral left turning

The left turning spiral is also a conversion used in video applications. It is specially needed during region growing and edge detection. The sample are produced row by row and have to be read in a spiral pattern like in Figure 4.3.

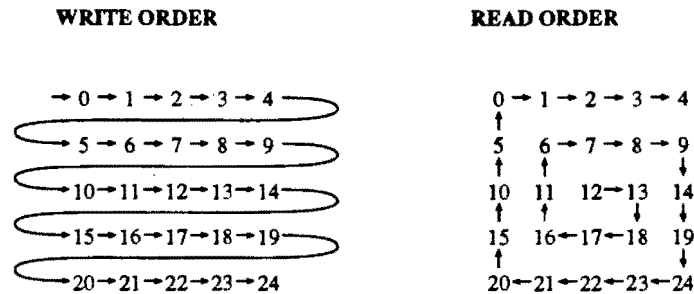


Figure 4.3: Left spiral transformation

The minimal skew for this problem is 15 clock cycles, assuming a read or write action every clock cycle. From this problem also only square examples are used.

4.4 Radix 2 Fast Fourier Transformation

The Fast Fourier Transformation used in the tests is a radix 2 constant geometry FFT. With this geometry the butterfly outputs are not put back on the place where they come from. That is why it is also called 'not in place'. The indexing is kept constant from stage to stage in this way a flexible high level description for different sizes of FFT is easier to make (see appendix A). The signal flow graph is shown in Figure 4.4.

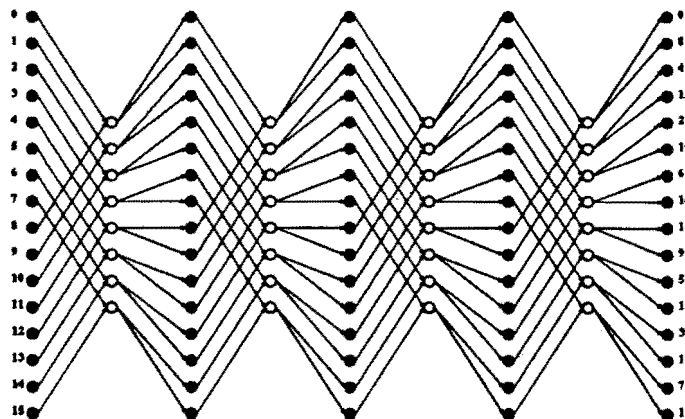


Figure 4.4: Constant geometry, radix 2, 16 points, not in place, FFT

The numbers at the inputs and outputs are the indices associated with the

different samples.

4.5 Radix 4 Fast Fourier Transformation

For the radix 4 FFT applies the same as for the radix 2 FFT, it is also a constant geometry FFT. The difference is that not two but four samples at the same time are used in the computation. The signal flow graph is shown in Figure 4.5

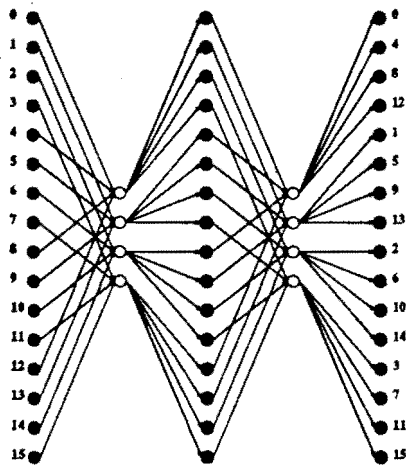


Figure 4.5: radix 4, 16 points, not in place, FFT

Chapter 5

Resulting IPBs

For different sizes of the test problems presented in chapter 4, IPBs are generated. The results of MATCHBOX using the three different techniques are compared with each other and with the technique of regular placement. All the areas presented in the chapter and in the appendices are estimates based on a 1.5 μm technology. The areas are all given in mm^2 .

5.1 Results Matchbox

The results from MATCHBOX will be presented in three parts. First the memory sizes needed in the IPBs, for the different techniques will be discussed. Then The chosen architecture for the address generator and the number of entries in the storage unit, before and after minimisation will be shown. After this the overall smallest IPB will be extracted from these results.

5.1.1 The memory sizes

As mentioned in the chapter about MATCHBOX relative addressing always comes up with a minimal or almost minimal memory size. But in some cases absolute addressing needs the same or only slightly more memory space. The actual memory sizes for the different techniques are shown in appendix C. From these tables it can be seen that the size of memory needed for counter addressing is always relatively large in comparison with the other techniques.

5.1.2 The address generators

For the address generators there are several architectures possible. For all the different techniques the sizes of the different architectures are shown in appendix D. The empty places in these tables indicate that that architecture is not relevant or not possible for that technique. For the storage unit of the address generator a PLA is chosen in all cases. The number of entries of these PLAs before and after minimization is shown. It can be seen that the difference between the PLA before

and after minimisation can be rather big, especially the PLA which contains deltas. Further can be concluded from this table, that counter addressing will always offer the smallest write address generator. Also can be seen that aiming for a counter on the write side barely influences the size of the address generator on the read side. The other address generators can also be relatively small especially the address table architecture. This is especially the case for applications with only a few samples. In these cases the address table is not very expensive, and will the small overhead cost of the address table architecture result in a cheap address generator. When the number of samples increases, the address table will become more expensive because of rapidly growing cost of the address table. From the results can be seen that in general the address table cannot be minimized as well as the delta table.

5.1.3 The total cost of the inter processor buffers

Because of the number of techniques and the number of architectures a lot of different inter processor buffers can be generated. From all the architectures the smallest solution is taken and put in a graph, in order to be able to make a good comparison. The architectures that are used in these smallest IPBs can be found in appendix D. The graphs can be seen in Figure 5.1 for matrix transposition and in Figure 5.2 for radix 2 FFT. The graphs of these two applications show the main characteristics of the different techniques, so for the other applications no graphs are generated. In the percentage graphs the percentage of the area used by address generation and memory can be seen. In Figure 5.1 and Figure 5.2 the *R* stands for relative location assignment the *A* for absolute location assignment and the *C* for counter addressing. These graphs show that for relative and absolute addressing a rather big part of the area is used for address generation. With counter addressing it is the other way around, in this case the mayor part of the area is taken by the memory.

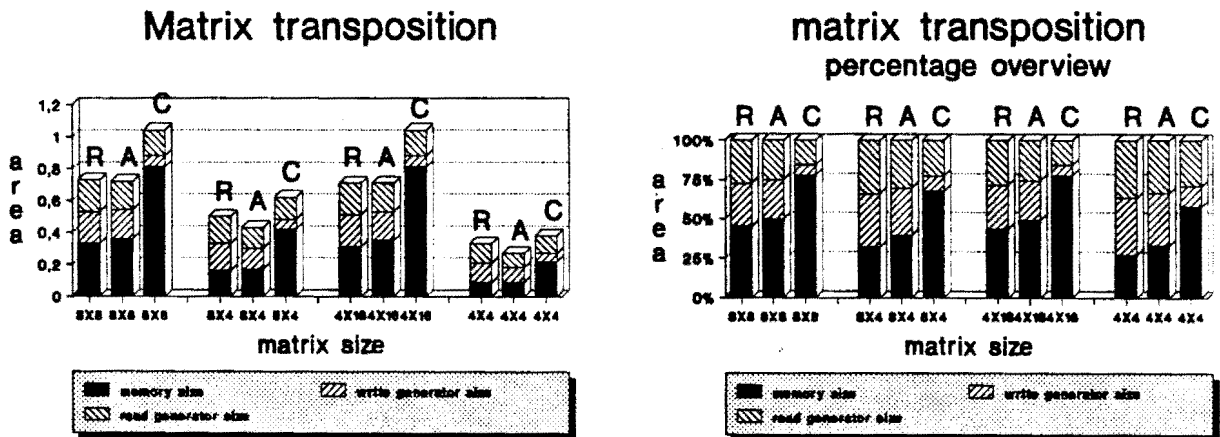


Figure 5.1: Results of MATCHBOX for matrix transposition

In Figure 5.1 can be seen that absolute location assignment results in the smallest IPB, but when the matrices become bigger relative location assignment will turn out to be the smallest. The latter can not be concluded from Figure 5.1, but several from test on bigger matrices. Relative location assignment has yet another advantage for bigger applications, it comes up much faster with a valid schedule than absolute location assignment.

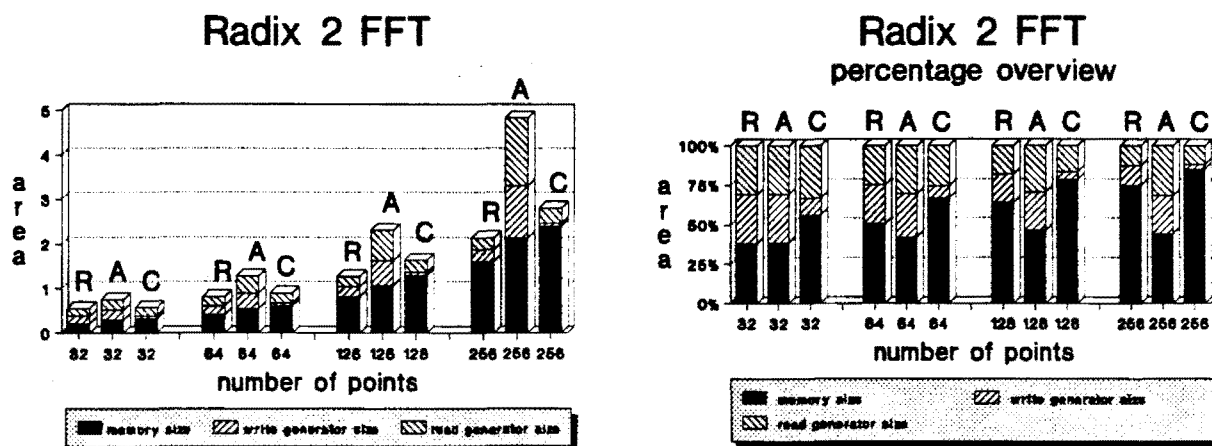


Figure 5.2: Results of MATCHBOX for radix 2 FFT

In Figure 5.2 it can be seen that for radix 2 FFT in most of the cases relative location assignment results in the smallest IPB. For the 32 points FFT counter addressing is the smallest. Also can be seen that for larger FFTs the percentage of area used for address generation with relative addressing decreases. The same holds for counter addressing. This is also the case for matrix transposition. The other applications of which the address generator and memory costs are shown in appendix D show similar results.

5.2 Results regular placement

The regular placement program offers a lot of freedom in setting the amount of regularity in one of the address generators. That is why a selection of the most promising options is made. For most of the applications it is tried to increase the regularity in the write address generator, except for the radix 2 FFT and radix 4 FFT. For these application the read address generator is regularized. This is done because for these applications regularizing the read address sequence resulted in a incretion of amount of regularity in the write address generator. This is only possible when the application doesn't read a sample more than once. Further IPB are made with *Maximum Regularity* both for *Relative* (MRR) and *Absolute* (MRA) location assignment. Also it is tried to decrease the amount of memory needed in the IPB by decreasing the regularity in the address generator.

This is done with the "non fixed minimum" setting of the regular placement program. These results are called minimum memory solutions. So we have *Minimum Memory with Absolute* (MMA) and *Relative location assignment* (MMR). The chosen architecture, the costs of the address generators and the memory costs can be found in appendix E. The total cost of the IPBs generated with regular placement and the smallest IPB resulting from the techniques of MATCHBOX are shown in the following tables.

matrix tr.	MMR	MRR	matchb. (R)	MMA	MRA	matchb. (A)
8 x 8	0.94	1.01	0.73	0.74	1.04	0.72
4 x 16	0.93	0.96	0.71	0.97	1.04	0.71
4 x 4	0.34	0.41	0.33	0.31	0.38	0.27
8 x 4	0.59	0.61	0.50	0.57	0.62	0.43

Table 5.1: IPB costs for matrix transposition (mm^2)

radix 2 FFT	MMR	MRR	matchb. (R)	MMA	MRA	matchb. (A)
32 points	0.73	0.71	0.55	0.66	0.56	0.57
64 points	1.06	1.06	0.81	0.89	0.89	0.90
128 points	1.74	1.71	1.27	1.64	1.60	1.64
256 points	3.00	2.98	2.14	2.98	2.74	2.81

Table 5.2: IPB costs for radix 2 FFT (mm^2)

spiral left	MMR	MRR	matchb. (R)	MMA	MRA	matchb. (A)
4 x 4	0.35	0.43	0.33	0.34	0.38	0.29
6 x 6	0.62	0.69	0.51	0.61	0.70	0.49

Table 5.3: IPB costs for spiral left turning transformation (mm^2)

Zig - zag	MMR	MRR	matchb. (R)	MMA	MRA	matchb. (A)
8 x 8	0.69	0.72	0.58	0.58	0.62	0.50
6 x 6	0.50	0.49	0.42	0.39	0.43	0.37

Table 5.4: IPB costs for zig - zag transformation (mm^2)

Radix 4 FFT	MMR	MRR	matchb. (R)	MMA	MRA	matchb. (A)
16 points	0.45	0.48	0.42	0.38	0.34	0.34
64 points	0.90	0.94	0.96	0.91	0.76	0.94

Table 5.5: IPB costs for radix 4 FFT (mm^2)

From these tables it can be concluded that regular placement does not result in a significant reduction in cost for the total inter processor buffer. This is because aiming at a regular address sequence turns out to be rather expensive in memory cost, compared with the techniques of MATCHBOX.

Chapter 6

Merging of IPBs

To make a more flexible IPB it is tried to merge different applications into one IPB. The architecture that could be used for this merged IPB is one with a storage unit divided in different parts. Each part of this storage unit is used for a different application. Because regular placement aims at an as optimal as possible use of the run length delta architecture the merged IPBs with only this architectures for the address generators are compared. A possible architecture is shown in appendix F. In this architecture can be seen that the storage is divided in different parts. To address these different blocks in the storage unit, the start and end points of the blocks in the storage unit has to be stored. A control signal makes the selection of the start and end points belonging to a certain application. The comparator is used to determine if the end point is reached. If the end point is reached the counter will load the start point again. For each application the first run length has to be stored too, again the control signal determines which first runlength is used.

6.1 Results MATCHBOX and regular placement

To compare the results of MATCHBOX and regular placement the number of entries in the run length delta storage units are compared. It is tries to make a multi application IPB for two sorts of applications, for matrix transposition and for radix 2 FFT. The results can be seen in the following tables.

Matrix transposition (Regular placement relative)		
memory size = 83		
alg. size	Write # entries	Read # entries
4 x 4	2	10
8 x 4	2	18
8 x 8	6	30
4 x 16	4	18
total	14	76

Table 6.1: Matrix transposition regular placement

Matrix transposition (Relative loc. assignment)		
memory size = 51		
alg. size	Write # entries	Read # entries
4 x 4	13	11
8 x 4	30	37
8 x 8	46	40
4 x 16	64	57
total	153	145

Table 6.2: Matrix transposition relative location assignment (MATCHBOX)

Matrix transposition (Absolute location assignment)		
memory size = 55		
alg. size	Write # entries	Read # entries
4 x 4	18	27
8 x 4	39	37
8 x 8	56	67
4 x 16	74	71
total	187	202

Table 6.3: Matrix transposition absolute location assignment (MATCHBOX)

Matrix transposition (Counter addressing)		
memory size = 128		
alg. size	Write # entries	Read # entries
4 x 4	2	17
8 x 4	2	33
8 x 8	2	33
4 x 16	2	17
total	8	100

Table 6.4: Matrix transposition counter addressing (MATCHBOX)

Radix 2 FFT (Regular placement relative)		
memory size = 383		
alg. size	Write # entries	Read # entries
256 points	34	2
128 points	30	2
64 points	26	2
32 points	22	2
total	112	8

Table 6.5: Radix 2 FFT regular placement

Radix 2 FFT (Relative loc. assignment)		
memory size = 255		
alg. size	Write # entries	Read # entries
256 points	9	9
128 points	8	8
64 points	7	7
32 points	6	6
total	30	30

Table 6.6: Radix 2 FFT relative location assignment (MATCHBOX)

Radix 2 FFT (Absolute location assignment)		
memory size = 342		
alg. size	Write # entries	Read # entries
256 points	1321	2300
128 points	600	1002
64 points	265	443
32 points	124	189
total	2310	3934

Table 6.7: Radix 2 FFT absolute location assignment (MATCHBOX)

Radix 2 FFT (Counter addressing)		
memory size = 384		
alg. size	Write # entries	Read # entries
256 points	2	2049
128 points	2	898
64 points	2	386
32 points	2	161
total	8	3494

Table 6.8: Radix 2 FFT counter addressing (MATCHBOX)

So we merge four sizes of an application into one IPB. The application with the most samples determines the memory size. In the Tables 6.5 till 6.4 can be seen that the techniques of PHIDEO are not capable of using the extra amount of memory, offered by the largest application, in order to reduce the number of entries in the storage unit for the smaller applications. Because all the techniques of PHIDEO aim at either minimum memory or minimal addressing hardware, a trade of between addressing cost and memory cost is not possible. Regular placement can use this extra amount of memory. For the matrix transposition regular placement needs more memory than relative and absolute location assignment. But because of good use of the extra amount of memory it was possible to reduce the number of entries in the run length delta storage unit. For the Radix 2 FFT one can see that relative addressing will definitely result in the smallest solution, it not only needs the smallest memory size it also needs the fewest entries in the run length delta storage unit. Whether this merging with the use of regular placement will result in a area reduction is difficult to predict. This is because it is not known how large the overhead cost of the architecture will be and how much the merged storage unit can be minimized. This could be something for future research.

6.2 Merging of N applications

In the search for regularity a nice solution for a IPB for matrix transposition is found. This IPB is capable of handling all sizes of matrix transposition as long as the memory size allows it. The memory size needed can be derived when the matrix size is known. The deltas between the consecutive addresses can also be calculated when the matrix size is known, so no storage unit is needed any more.

Looking at matrix transposition one can see that writing the samples in the memory in a regular way can be done as long as the memory is large enough. This can be seen in Figure 6.1

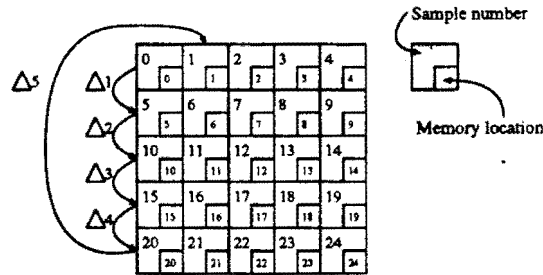


Figure 6.1: Storage of 25 samples in 25 memory locations

In this example there are 25 samples and 25 memory locations. Writing with a regular address sequence is now possible but reading with a fully regular address sequence is not. $\Delta 5$ differs from $\Delta 1$ till $\Delta 4$. To solve this we just remove memory location 24 and store sample 24 in memory location zero. This is allowed when we assume that sample zero is read before sample 24 is written in the memory. The storage of the samples is now like in Figure 6.2.

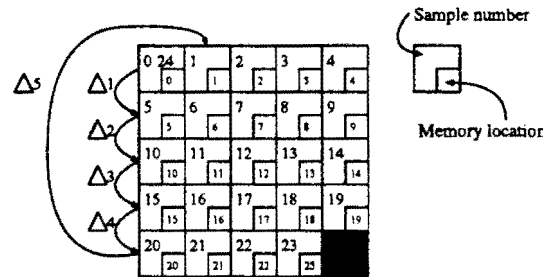


Figure 6.2: Storage of 25 samples in 24 memory locations

One can see now that writing is done regularly (with one delta) and reading is done regularly too ($\Delta 5$ modulo 24 equals $\Delta 1$ till $\Delta 4$). For the next frame the samples can be written in the memory with the delta used for reading the samples of the previous frame. The reading of the samples of this next frame can then be done again with a delta of one. So for a square matrices it hold that:

Write(1) $\Delta = D1$
 Read(1) $\Delta = R1$
 Write(2) $\Delta = R1$
 Read(2) $\Delta = D1$
 Write(3) $\Delta = D1$
 Read(3) $\Delta = R1$

These deltas can easily be calculated when the number of columns and the number of samples is known. Start with a write(1) $\Delta = D1 = 1$. How $R1$ is calculated can be seen in Equation 6.1.

$$R1 = \# \text{ columns} \times D1 \pmod{B} \quad (6.1)$$

with B = Number of samples - 1

The start position of the write sequence is not the same every frame. It is not possible to start on the same position every frame because lifetime clashes will occur than. However a good start address can be calculated so no life time clashes will occur. The start address is calculated with Equation 6.2.

$$\text{Start address} = \text{previous start address} + \text{previous Delta} \pmod{B} \quad (6.2)$$

For the five by five matrix transposition this results in the following deltas and start addresses:

Δ number	Δ size	start address
0	1	0
1	5	1
2	1	6
3	5	7
4	1	12
5	5	13
6	1	18
7	5	19
8	1	0

Table 6.9: deltas and start addresses for a 5×5 matrix

For non square matrices the deltas are not the same after one write and one read but the calculation of the deltas is the same, see Equation 6.3

$$\begin{aligned} \Delta 2 &= \# \text{ columns} \times \Delta 1 \pmod{B} \\ \Delta 3 &= \# \text{ columns} \times \Delta 2 \pmod{B} \\ \Delta 4 &= \# \text{ columns} \times \Delta 3 \pmod{B} \\ \Delta 5 &= \# \text{ columns} \times \Delta 4 \pmod{B} \end{aligned} \quad (6.3)$$

with B = Number of samples - 1

As an example the deltas and start addresses for a 7×5 matrix transposition are shown in Table 6.10. After how many times the delta repeat is not interesting any more in this case because only the pervious delta and start address have to be stored. The next delta and start address can be calculated from these values. These deltas and start addresses are calculated with a UNIX script from which the source code can be found in appendix G.

Δ number	Δ size	start address
0	1	0
1	5	1
2	25	6
3	23	31
4	13	20
5	31	33
6	19	30
7	27	15
8	33	8
9	29	7
10	9	2
11	11	11
12	21	22
13	3	9
14	15	12
15	7	27
16	1	0

Table 6.10: delta and start addresses for a 7×5 matrix

The IPB based on this addressing principle will be a parameterizable IPB. By giving the number of samples and the number of columns the addresses can be calculated. Again will the largest application determine the memory size of the IPB.

Chapter 7

Conclusions and recommendations

With regular placement a trade off can be made between address generator cost and memory cost. Looking at the size of the total IPBs generated with regular placement one can conclude that regular placement does not offer a solution which results in a smaller IPB. When we look more closely to the results it appears that regular placement, is too expensive in memory costs. The amount of area gained by regularizing the addressing sequences is too small to realize a smaller IPB. Also one can see that although there is aimed at an optimal use of the run length delta architecture, MATCHBOX not always selects this architecture to generate the regular sequence, just because it is more expensive than the other architectures.

For merging different applications into one IPB regular placement is a good technique. This because regular placement is capable of using the extra amount of memory to increase the regularity in one address sequence. This will result in less entries in the storage unit for the applications for which the used memory is oversized.

The technique for making a parameterizable IPB for matrix transposition is a nice solution, but it is a manual solution, and it is only applicable for matrix transposition. So its a very dedicated solution.

Overall we can conclude that the techniques and target architectures of MATCHBOX cover the design space very well. Regular placement adds an extra utility to these techniques. With regular placement the user can determine beforehand the amount of regularity in one address generator. Also the possibility to use oversized memories for increasing the regularity in the address sequence is a possibility offered by regular placement and not by the techniques of MATCHBOX.

Further research is needed on the merging of applications in one IPB. Especially on the estimation of the needed chip area of the merged IPB. Also an investigation of a really programmable IPBs is recommended. With programmability we mean an architecture with a micro controller. The program of this micro controller should be derived from the high level description of the application.

Appendix A

PIF

A.1 PIF description of matrix transposition

```
func input() out = inbuf {0.0} [1];  
func output(in) = outbuf {0.0} [1];
```

```
#define XSIZE 5  
#define YSIZE 5  
#define PERIOD 1  
#define GLOBAL PERIOD*XSIZE*YSIZE  
  
{GLOBAL}  
  
    (1 : 0 .. YSIZE-1) {PERIOD*XSIZE} ::  
    begin  
        (c : 0 .. XSIZE-1) {PERIOD} ::  
        begin  
{in}            x[1][c] = input();  
                end;  
        end;  
  
    (c : 0 .. XSIZE-1) {PERIOD*YSIZE} ::  
    begin  
        (1 : 0 .. YSIZE-1) {PERIOD} ::  
        begin  
{out}          = output(x[1][c]);  
                end;  
        end;  
  
% in = [0,];  
% out = [,2*GLOBAL];
```


A.2 PIF description of 8×8 zig-zag transformation

```

#define P 1
#define N 1

infunc in {1} = inbuf [1];
outfunc out {1} = outbuf [1];

signal a = 8;

memory all = {1.0,1.0,0.0,0.0};

{N*P*64}
(b : 0 .. N-1) {8} ::
  (r : 0 .. 7 ) {8*N*P} ::
    ( c : 0 .. 7 ) {P} ::
{in}      a[b][r][c] = in() [0,0];

(b : 0 .. N-1) {64*P} ::
  begin
{a00}    = out(a[b][0][0]) [,N*P*64];
{a01}    = out(a[b][0][1]); % a01 - a00 = P*1;
{a10}    = out(a[b][1][0]); % a10 - a00 = P*2;
{a20}    = out(a[b][2][0]); % a20 - a00 = P*3;
{a11}    = out(a[b][1][1]); % a11 - a00 = P*4;
{a02}    = out(a[b][0][2]); % a02 - a00 = P*5;
{a03}    = out(a[b][0][3]); % a03 - a00 = P*6;
{a12}    = out(a[b][1][2]); % a12 - a00 = P*7;
{a21}    = out(a[b][2][1]); % a21 - a00 = P*8;
{a30}    = out(a[b][3][0]); % a30 - a00 = P*9;
{a40}    = out(a[b][4][0]); % a40 - a00 = P*10;
{a31}    = out(a[b][3][1]); % a31 - a00 = P*11;
{a22}    = out(a[b][2][2]); % a22 - a00 = P*12;
{a13}    = out(a[b][1][3]); % a13 - a00 = P*13;
{a04}    = out(a[b][0][4]); % a04 - a00 = P*14;
{a05}    = out(a[b][0][5]); % a05 - a00 = P*15;
{a14}    = out(a[b][1][4]); % a14 - a00 = P*16;
{a23}    = out(a[b][2][3]); % a23 - a00 = P*17;
{a32}    = out(a[b][3][2]); % a32 - a00 = P*18;
{a41}    = out(a[b][4][1]); % a41 - a00 = P*19;
{a50}    = out(a[b][5][0]); % a50 - a00 = P*20;
{a60}    = out(a[b][6][0]); % a60 - a00 = P*21;
{a51}    = out(a[b][5][1]); % a51 - a00 = P*22;
{a42}    = out(a[b][4][2]); % a42 - a00 = P*23;
{a33}    = out(a[b][3][3]); % a33 - a00 = P*24;
{a24}    = out(a[b][2][4]); % a24 - a00 = P*25;
{a15}    = out(a[b][1][5]); % a15 - a00 = P*26;
{a06}    = out(a[b][0][6]); % a06 - a00 = P*27;
{a07}    = out(a[b][0][7]); % a07 - a00 = P*28;
{a16}    = out(a[b][1][6]); % a16 - a00 = P*29;
{a25}    = out(a[b][2][5]); % a25 - a00 = P*30;

```

```
{a34} = out(a[b][3][4]); % a34 - a00 = P*31;
{a43} = out(a[b][4][3]); % a43 - a00 = P*32;
{a52} = out(a[b][5][2]); % a52 - a00 = P*33;
{a61} = out(a[b][6][1]); % a61 - a00 = P*34;
{a70} = out(a[b][7][0]); % a70 - a00 = P*35;
{a71} = out(a[b][7][1]); % a71 - a00 = P*36;
{a62} = out(a[b][6][2]); % a62 - a00 = P*37;
{a53} = out(a[b][5][3]); % a53 - a00 = P*38;
{a44} = out(a[b][4][4]); % a44 - a00 = P*39;
{a35} = out(a[b][3][5]); % a35 - a00 = P*40;
{a26} = out(a[b][2][6]); % a26 - a00 = P*41;
{a17} = out(a[b][1][7]); % a17 - a00 = P*42;
{a27} = out(a[b][2][7]); % a27 - a00 = P*43;
{a36} = out(a[b][3][6]); % a36 - a00 = P*44;
{a45} = out(a[b][4][5]); % a45 - a00 = P*45;
{a54} = out(a[b][5][4]); % a54 - a00 = P*46;
{a63} = out(a[b][6][3]); % a63 - a00 = P*47;
{a72} = out(a[b][7][2]); % a72 - a00 = P*48;
{a73} = out(a[b][7][3]); % a73 - a00 = P*49;
{a64} = out(a[b][6][4]); % a64 - a00 = P*50;
{a55} = out(a[b][5][5]); % a55 - a00 = P*51;
{a46} = out(a[b][4][6]); % a46 - a00 = P*52;
{a37} = out(a[b][3][7]); % a37 - a00 = P*53;
{a47} = out(a[b][4][7]); % a47 - a00 = P*54;
{a56} = out(a[b][5][6]); % a56 - a00 = P*55;
{a65} = out(a[b][6][5]); % a65 - a00 = P*56;
{a74} = out(a[b][7][4]); % a74 - a00 = P*57;
{a75} = out(a[b][7][5]); % a75 - a00 = P*58;
{a66} = out(a[b][6][6]); % a66 - a00 = P*59;
{a57} = out(a[b][5][7]); % a57 - a00 = P*60;
{a67} = out(a[b][6][7]); % a67 - a00 = P*61;
{a76} = out(a[b][7][6]); % a76 - a00 = P*62;
{a77} = out(a[b][7][7]); % a77 - a00 = P*63;
    end;
% a = all;
```

A.3 PIF description of 4×4 left turning spiral

```

func input() out = inbuf {0.0} [1];
func output(in) = outbuf {0.0} [1];

#define XSIZE 4
#define YSIZE 4
#define PERIOD 1
#define GLOBAL PERIOD*XSIZE*YSIZE

{GLOBAL}

    (1 : 0 .. YSIZE-1) {PERIOD*XSIZE} ::
    begin
        (c : 0 .. XSIZE-1) {PERIOD} ::
        begin
{in}            x[l][c] = input();
                end;
        end;
    end;

{out_1}        = output(x[2][2]);
{out_2}        = output(x[2][1]);
                %out_2 -out_1 = 1;
{out_3}        = output(x[1][1]);
                %out_3 -out_1 = 2;
{out_4}        = output(x[1][2]);
                %out_4 -out_1 = 3;
{out_5}        = output(x[1][3]);
                %out_5 -out_1 = 4;
{out_6}        = output(x[2][3]);
                %out_6 -out_1 = 5;
{out_7}        = output(x[3][3]);
                %out_7 -out_1 = 6;
{out_8}        = output(x[3][2]);
                %out_8 -out_1 = 7;
{out_9}        = output(x[3][1]);
                %out_9 -out_1 = 8;
{out_10}       = output(x[3][0]);
                %out_10 -out_1 = 9;
{out_11}       = output(x[2][0]);
                %out_11 -out_1 = 10;
{out_12}       = output(x[1][0]);
                %out_12 -out_1 = 11;
{out_13}       = output(x[0][0]);
                %out_13 -out_1 = 12;
{out_14}       = output(x[0][1]);
                %out_14 -out_1 = 13;
{out_15}       = output(x[0][2]);
                %out_15 -out_1 = 14;
{out_16}       = output(x[0][3]);
                %out_16 -out_1 = 15;
% in = [0,];
% out_1 = [,2*GLOBAL];

```

A.4 PIF description of radix 2 FFT

```

#define P 1
#define N 256
#define STAGES 8

func input()      out = {P} inbuf {0.0} [1];
func output(in)   = {P} outbuf {0.0} [1];
func add(in1,in2) out = {P} alu   {1.0} [1];
func sub(in1,in2) out = {P} alu   {1.0} [1];

signal x = 8;

memory mem = {1.0,1.0,0.0,0.0};

{(STAGES+1)*N*P}

    (i : 0 .. N-1 ) {1*P} ::
begin
{in}      x[0][i] = input() [0,];
end;

    ( s : 1 .. STAGES ) {N*P} ::
begin
    (i : 0 .. N/2-1) {2*P} ::
begin
{s1}      x[s][2*i] = add(x[s-1][i],x[s-1][i+N/2]);
{s2}      x[s][2*i+1] = sub(x[s-1][i],x[s-1][i+N/2]);
end;
end;

    (i : 0 .. N-1 ) {1*P} ::
begin
{out}     = output(x[STAGES][i]) [,2*STAGES*N*P];
end;

% s1 - in >= N*P;
% s2 - in >= N*P;
% out - s2 >= STAGES*N*P-1;
% out - s1 >= STAGES*N*P-1;
% s2 - s1 = P;

% in -> out = mem;
% out <- in = mem;

```

A.5 PIF description of radix 4 FFT

```

#define P 1
#define N 64
#define STAGES 3

func input()      out = {P} inbuf {0.0} [1];
func output(in)   = {P} outbuf {0.0} [1];
func butterfly (i1,i2{1},i3{2},i4{3}) o1{12},o2{13},o3{14},o4{15} = alu {1.0} [1];

signal x = 8;

{(2*STAGES+2)*N*P+8}

  (i : 0 .. N-1 ) {P} ::
{in}      x[0][i] = input() [0,];

  ( s : 1 .. STAGES ) {2*N*P} ::
begin
  (i : 0 .. N/4-1) {8*P} ::
begin
{s}      x[s][4*i], /* out1*/
        x[s][4*i+1], /* out2*/
        x[s][4*i+2], /* out3*/
        x[s][4*i+3] = /* out4*/
        butterfly(
                x[s-1][i], /* in1*/
                x[s-1][i+<N/2>], /* in2*/
                x[s-1][i+<N/4>], /* in3*/
                x[s-1][i+<3*(N/4)>] /* in4*/
        );

        end;
  end;
  (i : 0 .. N-1 ) {P} ::
begin
{out}    = output(x[STAGES][i]) [,4*STAGES*N*P];
end;

% s - in >= N*P;
% out - s >= 2*STAGES*N*P+8;

```

Appendix B

Regular placement program

The regular placement program is called by :

```
rp [life time file]
```

The life time file syntax is like:

```
<framelength>  
<sample name> <write timepoint> <read timepoint>  
<sample name> <write timepoint> <read timepoint>  
.  
.  
.  
.  
.  
.
```

When the program is executed several questions are asked. Which options are set by which answers will be explained next.

```
rp [lifetime file]
```

```
.  
.  
.  
calculate minimal memory size ? (1 = yes / 0 = no)
```

By answering yes here the M_{lowb} and some other values which play no further role in the program will be calculated, M_{lowb} is necessary when binary search through the memory is wished.

relative (1) or absolute (0) add ? (0/1)

Here the choice is made between relative or absolute location assignment. When absolute location assignment is selected the program determines whether it is necessary to expand the frame. If expansion is needed the program has to be started again with the expanded data schedule which is generated by the program. This expanded data schedule can be found in the file [life time file].exp. For relative location assignment expansion is never needed.

regularize on read side? (1= yes / 0 = no)

If yes is selected here the program will try to regularize the read address generator.

linear or binary search through mem sizes ? (0 = bin / 1 = lin)

linear or binary search selection.

give start size memory :

give max seq length :

min seq length fixed ? (0= no / 1 = yes)

give min seq length :

lin seq decr. (1) or single step(0) (1/0) ?

With these questions the user can determine the amount of regularity in the address sequence and the amount of memory sizes that is checked. By setting the non fixed option for the minimum sequence length every regular sequence will be at least half of the number of non placed samples. With this option the minimum sequence length should be half of number of samples and the maximum should be equal to the number of samples. With last option "lin seq decr. or single step" one can choose between shortening the placed sequence one at a time or in one step to the minimum sequence length. The single step option increases the computation speed.

During the execution of the program the placing and removing of the samples in the memory can be shown on a graphical display.

This files generated by the regular placement program are: XXX stand for the name of the life time file.

- XXX.exp : contains the expanded life time file.
- XXX.opt : contains the results of regular placement.
- XXX.plar : contains the PLA filling for the run length delta storage unit for the read address generator.
- XXX.plaw : contains the PLA filling for the run length delta storage unit for the write address generator.

- `XXX_mbox.read` : contains the data schedule with addresses resulting from regularizing the read address generator.
- `XXX_mbox.write` : contains the data schedule with addresses resulting from regularizing the write address generator.

The "`_mbox`" files are the input file for `MATCHBOX`. In that way an estimate of the IPB area cost can be made. To call `MATCHBOX` just type `matchbox -hg [XXX_mbox.read]` or `matchbox -hg [XXX_mbox.write]`

Appendix C

Matchbox memory sizes

In this appendix the memory sizes needed in the IPBs for the different techniques of PHIDEO are presented. The meaning of the abbreviations used in the different tables:

relative = relative location assignment

absolute = absolute location assignment

counter = counter addressing

memory loc. = Number of memory locations

memory size = Size of the memory with '# memory loc.' locations in mm²

application	technique	# memory loc.	memory size
matrix 8 x 8	relative	51	0.33
	absolute	55	0.36
	counter	128	0.81
matrix 8 x 4	relative	23	0.16
	absolute	25	0.17
	counter	64	0.42
matrix 4 x 16	relative	47	0.31
	absolute	53	0.35
	counter	128	0.81
matrix 4 x 4	relative	11	0.09
	absolute	12	0.09
	counter	32	0.22

application	technique	# memory loc.	memory size
Radix 2 FFT 32 points	relative	31	0.21
	absolute	43	0.29
	counter	48	0.32
Radix 2 FFT 64 points	relative	63	0.41
	absolute	84	0.54
	counter	94	0.60
Radix 2 FFT 128 points	relative	127	0.81
	absolute	169	1.07
	counter	205	1.29
Radix 2 FFT 256 points	relative	255	1.60
	absolute	342	2.14
	counter	384	2.40

APPENDIX C. MATCHBOX MEMORY SIZES

application	technique	# memory loc.	memory size
Radix 4 FFT 16 point	relative	16	0.12
	absolute	17	0.12
	counter	23	0.16
Radix 4 FFT 64 point	relative	64	0.42
	absolute	107	0.48
	counter	174	0.68

application	technique	# memory loc.	memory size
zig - zag 6 x 6	relative	16	0.12
	absolute	18	0.13
	counter	36	0.24
zig - zag 8 x 8	relative	29	0.20
	absolute	33	0.22
	counter	64	0.42

application	technique	# memory loc.	memory size
spiral left 6 x 6	relative	24	0.17
	absolute	28	0.19
	counter	72	0.46
spiral left 4 x 4	relative	12	0.09
	absolute	13	0.10
	counter	32	0.22

Appendix D

Matchbox address generator sizes

In this appendix the results of MATCHBOX are shown. The meaning of the different abbreviations used in the different tables:

T = Used *Technique*

R = *Relative location assignment*

A = *Absolute location assignment*

C = *Counter addressing*

Co = *Counter architecture*

AT = *Address Table architecture*

DT = *Delta Table architecture*

DTM = *Delta Table architecture with Modulo hardware*

RDT = *Run length Delta Table architecture*

RDTM = *Run Length Delta Table architecture with Modulo hardware*

bm = Number of entries in the run length delta PLA *before minimisation*

am = Number of entries in the run length delta PLA *after minimisation*

S = *Size of the address generator in mm²*

Write address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			Co
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	S
matrix 8 x 8	R							60	40	0.20				46	35	0.22	
	A	128	109	0.19	128	55	0.18				56	51	0.21	56	50	0.25	
	C																0.07
matrix 8 x 4	R							32	28	0.17				30	28	0.17	
	A	64	41	0.13	64	25	0.14				40	24	0.17	39	21	0.21	
	C																0.06
matrix 4 x 16	R							64	49	0.20				64	52	0.20	
	A	128	94	0.18	128	66	0.19				74	64	0.23	74	65	0.28	
	C																0.07
matrix 4 x 4	R							16	11	0.12				13	11	0.13	
	A	32	22	0.09	32	18	0.11				20	16	0.14	18	13	0.16	
	C																0.05

APPENDIX D. MATCHBOX ADDRESS GENERATOR SIZES

Read address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
matrix 8 x 8	R							64	35	0.20				40	36	0.22	
	A	128	108	0.18	128	68	0.16				68	58	0.21	67	58	0.25	
	C	128	127	0.20	128	7	0.16				33	7	0.17	33	7	0.22	
matrix 8 x 4	R							32	30	0.17				30	29	0.18	
	A	64	41	0.13	64	28	0.14				40	33	0.16	37	29	0.19	
	C	64	63	0.14	64	6	0.14				33	8	0.15	33	8	0.19	
matrix 4 x 16	R							64	52	0.20				57	52	0.22	
	A	128	94	0.18	128	62	0.18				72	60	0.22	71	61	0.26	
	C	128	127	0.20	128	7	0.16				17	9	0.17	17	9	0.22	
matrix 4 x 4	R							16	9	0.12				11	10	0.14	
	A	32	22	0.09	32	20	0.11				28	25	0.12	27	25	0.16	
	C	32	31	0.11	32	5	0.12				17	6	0.13	17	6	0.17	

Write address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
Ra. 2 FFT 32 p.	R							192	31	0.20				6	6	0.17	
	A	192	174	0.23	192	127	0.23				126	109	0.26	124	109	0.30	
	C																0.06
Ra. 2 FFT 64 p.	R							448	43	0.24				7	7	0.20	
	A	448	406	0.41	448	271	0.35				267	236	0.41	265	239	0.46	
	C																0.07
Ra. 2 FFT 128 p.	R							1024	67	0.28				8	8	0.23	
	A	1024	891	0.76	1024	558	0.56				800	540	0.68	800	540	0.73	
	C																0.08
Ra. 2 FFT 256 p.	R							2304	73	0.33				9	9	0.27	
	A	2304	2051	1.78	2304	1201	1.17				1321	1171	1.37	1321	1170	1.43	
	C																0.08

Read address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
Ra. 2 FFT 32 p.	R							192	31	0.20				6	6	0.17	
	A	192	169	0.23	192	147	0.24				190	140	0.24	189	139	0.28	
	C	192	188	0.24	192	50	0.19				162	35	0.22	161	20	0.25	
Ra. 2 FFT 64 p.	R							448	43	0.24				7	7	0.20	
	A	448	411	0.41	448	326	0.39				445	325	0.40	443	314	0.44	
	C	448	442	0.43	448	75	0.23				386	63	0.27	386	25	0.29	
Ra. 2 FFT 128 p.	R							1024	67	0.28				8	8	0.23	
	A	1024	912	0.75	1024	751	0.68				1022	752	0.74	1002	750	0.89	
	C	1024	1018	0.82	1024	108	0.27				898	85	0.32	898	36	0.34	
Ra. 2 FFT 256 p.	R							2304	73	0.33				9	9	0.27	
	A	2304	2210	1.90	2304	1692	1.54				2299	1671	1.53	2300	1667	1.59	
	C	2304	2298	1.97	2304	113	0.33				2050	71	0.36	2049	23	0.33	

Write address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
Ra. 4 FFT 16 p.	R							48	23	0.15				31	28	0.17	
	A	48	41	0.13	48	33	0.14				34	31	0.17	34	29	0.20	
	C																0.05
Ra. 4 FFT 64 p.	R							256	245	0.28				193	94	0.30	
	A	256	245	0.28	256	161	0.26				189	157	0.33	189	154	0.37	
	C																0.07

Read address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
Ra. 4 FFT 16 p.	R							48	28	0.15				45	36	0.16	
	A	48	45	0.13	48	33	0.14				48	33	0.14	46	31	0.18	
	C	48	44	0.13	48	20	0.14				35	20	0.16	33	10	0.19	
Ra. 4 FFT 64 p.	R							256	103	0.26				256	103	0.26	
	A	256	246	0.28	256	180	0.27				255	179	0.28	255	179	0.33	
	C	256	252	0.28	256	38	0.19				195	39	0.24	193	21	0.28	

Write address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
zig - zag 6 x 6	R							36	32	0.15				36	32	0.15	
	A	36	32	0.12	36	23	0.14				22	20	0.16	21	17	0.19	
	C																0.06
zig - zag 8 x 8	R							64	58	0.19				62	60	0.19	
	A	64	62	0.14	64	36	0.16				35	32	0.19	34	32	0.23	
	C																0.06

Read address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
zig - zag 6 x 6	R							36	33	0.15				35	32	0.15	
	A	36	32	0.12	36	30	0.14				32	29	0.14	30	26	0.18	
	C	36	35	0.13	36	22	0.15				20	16	0.16	18	16	0.20	
zig - zag 8 x 8	R							64	58	0.19				62	59	0.19	
	A	64	61	0.14	64	52	0.16				48	43	0.17	47	42	0.21	
	C	64	63	0.14	64	38	0.16				26	20	0.16	26	20	0.20	

Write address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
Spiral left 6 x 6	R							36	31	0.17				32	30	0.17	
	A	76	67	0.15	72	42	0.16				46	42	0.19	46	42	0.22	
	C																0.07
Spiral left 4 x 4	R							16	13	0.12				16	13	0.12	
	A	32	28	0.10	32	18	0.11				20	16	0.14	20	16	0.16	
	C																0.05

Read address generator sizes																	
appl.	T	AT			DT			DTM			RDT			RDTM			C
		bm	am	S	bm	am	S	bm	am	S	bm	am	S	bm	am	S	
Spiral left 6 x 6	R							36	31	0.17				33	29	0.18	
	A	72	67	0.15	76	64	0.17				56	50	0.17	56	51	0.20	
	C	72	71	0.17	72	46	0.19				25	25	0.18	25	25	0.22	
Spiral left 4 x 4	R							16	12	0.12				16	12	0.12	
	A	32	24	0.09	32	25	0.11				26	22	0.13	26	22	0.13	
	C	32	31	0.11	32	13	0.12				17	11	0.13	17	11	0.17	

Appendix E

Result regular placement

In this appendix the results of regular placement are shown. The meaning of the different abbreviations used in the different tables:

- MMR = *Minimal memory Memory with Relative location assignment*
- MRR = *Maximal Regularity with Relative location assignment*
- MMA = *Minimal memory Memory with Absolute location assignment*
- MRA = *Maximal Regularity with Absolute location assignment*
- Co = *Counter architecture*
- AT = *Address Table architecture*
- DT = *Delta Table architecture*
- DTM = *Delta Table architecture with Modulo hardware*
- RDT = *Run length Delta Table architecture*
- RDTM = *Run length Delta Table architecture with Modulo hardware*

matrix transposition							
size	Techn.	write add.		read add.		memory	
		arch.	size	arch.	size	# loc.	size
4 × 4	MMR	DTM	0.12	DTM	0.12	13	0.10
4 × 4	MRR	RDTM	0.13	DTM	0.14	19	0.14
4 × 4	MMA	AT	0.10	AT	0.10	15	0.11
4 × 4	MRA	Co	0.05	AT	0.11	32	0.22
8 × 4	MMR	RDTM	0.15	DTM	0.17	43	0.29
8 × 4	MRR	DTM	0.17	DTM	0.18	35	0.24
8 × 4	MMA	AT	0.14	AT	0.14	43	0.29
8 × 4	MRA	Co	0.06	DT	0.14	64	0.42
8 × 8	MMR	DTM	0.20	DTM	0.21	83	0.53
8 × 8	MRR	RDTM	0.18	DTM	0.20	99	0.63
8 × 8	MMA	DT	0.16	DT	0.17	63	0.41
8 × 8	MRA	Co	0.07	DT	0.16	128	0.81
4 × 16	MMR	DTM	0.21	DTM	0.22	78	0.50
4 × 16	MRR	RDTM	0.18	DTM	0.20	91	0.58
4 × 16	MMA	DT	0.17	DT	0.19	95	0.61
4 × 16	MRA	Co	0.07	DT	0.16	128	0.81

APPENDIX E. RESULT REGULAR PLACEMENT

Radix 2 FFT							
size	Techn.	write add.		read add.		memory	
		arch.	size	arch.	size	# loc.	size
32 points	MMR	RDTM	0.21	DTM	0.21	47	0.31
32 points	MRR	RDTM	0.21	RDTM	0.19	47	0.31
32 points	MMA	RDT	0.18	RDT	0.17	47	0.31
32 points	MRA	RDT	0.18	Co	0.06	48	0.32
64 points	MMR	RDTM	0.24	RDTM	0.21	95	0.61
64 points	MRR	RDTM	0.24	RDTM	0.21	95	0.61
64 points	MMA	RDT	0.21	Co	0.07	95	0.61
64 points	MRA	RDT	0.21	Co	0.07	95	0.61
128 points	MMR	DTM	0.27	RDTM	0.27	191	1.20
128 points	MRR	RDTM	0.27	RDTM	0.24	191	1.20
128 points	MMA	RDT	0.23	RDT	0.21	191	1.20
128 points	MRA	RDT	0.23	Co	0.08	205	1.29
256 points	MMR	RDTM	0.21	RDTM	0.30	383	2.39
256 points	MRR	RDTM	0.18	RDTM	0.28	383	2.39
256 points	MMA	RDT	0.17	Co	0.08	384	2.40
256 points	MRA	RDT	0.07	Co	0.08	384	2.40

Spiral left turning							
size	Techn.	write add.		read add.		memory	
		arch.	size	arch.	size	# loc.	size
4 x 4	MMR	DTM	0.12	DTM	0.14	14	0.11
4 x 4	MRR	RDTM	0.13	DTM	0.14	23	0.16
4 x 4	MMA	AT	0.11	AT	0.11	17	0.12
4 x 4	MRA	Co	0.05	AT	0.19	32	0.22
6 x 6	MMR	DTM	0.19	DTM	0.18	36	0.24
6 x 6	MRR	RDTM	0.16	RDTM	0.17	53	0.35
6 x 6	MMA	DT	0.16	AT	0.16	43	0.29
6 x 6	MRA	Co	0.07	AT	0.17	72	0.46

Zig - zag transformation							
size	Techn.	write add.		read add.		memory	
		arch.	size	arch.	size	# loc.	size
8 x 8	MMR	DTM	0.19	DTM	0.20	46	0.30
8 x 8	MRR	RDTM	0.16	DTM	0.20	55	0.36
8 x 8	MMA	AT	0.14	AT	0.14	46	0.30
8 x 8	MRA	Co	0.06	AT	0.14	64	0.42
6 x 6	MMR	DTM	0.17	DTM	0.17	23	0.16
6 x 6	MRR	RDTM	0.15	RDTM	0.17	25	0.17
6 x 6	MMA	AT	0.12	AT	0.12	21	0.15
6 x 6	MRA	Co	0.06	AT	0.13	36	0.24

Radix 4 FFT							
size	Techn.	write add.		read add.		memory	
		arch.	size	arch.	size	# loc.	size
16 points	MMR	DTM	0.17	DTM	0.16	17	0.12
16 points	MRR	DTM	0.17	RDTM	0.15	23	0.16
16 points	MMA	AT	0.13	RDT	0.13	17	0.12
16 points	MRA	AT	0.13	Co	0.05	23	0.16
64 points	MMR	DTM	0.25	DTM	0.23	87	0.42
64 points	MRR	DTM	0.23	RDTM	0.21	105	0.50
64 points	MMA	AT	0.25	RDT	0.24	87	0.42
64 points	MRA	DT	0.19	Co	0.07	105	0.50

Appendix F

architecture merged IPB

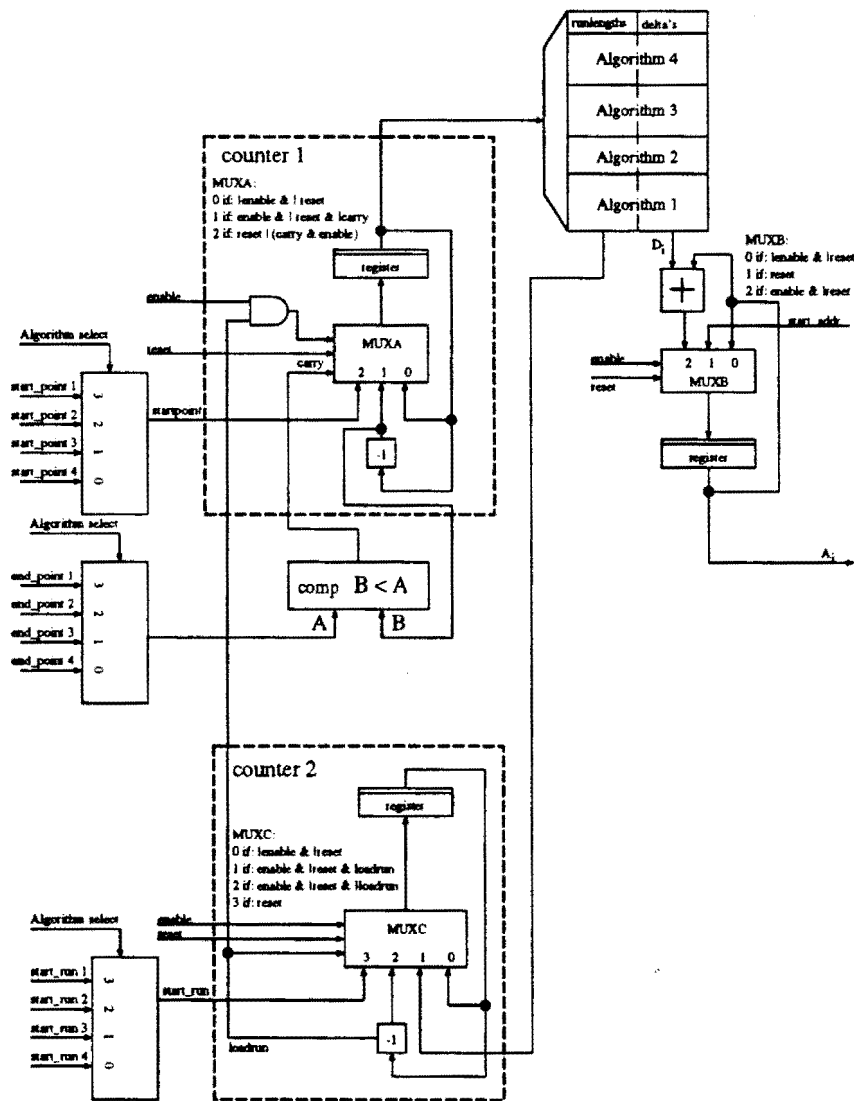


Figure F.1: Architecture for multi algorithm IPB

Appendix G

Unix script

This script determines the consecutive delta and start addresses for matrix transposition.

```
#!/bin/ksh
r=0
c=0
case $# in
  2)  r=$1
      c=$2
      ;;
  *)  echo "Usage: calcrep <rows> <cols>"
      exit;;
esac

p=0
i=0
d=1
s=0
((m=(r*c)-1))
while ((s != m || (i == 0) ))
do
    echo "d"$i" = "$d "st add ="$s
    ((s=s+d))
    if ((s > m))
    then
        ((s = s - m))
    fi
    ((k=(d*c)/m))
    ((d=(d*c)-(k*m)))
    ((i=i+1))
done
echo "d"$i" = "$d
```


Bibliography

- [1] van Meerbergen, J.L. et al., Relative Location Assignment for repetitive Schedules. in : Proc. European Conference on Design Automation (EDAC) Paris, February 1993. p. 403-407.
- [2] Lippens, P.E.R. et al., PHIDEO: A Silicon Compiler For High speed Algorithms, in: Proc. European Conference on Design Automation (EDAC), Amsterdam, February 1991. p. 436-441.
- [3] Chien-In Henry Chen and G.E. Sobelman, Singleport/Multiport Memory Synthesis in Data Path Design, in : Proc. of the ISCAS, New Orleans, May 1990. p. 1110-13.
- [4] Balakrishnan M. et al., Allocation of Multiport Memories in Data Path Synthesis. IEEE Trans. on CAD, Vol. 7, No. 4 April 1987. p. 536-40.
- [5] Tseng, C.J. and D.P. Siewiorek, Automated Synthesis of Data paths in digital systems. IEEE trans. CAD, Vol. CAD-5, No. 3, July 1986. p. 379-395.
- [6] Grant, D.M. and P.B. Denyer, Address Generation for Array Access Based on Modulus m Counters, in: Proc. European Conference on Design Automation (EDAC), Amsterdam, February 1991. p. 118-122.
- [7] Kurdahi, F.J. and Parker A.C., REAL : A program for register allocation, in : Proceedings of the 24th Design Automation Conference, July 1987. p. 210-215.
- [8] Parhi, K.K., Video data format converters Using Minimum Number of Registers. IEEE Transactions on Circuits and systems for Video Technologie, Vol. 2, no. 2, June 1992. p. 255-267.
- [9] Stok L., Transfer free Register allocation in Cyclic Data Flow Graphs, in : Proceedings European Conference on Design Automation, Brussels, February 1992. p. 181-185.
- [10] Woudsma R., et al., PIRAMID: An architecture driven silicon compiler for complex DSP applications, in : Proceedings IEEE International symposium on circuits and systems, 1990. p. 2696-2700.

- [11] Garey M.R., et al., The complexity of coloring circular arcs and chords, SIAM journal on algebraic and discrete methods (1), 1980, p. 216-227.
- [12] Tucker A., Coloring a family of circular arcs, SIAM journal on applied mathematics (29), 1975, p. 1-24.