Application of a Configurable Custom Computing Machine for Artery Extraction Filtering of 3D MRI Data

Alte de Boer*
Computer Systems Laboratory
HPL-95-95
August, 1995

custom computing, segmentation, MRI

An existing software algorithm for a three dimensional filter has been converted to a logic circuit, which was implemented on a configurable custom computing machine, Teramac. For this process a program was written that generates the hardware description of the filter according to filter parameters.

The implemented algorithm was a three dimensional filter that will extract the artery structure from three dimensional medical data like MRI.

The design is implemented on 1/16 of a Teramac machine and outperformed the software algorithm on an HP735 workstation by a factor of four. Designs for 1/2 Teramac are expected to run at more than 100 times the speed of a workstation.

**Universiteit Twente*, Enschede, The Netherlands. Alte de Boer worked as an intern for HP Laboratories, 1995.

Application of a Configurable Custom Computing Machine for Artery Extraction Filtering of 3D MRI Data

Alte de Boer

Internship at Hewlett-Packard Laboratories
February through May 1995

Supervisors:	Tom Malzbender	(HP-Labs)
	Phil Kuekes	(HP-Labs)
	Mark Bentum	(University of Twente)

Vakgroep besturingssystemen en computertechniek
Afdeling netwerktheorie
Faculteit der electrotechniek
Universiteit Twente

Contents

1. Introduction

Coronary artery blockages and other artery diseases are the number one death cause in the United States nowadays. It would be extremely helpful to doctors if they were able to see what state a patient's arteries are in.
MRI or CT technology can be used to scan the patient's artery structure. Although these scans give large amounts of valuable information, due to the fact that this inevitably comes contained in huge amounts of irrelevant data the information is hard to use. Currently, most doctors manually cycle through the printouts of the two dimensional images and mentally construct the three dimensional structure of the arteries.
The analysis will be significantly improved if a filter can be applied that removes all data except that containing information about the arteries. Various existing three dimensional visualisation techniques can then be applied to show just the artery structure.
At HP-labs an artery extraction filter is under development, that allows a three dimensional data set to be filtered without any user interaction.

1.1 Artery Extraction

To extract the artery structure contained in a scan, the MRI data is subjected to a three dimensional filter that delivers a maximum response for bright tubular structures [1]. Blood shows up brightly (high-valued) in MRI scans because of its high density of hydrogen atoms, whose resonance frequency MRI scanners are tuned to. Therefore the filter will react to blood vessels, as these are bright tubular structures, and not, for example, to blood pools like heart

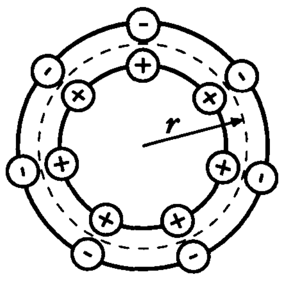


Figure 1.1 - The artery extraction filter

chambers.
The basic structure of the filter is shown in Figure 1.1. For each point shown, the value of the MRI data is either added or subtracted (indicated by plus and minus). This filter will respond

maximally to bright in-plane circles of radius *r* in a dark surrounding. The equal number of points to be added and points to be subtracted ensures that the filter produces a zero result when it is applied to an area of uniform brightness.
Arteries can be detected by placing the forementioned filter at a certain point in 3D-space and then rotate and resize it through a specified number of angles and multiplications. For each of these the resulting filter value is stored and when all rotations and resizings have been performed, the maximum filter value is determined. A high maximum value corresponds to a high probability that the point under observation is part of an artery. The orientation and size of the potential artery are indicated by ***which*** angle and multiplication produced the maximum.
In order to make the filter more sensitive to differences in orientation two tiers are used, as is

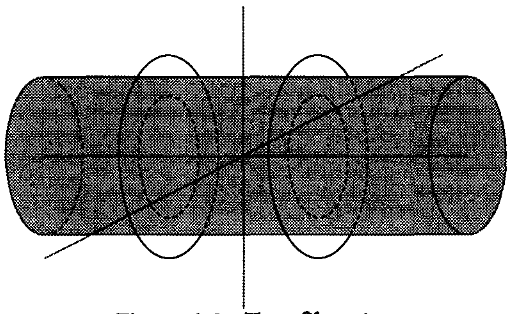


Figure 1.2 - Two filter tiers

shown in Figure 1.2. The grey area shows an artery that the filter responds to maximally.
The process of rotating and resizing (see Figure 1.3) is done for every voxel in the input data set. Every maximum filter value and corresponding orientation and size that is calculated is stored in an output data set, which can be further processed. For example a threshold operation can be applied to the filter values, which will make the arteries show up. More sophisticated methods can also use the orientation and size information to identify arteries.
The algorithm described above is very computationally intensive. However it is also very

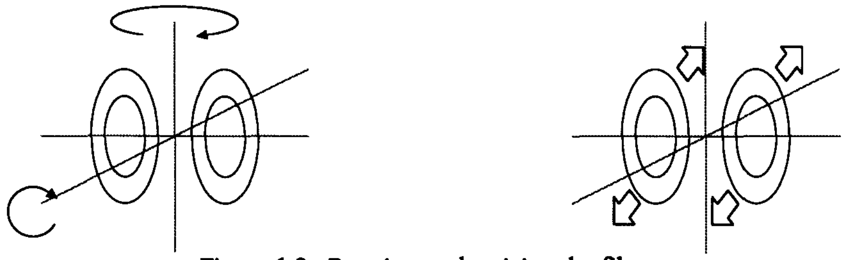


Figure 1.3 - Rotating and resizing the filter

repetitive, which makes the algorithm ideally suited for parallelisation.

1.2 Teramac

Teramac is a prototype realized by an HP-Labs research project [2]. It is a configurable custom computing machine capable of executing synchronous designs of up to one million gates at rates up to one megahertz. A massively-parallel computer like Teramac is very well suited to do the highly parallel artery extraction filtering on.

1.2.1 Hardware

A fully configured Teramac consists of sixteen boards, but currently an eight-board system is the largest possible configuration. Every board consists of 108 custom designed FPGAs called PLASMA (Programmable Logic And Switch MAtrix) and 32 megabytes of RAM.
Every PLASMA-chip consists of 256 logic units called PALEs. Every PALE has a six bit wide input and a two bit wide output and is capable of performing any logic function between them by using an internal look-up table. Every output bit has a register bit associated with it, that it may be connected with.
The 32 megabytes of RAM on each board are arranged as four banks with a 32 bit wide output each. It is possible to do a read and a write action at the same time, using different addresses.

1.2.2 Compiler

The Teramac compiler convert a user's circuit description into a structure of PALEs. The first part of the compilation is a filter that transforms an input file into an internal format used by the rest of the compiler. A filter exists for the hardware development package Tsutsuji. The second part is a merger, that tries to rearrange the logic gates in groups that repect the 6-input, 2-output limitation of the PALEs. Next, the logic circuit is partitioned such that the interconnections between parts are minimal. These parts are then assigned to specific PLASMA chips. The interconnections between these chips are done by the global router. The next step is the local placing and routing of the individual PLASMA chips, after which the configuration can be mapped to a bitstream that is compatible with the Teramac hardware. A timing analysis is performed that calculates the maximum possible clock speed for the user's circuit.

1.2.3 Tsutsuji

Tsutsuji is a gate array development package, that allows the user to design a circuit with flexible logic elements using a graphical environment [3]. Designs may also be defined using Logic Description Format (LDF), a hardware description language. A simulator is available that allows the user to ensure the design works according to specification.
Tsutsuji converts the user's circuit into a netlist, that can be further converted to a gate array mapping. The netlist can also be used by the Teramac compiler, which makes Tsutsuji an ideal development package for Teramac designs.

2. Artery Extraction on Teramac

This chapter explains the Teramac artery extraction design. First an overview of the design is given, then the design details of each part of the structure are explained in full.

2.1 Design Overview

In this section the basics of the structure are outlined, after which the two key elements in the design, the filter operator and the memory interface, are discussed seperately.

2.1.1 The Basic Structure

The tube filter can be seen as a convolution, which yields a result for one filter orientation and radius. In the following text the calculation of the maximum of all convolutions is called a filter operation. This operation can be applied to any point (x, y, z) in the data set and produces for that point the output: the maximum filter value and the orientation and radius that produced that maximum.
The basic structure of the design is rather straightforward (see Figure 2.1). The calculation of

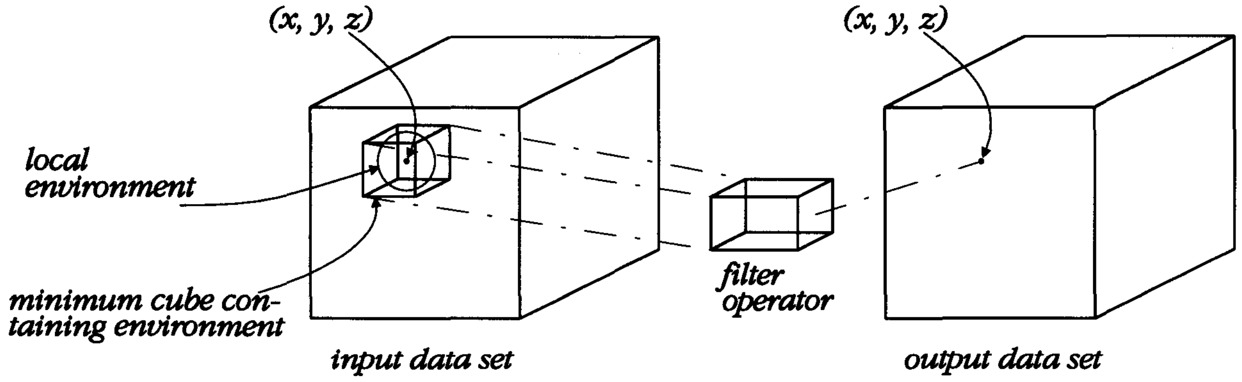


Figure 2.1 - Calculating a filter value for the point (x, y, z)

the maximum filter value for a point (x, y, z) in the data set is an operation on a number of points. All these points are in a local spherical environment, the details of which have been laid down by the definition of the filter, around the point (x, y, z). Now a cube can be defined that is the smallest possible cube containing the local environment whilst having the point (x, y, z) as its middle point. All points of this cube are fed into the filter operator, that produces the maximum filter output and the corresponding orientation and radius for the point (x, y, z).
The filter operation is now successively applied to all points in the input data set. The results are stored in an output data set.

The next two sections will address the filter operator and the memory structure that supplies the operator with the necessary data.

2.1.2 The Filter Operator

The filter is built up as shown in Figure 2.2. All the points of one minimum cube are fed into an interconnection module, which contains only wiring and no logic. The next element is the filter kernel, which calculates the filter value for one orientation and radius. Its inputs are all the points that form the circles of one tube filter. The first ½*n* points are added together, the second ½*n* points are also added together and the two results are subtracted from eachother. The interconnection module connects each kernel with the right points in the minimum cube. It happens that points in the cube are used more than once and that some are not used at all. Every filter kernel produces an output value for a certain filter orientation and size. All these output values are wired to the last stage in the filter operator: the maximizer. This unit outputs the largest filter value and a signal that contains the number of the kernel that produced that maximum.

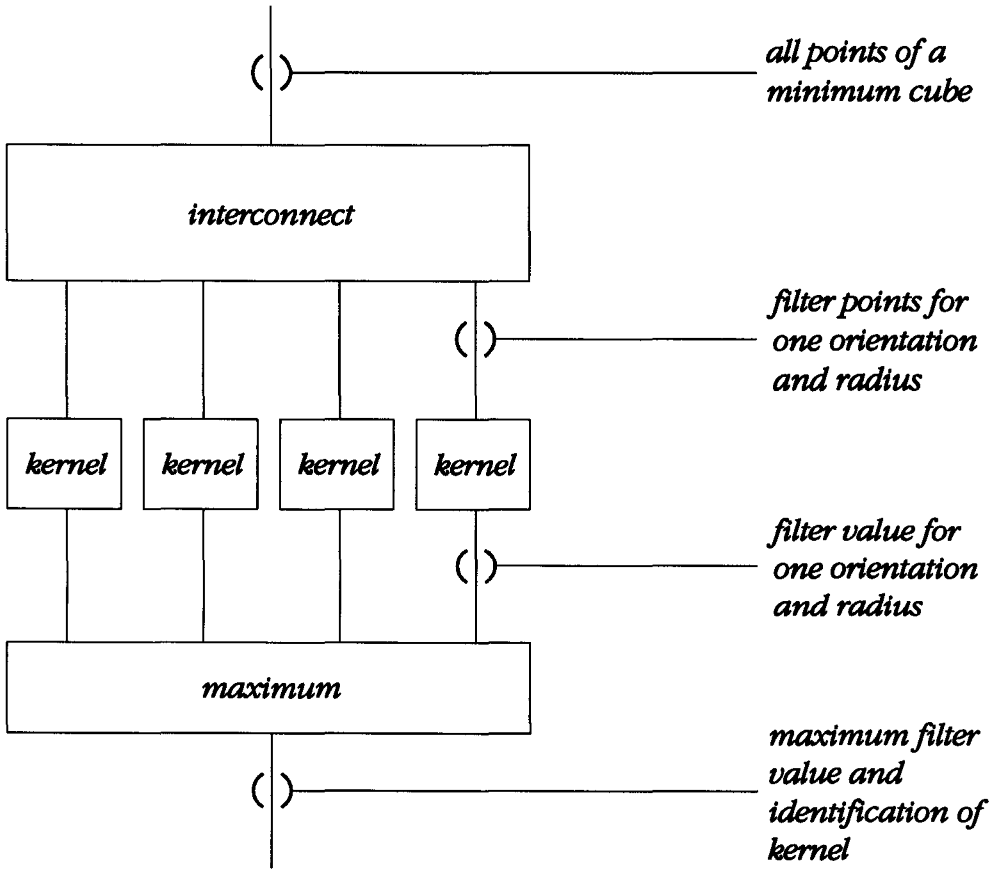

all points of a minimum cube
interconnect
filter points for one orientation and radius
kernel
kernel
kernel
kernel
filter value for one orientation and radius
maximum
maximum filter value and identification of kernel

Figure 2.2 - Parallel filter calculation

2.1.3 The Memory Structure

The bandwidth of the available memories is far insufficient to provide the data for the interconnector directly from the memories. To reduce the data flow from the memories a cache in the *x* direction is used (see Figure 2.3).

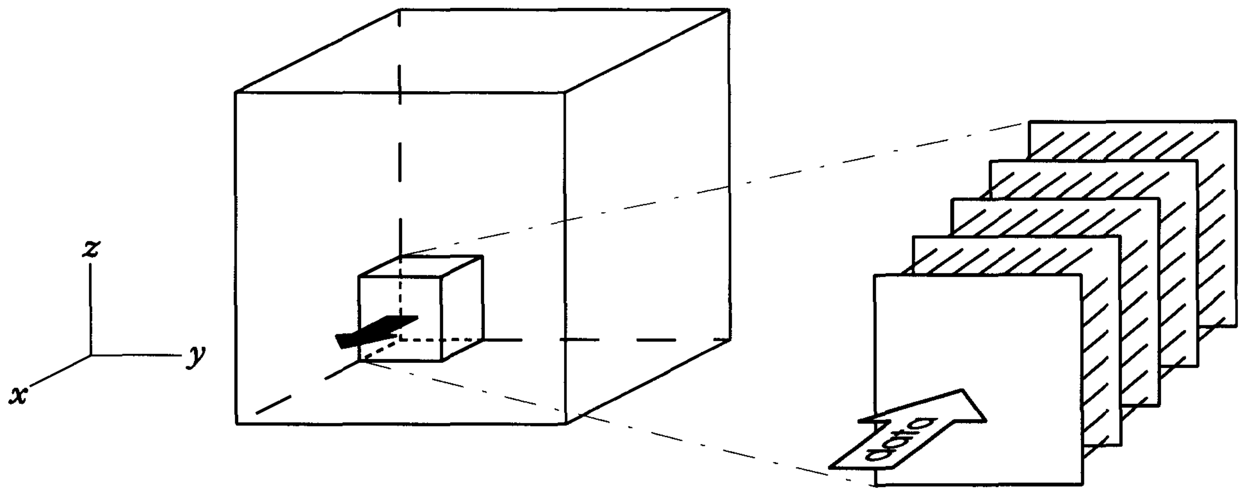


Figure 2.3 - A cache in the *x* direction, consisting of register planes

The minimum cube of the filter operator is successively applied to all points in the data set. To do this it moves mainly in the *x* direction. When it has done all *x* points it does one step in the *y* direction and when the highest *y* coordinate is reached a step in the *z* direction is done. A cache is constructed that buffers movements in the *x* direction. It consists of register planes that are connected to eachother in the *x* direction. Every cycle all planes shift one place further, one new plane of data arrives at the input and the last plane of data is discarded. The resulting cache cube can be abstracted as shown in Figure 2.4.

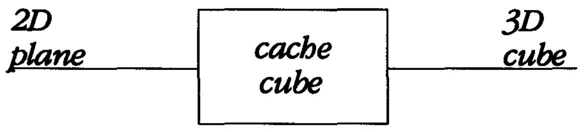


Figure 2.4 - Input and output of the cache cube

In addition to the *x* cache a cache in the *y* direction is used. In order to understand why this cache is necessary, suppose there would not be one. The cache cube requires a plane of data (size $n \times n$) each cycle. This data needs to be available in parallel, so n^2 memories are used and the data set (size $m \times m \times m$) is evenly distributed between these memories. See Figure 2.5

for an example where $n = 3$. When reading a plane where $0 \leq y \leq 2$ all inputs from the cache cube can be connected directly to the corresponding memories. When reading the next plane in the y direction, $1 \leq y \leq 3$, the data at $y = 1$ and $y = 2$ will of course be delivered by the same

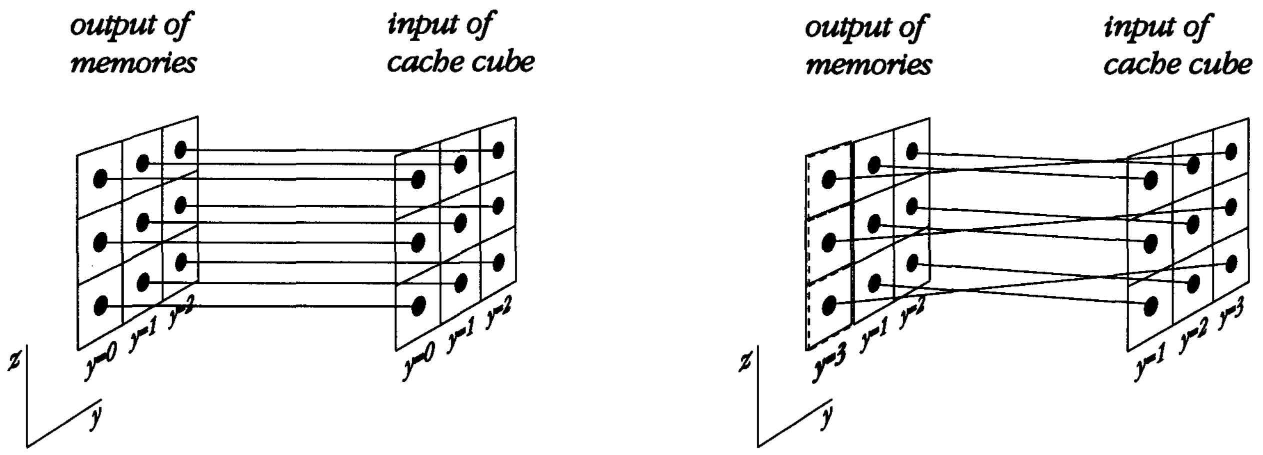


Figure 2.5 - Shift effect at the output of parallely addressed memories

memories as in the previous plane. The data at $y = 3$ is in the memory where $y = 0$ was in the previous plane, but at the next address (some addressing logic takes care of this). It can be seen that all the required data items can be read from memory, but that they appear in the wrong order and will have to be rolled back before being fed into the cache cube. This roll (or translator) unit would have to handle an arbitrary number of rolls in both dimensions (the same problem occurs when z is incremented by one). Such a unit could, for example, be constructed with n^2 times a n^2-to-1 multiplexer, but this would result in a huge structure.

A more feasible, though also more complicated, way to solve this problem is to cache in another dimension. In addition to the n^3 sized cache cube a $n^2 \times m$-sized cache bar is

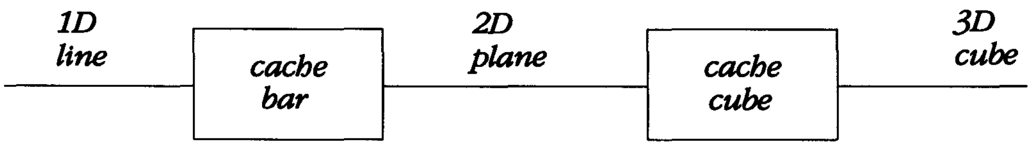


Figure 2.6 - Data complexity before and after caches

constructed (see Figures 2.6 and 2.7). The cache works by using the fact that when reading a $n \times n$-plane at any location (x, y, z), most of the data of that plane $(n\text{-}1 \times n)$ will be needed again when the filter operator comes to location $(x, y\text{+}1, z)$. Therefore the cache bar stores the planes that are fed into the cache cube for all x coordinates. When the filter operator moves to

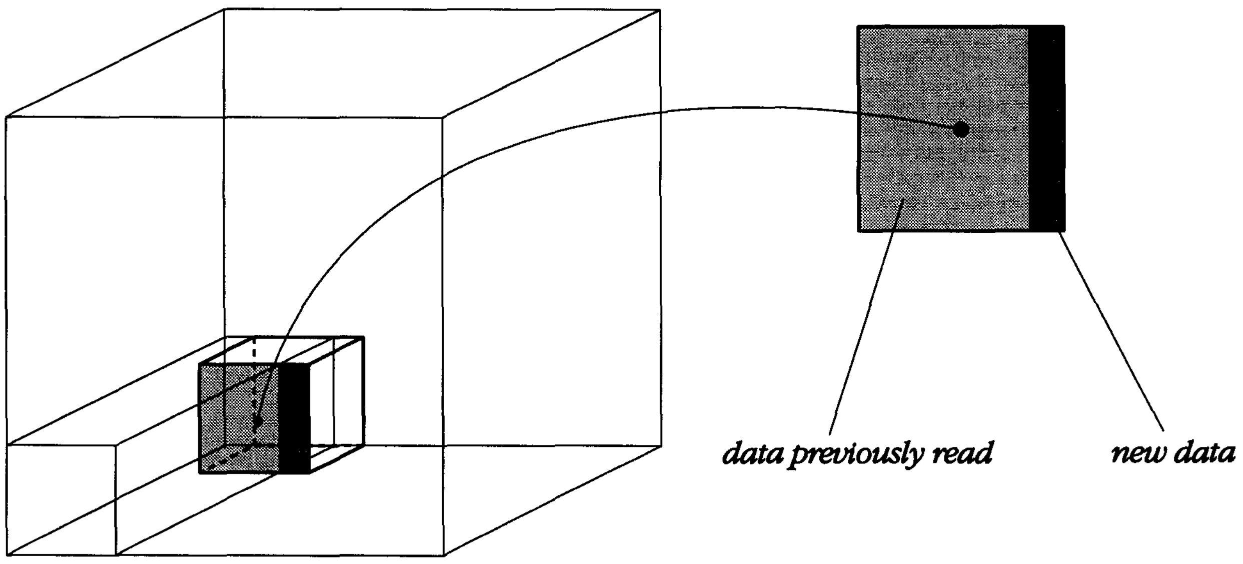


Figure 2.7 - A secondary cache delivers previously read data

the next y coordinate, it will read a n-1 × n-plane from the cache bar and add to it a 1 × n-line, read from the data set memory. The resulting n×n-plane is fed into the cache cube, and part of it (n-1 × n) is written back to the cache bar for use on location (x; y+2, z). The arbitrary plane shift is thus converted to a fixed plane shift, which can be implemented with just wires. A translator is still necessary, but its complexity has been reduced. The 1×n-line coming from the data set will have to be roll-adjusted because of movement in the z direction. This can for example be done with n times a n-to-1 multiplexer, which is feasible.

2.2 Design Details

2.2.1 Design Details of the Filter Operator

The filter kernels are connected to the interconnector, which is is a wire-only unit that routes the kernels to the data they need to calculate their particular filter value. The total number of wires in the interconnector is equal to the multiplication of the number of kernels, the number of filter points per kernel and the number of bits per filter point. This rapidly adds up to tens of

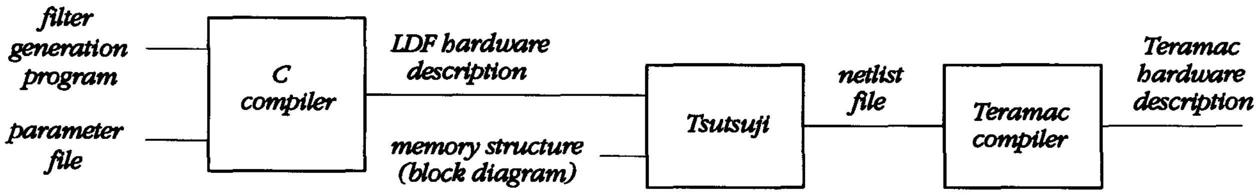

filter generation program
parameter file
C compiler
LDF hardware description
memory structure (block diagram)
Tsutsuji
netlist file
Teramac compiler
Teramac hardware description

Figure 2.8 - Interaction between compilers

thousands of wires. It is desirable that the filter definition can be easily, quickly and reliably changed in order to study the performances of different filter definitions. Therefore a filter generator program was written in C that produced a circuit description in LDF for the interconnector using several parameters. To allow the number of kernels to change automatically too, the program was later expanded to produce a circuit description for the combination of the cache cube, the interconnector, the kernels and the maximizer. Figure 2.8 shows how the different compilers communicate.

<u>Design Details of the Interconnector</u>

The interconnector links the n^3 cache cube registers to the kernels and consists solely of wires. It has n^3 byte-sized inputs (some of which will be unused since the local environment is spherical) and as many outputs as there are orientations and radii. The routing is performed by the filter generator program. For every radius and orientation the byte registers that correspond to the filter ring points are routed to a kernel.

A simple optimization is performed. If one point will be both added and subtracted, it need not be routed. Instead, the two kernel inputs are grounded. The Tsutsuji-compiler will then optimize the carry-save adder, so that it uses less hardware. The effect that a filter point from the outer ring and one from the inner ring come from the same cache cube register happens at small radii. However, it turned out that the amount of hardware that is saved by this optimization is rather insignificant.

A different interconnection algorithm has been implemented that is aimed at a reduction of the number of wires in the interconnector. When the cache cube registers take up a significant part of the hardware (which is the case, for example, for the implemented 7x7x7 design on one

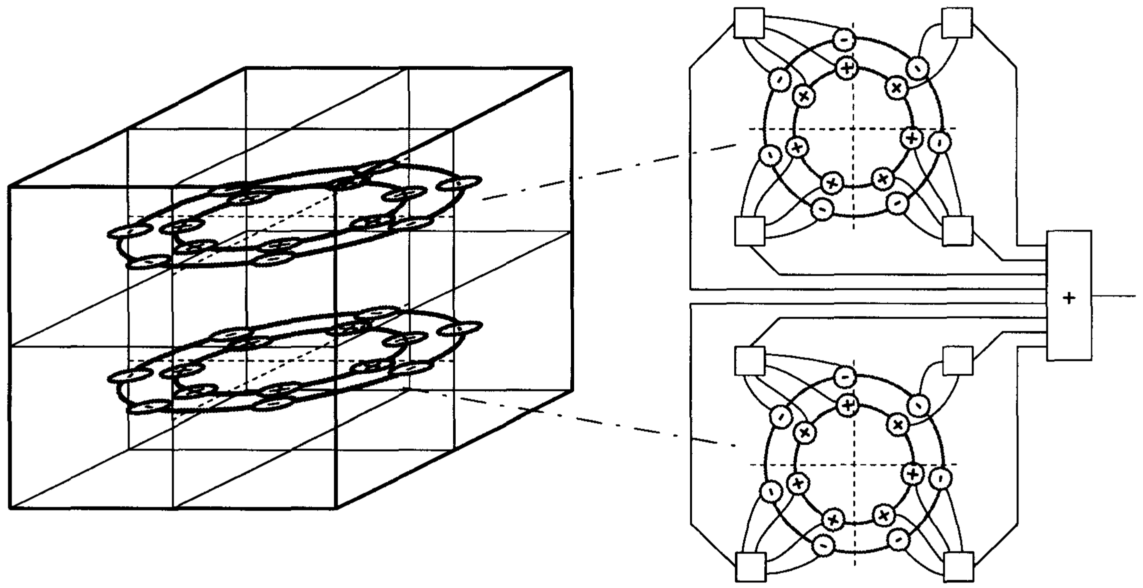


Figure 2.9 - Sectioning the cache cube and performing local calculation

Teramac board) the wiring gets extra difficult for the Teramac router. The cache cube registers will most probably have been placed all over the available hardware, whereas the separate kernels are likely to be locally concentrated. Because the interconnector has to route the registers to the kernels, each wire is routed from a global location to a local one, which requires long wires. Using a different algorithm, in which the kernels are split up and become part of the interconnector, it has been tried to reduce this wire length. The cache cube has been divided into eight sections, as shown in Figure 2.9. All points for one filter orientation and radius that are in the same section are locally processed with a section kernel, which has the same architecture as a full kernel. From each section the result from the section kernel is routed to the main adder. Only this last step is a routing from global to local. Due to the reduction in the number of wires, caused by the additions in the section kernels, there are less long wires. The price to pay for this is the increased use of hardware, as the sum of the section kernels and the main adder are less space efficient than a single kernel. Therefore this strategy is expected to be useful only for designs in which the routing of the hardware is a bigger problem than the amount of hardware (this is usually the case). Please note that this algorithm only provides the possibility for cube division. It is left to the Teramac compiler to actually place local units close together.

Design Details of the Kernel

The most straightforward way to create a filter kernel is shown in Figure 2.10. The two adders at the top are carry-save adders, which are efficient structures for adding many numbers together. The minus element is a two's complementer. The filter at the end of the kernel prevents negative filter values from being output. These values exist, but give no extra information as a zero filter value is not better than a negative filter value. Making all negative values zero allows the use of a simpler maximizer that need not distinguish between negative and positive values.

A different kernel structure is shown in Figure 2.11. It consists of only one carry-save adder. The inverters in input bus B will not occupy any Teramac hardware, because the compiler can adjust the PALE lookup table so that the PALE can take the uninverted signal as input. The inverters are a one's complement replacement of the two's complementer in the previous design. The inverted signals are added to the uninverted signals from bus A. This will lead to an output signal whereto inputs from bus A make a positive contribution, and signals from bus B a negative

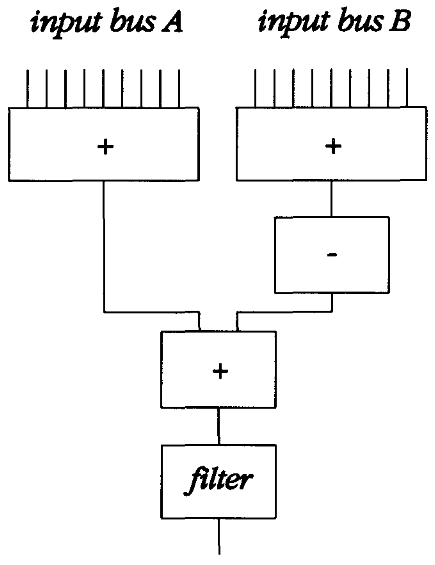


Figure 2.10 - The basic filter kernel

one. The difference with the previous filter kernel however is an offset of the zero value. Each input in bus B causes an offset of 2^b-1, where *b* is the number of bits each input has. An offset in the zero value has no effect on the rest of the circuit, so it need not be adjusted. It could be readjusted at the end of the maximizer, or this task could be left to the computer that processes the resulting filter values.

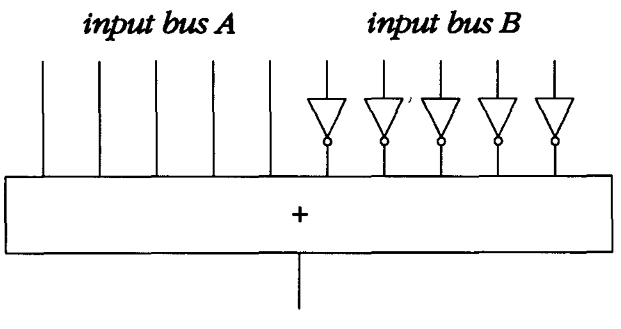


Figure 2.11
Filter kernel consisting of only a carry-save adder

Because the filter kernels take up most of the hardware that is required for a filter design, the second kernel, which has the smallest size, was chosen.

Design Details of the Maximizer

The maximizer is structured as a tree, as is shown in Figure 2.12. Every maximizer unit has four inputs, two of them contain the filter values that need to be compared and the other two contain an identification number. The maximum of the two filter values is output, as is the identification number that belongs to this maximum. In the topmost level the identification numbers are hard-wired, on the lower levels the numbers are taken from the maximizer one level higher.

If the total number of values on any level is odd, a buffer is added that passes its input value and identification number through to the output. This is done to make the structure pipelinable. The current design uses a pipeline register after every level, not because the separate maximizer units have such a long propagation time, but because this was an easy way to implement the maximizer, the pipeline registers are "free" (they exist in every PALE) and a long pipeline has few disadvantages.

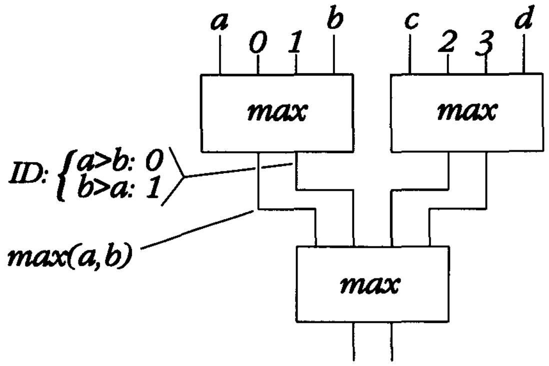


Figure 2.12 - Three maximizer units

An alternative maximizer unit was developed that provided automatic identification generation. It works as follows: A layer receives an ID of *b* bits, adds one bit to it in the most significant position, and outputs *b*+1 bits. The value of the added bit is determined by which of the two input values was the maximum. Buffers always add a zero bit and are placed at the "high" end of a level only, i.e. at the end where the kernel that will produce the highest ID-number also is. The advantage of automatic ID generation is a reduction in size of the multiplexers that select between the two IDs. Due to confusion about an assumed design error this unit was not used.

A fundamentally different maximizer structure was designed that was not based on a tree, but compared all inputs in parallel. It has been tested and worked, but turned out to have the same complexity as the tree structure. The amount of logic consumed was slightly less, but in the same order of magnitude as that for the tree structure. The tree structure was eventually chosen because of its simplicity.

2.2.2 Design Details of the Memory Structure

The memory structure has been implemented for a 7x7x7-filter meant for a single board. It has been designed with block diagrams in Tsutsuji, which can be found in the appendix. The top level of the design (the first page in the appendix) will be discussed here.

Filter module

The key element in the design is the *filter* module (middle right in the circuit diagram), of which the LDF-description has been generated by the filter generator program. The *filter* module already includes the cache cube registers, so the input to the module consists of a plane of 7x7 bytes (392 bits). Its outputs are the maximum filter value and corresponding identification for one voxel in the data set.

Cache bar

The top three memory banks (top left) hold the cache bar. Teramac does not have enough registers to store the entire cache bar, so the data is held in memories. The fourth (bottom) memory bank in the diagram contains the complete MRI data set and also holds the results (in a different part). From the results (13 filter value bits and 5 identification bits) only the top 8 filter value bits are written to memory. This was done for compatibility with existing volume visualisation tools that expect one byte of data per location.
Due to the limited memory bandwidth, four memory cycles need to be performed to get the necessary data. For the cache bar this means that the bytes are fetched from the memories two bits at a time for four times. These bits are joined together to bytes in the module *sertopar*. After the fourth memory cycle the complete 6x7 plane of bytes can be read from the *parout* output. This is merged with the newly read 1x7 line from the main memory and fed into the filter. The *sertopar* module also has a serial output that contains a delayed copy of the data that was read at the serial input. This serial data is joined with the serialised 1x7 line data, so that a serial version of a 7x7 plane is created. This plane is shifted in the negative y direction and the resulting 6x7 plane is written back to the cache bar at the addres that is four less than the read address.

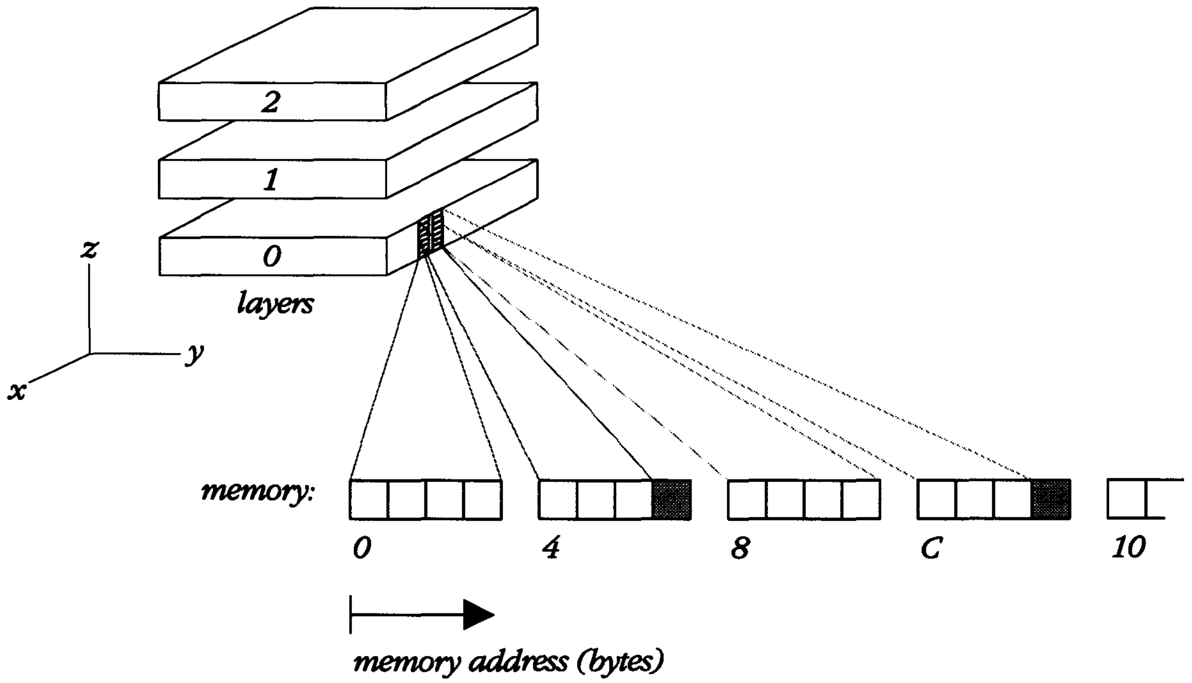


Figure 2.13 - Data format for Teramac memories

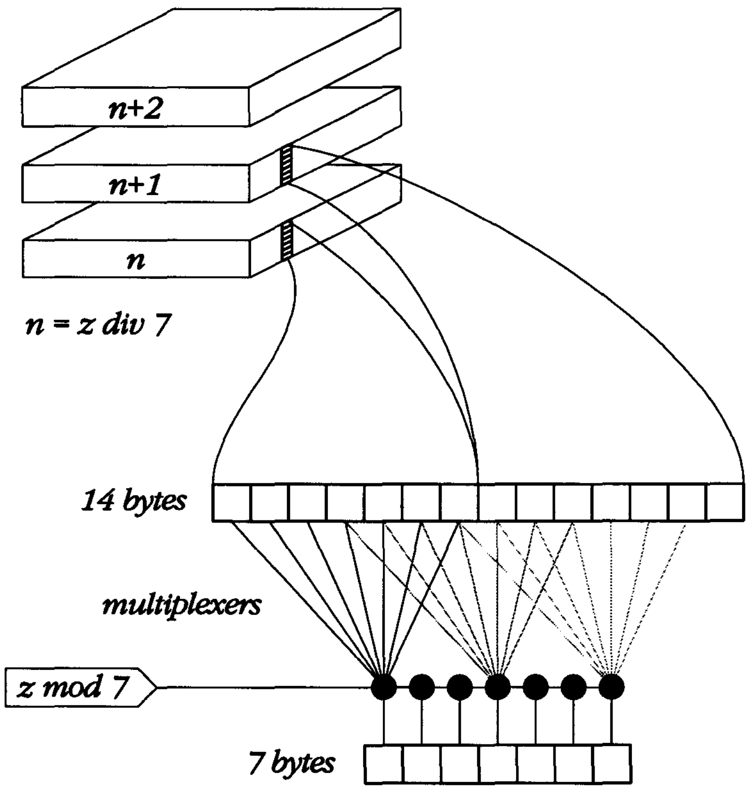


Figure 2.14 - The adjuster

Main Data Set Memory

The *adjuster* is a one dimensional translator (or roll) unit, needed to undo the shift effect generally occuring at the output of parallely used memories containing distributed data (described in 'The Memory Structure'). The *y* cache needs seven data bytes that are organised in the *z* direction in the data set. Because only four read cycles are available, it is not possible to do a read action for every required byte. Therefore, the data has to be arranged in such a way that a 32 bit read operation yields more than one usable byte, which means that bytes that are likely to be needed at the same time should be grouped. A data format has been chosen (see Figure 2.13) that stores seven points in the *z* direction in two consecutive 32 bit words (eight bits remain unused). Most of the times data will be needed from two different layers. To allow this all four memory cycles are used; two to read the seven points from the layer that contains the points with the lowest *z* coordinates and two to read the seven points from the layer above. The *adjuster* module selects from the set of fourteen the required seven bytes (see Figure 2.14). These bytes are then added to the 6x7-plane of data coming from the cache bar.

Counters and Address Calculators

Several counters provide the necessary addressing signals. The *phasecounter* counts the four phases of the memory cycle. In phase three the four memory accesses have been performed and byte-sized outputs are available at the input of the *filter* module. The enable input that allows the registers in the cache cube to change to a new value is therefore connected to the phase three signal. The *xyzcounter*, which holds the coordinates of the point in the data set that is currently processed, is incremented at the end of phase three.
The coordinates *x*, *y* and *z* and the current memory phase are used by the module *zadr_proc*, which calculates the read address for the main data set memory. This memory uses a *z* address that is divided by seven, because the data format contains chunks of seven bytes in the *z* direction. The modulo seven of the *z* address is wired to the adjuster, which it uses to select the right seven bytes from the fourteen read from memory.
The *x*, *y* and *z* coordinates are also used to form the write address of the main memory, at which the filter value results are written. However, the filter contains a delaying pipeline, so the addressing signals also need to be delayed. This is done with a simple register sequence in the *delayer* module. The *wadr_calc* module splits the byte address into a word address and a byte enable signal (the byte that carries the filter value result has been copied so that a 32 bit word contains four copies of this byte).
The *x* coordinate and phase signal form the cache bar address at which the 6x7 plane is read. The *x* coordinate selects the address, the phase bits select which quarter of the bytes in the plane is read. The *minus4* module ensures that, after a new 6x7 plane is calculated, it is written back at the previous *x* location.
When the *xyzcounter* reaches its final value, the *rdy* output is asserted. To allow the pipeline to flush, the filter then waits for a specific *x* coordinate before sending a breakpoint signal which stops the Teramac clock.

3. Results

The 7x7x7 filter design with 18 kernels was implemented on a single Teramac board. Two 128^3-sized input files were created as test matrices, one synthetic set containing a straight tube and one MRI set containing authentic data of the head. Both sets where processed by the Teramac filter and by the original software algorithm. The results were compared bit-by-bit by a comparison program and found to be identical (the integer values that the software program produced were treated the same way as the results in the Teramac design, i.e. from thirteen bits only eight bits were used in the comparison with the eight bits of the Teramac design). The least significant five bits and the identification bits were not tested.
The compiler predicted a maximum clock frequency of 600 kHz, which was used for testing the design. Higher frequencies were tried but the design failed at around 640 kHz, which shows that the frequency prediction by the compiler was accurate.
The time to process the 128^3 dataset with Teramac at 600 kHz was 14 seconds. Calculation time for the software algorithm was 55 seconds on an HP735 workstation. The speed difference is approximately a factor of four.
The time taken to compile the design was approximately a second for the filter generator program, a minute for the Tsutsuji compiler and a couple of minutes for the Teramac compiler.
The main bottleneck in the design turned out to be the interconnection. The size of the filter design was limited by the amount of interconnection available in Teramac, not by the amount of logic available.

4. Modifications and Expectations

This chapter discusses various optimizations that can be done to improve the performance or reduce the amount of hardware required. Also methods are described that only appear to offer an improvement as an introduction to methods that really do. Expectations about attainable speeds are given at the end of the chapter.

4.1 Modifications

4.1.1 Speed Improvements

Pipelining the Carry-Save Adders
In the pipelined filter design the wires that cause the longest delays (the ones that pass most router chips) are probably the ones from the cache cube to the kernels. It is therefore desirable to place pipeline registers as early in the kernels as possible. Placing registers before the kernels is inadvisable, because the number of wires there is very high. The best solution would be to create a new carry-save adder that has pipeline registers right after the first stage of its tree structure and possibly also ones after lower stages. These registers come "free" with every PALE, so this requires no extra hardware.

Duplicating Structures to Prevent Long Wires
Some addressing and enabling signals are used throughout the design and may therefore become quite long. It is possible that one of these signals determines the maximum clock frequency, although the rest of the design could be clocked a lot faster. Also, these long connections can use up wires in areas where the density of available wires is low. For example, when using a multiboard design, the addressing lines to all the memory banks may consume quite a few of the limited inter-board wires. Duplicating structures can solve both these problems, at the expense of increased hardware requirements. In the multiboard example each board can have its own address generator with a separate reset input, which saves having to wire the addresses over the interboard busses.

Two Voxels in Parallel
Another way to improve speed is to calculate the filter value for two neighbouring voxels in parallel, by using twice as many kernels. The two voxels share most of their local environment, so only a slight increase in cache cube registers and in memory bandwidth is needed. The important disadvantage of this method is that the registers in the cache cube are now connected to more kernels, so the wiring complexity increases.

Multiple Cache Cubes
Since in most designs the wiring complexity is the most limiting factor, the reverse of the method mentioned in the previous paragraph can be applied. The cache cube is duplicated and both copies are fed with the same data stream from memory. Each cube has been wired to half the necessary number of kernels. This results in a lower wiring complexity per cache cube, at the expense of doubling the number of cache cube registers. The memory bandwidth remains the same, because both cache cubes receive a copy of a single data stream.

4.1.2 Fitting More Kernels

Multiplexing
To allow the calculation of more filter values than there is hardware for kernels, it is possible to use multiplexing. The interconnector does not connect to the kernels directly, but to a large 2-to-1 multiplexer, that connects to half the number of kernel units. Two cycles are necessary to compute all filter values. The important disadvantage is that the wiring remains the same. All wires previously connected to the filter kernels are now connected to the multiplexer. The hardware saved by using less kernels does not come available, as it is needed for routing.

Bitserial arithmetic
Another method to calculate more filter values than hardware allows is to use bitserial arithmetic. Instead of the full eight bits of a byte, only one, two or four bits are wired from the cache cube to the kernels. To accept the byte fractions the kernels have to be adapted, which should make them smaller, allowing more kernels to be placed. The kernels produce a single 13 bit wide output so the relatively small maximizer tree can remain the same. The cache cube has to be adapted slightly, but will not change in size. Basically, this method is another way of multiplexing, which, however, saves on both kernel hardware and interconnection wires.

4.1.3 Other Modifications

Removal of Data Set Conversion and Interleaving
Currently the data set needs to be converted and interleaved to a format that the tube filter uses. By changing the movement direction of the filter through the data set this conversion becomes unnecessary. Currently the order of movement is $x \cdot y \cdot z$ (x is the fast moving variable). Unconverted datasets are organised $x \cdot y \cdot z$. In the tube filter the first and the second movements are cached (by the cache cube and cache bar respectively) and multiple points in the third direction (z) need to be fetched from memory. To be able to access these points fast the data set has been converted so that points in the z direction are placed next to each other.

Conversion becomes unnecessary if the order of movement of the filter is changed to $z \cdot y \cdot x$. The z and y direction are then cached by the cache cube and bar respectively, and the necessary multiple points in the x direction can immediately be accessed in the data set. The modifications in the design to implement a change of movement direction are confined to changing some signal names and swapping some address lines in the module that calculates the address of the calculated filter values.

Parameterising the Memory Structure

The design of the filter operator is fully parameterised, but the memory structure is specifically designed for a 7x7x7 cache cube and meant to be run on a single Teramac board. To allow maximum freedom in the choice of tube filters and the use of Teramac hardware a program could be written that automatically produced the memory design as a function of several parameters (size of cache cube, amount of memories available, number of pipeline stages in the filter), just like the filter design is produced by the filter generator program. This structure could then be used as a general purpose three dimensional filtering tool, which accepts any filter module that uses a three dimensinal local environment as input and produces a single value as output.

4.2 Expectations

Currently the Teramac design outperforms an HP735 workstation by a factor of four. If the 7x7x7 design is copied and implemented on eight boards, each board can work on an eighth of the data set, improving the speed by a factor of eight. Therefore, a factor of 32 times workstation speed is without doubt possible.
If a new 7x7x7 design is created that uses all eight boards, the four memory cycles, introduced because of limited memory bandwidth, may be replaced by one. So, use of the extra memories that an eight-board system offers will already lead to an improvement factor of four. The amount of logic that can be implemented is increased with a factor of eight, so theoretically eight times as many kernels can be calculated, or eight voxels can be processed in parallel. In the single-board design part of the logic is consumed by the cache cube. In the eight-board design the cache cube can be distributed across the boards, giving an extra improvement of approximately 15%. All this would lead to a speed that outperforms a workstation by a factor of almost 150.
Implementation hereof is, however, not trivial. In the tube filter the cache cube, distributed over Teramac, has lots of connections to local units, the kernels. This routing from global to local units requires a lot of global interconnect in the form of interboard wires, which are not available in large quantities. It is expected that the use of multiple (eight) cache cubes allows the routing of the biggest and therefore fastest designs.

5. Conclusions

A software algorithm for artery extraction from MRI data has been converted to a logic circuit, which was implemented on a configurable custom computing machine, Teramac.

The filter design was parameterisable, in that a program was created that generated the hardware description of the filter according to certain filter parameters.

The interface between the memories and the filter could not be changed with parameters, but was fixed to handle a 7×7×7 design on a single Teramac board only.

The design was implemented on one Teramac board and outperformed the software algorithm on an HP735 workstation by a factor of four.

Designs for eight boards are expected to run at more than 100 times the speed of a workstation.

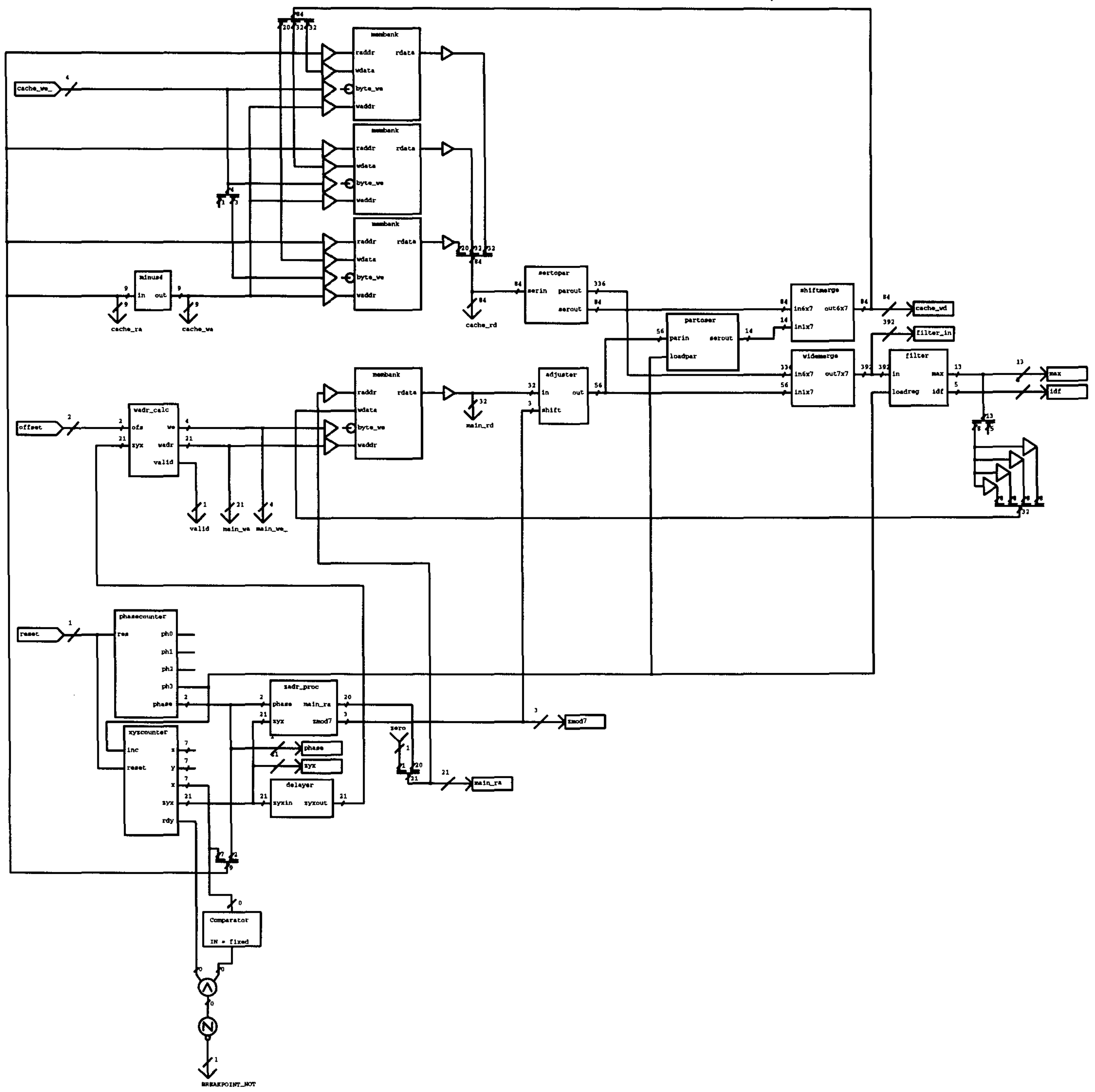
6. References

[1] D. Vandermeulen, D. Delaere, P. Suetens, H. Bosmans, G. Marchal, "Local Filtering and Global Optimisation Methods for 3D Magnetic Resonance Angiography (MRA) Image Enhancement", SPIE Visualization in Biomedical Computing, Volume 1808, pages 274-288, October 1990.

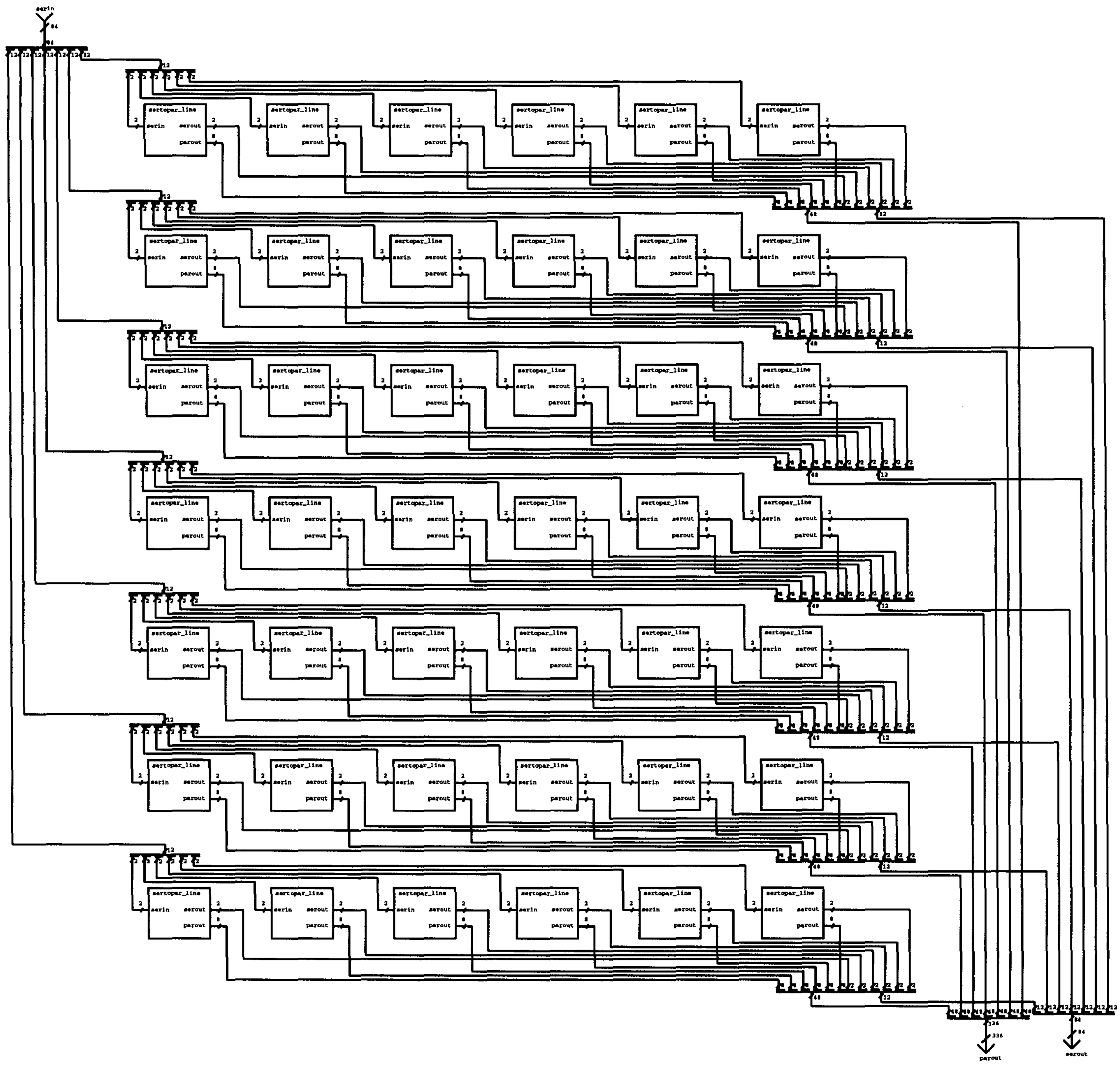
[2] R. Amerson, R.J. Carter, W.B. Culbertson, P. Kuekes, G. Snider, "Teramac – Configurable Custom Computing", IEEE Symposium on FPGAs for Custom Computing Machines, April 1995.

[3] W.B. Culbertson, T. Osame, Y.Otsuru, J.B. Shackleford, M. Tanaka, "The HP Tsutsuji Logic Synthesis System", pages 38-51, Hewlett-Packard Journal, August 1993.

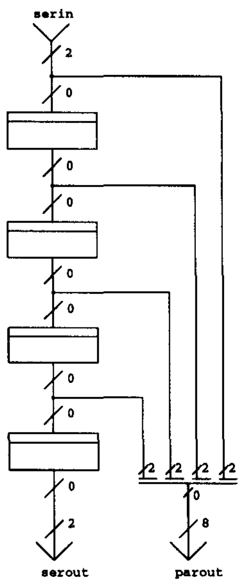
Appendix



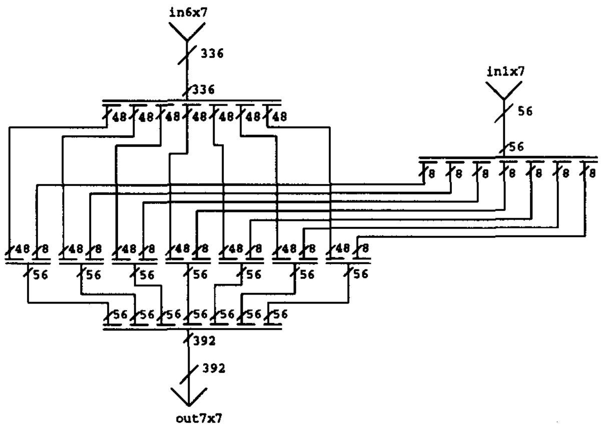
7x7x7 – the top level of the memory structure


serin
84
sertopar_line
serin
serout
parout
2
8
12
48
336
parout
serout

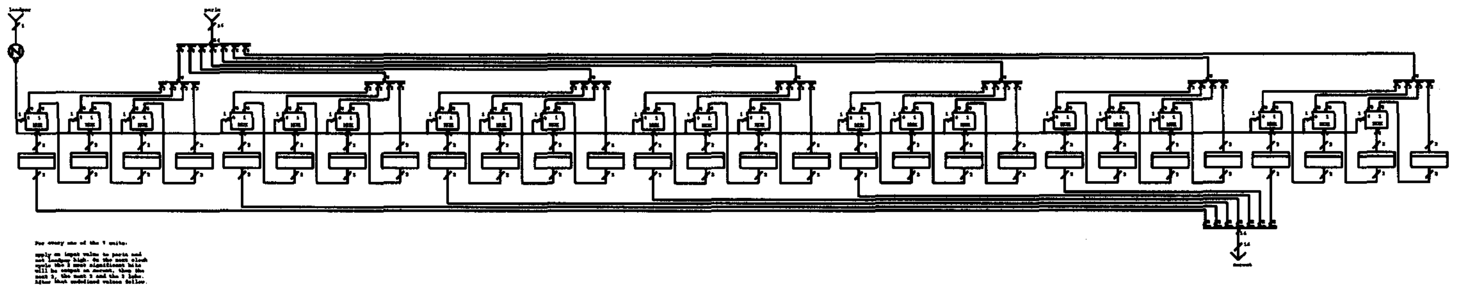
sertopar



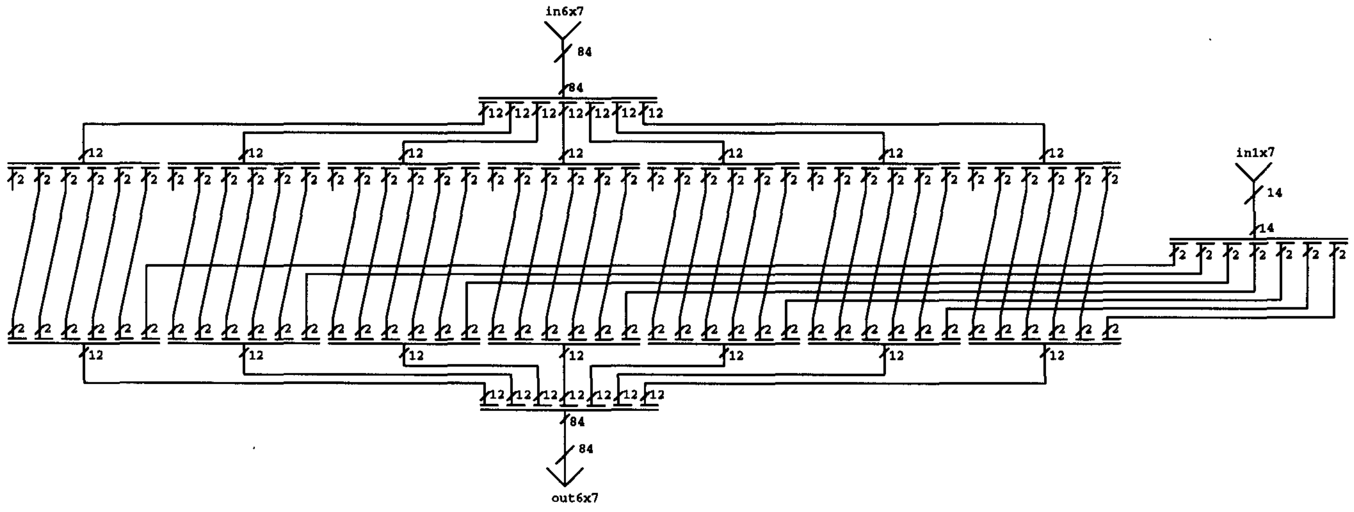
sertopar_line



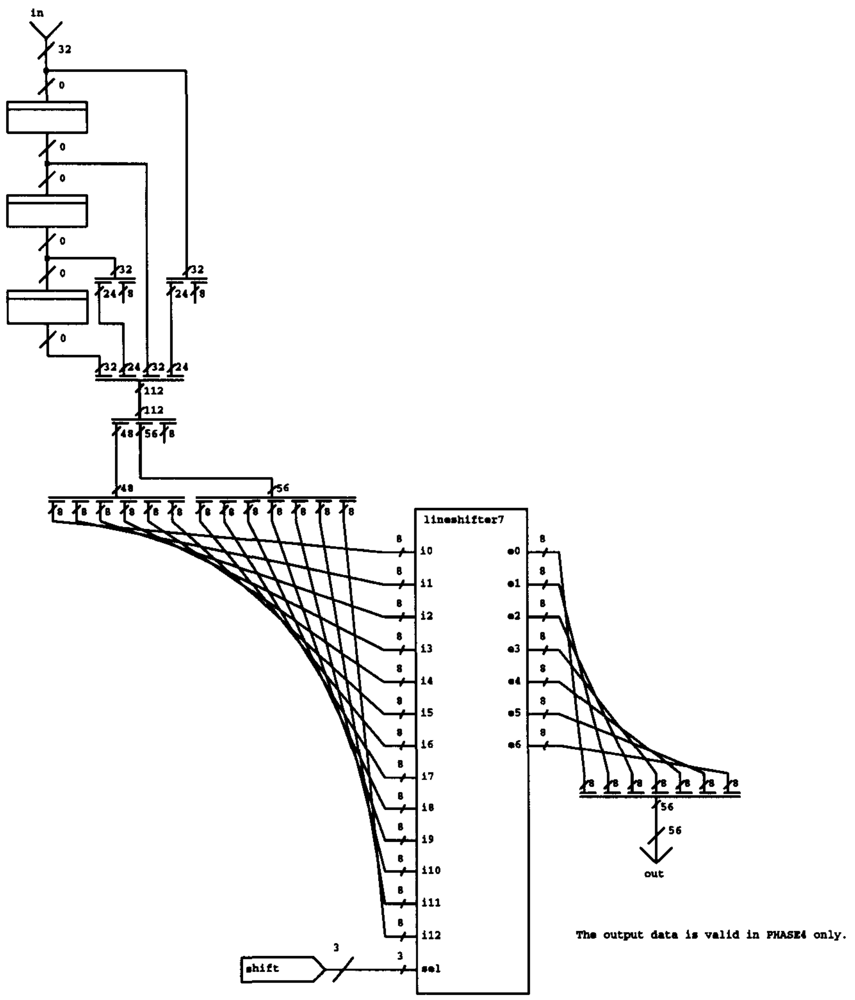
widemerge


For every one of the 7 units:
apply an input value to parin and set loadpar high. On the next clock cycle the 2 most significant bits will be output on serout, then the next 2, the next 2 and the 2 last. After that undefined values follow.

partoser

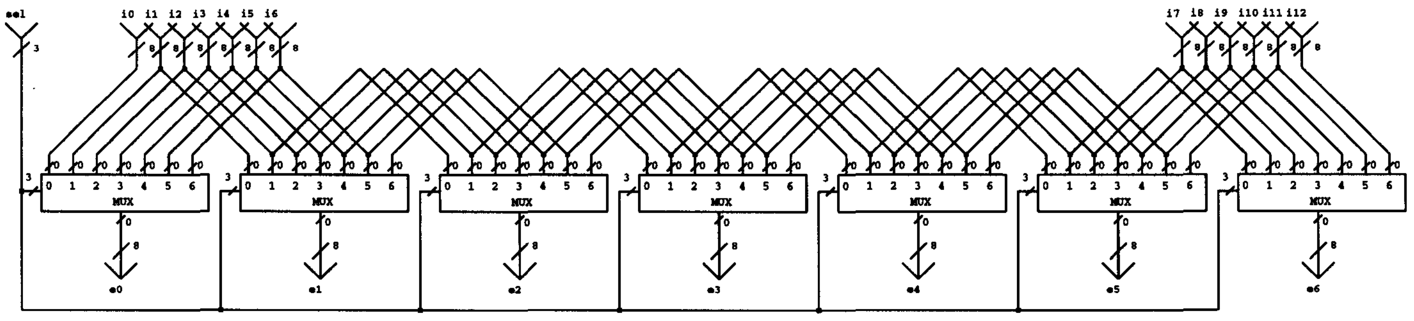


shiftmerge



The output data is valid in PHASE4 only.

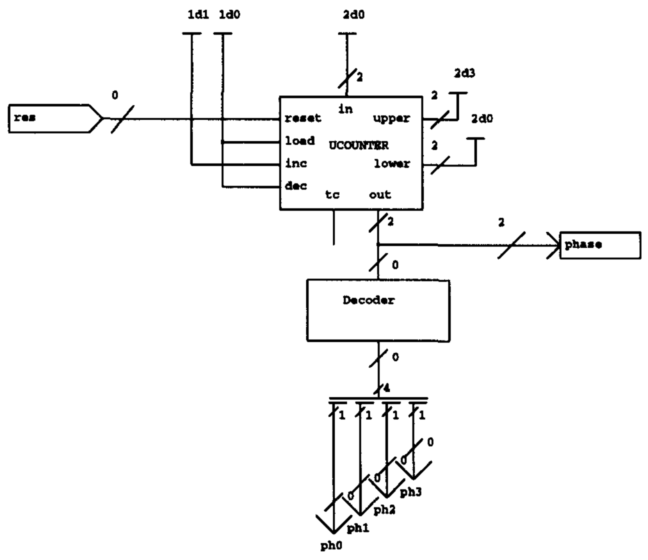
adjuster



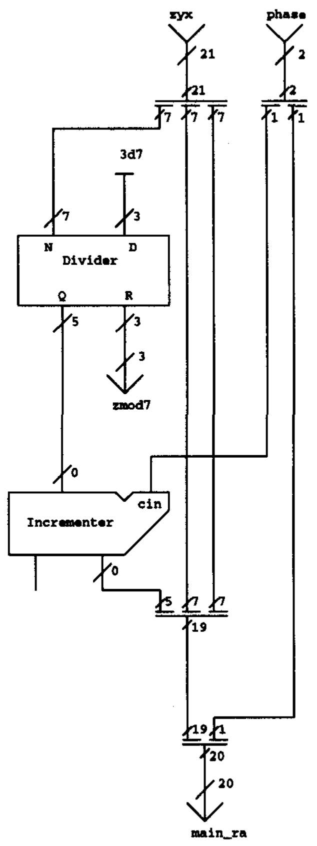
NB: e(x) = i(x + sel)

eg: if sel=3
e2 = i5
e4 = i0

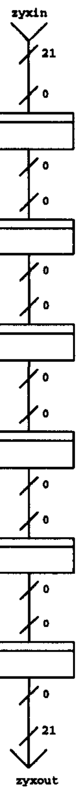
lineshifter7



phasecounter

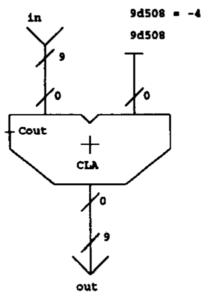


zadr_proc

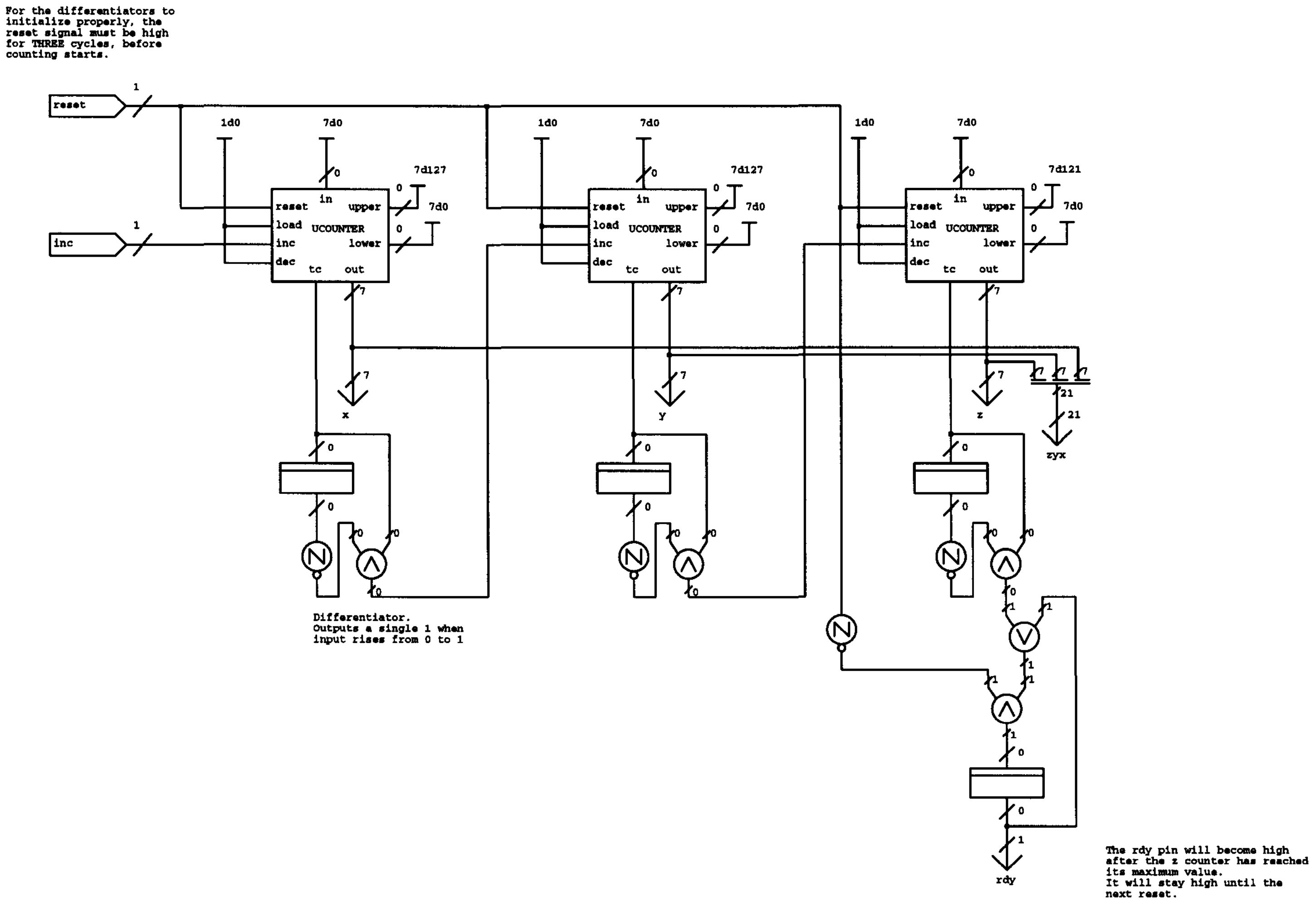


Delayer deals with the delay in the filter module caused by pipelining. The number of registers here is equal to the number of pipelinestages in the filter module.
NB: the whole registercube is to be counted as one stage.

delayer



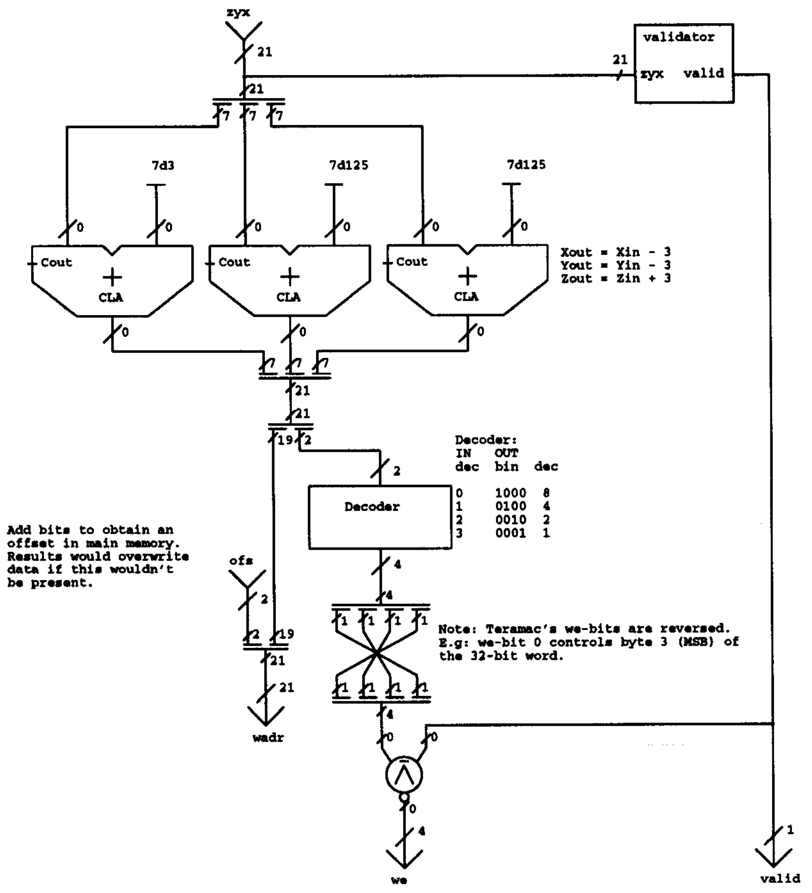
minus4


For the differentiators to initialize properly, the reset signal must be high for THREE cycles, before counting starts.

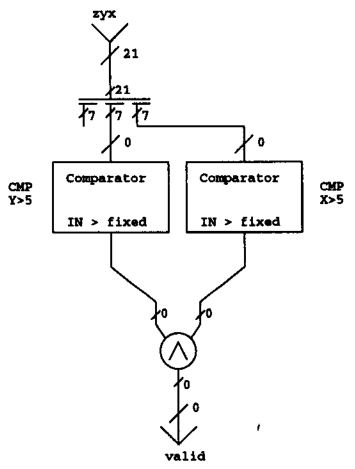
Differentiator. Outputs a single 1 when input rises from 0 to 1

The rdy pin will become high after the z counter has reached its maximum value. It will stay high until the next reset.

xyzcounter


zyx
validator
zyx valid
7d3
7d125
7d125
Cout
CLA
Xout = Xin - 3
Yout = Yin - 3
Zout = Zin + 3
Decoder:
IN OUT
dec bin dec
0 1000 8
1 0100 4
2 0010 2
3 0001 1
Decoder
Add bits to obtain an offset in main memory. Results would overwrite data if this wouldn't be present.
ofs
Note: Teramac's we-bits are reversed. E.g: we-bit 0 controls byte 3 (MSB) of the 32-bit word.
wadr
we
valid

wadr_calc



validator