Exploiting Typical DSP Data Access Patterns and Asynchrony for a Low Power Multiported Register Bank

M. Lewis
Ericsson Microelectronics AB
Isafjordsgatan 16
S-164 81 Kista, Sweden
mike.lewis@mic.ericsson.se

L. Brackenbury
AMULET Group,
Department of Computer Science,
University of Manchester, Oxford Road
Manchester M13 9PL, UK
lbrackenbury@cs.man.ac.uk

Abstract

CADRE (Configurable Asynchronous Dsp for Reduced Energy) is a low-power asynchronous DSP (digital signal processor) architecture intended for digital mobile phone chipsets. Central to the architecture are the X and Y register banks, which supply the four processing units with the data they require and to which results are written. The register banks each require 10 read and 6 write ports to service all possible requests, leading to a large and power-hungry unit if implemented directly. Instead, typical DSP data access patterns are exploited to produce a partitioned design which offers fast and low-power operation in typical cases but also caters for worst-case patterns. Power consumption and performance results for the register bank with the DSP running typical algorithms are presented, and it is shown that the register bank consumes only 8% of total power (core and memory) in what is already a highly power-efficient system.

1. Introduction

CADRE is a 16-bit DSP architecture intended for low-power embedded applications such as digital mobile phone chipsets. The design of the architecture is based on the principle that a reduction in power consumption can be traded for an increase in die area by providing multiple processing elements, which allow the voltage to be reduced while maintaining throughput (so-called *architecture driven voltage scaling* [1]). The CADRE architecture contains 4 functional units (ALUs), as shown in Figure 1, each operating with an average period of 25ns to give a total throughput of 160 million operations per second. A fuller description of the architecture can be found in [2].

One of the challenges in the design of the architecture is to supply the functional units with data at a sufficient rate while minimising the associated power consumption. This power consumption is made up of the power consumed within the main memory RAM units themselves, and the power required to transmit the data across the large capacitance of the system buses. Memory accesses can form the largest component of power consumption in data-dominated applications [3], and a study of the Hitachi HX24E DSP [4] showed that memory accesses caused a significant proportion (~20%) of the total power consumption even where the activity of the system is not dominated by memory transfers. As technologies scale further into the deep sub-micron region, the proportion of the power consumption associated with memory transfers will increase, due to the increased relative cost of driving long interconnections.

Fortunately, DSP programs tend to display very strong locality of reference, so the memory hierarchy approach can work very well to both reduce power consumption and increase operating speed. For this reason, a large register file made up of two 128 word banks (labelled X and Y) is included in the design. Many DSP algorithms map naturally onto two separate banks, e.g. data in one bank and filter coefficients in the other, or the real part of data in one bank and the complex part in the other.

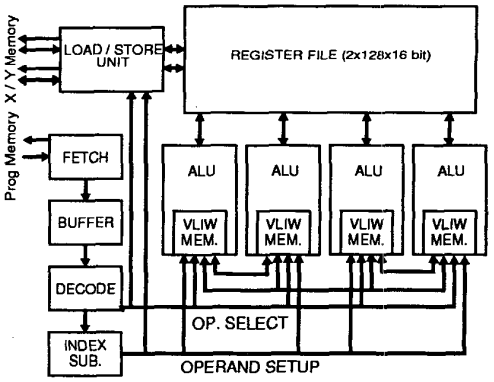

Figure 1 Block Diagram of CADRE

Having a large explicit register file has a number of advantages when compared to the alternative of having few local registers and a cache between main memory and the DSP. In a traditional DSP architecture, data is located in memory by means of address registers whose contents are updated by address generation units (AGUs). The AGUs update the addresses in parallel with arithmetic instruction execution, in the pattern required by the algorithm. These generally support features such as circular buffers and bit-

1522-8681/01 $10.00 © 2001 IEEE

reversed addressing, and have the same number of bits width as the maximum addressable space of the processor (24 bits for CADRE). CADRE can require up to 8 operands for the functional units per cycle, with each access potentially requiring an address update. The hardware required to perform these 8 updates would clearly represent a significant area and power overhead if implemented by 24 bit AGUs.

Having an explicitly addressable register file allows data to be located in the register file by means of 7-bit index registers. These can be updated much more quickly with much lower hardware and power cost, than the 24-bit wide address registers. In the CADRE architecture, address registers are only used for loading and storing data between the register file and memory, with transfers of 16 or 32 bits allowed from each bank per operation.

The notation used in the paper is that a store involves the reading of a value from the register bank and its transfer to memory, while a load involves the reading of a value from memory and its transfer to the register bank.

2. Register bank design

A typical multiported register cell with n read and m write ports is shown in Figure 2. The data is stored by the cross-coupled weak inverters. Each read port connects to one bit line (*Nop1...Nopn,* which go to all of the cells at that bit position in the register bank) on which the read value is placed, and one word line (*en_op1...en_opn,* which go to all of the cells in that word of the register bank) through which the word to be read from the register bank is selected and which enables the precharged bit lines to be discharged depending on the contents of the register cells. An example of how the bit and word lines are connected is given in Figure 3. Each write port connects to one word line, (*en_w1...en_wm*) selecting the word to be written and enables the value stored on the bit line (*wb1...wbm*) to be driven onto the weak inverters.

By necessity, the read and write transistors are larger than those for the weak inverter, as the read ports drive the large capacitance of the bit lines and the write ports need to overdrive the weak inverter. It is therefore the number of ports which control the overall size of the register bank. The physical size of the register bank dictates the length of the bit lines, and it is the charging and discharging of these lines which represents one of the major sources of power consumption in the register bank. It is claimed [5] that, if the size is limited by the wiring pitch of both the bit lines and word lines, that the area of the register bank can be expected to increase quadratically with the number of ports. Consequently, despite a number of power saving measures that can be employed, the register bank is likely to be a major component of the power consumption.

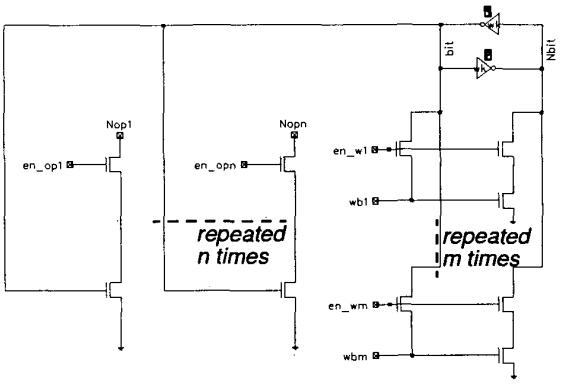

Figure 2 Multiported register cell

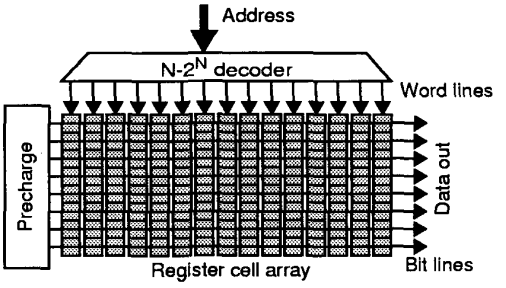

Figure 3 Word and bit lines in a register bank

One way of avoiding the energy and area cost of a large centralised multiported register bank is to divide it into a number of smaller banks, each of which are associated with a smaller number of processing elements. However, this requires that data access patterns can be mapped onto this configuration, and adds additional complexity for the programmer or the compiler. An automatic way of performing this mapping is proposed in [6], but this adds hardware complexity and is not necessarily well suited to DSP algorithms, where individual data values tend to be processed by many or all of the functional units.

The register bank for CADRE requires 10 read ports (2 reads from each functional unit, and data to be read for stores to memory from two sequential registers aligned on an even boundary), and 6 write ports (1 writeback from each functional unit, and 2 writes to sequential even-aligned registers for data arriving due to loads from memory). The proposed design exploits the timing flexibility of asynchronous pipelines and the data access patterns of typical applications, to give the appearance of two unified 128-word register files with the requisite number of read and write ports at a much lower area and power cost than a conventional multiported register bank. It also offers the potential for faster reads than could be expected

from a conventional implementation, when using common data access patterns.

3. Data access patterns

Many DSP algorithms require access to sequential addresses, such as for sequential data values and filter coefficients, and write the results back in sequential order. When parallelized, this maps onto simultaneous requests to four consecutive addresses. Two important examples of this are the FIR filter algorithm and the calculation of autocorrelations (autocorrelations being the dominant processing component of many speech compression algorithms).

3.1. FIR filter data access patterns

A N-point finite impulse response (FIR) digital filter is characterized by the equation:

$$y(n) = \sum_{i=0}^{N} x(n-i)c(i)$$

When mapped onto four functional units, this leads to simultaneous accesses to $x(n)$, $x(n-1)$, $x(n-2)$ and $x(n-3)$ from X memory, and $c(0)$, $c(1)$, $c(2)$ and $c(3)$ from Y memory, and so on for all values of i at each data index n.

3.2. Autocorrelation data access patterns

Autocorrelation is characterized by the equation:

$$r(k) = \sum_{n=0}^{N} x(n)x(n-k)$$

When implemented directly with four functional units, this can require simultaneous accesses from up to 8 data locations. However, the situation can be improved by splitting the data into two halves with one half residing in the X register bank and the other in the Y register bank. In this way, no more than 4 reads occur to each register bank, and the final result can be calculated with a summation after processing the blocks.

Where more than one autocorrelation value must be calculated, further optimizations can be made by concurrently calculating sets of consecutive autocorrelation results to give sequential data accesses, which also minimizes multiplier switching activity by keeping one input constant over four operations. This leads to the register access patterns shown in Table 1 for each data point, with MAC A-D representing the four separate multiply-accumulate functional units. The summation can be performed in any order, and in this implementation MAC A and MAC C process even data points in the X and Y register banks respectively, while MAC B and MAC D process odd data points. In practice, the functional units in CADRE contain only 4 accumulators, so autocorrelation values for 4 values of lag k (0...3, 4...7, etc.) can be calculated on each pass through the data.

MAC A		MAC B		MAC C		MAC D		k
X:n	X:n	X:n+1	X:n+1	Y:n	Y:n	Y:n+1	Y:n+1	0
X:n	X:n-1	X:n+1	X:n	Y:n	Y:n-1	Y:n+1	Y:n	1
X:n	X:n-2	X:n+1	X:n-1	Y:n	Y:n-2	Y:n+1	Y:n-1	2
X:n	X:n-3	X:n+1	X:n-2	Y:n	Y:n-3	Y:n+1	Y:n-2	3
X:n	X:n-4	X:n+1	X:n-3	Y:n	Y:n-4	Y:n+1	Y:n-3	4
X:n	X:n-5	X:n+1	X:n-4	Y:n	Y:n-5	Y:n+1	Y:n-4	5
X:n	X:n-6	X:n+1	X:n-5	Y:n	Y:n-6	Y:n+1	Y:n-5	6
X:n	X:n-7	X:n+1	X:n-6	Y:n	Y:n-7	Y:n+1	Y:n-6	7

Table 1. Autocorrelation data access patterns

4. Register bank structure

The sequential nature of data accesses suggest that one way to improve the performance and power consumption of the register banks in this application would be to divide them into N address-interleaved sub-banks, with the sub-banks containing sequential register numbers repeating every N^{th} digit. Given that there are 4 functional units, and that operations are mapped onto separate X and Y banks, an obvious choice of N for this design would be 4, with a sub-bank size of 32. Usefully, optimised custom layout cells are available from the AMULET3 processor, which has a 32-entry register bank.

This sub-division means that sub-bank 0 contains registers $4n$, sub-bank 1 contains registers $4n+1$, sub-bank 2 contains registers $4n+2$ and sub-bank 3 contains registers $4n+3$ (with $n = 0...7$) as shown in Figure 4. Write- and read-requests are distributed to the appropriate sub-bank, by two very different mechanisms.

When the code is written so that all of the register accesses to each bank occur in different sub-banks, the power consumption and delay incurred at each read port will be that of an access to a single-ported 32-entry register file, with some overhead from the routing and arbitration circuitry. Where contention for register sub-banks exists, a number of access cycles can be performed until all of the requests have been answered. In the asynchronous domain, this represents no difficulty: surrounding stages will simply wait until the accesses have completed. The programmer need only ensure that the *average* data access patterns are good to ensure that overall performance will not be affected. By contrast, in a synchronous system it would be necessary to ensure that, at most, only a small number of

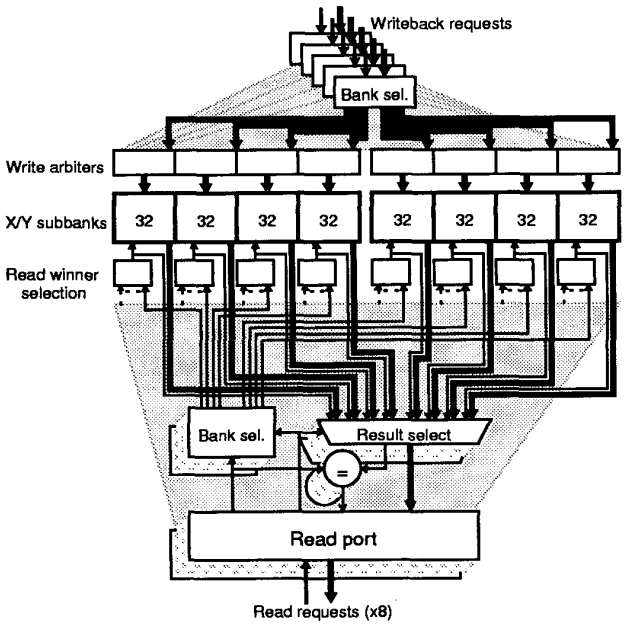

Figure 4 Register bank organization

access contentions occurred so that the operations are guaranteed to complete within the given clock period, which would be extremely difficult.

5. Write mechanism

Write-requests to the register bank arrive asynchronously: while there is likely to be some correlation between the times of writeback requests from the functional units, data returned by loads from external memory can arrive at arbitrary times. It is expected that contention for the sub-banks is unlikely between writebacks from functional units, as few algorithms write back data other than in a sequential manner. Contention is somewhat more likely between loads from memory to the registers and writebacks from functional units to the registers, since the timing of load completion is unknown and the destination register for the load is likely to be in one of the next groups of 4 registers to those currently being written back.

The chosen mechanism for distributing writes is shown in Figure 5. When a write-request arrives at one of the writeback ports, it is routed to one of the arbiter blocks in each of the 8 sub-banks. The selection is based on bit 7 (X/Y select) and bits 1:0 (sub-bank selection) of the register selection *reg[7:0]*. Similarly, the data and the address within the sub-bank (*reg[6:2]*) are also passed to the target sub-bank. A similar process occurs for arriving load completions, except that only one load can occur to each of the X and Y register banks and, when a 32-bit load is selected, the targets are either sub-banks 0 and 1 or sub-banks 2 and 3.

At the input to each sub-bank, an arbiter block accepts possible write-requests from all of the write ports, and contention for that sub-bank is resolved amongst the pending requests. The data and register selection of the winning request are passed to the sub-bank write input, and the write process occurs. Once the write has completed, the acknowledge is passed back to the winning write port, the winning request is removed and any other contending requests can gain access in whichever order that the arbiters determine.

Figure 6 shows the organization of the arbiter blocks, and the arbitration component used to construct it. At the input to each arbiter, the incoming requests vie for control of the mutex element. The winning request then gains control of the multiplexers, causing the appropriate register and data values to be passed through. It can be seen that the organization of the arbiter components is asymmetric: load completion is arbitrated after all of the writeback requests, making load completion somewhat faster and giving it higher priority. If a conflict occurs between the writebacks and incoming data on the final instruction of a loop, it is important that the new data should arrive first, so that the register read for the next iteration of the algorithm can begin. The writeback occurs in the pipeline stage following the register reads, so that the writebacks will then occur in parallel with the reading of the fresh data. If the priority were reversed, then the writebacks would complete and the execution stage of the pipeline would become empty. However, the register read in the previous stage would be unable to start until the loading of fresh data had completed, leading to a bubble being introduced in the pipeline while the read is performed.

The individual arbitration circuits are not symmetrical in terms of the delay that they impose: the multiplexers are normally set to pass input A, and if input B wins control it is necessary to delay the output until the multiplexers have changed their selections. A slightly fairer technique, which is also likely to be faster, would be to use a tree arbiter with arbitration off the critical path, such as that proposed in [7], to determine the winning request and then select the data and address corresponding to the winner (e.g. by using tri-state drivers). However, writebacks to the register bank from the functional unit accumulators are an infrequent event in the CADRE architecture, and the repeated tree structure gave a simple (and readily expandible) design

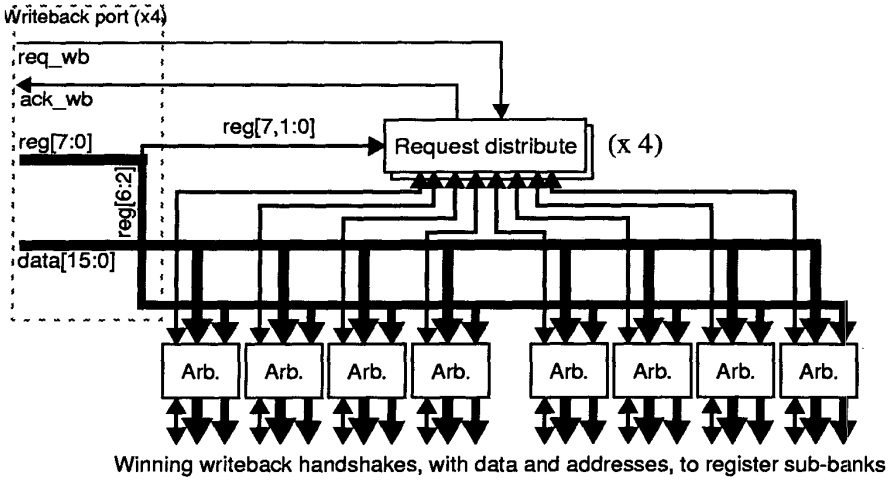

Figure 5 Write request distribution

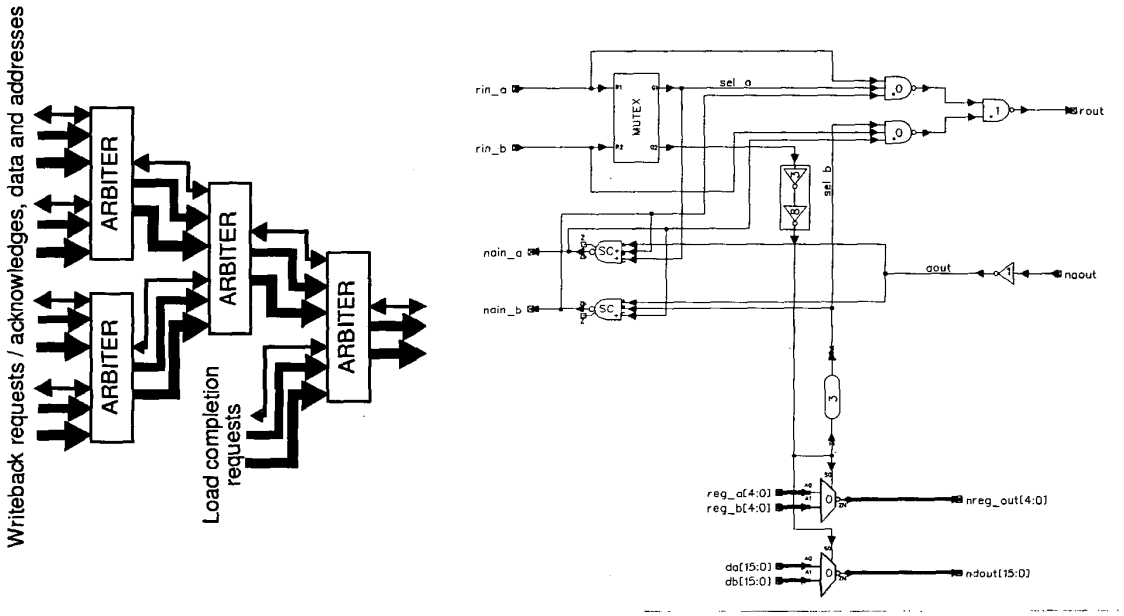

Figure 6 Arbitration block structure and arbitration component

which operated well within the target speed for typical access patterns.

6. Read mechanism

6.1. Overview

In contrast to write requests, read requests to the register banks tend to arrive at approximately the same time as they originate from a single triggering event. Also, it is very

much more likely that read requests from the functional units will conflict with one another in their choice of sub-bank, particularly in the case where they all require access to exactly the same register (as occurs in the autocorrelation example in Table 1). For these reasons, an asynchronous arbiter tree will give poor performance as the chances of metastability in the mutual exclusion elements is maximized due to this near-synchronization of requests. In addition, when a number of functional units all require access to exactly the same register, it is undesirable that the same register should be read multiple times, for reasons of both performance and power consumption [8].

The method proposed here uses distributed requests coordinated by a central read controller, and avoids redundant reads as an inherent part of the mechanism by which a multiported register file is simulated. The register bank waits for all read requests to have arrived before commencing: this synchronisation incurs little penalty, since incoming requests are already nearly synchronised, but greatly simplifies the design of the hardware by avoiding the need for arbitration.

The read mechanism is shown in more detail in Figure 7. The system consists of the register sub-banks, which are connected to the read ports by a switching network. The switching network allows any read port to pass a request for a register to any sub-bank, and for register data and the register selection address from each sub-bank to be passed back to any of the read ports. The read process occurs over one or more cycles, managed by the read controller. The read cycle begins with one of the contending requests from the read ports being selected as the winner at each register sub-bank. Reads are then performed, and the winning register selections and the associated register data are passed back to the read ports. The controller then signals for the read ports to evaluate whether or not their request has been the winner: if so, the read port captures the data and removes its request. If there are any requests outstanding at the end of the cycle, the controller begins another read cycle after a delay to allow changes to the request signals to propagate across the switching network.

In practice, read requests arrive in pairs from each functional unit, so there is one control circuit for every two ports. However, for simplicity only a single port is shown in the figure. Requests arriving from the read ports are synchronized by the lock unit, which then signals the read controller to begin the read operations.

Data being written into the register bank as the result of a load from main memory may arrive at any time. This implies a possible hazard, where a load is initiated and a subsequent instruction attempts to access the data before it has arrived from memory. It is therefore necessary to enforce locking of registers which are the target of load instructions, to ensure that this does not occur. Before reaching the read ports, each active read request is compared against any currently active register locks. If a conflict exists, the read request is stalled until the lock is removed when the loaded data is written to the register. If no conflict exists, the read request is passed on to the read port.

6.2. Read operation

For reasons of synchronization, a request must be sent to each read port regardless of whether a read is actually required or not: a read enable signal is bundled with the request / acknowledge interface to the read port. When a read request, whether enabled or not, arrives at a read port, the port asserts the *go* signal to the lock unit.

While the *go* signals are being passed to the lock unit, each enabled read port passes its choice of register (5 bits) and a read request signal to the relevant sub-bank. At each register sub-bank, a simple priority selector chooses one of the active requests according to some arbitrary ordering, and passes the associated register selection to the sub-bank. The ordering chosen could be exploited by the designer, by connecting slower processing elements to the ports with higher priority: the slowest functional unit is guaranteed to begin operation in the earliest possible read cycle, with subsequent read cycles occurring concurrently with this slow operation.

Once *go* signals have been issued by all of the read ports, new register locking information and details of loads and stores are accepted from the load / store unit: this is the point where synchronization occurs. The new register locking information does not affect the state of any of the currently pending reads, allowing reads from a register and loads writing to that register to take place in the same parallel instruction (read-before-write ordering is enforced by the lock unit). Once the load / store information is latched, the *req_go* signal is asserted to the read controller to begin the first read cycle.

The read controller is responsible for coordinating requests from the read ports and performing read cycles as long as any read requests are outstanding. Each read cycle begins by sending the *req_read* signal to all of the sub-bank inputs. All of the sub-bank input selectors with at least one active read request perform read operations on their sub-banks, and respond on *ack_read*. Sub-banks with no active read requests remain idle, responding immediately with *ack_read*. Along with the output data, the register selection address of the winning request is also passed back across the switching network to the requesting read ports allowing

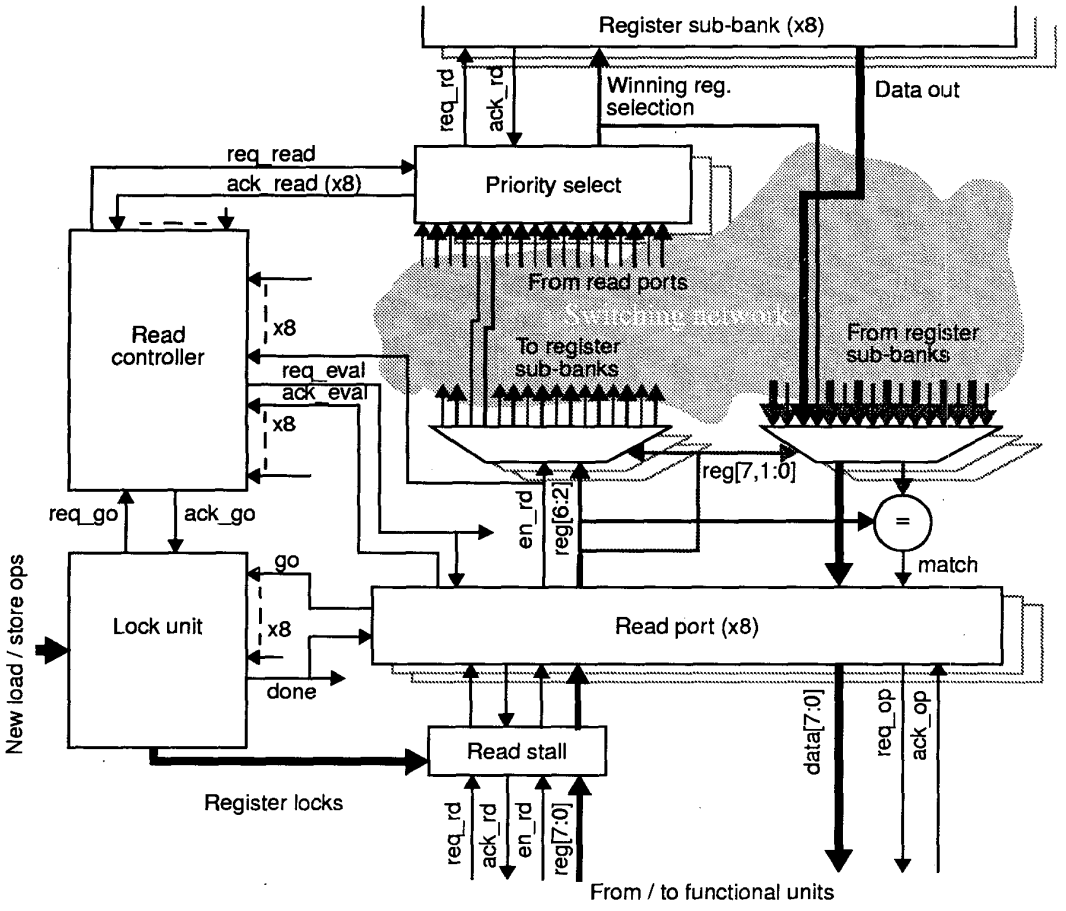

Figure 7 Read mechanism

them to determine when their register request has been satisfied.

Once each sub-bank read has completed, the read controller asserts *req_eval* to all of the read ports, to indicate that the output data from the register sub-banks is valid. Each read port has compared the winning register selection with its desired register in parallel with the register read process, so any read port whose request has been satisfied can capture the data immediately and remove its read request. This means that, if multiple read ports are requesting the same register, all of the read ports will have their requests satisfied by a single read cycle. Each read port responds with *ack_eval* once the capture / non-capture of data is complete and the read cycle is completed once all read ports have responded with *ack_eval*. As soon as the data has been captured by each port, it is passed to the functional unit which requested it using *req_op / ack_op*.

After the cycle has completed, another cycle is begun by the read controller if any read requests are still outstanding. Matched delays in the control path ensure that changes to the read-requests have time to pass across the switching network before the next read cycle begins. Once the final cycle is performed, with all read requests satisfied, the read controller finishes the read process by responding with *ack_go*. The lock unit, in turn, completes the handshake process with the read ports. The read ports only complete their input handshake cycle once both the read process has completed, and the functional units have accepted the new data: this means that, while data will be passed forward from the register bank to the functional units as soon as it is available, new read requests will only be accepted at the input of the register bank once the whole read process has completed. Free access without this synchronization would require arbitration for access to the read controller: as mentioned previously, the near-synchronization of the read

requests makes this undesirable. Also, the synchronization steps greatly simplify the locking mechanism.

7. Implementation

The CADRE DSP has been designed at the schematic level, using a mixture of standard cells and full-custom layout cells on a 0.35µm 3 metal layer process. The register bank consists of 89,835 transistors, out of which 50,736 transistors are used in the register sub-banks themselves. Of the remaining 39,100 transistors, approximately 11,000 are used in the control circuits with the remainder performing switching functions.

Control circuits for the DSP were specified using signal transition graphs, and synthesised into speed-independent circuits using the *Petrify* tool [9].

8. Simulation and Testing

All testing was performed by simulation of the processor architecture of Figure 1. The *Powermill* circuit simulator, which claims SPICE-like simulation accuracy, was used to perform the analyses. Memory models were used to simulate the system program and data memories. These modelled the power consumption of the memory units, using the power consumption figures from the RAM blocks of the AMULET3i embedded processor system [10]. Since layout of the CADRE processor has not begun, parasitic wiring capacitances were not available for inclusion in the simulations.

Timing information for reads and writes to the register bank were collected by using the functional modelling interface supported by *Powermill*. C models were written to record the time required to perform a writeback to the register file, and to record the time required to perform reads. The writeback time was measured as the time taken from the start to the finish of the write request handshake at each of the write ports. The read time was measured at each active read port, as the time taken from the assertion of the *go* signal to the completion of all the read requests at that port.

8.1. Read and write timing

The first test performed simply measured the dependence on the read and write process times on the number of conflicting elements. First, a succession of reads were performed on a single sub-bank; with the number of conflicting requests at the sub-bank increasing from one to nine. Second, a succession of writes were performed on a sub-bank, with the number of conflicting write requests increasing from one to four. The random nature of the arbitration for the write process means that there some variability is to be expected in the write times when a deliberate conflict is being introduced. In theory, the time to resolve metastability in the arbiters is unbounded. To give a reasonable assessment of the practical performance, each number of requests was performed using a variety of different write port configurations (to use different paths through the arbiter tree), and were repeated to perform a total of 100 attempts for each case.

8.2. Testing with DSP algorithms

To evaluate the effectiveness of the register file partitioning and the analysis of data access patterns, and the power impact of the register bank on the whole system, extensive testing was performed using real DSP algorithms. The chosen algorithms were a 20 point FIR filter, a 64-point complex FFT, and the LPC (linear predictive coding) analysis section from a GSM speech compression algorithm. The FIR filter and FFT were performed using both random data and speech data taken from the European Telecommunications Standards Institute's standard speech sequence for speech codec testing, with 256 samples being processed in each case. The LPC analysis program was performed on speech data alone, on a GSM speech frame of 160 data samples.

9. Simulation Results

9.1. Read timing

The maximum read times for each level of conflict are shown in Table 2. The results demonstrate that the first read cycle takes place quickly, within 5ns. Subsequent read cycles are slower, taking between 7-8ns to complete. This is because the *req_eval / ack_eval* cycle must be completed before another read cycle can be started, while the data from the first read cycle can be captured as soon as the *req_eval* signal has been issued. The figures presented are for the time taken to perform the last read cycle: other requests will be serviced in earlier read cycles, and will take proportionately less time.

9.2. Write timing

The measured worst case write cycle times for each level of conflict are shown in the right-hand column of Table 2. It can be seen that the time per write does not increase in proportion to the number of writes, since the incremental increase reduces somewhat. This is due to other requests propagating further through the arbiter tree while the first write requests are serviced, reducing subsequent write times.

Number of requests per bank	Read cycle time	Slowest write access time
1	5ns	10ns
2	12ns	18ns
3	19ns	26ns
4	26ns	32ns
5	34ns	
6	41ns	
7	48ns	
8	55ns	

Table 2. Read and write times with different levels of contention

9.3. Performance for DSP algorithms

The average, minimum and maximum read and write cycle times for the different DSP algorithms are shown in Table 3. It can be seen that, in all cases, the average read time is close to the minimum read time which illustrates the efficient performance of this asynchronous system.

The FFT has the worst read performance, as it is difficult to schedule all of the operations so that they do not conflict due to the bit-reversed addressing. However, the average case performance is still less than twice the minimum case, and is substantially less than the target cycle time of 25ns..

Algorithm	Read times			Write times		
	Min	Max	Avg	Min	Max	Avg
FIR filter	5ns	35ns	7ns	9ns	16ns	10ns
FFT	5ns	42ns	9ns	9ns	24ns	10ns
LPC analysis	5ns	12ns	5ns	9ns	11ns	9ns

Table 3. Register access times for DSP algorithms

The FIR filter algorithm could be expected to always have good performance, since it can be designed so that no conflicts occur. However, when the buffer size is not an even multiple of 4 (as is the case here, due to the way in which the parallelism is implemented) there are boundary cases where the sequential ordering breaks down. This, combined with additional delays due to store operations, leads to the higher maximum read time.

The GSM LPC analysis code demonstrates the best average and maximum read time. The code has, at worst, two read cycles required when implementing the autocorrelation portion of the algorithm.

In all cases, the average write time is very close or identical to the minimum value. The FFT and the FIR filter algorithms suffer similar difficulties in their write accesses as they do for their read accesses. By contrast, the LPC analysis algorithm never experiences write contention: the higher maximum write time is solely due to the worst-case delay through the writeback arbiter tree.

9.4. Power consumption results

Energy per operation for the whole DSP system running the test algorithms are given in Table 4. This gives the total system energy including the memory models, the energy dissipated in the whole register bank, that dissipated in the register subbanks themselves, and the number of accesses to the register bank and the data memories (the data memory consumes 0.67nJ per access). The simulations do not take into account capacitances due to interconnections, with the overhead of the switching network between the ports and the sub-banks representing the greatest load. However, for each operation only one path is driven from each port to a single sub-bank, and normally-closed operation of latches are used to avoid unwanted transitions from propagating across the network and out through the read ports.

9.4.1. Effect of split register architecture

It can be seen from Table 4 that, averaged over the different runs, the register bank consumes less than 9% of the total energy. The register bank uses decreasing amounts of energy per access for the FFT, FIR filter and LPC analysis tests respectively: this corresponds to how efficiently the algorithms make use of the register sub-bank interleaving. Accesses to the register bank require approximately one third of the energy required for an access to the main memory, and the register bank is accessed between 6 and 21 times more frequently than the memory. Even were the inclusion of wiring capacitances to increase the register bank energy consumption disproportionately, there is still a clear energy benefit from use of the register bank. A direct comparison is difficult however: the energy consumption figures for the main memory are based on those for an 8 kilobyte single-ported RAM. To service the functional units would require higher speed and / or multiple ports, both of which would dramatically increase the energy consumed by the memory system.

If it is assumed that power consumption of register banks increases in proportion to the square of the number

Algorithm	Total energy	Register bank						Data memory accesses
		Energy per instr.		Accesses	Accesses per instr.	Energy per access		
		Total	Subbank			Total	Subbank	
FIR filter (random)	4.15nJ	0.34nJ	0.12nJ	11620	1.9	0.18nJ	63pJ	556
FIR filter (speech)	3.59nJ	0.32nJ	0.11nJ	11620	1.9	0.17nJ	58pJ	556
FFT (random)	4.81nJ	0.5nJ	0.16nJ	8032	1.8	0.28nJ	89pJ	1096
FFT (speech)	4.84nJ	0.51nJ	0.15nJ	8032	1.8	0.28nJ	83pJ	1096
LPC analysis	3.47nJ	0.15nJ	0.04nJ	1004	0.7	0.21nJ	57pJ	180
averages	*4.2nJ*	*0.36nJ*	*0.12nJ*	-	-	*0.22nJ*	*70pJ*	-

Table 4. Energy per parallel instruction and per register bank access

of ports as suggested in [5], then the average power for a conventional multiported implementation could be greater by a factor of 64 than the interleaved scheme presented here: the register sub-banks have only 2 ports, while a unified implementation would require 16 ports. This gives an indication of how much benefit can be obtained from using the proposed architecture rather than a direct multiported register bank.

The actual benefit will be less than the factor of 64 implies (although still significant) as the quadratic assumption can be considered an 'upper limit' and the figures take no account of the wiring capacitance of the switching networks for reads and writes. A more conservative estimate can be made by extrapolating the data available. In a direct multiported implementation, each bit line for each read port would be connected to 256 register cells, as opposed to 32 in the current implementation. Assuming that the energy consumed by the register subbanks is dominated by the capacitance on the bit lines, it would be expected that each read to a direct implementation would require 8 times as much energy as each read to the register subbanks (70pJ) when neglecting wiring capacitances. This would lead to an average read energy of 0.56nJ, 2.5 times greater than that of the entire partitioned implementation. This estimate for the direct implementation neglects the cost of the register locking mechanism, which would further increase the energy per read.

Similar estimates may be applied to the read access times: each register cell must discharge 8 times as much capacitance on the bit lines. The discharge time for the 32 entry register subbanks is 0.61ns, out of a total access time of 1.71ns. Assuming that the access time scales only with the time required to discharge the word lines gives an estimated access time of 6.0ns, again neglecting the overhead of the locking mechanism. These results lead to an energy-delay product for the direct implementation of 3.3ns.nJ, as compared to an average energy-delay product for the partitioned implemenation of 1.5ns.nJ.

The above comments assess only the direct effect of the register bank on memory activity, and the effect of the register bank structure: use of a register bank also has collateral effects such as the ability to address data through 7-bit index registers rather than 24-bit address registers. Other figures collected in the same simulations show that an update to the index registers requires on average 0.15nJ while an update to the address registers requires 3.4nJ; and there are on average 14 index registers updates to every address register update.

10. Conclusions

An architecture for an asynchronous register bank has been presented. This allows the appearance of a large highly-ported register file to be presented to the programmer, while maintaining the power and speed advantages of using small single-ported register files.

The design takes advantage of the ability of asynchronous systems to exploit average-case operation times. Specifically, the data access patterns for DSP are suited to an interleaved address division of the register banks, but asynchronous operation allows deviations from these patterns to be accepted with only a modest decrease in average operation speed for typical algorithms. Where different data access patterns exist, similar techniques could be used with different partitioning schemes.

The proposed technique could also be applied to synchronous systems, but this would require either that the programmer guarantees only a limited amount of conflict (which is in practice extremely difficult) or that complex control logic be included to stall the pipeline when non-ideal access patterns occurred. In addition, the access times would then be coarsely quantized into an integral number of clock cycles; reducing the average-case performance.

Power consumption measurements made of the register bank within the CADRE architecture while executing real DSP algorithms indicate that the register bank consumes only a small proportion of the total power, giving a significant advantage in terms of energy per access over the main memory despite the fact that this memory is a fairly small single-ported RAM: it would be expected that a multiported RAM would consume very much more power and area. Estimates made for a direct multiported register implementation suggest that it would have an average energy per access around 2.5 times greater, with more than twice the average energy-delay product of the partitioned design.

Overall, the proposed register architecture gives the programmer an extremely simple and flexible programming environment, while maintaining fast access times on average, and minimising power consumption. The CADRE project is now proceeding to layout, transferred onto a smaller scale process technology, and this will allow the benefits of the register architecture to be assessed in greater depth.

11. References

[1] A.P. Chandrakasan, R.W. Brodersen, "Minimizing Power Consumption in Digital CMOS Circuits", *Proc. IEEE* vol. 83 no. 4, April 1995

[2] M. Lewis, L. Brackenbury, "CADRE: A Low-Power, Low-EMI DSP Architecture for Digital Mobile Phones", *VLSI Design*, Gordon and Breach Science Publishers, in press

[3] J.P. Diguet, S. Wuytack, F. Catthoor, H. De Man, "Formalized Methodology for Data Reuse Exploration in Hierarchical Memory Mappings", *IEEE Transactions on VLSI Systems*, Vol. 6 No. 4, December 1998, pp. 529-537

[4] H. Kojima, D. Gorny, K. Nitta, K. Sasaki, "Power analysis of a programmable DSP for architecture / program optimization", in Tech. Dig. IEEE Symp. Low Power Electron., pp. 26-27, Oct. 1995

[5] V. Zyuban, P. Kogge, "The Energy Complexity of Register Files", *Proc. International Symposium on Low-Power Electronics and Design*, 1998, pp. 305-310

[6] V. Zyuban, P. Kogge, "Split Register File Architectures for Inherently Lower power Microprocessors", Power-Driven Microarchitecture Workshop, in conjunction with ISCA'98, 1998, pp. 32-37

[7] M.B. Josephs, J.T. Yantchev, "CMOS Design of the Tree Arbiter Element", *IEEE Transactions on VLSI Systems*, Vol. 4, No. 4, Dec. 1996, pp. 472-476

[8] U.S. Patent No. 5,657,291, issued Aug. 12, 1997 to A. Podlesny, G. Kristovsky, A. Malshin, "Multiport Register File Memory Cell Configuration for Read Operation

[9] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Petrify: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers", *IEICE Transactions on Information Systems*, vol. E80-D, no. 3, pp. 315-325, March 1997

[10] J.D. Garside et. al., "AMULET3i - An Asynchronous System-on-Chip", Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, April 2000, pp. 162-175