LOW POWER ASYNCHRONOUS DIGITAL SIGNAL PROCESSING

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy in the
Faculty of Science & Engineering

October 2000

Michael John George Lewis

Department of Computer Science

Contents

List of Figures

List of Tables

Abstract

Cellular phones represent a huge and rapidly growing market. A crucial part of the design of these phones is to minimise the power consumption of the electronic circuitry, as this to a large extent controls the size and longevity of the battery. One of the major sources of power consumption within the digital components of a mobile phone is the digital signal processor (DSP) which performs many of the complex operations required to transmit and receive compressed digital speech data over a noisy radio channel.

This thesis describes an asynchronous DSP architecture called CADRE (Configurable Asynchronous DSP for Reduced Energy), which has been designed to have minimal power consumption while meeting the performance requirements of next-generation cellular phones. Design for low power requires correct decisions to be made at all levels of the design process, from the algorithmic and architectural structure down to the device technology used to fabricate individual transistors.

CADRE exploits parallelism to maintain high throughput at reduced supply voltages, with 4 parallel multiply-accumulate functional units. Execution of instructions is controlled by configuration memories located within the functional units, reducing the power overhead of instruction fetch. A large register file supports the high data rate required by the functional units, while exploiting data access patterns to minimise power consumption. Sign-magnitude number representation for data is used to minimise switching activity throughout the system, and control overhead is minimised by exploiting the typical role of the DSP as an adjunct to a microprocessor in a mobile phone system.

The use of asynchronous design techniques eliminates redundant activity due to the clock signal, and gives automatic power-down when idle, with instantaneous restart. Furthermore, elimination of the clock signal greatly reduces electromagnetic interference.

Simulation results show the benefits obtained from the different architectural features, and demonstrate CADRE’s efficiency at executing complex DSP algorithms. Low-level optimisation will allow these benefits to be fully exploited, particularly when the design is scaled onto deep sub-micron process technologies.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

The Author

Mike Lewis obtained an M.Eng degree in Electronic and Information Engineering from Churchill College, Cambridge in 1997. His Masters thesis concerned the application of statistical signal processing techniques to the reconstruction of degraded audio signals, and this interest in signal processing was continued during the three years of research which led to this thesis, with the AMULET Group of the Department of Computer Science at the University of Manchester.

The author was responsible for virtually all aspects of the CADRE architecture, from the initial conception through to the implementation and testing. Full-custom layout cells from the AMULET3 processor were used where appropriate, a small number of which were laid out by the author.

Acknowledgements

I would particularly like to thank my supervisor Dr. Linda Brackenbury, who has given me invaluable support and guidance, and kept my work on course. I would also like to thank my advisor Professor Steve Furber, who has given me many useful comments and insights.

Thanks also to Peter Riocreux and Mike Cumpstey for their comments in the Powerpack meetings during which I hammered out the early structure of my work, and to Dr. Jim Garside who provided useful advice and answers to many of my technical questions. Special thanks go to Steve Temple, who maintained our Compass CAD tools through adversity and helped whenever I have had difficulties.

Thanks to everybody who has helped by proof-reading my thesis, and for the other innumerable favours which I have received. I cannot imagine a better group of people to work and socialise with than the members of the AMULET group: thank you all.

My heartfelt thanks to my partner Cia, for her love and support which have made some very difficult times bearable. And thanks to Ying, without whom we would never have met. Finally, thanks to my parents for their continuing support of all kinds.

The work presented in this thesis was funded by the EPSRC / MoD Powerpack project, grant number GR/L27930. The author is grateful for this support.

Chapter 1:   Introduction

Over the past twenty years, the mobile phone has emerged from its early role as toy for a few wealthy technophiles to establish its current position as a true mass communication medium. Sales of mobile phone handsets are vast and rapidly increasing, with the number of subscribers having increased from 11m in 1990 to 180m people in 1999 [1]. Part of this rapid growth can be attributed to the decrease in price of the handsets, to the point that mobile network operators are able to actually give away handsets, defraying the cost in the revenue gained from contract fees and call costs. The low unit price makes this market extremely competitive, with manufacturers vying with one another to find differentiating features that give their phones a competitive advantage over those of their rivals. However, one factor dominates when distinguishing between phones: the size and weight of the handset. This is largely controlled by the trade-off between battery size and battery lifetime, which itself is controlled by the power consumption of the circuitry within the handset. Licensing of radio bands for third-generation cellphones, supporting high bandwidth data transfer, have recently taken place with bids reaching unprecedented levels [2]. The high commercial stakes and the imminent arrival of new high performance technologies therefore make mobile phones a very important application for low power circuit design.

Modern cellphones are based on digital communication protocols, such as the European GSM protocol. These require extremely complex control and signal processing functions, with the phones performing filtering, error correction, speech compression / decompression, protocol management and, increasingly, additional functions such as voice recognition and multimedia capabilities. This processing load means that the digital components of the phone consume a significant proportion of the total power. The bulk of the remaining power is used for radio transmission. The required radio power is fixed by the distance to the base station and the required signal-to-noise ratio, and will decrease as the number of subscribers increases and cell sizes decrease to compensate. Also, mobile communication devices will increasingly be used as part of local wireless communication networks such as the Bluetooth wireless LAN protocol [3], where the transmitted power is very low. It is therefore clear that the key to reduced power consumption for both current and future generations of mobile phone must be found in the digital subsytems.

These digital subsystems are typically based on the combination of a microprocessor coupled by an on-chip bus to a digital signal processor core. The microprocessor is responsible for control and user-interface tasks, while the DSP handles the intensive numerical calculations.

An example of a current part for GSM systems is the GEM301 baseband processor produced by Mitel Semiconductor [4], which contains an ARM7 microprocessor coupled to an OAK DSP core. A study of the literature for this product revealed that within the digital subsystem, the DSP is responsible for approximately 65% of the total power consumption when engaged in a call using a half-rate[1] speech compression / decompression algorithm (*codec*).

It can be expected that this proportion of the total power consumption will increase in future generations of mobile phone chipsets as the complexity of coding algorithms increases. For this reason, it would appear that the most benefit can be gained by reducing the power consumed by the DSP core. This thesis deals with the role of the DSP in mobile communications, and how the design can be optimised for this important application.

1.1 Digital Signal Processing

A generic analogue signal processing circuit, as shown in Figure 1.1a, consists of one or more input signals being processed by a bank of analogue circuitry such as op-amps, capacitors, resistors and inductors to produce an output with the desired characteristics. Subject to a few conditions, such a system can be described in terms of its transfer function H(s) in the Laplace transform domain. The digital counterpart to this, in Figure 1.1b, simply converts the input signals to sampled digital form, processes them according to some algorithm, and converts the output of this algorithm back into analogue form. A

1. The GSM protocol defines transmission of speech data with two different levels of compression, or *rates*. Full-rate compression produces output data that occupies an entire transmission frame. Half-rate compression produces output such that two speech channels can fit into a single transmission frame.

digital system meeting similar conditions to its analogue counterpart can also be described by a transfer function H(z), this time in the Z-transform domain.

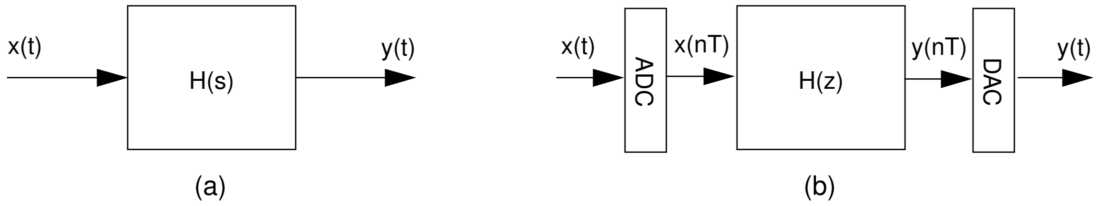


Figure 1.1 A traditional signal processing system, and its digital replacement

The fundamental mathematics describing both types of system have been known for nearly 200 years: Laplace [5] developed the transform that bears his name for describing linear systems, but according to Jaynes [6] he also developed a mathematics of finite difference equations that describes "...almost all of the mathematics that we find today in the theory of digital filters".

Although complete systems can perform very complex functions, the majority of signal processing operations can be broken down into combinations of the primitive mathematical operations shown in Table 1.1 [9].

FIR filter	$y(n) = \sum_{k=0}^{M-1} a_k x(n-k)$
IIR all-pole filter	$y(n) = \sum_{k=1}^{N-1} b_k y(n-k) + x(n)$
General filter	$y(n) = \sum_{k=1}^{N-1} b_k y(n-k) + \sum_{k=0}^{M-1} a_k x(n-k)$
Cross-correlation	$C_{xy}(m) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)y(n+m)$
Discrete Fourier transform	$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi k\frac{n}{N}}$
Autocorrelation	$C_{xx}(m) = \frac{1}{N}\sum_{m=0}^{N-1-m} x(n)x(n+m)$

Table 1.1: DSP primitive mathematical operations

1.1.1 Evolution of digital signal processors

The techniques of digital signal processing have been used to analyse scientific data since the advent of the mainframe computer, with the operations occurring off-line rather than in real time. However, the rapid development of integrated circuits has led to the practical application of digital signal processing techniques in real time systems. It is essentially the arrival of low-cost high performance digital signal processing that has enabled the mobile telecommunications revolution which we see around us.

The development of digital signal processors has largely tracked the development of general purpose microprocessors through improvements in device technology. However, DSPs have evolved a number of distinguishing architectural features. The fundamental DSP operations in Table 1.1 are all based around the summation of a series of products. The key operation within digital signal processors is therefore the multiply-accumulate (MAC) operation, and one of the main distinguishing features of a DSP as opposed to a general purpose processor is the dedication of a significant amount of area to a fast multiplier circuit in order to optimise this function [7], [9].

As early as 1984, real-time digital signal processing had established itself in a number of applications [7]. These included:

- Voice synthesis and recognition
- Radar
- Spectral analysis
- Industrial control systems
- Digital communications
- Image processing including computer axial tomography, ultrasound, lasers
- High speed modems and digital filters for improving telephony signal quality
- Audio reverb systems
- Psychoacoustics
- Robotic vision systems

The performance requirements for many of these applications could only be met at this time by costly custom circuits, with little or no flexibility. Few of these applications were intended for the mass consumer market, although a few notable exceptions existed such as the Texas Instruments *Speak'n'Spell* children's toy.

Possibly the first truly programmable DSP chip was the Intel 2920, "the first single microcomputer chip designed to implement real-time digital sampled data systems" [7]. This architecture very closely mirrored the generic signal processor of Figure 1.1b, with a multi-channel analogue-to-digital converter, a small scratchpad memory, an ALU and shifter to implement multiplication by a constant, and a multi-channel digital-to-analogue converter controlled by a program EPROM of 192 words [10]. However, the architecture had little flexibility, and the lack of a multiplier leads some to claim that it wasn't a 'real' DSP: in his after-dinner speech at DSP World in Orlando in 1999 [8], Jim Boddie (formerly of Bell Labs, currently executive director of the Lucent / Motorola StarCore development center) claimed this honour for the Bell Labs DSP1, which was released in 1979.

An early DSP chip with increased flexibility was the pioneering Texas Instruments TMS32010 DSP chip from 1982, whose architectural influences can be seen in many of the designs which followed [11]. This was a NMOS device, operating at a clock rate of 20MHz with a 16 bit data word length. Included in the architecture were a 16 by 16 bit multiply with a 32 bit accumulate in two clock cycles, separate data buses from instruction and data memory, a barrel shifter and a basic data address generator. It was also "the first (DSP) oriented chip to have an interrupt capability" [7], making it comparable in flexibility to the general purpose microprocessors of the time and suited to computationally intensive real-time control applications such as electric motor control and engine management units. However, this processor was somewhat restricted by an address bus shared between program and data memories, slow external memory accesses, limited addressing for external data and slow branch instructions [9]. Some of these restrictions were removed by its successor, the TMS32020, which had expanded internal memory, faster external memory accesses for repetitive sequences and more flexible address generations.

One of the early 'third generation' DSPs was the Analog Devices ADSP-2100 [12], which has most of the features common in subsequent devices. This had separate address buses to program and data memories, avoiding resource conflicts and allowing sustained single-cycle multiply accumulate operations at 12.5MHz. Sustained operation was supported with flexible data address generators, pipelining and a zero-overhead branch capability.

1.2 Architectural features of modern DSPs

The evolution of the architecture of modern DSPs has centred about the requirement to perform the multiply-accumulate operations for the various algorithms at the maximum possible rate. While a fast multiplier circuit is clearly necessary, this alone is not sufficient to guarantee high performance. The surrounding architecture must also be structured in such a way that the instructions and data for each operation can be supplied at a speed that does not limit the performance. This has led to a number of architectural features that are common to virtually all current DSPs, as shown in Figure 1.2.

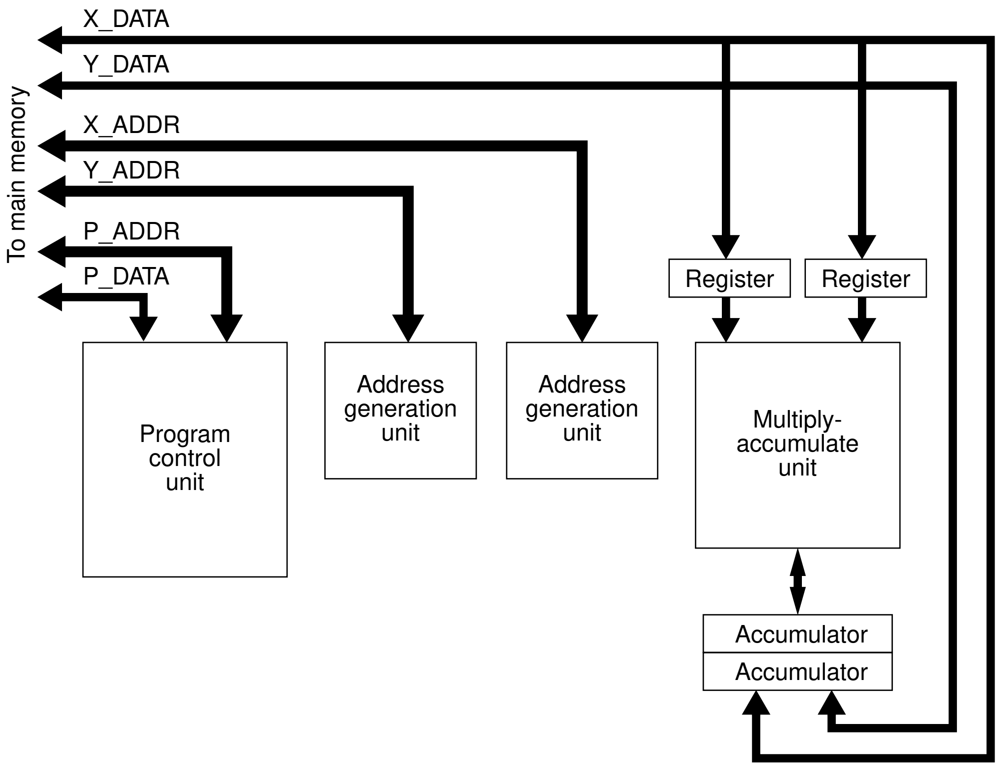


Figure 1.2 Traditional DSP architecture

1.2.1 High performance multiplier circuits

The multiplication of two binary numbers is essentially a succession of shift and conditional add operations, as illustrated in Figure 1.3a. Different multiplier implementations adopt different strategies in order to perform the required sequence of operations. In a general-purpose microprocessor, a multiplier may be implemented by means of an adder circuit and shifters, sequentially performing the series of shifts and adds with the product accumulated in a latch, as shown in Figure 1.3b. This is efficient in area but slow. DSP multipliers, therefore, trade an increase in area for faster multiplication by performing the additions simultaneously, in parallel. This gives the tree multiplier configuration of Figure 1.3c. A number of refinements to this configuration are possible, to speed the summation process and to reduce the number of summations which need to be performed. More details can be found later in this thesis, in the section “Arithmetic / logic datapath design” on page 184.

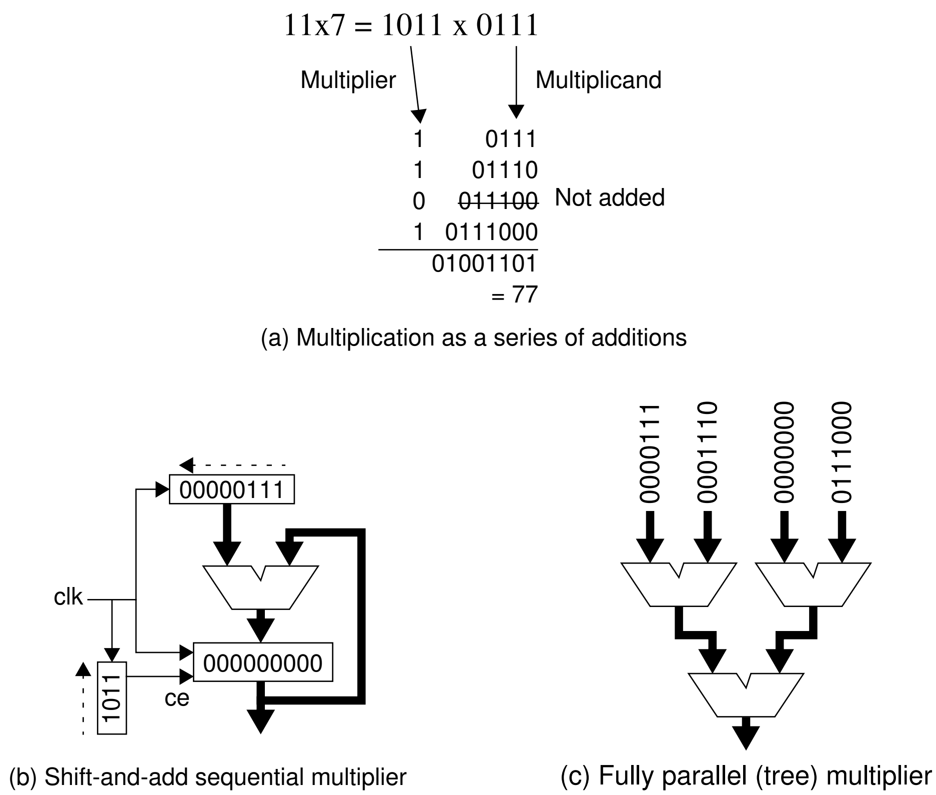


Figure 1.3 Multiplication of binary integers

1.2.2 Memory architecture

With considerable resources dedicated to high speed arithmetic circuits, it is important to keep them occupied as much as possible. This requires DSPs to maintain a high throughput of data between memory and the processor core. Conventional microprocessors have historically used the Von Neumann architecture, where programs and data are viewed as occupying the same contiguous memory space thereby allowing data to be freely interspersed within the program being executed. Program and data words are fetched from memory using the same bus, which leads to a potential bottleneck. To avoid this, digital signal processors are usually based around the Harvard architecture, where program and data memories are separated and accessed through independent buses. Merely separating program and data memories is generally insufficient, as many DSP algorithms require two new data operands per instruction, and so some form of modified Harvard architecture is chosen such as in the Motorola 56000 series DSP [14], which has three separate memories: P (program) and X/Y data memories. Many DSP algorithms map quite naturally onto this architecture, such as the FIR filter where data and filter coefficients reside in X and Y memories respectively. Usually, this separation of memories only applies to the on-chip memory around the processor core, with a larger unified store elsewhere. Viewed in this context, the separate memories act as independent instruction and data caches, although they are usually under the explicit control of the programmer.

1.2.3 Data address generation

Given pathways over which the data can be transferred, the other requirement to keep the arithmetic elements fully occupied is to be able to locate the data within the memories. A general-purpose microprocessor uses the same arithmetic circuits to perform both operations on data and calculations on memory pointers. However, this means that time is spent with the expensive multiplier circuits idle. To allow the maximum throughput to the multipliers, DSPs use separate address generator circuits to calculate the address sequences for data memory accesses in parallel with the multiply-accumulate operations. The data address generators provide support for the specific access patterns required in DSP algorithms; namely *circular buffering* and *bit-reversed addressing*.

Circular buffers are used in many algorithms where processing iterates over a fixed block of addresses. A buffer occupying `buffer_size` memory locations from `buffer_base` can be described in C as follows, with `addr` being the current pointer to the data and `offset` being the change in address.

```
addr = addr + offset;
if (addr > (buffer_base + buffer_size) )
{
        /* Gone past end of buffer */
        addr = addr - buffer_size;
}
else if (addr < buffer_base)
{
        /* Gone past start of buffer */
        addr = addr + buffer_size;
}
```

Having this type of construct implemented in hardware means that, for example, FIR filters can be performed without any interruption to the sequence of multiply-accumulate operations by setting up circular buffers for the data and filter coefficients.

Bit-reversed addressing is primarily required for fast Fourier transforms [9] [13] [15], where the rearrangement of the discrete Fourier transform equation, in Table 1.1 on page 16, requires the data to be accessed in bit-reversed sequence from the start (base) address as shown in Table 1.2. This can be performed either by physically reversing the order of the wires entering and leaving the address offset adder, or by reversing the direction of carry propagation.

Stage	Address fetched
0 (000)	Base + 0 (000)
1 (001)	Base + 8 (100)
2 (010)	Base + 4 (010)
3 (011)	Base + 6 (110)
4 (100)	Base + 2 (001)
5 (101)	Base + 5 (101)
6 (110)	Base + 3 (011)
7 (111)	Base + 7 (111)

Table 1.2: Bit-reversed addressing for 8-point FFT

1.2.4 Loop management

In many DSP algorithms, the majority of time is spent executing a fixed number of iterations of a loop. In a conventional microprocessor such a loop would be managed by decrementing a loop counter after each pass through the loop and performing a conditional branch, written in pseudo assembly language as follows:

```
move #count,d1
loop:
        {perform operation}
        sub #1,d1
        bnz loop
```

However, where pipelining is employed, such program structures cause branch hazards due to the dependency of an early stage of the pipeline (instruction fetch) on the result of a previous calculation. This necessitates either the pipeline to be stalled, which is simple but interrupts processing, or complex speculative execution to be implemented, where the branch direction is 'guessed' and incorrect instructions are flushed from the pipeline should the guess prove wrong. Also, the calculation of the branch target is a further overhead on each iteration unless a branch target buffer is employed.

Where loops with a fixed number of iterations are employed, it is possible to bring additional hardware to bear, taking the subtraction of the loop counter out of the processing pipeline and thereby eliminating the possibility of branch hazards. This leads to the following loop structure:

```
do #count,n
        {perform operation}
```

The 'do' instruction causes the start address and end address of the loop to be calculated and stored, and an internal loop counter to be loaded. When the program sequencer detects the end address, the start address is immediately loaded into the program counter without interrupting program flow. At the same time, the loop counter is updated in parallel with the execution of the instructions in the loop. Once the loop counter reaches zero, loop mode ends and execution proceeds normally. Many algorithms also require nested loops, which can be achieved through the use of a stack for the loop start address, loop end address and loop count.

1.2.5 Numerical precision, overflows and rounding

In a digital signal processing system, the precision with which signals can be stored, and therefore the maximum available signal to noise ratio of the processing system, is defined by the total number of bits with which data is represented in digital form. Two main forms of representation are used: floating point and fixed point. Floating point representation is the more flexible form, with data represented by a *mantissa* and an *exponent*. The number of bits allocated to the mantissa defines the precision, while the size of the exponent controls how large a dynamic range can be represented. The ability to represent a very wide dynamic range with constant precision makes programming of floating point systems very straightforward, reducing possible problems of over- and underflow.

The drawback with floating point representation is that the required arithmetic units are large, complex and power-hungry. For this reason, fixed point representation is preferred for low power systems. A fixed point representation is like a floating point number with no exponent bits. The precision is maximized, but the dynamic range is fixed to that which can be represented by the number of bits available. The fixed dynamic range causes problems when the magnitude of a result exceeds the maximum possible value (overflow), or the magnitude of a result is smaller than the minimum possible value (underflow). Overflow, underflow and the maintenance of the dynamic range of signals cause significant difficulties in the design of algorithms. However, a number of hardware elements commonly included in fixed point DSPs can ease the programming task.

One approach for reducing the effects of overflow is to implement *saturation arithmetic* in the processing elements. When a result exceeds the maximum possible positive or negative value, it is simply limited to that maximum value. This avoids the very large error that would be introduced by a conventional 2's complement binary overflow.

The result of a multiply or multiply-accumulate operation in a DSP goes to a high precision accumulator, which holds at least twice the number of bits as the values being multiplied. It is common for the accumulators to have some additional *guard bits*, which guarantee that a certain number of operations can be performed before overflow can occur. Rounding of the least significant portion of the accumulator reduces the error when converting back from the high precision accumulator representation to the lower precision representation (for example, when storing the result of a calculation in memory), and it is also common to implement the saturation arithmetic at this point, rather than when performing calculations, so that any possible loss of precision occurs as late as possible in the process.

Maintaining the signal to noise ratio in the processing, and avoiding underflow or overflow, requires that the input signal be scaled appropriately. This can be achieved most easily by the use of a shifter. Additional hardware to detect when data is approaching overflow or underflow can be used to implement automatic shifting of the data to maintain the precision, giving so-called ‘block’ floating point where an exponent is stored for a block of data at a time and updated at the end of processing.

1.3 Architecture of the GSM Mobile Phone System

While the next generation of mobile communications devices are very much on their way, the large investment in current GSM networks and the huge number of subscribers mean that the GSM system is likely to remain in use for some time to come. This section examines the requirements of current GSM systems, and the evolution towards third-generation mobile communications.

In the early 1980s, a variety of analogue cellular telephone systems were gaining popularity throughout Europe and the rest of the world, particularly in Scandinavia and the UK. Unfortunately, each country developed its own system meaning that users could only operate their mobile phone within a single country and manufacturers were limited in the economies of scale that they could apply to each type of equipment.

To overcome these difficulties, the Conference of European Posts and Telecommunications (CEPT) formed the Groupe Spécial Mobile (GSM) to develop a common public land mobile system for the whole of Europe. Some of the aims of the new system were to provide good subjective speech quality, to be compatible with data services and to offer good spectral efficiency, all done while keeping a low handset cost. In 1989, responsibility for the emerging standard was passed to the European Telecommunications Standards Institute (ETSI) and phase I of the GSM specifications was released in 1990.

In contrast to the established analogue cellular telephone systems of the time (AMPS in North America, TACS in the U.K.), GSM was a digital standard. A digital protocol gives flexible channel multiplexing, allowing a combination of frequency division multiplexing (FDMA), time division multiplexing (TDMA) and frequency hopping. Frequency hopping allows the effects of frequency-dependent fading to be reduced, while TDMA and FDMA provide high capacity when coupled with compression and error-correction coding of the speech data. A digital transmission channel allows data and image traffic to be carried without the need for a modem, and decouples channel noise from speech transcoding noise.

The overall network aspects of the GSM system (GSM layers 2 and 3), including such issues as subscriber identity, roaming, cell handover management etc., are extremely complex: the whole standard fills thousands of pages over many documents. A good introduction is given in [16], while an overview can be found in [17]. For the purposes of this thesis, the points of interest are the computationally intensive real-time tasks required at the mobile station relating to the speech transcoding [18] [19] [20], channel coding [21] and equalization [22] (GSM layer 1). A block diagram of the encoding and decoding processes is shown in Figure 1.4.

20ms of digitised speech data, sampled at 8kHz, is processed by the speech coder. This produces a compressed data block of 260 bits. Error correction coding is performed on this data, with a combination of block coding of certain bits followed by convolutional coding. The error control coding increases the size of the data to 456 bits. This data is then split into 8 subframes of 57 bits by the interleaver, and these subframes are grouped into 24 blocks of 114 bits per 120ms. A further two blocks of signalling data are added, to produce the TDMA traffic channel as shown in Figure 1.5. The fundamental transmission unit in the TDMA system is the burst period (BP). This contains 114 bits of data, 6 dummy tail bits, a further 8.25 bit guard period, 2 bits to indicate whether the data is being used for signalling purposes, and a training sequence in the middle of the burst period. The training sequence is used to allow an adaptive equaliser in the receiver to compensate for the channel characteristics under which the current block is transmitted.

8 of the burst periods grouped together makes up a TDMA frame, and each user is allocated one burst period in each frame (allowing up to 7 other users to simultaneously

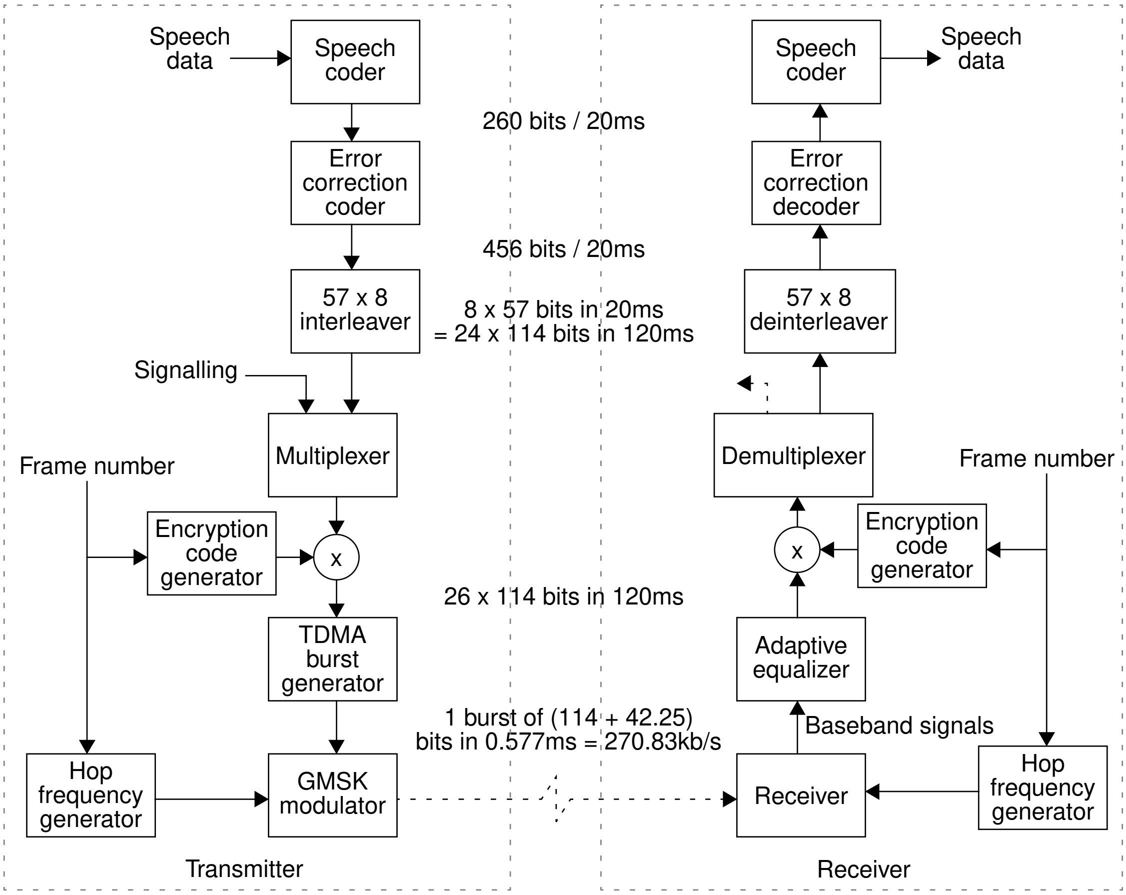


Figure 1.4 Simplified diagram of GSM transmitter and receiver

make use of that frequency). The TDMA transmissions take place over 124 200kHz-bandwidth channels spread over a 25MHz band. Different 25MHz bands are employed for the uplink from the mobile station to the base station and the downlink in the opposite direction, and the transmit and receive bursts are separated in time by 3 burst periods. This separation in both time and frequency eases the complexity requirements of the radio transceiver in the mobile station.

At the receiver, the RF signal is demodulated and the baseband in-phase and quadrature signals are sampled and processed by an adaptive filter. This filter is optimized for the channel conditions for each burst by making use of the training sequence in the middle of the burst period. The data subframes are then extracted, deinterleaved and decoded using a Viterbi decoder followed by a block decoder. Finally, the speech data is decoded, and converted back to an analogue audio signal.

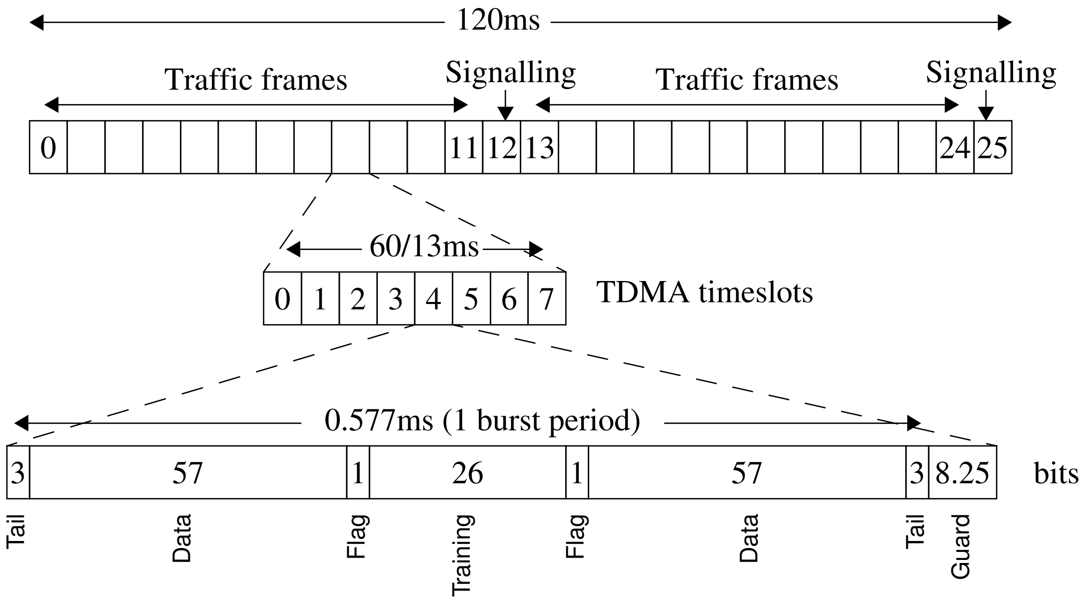


Figure 1.5 TDMA frame structure in GSM

When the original GSM specification was drawn up, it was envisaged that the majority of the processes would be carried out by ASIC components. However, it was generally accepted at the time that the speech transcoding was best performed by a programmable DSP and, once included in the system, other tasks such as equalisation and channel coding were assigned to give increased flexibility [23]. As the power of DSPs has increased, so the proportion of tasks allocated to it have grown. A typical division of the tasks within current baseband processors is shown in Figure 1.6. The main GSM layer 1 tasks in terms of DSP utilisation are channel equalization, channel coding (which is dominated by the Viterbi decoder), and speech coding. A brief description of these functions and the processing required by them now follows.

1.3.1 Channel equalization

The channel equalization is not specified by the GSM standard, allowing manufacturers to differentiate their products by the use of proprietary equalization schemes. The purpose of the equalizer is to compensate for inter-symbol interference, multipath fading and adjacent channel interference. The general form of a channel equalizer is shown in Figure

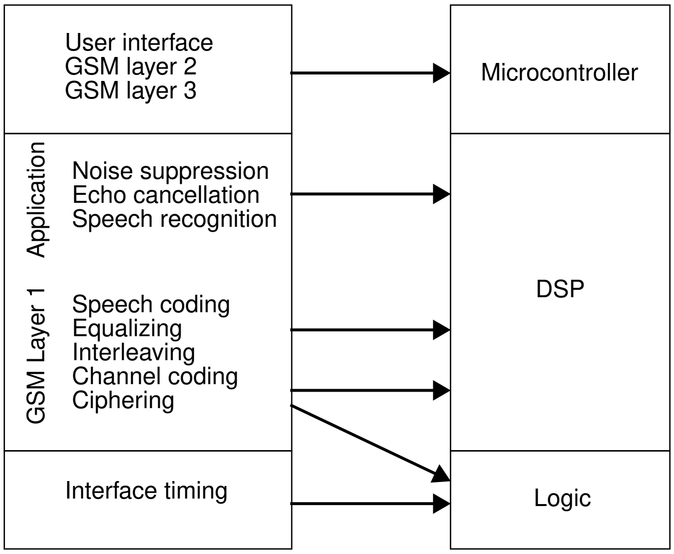


Figure 1.6 Division of tasks between DSP and microcontroller (after [23])

1.7. The training portion of the received burst period is used to adapt the filter parameters so as to minimise the error between the output of the filter and the known sequence. Once the filter has been optimised, it is used to process the entire burst. Commonly, a FIR filter is used as the processing element. A variety of techniques exist to optimise the filter parameters, such as the LMS algorithm or simpler variants using gradient descent of the error function [24]. A technique commonly employed in GSM systems is maximum-likelihood sequence estimation (so-called Viterbi equalization) [25]. In these systems, the channel impulse response is estimated from the training sequence. Given the received sequence, the most likely transmitted sequence can be estimated using a trellis search similar to the soft-decision Viterbi algorithm for error control coding. This is computationally expensive, but any hardware accelerators added to perform this function can also be used to perform Viterbi decoding for the channel coding part of the specification.

1.3.2 Error correction and Viterbi decoding

As mentioned previously, there are two levels of error control coding employed in the GSM system [21]: cyclic redundancy coding (block codes) followed by convolutional

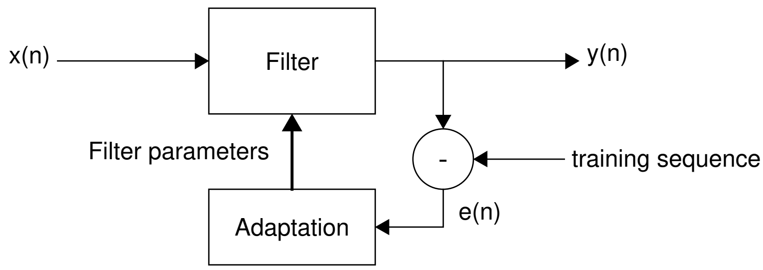


Figure 1.7 Adaptive channel equalization

coding. The type of coding used depends on the type of data being transmitted over the channel.

For speech channels, the data is split into two classes: class 1 bits are those that have been found to be subjectively most important to the resulting speech quality, with the remainder being class 2 bits. Class 1 bits have error coding performed on them, while class 2 bits are transmitted without error correction. Full and half rate speech channels use single level cyclic redundancy coding (CRC) to check for transmission errors, with the transmitted block being discarded if an error is detected. Enhanced full-rate speech channels use a two-level cyclic redundancy code. Control channels are protected with Fire coding, a special class of cyclic code designed to correct burst errors [26]. One of a number of different convolutional coding schemes are then applied, depending on the type of data to be transmitted.

Generation of both cyclic and convolution codes is readily achieved using simple shift register and XOR gate structures, such as the one shown in Figure 1.8. These functions can be performed by the DSP, but frequently it is more power-efficient to allocate these tasks to simple coprocessor circuits. Decoding of cyclic codes can be done using very similar shift-register based circuits such as the Meggitt error trapping decoder [26].

Decoding of convolutional codes is a very much more complex matter. The most common method for decoding convolutional codes is to use the Viterbi algorithm [27]. The encoder can be thought of as a simple state machine with 2^{k-1} states, where k is the constraint length of the code (5 in the example shown in Figure 1.8). Each input bit causes a state change, and a particular symbol to be transmitted.

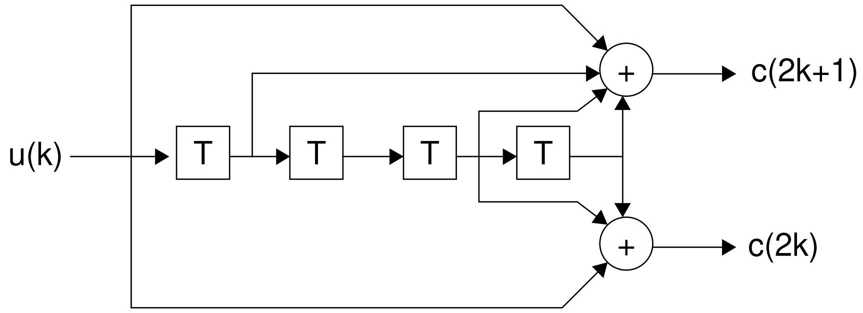


Figure 1.8 1/2 rate convolutional encoder for full-rate channels

The task for the decoder is to examine the received code symbols and determine which sequence of state changes (and therefore which sequence of transmitted symbols) occurred at the encoder. The Viterbi algorithm selects the path which gives an encoded sequence with the minimum Hamming distance (number of different bits) to the received value, and produces an output appropriately. The method used to decode a received sequence is to start in the initial state, and follow all possible paths from there, summing the total difference (*path metric*) between the received sequence and the theoretical transmitted sequence. Where two paths combine, the path with the lower total path metric is chosen as the survivor: this is where the difference lies between the Viterbi approach and the brute-force approach of checking all possible paths, and allows the processing complexity to be independent of the number of transmitted bits.

For each state, there are two possible paths leading to that state, and two leaving it. Therefore; for each symbol received, it is necessary to perform two additions to calculate the two path metrics leaving each node, to perform a comparison to determine the path with the lower error arriving at each node, and to select the path with lower error to be the new distance metric that will proceed forward from that node. For the constraint length 5 and 7 codes used in GSM full- and half-rate speech channels, the load corresponds to the evaluation of 32 and 128 path metrics per received symbol.

1.3.3 Speech transcoding

The speech compression algorithms used in the GSM system are classified as ‘analysis by synthesis’ (AbS) techniques [28]. A model of human speech generation is used, and the parameters of the model are chosen such that the synthesised speech resembles the original speech as closely as possible. It is then the parameters of this model that are transmitted, and these parameters are used to synthesise the speech signal at the receiver. AbS techniques form a compromise between high quality high bit-rate transmission techniques such as PCM at 64kbit/s, and low quality low bit-rate techniques such as vocoding which produce a very artificial sounding result at 2kbit/s and below. The particular form of model used in the GSM system is shown in Figure 1.9. This class of model uses linear predictive coding (LPC) to model the frequency response of the human vocal tract, driven by a long term prediction (LTP) filter which models the pitch component supplied by the vocal chords. The whole system is driven by a residual excitation signal, which is derived differently for the different classes of speech transcoding (full rate, enhanced full rate or half rate). Speech transcoding was the part of the original GSM specification that was considered most suited to DSP implementation: the following section describes the original full rate coder in some detail, and highlights the differences in the newer half rate and enhanced full rate schemes. The encoding is the most computationally intensive part of the transcoding process, as it involves estimation of the parameters of the various components of the AbS system. The decoder is given the relevant parameters, and is simply required to implement the speech synthesis system using those parameters.

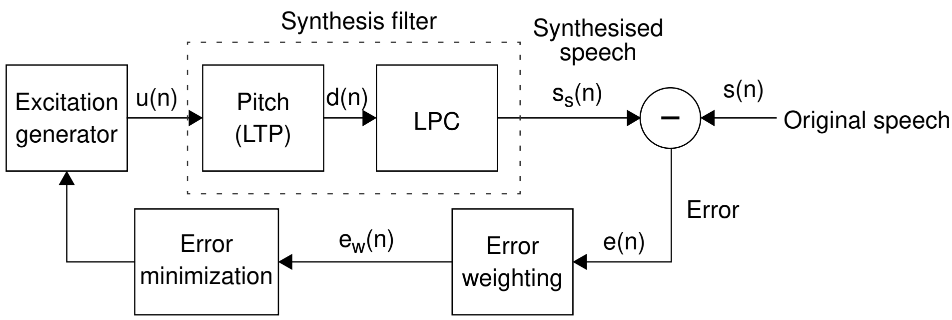


Figure 1.9 Analysis-by-synthesis model of speech

The full-rate GSM speech encoding process, as specified in [18], is described in some detail in appendix A. The encoding algorithm consists of a variety of different stages described in the appendix, and a summary of the approximate computational load of each stage of the full-rate GSM speech coder is shown in Table 1.3. It can be seen that the number of multiply and multiply accumulate operations far exceeds the number of simple additions or comparisons required. The processing load is dominated by the calculation of the parameters for the long-term prediction filter (LTP analysis), due to the large number of autocorrelations that need to be calculated to find the optimal lag value.

Processing stage	Multiplies / MACs	Additions / compares
Pre-processing	480	480
Autocorrelation	1249	-
Schür recursion	144	-
LAR quantization	8	32
Short-term analysis filtering	2560	48
LTP analysis	13144	172
RPE encoding	2033	244
Totals	19618	976

Table 1.3: Computation load of GSM full-rate speech coding sections

Half-rate and enhanced full-rate coding

Half-rate speech transcoding attempts to provide the same perceptual quality as the full-rate transcoding but with half the number of bits, as the name suggests. The encoding technique is vector-sum excited linear predictive coding (VSELP). This technique uses the same analysis-by-synthesis model of speech as used in the full-rate speech codec, but the excitation is generated by selecting an optimal sum of code vectors from a stored codebook, rather than using a simple set of regular pulses. VSELP is computationally more expensive than full-rate coding, and greater effort is made to optimize other parameters of the AbS model and to quantize the data efficiently, to compensate for the reduced amount of information that can be transmitted.

Enhanced full-rate speech transcoding aims to give significantly higher quality speech at the same bit-rate as full-rate transcoding. Algebraic code-excited linear predictive coding (CELP) is used, which is similar to VSELP except the code vectors are generated by a combination of a fixed codebook and an adaptive (alegebraic) codebook. A more complex

LPC model is used than for half-rate or standard full-rate transcoding, with 10 parameters updated twice per frame. Also, windowing is used to give smooth transitions from frame to frame. The overall computational complexity is claimed in [4] to be similar to that for half-rate speech transcoding.

The DSP operations underlying both these more advanced transcoding schemes and many other proposed codecs are fundamentally very similar to those required for the full-rate transcoder, as they are based on the analysis-by-synthesis model. Fundamental to all of them are the estimation of LPC parameters, the estimation of lag and gain for LTP parameters, and the development of an optimal excitation sequence by minimising the error between the synthesised result and the original speech. As for the full-rate transcoder, it can be expected that the calculation of autocorrelation values required at many stages throughout the encoding process will be the dominant processing load.

<table>
<tr><th colspan="2">Function</th><th>Load</th></tr>
<tr><td rowspan="4">Equalisation</td><td>Square distance calculation (20 MIPS)</td><td rowspan="4">42 MIPS</td></tr>
<tr><td>Add-Compare-Select (ACS) operation (10 MIPS)</td></tr>
<tr><td>Complex MAC for channel estimation and reference generation (9 MIPS)</td></tr>
<tr><td>others (3 MIPS)</td></tr>
<tr><td rowspan="2">Channel decoding</td><td>ACS operation (3 MIPS)</td><td rowspan="2">4 MIPS</td></tr>
<tr><td>others (1 MIPS)</td></tr>
<tr><td colspan="2">Voice coding</td><td>4 MIPS</td></tr>
<tr><td colspan="2">Voice decoding</td><td>2 MIPS</td></tr>
<tr><td colspan="2">Channel coding</td><td>0.1 MIPS</td></tr>
<tr><td colspan="2">others</td><td>0.9 MIPS</td></tr>
<tr><td colspan="2">Total</td><td>53 MIPS</td></tr>
</table>

Table 1.4: Required processing power, in MIPS, of GSM baseband functions

1.3.4 Summary of processing for GSM baseband functions

A summary of the processing requirements of the GSM baseband functions have been presented by Kim et al. [29], and are repeated in Table 1.4. The total processing load was estimated at 53 MIPS, and was dominated by the channel equalisation functions which required 42 MIPS. The conclusion reached by the authors of this paper was to include dedicated hardware for this function, similar to that incorporated in the GEM301 baseband processor.

1.3.5 Evolution towards 3rd generation systems

Current digital mobile phone architectures are considered to be the second generation of cellular systems (since FM analogue systems were the first generation to be used commercially). The basic elements of a third generation (3G) cellular system are as follows [30] [31]:

- Integrated high-quality audio, data and multimedia services
- High transmission speed incorporating circuit- and packet-switched services
- Support for variable and asymmetric data rates for receive and transmit
- Use of a common global frequency band
- Global roaming with a pocket-sized mobile terminal
- Use of advanced technologies to give high spectrum efficiency, quality and flexibility

A standard for third-generation services is being developed by the International Telecommunication Union, known as IMT-2000 (International Mobile Telephony) [32] [33]. The main proposals for this standard all use forms of code-division multiple access (CDMA) as the radio transmission technology. CDMA is a form of direct-sequence spread spectrum modulation, where the transmitted signal is modulated by a high speed pseudo-random code sequence. This causes the transmitted energy to be spread over a wide spectrum. At the receiver, the signal is correlated with the same code sequence which regenerates the original signal. All users transmit in the same frequency band, but use different pseudo-random codes; the correlation process picks only the desired signal out with the other signals appearing as low-level random interference. One of the main advantages of this type of modulation is that the effects of frequency-specific interference is reduced, as the desired signal is spread over a wide frequency band.

The correlation process in CDMA is a major processing demand: the chip rate (code sequence rate) is hundreds or thousands of times the symbol rate. Also, a number of separate correlators are required for the Rake channel equalisation system specified in IMT-2000. The correlators have their code sequences staggered by a chip period each, to attempt to gather as much of the energy lost by multipath (delay) effects. It is likely that this would be dealt with by a separate co-processor in a 3G implementation: given a flexible design, this coprocessor could br also used with a variety of CDMA protocols allowing, for example, an integrated cellphone and GPS receiver [23] [34].

The other component of 3G systems likely to demand dedicated hardware is the task of channel coding. While current DSP systems have sufficient processing power to perform the Viterbi decoding algorithms required by GSM systems, 3G systems will have symbol rates up to 100 times greater and dedicated hardware will be required to give the required performance with reasonable power consumption such as the bit-serial architecture proposed in [35]. To maintain low power beyond these bit rates requires even greater optimizations, such as the serial-unary arithmetic used in [36] where the metrics are represented by the number of elements stored in an asynchronous FIFO.

1.4 Digital signal processing in 3G systems

Even with many of the radio link aspects of 3G systems farmed out to coprocessors, the new types of traffic and the demand for new applications are likely to significantly increase the load on the programmable DSP [23]. Future generations of speech codec are likely to require many more MIPS to provide improved voice quality at the same or lower bit rates, and multimedia traffic such as streaming hi-fi audio and video will require large amounts of processing power operating alongside the speech codec. Ancillary applications such as voice recording, echo cancellation and noise suppression and speech recognition are finding their way into current GSM handsets, and are likely to be standard features in future generations of mobile terminal.

The high level of competition and demands for new applications emphasise the need for readily programmable and flexible low-power DSP architectures, to minimise the development cycle time and cost for new generations of products and to ease the period of transition before the next generation of standards are fully decided.

To a great extent, DSP manufacturers have relied on improvements in process technology to provide the required improvements in processing speed and power consumption: the basic structures of DSP architectures have remained relatively unchanged. However, increasingly deep sub-micron process technologies pose a new and different set of problems to the designer, and the optimum architecture is likely to be somewhat different to those that have gone before. This thesis presents an investigation into the design of a DSP architecture from the viewpoint of reducing power consumption in next-generation mobile phone handsets.

1.5 Structure of thesis

A wide variety of techniques for low power design are described in chapter 2 of this thesis, ranging from device technologies to architectural styles. A number of these techniques have been brought to bear in the design of the CADRE processor. The CADRE architecture and the techniques employed are described in chapter 3. The design process through which the architecture was implemented is described in chapter 4. In chapters 5 to 8, the implementation of various components of CADRE are discussed. In chapter 9, the CADRE architecture is evaluated and compared with a number of other DSP architectures. Finally, in chapter 10, a number of conclusions are made about the processor, and proposals for how the architecture can be improved are discussed.

1.6 Research contribution

The work presented in this thesis, as part of the POWERPACK low power design project, brings to bear a wide variety of low power design techniques to the problem of digital signal processing for mobile phone handsets. The result is a DSP architecture which differs significantly from those commercially available, and has features that are intended to reduce power consumption dramatically, particularly in deep sub-micron technologies. The following papers have been published presenting details of the DSP architecture.

M. Lewis, L.E.M. Brackenbury, “CADRE: A Low-Power, Low-EMI DSP Architecture for Digital Mobile Phones”, *VLSI Design* special issue on low-power architectures (in press).

M. Lewis, L.E.M. Brackenbury, “A low-power asynchronous DSP architecture for digital mobile phone chipsets”, *Proc. Postgraduate Research in Electronics, Photonics and* *related fields* (PREP 2000), April 2000, (Awarded Best Paper prize in the Signal Processing and Communications track)

This work also investigates the potential of asynchronous design for reducing power consumption, and includes a number of novel asynchronous circuits that exploit the characteristics of asynchronous designs (in particular, the inherent timing flexibility) to reduce power consumption and complexity. The following papers concerning aspects of asynchronous design for low power have been published.

M. Lewis, L.E.M. Brackenbury, “An Instruction Buffer for a Low-Power DSP”, *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 176-186, IEEE Computer Society Press

P.A. Riocreux, M.J.G. Lewis, L.E.M. Brackenbury, “Power reduction in self-timed circuits using early-open latch controllers”, *IEE Electronics Letters*, Vol. 36, January 2000, pp.115-116

M. Lewis, J.D. Garside, L.E.M. Brackenbury, “Reconfigurable Latch Controllers for Low Power Asynchronous Circuits”, *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems,* April 1999, pp. 27-35, IEEE Computer Society Press

Chapter 2: Design for low power

2.1 Sources of power consumption

In order to design circuits that consume as little power as possible, it is vital to understand the sources of power dissipation. In a CMOS circuit, power dissipation can be summarised by: [37]

$$P_{avg} = P_{switching} + P_{short} + P_{leakage} \qquad (1)$$

$$= fC_L V_{dd}^2 + I_{SC} V_{dd} + I_{leakage} V_{dd} \qquad (2)$$

The first two components are the dynamic power dissipation caused by switching activity at the various nodes within the circuits, while the third component is caused by static leakage. The following section examines these sources of power consumption in more detail.

2.1.1 Dynamic power dissipation

A generalised CMOS gate consists of a pull-up network made of PMOS transistors connected between the positive supply voltage and the output node, and a pull-down network made of NMOS transistors connected between the output node and the negative supply voltage. The simplest CMOS circuit is the inverter, as shown in Figure 2.1. Various capacitances exist, both within the circuit and also within the load connected to Z. For convenience of analysis, these are lumped together into a single capacitance C_L at the output. As the output charges to logic ‘one’ ($V_{in} = V_{dd}$), current flows into the load capacitance C_L, charging it to V_{dd}. During this process an energy of $C_L V_{dd}^2$ is drawn from the supply, with half of the energy stored in the capacitor and half of the energy dissipated in the resistance of the PMOS transistor. When the output returns to zero, the stored energy is dissipated in the resistance of the NMOS transistor. The average power drawn from the supply is therefore given by the energy $C_L V_{dd}^2$ times the frequency *f* of power-consuming (zero to one) transitions at the output Z.

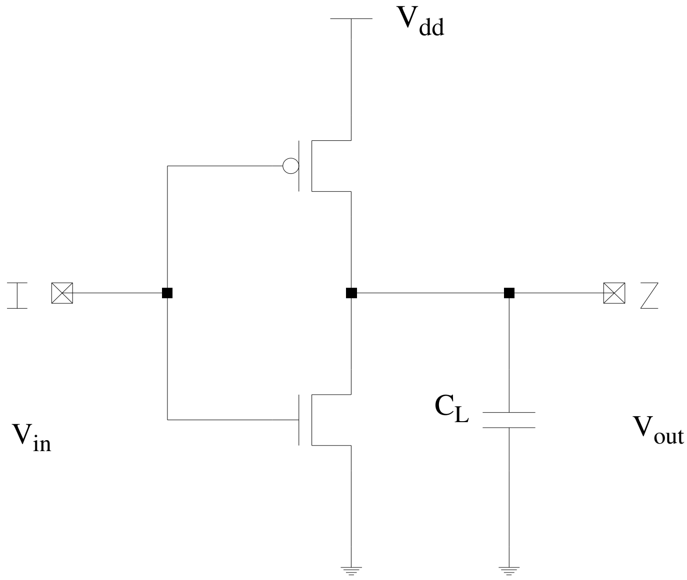


Figure 2.1 A simple CMOS inverter

This simple view of power consumption is based on the assumption that inputs change instantaneously, the switching times of the transistors are negligible and only one of the transistors is conducting at any time. However, in practice there is a brief moment during each switching transition when both transistors are conducting, allowing a short-circuit current to flow directly from V_{dd} to ground. This conducting period is defined by the input signal to the gates, and for a simple inverter is given by the condition $V_{tn} < V_{in} < V_{dd} - |V_{tp}|$, where V_{tn} and V_{tp} are the NMOS and PMOS transistor threshold voltages. This relationship implies that it is very important to minimise the transition times of input signals, so as to keep the time spent in the conducting region to a minimum. When this is done, short circuit currents generally make up less than 10% of the total switching power dissipation [38].

2.1.2 Leakage power dissipation

Leakage power is the component of power not caused by switching activity, and constitutes a fairly small proportion of the total power consumption of most chips at full activity. However, in systems where large amounts of time are spent in stand-by mode, it can have a significant effect on battery life. The leakage power dissipation comes from reverse-biased diode leakage currents, for example between transistor drains and the surrounding bulk, and from sub-threshold leakage currents in transistors which are biased off. Sub-threshold current decreases exponentially as the gate-source voltage is reduced below the threshold voltage, which can lead to potential problems in devices with low threshold voltages as the leakage current can remain quite high.

2.2 Power reduction techniques

The simple expression for power consumption given in (2) suggests three main ways of reducing the switching power dissipation:

- Reducing the supply voltage V_{dd}.
- Reducing the switched capacitance C_L.
- Reducing the rate of switching f.

2.2.1 Reducing the supply voltage

Of these techniques, reducing the supply voltage has the greatest effect due to the quadratic relationship between supply voltage and switching power consumption. However, this is done at the expense of operating speed. A simple approximate estimate of the effects on operating speed, based on the time taken to charge and discharge a node in the circuit, is given by

$$T_D \approx \frac{C_L V_{dd}}{I_{max}} = \frac{C_L V_{dd}}{k(W/L)(V_{dd} - V_t)^2} \tag{3}$$

where T_D is the switching time, I_{max} is the maximum switching current, k is a process-dependent parameter, W/L is the ratio of channel width to channel length of the transistor being switched and V_t is the threshold voltage of the switching device. It can be seen that the delay is approximately inversely proportional to V_{dd} for $V_{dd} \gg V_t$. However, as V_{dd} approaches V_t, the delay increases rapidly.

The simple first-order model fails to take into account the effects of carrier velocity saturation. With transistor feature sizes now significantly less than 1µm, the high electric field strengths cause charge carriers (holes or electrons) in the device to reach a limiting velocity [39]. For this reason, the current in the device is no longer quadratic in V_{dd} but is instead roughly proportional, and so the operating speed becomes roughly independent of the supply voltage

$$T_D \approx \frac{C_L V_{dd}}{W C_{ox}(V_{dd} - V_t) v_{max}} \qquad (4)$$

where C_{ox} is the process-dependent gate capacitance per unit area and v_{max} the carrier saturation velocity. One of the main arguments for the move from 5V to 3.3V operation for sub-micron integrated circuits was that this effect allowed power consumption to be reduced by 60% with little loss in operating speed [40], although reliability issues were clearly also a factor. However, the speed penalty when the supply voltage approaches the threshold voltage remains according to this approximation, albeit reduced somewhat.

Equation 4 suggests that it should be possible to maintain a given operating speed at reduced supply voltage by lowering the threshold voltages of the transistors in the circuit. However, this causes an increase in static leakage current and can reduce the noise margins of some logic structures. To some extent, reduced noise margins can be tolerated in low-power circuits as the magnitude of currents being switched is also reduced proportionately [40]. The increased leakage current is a more serious problem for a low-power design, and a compromise must therefore be made between the increase in leakage current and the reduction in switching power. An analysis of power consumption for deep sub-micron circuits with typical characteristics such as activity levels and wiring lengths suggests that the minimum power dissipation is reached at the point where the switching and leakage powers are approximately equal [41]. However, this is clearly not acceptable in a device intended for use in a mobile phone, where a large proportion of the time is spent in an idle state.

One solution to this problem is to place an additional transistor in either or both of the N- and P-stacks of the logic gates, and use these to prevent the leakage currents (although the extra series resistance reduces drive capabilities of the logic transistors somewhat if the loading on the inputs is kept the same). In Multi-Voltage CMOS (MVCMOS) [42], these additional transistors are driven by a 'sleep' signal, which lies outside of the normal supply voltage ranges. This means that P- transistors are driven at a voltage slightly greater than V_{dd} and N- transistors are driven to a voltage slightly lower than V_{ss}, ensuring that the devices are switched hard off. An alternative approach is to use reduced threshold voltage devices for the processing logic but to use higher threshold voltage devices to block leakage currents [43]. In this case it can also be possible to integrate the high threshold device into a non-critical portion of the logic function such as the precharge transistor in dynamic designs, thereby removing both the area and speed penalty of additional series transistors and avoiding the need for a separately derived ‘sleep’ signal. Another technique that does not depend on additional series transistors is the Dual-V_{dd} Dual-V_{th} (DVDV) approach [44], where a combination of different threshold voltages and supply voltages are used. Devices on the critical path use a higher supply voltage with a higher threshold voltage to give the most performance, while devices with slightly lower requirements use a lower supply voltage with reduced threshold voltage, and devices with the least performance use a lower supply voltage with the higher threshold voltage to minimise leakage power.

While these techniques offer great opportunities, there are some drawbacks. Generation of multiple supply voltages incurs a significant cost in a system, although DC-DC converters can be made with very high power efficiencies up to 95% [45]. Also, fabrication is complicated by the need to reproduce devices accurately with more than one threshold voltage, and in deep sub-micron devices threshold voltage is becoming increasingly difficult to control accurately [46]. The variability of threshold voltage therefore affects the choice of optimum threshold voltage and supply, as the variation can cause the power consumption and performance to be degraded from that predicted by theory.

Architecture-driven voltage scaling

While it is possible to reduce supply voltages and keep the loss in processing speed to a minimum by reducing threshold voltages, this incurs the penalty of increased leakage current as described above. Where power consumption is the prime concern and performance requirements are fixed it is possible to trade some loss of speed for a reduction in total power consumption. This is certainly the case for DSP applications in cellular phones, where the workload imposed by the particular protocol is fixed (although this will arguably become less true when the DSP is available for other applications).

The quadratic relationship between supply voltage and power consumption makes other trade-offs possible. When extra area is available (as is increasingly the case with ever-shrinking design rules) it can be possible to trade additional area for reduced power consumption using a technique known as *architecture-driven voltage scaling* [38] [40]. Given a processing load which can be met by a single processing element operated with supply voltage V_{ref}, the same processing load can be met by N processing elements at a supply voltage of V_N (if the task is such that it can be distributed in this way). Assuming that the supply voltage is low enough to avoid velocity saturation in the switching devices, and neglecting the effects of threshold voltages, V_N is given simply by

$$V_N = \frac{V_{ref}}{N} \quad (5)$$

The energy consumed by each processing element is therefore reduced by a factor N^2, but the number of processing elements has increased by factor N and so the total energy consumed per operation has reduced by a factor N. As the number of operations per second has also remained the same, the power consumption has also scaled down by this factor.

This simple analysis takes no account of the overhead in circuitry required to distribute the data to the processing elements and then recombine the results. When this is taken into account, and delays due to non-zero threshold voltages are also taken into account, the minimum power consumption occurs with 4 processing elements (for $V_t = 0.8V$) [40]. When leakage currents in deep sub-micron technologies are also taken into account, it is suggested that the optimal number will reduce [41]. However, this analysis is based on maximising the total performance, and may not be entirely applicable to low-power embedded systems. Even so, the optimal number of datapaths only reduces to 3 by the time 0.10µm technology is reached with the penalty for using 4 (or 2) being very small.

An alternative (possibly complementary) form of architecture-driven voltage scaling, with less area overhead, is to insert N pipeline registers at appropriate points in the circuit, to reduce the critical path by a factor of N. This allows a similar reduction in supply voltage, and hence power consumption, with very little additional area or power overhead and without a loss in total throughput. However, in this case the total latency of the circuit will increase by a factor of slightly greater than N.

Adaptive supply voltage scaling

Where processing demand is variable, it is possible to vary the supply voltage dynamically in order to meet the required throughput. This is known as Adaptive Supply Voltage Scaling or Just-In-Time Processing. Examples have been demonstrated of a DCC error correction circuit [47] where an incorrect code word requires three times as much processing as a correct code word. As 95% of code words are correct, this allows for a power saving of up to 80% by operating at reduced voltages during sequences of correct code words. Another application that has been demonstrated is a FIR filter bank for a hearing aid [48], where the supply voltage is reduced when processing low-level background noise. In both of these cases, asynchronous circuits were used for the processing. This power saving strategy would be rather more difficult to implement using synchronous circuits, as it would be necessary to reduce the clock speed to match the increase in circuit delay, although this has been implemented successfully in a number of cases; most notable being the Transmeta 'LongRun' technology [49].

Reducing the voltage swing

Instead of (or as well as) reducing the power supply voltage, it is possible to reduce power by limiting the voltage swing at nodes within the circuit. If the voltage swing is reduced to V_s, the total power consumption becomes

$$P_{switching} = fC_L V_s V_{dd} \quad (6)$$

A variety of techniques exist for use with differential dynamic logic to reduce the voltage swing experienced by the large NMOS pull-down trees and other circuit nodes [50] [51]. However, while differential dynamic logic offers very high performance, it is not necessarily a good solution for general use in low power systems as every gate produces at least two transitions (evaluate and precharge) thereby eliminating the possibility to exploit correlations in the data to reduce switching activity.

For general-purpose CMOS logic, the use of reduced voltage swing techniques is complicated by the need to restore voltages to full rail in order to prevent static short-circuit current in subsequent logic gates. The overhead of swing restoration means that the techniques are only applicable for situations where the node capacitance C_L is large enough to give a useful overall power reduction, such as when driving long on-chip buses. A variety of techniques exist which range in complexity, immunity to induced noise and available power reductions [52]. These vary from the very simple, where transistor threshold drops are used to reduce the voltage swing, to more complex designs requiring multiple supply voltages, low threshold voltage transistors, or differential signalling over two wires per bit. The possible energy reductions range from around 55% for the simplest techniques, to a factor of four to six for the most complex.

Adiabatic switching

In a conventional CMOS circuit, an energy of $C_L V_{dd}^2$ is drawn from the power supply each time the output load capacitance is charged. This energy can be reduced by ramping up or down the supply voltage as the capacitance is charged or discharged, minimising the potential across the resistance of the transistor, at the expense of switching speed: in the limiting case, no energy is dissipated but the capacitor takes forever to charge! The speed penalty can be offset to some extent as long as the voltage swing of the supply is greater than the required voltage swing at the output node [53]. This minimizing of the dissipated energy and recycling of charge is known as *adiabatic switching* [40], by analogy with thermodynamic systems which do not exchange heat with their environment.

The voltage ramps required to charge and discharge the load capacitances are produced by inductive pulsed power supplies: the waveforms generated are sinusoidal, which is a slight deviation from the ideal ramp profile but is easier to generate. The inductors are used to store energy from discharging nodes ready to be transferred into charging nodes.

Practical systems require multi-phase pulsed power supplies to minimise non-adiabatic losses due to non-ideal charging waveforms, such as the reversible energy recovery logic proposed in [54]. It is argued in [53] that this type of energy-recovery logic is best suited to deeply pipelined systems, to recover the energy used to drive clock lines.

However, these techniques appear to have limited application: in [53], some benefit was seen by using adiabatic driving of the clock signal, but a fully adiabatic design fared worse than conventional CMOS. Also, adiabatic techniques seem to only be suited to very low speed circuits: the comparison of carry look-ahead adders using adiabatic logic and standard CMOS in [54] found that the adiabatic techniques only gave an advantage at operating speeds of below 3MHz, and the clocked power supply generator consumed a large amount of power which was not considered in the comparison.

An approximation to adiabatic operation, without the need for the complex pulsed supply generators, can be obtained using multiple supply voltages switched across the load to approximate a ramped supply. However, it is hard to build sufficient capacitance into the supply rails to store the returned energy without adding costly external capacitors, and the additional transistors, the required drive circuitry and the multiple supply voltages add significant complexity. Both the pulsed-supply and multiple supply systems incur significant overheads and suffer from reduced maximum speeds. This makes the techniques only suited to areas where very high loads are being driven, such as pad drivers.

While adiabatic techniques are difficult to apply in practice, it is possible to obtain a limited amount of benefit from the recycling of charge relatively easily. One of the largest capacitative loads within an integrated circuit are internal buses. The activity on the bus can be exploited to reduce overall power consumption by shorting bus-lines which are to be discharged to bus-lines which are to be charged, thereby recycling the energy stored in the load capacitance somewhat [55]. The amount of charge that can be recovered depends on the switching profile of the data on the bus, but studies with real data have shown that average energy savings of 28% are possible [56].

2.2.2 Reducing switched capacitance

The second ‘physical’ parameter that the designer can alter to reduce switching power dissipation is the capacitance of circuit nodes. It should be noted that reducing switched capacitance and reducing supply voltages are complementary techniques: reducing node capacitances increases switching speed, which can compensate somewhat for reduced supply voltages. The lumped node capacitance C_L is actually made up of a number of different physical capacitances described in [40], as shown in Figure 2.2.

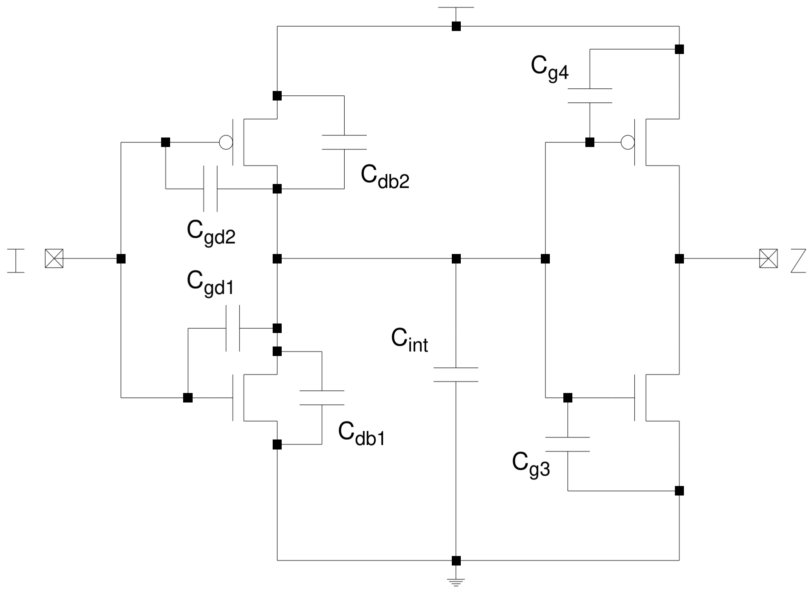


Figure 2.2 Components of node capacitance C_L

The lumped gate capacitance of a transistor (C_{g3} and C_{g4} in the figure) is composed of the gate-bulk capacitance C_{gb} and the gate-source and gate-drain capacitances C_{gs} and C_{gd}. The value of each of these depends non-linearly on the operating mode of the transistor, but the total remains close to $C_{ox}WL_{eff}$ (where C_{ox} is the capacitance per unit gate area, W is the gate width and L_{eff} is the effective gate length), and so this value is used when estimating load capacitance.

In a real transistor, the source and drain regions overlap the gate to a certain extent. These areas of overlap reduce the effective length of the gate, and cause gate-drain and gate-source capacitances. These capacitances contribute to C_{g3} and C_{g4}, and cause the gate-drain capacitances C_{gd1} and C_{gd2} in Figure 2.2. The effective gate to drain capacitances are increased due to the Miller effect: because the gate voltage is moving in the opposite direction to the drain voltage, the capacitors can be treated as having twice their value when referred to earth.

The sources of the transistors are connected to the supply rails, which are treated as AC grounds. However, a capacitance exists between the drain diffusion and the bulk, represented by C_{db1} and C_{db2}, formed across the reverse-biased PN junction between the drain and the lightly-doped bulk. The capacitance is therefore voltage dependent

(dependent on the junction depletion width). The capacitance is made up of two parts: the parallel plate capacitance between the bottom of the drain region and the bulk, and the sidewall capacitance (which is larger per unit area, due to the highly doped stopper implant around the edges).

The final component of capacitance, C_{int}, is the capacitance of the interconnections between the stages. This capacitance consists of capacitance between the wire and the substrate across the (thick) field oxide, capacitance between neighbouring wires on the same routing level, and capacitance between adjacent wires. The capacitance to the substrate is made up of a parallel plate component, which is proportional to the width and length of the wire, and a fringing component which is proportional only to the length. The capacitance to the substrate also depends on the routing level: wires at lower levels have a higher capacitance, as they are closer to the substrate.

The analysis of capacitance in the case of chained inverters over-simplifies matters somewhat. In more complex logic gates, further internal nodes exist within the pulldown and pullup networks. Only some of these nodes may be charged or discharged during evaluation of the logic function. This charging or discharging and the resultant power dissipation is dependent on the particular function of the gate and the combination of inputs.

Feature size scaling

Scaling of feature sizes reduces all dimensions of transistors by (approximately) the same factor S [39]. The gate capacitance is approximated by $C_{ox}WL_{eff}$: W and L_{eff} are both reduced by factor S, but the gate areal capacitance C_{ox} is inversely proportional to the gate oxide thickness t_{ox}, which is also scaled by the same factor. This causes an overall reduction in gate capacitance by approximately a factor of S. The drain- and source-to-bulk capacitances C_{db} and C_{sb} are independent of gate oxide thickness and should scale as $1/S^2$, although the sidewall component does not necessarily scale to the same extent.

Feature scaling is clearly very beneficial for reducing gate capacitance. However, the picture is less rosy when wiring capacitance is considered. In order to keep RC delays

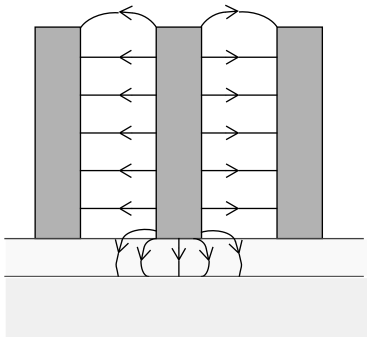


Figure 2.3 Wire capacitances in deep sub-micron technologies

along interconnections at a reasonable level, wires cannot scale in size to the same extent (even with low resistivity metals such as copper coming into use). To compensate for ever-greater packing density of gates, the distance between wires and their width is decreasing: to maintain low resistance, the wires must therefore be made taller as shown in Figure 2.3. This leads to increased capacitance between adjacent wires, leading to more crosstalk, while capacitance from the wire to the bulk semiconductor becomes dominated by fringing effects, and cannot be reduced by making the wires narrower. When coupled with the reductions in gate capacitance, the increased gate density and the increased interconnect density, it is clear that the interconnect capacitance will be an increasingly dominant component in the total node capacitance. This will also cause it to be a limiting factor in the total operating speed, particularly when transmission line (RC) effects are taken into consideration.

Transistor sizing

In order to achieve the highest possible speed, one must size the transistors in the logic gates appropriately. The simple case of driving a large load through a chain of inverters is well known, and a similar approach can be used on general logic structures by considering the drive capability of each gate, the amount of off-path loading and the load of each gate in the path. This technique has been generalised into the ‘theory of logical effort’ [57] for calculating the optimal topology and number of stages for a given logic function. However, where speed is not critical then circuits built with these techniques consume more power than is necessary; for example, in the case of an inverter chain then as much power can be dissipated in the inverters as goes into the load capacitance [58]. For circuits off the critical path, therefore, gates with smaller devices and greater ratios of input to output loads will consume less power than those designed for optimal performance. However, care must be taken that edge speeds do not become too slow in order to prevent excessive short-circuit switching current.

Layout optimization

Clearly, as wire capacitances come to dominate in deep sub-micron designs, the choice of circuit layout and routing will come to dominate both the power consumption and the performance of the design. While global communication pathways such as buses can be clearly identified and various approaches used to reduce their power dissipation such as reduced voltage swing signalling, the wiring required to implement local interconnections are still of great importance particularly as local interconnections are on the lower routing layers, and therefore have higher capacitances both to ground and to one another.

At the circuit level, structures with interconnections only to nearest-neighbours such as systolic arrays minimise the length and therefore the capacitance of the interconnections. A study of a number of different multiplier topologies [59] found that the average net length varied by a factor of almost six. For all circuits, the placement and routing of a circuit must be optimised, either manually or using a tool with suitable intelligence. One approach is to use hierarchical place and route to exploit structure in the design and to ease the task of the place and route tool. It was shown in [59] that hierarchical place and route could reduce the average net length by a factor of 3.6. However, the power consumption did not track the average net length as strongly. This is due to the switching characteristics of the various signals: those signals which change frequently are the most important to optimise.

SOI CMOS technology

A technology that shows much promise for low-power and high speed designs is silicon-on-insulator (SOI) CMOS technology. A SOI transistor is shown in cross-section in Figure 2.4. Instead of having the transistors formed by diffusion into the bulk silicon, the transistors are made in a thin layer of silicon separated from the substrate by a buried oxide insulating layer. This massively reduces the source / drain diffusion capacitances and the gate to bulk capacitance. Also, as the body of transistor is floating, SOI transistors suffer much less from the *body effect*, which causes transistor threshold voltage to change in transistor stacks and reduces their current drive capability. SOI transistors are also extremely well suited to use with low supply voltages as they have near-ideal sub-threshold leakage currents.

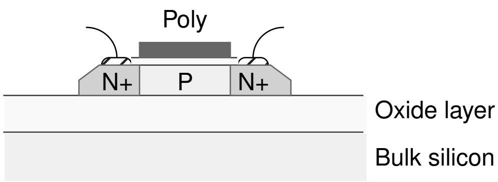


Figure 2.4 SOI CMOS transistor structure

While SOI has many benefits, there are a number of issues which the designer must be aware of, mostly caused by the floating body voltage [60] [61] [62]. SOI CMOS is also rather difficult to manufacture, due to the problems of generating a sufficiently high quality interface between the bulk silicon, the buried oxide layer and the active silicon. Despite these problems, a number of commercial high-performance microprocessor designs have been retargeted onto SOI successfully such that they passed commercial yield and reliability standards, with reportedly very little modification required to the circuits [63] [64].

2.2.3 Reducing switching activity

The third component of equation 2 that can be altered to reduce the overall power consumption is the rate of switching at each node in the circuit. The switching activity that takes place within a circuit can be divided into two components: activity which is required to calculate the desired result, and unnecessary activity that occurs as a by-product of other activity within the circuit. Clearly, to produce a given result there must be a certain minimum amount of activity within the circuit. However, this minimum is very hard to define, and in practice the amount of activity is dependent on a wide range of design decisions such as choice of circuit structures, system architecture and the required operating speed.

Reducing unwanted activity

Unwanted activity in a circuit can be broken down into two main components. The first component is caused when the inputs of a processing element such as an ALU go through a number of configurations before the correct data is presented. A typical case would be when one input arrives a little earlier than the other input: the ALU will wrongly calculate the result of the operation with one correct and one incorrect input, before proceeding to change to the correct result when both data values are present. These incorrect results can propagate from the output of the ALU to subsequent stages. Depending on the complexity of the processing logic within the ALU and the nature of the circuits downstream this can cause a large amount of energy to be wasted.

The second component of the unwanted switching activity comes from intermediate states generated at the outputs of logic gates when the inputs change. A typical example is a two-input NAND gate whose inputs change almost simultaneously from 1,0 to 0,1. According to the truth table of the circuit the output should remain at logic 1. However, depending on the exact relative timing of the input signals, a brief unwanted pulse may occur at the output. These *glitches* can vary in magnitude from a complete transition to a small partial swing before returning to the steady state. They can also propagate through the circuit and cause more unwanted activity in downstream stages. The impact of these glitches are hard to assess accurately, without electrical-level simulation, as they are critically dependent on the timing of the signals passing through the circuit. While it is possible to make a reasonably accurate assessment of glitch generation given an accurate timing model of a logic gate, it is a difficult problem to assess the effects of glitch propagation. The propagation of glitches is critically dependent on the electrical properties of gates downstream; so any error in simulation will tend to be magnified [65]. Statistical analysis of switching activity with glitching taken into consideration shows promise, with reported errors being around 6% [66]. However, this analysis relies on time being subdivided into discrete ‘timeslots’, based on the smallest gate delay in the cell library. The choice of timeslot appears to be dependent on the circuit under analysis, and this may prove to be the limiting feature of this approach.

A number of techniques exist to prevent the generation and propagation of unwanted transitions. Balancing the path lengths from different inputs to the output can reduce the number of intermediate states generated within a processing block [38]. Where the output of a processing stage is known to be unused, it is possible to gate the bus drivers at the output of that stage to prevent unnecessary switching activity propagating through the circuit [67].

Dynamic logic inherently prevents the propagation of unwanted switching activity, as it cannot pass signals when in the precharge phase, and does not generate glitching activity as each output can undergo at most one transition in the evaluate phase. However, the need to precharge each node increases the overall switching activity and eliminates the possibility of exploiting correlations between data, making dynamic logic unsuitable for low power designs in general; although the high speed, reduced node capacitance and elimination of short-circuit currents may make dynamic logic favourable in particular situations.

When the arrival of operands is skewed in time, opaque latches can be used to delay evaluation until both operands are valid, which also stops any glitches associated with the evaluation of the new operands. However, this adds additional delay and a small amount of additional area. When the circuit in question is off the critical path, there is a strong argument for using opaque latches. Otherwise, the increased delay may be unacceptable.

One application where the use of opaque latches can have a significant effect on power consumption is in asynchronous micropipelines [88], and a variety of ways in which latches can be operated to reduce power consumption are presented in section 2.3.2 on page 73.

Choice of number representation and signal encoding

One characteristic which distinguishes the data seen in signal processing applications from that of general purpose processing is that the signals under question undergo gradual changes, leading to correlations between successive data values [40]. This can have a large effect on the switching activity within a circuit. Depending on the amplitude of the signal, a certain number of the low-order bits will tend to be completely random with a transition probability of 0.5 at each bit position for each input, while the upper bit positions will have decreased switching probabilities, as depicted in [40].

The amount to which these correlations can be exploited depends on the number system chosen to represent the data. This decision impacts upon the power consumption of the system in a number of different ways. Firstly, the choice of number representation affects the complexity of the arithmetic elements required to maintain a certain level of throughput. Secondly, the type of number representation used has an influence on the switching activity both on buses and within processing blocks. Finally, the compactness of the encoding bears upon the amount of memory required for storage and hence the amount of power consumed in transferring data to and from memory. Only fixed-point number representations will be considered here, although many of the points considered could also be applied to the design of floating-point systems. However, these systems are generally more complex and hence will not be chosen for a low-power system when possible.

The number system most commonly employed for general purpose fixed-point digital signal processing is the *2s complement* numbering scheme. This representation has the form:

$$Z = -b_m 2^m + \sum_{i=0}^{m-1} b_i 2^i \qquad -2^m \le Z < 2^m - 1 \tag{7}$$

Its primary drawback is the large number of redundant ones required to represent small negative numbers. This means that, for a digitised signal with small fluctuations about zero, there will be a high switching activity in the sign extension bits. One way of reducing this effect was proposed by Nielsen and Sparsø [48] for use in a FIR filter bank for a hearing aid. Power was reduced by dividing the datapath into two eight-bit segments and enabling only the least-significant eight bits during periods of low input magnitude. This also reduced power consumption by allowing the multiplier and adder circuitry to be partially deactivated.

An alternative representation, that avoids the switching of the sign extension bits, is the *sign-magnitude* numbering scheme, where only a single bit is used to represent the sign. This representation has the form:

$$Z = (-1)^{b_m}\left(\sum_{i=0}^{m-1} b_i 2^i\right) \qquad -2^m + 1 \le Z < 2^m - 1 \tag{8}$$

The drawback of this numbering system is that, in order to add numbers with different signs, it is necessary to either convert both numbers to 2s complement format and use a conventional adder, or use a dedicated subtracter circuit. If the former option is chosen, the extra transitions generated will reduce the benefit of switching to sign-magnitude representation. The latter option has a penalty in area. Chandrakasan and Brodersen [38] concluded that sign-magnitude representation is best used in designs where a large capacitive load is being driven such as external memory buses, etc. In this case the power overhead of converting to and from 2s complement representation within minimum-geometry arithmetic circuits is negligible compared to the power saving from the reduced switching activity on the bus.

One of the main problems in digital arithmetic is the possible dependency of the highest order bits in the result on the lowest order bits, due to carry propagation. A class of number systems that can eliminate long carry chains, and which also do not require sign-extension bits, are redundant signed digit representations as proposed by Avizienis [68]. Redundant number systems are defined by the following equation:

$$Z = \sum_{i=0}^{N} z_i r^i \qquad -r + 1 \le z_i \le r - 1 \tag{9}$$

The restrictions on the values for z_i are based on the requirement that there be a uniquely defined zero value. For radices greater than two, it is possible to add two numbers together so that the output from a given pair of digits is dependent only on their values and the value of a transfer digit from the next lower order digit. Other redundant number systems such as carry-save or borrow-save can be shown to be special cases of this type of number system [69]. Another attractive aspect of these signed-digit representations is that, due to the different possible representations of any value, it is possible to choose the representation with the minimum Hamming weight (i.e. the representation with the greatest number of zero digits) [70]. In order to reduce the amount of redundancy, it is common to use a *modified signed-digit* representation which is of radix two and where each digit is taken from the set of values (-1,0,1). This allows more compact encoding, at the expense of an increase in the possible carry propagation under addition to two places.

Another method for avoiding carry propagation problems is to use a *residue number system* [71]. In these number systems, a number is represented by its remainder modulo for a number of different relatively prime bases. Addition is performed by simply taking the sum of each pair of remainders; with multiplication being a trivial extension to this. However, there is no way of directly determining the sign of a number in residue form, which means that comparison, and hence division, is difficult [72]. Also, it is difficult to convert to and from residue number systems [73]. This limits the usefulness of residue numbers for all but special cases, although they have been shown to be very effective when comparisons are not necessary and when residue number systems can be used throughout [74].

A number of techniques have been designed specifically for reducing the number of transitions required to transmit values across buses. One technique is to exploit correlations between successive number values, such as delta encoding where only the changes to a number are transmitted. However, this requires that an addition is performed for every data item received, which can remove any power benefit from the reduced bus activity. A simpler method of encoding is to use a transition signalling scheme, where a transition on a wire indicates a one and no transition indicates a zero. Encoding and decoding is done by a simple XOR between the data value and the last data value transmitted or received. Power can also be reduced by using lossless compression to reduce the amount of redundant data being transmitted, although this must be balanced against the power required to perform the compression and decompression in the first place. For data which is more random in nature, the bus-invert coding method [75] analyses successive data words and, if two words differ in more than half of their bits, the second item is inverted prior to transmission. To allow recovery of the data, a separate line signals when the inversion state has changed.

Another way of reducing bus activity is to use N-hot or N-of-M encoding schemes, where a value is represented by a high value on N lines out of M. This can be very efficient for small values of M, but to represent large numbers requires M to become prohibitively large, and it is impossible to perform arithmetic directly with all but the simplest and least compact coding schemes.

These encodings are of most use when a very large effective capacitance is being driven, such as in off-chip communication buses. In these cases the power penalty to encode and decode the data is outweighed by the large savings in power dissipated within the bus, and in general they can be combined with other power-saving techniques such as the use of reduced-swing drivers and receivers. While bus coding techniques can have a significant impact on the overall power consumption, they do not specifically impact on the power consumption within arithmetic elements as these encodings are not directly suitable for performing arithmetic operations.

Evaluation of number representations for DSP arithmetic

In order to investigate the effects of different number representations, a simulated 64 tap FIR low pass filter operation was performed on a 5.4 second excerpt of sampled speech ("Oh no, not cheese... can't stand the stuff. Not even Wensleydale?"). The models used for simulation were based on the data ALU of the Motorola DSP56000 series. This has 24 bit operands and 56 bit accumulators in 2s complement representation. The speech data and the FIR filter coefficients both had a precision of 16 bits in 2s complement representation.

As an initial study, the simple model shown in Figure 2.5 was used, in which only transitions at the multiplier inputs and outputs and the accumulator outputs were measured. The number systems used were 2s complement, sign-magnitude and modified signed-digit representation. The adding scheme used in the modified signed-digit model was based on that used by Takagi et al. [76]. The results obtained are shown in Table 2.1. The models were written in C++, where overloading of the assignment and arithmetic operators was used to produce data types which kept track of transition counts.

It can be seen that, at the multiplier inputs, 2s complement representation has more activity than either sign-magnitude or signed-digit forms. The extra transitions are due to fluctuations of the sign-extension bits, which are eliminated in the other two

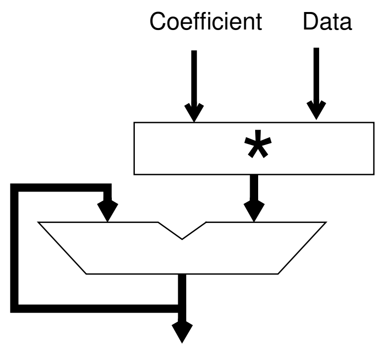


Figure 2.5 Multiply-Accumulate Unit Model.

Position	2s Complement	Sign-Magnitude	Signed digit
Data input	7.5	5.8	6.4
Coefficient input	8.3	5.7	6.5
Multiplier output	20.7	10.9	15.5
Accumulator output	14.8	11.5	17.0

Table 2.1: Average Transitions per Operation

representations. However, signed-digit representation shows greater switching activity than sign-magnitude, and at the outputs of the multiplier and adder, the number of transitions is even greater. This is due to the extra bits required to represent each number. It can also be seen that the smoothing effect of the accumulator reduces the number of transitions for 2s complement representation.

The conclusion from this study is that redundant signed-digit representation is not suitable as the number representation throughout a system as the reduction in switching activity is questionable and the storage required is greater, although it still has a role in circuit components where carry propagation is to be avoided or where redundant representations can reduce circuit complexity, such as internal representations within multipliers [76], [77]. The results suggest that sign-magnitude has an overall advantage over 2s complement number representation, which merits further investigation taking into account more of the details of implementing the multiplier and adder in sign-magnitude and 2s complement arithmetics.

To this end, detailed models of MAC units using 2s complement and sign-magnitude number systems were developed, as shown in Figure 2.6 and Figure 2.7. The 2s complement MAC model was based on the arithmetic circuits from the AMULET 3 multiplier and adder [125]. The multiplier uses modified Booth encoding and a 4:2 carry-save compression tree for the partial products. The partial sum and carry are combined by a full adder with a fast carry resolution network at the final stage. The sign-magnitude MAC model used Booth coding of the multiplier, but used a modified signed-digit representation (+1 / 0 / -1 at each bit position) for the partial products to avoid sign extension and carry propagation, with 2:1 compression at each stage [77]. Again, the AMULET 3 adder circuit was used to combine the positive and negative portions of the result. The simulation models were again written in C++, but were extended so that they recorded not only transitions at the inputs and outputs of circuits, but also the internal transitions within the circuits.

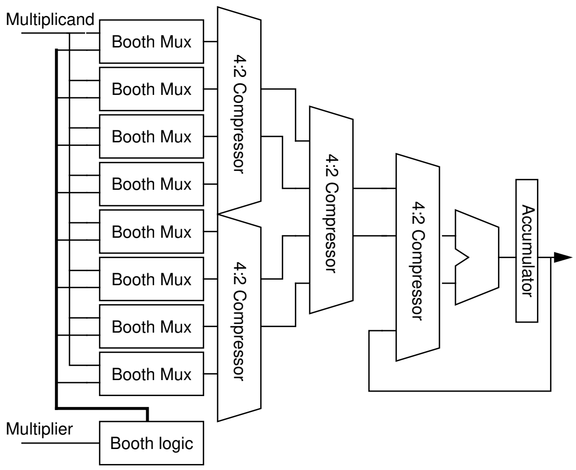


Figure 2.6 2s Complement Model Structure

The total numbers of transitions within the various sections of each MAC model are shown in Figure 2.8. It can be seen that, in almost all cases, the sign-magnitude number representation exhibits significantly fewer transitions. The total increase in switching activity for 2s complement number representation over the entire MAC unit is approximately 10%, although the increase is much greater in some sections. The greatest difference between the number systems is seen in the Booth multiplexers and the compressor tree. The use of modified signed-digit representation for the partial products

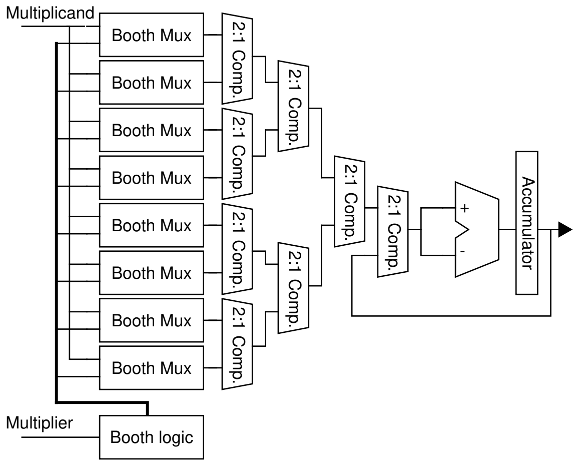


Figure 2.7 Sign-Magnitude Model Structure

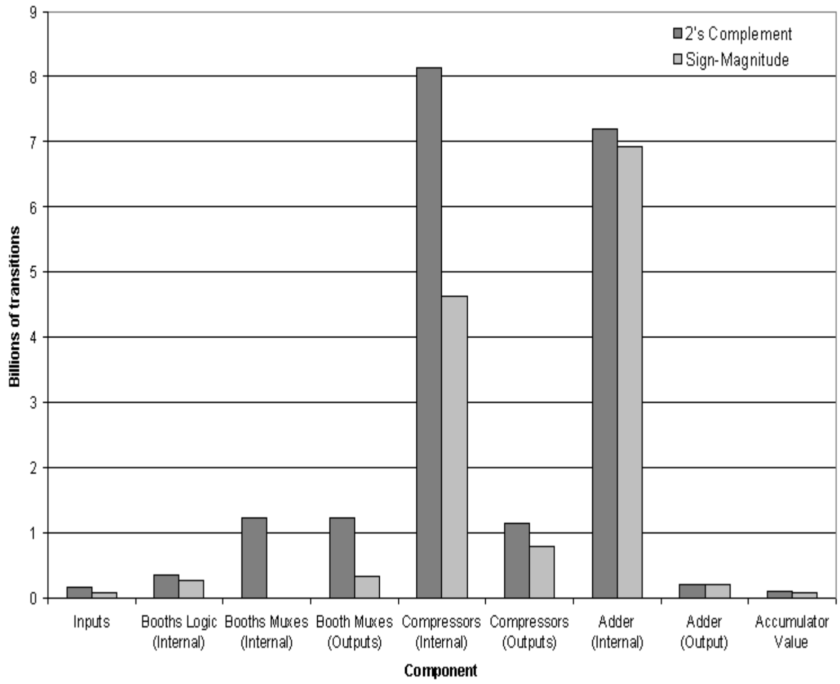


Figure 2.8 Total Transitions per Component

in the sign-magnitude MAC model means that the multiplexer requires no internal nodes, as negative values are produced simply by routing the number to the negative input of the compressor tree. Despite the fact that the sign-magnitude compressor tree has more stages, as it only has 2 inputs at each stage, the total number of transitions for the 2s complement compression tree is much greater due to the fluctuating sign-extension bits.

The differences between the number representations for the adder and the accumulator are much smaller. The main reason for this was the choice of adder circuit from the AMULET 3 processor that was used in both cases. This circuit uses dynamic logic, which causes most of the nodes of the circuit to be precharged high before each evaluation (behaviour which was modelled faithfully). This means that any node which evaluates to zero will undergo two transitions on every cycle. For a signal processor dealing with signals of wide amplitude range, there will often be zeros in the high bit positions and so a dynamic circuit will cause many more transitions than an equivalent static circuit. Transitions within the adder swamp the differences between the two number systems here, and when the internal nodes of the adder are not considered, the 2s complement MAC unit exhibits 96% more transitions than the sign-magnitude MAC.

The results from the modelled multiply-accumulate units using 2s complement and sign-magnitude number representations suggest that sign-magnitude number representation causes significantly fewer transitions within the circuits than 2s complement representation.

The bias to the results caused by the dynamic adder circuit indicates that dynamic logic may not be a good choice for low power circuits, despite the fact that it can offer high speeds and reduced area. The trade-off between the reduced node capacitance and the increased switching activity in dynamic designs merits further study.

The models used take no account of the capacitance of the various nodes in the circuit, and therefore the relative significance of a transition at each node on the energy consumption. It has been suggested that a bus line could have a capacitance 2-3 orders of magnitude greater than that of an internal node, and therefore a transition on the bus line would correspond to 100-1000 transitions at an internal node [39], [75]. This adds further weight to the advantage of sign-magnitude representation: while arithmetic using this representation is significantly more complex to implement, the savings in bus transitions could be expected to easily outweigh any additional transitions in the datapath as the datapath would be designed using minimum-geometry devices. The area penalty of the additional complexity is also becoming less significant, as design rules continue to ever smaller scales.

Overall, this study suggests a clear advantage in using sign-magnitude representation for digital signal processing when compared to 2s complement number representation, suggesting possible reductions in power consumption of greater than 50% where static circuitry is used.

Algorithmic transformations

Given sufficiently flexible processing structures, it is often possible to reorganise the manner in which a signal processing operation is performed to maximise the benefit obtained from data correlations, minimise the switching activity within the processing units and reduce the number of memory accesses required. For the case of the ubiquitous sum-of-products calculations, it is possible to calculate a number of successive outputs simultaneously, thereby keeping one of the inputs to the multiplier constant over a number of different calculations [79] [80] [81]. This can dramatically reduce the switching activity within the multiplier, and can also reduce the number of memory accesses required. Switching activity can be reduced further by reordering the sequence of both inputs so that the number of bits changing at each of the multiplier inputs is minimised, such as by reordering the filter coefficients in a FIR filter [82] or by analysing the data and coefficient characteristics for any general sum-of-products computation algorithm [83].

Reducing memory traffic

When performing a given operation on a set of data, it is necessary to read the data from a source and write it back to a destination. In DSP or microprocessor based systems, this data will typically reside in one or more memories and the power dissipation associated with accessing these memories can form a significant proportion of the total system power consumption. Also, in programmable systems, the instructions defining the algorithm must be fetched from memory.

The power dissipated by memory accesses can be broken into two main areas. The first main area is the power dissipation within the memory units themselves, by the address decoding logic, the precharging of bit lines and sense amplifier currents. These components generally increase in magnitude with increasing memory size.

The second main area is the power dissipated in transmitting the required signals across the large capacitances of the buses between the memory and where the data is required. When data resides within off-chip memories, this component can be orders of magnitude greater than when the data is located on-chip; but even when the memories are on-chip, interconnect capacitances are becoming increasingly significant in overall power dissipation due to shrinking feature size in logic circuits.

It is clear that two factors adversely affecting the power consumption of memory accesses are the size of the memory and the distance of the memory from where the data is required. Therefore, the use of a single large memory servicing an entire system is the worst possible case for power consumption. However, other aspects of system design may make this the only practical solution, particularly when processing large data sets.

It is possible to reduce the impact of this power dissipation by exploiting locality of reference: data tends to be reused (particularly in many DSP algorithms), and a limited set of data tends to be processed in a given period of time. This allows power savings to be made by making copies of the data from the main memory in smaller memories, located closer to where the data is required. When the data is accessed a large number of times, this can give considerable power savings and also gives faster data accesses; indeed, much of the previous work on memory hierarchies has looked solely at the speed benefits.

Two alternative (although not necessarily mutually exclusive) styles of memory hierarchy exist. In traditional microprocessors, caches are used: these are at least partially transparent from the viewpoint of the programmer. Requests for data in memory are automatically checked against the contents of the cache memory at each level in the hierarchy. The data item is supplied from the cache if found; if not, the cache is automatically filled with the required data and the neighbouring data from main memory. In signal-processing and real-time systems, it is more common for the smaller memories to be explicitly under the control of the programmer. This allows finer control of the timing for critical sections of execution. From the power viewpoint, the data look-up mechanism of cache memories represents an overhead, particularly in set-associative caches where content-addressable memory is used.

The size and number of memories in the hierarchy has a strong effect on memory access power consumption and speed [84]. For a given algorithm operating on a particular set of data, it is clear that there must be a certain minimum number of memory accesses; and it also seems intuitively reasonable that there must be a memory hierarchy organisation that can minimize the power consumption in a given case. However, even when all of the access patterns are known, the search space to minimize the overall power and area cost of the memory hierarchy is very large although methods have been described that attempt to formalise the problem and make it more tractable [85].

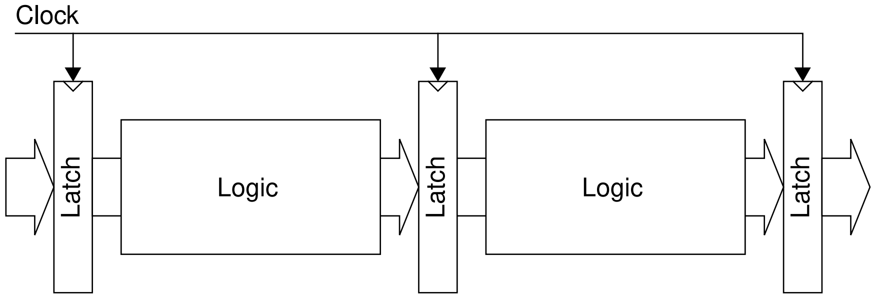
When redundancy exists in the data being read from memory, an effective way of reducing both the amount of memory required to store the data and the number of memory accesses required to read it is to use standard data compression algorithms [75]. The power and time penalty of encoding and decoding the data must be balanced against the possible savings due to reduced memory activity and reduced memory size; however, if the compression happens between stages in the memory hierarchy so that data needs to be compressed and decompressed infrequently, significant power reductions would appear to be possible.

Similar techniques can be applied to the instruction stream. It is possible to exploit the fact that only a limited number of instructions are typically executed, by storing the instructions in advance in a small look-up table. Instead of fetching whole instructions from memory, only the index into this look-up table needs to be fetched. Studies of this technique for both RISC microprocessors [86] and DSPs [87] have demonstrated considerable reductions in code size and memory activity.

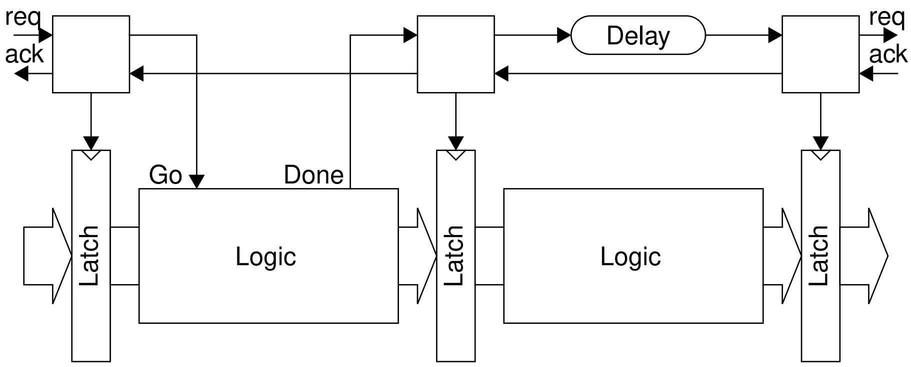
2.3 Asynchronous design

In a conventional (synchronous) system, all activity is governed by the clock as shown in Figure 2.9 (a). Data is captured by latches in the pipeline at a particular point in the clock cycle, and the processing logic between latches then has the rest of the clock cycle (minus the setup time on the latches) to produce the correct result.

By contrast, an asynchronous system has no overall clock: transfer of data is managed by local communication (handshakes) between adjacent processing elements. A class of asynchronous circuit known as a micropipeline [88] is shown in Figure 2.9 (b).



(a) Synchronous processing pipeline



(b) Asynchronous micropipeline

Figure 2.9 Synchronous and asynchronous pipelines

2.3.1 Asynchronous circuit styles

One of the most difficult aspects of asynchronous design is to determine when processing has finished in a stage. There are two main approaches to doing this, using either delay-insensitive circuits or bundled-data with matched delays.

Delay insensitive design

The delay-insensitive method adds redundancy to the data so that validity information is carried along with the data in the datapath: the *request* signal is implicit in the data. Formal definitions of the requirements for delay-insensitive coding schemes have been derived [91]. It can be demonstrated that one-hot and dual-rail encoding are valid DI schemes. One-hot coding is an N to 2^N line coding system, where a valid value is indicated by an active signal on one of the 2^N lines. Dual-rail encoding uses pairs of signals to represent a single bit, where typically (0,0) indicates invalid data, (1,0) and (0,1) indicate a 1 or 0 respectively, and (1,1) is not used and is undefined. A desirable feature of a DI coding scheme is to be able to exploit concurrency in the processing, by splitting the data into one or more sections which can be processed independently. This can be done when individual bits are represented with dual-rail coding, but not when the entire value is represented by one-hot coding. A compromise between the two approaches is to use 1-of-4 encoding, where two data bits are represented by a value on one of four signals.

Dual-rail circuits may be implemented using dynamic structures such as the AND gate shown in Figure 2.10, where the desired logic function is replicated in true and complement form in the n-stacks of the gate. When the input is inactive (all inputs low) neither n-stack conducts. When a valid set of inputs is presented, one or other n-stack will conduct and discharge the appropriate output node. The outputs of the circuit are then inverted ready to drive the next stage, so that the subsequent stage will only evaluate when the preceding circuits have completed their evaluation (so-called domino logic).

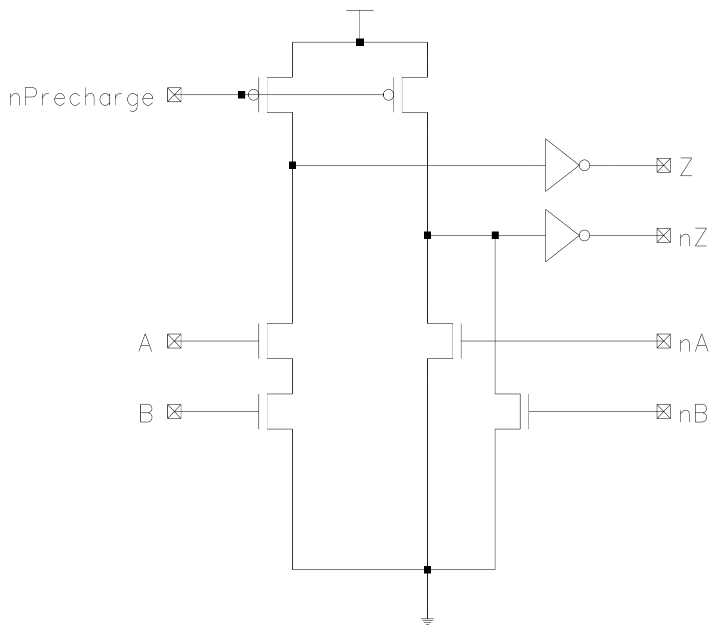


Figure 2.10 Dual-rail domino AND gate

The fact that dual-rail logic is delay insensitive means that it is possible to generate circuits automatically from specifications written in programming languages such as Tangram [92] [93], CSP [94], CCS [95] or Balsa [96]. These circuits can be proven to operate according to their specification. A drawback is that they tend to be larger than necessary, and possibly slower than a conventional circuit implementing the same function. A method for simplifying such circuits by repeated provable refinements has been demonstrated [97] which allows some of this complexity to be reduced. It has been shown that commercial boolean logic simplification tools such as Synopsys can be used with one class of delay-insensitive logic known as Null Convention Logic (NCL), by mapping NCL gates to semi-equivalent boolean 'image' gates [98].

Dual-rail circuits lend themselves to being used in very fast iterative structures, as they are inherently self-timed and can operate with negligible control overhead when driving other dual-rail circuits. An iterative division circuit has been developed [99] for which the critical path consists purely of dual-rail arithmetic elements without control circuit overhead, and for which the signal statistics have been analysed and the common cases made fast by appropriate transistor sizing. Dynamic circuits are commonly used for the highest performance synchronous systems, due to the reduced node capacitance and the fact that dynamic logic stages can incorporate latching functions without additional latency. Some of the highest performance asynchronous pipelines reported to date have been designed using these techniques, achieving throughputs of 860 million items per second for a dual-rail design with completion detection, and up to 1.2 billion items per second for a single-rail design without completion detection [100].

The drawback with dual-rail design is that the duplication of the logic function for the true and complement cases requires that the circuits consume more power and occupy more area than a conventional single-rail circuit. Dual-rail implementations of circuits have been shown to require approximately twice the number of transistors compared to a conventional single-rail circuit [101]. Also, the need to precharge every node for each data item when using a dynamic implementation increases switching activity, with one transition always occurring in dual-rail circuits. The alternative would be to use conventional static gates, but the inclusion of a P-stack exacerbates the area penalty inherent with dual-rail circuits and may also increase the switched capacitance.

A number of ways of reducing the complexity and/or power consumption of dual rail circuits have been proposed. A modified form of the dual-rail logic has been developed where the n-stacks are isolated from the precharged output nodes before evaluation [102]. This avoids charging and discharging the capacitances of the whole n-stacks and gives improved overall power consumption when compared to conventional dual-rail circuits. However, charge sharing requires that a swing-restoring amplifier be included in each stage to restore the output signals to the correct values.

A simpler way of reducing the complexity of dual-rail circuits is to use dual-rail logic only on the critical path, and have single-rail static logic to implement the remaining logic functions. Care is required when interfacing between static circuits and dual-rail dynamic logic to prevent errors due to glitches from the static circuitry. It has been shown that this can be done safely as long as the static signals are stable before the dual-rail signals arrive [103] [104].

When typical data characteristics can be exploited, it is actually possible to make a dual-rail circuit that has a lower transistor count than a synchronous circuit of equivalent average throughput. A dual-rail self-timed 32 bit adder circuit has been developed [105], which uses a simple ripple carry structure and completion detection on the carry path. Assuming purely random data, the average maximum carry ripple length will only be 5 or 6 bits, which will give good average throughput. In practice, the average carry length of real data is somewhat longer than this, which means the throughput will not be quite as good as anticipated. Also, the completion detection circuit generally requires a logic tree with a fan-in equal to the width of the datapath, which adds to both the delay of the stage and the power consumption [106], [107]. This means that the power-delay product is not quite as good as the best synchronous alternative, but the circuit still has the best average throughput for its size. Some similar circuits that implement other methods of addition have been developed [108] [109] which may offer better power-delay products.

Dual-rail self-timed logic has been shown to work very well in iterative structures where the circuit can operate at its own speed with little control overhead, such as the division circuits mentioned previously. However, there is somewhat less advantage to be gained by using self-timed logic with variable completion times to implement a processing pipeline [106] [107] [110]. No benefit will be obtained by completing processing early if the subsequent pipeline stage is not free to accept the data. This may mean that the average performance obtained does not justify the power and area overheads of the dual-rail logic and completion detection circuits.

Bundled-data design

Bundled-data asynchronous circuits are based around micropipelines [88], in which the passing of data between adjacent processing elements is managed by handshakes as shown in Figure 2.11. An asserted *request* signal indicates that the data is valid: the receiving device captures the new data, and indicates this by asserting the *acknowledge* signal. The sending device then disasserts *request*, and the receiving device subsequently removes *acknowledge*. Variants of this protocol exist, which differ in when the data may be removed. Two examples are pictured: *broad* protocol maintains valid data up to the point where *acknowledge* is removed, while *broadish* protocol may remove the data at the same time as *request*.

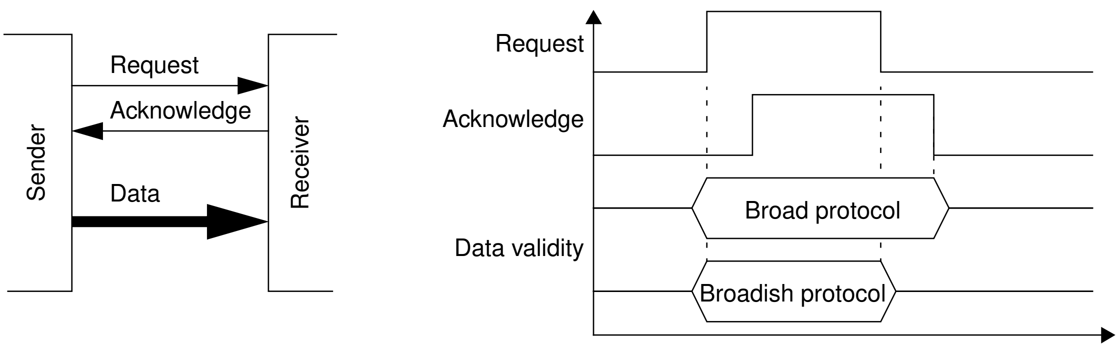


Figure 2.11 Handshakes in asynchronous micropipelines

These four operations define a 4-phase micropipeline: 2-phase pipelines are also possible, where request and acknowledge events are indicated by transitions on the appropriate signals. However, 2-phase control circuits are significantly more complex than 4-phase circuits.

Timing in bundled data circuits uses a delay in the control path that matches the processing delay in the datapath (although micropipelines can also use delay-insensitive techniques for particular stages, so that the datapath explicitly indicates when it has completed an operation). Where a matched delay is used, the delay must be at least equal to the worst case datapath delay. A typical way of doing this is to replicate the circuits in the critical path, as an extra bit. This means that variations in process and operating conditions should affect the matched delay in the same way as the datapath, avoiding the need for the same safety margins used in clocked designs. Also, a clocked design must cater for the global worst case, while only the local worst-case needs to be considered for the asynchronous bundled-data method.

One way to avoid always using worst-case delays and to achieve some data-dependent timing, without using a fully delay-insensitive design, is to have a number of different matched delays for different input data cases [110] [111] [112]. The case-detection circuitry operates concurrently with the datapath, and the appropriate delay is enabled according to whether worst-case or average-case input data is being presented. An example of this is a Brent and Kung adder [111] [112], where in 90% of cases the final stage of carry resolution is unnecessary. The case-detection circuitry looks for long carry propagate chains, and allows early completion if no such chains exist.

The main drawback with bundled-data designs are the extra designer effort required to verify that the delays match the data for the worst-case data under all possible process characteristics. However, where minimum power consumption is the main goal then the bundled-data method with a static CMOS processing pipeline offers reduced switching activity when compared to dual-rail dynamic implementations.

Asynchronous handshake circuits

While there is a certain amount of freedom in how to implement the datapath parts of asynchronous circuits, it is necessary to impose more restrictions when implementing the control portions. Handshake circuits are asynchronous state machines, and are defined by their *interfaces*, i.e. the sequences of transitions which their input and output signals can go through. This means that it is very important that the logic generating the signals must be hazard free, as a glitch could be viewed as an incorrect signal transition by a downstream stage.

A number of formalised methods exist for generating a circuit from a given specification, which vary according to the assumptions made about logic gates [113]. The strictest set of assumptions for designing circuits is the *delay-insensitive* model. This states that the delay through any part of a circuit, including a wire, is unbounded. However, the set of control circuits that can be produced using this assumption is limited. A more practical form of the assumptions is the *quasi delay-insensitive* model, where gates and wires are assumed to have an unbounded delay, but forks in wires are assumed to be *isochronic*, i.e. the delay on each path of a fork is the same. This allows more useful circuits to be generated [114]. A similar set of assumptions is made in the *speed-independent* model, but in this case the delay in wires is assumed to be negligible when compared to gate delays; effectively absorbing the wire delay into the gate. Both the QDI and speed-independent models can potentially experience problems when the isochronic fork assumption does not hold. However, if occurrences of forks are controlled (e.g. kept local to a single gate or a small number of gates) then they need not present very serious problems. Other assumptions in use are the *bounded-delay* assumption, where gates are assumed to have delays within a specific range, the *fundamental-mode* assumption [115] where circuit state is assumed to change between successive input changes, and the *burst-mode* assumption, where circuit state is assumed to have time to settle between successive *bursts* of input changes [116].

The work presented in this thesis uses the speed-independent model, with the specifications represented in *signal transition graph* (STG) form. An STG consists of an ordered network of signal transitions on the various input, output and internal signals of the circuit in question. An example is shown in Figure 2.12.

The STG of Figure 2.12 describes the relation between the *environment* (which sets the inputs a, b) and the circuit which generates the output c. Transitions on the signals are denoted by the name of signal followed by a '+' or '-' to indicate the direction of the transition. Each transition has one or more input arcs, and one or more output arcs. A transition is only *enabled* if all of the transitions on its inputs have fired; after which the transition can occur at any time. The circles containing dots represent the reset state of the

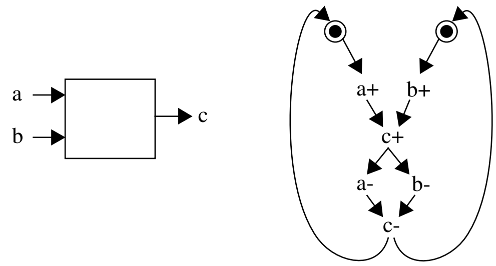


Figure 2.12 A simple signal transition graph (STG)

circuit: in this case, transitions a+ and b+ are enabled. This means that the environment is able to drive signals a and b high, although no ordering is specified: the transitions can occur at any time. When both transitions have occurred, transition c+ is enabled. This means that the circuit can drive its output high. After the transition c+ has fired, transitions a- and b- are enabled: the environment responds to c+ by driving a and b low in some unspecified order. Finally, once a and b are both low, the circuit can set c low, returning the circuit to the reset state. This specification describes a *Muller C-element*, a basic component in asynchronous designs.

The *Petrify* tool takes STG specifications, and synthesises hazard free speed-independent circuits from them [117]. For a hazard free circuit to be possible, a condition known as complete state coding (CSC) must be met by the specification (which can loosely be described as the state of all output and internal signals being precisely defined by the state of all other signals). If this condition is not met, Petrify attempts to add internal state signals to satisfy the condition while maintaining the original interfaces. However, this is an extremely computationally expensive task, and in practice the designer is best served by designing specifications which already have CSC and using Petrify to identify failures in this. The other condition that must be met for a speed-independent circuit is that once enabled, an output must always be able to complete its transition without being disabled.

2.3.2 Latch controllers for low power asynchronous circuits

Latch controller circuits [89] [90] are elements of micropipelines responsible for negotiating data transfers between stages, and passing the data at the appropriate time.

The latch controllers considered here use the *broad* and *broadish* classes of four-phase handshake protocol, as shown in Figure 2.11.

The latch controller opens and closes the data latches at the appropriate points in time, depending on the protocol being used and the operating mode. These operating modes are shown in Figure 2.13; the latch is open when enable is high. The extra transition required to open the latches and capture the data in normally-closed operation slows down the response of the handshake circuit. These circuits can then be built up into pipelines with processing logic between the sending and receiving latches. Timing is managed through either a completion signal from the processing logic or a matched delay in the handshake path. A typical pipeline structure is shown in Figure 2.9(b).

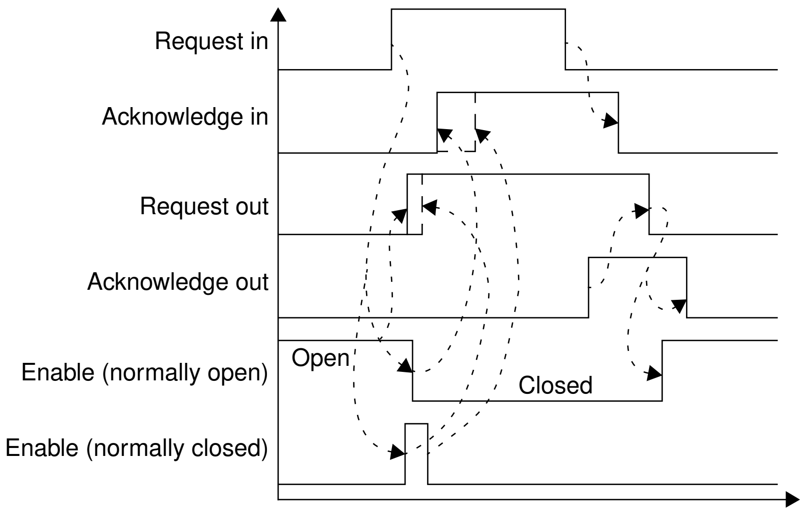


Figure 2.13 Pipeline latch operating modes

In many applications, a high maximum throughput is required but this maximum throughput is only needed for small periods of time, with periods of lower load between them. New forms of the broad and broadish protocol latch controller circuits have been developed within the group, based on the original normally-open latch controller designs that were already in use. The new reconfigurable latch controller circuits allow the operating mode of the pipeline latches to be selected by means of an external *Turbo* signal. When maximum throughput is required, the *Turbo* signal is made high and the latch controller circuit operates in normally-open mode. When the circuit is less heavily loaded, *Turbo* can be made low. The latch controller circuit then operates in normally-closed mode and spurious transitions are blocked.

It is necessary to generate the *Turbo* signal at some point in the circuit: one way of doing this would be to have a FIFO buffer at the input of the processing pipeline, and use the state of this buffer to control the operating mode in a manner similar to that used for adaptive voltage-scaling techniques [47]. Alternatively, *Turbo* may be placed under software control when used in microprocessor-based designs. However, the extra circuitry needed to generate and propagate the Turbo signal may add significantly to both complexity and power consumption, particularly if the operating mode changes frequently. An alternative solution, uses local timing information from the matched delays to open the latches just as the data stabilises, as shown in Figure 2.14. This means that the latches are ready to accept the new data just as it becomes available, allowing the same speed to be achieved as normally-open operation with fewer spurious transitions [124].

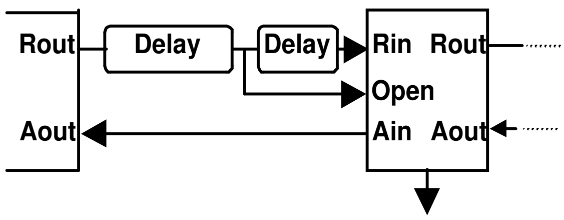


Figure 2.14 An early-open latch controller

The new latch controller circuits were tested with a substantial design consisting of a pipelined 32x32 bit multiplier datapath. This multiplier consists of four pipeline stages to generate the partial products and calculate the partial sum and partial carry, followed by a final adder stage to resolve the carries. The circuit is based around arithmetic elements of the AMULET3i processor [125]. It contains approximately 31000 transistors and occupies an area of 2.4x1.2mm in 0.35μm CMOS. Full-custom layout for the datapath was used, in order to provide more accurate results. Interconnection delays are becoming increasingly significant as design rules are scaled down. A full layout simulation displays timing behaviour significantly different from a circuit simulation that does not take these interconnection lengths into account. The generation and propagation of glitches is critically dependent on timing, so that precise simulations are essential for accurate power estimation.

All of the tests were performed on the broad and broadish protocol designs of the conventional normally-open latch controllers, the reconfigurable latch controllers in both normally-open and normally-closed operating modes, and the early-open latch controllers. Synopsys' *Timemill* was used to analyse the throughput of the pipeline. Random data was used for the input as the performance is not data-dependent.

Table 2.2 shows a reduction in maximum throughput between normally open and closed modes of 6.4% and 7.3% for the broad and broadish configurable latch controllers respectively. For the broad protocol designs, the variable mode designs also show a reduction in maximum speed when compared to the conventional latch controller, but no reduction was observed for the broadish protocol.

	Standard	Reconfigurable		Early Open
		Normally Open	Normally Closed	
Broad	101	97	90	95
Broadish	106	106	98	106

Table 2.2: Millions of multiplications per second with different latch controllers

A decrease in throughput is to be expected with the variable mode latch controllers due to extra complexity within the latch controller circuits. The variable mode latch controllers have an extra input on the gate controlling the latch enable signal. This requires a pair of extra transistors in the gate tree and also implies extra capacitance, both of which slow the critical path through the latch enable. The broadish protocol allows for the latches to be freed up before the Acknowledge cycle has completed at the output. This overlap hides the performance reduction in the broadish protocol when operating at maximum capacity.

Synopsys' *Powermill* was used to analyse the relative energy consumptions of the circuits. Tests were performed with each type of latch controller, for different levels of pipeline occupancy, as this strongly affects the power consumption due to the distance that spurious transitions may propagate. Also, the effect of skewing the inputs in time was investigated. Power consumption is strongly data dependent, and so tests were performed with both random input data and simulated data from an 8-pole FIR low-pass filter operation on an excerpt of sampled speech.

The graphs of energy consumed per operation against pipeline occupancy (with non-skewed inputs) are presented in Figure 2.15. These show that, as expected from the simple model of spurious transitions propagating along the pipeline, the difference between operating with normally-open and normally-closed latch controllers becomes very small when the pipeline is fully occupied. However, when operating with a single input value at a time, the difference between the operating modes becomes much more significant. A decrease in energy per operation of 21% was observed for normally-closed mode compared with normally-open mode, while the early-open latch controller displayed a decrease of 20-24% compared to standard designs. The difference becomes even greater when the multiplier and multiplicand inputs are skewed in time, giving a 32% and 26-28% decrease in energy respectively.

When operated with FIR filter input data and the configurable latch controller, the energy per operation was approximately halved, and there was much less difference between the operating modes (8%). This is due to correlations between successive multiplier values, and that with this data set, one input is usually held at a constant value between successive data points.

The results in Figure 2.15 show just how much energy can be dissipated by spurious transitions when the pipeline allows them to propagate. In asynchronous micropipeline-based circuits, this occurs when operating at less than maximum throughput. The presented techniques prevent this from happening. For the reconfigurable latch controllers, these techniques rely on a variable processing load and have the expense of some control overhead in generating the *Turbo* signal. For the early-open latch controllers, there is no significant circuit overhead and the overall speed of the latch controllers are not reduced significantly, meaning that a variable demand is not required. However, to be at its most effective the early-open technique relies on a certain amount of design effort to match the early-open signal to the opening time of the latches.

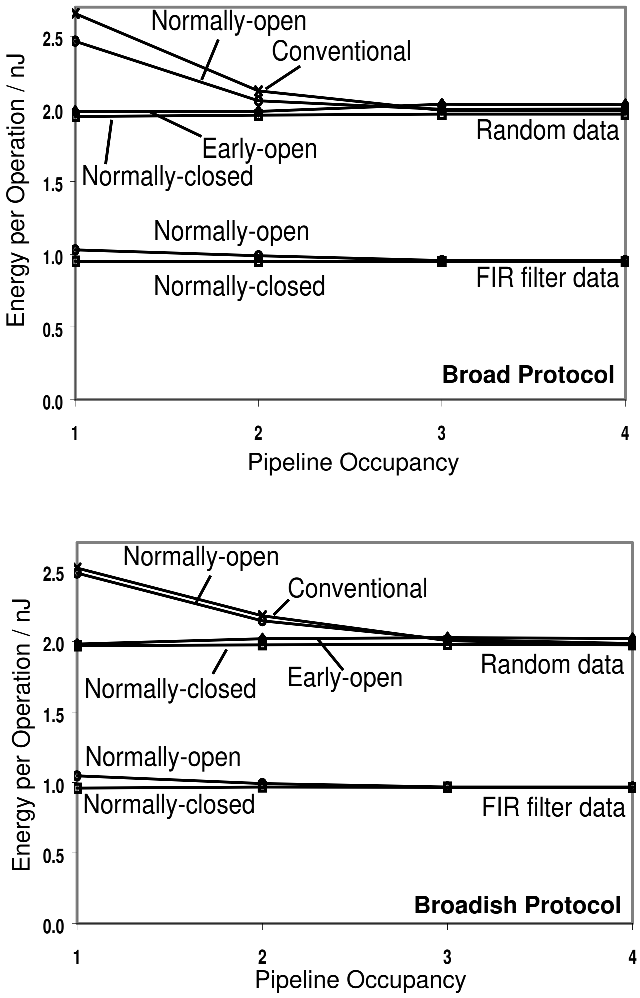


Figure 2.15 Energy per operation using different latch controller designs

2.3.3 Advantages of asynchronous design

Elimination of clock distribution network

Asynchronous circuits have a number of key advantages over clocked circuits when design for low power is being considered. The defining feature of a synchronous circuit is the global clock which must be distributed throughout the circuit. This causes unwanted switching activity at every node to which the clock is connected, whether or not that part of the circuit is performing useful work. The wide distribution means that the clock signals themselves have high capacitance and by definition undergo a power-consuming transition every cycle, so a significant amount of power is consumed in simply generating the clock signal. In high-speed synchronous circuits, the task of preventing clock skew when distributing the high frequency clock across the circuit requires considerable additional circuit overhead in terms of clock buffers and phase-locked loops. This is wasteful of both area and power. An example of the scale of this problem in modern processors is the DEC Alpha, in which around 30% of the total system power is spent simply in clock distribution [118]. In asynchronous circuits, this clock distribution network is replaced by local communication between stages, avoiding distribution problems. A certain amount of power is dissipated by the handshake circuits. However, designs of moderate-speed processors such as the AMULET2e demonstrate comparable levels of power consumption when compared with their synchronous counterparts. Clock distribution problems are increasing as processes shrink and clock speeds increase, so it can be expected that as higher levels of performance are reached then the power benefit of using asynchronous circuits will become noticeable.

Automatic idle-mode

Clock gating and similar techniques can be employed to reduce power consumption in sections of the circuit where no useful work is being done, but this involves extra circuitry and effort from the point of view of the system designer. Also, where phase-locked loops are used, it is necessary to have a sufficient delay from restarting the clock to allow the PLLs to stabilise. Asynchronous circuits inherently cease their switching activity when no work needs to be done, and can go from idle to full activity instantaneously. This behaviour occurs at a very fine grain both temporally and spatially, down to the level of a single handshake circuit, enabling much greater reductions in power due to idle components than can be achieved with any practical clock gating scheme. Idle power consumption is extremely important in embedded mobile applications, as the systems are effectively event-driven. The need to respond quickly to certain events limits the application of clock gating techniques, since an instruction must be used to un-gate the clock before subsequent instructions can use the resources in the gated circuit. By contrast, an asynchronous design will have immediate access to idle components.

Average case computation

In clocked systems, it is necessary to have the entire circuit operating at a speed governed by the slowest single circuit element as all data transfers are governed by the global clock. This means that circuits in the critical path often require considerable design effort and extra complexity to ensure that worst-case data can be dealt with inside the desired clock period. Asynchronous systems manage the transfer of data at a local level, and have flexibility in the time taken by any individual circuit stage. Asynchronous circuits can be designed with completion detection or data-dependent delay. This means that circuits can be designed to maintain a high throughput for typical data, and the pathological cases be simply given longer to complete. While the practical benefits of data-dependent delays may be limited by surrounding stages where a single type of operation is performed in a pipeline, the benefits of average case computation can be realised very effectively when most of the variation in delay is small except for rare worst cases.

Reduced electromagnetic interference

In synchronous systems, all activity is focused around the edges of the clock when data is passed through latches and processing logic calculates the next results. This localization in time causes sharp spikes in current consumption occurring around each active clock edge; and these spikes cause large amounts of electromagnetic energy to be radiated at harmonics of the clock frequency (as well as causing potential electromigration damage to power supply interconnections on chips). In contrast, an asynchronous system has its activity spread out: even when the overall throughput of a processing pipeline is fixed, new data ‘ripples’ through the pipeline with natural variations in each stage blurring any driving frequency. This means that asynchronous circuits radiate very little electromagnetic interference. A comparison of electromagnetic radiation has been performed of the asynchronous AMULET2e and the comparable clocked ARM processor executing the same programs [119], and the AMULET causes dramatically less radiated energy, without any visible harmonics. In contrast, the ARM processor has harmonic spikes visible in the spectrum going well beyond 1GHz (i.e. well into GSM mobile phone operating frequencies). Clearly, for wireless mobile communication devices it is important to minimise emissions from the digital components to avoid interference with the sensitive radio receiver circuits.

Modularity of design

Asynchronous circuits have precisely specified interfaces: with bundled data interfaces, output data is specified as being stable when (for example) an output request signal is asserted while with delay-insensitive interfaces, validity is encoded in the data itself. The precise specification simplifies the task of designing large systems. The task is reduced to that of designing the component modules and verifying that their interfaces are implemented correctly. The precisely defined interfaces also simplify integration of the asynchronous modules into the final system, as signal specifications are independent of any global timing reference and there is no need to worry about clock skew, and the module-based approach also simplifies design reuse.

Delay-insensitive interfaces offer the ultimate in composability, at the expense of some circuit overhead. As wire delays become increasingly significant, the task of ensuring that a number of different on-chip peripherals all function together in a reliable manner is becoming a very serious issue. DI interfaces are guaranteed to work correctly, regardless of wire delays. As there is no need to build timing margins into DI signals, they can operate significantly faster than other distribution techniques; allowing the wiring overhead to be reduced by time-division multiplexing of signals. The power cost of driving the wiring capacitances can be made equivalent to a single-rail bus carrying random data by using 1-of-4 encoding with 2-phase transition signalling: a single signal transition is required for every two bits transmitted. This form of encoding needs very simple circuits at the transmitter and receiver.

One possible solution to the problem of composing heterogeneous systems-on-chip is to use asynchronous interconnections between modules, with a mixture of asynchronous and synchronous modules. Synchronous modules are surrounded by an asynchronous wrapper with a locally-generated clock which can be stopped and started as required [120] [121].

2.3.4 Disadvantages compared to clocked designs

Lack of tool support

The development of the clock as a design aid was originally intended to simplify the design and verification of circuits, by separating the task of designing the logic function from that of designing the timing specification. If a logically correct circuit would not work at a given speed, then the clock could simply be slowed down until it could generate the output in the given time with sufficient additional margin. So successful was this premise that over the last thirty years, vast amounts of money have been invested in computer-aided design tools and methods. However, with the drive for ever-increasing clock speeds and smaller process sizes, the paradigm of the clock has begun to cause as many problems as it is solving. Despite this, the semiconductor industry has such an investment in synchronous design tools it is unlikely to relinquish the techniques quickly, except in very specific applications.

The dominance of synchronous design means that there are virtually no commercial design tools available explicitly to support the asynchronous designer. Many of the design tools can be used equally well in either field, such as schematic or layout editors, but tools such as automated logic synthesis and formal verification tools are still only available from academia; and technology mapping and automated place-and-route tools that are aware of the issues required by asynchronous designs are still unavailable.

Reduced testability

Testing is extremely important for any commercial VLSI device, to detect defects in manufacturing. Typically in clocked circuits, testing is performed by a scan-path interface where pipeline latches operate as a large shift register. This allows test patterns to be fed through the datapath to check correct operation. The difficulty in testing asynchronous circuits is that they tend to contain very much more state information than clocked circuits. As well as the pipeline latches, every handshake circuit contains memory elements which must be included in the test process to be certain that no faults exist.

In practice it may be possible to use standard synchronous test pattern generators to produce test vectors for bundled datapaths, and to use knowledge of the handshake circuits to manually design tests for those components. Certain classes of DI circuit such as NCL have very good testability properties; and as the control and datapath functions are merged to some extent they can be tested with an appropriate set of input vectors.

Another approach for testing asynchronous circuits is to use built-in self test, where a test pattern generator feeds test inputs through the device and checks that the correct results appear. This can be applied to circuits which have no specific design-for-test features, with little impact on performance or total area [122].

Chapter 3: CADRE: A new DSP architecture

3.1 Specifications

The OAK DSP in the GEM 301 baseband processor maintained a maximum throughput of about 40 MIPS, and it is claimed that all of the baseband functions for GSM require a total of 53 MIPS [29]. Based on this, it is expected that the next generation of mobile phone chipsets will require a throughput of greater than 100MIPS from the DSP. A target performance of 160MIPS has been chosen for the new design presented in this thesis, which is intended to meet the requirements for this application comfortably and represents an approximately fourfold increase in throughput over the OAK chip.

The GSM standard specification requires 16 bit arithmetic with 32 bit accumulators, but an additional 8-bit guard portion for the accumulators is to be included in the new design to give a total of 40 bits: this simplifies program design by allowing up to 128 summations before overflow is possible. It is envisaged that this processor will be operating in conjunction with a 32-bit microcontroller such as an ARM, so interfaces to memory are 32 bits wide, as are the instructions. The new processor is to have a 24-bit address bus width, thereby allowing memory addresses to be comfortably stored as immediate values within the 32-bit instructions.

3.2 Sources of power consumption

The power consumption in an on-chip processing system as described here can be broken down into two main areas. The first main area is the power cost associated with accesses to the program and data memories. This is made up of the power consumed within the 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, and a study of the Hitachi HX24E DSP [130] 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.

The second main area of power consumption comes from the energy dissipated while performing the actual operations on the data within the processor core. This is made up of the energy dissipated by transitions within the datapath associated with the data, and the control overhead required to perform the operations on the data.

3.3 Processor structure

The challenge for the new DSP is to meet the required throughput without excessive power consumption. An instruction rate of 160 MIPS is not large when compared with current high-performance microprocessors. However, the demands of low power consumption and low electromagnetic interference mean that lower operating speeds are preferred. Meeting the required throughput at a lower operating speed necessitates the use of parallelism, where silicon die area is traded for increased throughput. This allows simpler and more energy efficient circuits to be used within each processing element, and for the supply voltage to be reduced for a given throughput (architecture driven voltage scaling, as described in section 2.2.1 on page 43). Multiple functional units also provide flexibility for the programmer to rearrange operations so as to exploit correlations between data [126]. Silicon die area is rapidly becoming less expensive; indeed, one of the emerging challenges is to make effective use of the vast number of transistors available to the designer [127]. This makes parallelism and replication very attractive. Most new DSP offerings by the major manufacturers incorporate some form of parallelism, such as the LSI Logic Inc. ZSP164xx DSPs [128] with 4-way parallelism or the Texas Instruments TMS320C55x low-power DSPs [129] which feature two multiply-accumulate units and two ALUs.

3.3.1 Choice of parallel architecture

The OAK DSP core in the GEM301 baseband processor maintains a maximum throughput of approximately 40MIPS when engaged in a call using a half-rate codec. This is a uniscalar device, and so four way parallelism has been chosen to reach the required throughput of 160MIPS. Four-way parallelism also gives near-optimal power reduction according to analyses of architecture-driven voltage scaling [38] [40]. The choice and layout of the functional units were decided upon by examining a number of key DSP algorithms [9] to see how parallelism could be exploited. To give a starting point for the instruction set, the benchmark algorithms for the Motorola 56000 DSP series [14] were chosen, as the author has some experience with this range of processors. The chosen algorithms were FIR filters, IIR filters and fast Fourier transforms; the FIR filter and FFT will be illustrated here.

FIR Filter algorithm

The first algorithm considered was the FIR Filter algorithm. This is expressed by the equation $y(n) = \sum_{k=0}^{M-1} c_k x(n-k)$ and there are clearly a number of ways in which this sum of products can be implemented in parallel form. The time-consuming portion of this algorithm is the succession of multiply-accumulate (MAC) operations and so, to speed up execution by a factor of four, it is necessary to have four functional units capable of performing these multiply-accumulate operations.

A simple way of distributing the arithmetic for this algorithm is to have each MAC unit process a quarter of the operations on each pass of the algorithm, storing the partial sum in a high-precision accumulator within the unit. At the end of the pass, a final summation of the four partial sums is performed. These final sums require additional high-precision communication paths between the functional units to avoid loss of precision, and to perform the sum in the shortest possible time requires two of these pathways (assuming only 2-input additions). The distribution of operations to the various functional units (*MAC* A-D) is shown in Table 3.1.

Arithmetic operations are of the form, ‘`operation src1,src2,dest`’ where `src1` and `src2` are 16 or 40 bit values and `dest` specifies the destination accumulator. Where one of the sources is an accumulator from another functional unit, the notation `mac[a-d]:src` is used to indicate which functional unit and accumulator is involved. The `mpy` operation is a 16x16 bit multiply, the `mac` operation is a 16x16 bit multiply with the result being added to the destination accumulator, and the `add` operation is a 40 bit addition. Bold type indicates the operation in the algorithm after which the result is available.

MAC A	MAC B	MAC C	MAC D
mpy x_n,c_0,a	mpy x_{n-1},c_1,a	mpy x_{n-2},c_2,a	mpy x_{n-3},c_3,a
mac x_{n-4},c_4,a	mac x_{n-5},c_5,a	mac x_{n-6},c_6,a	mac x_{n-7},c_7,a
⋮	⋮	⋮	⋮
mac x_{n-i},c_i,a	mac x_{n-i-1},c_{i+1},a	mac x_{n-i-2},c_{i+2},a	mac x_{n-i-3},c_{i+3},a *(i = 4,8,…)*
⋮	⋮	⋮	⋮
mac x_{n-M+4},c_{M-4},a	mac x_{n-M+3},c_{M-3},a	mac x_{n-M+2},c_{M-2},a	mac x_{n-M+1},c_{M-1},a
–	add maca:a,a,a	–	add macc:a,a,a
–	–	–	**add macb:a,a,a**

Table 3.1: Distribution of operations for simple FIR filter implementation

When more than one item of new data is available at a time (such as when processing is block-based) it is possible to optimise the FIR filter algorithm to reduce power consumption, by transforming the algorithm so that 4 new data points are processed on each pass. The transformed sequence of operations is shown in Table 3.2. The benefit of this transformation is that correlations between both the data values and the filter coefficients can be exploited. In the new arrangement, the filter value is held constant at one input of the multiplier over four successive multiplications while successive data values are applied to the other input. This dramatically reduces the amount of switching activity within the multiplier, at the expense of requiring more instructions and more accumulator registers in each functional unit. Where the coefficients are being read from main memory, this technique also reduces the frequency of coefficient reads by a factor of four. This technique can be extended to use as many accumulators as are implemented within the functional units [81] [83]; however, it was felt that 4 accumulators per functional unit gave a good trade-off between complexity and possible power savings, and was sufficient to implement the algorithms under consideration in an efficient manner.

Each functional unit now maintains 4 partial sums, one for each of the passes of the FIR filter algorithm, and these partial sums are again brought together at the end of processing. In this case, 4 high precision pathways between the functional units would be beneficial, but this represents too great an area overhead. Instead, it was noted that the summation of results across the functional units occurs in a pairwise fashion, and so it was decided to group the functional units into two pairs (Mac A and B, Mac C and D) connected by local high precision buses, with all four units connected by a single global high precision bus. As a shorthand, these buses are named LIFU1&2 (Local Interconnect of Functional Units) and GIFU (Global Interconnect of Functional Units). This arrangement, as shown in Figure 3.1, provides the benefits of having three high precision pathways for most operations, but incurs the area expense of only two global pathways. Driving shorter local buses also causes less power consumption. Despite only having three pathways to perform summations over, it is still possible to keep all of the functional units occupied by interleaving the summation of the partial results with the final set of multiplications. Details of this have been omitted from Table 3.2 for the sake of clarity.

MAC A	MAC B	MAC C	MAC D
mpy x_{n}, c_{0}, a	mpy x_{n-1}, c_{1}, a	mpy x_{n-2}, c_{2}, a	mpy x_{n-3}, c_{3}, a
mpy x_{n-1}, c_{0}, b	mpy x_{n-2}, c_{1}, b	mpy x_{n-3}, c_{2}, b	mpy x_{n-4}, c_{3}, b
mpy x_{n-2}, c_{0}, c	mpy x_{n-3}, c_{1}, c	mpy x_{n-4}, c_{2}, c	mpy x_{n-5}, c_{3}, c
mpy x_{n-3}, c_{0}, d	mpy x_{n-4}, c_{1}, d	mpy x_{n-5}, c_{2}, d	mpy x_{n-6}, c_{3}, d
⋮	⋮	⋮	⋮
mac x_{n-j}, c_{j}, a	mac x_{n-j-1}, c_{j+1}, a	mac x_{n-j-2}, c_{j+2}, a	mac x_{n-j-3}, c_{j+3}, a
mac x_{n-j-1}, c_{j}, b	mac x_{n-j-2}, c_{j+1}, b	mac x_{n-j-3}, c_{j+2}, b	mac x_{n-j-4}, c_{j+3}, b
mac x_{n-j-2}, c_{j}, c	mac x_{n-j-3}, c_{j+1}, c	mac x_{n-j-4}, c_{j+2}, c	mac x_{n-j-5}, c_{j+3}, c
mac x_{n-j-3}, c_{j}, d	mac x_{n-j-4}, c_{j+1}, d	mac x_{n-j-5}, c_{j+2}, d	mac x_{n-j-6}, c_{j+3}, d *(j = 4, 8, …)*
⋮	⋮	⋮	⋮
mac x_{n-M+1}, c_{M-4}, a	mac x_{n-M}, c_{M-3}, a	mac x_{n-M-1}, c_{M-2}, a	mac x_{n-M-2}, c_{M-1}, a
mac x_{n-M}, c_{M-4}, b	mac x_{n-M-1}, c_{M-3}, b	mac x_{n-M-2}, c_{M-2}, b	mac x_{n-M-3}, c_{M-1}, b
mac x_{n-M-1}, c_{j}, c	mac x_{n-M-2}, c_{M-3}, c	mac x_{n-M-3}, c_{M-2}, c	mac x_{n-M-4}, c_{M-1}, c
mac x_{n-M-2}, c_{j}, d	mac x_{n-M-3}, c_{M-3}, d	mac x_{n-M-4}, c_{M-2}, d	mac x_{n-M-5}, c_{M-1}, d
add macb:a,a,a	add maca:b,b,b	–	add macc:a,a,a
add macd:a,a,a	–	add macd:c,c,c	add macc:b,b,b
add macb:c,c,c	**add macd:b,b,b**	–	add macc:d,d,d
–	add maca:d,d,d	**add maca:c,c,c**	–
–	–	–	**add macb:d,d,d**

Table 3.2: Distribution of operations for transformed block FIR filter algorithm

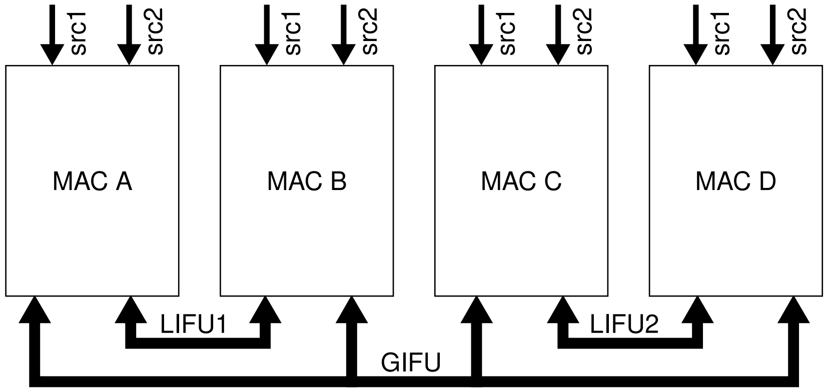


Figure 3.1 Layout of functional units

Fast Fourier Transform

The fast Fourier transform is actually a 'parallelised' form of the discrete Fourier transform described by the equation $X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi k\frac{n}{N}}$. The algorithm consists of a series of passes of the 'FFT butterfly' operator across the data. The butterfly operates on two (complex) data values *a* and *b* to produce two output data values *A* and *B* according to the equations $A = a + W_i b$ and $B = a - W_i b$, where W_i is the value of a complex exponential (the so-called 'twiddle factor'). The calculation of each butterfly requires a complex multiply and two complex additions. In general, the complex multiplication $W_i b$ requires four real multiply operations and two real additions, to calculate $Re(W_i \times b) = Re(W_i) \times Re(b) - Im(W_i) \times Im(b)$ and $Im(W_i \times b) = Im(W_i) \times Re(b) + Re(W_i) \times Im(b)$. Two further complex additions are then required to generate A and B, requiring four real additions in total. However, if the functional units support shifting of one of the operands, to produce a multiplication by a factor of two, then it is possible to avoid two of the final additions by using the following algorithm:

$$Re(A) = Re(a) + Re(W_i) \times Re(b) - Im(W_i) \times Im(b) \tag{10}$$

$$Im(A) = Im(a) + Im(W_i) \times Re(b) + Re(W_i) \times Im(b) \tag{11}$$

$$Re(B) = Re(a) - Re(W_i) \times Re(b) + Im(W_i) \times Im(b) = 2 \times Re(a) - Re(A) \tag{12}$$

$$Im(B) = Im(a) - (Im(W_i) \times Re(b) + Re(W_i) \times Im(b)) = 2 \times Im(a) - Im(A) \tag{13}$$

A natural way of performing these calculations within the functional units is to use them in pairs, to perform the complex operations for two butterflies simultaneously. The mapping of the FFT butterfly is shown in Table 3.3. This mapping requires two write ports to the accumulator bank in each functional unit, so that the moves can take place in parallel with the operations (with read-before-write sequencing being enforced within the functional units). The italicised move operations only require a separate instruction on the first FFT butterfly of each pass, as they can take place in parallel with the final add of the accumulators when a number of butterflies are being performed in succession. A full implementation of this algorithm can perform 4 complex FFT butterflies with 6 parallel instructions, with all of the functional units fully occupied throughout.

MAC A	MAC B	MAC C	MAC D
move a_{1r},a	*move a_{1i},a*	*move a_{2r},a*	*move a_{2i},a*
move a,b mac W_{1r},b_{1r},a	move a,b mac W_{1i},b_{1r},a	move a,b mac W_{2r},b_{2r},a	move a,b mac W_{2i},b_{2r},a
mac −W_{1i},b_{1i},a	**mac W_{1r},b_{1i},a**	**mac −W_{2i},b_{2i},a**	**mac W_{2r},b_{2i},a**
add 2b,−a	**add 2b,−a**	**add 2b,−a**	**add 2b,−a**

Table 3.3: Distribution of operations for FFT butterfly

Choice of number representation

The study of number representations presented in section 2.2.3 on page 58 showed that sign-magnitude representation offered significantly reduced switching activity for DSP algorithms, and so this arithmetic has been used within the new DSP. The reduced switching activity due to the data representation affects power consumption throughout the system. This is particularly significant when the large capacitance of system buses to memory is considered.

3.3.2 Supplying instructions to the functional units

Having chosen a parallel structure for the processor, the next challenge is to devise a method of supplying independent instructions to the functional units at a sufficient rate without excessive power consumption. In a general-purpose superscalar microprocessor, this task is often managed by a dedicated scheduling unit which analyses the incoming instruction stream and dispatches independent instructions to the available resources. This approach has been adopted by ZSP Corporation for the ZSP164xx DSPs. However, the scheduling unit is a complex device which consumes significant amounts of power, so for power-critical applications it makes more sense to remove this task from the processor. Instead, the programmer (or, more often, the compiler) can group independent instructions, in advance, into a single *very long instruction word* which can be read from memory and directly dispatched to the functional units. The VLIW approach is becoming the more common method for managing parallelism in current DSPs. The main drawback with conventional VLIW is that, where dependencies exist, it is necessary to insert NOPs within the instruction word which reduce the code efficiency. This can be tackled to some extent by using variable length instructions, such as the EPIC (Explicitly Parallel Instruction-set Computing) technique [131] at the expense of greater complexity of instruction decoding. Variable length instructions of this type are employed in the Texas Instruments TMS320C55x DSPs. However, in the case of both superscalar and VLIW approaches it is necessary to fetch instruction words from program memory at the full rate demanded by the functional units.

DSP operations tend to be characterised by regular repetition of a number of short, fixed algorithms. It is possible to exploit this characteristic to reduce the quantity of information that needs to be fetched from program memory, thereby reducing power consumption. One possible method would be to cache the incoming instruction stream, to exploit the locality of reference in the memory accesses. However, an energy overhead is associated with the process of searching for a hit in cache memory, particularly when multi-way associative caches are used. In addition, it is still necessary to fetch instructions and update the program counter at the full issue rate of the processor or to use a very wide instruction path.

In CADRE, the VLIW encodings for the required instructions can be stored, in advance, in configuration memories located within the functional units themselves. These stored operations can then be recalled with a single word from program memory, dramatically reducing the amount of information that needs to be fetched, and also reducing the required size of main memory. Commercial DSPs already exist which make use of configurable instructions, such as the Philips REAL DSP core [132] or the Infineon

CARMEL DSP core [133]. However, both of these have a single global configuration memory for the entire core, which is only used for specialised instructions. The scheme adopted in CADRE differs in that *all* parallel execution is performed using preconfigured instructions. Compressing instructions and reducing instruction fetch activity by means of a look-up table has been proposed before, for embedded microprocessors [86] and DSPs [87]; however, in these cases a simple index into the look-up table was used to refer to the instructions, and single look-up table was used for the entire processor. In the new design, two separate indices are used to specify different aspects of parallel operation, and components of the parallel operations can be flexibly disabled or made conditional when the instructions are recalled. Also, the configuration memory is broken up, with separate configuration memories located within each functional unit, to reduce the distance over which the data needs to travel and hence the power consumption. Locating the memories within the functional units also increases modularity, and allows any arbitrary type of functional unit to be inserted into the architecture (although to speed design, identical functional units are being used in the prototype). In the current design the configuration memories are RAMs, allowing reconfiguration at any point in execution. For a given application, it may be desirable to turn part of this storage into ROM to encode a few standard algorithms. The configurable nature of the new DSP leads to its name: CADRE-Configurable Asynchronous DSP for Reduced Energy.

3.3.3 Supplying data to the functional units

Given a parallel processing structure, and a means of supplying instructions to it, the next design issue is to supply data at a sufficient rate, without excessive power consumption. This is clearly a serious problem, as each functional unit can require two operands per operation and may also need to write data back from the accumulators, giving a total of eight reads and four write accesses per cycle.

CADRE, in common with many other current DSPs, uses a dual Harvard architecture where one program memory and two separate data memories (labelled X and Y) are used. This avoids conflicts between program and data fetches, and many DSP operations map naturally onto dual memory spaces (e.g. data and coefficients for a FIR filter operation). The memory hierarchy principle works well for DSPs, as many algorithms display strong locality of reference. For this reason, a large register file of 256 16-bit words was included in CADRE, segmented into X and Y register banks to match the main memory organisation.

The large register file allows for a high degree of data reuse (allowing, for instance, a complete GSM speech data frame of 160 words to be stored), and a large explicit register file offers a significant advantage over having a cache and fewer registers as is common in traditional DSP architectures. In the programmer's models of most traditional DSP architectures, as shown in Figure 3.2a, operands are treated as residing within main memory and are accessed by indirect reference using address registers. These address registers must be wide enough to address the entire data space of the processor, 24 bits in this design. After each operation, it is generally necessary to update these address registers to point to the next data item. The data address generators (DAG) generally provide support for the algorithm being executed, with circular buffering or bit-reversed addressing, and therefore require complex circuitry. Even if all eight of the fetched data items reside within the cache, there is still a significant power consumption associated with these address register updates (up to eight of them), and this power must be added to that required for the cache lookups.

In the new architecture (Figure 3.2b), 24-bit address registers are used only for loading and storing data in bulk between the data register file and main memory. 32-bit ports from the register bank to both X and Y memory allow up to 2 registers from each bank to be transferred simultaneously using a single address register for each bank. Once the data is loaded into the register bank, it can be accessed indirectly by means of 7-bit index registers. The 7-bit data index generators (DIG) give much faster updates at a much lower power cost than their 24-bit counterparts. Also, a multi-ported register file is significantly less complex and consumes substantially less power than a multi-ported cache memory, particularly if the cache is an associative design. The choice of 128-word register banks allows a single 32-bit instruction to set the value of four index registers, with 4 bits to encode the instruction.

The use of index registers to access data also allows more efficient use of configuration memory: rather than storing direct register selections for each different algorithm to be executed, it is possible to use indirect references via index registers. If each algorithm is designed to use the same index registers, then the same configuration memory entry can

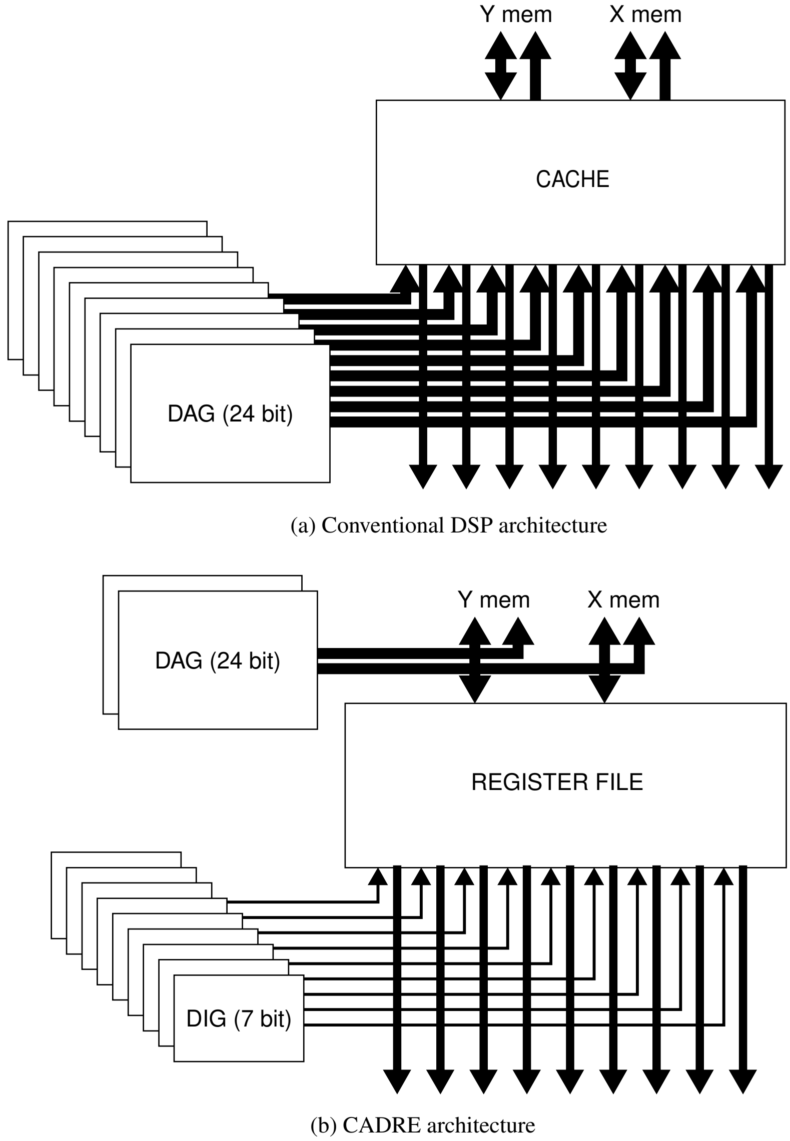


Figure 3.2 Reducing address generation and data access cost with a register file

be used for all of the algorithms with the index registers set in advance to point at the correct data. CADRE contains 8 index registers named i0-i3 and j0-j3.

The use of a register file gives CADRE a reasonably simple RISC-like structure, as shown in Figure 3.3. This leads to a very simple programmer's model: the data need only be loaded into the register bank before it is accessible to all of the functional units. This also improves the locality of communications, as most of the pathways on the processor can be made quite short. CADRE is far closer to a conventional programmable processor architecture than, for example, the Pleiades configurable signal-processing architecture, which is formed by a heterogeneous collection of semi-autonomous functional units and memories connected by a central communication network [134] and so is more reminiscent of an ASIC.

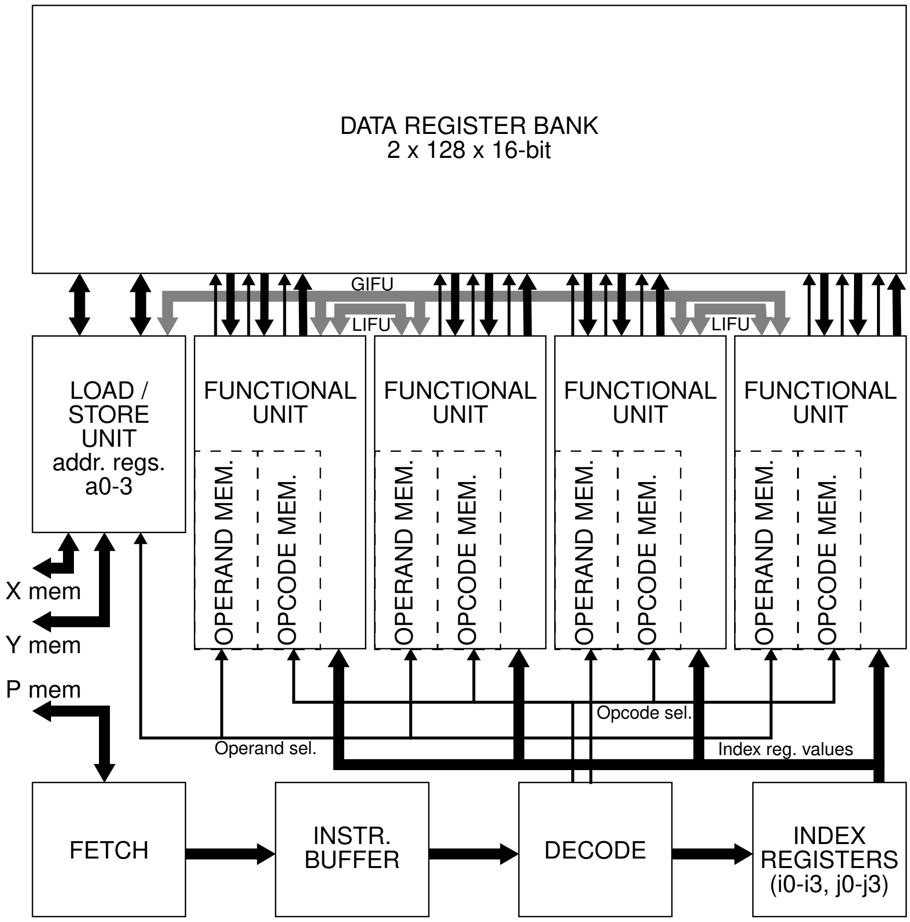


Figure 3.3 Top level architecture of CADRE

3.3.4 Instruction buffering

Most DSPs include some form of hardware loop instruction, allowing an algorithm to be executed a fixed number of times without introducing branch dependencies. In the CADRE architecture, this function is managed by a 32 entry instruction buffer, which also manages the loop count meaning that subsequent stages see an entirely flat instruction stream, and supports up to 16 nested loops. The highly compressed instructions mean that even fairly complex DSP kernel routines can fit within this space, and can be executed without the need to access main memory. A study of the Motorola M-Core ISA found that main I-cache references could be reduced by about 38% through the use of a 32-entry loop cache, with little benefit being obtained by using more than 32 entries [135]. A study of the Hitachi HX24E DSP [136] showed that power consumption could be reduced by between 25% and 30% by employing a 64 entry instruction buffer: this was sufficiently large for simple algorithms, but not for example a FFT. The compressed instructions for CADRE allow more complex algorithms to be stored, despite the use of a smaller buffer. The use of an instruction buffer to reduce power consumption has also been adopted for the new Texas Instruments TMS320C55x processors.

Apart from the looping behaviour, the buffer acts as a FIFO ring-buffer to store prefetched instructions, meaning that the next set of instructions can be prepared while either executing the current algorithm or when waiting for new data to arrive. The combination of the large register file and the compressed instruction buffer can greatly reduce the number of memory accesses, as is demonstrated by the results in section 9.3.3 on page page 202.

3.4 Instruction encoding and execution control

In keeping with a RISC-like philosophy, the instructions for the DSP all consist of 32 bit words. Instructions are split into two classes: compressed parallel instructions, or all other control and setup instructions. Control and setup instructions are responsible for tasks such as setting up index and address register values and initializing loops, after which the processing work can be done by the compressed parallel instructions without disturbance. A full description of the instructions for the processor can be found in Appendix B.

Compressed parallel instructions are described by a 32 bit instruction which maps onto a 320 bit long instruction word, stored in 10 separate 128 x 32-bit configuration memories, as shown in Figure 3.4.

Within each functional unit are two separate 32 bit configuration memories, the opcode and operand memories. The configuration words from opcode memory set up the sequence of operations to be performed by the ALU, which can consist of any combination of:

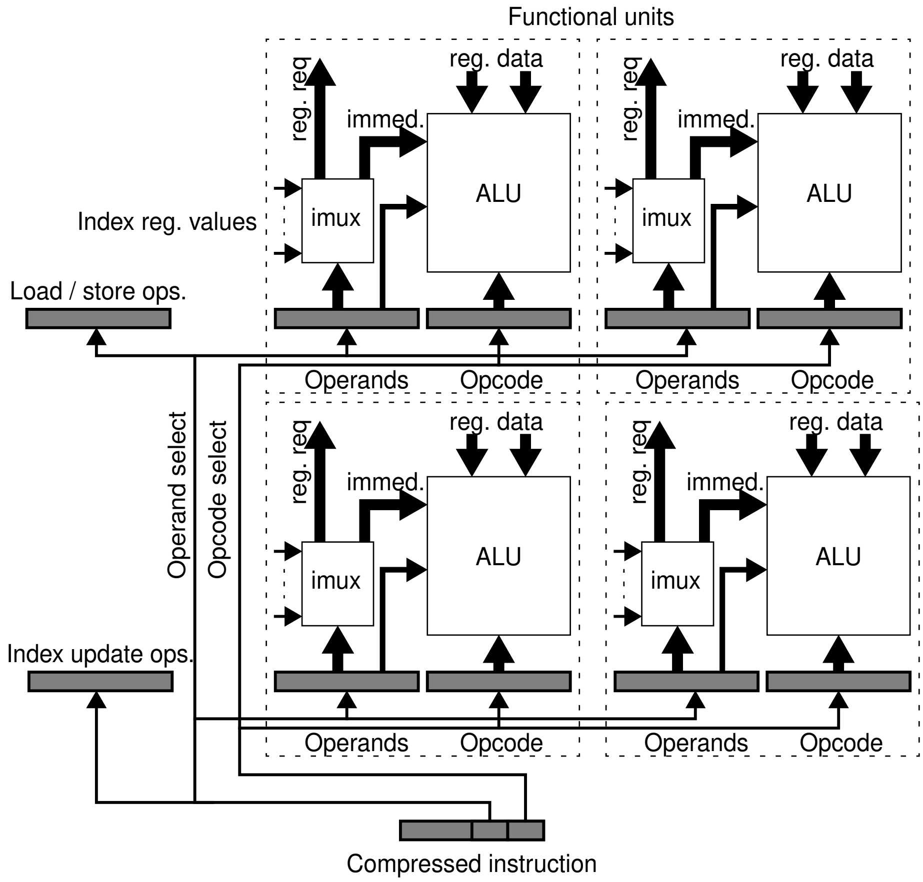


Figure 3.4 Parallel instruction expansion

- An ALU operation (with the result being written to the ALU accumulators).
- A parallel move to the ALU accumulators.
- A writeback from the accumulators to the register bank.

Also, the opcode configuration word is responsible for setting up additional functions such as driving of the GIFU / LIFU.

The configuration words from the operand memory specify the source of the data for the operations in the ALU, the destinations for the operations, and the target register of any writeback. The source data for operations are selected by the input multiplexer (imux), and can be either an indirect reference to the register file (using one of the eight index registers), a direct reference to the register file, or an immediate value stored in the operand memory.

The remaining two configuration memories are located outside the functional units. The first of these holds details of how the index registers are to be updated. The second specifies load or store operations to be performed in parallel with the arithmetic operations, and includes details of the address registers to be used to access memory, how the address registers are to be updated, and which register locations are to be used (specified either directly, or indirectly using an index register value).

Bit position	**Function**
0-6	Opcode config. memory address
7-13	Operand / load-store / index config. memory address
14	Enable for load/store operations
15	Global enable of writes to accumulators
16	Global enable of writebacks
17	Enable index register updates
18-22	Condition code bits
23-26	Enable operations in functional unit 1-4
27-30	Select conditional operation in functional unit 1-4
31	0 - indicates a parallel instruction

Table 3.4: Parallel instruction encoding

Compressed parallel instructions are indicated by means of a zero in the most significant bit position, so that they can be rapidly identified. The instruction format is shown in Table 3.4. Each 32 bit parallel instruction contains two 7-bit fields to select the configuration memory entries required for the operation: bits 0-6 select the opcode configuration memory word to be used, while bits 7-13 select the operand memory word to be used and also which load/store and index update operations are to be performed. Splitting the configuration memory in this way allows the maximum amount of reuse for configuration memory locations; for example, many algorithms may require four parallel multiply-accumulate operations, but may require different patterns of register accesses.

To provide even more flexibility in operation, and to reduce configuration memory requirements still further, it is possible to disable components of the stored parallel operation selectively from within the compressed instruction word. This allows each configuration memory location to specify the maximum number of possible concurrent operations, avoiding redundancy of storage, and each algorithm can then select only those parallel components required at the time. Bits 14-17 respectively of the compressed instruction are master enables for the load / store operations, writes to the accumulators, writebacks to the register bank and updates to the index registers; and bits 23-26 enable or disable arithmetic operations in each of the functional units.

A demonstration of the benefit that can be obtained by allowing portions of a parallel operation to be disabled is given in Figure 3.5. This shows an algorithm with a number of serial operations: the input to the algorithm is processed by operation 1, the result of operation 1 is processed by operation 2, the result of 2 by 3, the result of 3 by 4, and the result of operation 4 is written back to the register file. This type of algorithm can be mapped onto a parallel structure by the use of software pipelining. In the first instruction, the first input word is loaded from memory. Then, this is processed by operation 1 while the next input word is loaded from memory. Operation 2 then processes the result of operation 1, while operation 1 processes the previously fetched input word and the third input word is loaded from memory. This develops, with each of the operations processing data from the previous sample in the sequence, until the software pipeline is operating fully (within the DO loop), and processing is occurring simultaneously in all of the functional units. Finally, when all of the data has been fetched from memory, the last data word empties out of the software pipeline and is finally written back. The ability to enable and disable portions of the parallel operation means that the whole algorithm can be encoded using a single configuration word in operand and opcode memories; which encodes the instruction for the fully-developed pipeline. All of the other instructions can be created by disabling certain portions of that instruction, without the need to store additional instructions containing partial NOPs in the configuration memory.

Arithmetic operations in each of the functional units can also be made conditional, using bits 27-30. Each functional unit maintains an internal condition code register, and the state of this can be tested against the condition code provided in the instruction. Conditional execution reduces the need for branch instructions, which disrupt normal pipeline operation unless expensive branch prediction is used.

```
{
      load
}
{
      operation 1

      load
}
{
      operation 1
      operation 2

      load
}
{
      operation 1
      operation 2
      operation 3

      load
}
{
      operation 1
      operation 2
      operation 3
      operation 4

      load
}


do #count
      {
            operation 1
            operation 2
            operation 3
            operation 4

            writeback

            load
      }
enddo
```

```
{
      operation 1
      operation 2
      operation 3
      operation 4

      writeback
}
{
      operation 2
      operation 3
      operation 4

      writeback
}
{
      operation 3
      operation 4

      writeback
}
{
      operation 4

      writeback
}{
      writeback
}
```

Figure 3.5 An algorithm requiring a single configuration memory entry

A further form of conditional execution is provided, beyond testing of the condition codes within the functional units, which is intended to improve the regularity and reduce the size of software-pipelined code. As shown in Figure 3.5 and Figure 3.6, additional code is required before and after the main loop to set up and empty the software pipeline. The use of *loop conditional* instructions allows some of the pre- and post-loop code to be merged into the loop. Loads and stores, arithmetic operations, and writebacks can all be made conditional on whether the processor is executing the first or last instruction in the loop. For the example of Figure 3.6, use of these loop conditionals gives the new code two less instructions outside of the loop body, as shown in Figure 3.6b.

```
{
      load
}
{
      operations (1)
}
do #count
      {
            operations (2)

            load
      }
      {
            operations (1)

            writeback
      }
enddo
{
      operations (2)
}
{
      writeback
}
```

(a) Without loop conditionality

```
{
      load
}
do #count+1
      {
            operations (1)

            writeback nfirst
      }
      {
            operations (2)

            load nlast
      }
enddo
{
      writeback
}
```

(b) With loop conditionality

Figure 3.6 Using loop conditionals to reduce pre- and post-loop code

3.4.1 Interrupt support

DSP pipelines are traditionally optimised for repeated execution of small DSP kernel routines, and are less efficient at executing control-oriented code. However, most manufacturers add extra hardware to their designs, such as branch prediction, speculative execution, complex interrupt structures and support for exact exceptions, to improve the control performance and allow the processor to be used as a stand-alone device. CADRE is intended to operate in conjunction with a microprocessor, and so a considerable amount of this hardware can be eliminated by allowing the microprocessor to handle control tasks and for the DSP to operate in the role of a coprocessor. Obviating the requirement for additional hardware, through the proper allocation of tasks between the two devices in this application, contributes to lowering the overall power consumption. The microprocessor prepares tasks for the DSP, and instructs it to perform them through a simple interrupt structure which also allows for synchronisation with data. Under normal circumstances, the DSP will only respond to an interrupt when halted, i.e. when it has completed the current task. This allows the processor state to be managed without the need for exact exceptions. If necessary, the host microprocessor can issue a non-maskable interrupt, which will cause the DSP to respond immediately at the expense of losing the current processor state. Situations where non-maskable interrupts would be issued are cases when the processor has failed to complete the current task in the time available, or when an urgent event must be tended to, and so it is acceptable to discard the data and either repeat the operation later or not as required by the application.

3.4.2 DSP pipeline structure

A block-level representation of the DSP pipeline is shown in Figure 3.7. The fetch stage autonomously fetches instructions from program memory, from where they are passed on to the instruction buffer stage. From here, the instructions pass on to the decode stage, where the most-significant bit is examined to separate them into compressed parallel operations and control / setup instructions. Control and setup instructions are decoded and executed without further pipelining, to minimise setup latency. However, to avoid conflicts if the resources to be accessed lie in a downstream pipeline stage it is necessary for intermediate stages to first become free.

If a compressed parallel instruction is detected, then a read is initiated in the operand configuration memories, index update memory (within the decode block) and load/store memory (within the load/store unit). Within the load/store unit, the appropriate address registers are selected and are updated appropriately.

The next stage of operation is for each functional unit and the load/store unit to capture those index register values which are required for indirect references to the data registers, and for the index register values to be updated according to the current instruction.

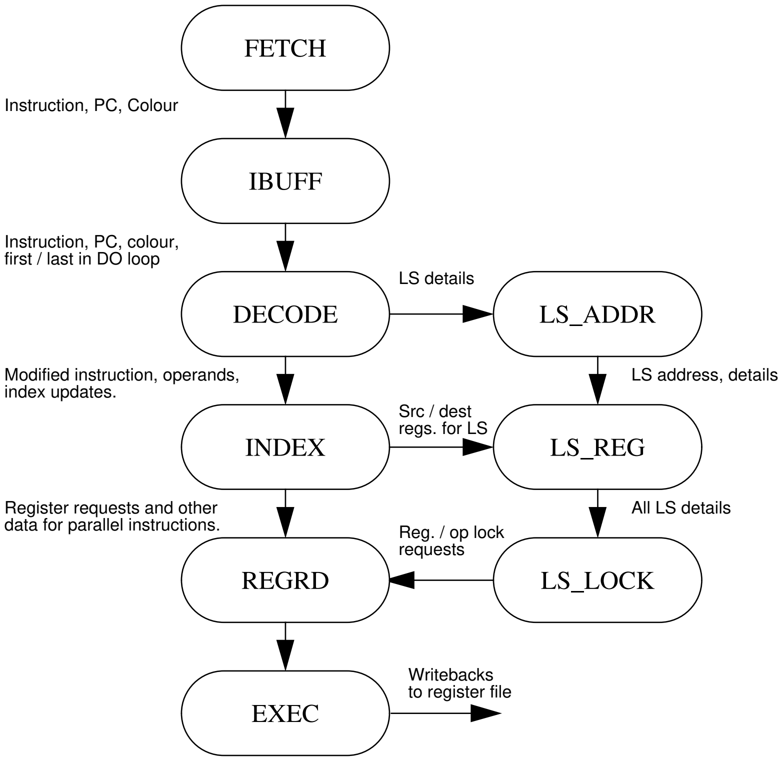


Figure 3.7 CADRE pipeline structure

Once the register sources are known, each functional unit requests the specified data from the register bank. While the registers are being read, the opcode configuration memories are read to set up the operations to be performed in each functional unit, and any parallel load or store operation is initiated through communication with the register bank. The load or store operations are then free to complete autonomously, with a locking mechanism preventing reads from registers that are the target for pending loads. Should an attempt be made to initiate another load or store operation while one is still pending, the pipeline stalls until the load or store completes.

After the register and configuration reads have completed, both the data and setup information is valid. At this point, the requested arithmetic operations take place in the functional units along with the associated parallel moves and writebacks to the register file.

The pipeline is somewhat bottom-heavy: that is, multiply-accumulate instructions in the functional units of the EXEC stage are likely to require significantly more time than the operations in the earlier stages. However, in an asynchronous system this proves to be beneficial for keeping the multiply-accumulate stages fully occupied. Setup instructions, such as changes to the index registers or DO loop setups, may be interleaved between parallel arithmetic operations so that the following parallel instructions will ‘catch up’ with the preceding parallel instructions. Considerable amounts of time are also left for driving signals such as the index registers across the whole chip, making the architecture more robust to process shrinks.

3.5 Summary of design techniques

A broad selection of the low-power design techniques used in chapter 2 have been employed in the architecture for CADRE. At the core of the design, architecture-driven voltage scaling using four functional units allows a given workload to be performed with the minimum supply voltage. This also relaxes timing requirements on each stage somewhat, allowing pipeline latches to be operated in normally-closed mode to block glitches.

Configuration memories within the functional units allow very complex operations to be distributed efficiently over the parallel resources, without fetching excessive amounts of information from the main program memory. The instruction buffer reduces the amount of memory activity still further. By reducing the distance over which data must travel, the amount of data required and the size of the memory from which the data must be fetched, the total switched capacitance per instruction is minimised.

A large register file allows data to be reused, again minimising switched capacitance by reducing the average distance over which data must be transmitted and the size of the memory accessed. Using index registers to access the data in the register bank reduces the power consumption of address generation.

Sign-magnitude numbering is employed in the data processing elements, to exploit the typical characteristics of data in DSP applications and reduce the overall switching activity both within the functional units and on buses throughout the system. Finally, the role of CADRE alongside a host microcontroller allows the control functions of the DSP to be kept to a minimum, simplifying the processor design.

To be truly effective, design for low power must consider all levels of design. The high level architectural features discussed in this chapter set the framework for a low power design. This must be complemented by the lower level techniques such as correct choice of circuit structure, optimisation of transistor sizing and layout. A 0.35μm CMOS process was the most advanced technology available when carrying out the work presented in this thesis, but the techniques are applicable to smaller scale processes and advanced technologies such as SOI.

Chapter 4: Design flow

4.1 Design style

The asynchronous design style chosen for CADRE is based on 4-phase micropipelines [90], with bundled data. This style has been chosen as it gives simpler circuits with lower power consumption than delay-insensitive asynchronous designs, at the cost of greater design effort in matching delays in control paths with the delays in the datapath. The broadish data validity scheme is used for most interfaces, except where specific circuits require broad protocol validity for their inputs.

The circuits of the processor are divided into two classes: asynchronous control circuits and datapath circuits. The control circuits implement the interfaces between different stages and control the operation of the datapath. The interfaces and the operation of the control circuits are specified using signal transition graphs, with hazard-free implementations produced by using the *Petrify* asynchronous circuit synthesis tool [117]. Datapath circuits consist of conventional processing logic, multiplexers, latches, etc.

4.2 High-level behavioural modelling

Before beginning the circuit design of a complicated device such as a processor, it is desirable to have an abstract high-level model of its operations to test the architecture and as a reference against which to verify correct operation. The LARD language [138] facilitates the modelling of complex asynchronous systems, with in-built support for asynchronous communication channels. However, due to the short time available for the design of CADRE, it was felt that there was insufficient time to develop a complete separate model. Instead, a compromise was made whereby the modelling process was integrated with the general design of the processor.

4.2.1 Modelling environment

From the outset, it was intended that the Synopsys Timemill and Powermill simulation tools would be used to perform simulations of the design from the schematic entry stage onwards. These simulation tools offer SPICE-like accuracy, but at a fraction of the computational load for large designs. A standard component of these tools is the Analog/ Digital Functional Model Interface (ADFMI), which allows the designer to produce behavioural models in the form of C language functions. Previously, this feature had been used to produce test environments for circuits; it was now decided to use the modelling features of ADFMI to support the design of the processor. The advantage of this technique is that the same simulation environment and set of tests can be maintained while the design is hierarchically refined, with circuit blocks being replaced by functional models at whatever level of complexity is appropriate. This allows the operation of the processor to be studied and conclusions drawn at whatever actual stage of development has been reached. Also, a particular part of the design can be tested in its place, with the rest of the circuits operating in the form of models to reduce simulation time. As a final aid to the design process, functional model blocks can be made to report the state of various parts of the design (such as register contents, memory contents, etc.) to log files. Graphical displays of this data can be made either in real time as the simulation is occurring, or played back later, as an aid to debugging both the design and the test programs being run on the design. The presented approach is valid for many other simulation systems that allow co-simulation of circuits with behavioural modelling languages like Verilog or VHDL.

For the design of CADRE, blocks with complex functions (such as the instruction buffer, register file, configuration memories, index units and functional units) were initially modelled as whole units implementing both the asynchronous interfaces and the processing functions. Simpler elements, such as the fetch unit or instruction decode unit were modelled at a lower level, with datapath elements and asynchronous control circuits represented by separate models. Some trivial datapath functions were implemented directly with circuits, when the effort of producing a C model would have been disproportionately large.

Once confidence had been obtained in the operation of the design at the highest level of abstraction, it was then possible to refine the design by specifying the datapath and control elements of the more complex units. Simulation could then be performed again, with C models for the new lower levels of hierarchy. Finally, once the design was completely in the form of C models for asynchronous control circuits and datapath elements, it was possible to progressively substitute actual circuits in place of the C models. This was done for the datapath circuits first, so that any unexpected difficulties or late changes in implementation could be catered for simply, before the control circuits were synthesised. The final stage in the design flow would be to incorporate back-annotated layout information into the simulations as physical layout progressed, although it has not been possible to reach this stage in this work. The overall design flow is represented in Figure 4.1.

4.2.2 Datapath model design

The datapath elements of the processor were the simplest to model, as the only requirement was to generate the appropriate logical or arithmetic function in response to the signals on its control inputs.

A Perl script was produced to automate the production of the C models. This script takes as an input the schematic block representing the circuit's inputs and outputs, and produces a skeleton C model implementation with the input and output signals defined. The delays for driving the output signals are defined in a header file which contains delays for the entire design, and to add further rigour to the testing the delays are generated with a user-definable random element.

4.2.3 Control model design

The asynchronous control circuits were specified using signal transition graphs (STGs). These give a complete description of the essential behaviour of the circuit and, rather than manually produce a C functional model that would implement this behaviour, it was seen that a model could be automatically produced from the specification relatively easily, using an extension of the technique used to produce skeletal datapath models.

The Perl script used to generate the skeletal models was modified to process STGs, in the same format as that accepted by the Petrify tool, along with the schematic design file. The only constraint was for the input and output signals on the schematic to have the same

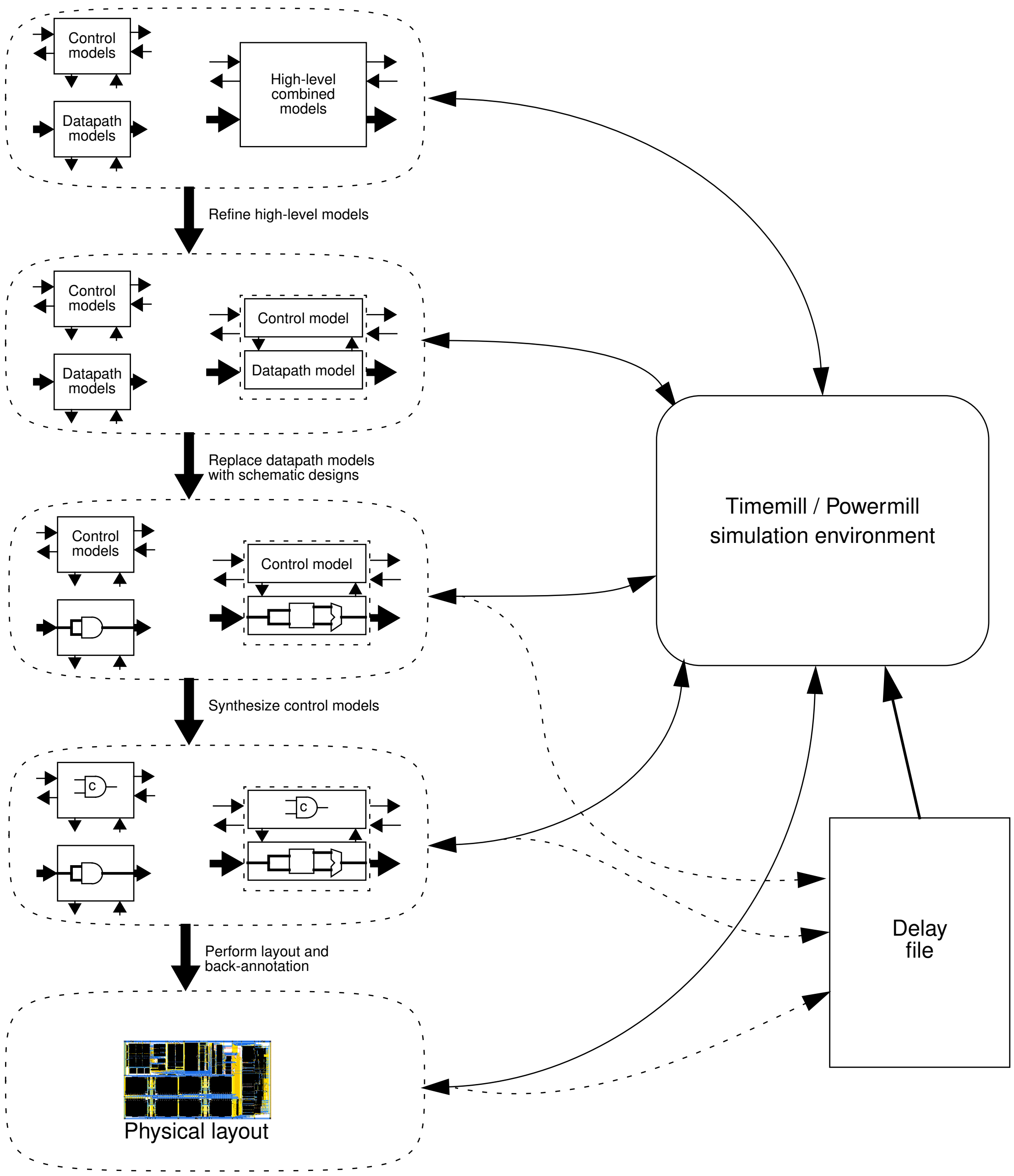


Figure 4.1 STG / C-model based design flow for the CADRE processor

names as the signals in the STG specification, although it would be possible to modify the script to prompt the user where uncertainty in the names existed.

The technique used to emulate the STG operation is very straightforward. An example of a simple schematic block and its STG specification is given in Figure 4.2. Each arc in the STG between two transitions represents a place where a token can reside, and the initial state of the system is with a token in the labelled place P0. The Perl script analyses the STG and generates the state structure shown in Figure 4.3, which contains boolean variables for each of these places indicating whether they contain a token or not. This state structure also contains a number of other variables necessary to deal with the operation of internal and output signals.

The core of the functional model is the so-called *evaluation function*, which is called by the simulator every time an input to the modelled circuit changes, or when an event is scheduled by the model itself. The basic structure of this function is given in Figure 4.4. A certain amount of setup and reset code is omitted for the purpose of illustration. At the heart of the evaluation function is a loop in which all of the transitions are checked in turn. A loop is needed to ensure that all output or internal transitions following an input transition are triggered correctly.

Examples of code to check input, output and internal transitions are given in Figure 4.5. In each case, all of the places leading into the transition are checked for tokens. Only those transitions for which all the tokens are present are then processed further.

For the case of an input transition, the state of the corresponding signal is tested. If the signal has undergone the appropriate transition, then the tokens are taken from the input places and placed into the output transitions. The ‘active’ flag is also set to cause the evaluation loop to be repeated.

An output transition consists of two subsections of code occurring during different calls to the evaluation function. In the first part, the tokens are taken from the input places and the output signal is set to the appropriate state. To ensure that any internal transitions following the output transition occur in the proper order, an event is scheduled to occur after the output delay of the signal (which is set by the model). A flag is set in the state structure to indicate that this transition has fired, and the time at which the transition is to complete is stored. When the scheduled time is reached, the evaluation function is called again and the second part of the code is executed. This puts the tokens in the output places of the transition, and sets the ‘active’ flag. Internal transitions are handled in a very similar way to output transitions except that no signal needs to be set by the model: the internal signal remains entirely abstract. The size of the delays for internal and output transitions are all stored in the common delay file used for all models.

This allows a rapid exploration of the possible options for the control circuit design, as changes to the STG can be implemented with ease. The Petrify asynchronous circuit synthesis tool has limited practical ability to synthesise full circuits. Instead, logic equations for the various signals are produced and it is necessary to map these manually onto the available standard cells. To go through this task every time a change is made to the STG is very laborious. Instead, functional models can be generated directly from the specification and synthesized and mapped onto the available technology once the specification has become stable (as long as care is taken to ensure that the STG has consistent state coding). The 'Visual STG lab' software package was used to enter the STGs, giving an intuitive graphical interface to input the data.

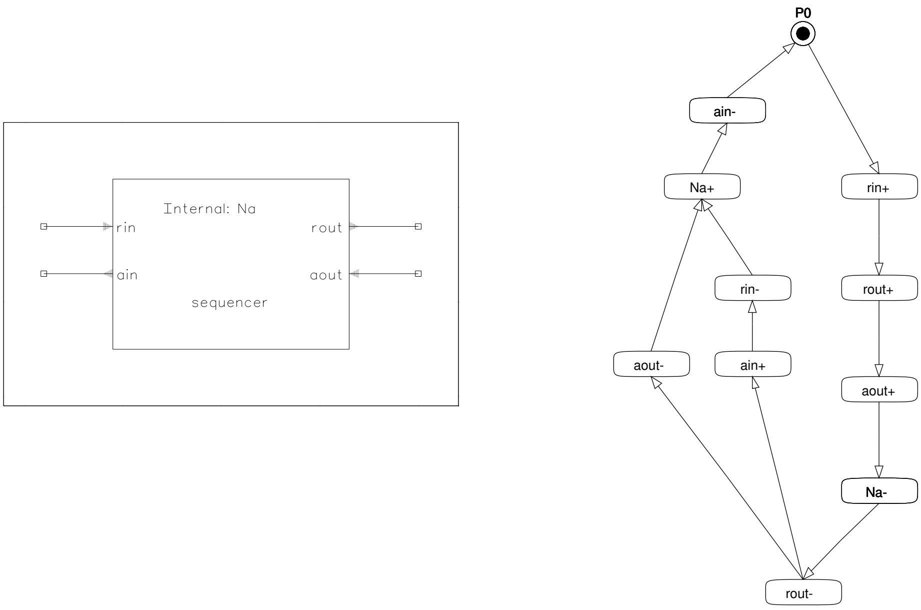


Figure 4.2 A simple sequencer and its STG specification

4.2.4 Combined model design

Combined models contain a mixture of asynchronous interfaces and datapath logic and were designed using the same Perl script that produced pure control circuits. Signal transition graphs specify the operation of the asynchronous interfaces, and the data

```
struct state_struct
{
/* State struct members generated from STG */

/* Places (implicit and named) */
    char ain_pl_to_rin_mi, P0, rin_pl_to_rout_pl, Na_mi_to_rout_mi;
    char rout_mi_to_aout_mi, aout_mi_to_Na_pl, rout_pl_to_aout_pl;
    char rin_mi_to_Na_pl, Na_pl_to_ain_mi, aout_pl_to_Na_mi;
    char rout_mi_to_ain_pl;

/* Output and internal delay time storage */
    FMTIME rout_t, ain_t, Na_t;

/* Output and internal delay wait flags */
    char ain_pl_w, ain_mi_w, rout_pl_w, rout_mi_w, Na_pl_w, Na_mi_w;
};
```

Figure 4.3 State structure indicating STG token positions

```
void sequencer_eval()
{
   struct state_struct *state;
   int rin_id;
   int aout_id;
   int ain_id;
   int rout_id;
   int active = 1;

   rin_id = fmGetPortId("rin");
   aout_id = fmGetPortId("aout");
   ain_id = fmGetPortId("ain");
   rout_id = fmGetPortId("rout");

   while (active) {
      active = 0;
      /* ... check all transitions */
   }
}
```

Figure 4.4 Evaluation function body

processing functions of the model can then be implemented within the framework produced by the asynchronous interfaces. Delays for internal processing can be incorporated by adding dummy internal transitions, for which the delays can either be specified in the delay file or made data-dependent.

The overall experience of using this method to produce asynchronous control circuits and combined models was very positive. Going directly from specifications to behavioural models gives a rapid way of developing, testing and modifying complex asynchronous specifications in situ. The method could be enhanced relatively easily by adding

```
/* Evaluation for transition rin+ on rin */
if (state->P0) {
   /* Input */
   if (fmGetPortStateById(rin_id) == ONE) {
      active = 1;
      state->P0 = 0;
      state->rin_pl_to_rout_pl = 1;
   }
}

/* Evaluation for transition rout+ on rout */
if (state->rin_pl_to_rout_pl) {
   /* Output */
   fmSetPortStateById(rout_id, ONE);
   state->rout_t = fmCurrentTime() + SEQUENCER_ROUT_PL_DEL / 100.0;
   fmScheduleEvent(fmevalelement, state->rout_t, 0, 0);
   state->rout_pl_w = 1;
   state->rin_pl_to_rout_pl = 0;
}
if (state->rout_pl_w && tcmp(fmCurrentTime(),state->rout_t)) {
   active = 1;
   state->rout_pl_w = 0;
   state->rout_pl_to_aout_pl = 1;
}

/* Evaluation for transition Na- on Na */
if (state->aout_pl_to_Na_mi) {
   /* Internal */
   state->Na_t = fmCurrentTime() + SEQUENCER_NA_MI_DEL / 100.0;
   fmScheduleEvent(fmevalelement, state->Na_t, 0, 0);
   state->Na_mi_w = 1;
   state->aout_pl_to_Na_mi = 0;
}
if (state->Na_mi_w && tcmp(fmCurrentTime(),state->Na_t)) {
   active = 1;
   state->Na_mi_w = 0;
   state->Na_mi_to_rout_mi = 1;
}
```

Figure 4.5 Evaluation code for input, output and internal transitions

automatic checking of the specifications: currently, if the environment produces incorrect transitions on signals, they will simply be ignored by the model. It would be relatively easy to add extra code to the models to report errors, and combined models could also use these methods to check bundling constraints on input interfaces. Also, there is no intelligence used to determine the initial states of output signals: currently, they are reset to zero by default, unless their names start with ‘N’ to indicate an active-low signal in which case they are reset to one. Tracing of token flows around the STG could be used to determine the correct conditions automatically.

4.2.5 Integration of simulation and design environment

The final part of automating the design flow was the Perl 'glue' script that enables C models to be substituted automatically for schematic components in simulations where desired. This was integrated with the design environment by adding an attribute tag named 'type' to instances of subcircuits. The Perl script analyses the netlist for the design, searching for this attribute tag. Where the attribute has its value set to 'cmodel', the part of the netlist defining that subcircuit is removed, and replaced with a reference to the C functional model with the same name. Once the netlist has been processed, the Perl script generates a final C function that registers all of the functional models with the simulator on start-up and produces a shell script that invokes the simulator in the correct manner.

4.3 Circuit design

The design was performed using a 0.35µm 3-metal CMOS process, although the design rules for this process are intended to be transferred easily onto other technologies. The majority of the design was performed using the library of standard cells available, which includes a wide range of the Muller C gates that are used in the design of asynchronous control circuits, and other key asynchronous circuit elements such as arbiters [139]. An arbiter allows choice to be made safely between two separate asynchronous events, and consists of a flip-flop followed by a filter circuit to prevent an output being generated until any metastability in the flip-flop has been resolved. Full-custom design was used for large regular structures such as the instruction buffer storage elements, the configuration memories, the register file and the datapath components of the functional units. To reduce design time, components from the AMULET3 processor were reused when it was possible to do so, albeit often in a modified form.

4.4 Assembler design

To be able to produce test programs quickly and easily, it was necessary to write an assembler for CADRE. For a conventional processor, this would be a trivial task. However, the compressed parallel instructions supported by CADRE make the task rather more difficult. An example of the assembly language designed for CADRE is shown in

Figure 4.6, which is a very simple vector product program. Curly braces are used to indicate parallel instructions.

```
org 0

ipdata equ 0x0000
count equ 512

; Set up address register
; to point to input data
move #ipdata,r0
move #2,nr0
move #-1,mr0

; Load the first data and clear
; the destination accumulators
{
   move #L0,maca:a
   move #L0,macb:a
   move #L0,macc:a
   move #L0,macd:a

   loadl x:(r0),x:0
   loadl y:(r0)+nr0,y:0
}

; Set up a DO loop to process the data
do #count
   ; Main processing function
   ; calculates the squared magnitude
   {
      mac x:0,x:0,maca:a,maca:a
      mac x:1,x:1,macb:a,macb:a
      mac y:0,y:0,maca:a,maca:a
      mac y:1,y:1,macb:a,macb:a

      loadl nlast x:(r0),x:0
      loadl nlast y:(r0)+nr0,y:0
   }
enddo

; Add the running totals together
{
   ; Could use GIFU or LIFU
   add maca:a,macb:a,maca:a
   add macc:a,macd:a,macc:a
}
{
   ; Can only use GIFU
   add maca:a,macc:a,maca:a
}
halt #3
```

Figure 4.6 An example of assembly language for CADRE

To simplify the design of the assembler, it was split into two programs. The first program processes only the parallel instructions, produces configuration data for the processor and replaces the parallel instructions with the appropriate ‘exec’ commands to recall the stored instructions (refer to Appendix D on page 260 for details of the parallel instruction encoding). The second program is a conventional assembler, which converts the mnemonics to the binary instructions for the processor (refer to Appendix B on page 248 for details of the main instruction set).

The difficulty in producing an assembler for the parallel components of the code stems from the fact that there is usually more than one way of encoding each part of a parallel instruction. An example of this is the first summation of running totals in Figure 4.6: the summation of the totals in MAC A and MAC B can take place either using LIFU1 or the GIFU. Similarly, the summation of the totals in MAC C and MAC D can take place on LIFU2 or the GIFU: however, if the first instruction is encoded to use the GIFU, the second is constrained to use LIFU2. Also, as many parallel instructions as possible should be mapped to a given configuration memory location by the assembler.

To deal with this problem, a list of the possible encodings is generated at each stage. When a parallel instruction is entered, the list is empty. The first instruction in the summation example causes two different alternatives to be generated and placed in the list. On reaching the second instruction, the possible encodings for this are generated in a separate list. Each of these possible encodings is compared with all of the encodings in the main list, and all of the compatible combinations are stored and become the new running list. Once the end of a group of parallel instructions is reached, unused components of the parallel instructions are disabled in each of the stored encodings, and the appropriate 'exec' instruction for each encoding is generated. Figure 4.7 shows how the possible encodings for the choice of summation path in the example would be generated.

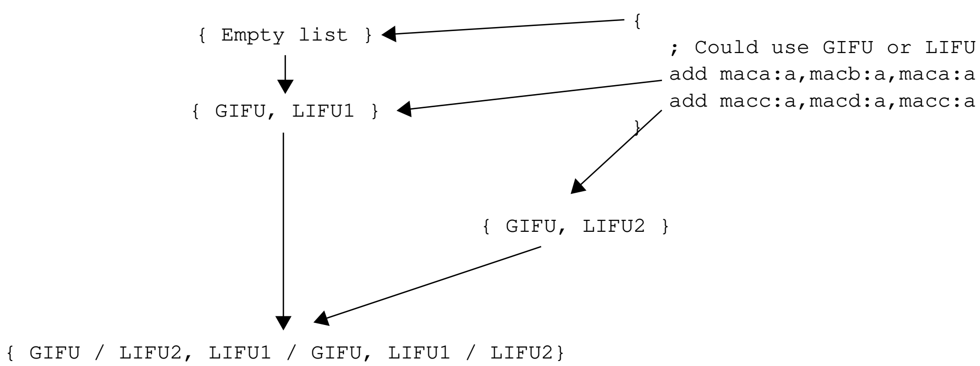


Figure 4.7 Different encodings for a parallel instruction

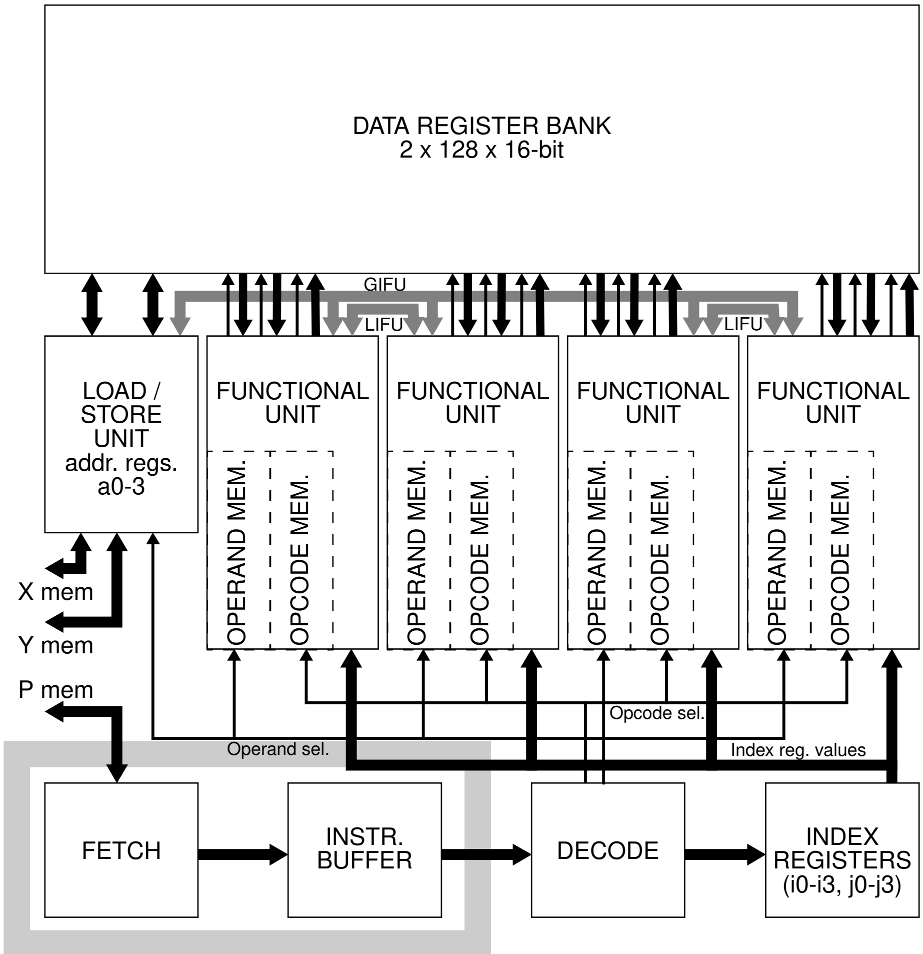
At the end of the input file, each parallel instruction in the code will be represented by a list of possible encodings. The final task is to go through the list of encodings to see which of them can be merged onto a single opcode or operand configuration memory location.

The first stage of processing attempts to reduce the number of possible options by discarding the least power efficient encodings. In the example of Figure 4.7, the encodings that use the GIFU drive a greater load than the encoding that uses LIFU1 and LIFU2, so only the third encoding will be kept.

Next, it is necessary to perform a search on the list of possible encodings of all of the instructions, to determine the minimum number of configuration memory locations that they can be stored in. An exhaustive search has exponentially increasing complexity, and was found to be impractical for all but trivial programs. Instead, the search is terminated for each instruction as soon as another instruction was found with which it can be combined. So far, no program has been found for which the quick search results are different to those gained by an exhaustive search (although some programs could not be assembled using the exhaustive search due to the required run time).

The assembler automates the encoding and compression of the parallel instructions. However, it is necessary for the designer to be aware of the compression process for it to be fully effective and to make consistent choices of, for example, index registers or functional units in the parallel instructions. It would be desirable to have a tool to assist in the programming that would allow abstraction in these choices. The programmer would then use a form of high-level language or a graphical representation, independent of many of the physical choices that restrict the compression of the instructions. Once the entire design has been entered, the tool could then make the appropriate decisions about how the algorithm would be mapped so as to minimize the configuration memory footprint.

Chapter 5: Instruction fetch and the instruction buffer



5.1 Instruction fetch unit

The instruction fetch unit is responsible for reading instructions from program memory, passing them to the instruction buffer and updating the program counter. It begins to operate autonomously as soon as reset is released. The only factor complicating the operation of the instruction fetch unit is the need to handle branch instructions. When a branch is executed in the decode stage of the pipeline, the fetch unit must stop fetching instructions from the current stream and change the program counter to the new value.

Since the operation of the decode stage is asynchronous with respect to the operation of the fetch stage, arbitration is necessary to decide when to stop fetching new instructions.

By the time the decision has been made to take a branch, it is likely that a number of instructions will have been fetched from the branch shadow. It is necessary to flush these instructions, and this is done by means of an instruction colouring mechanism. Each instruction fetched from memory has an additional ‘colour’ bit attached to it, indicating from which control stream the instruction originates. The decode stage analyses the colour bit of incoming instructions, and discards those whose colour does not match the current operating colour. Since no further branch instructions can be originated until the flush is complete, a single bit suffices.

As well as the instruction and its associated colour, the PC value must also be passed to the decode stage to allow PC-relative branches and to provide the return address for branches or jumps to subroutines. To simplify provision of the return address from subroutines, the PC value of the *next* instruction is sent.

5.1.1 Controller operation

Before each instruction is fetched, it is possible for the fetch operation to be interrupted by a branch request. Since the arrival of branch requests is asynchronous with respect to the fetch unit controller, arbitration is necessary to decide whether or not to go ahead with a fetch cycle. The mechanism by which arbitration takes place is shown in Figure 5.1. At the beginning of each cycle, the fetch unit controller attempts to begin a cycle by asserting *fetch_req*. This passes to the mutual exclusion element (mutex), which is based on an asynchronous arbiter. As long as *bra_req* has not arrived before *fetch_req* goes high, control is gained of the mutex and the fetch operation can proceed. At the end of the fetch operation, the mutex is released and *bra_grant* goes high to indicate that a branch request is pending. Should *bra_req* and *fetch_req* arrive simultaneously, the mutex element makes a decision regarding which one will be serviced.

At the beginning of a fetch cycle, a fetch request is issued to program memory system along with the PC value. At the same time, a request is issued to the PC incrementer block along with the current PC value. Once both the program memory and the PC incrementer

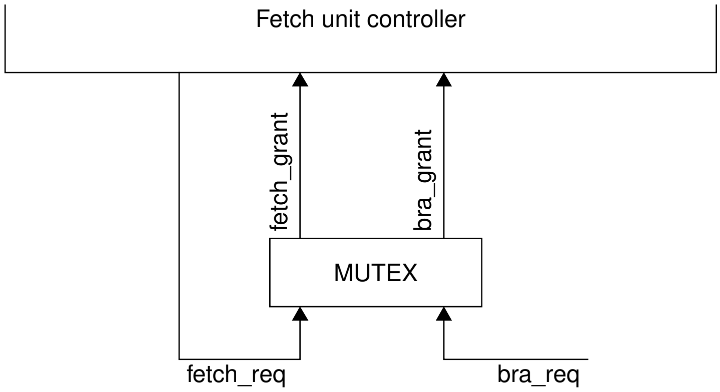


Figure 5.1 Fetch / branch arbitration

have completed their functions, the instruction word and incremented PC are captured and passed to the instruction buffer along with the current operating colour. A normally-closed latch at the output of the fetch unit prevents intermediate values from driving the moderately large load of the instruction buffer. Finally, the stored PC value is updated with the incremented PC value.

If a branch request is currently pending, the fetch cycle is locked out by the mutex as soon as a fetch cycle ends. Instead, the PC value is updated from the branch target address supplied by the decode stage, the instruction colour is toggled and an acknowledge is issued to the decode stage. Once the branch request is removed, the fetch unit may proceed to fetch instructions from the new address.

5.1.2 PC incrementer design

It is accepted that a ripple-carry adder is among the simplest, smallest and least power-hungry adder designs [106]. However, it is also one of the slowest in the worst case, due to the need to propagate the carry signal across the entire chain of full adders. For a synchronous system, it is necessary to either slow the entire system to meet the worst case speed of the ripple-carry adder or to use a faster but more complex and power-hungry adder design that resolves the carry more rapidly. In an asynchronous system it is possible to tolerate variations in completion time, and one can design the adder circuit to indicate completion to take advantage of the average case statistics of the data being processed.

For the case of an incrementer, the average case statistics are extremely favourable. Consider a random 24-bit input value: for there to be exactly one stage of carry propagation, the least significant two bits must be '01'. There are 2^{22} 24-bit values with these two bits at the bottom, and so the probability of this chain length is $2^{22}/2^{24} = 0.25$. For exactly two stages of carry, the least significant three bits must be '011'. There are 2^{21} such 24-bit values, so the probability of this chain length is $2^{21}/2^{24} = 0.125$.

The mean propagation length is given by

$$\bar{L} = \sum_{n=1}^{23} n \times P(L=n) \tag{14}$$

Substituting in the probabilities for each carry chain length gives

$$\bar{L} = \frac{1}{2^{24}}(2^{22} + 2 \times 2^{21} + 3 \times 2^{20} \ldots + 23 \times 1) \tag{15}$$

$$= \frac{1}{2^{24}}((2^{22} + 2^{21} \ldots + 1) + (2^{21} + 2^{20} \ldots + 1) \ldots + (2 + 1) + 1) \tag{16}$$

$$= \frac{1}{2^{24}}((2^{23} - 1) + (2^{22} - 1) \ldots + (2^{2} - 1) + (2^{1} - 1)) \tag{17}$$

$$= \frac{1}{2^{24}}((2^{23} + 2^{22} + 2^{21} \ldots + 2) - 23) = \frac{2^{24} - 25}{2^{24}} \approx 1 \tag{18}$$

It can be seen that for a general N bit number, the mean propagation length will be

$$\bar{L} = \frac{2^{N} - (N + 1)}{2^{N}} \tag{19}$$

Since the average carry propagation length will be approximately just one position, it is clear that data dependent operation has very favourable properties for an incrementer. Fully data-dependent asynchronous ripple carry adders have been designed, such as that assessed in [106], where the carry is evaluated using dual-rail dynamic logic. However, dynamic circuits are not ideal from a power viewpoint due to the precharge transitions. Also, a dynamic design cannot be made easily using standard cell logic, and completion detection requires a broad fan-in tree which adds delay.

A compromise that gives reduced data dependence but simpler circuits is speculative completion as proposed in [111]. Speculative completion uses a number of different delays to model the circuit. If pathological data cases are detected, the outputs of the shorter delays are disabled and an appropriately longer delay is used. For the case of an incrementer, the circuits required to detect the pathological cases are trivially simple.

The PC incrementer circuit is shown in Figure 5.2. The PC is analysed in 6 groups of 4 bits, looking for chains of ones using 4-input NAND gates. The chain of delays is tapped at positions appropriate for the length of each carry propagate chain, with each tap disabled by an active low *kill* signal. The first delay is sufficient for the *kill* signals to stabilise, and is smaller than the others as it can also incorporate the delay through the OR tree from each of the taps to *inc_done*. The delays are asymmetric, with falling edges to experiencing much less delay, which ensures that the delay chain is reset between cycles.

Delay group	Delay (*inc_go*+ to *inc_done*+)
d_3	1.0ns
d_7	2.2ns
d_{11}	3.3ns
d_{15}	4.5ns
d_{19}	5.6ns
d_{23}	7.0ns

Table 5.1: PC Incrementer delays

The delays were matched to that of the ripple carry incrementer by simulating the worst-case delay in each group, using the *Timemill* tool. Table 5.1 gives the total delays for each length of carry chain. Split into groups in this way, the expression for the average delay becomes

$$\bar{d} = \frac{15}{2^{24}}\left(2^{20}d_3 + 2^{16}d_7 + 2^{12}d_{11} + 2^8 d_{15} + 2^4 d_{19} + \frac{16}{15}d_{23}\right) \qquad (20)$$

which gives an average delay $\bar{d} = 1.1\,\text{ns}$. The average case delay is only marginally larger than the shortest possible delay, and even the maximum delay is only a small part of the 25ns available for the fetch stage.

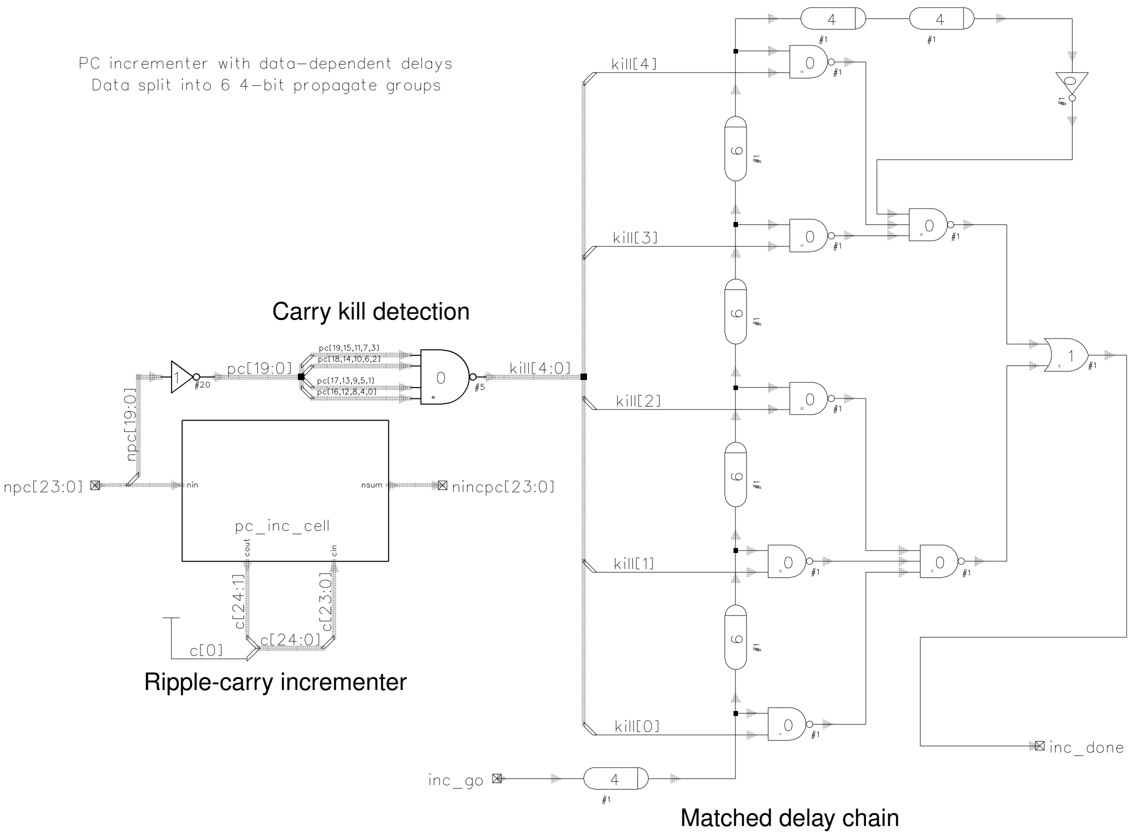


Figure 5.2 Data-dependent PC Incrementer circuit

5.2 Instruction buffer design

Most DSP architectures provide support for zero-overhead loops, where a DSP algorithm is executed a fixed number of times. In the instruction set for CADRE, these are performed by the 'DO' instruction. This instructs the DSP to execute the next *m* instructions *n* times, where *m* is a number from 1 to 32, and *n* is between 1 and 65536. DO loops can be exited prematurely by means of the conditional 'BREAK' instruction, whereby the current loop is exited at the end of the pass. Up to 16 DO instructions can be nested, by using an internal stack for the loop status.

The instruction buffer resides between the fetch unit and the decode stage, as shown in Figure 5.3. Under normal conditions, the instruction buffer simply acts as a 32-entry asynchronous FIFO between the fetch and decode stages. At the output of the instruction buffer, instructions are passed along with their associated colour and PC values to the decode unit, where the appropriate actions are then performed depending on the instruction (or the instruction is discarded, if the colour does not match the current operating colour). In most cases, this forward handshake between the instruction buffer and the decode stage is all that is required, and the first three stages of the pipeline operate in a strictly linear fashion. However, there are three exceptions to this: DO loop setup, BREAK instructions and branches.

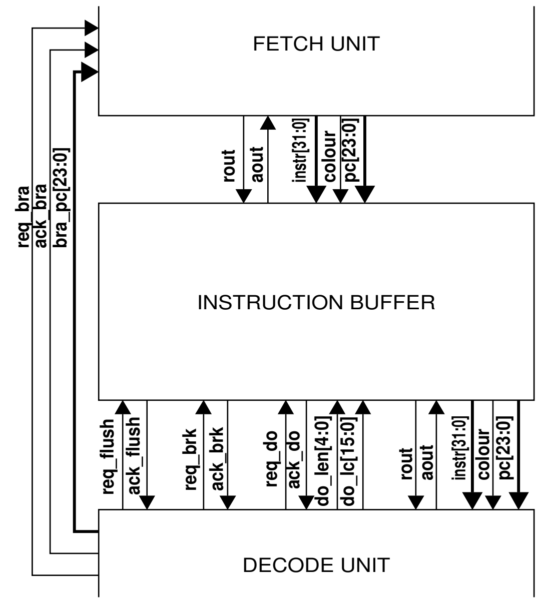


Figure 5.3 Adjacent pipeline stages and interfaces to the instruction buffer

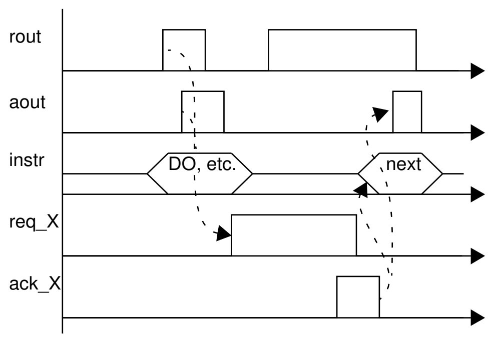


Figure 5.4 Signal timings for decode unit to instruction buffer communication

For these instructions, it is necessary for the decode unit to communicate back up the pipeline to the instruction buffer, with a reverse handshake on a separate request/

acknowledge pair. DO loops are set up by means of the *req_do*/*ack_do* signals and the bundled signals *do_len* (the number of instructions to be repeated) and *do_lc* (the number of repeats to be performed) The BREAK instruction causes the current loop to be exited at the end of the current pass, and this is done through *req_brk*/*ack_brk*. For the case of jumps and branches, it is necessary to exit any loops that are currently in progress, so that the new instruction stream can reach the instruction decode stage. This is done by means of the *req_flush* and *ack_flush* signals.

The basic sequence for each of these reverse handshakes is the same, and is shown in Figure 5.4. At some point after having latched a DO, BREAK or BRANCH instruction and having issued the acknowledge (*aout*), the decode unit sends the appropriate reverse request signal (*req_X*) back to the instruction buffer. The output stage of the instruction buffer will be asynchronously attempting to issue the next forward request (*rout*) during this time. However, this cannot be accepted by the decode unit as it is still occupied by the instruction that set up the reverse request. On receiving the reverse request signal, the instruction buffer performs the appropriate operation. It should be noted that the operation can cause the output of the instruction buffer to change. However, this deviation from the normal data bundling is acceptable as it is under the control of the reverse handshake, and the data is made stable before the reverse acknowledge issues from the instruction buffer back to the decode unit. The decode unit can then complete the instruction cycle, after which it can accept the forward request from the instruction buffer.

5.2.1 Word-slice FIFO structure

A micropipeline FIFO has the structure shown in Figure 5.5. When a data item arrives at the input, it propagates along the pipeline with each latch closing briefly to store the data until the next stage has acknowledged receipt. This design can have very good throughput, as the cycle time can notionally be reduced to that of a single stage. However, the input to output latency for an empty pipeline is poor as the data needs to pass through every latch. Power efficiency is also poor, as each latch and the associated controller performs an entire cycle when the data passes through it.

Many possible alternatives to the linear FIFO structure are possible, which can trade off complexity in the FIFO design against the length of path through which data must travel

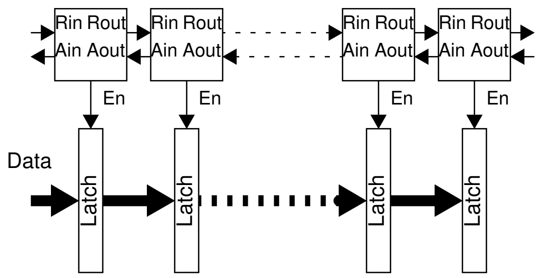


Figure 5.5 Micropipeline FIFO structure

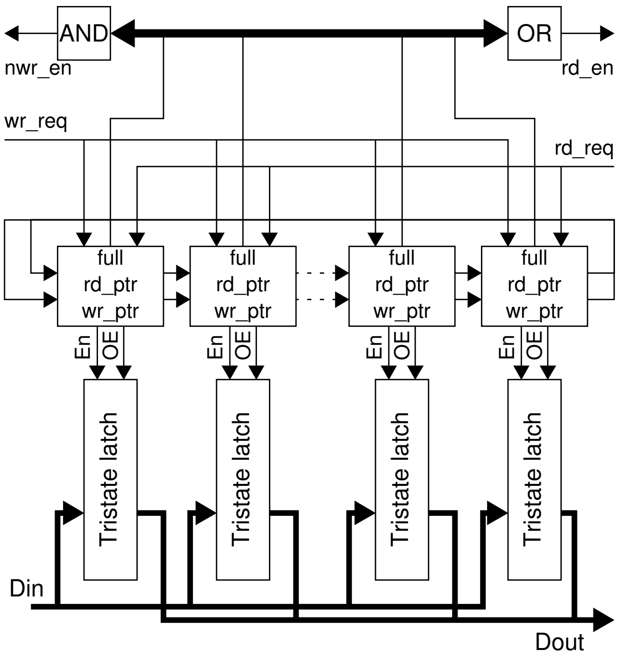


Figure 5.6 Word-slice FIFO structure

[140]. However, in order to implement the required looping behaviour easily the *word-slice* structure [141] was chosen. This is a ring-buffer like design, but has distributed rather than central control thus avoiding some of the problems of scalability associated with traditional ring buffer designs [88]. The basic structure is shown in Figure 5.6. The key difference between the word-slice design and the micropipeline design is that the word-slice FIFO has its latch rows in parallel rather than in series, with the outputs multiplexed by means of tri-state buffers. Each row of latches has an associated control element, which controls the write and output enables of the latches and records the current

state (full or empty) of the latch. The read and write position is controlled by means of tokens passed around the loop between these latch controllers. Output reads are enabled by an OR of the full indications from all of the latch rows (i.e. a read can be performed as long as there is data to read) and input writes are disabled by ANDing the full indications together. Stability of the AND and OR outputs is ensured by the use of matched delays within the write and read processes. The parallel nature of the structure means that there is only one latch delay between input and output when the FIFO is empty, lowering latency, and the power dissipation associated with the data passing through all of the latches is also eliminated [141].

5.2.2 Looping FIFO design

The operation of a standard word-slice FIFO can most easily be viewed in terms of tokens passing around a ring (Figure 5.7i). Each position in the ring buffer has a row of latches which are managed by a latch control unit. These control units have write and read request inputs and an output to indicate whether the stage is full or empty. Two separate overall control units communicate with all of the individual FIFO stages, to interface with input requests and to generate output requests.

When an input handshake occurs, the input handshake controller causes an event on the write input to all of the FIFO controllers. This causes the stage holding the write token to perform a latch write, the ‘full’ state for that stage to be set, and the write pointer to move one position forward. Write events are blocked when all of the elements hold full states.

The stage that holds the read token makes the latches’ tri-state outputs active. When any stage indicates that it is full, the output handshake controller produces read requests which, when acknowledged, cause the ‘full’ state to be reset and the read pointer to be moved on.

When performing a loop, it is necessary to prevent the FIFO stages from being emptied when they are read, so that they can be read repetitively. However, it is necessary for stages that have been read from to *appear* empty to the output controller to stop further output requests being generated if no new data has arrived (an error that could cause the read token to overtake the write token). To avoid this requires a separate ‘full’ indication

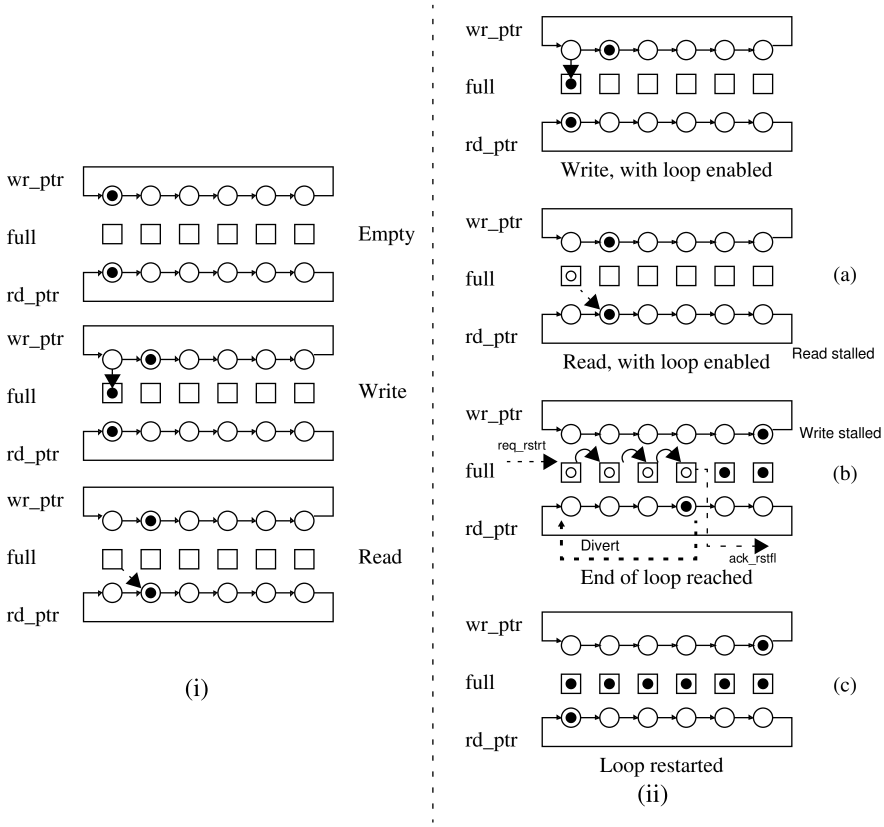


Figure 5.7 .Standard (i) and looping (ii) word-slice FIFO operation

to the input controller and ‘read request’ signal to the output controller. When performing a loop, read requests from each stage are cleared when the stage is read, without affecting the full indication. This is shown in Figure 5.7a, depicting a full stage with disabled read request by an unshaded dot in the ‘full’ boxes. When a pass through the loop has completed, a restart signal is issued which causes each of the FIFO stages to appear full again for the next loop. This operation is shown in Figure 5.7b and c. When not in loop mode, or when on the final pass through the loop, the output request behaves normally and the stages are cleared entirely when read.

Write and read token passing

A simplified view of the circuit making up the looping FIFO element is shown in Figure 5.8: a fuller description of the circuits used to implement the instruction buffer can be found in [137]. The write token flip-flops in all of the FIFO stages are connected together

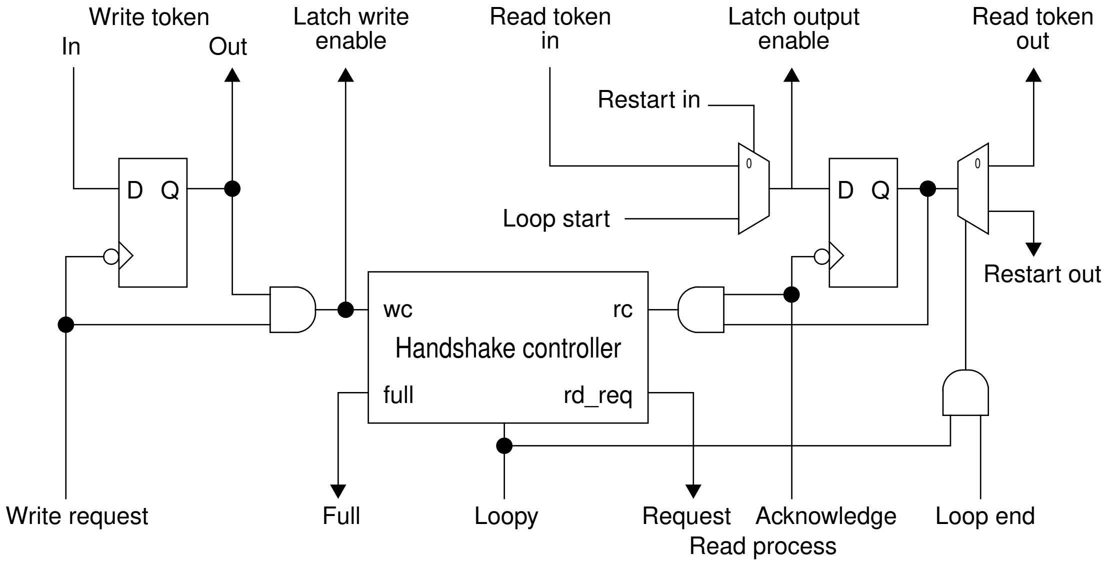


Figure 5.8 Looping FIFO element

to form a circular shift register, with the whole clocked by the write request signal from the input controller. The write token enters from the previous stage, and is accepted when the write request signal is driven high and then low (indicating a write in the previous stage). Once the element holds the write pointer, a further write request causes a write to occur in this stage: the latch write enable goes high, which opens the latches in the datapath. When the write request signal is removed, the latches close and capture the new data and the write token passes to the next stage. The write enable signal also indicates to the handshake controller that the stage should become full, which is indicated on the *full* signal to the input controller and the *rd_req* signal to the output controller.

The flip-flops holding the read token also form a shift register, clocked by the read acknowledge signal from the output controller. However, to incorporate looping behaviour it is necessary for the token to be passed out of the normal flow to indicate the end of a loop, and for the token to be received again at the start of a loop.

In normal (non-looping) operation, the read token from the previous stage is multiplexed to the flip-flop input and causes the tristate output of the latch row to be enabled: the enabled latch row corresponds to the *previous* FIFO stage. When the read acknowledge signal goes high and then low (corresponding to the previous stage being read), the read token is captured by the flip-flop, and passes out to the next stage to enable the tristate latch outputs. A subsequent high on the read acknowledge signal causes the handshake controller to clear the *full* and *rd_req* output, emptying the stage, and when the acknowledge signal goes low again the token is cleared from the flip-flop.

In loop mode, the *loopy* signal is set high and the FIFO stages at the beginning and end of the loop have their respective *loop start* and *loop end* signals set high. When the token reaches the stage at the end of the loop, the *restart out* signal is issued to the overall controller. The overall controller updates the loop count and sets the *restart in* signal, which causes the read token to re-enter the FIFO stage at the beginning of the loop. When the read acknowledge signal goes high in loop mode, only the *rd_req* signal is driven low by the handshake controller. The *rd_req* signal is restored for the next iteration of the loop by signals from the overall controller, which are not shown in the simplified figure.

5.2.3 Overall system design

In addition to the FIFO elements already described, the instruction buffer as a whole is made up of 3 other main parts: the input request interface that provides a 4-phase input interface, the output request interface that provides a 4-phase interface to the FIFO read signal, and the overall control unit. A block diagram of the top level structure, with the interface signals between each stage, is shown in Figure 5.9.

At the input request interface, write requests arrive on *Rin* whereupon the *nwr_req* signal is asserted to perform a write operation and the *Ain* signal is asserted. An internal matched delay is used to allow the write token to move and the full signal from the FIFO to stabilise, after which the input cycle either completes by returning *Ain* low or is stalled if the FIFO is full.

The control unit is the ‘brain’ of the instruction buffer, and interfaces the FIFO elements to the output, manages loops, and deals with reverse handshakes from the decode stage to set up loops or perform breaks and flushes. By handling both the forward and reverse handshakes at the output, it is possible to ensure that the data remains valid. The control unit is logically divided into the control core, made up of speed-independent logic, and the control datapath which is responsible for storing and updating the current loop status.

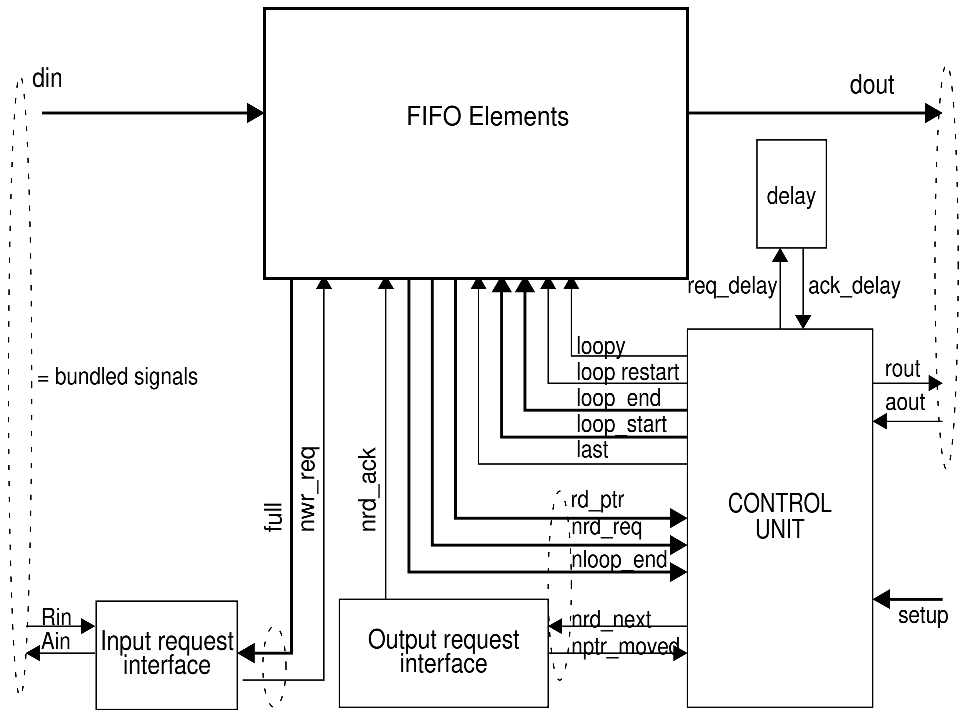


Figure 5.9 Looping FIFO datapath diagram

The main task of the control unit is to respond to read requests from FIFO elements, by initiating a handshake on *rout/aout*. When the decode stage acknowledges receipt of the data, the output request interface is signalled through *nrd_next* to move the read token to the next position. The timing for the move of the read token and the stabilisation of the signals from the FIFO is also managed by a matched delay, after which *nptr_moved* is asserted.

If the FIFO elements indicate that a loop end has been reached, the control unit updates the loop counter and restarts the loop. On the final pass through the loop, the next outermost loop (if any) is restored. Once the new token position is known to be correct, a final matched delay is used to mirror the delay from valid tristate FIFO output enables to valid data at the output.

5.2.4 PC latch scheme

It was mentioned previously that PC relative branch instructions require the associated value of the PC to be passed through the FIFO. This is unfortunate, as branches are comparatively rare instructions in this architecture and the requirement to store the PC initially seems to require an additional 24x32=768 latches which is a great waste of power and area. Fortunately, the sequential nature of the PC values means that this overhead can be greatly reduced. The instruction buffer contains a maximum of 32 sequential PC values, which means that, unless a carry out is generated from bit 4 of the PC, the upper 19 bits of the PC remain constant. A carry out will be reflected by a change in bit 5.

This behaviour is altered slightly when branches are considered: in this case, the PC can change to a random value. However, when a branch is taken the instruction colour tag is changed so that the decode stage can discard prefetched instructions in the branch shadow before any other instructions can occur. It is therefore possible to store only the lower 6 bits of the PC in the FIFO, and to use 4 sets of latches to store the upper 18 bits. One of the 4 latches is enabled for writes, based on the value of bit 5 of the input PC and the current input colour. Similarly, only one of the 4 latches is enabled for output by bit 5 of the output PC and the output colour. This saves a total of 504 latch elements.

5.2.5 Control datapath design

The control datapath, as shown diagrammatically in Figure 5.10, is internal to the control unit and maintains the current loop status. It is driven by the control core which handles all of the complex interactions between the signals from the FIFO datapath and the reverse requests from the decode stage. The control datapath consists of a row of latches that holds the current state (loop start and end position, *first*, *last*, and *loopy* status, and the current loop counter). When a DO loop is set up, the current position of the read pointer from the FIFO datapath (encoded into 5-bit binary) is added to the requested number of instructions to make up the loop. The current read pointer and the result of the calculation are used to set up the new loop start and end positions. Before the new loop status is loaded, the old status (if any) is pushed onto the 16-entry stack. When the loop is exited, the stacked data is reloaded and the stack is popped, thereby allowing nested loops.

On each iteration of the loop, the control core requests that the loop counter unit decrement the value of the loop counter (although the loop counter is actually stored in inverted form and incremented). In parallel with this, the result is checked to see whether it will be zero, which indicates the last iteration of the loop. The loop counter uses a simple

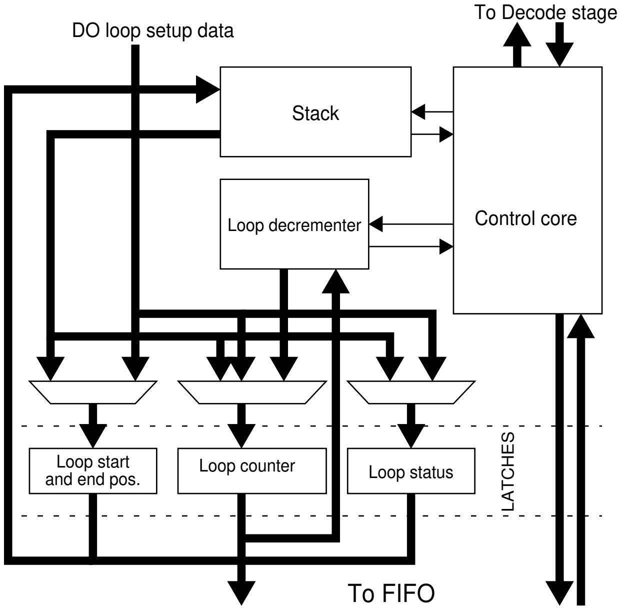


Figure 5.10 Top-level diagram of control datapath

data-dependent ripple-carry incrementer with a very similar design to that of the PC incrementer.

5.2.6 Evaluation of design

All testing of the instruction buffer was performed on netlists extracted from schematics, as the DSP construction has not yet moved into the layout phase. The initial verification of the design, during and after the design of the circuits, was done with the instruction buffer in situ, as part of the main DSP pipeline executing test programs under the *TimeMill* simulator. A selection of loops, nested loops, BREAKs and flushes were performed successfully. In addition, the loop counter unit was tested with a separate C simulation model, to set up and measure the delays for each level of carry propagation both within the loop increment circuit itself and for the incrementer cycle time including the time to latch the new value.

Once the functionality had been verified, a new testbed was designed in which the instruction buffer could be tested in isolation. This consisted of a C simulation model that feeds random instructions, using sequential PC values with random branches, to the input of the buffer at a selectable rate. The output from the buffer is then captured and compared with the value that should be present, and the latency from the input to the output of the buffer is measured.

As a baseline with which to compare the instruction buffer, a 32-element 4-phase micropipeline FIFO [90] was also designed (the 4-phase asynchronous interface making it easily interchangeable with the instruction buffer). The same tests were performed with the micropipeline design.

Two sets of tests were performed, using the *PowerMill* simulator to compare power and performance figures. The first set of tests fed 500 random values through each buffer at the maximum rate at which it would accept them. The second set of tests fed the same 500 values through each buffer at intervals of 20ns, which was significantly slower than the cycle time for both circuits. This models the case of the memory being slower than the stage into which the FIFO is feeding, and measures the latency from input to output. In both cases, current consumption was measured for each design.

5.2.7 Results

Loop counter performance

The delay figures for the loop count incrementer are shown in Table 5.2. The delays are shown for the four different possible groups of carry chain length.The results that have been obtained give a mean delay of $\bar{d} = 2.31ns$, which is close to the minimum delay as expected.

Max. number of carry stages	Inc. delay (input to output request) / ns	Loop counter cycle time / ns
3	0.66	2.25
7	1.41	3.13
11	2.48	4.33
15	3.12	5.04

Table 5.2: Incrementer delays

	Cycle time	Throughput	Latency
Instruction buffer	6.0ns	167MHz	2.7ns
Micropipeline	2.0ns	488MHz	26ns

Table 5.3: Maximum throughput and minimum latency

	Average energy per input cycle	
Rate	Maximum	50MHz
Instruction buffer	0.32nJ	0.48nJ
Micropipeline	0.67nJ	0.77nJ

Table 5.4: Energy consumption per cycle

The comparison between the instruction buffer and the micropipeline FIFO shows the instruction buffer to have a throughput that is less than that for the micropipeline design by a factor of three (although the micropipeline design does not have the additional circuitry required to perform looping). However, the micropipeline FIFO exhibits a latency that is a factor of ten greater than the instruction buffer. The cycle time results are acceptable, being much less than the 25ns cycle time dictated by the DSP application, even when added to the worst-case loop counter increment time. The low latency will ensure that instructions pass from memory to the decode unit as quickly as possible. Naturally, these figures will be degraded somewhat when interconnect delays and capacitances are taken into account but should still easily meet the specification requirements.

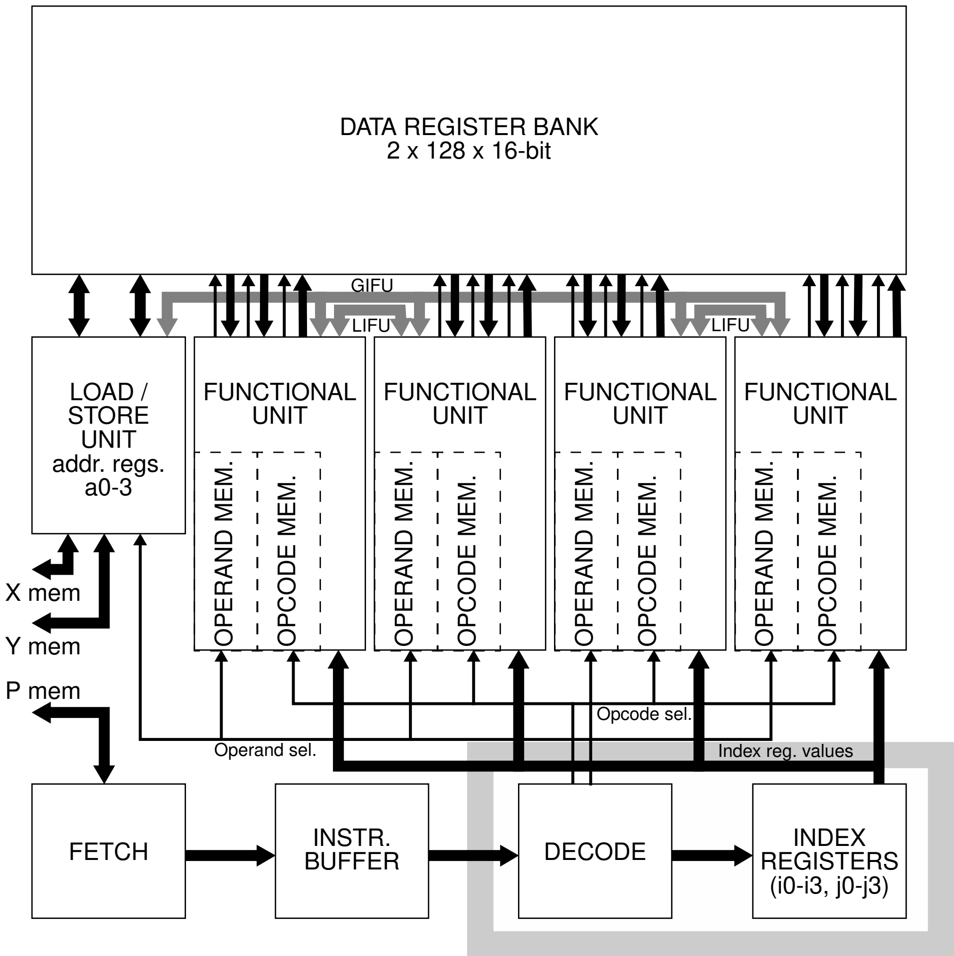
It was observed during testing that the bulk of the cycle time was required for the tri-state outputs of the latches to drive the broad output array. In a design that requires greater throughput it would be possible to split the outputs into two or more sections, with a controller for each section that moves a read pointer at a rate reduced by factors of two for each subdivision. This would allow the design to be scaled to an arbitrary degree, with the number of gate delays from input to output increasing only by the logarithm of the number of stages.

Compared to the micropipeline FIFO, the word-slice instruction buffer exhibited reduced energy per data value transferred in both the test cases, giving an energy per input of 48-62% of the energy for the micropipeline design. The fact that the instruction buffer outperforms the much simpler micropipeline FIFO is evidence that this was a good choice of circuit structure for low power. It also illustrates one of the key benefits of asynchronous design: while the instruction buffer has much more circuitry than the micropipeline FIFO, much of the circuitry in the instruction buffer is inactive during normal operation, and being idle consumes virtually no extra power. The arguments for splitting the tristate outputs into sections could also be applied to power consumption, by reducing the switched capacitance at the output. However, this would probably only be of benefit for larger sizes of buffer. Later results with back-annotated capacitances from the final layout should better answer this question.

Two improvements to the design of the instruction buffer suggest themselves. Currently the sequential way in which a loop is reset at the end of an iteration causes a delay that increases with the length of the loop. Instead of the current method for loop reset, it would be possible to use a latch to store the nesting level of the loop in each FIFO stage when it is read in loop mode. This would allow those FIFO stages who have the correct value stored to be reset in parallel at the end of an iteration, while other stages from outer loops are untouched.

The second improvement that suggests itself is somewhat more technically challenging; to reduce the time taken to flush wrong-coloured instructions following a branch. Presently, up to 32 instructions may have to be read and discarded by the decode stage after a branch instruction has executed. As the decode stage asserts control over both the input and output of the instruction buffer during a branch, it may be possible to implement a way of quickly purging unwanted instructions as an extension to the flush mechanism.

Chapter 6: Instruction decode and index register substitution



6.1 Instruction decoding

A summary of the instruction set for CADRE is presented in Appendix B. The instruction set was designed with two aims in mind. The first aim was that the most common instructions should have the simplest encoding, leading to faster decode times and reduced power consumption. The simplest encoding is for parallel instructions, which are indicated by a zero in the most significant bit. All other instructions (for processor control and setup) have a one in the most significant position, and have progressively more complex encodings of the subsequent bits. The second requirement was that instructions must be allocated according to the number of bits that they require. This works well, as the instructions that require the greatest number of bits (move-multiple to index registers and address register setup) are also two of the more common setup instructions.

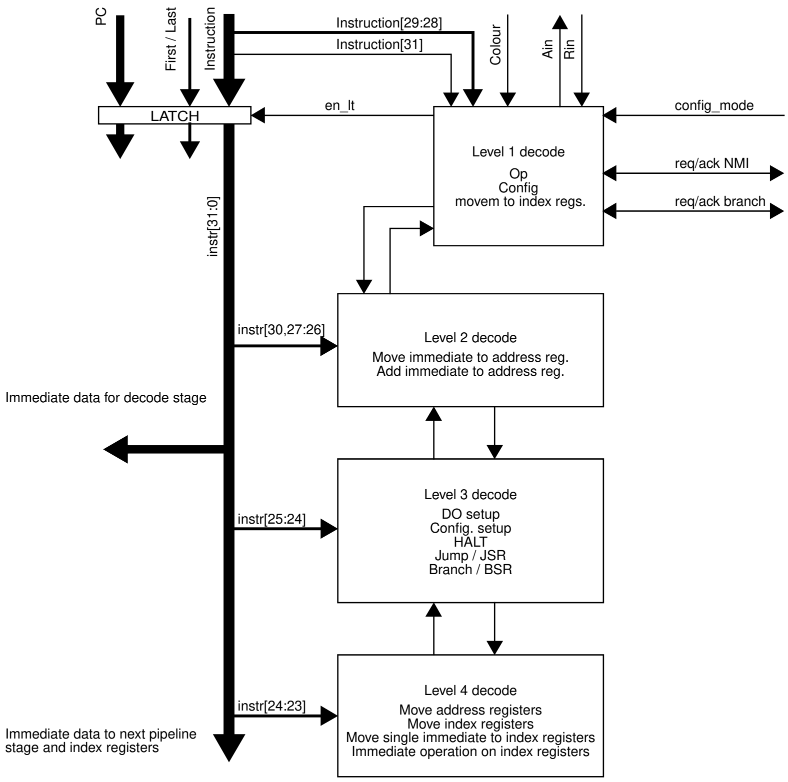


Figure 6.1 Structure of the instruction decode stage

The structure of the decode stage reflects the hierarchical design of the instruction set, with a succession of decoding levels as shown in Figure 6.1. The first stage of the decoding tree also performs the function of latch controller for the input of the decode pipeline stage. A request is then routed through the decoding hierarchy until a matching instruction is found, whereupon a request is issued to perform the appropriate task. When the task is completed, the resulting acknowledge signal is passed back up the tree to indicate completion.

6.1.1 First level decoding

The first level decoding stage is responsible for the following tasks:

- Controlling the input pipeline latches.
- Maintaining operating colour, and checking incoming instruction colour.
- Commencing parallel operation execution.
- Executing move-multiple-immediates to index registers.
- Beginning level 2 of decoding.
- Passing on configuration data.
- Changing operating colour on changes of control flow.
- Arbitrating between instructions and non-maskable interrupt requests.

The first decision that must be made, before decoding the instruction, is whether the incoming instruction matches the current operating colour, which is stored in this decoding stage. If the colours do not match, then an acknowledge is issued immediately.

Only if the colours match are the pipeline latches opened and, in parallel with this, bits 31 and 29:28 are checked to determine whether a parallel instruction or a move-multiple is to be executed. If neither of these cases apply, then a request is passed onto the next stage of the decoding hierarchy. While the forward request is being issued, the pipeline latches are closed to capture the data and an acknowledge is passed back to the instruction buffer to complete the input handshake.

One exception to the normal decoding process occurs when the processor is writing data to the configuration memories. The configuration process begins with an initiation instruction, specifying what type of configuration is to be performed and how many words of configuration data are to follow. The initiation instruction is decoded in a later stage of the decoding hierarchy and, before the acknowledge is issued, the *config_mode* signal is asserted. Subsequent instructions are passed directly to the configuration module, which releases *config_mode* once the configuration process is completed.

Parallel instructions

Two separate tasks are performed when a parallel instruction is to be executed. Firstly, reads to the four operand configuration memories, the load / store configuration memory and the index update configuration memory are requested. Bits 13 to 7 of the instruction word specifies the operand selection, and this is driven onto the operand bus and a request is sent to all of the operand configuration memories to initiate their read process. Once the data has been read and latched, each memory responds with an acknowledge.

While the configuration memories are being read, the condition field of the instruction is examined to determine whether a loop condition is requested. The loop conditions request that either execution, writeback or load / store is made conditional on either being (not) the first instruction in a DO loop or (not) the last instruction in a DO loop. The appropriate condition is evaluated, and the instruction passed on to the next pipeline stage is modified appropriately.

If the execution is to be made conditional, the condition field is modified to code for either AL (always) or NV (never) depending on the result of the test: this will only affect those instructions for whom the ‘conditional execution’ bit (bits 30:27 of the instruction) are set. If writebacks are to be made conditional, then the global writeback enable bit (bit 16 of the instruction) is set to indicate the result of the test. Similarly, if load / store operation is to be made conditional, the global load / store enable bit (bit 14 of the instruction) is set. Those bits that are not being driven by a loop condition are passed unaltered.

Once all of the configuration memories have been read and any conditional modifications have been performed, the instruction is passed on to the next pipeline stage. The next stage captures the instruction and responds with an acknowledge.

Move-multiple-immediate instructions

Move-multiple-immediate instructions allow 4 index registers or their associated update registers or modifier registers to be loaded with immediate data from a single instruction. This allows the process to be set up very quickly prior to or during the execution of an algorithm. The 4 7-bit register values are stored as immediate data in the 28 least significant bits of the instruction. Bit 30 indicates whether the i or j group of index registers is the target, bit 29 selects the update registers and bit 28 selects the modifier registers.

A move-multiple request is issued to the index registers, in the next stage of the pipeline, while the instruction word is passed unchanged. The move-multiple operation is not pipelined, to minimize latency, but the request is stalled if the next pipeline stage is not free. When the index registers have all captured the immediate data, they respond with an acknowledge signal.

Other instructions

Bits 28 and 29 of the move-multiple instruction are mutually exclusive: the target cannot be *both* the update and modifier registers. The encoding of bit 31=1, bits 29:28=11 is used to indicate all other possible control and setup instructions. If this pattern is detected, a request is issued to the next stage of the decoding hierarchy.

Changes of control flow

The first decode stage maintains control of the operating colour. All instructions which change the flow of control must, therefore, request changes of operating colour from the first decode stage. There are three cases when this can occur: conventional jump / branch instructions, cooperative branch interrupts following HALT, and the non-maskable interrupt.

Conventional jump or branch instructions are decoded by later levels of the hierarchy. When a branch is taken, a request is made for the colour to be changed. Only when the colour has been changed does the branch instruction issue an acknowledge back through the decoding hierarchy, ensuring that the operating colour is stable before the next instruction is read.

Similar to a branch instruction, execution of a HALT instruction causes the decode process to be suspended. When a cooperative branch interrupt is accepted following the halt, a colour change is requested; and only after this has been acknowledged does the halt instruction complete and execution continue.

The non-maskable interrupt adds somewhat more complexity, since this can arrive at any time. NMI requests are managed by means of a mutual exclusion element within the first decoding stage. Before each instruction is accepted from the instruction buffer, the decode stage attempts to gain control of the mutex. Should it succeed, operation proceeds as normal. However, if a NMI request has arrived from the interrupt controller, then this gains control of the mutex. The decode stage responds to this event by issuing an acknowledge to the interrupt controller, which is then allowed to request a change in operating colour. Only when the operating colour has been changed is the NMI request removed, freeing up the mutex and allowing execution to proceed.

Two special cases exist surrounding the operation of non-maskable interrupts. If the processor is currently in the middle of a configuration instruction, it is necessary to abort it. This is dealt with by a separate handshake process which occurs when the NMI request is accepted. The second problem occurs if a branch instruction has just been issued, causing a colour change, and the prefetched instructions from the branch shadow are being discarded. If an NMI occurs during this time, then the operating colour will change back, and the instructions in the branch shadow can be executed erroneously. To avoid this, NMIs are disabled until a colour match occurs with the instruction stream coming from the instruction buffer.

6.1.2 Second level decoding

The group of units responsible for the second and subsequent levels of decoding are shown in Figure 6.2. At the second level of decoding, the two instructions which require the longest immediate value are decoded. These instructions are move to address registers (or their associated update and modifier registers), and addition of an immediate value to the address registers. The immediate component of both these instructions spans the lower 24 bits. The immediate move instruction is indicated by 0 in bit 30 of the instruction. The immediate add operation is indicated by a 1 in bit 30, and 10 in bits 27:26. In both cases, the appropriate request and the immediate data is passed to the load / store unit, in the next logical pipeline stage. The requests are blocked until the pipeline stage is clear, to prevent any risk of hazards when accessing the address registers, after which the operations are performed without pipelining. Other instructions are classified into one of three groups, which are processed further by the third level of decoding.

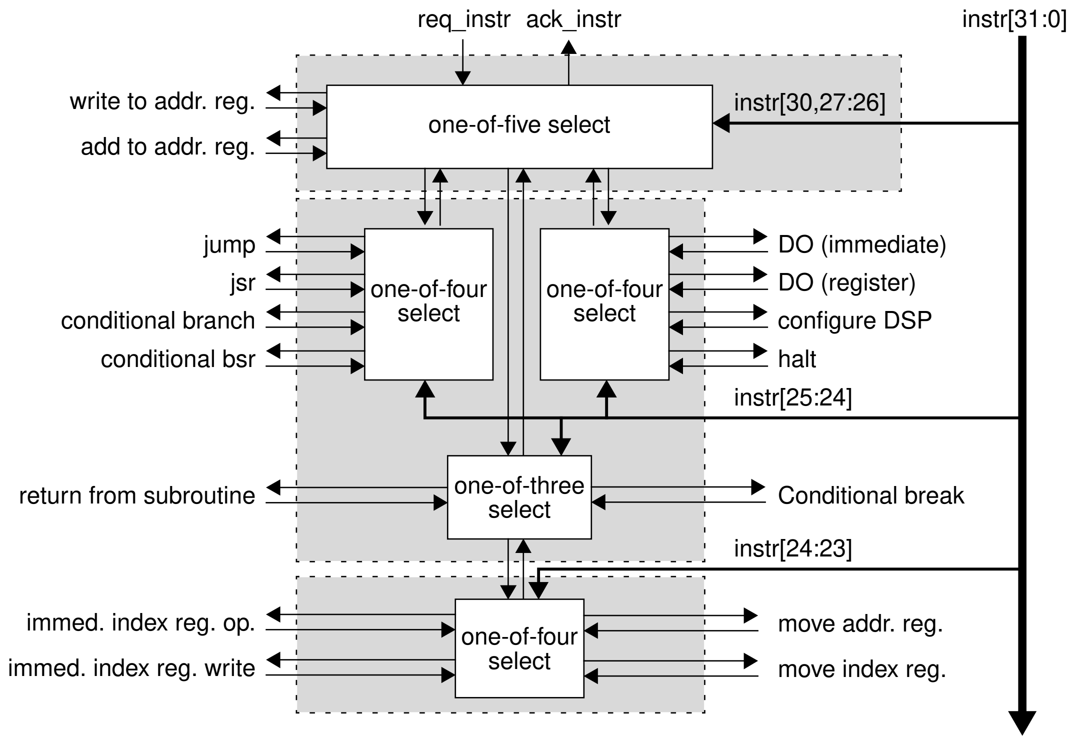


Figure 6.2 Second and subsequent instruction decode stages

6.1.3 Third level decoding

The first group of instructions dealt with at the third level of decoding consists of jump and branch instructions. The second group consists of DO setup, halt and configuration setup instructions. The third group consists of return from subroutine and loop break instructions; and also consists of all of the remaining instructions, which are passed onto the fourth and final decode stage. The choice within each group depends on the state of bits 25:24 of the instruction.

6.1.4 Fourth level decoding

The final stage of decoding deals with the least common instructions. These are moves of data between address registers, between index registers and between address and index registers, moves of single immediate data values to index registers, and immediate arithmetic operations on the index registers. The selection at this level is dependent on the state of instruction bits 23:24.

6.2 Control / setup instruction execution

Once a control or setup instruction has been decoded, the appropriate request is issued to one of a number of modules where the required operation is performed. These modules are located within the decode unit in the architectural diagram at the start of this chapter. Implementation details of these modules is beyond the scope of this thesis, but a brief summary of their functions and the instructions which they deal with follows.

6.2.1 Branch unit

The branch unit is responsible for all changes of control flow, including branch / jump instructions, return from subroutine, and interrupt response. The branch unit is also responsible for halt instructions. Included within the branch unit is a 16 entry stack for subroutine return addresses, and a 24-bit adder to calculate branch target addresses.

Conditional branch instructions are rare, and require a significant delay to gain access to the condition codes of the target functional unit which resides at the end of the pipeline, so the adder is implemented using a simple ripple carry structure which operates in parallel with the condition evaluation. When a change in control flow is required, the branch unit requests that the operating colour be changed, and then passes the new fetch address to the fetch unit and flushes any current DO loops from the instruction buffer.

When a halt is required, a request is passed to the load / store unit. This propagates along to the end of the pipeline before an acknowledge is issued which allows the halt state to be entered. This procedure ensures that any pending loads or stores are completed before a halt is indicated.

6.2.2 DO Setup unit

The DO setup unit is responsible for initialising DO loops, and also for performing conditional breaks from loop mode. DO loops are initialised by passing a loop count and instruction count to the instruction buffer. The loop count is obtained either from an immediate value in the instruction, or by requesting a register read from either the index registers (via the index interface) or the address registers (via the LS setup unit). The instruction count is always an immediate value.

Conditional breaks require that the status of the condition codes in the functional units be checked. If the condition is met then a break request is sent back to the instruction buffer.

6.2.3 Index interface

The index interface is responsible for performing writes to index registers, reads from index registers for DO loop setup, register to register moves and immediate operations. The read and write operations are not pipelined, but do require access to the following pipeline stage and therefore may be stalled.

6.2.4 LS setup unit

Similar to the index interface, the LS (load / store) setup unit communicates with the load / store unit, to perform writes and reads to address registers, register to register moves and immediate addition to address register values. The address registers are located in the following pipeline stage, and so access to them may also be stalled.

6.2.5 Configuration unit

The configuration unit is responsible for performing writes to the various configuration memories in the system. Configuration is initialised by an instruction which specifies the type of memory (opcode or operand) to be configured, the starting configuration address, and the number of addresses to be written. The configuration unit then maintains a count of the current configuration address and the number of entries remaining, and takes incoming instructions and passes them on in turn to either the 6 operand configuration memories or the 4 opcode configuration memories. The operand configuration memories occupy the same pipeline stage, so no stalling is required. However, the opcode configuration memories are located two pipeline stages downstream of the decode stage, so a delay may occur until the stages become free.

6.3 The index registers

The index registers are 7 bit values which are used to point to data in the register file. The index register units provide automatic updating of the register addresses as required by the algorithm currently being executed.

6.3.1 Index register arithmetic

Their are a total of eight index registers, grouped into two sets of four registers labelled i0-i3 and j0-j3.Their operation is based on the address generation scheme implemented in the Motorola 56000 series DSPs [14]. Each index register has associated with it two other registers; the update and modifier registers. The grouping of the registers is fixed: for example, index register i0 is always associated with update register ni0 and modifier register mi0. The update register is a 7 bit 2's complement value which can be added to or subtracted from the associated index register. The modifier register controls the two special functions supported by the index units: circular buffering, and bit-reversed addressing.

Circular buffering

Many algorithms require the repetitive processing of a fixed size block of data, where accesses to the data wrap around to the beginning of the block once the end of the block is passed. In a conventional microprocessor, this behaviour requires explicit bounds checking after each address update. The automatic provision of this function is one of the distinguishing features of DSP hardware.

To define a circular buffer of size N, the modifier register is set to $N - 1$. For example, a 20 entry circular buffer might go from register 0 to register 19, and the modifier register would be set to 19, the maximum index in the buffer. To prevent circular buffering, the modifier register is set to 127 (or -1 in 2's complement representation).

When using circular buffering, the start index of the buffers are restricted to the multiples of next higher power of 2 to the modifier register value. For example, with the modifier register set to 19, the buffer is allowed to start at register addresses 0, 32, 64 or 96. Index register values between the end of a buffer and the start of the next buffer are not allowed, and setting the register to this value will give undefined results on the next arithmetic operation.

When performing circular (modulo) arithmetic, the carry chain of the adder is split above the most significant bit of the modifier value. Below the split position, circular buffering is applied; while above it, standard 2's complement arithmetic is used. It is a requirement that the magnitude of the value added to the index register below the split point does not exceed the size of the buffer. However, an arbitrary value can be added above the split point which allows an algorithm to maintain a sequence of circular buffers and step between them. As an example: with the modifier register set to 19 (0010011 in binary) then the split position is located above the most significant bit, separating the circular buffer pointer (binary values 0-19) from the bits indicating the address where the buffer starts in the registers (0,32,64,96). If the index register is set to 18 and the value 33 is added to it the result is calculated in two parts. Firstly, the value 32 is added above the split position. Secondly, the value 1 is added below the split position with circular buffering. This gives the combined result 00/10010 + 01/00001 = 01/10011 = 18 + 32 + 1= 51. However, if the value 33 is added again, the result below the split point exceeds 19 and wraps round to zero as follows: 01/10011 + 01/00001 = 10/00000 = 64.

Bit-reversed addressing

Bit-reversed addressing is required as part of fast Fourier transform algorithm, and implies that the direction of carry propagation is reversed. Bit-reversed addressing is selected by setting the modifier register to zero.

6.3.2 Index unit design

All eight index registers can be updated simultaneously. To provide support for this, each index register is maintained by a separate index unit, which also stores the associated update and modifier values and contains the arithmetic elements required to perform the index update functions. Circuits for the index unit and details of their operation can be found in Appendix C on page 253.

The basic structure of the arithmetic element of the index units is shown in Figure 6.3. Index register arithmetic with circular buffering is performed in one or two steps. Firstly, the index register and update values are summed by the carry-save adder (CSA), with the third input set to zero. The carries are resolved by a ripple carry adder, which also implements the split in the carry chain. The result below the split point is compared with the current modifier register value to determine whether the result is within the bounds of the circular buffer. If it is, the operation is complete. However, if the bounds have been exceeded then the second step begins: an adjustment value is placed on the third input of the carry-save adder to bring the result back within the correct bounds. The carry resolution process is then repeated to calculate the final result. The two-step operation gives good average case performance, since the bounds are exceeded relatively infrequently, and may be implemented with a very simple and small circuit.

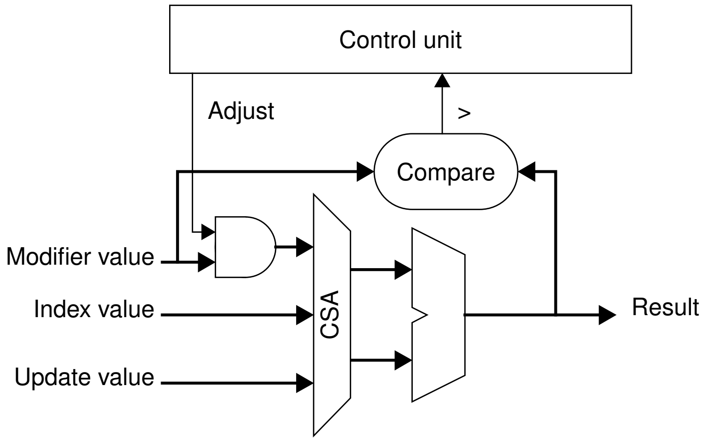


Figure 6.3 Index ALU structure

When bit-reversed addressing is selected (modifier register set to zero), the circular buffer mechanism is disabled and addition is always a single-step process. Two ways of implementing bit-reversed addressing were considered: using multiplexers on the input and output of the ALU and physically reordering the wires of the operands, or implementing a bidirectional carry chain where multiplexers on the carry path select normal or reversed carry propagation. The latter option was chosen, as it minimises wire lengths by maintaining nearest-neighbour connections. In retrospect, this is not the best solution: the carry multiplexers are on the critical path of the ripple-carry adder, which can impact twice on the performance during circular buffer operation. However, performance was well within the requirements for this design iteration.

6.4 Index register substitution in parallel instructions

Once a parallel instruction leaves the decode stage, two events occur. The relevant instruction components (operation selection, enable signals, condition codes) and the current index register values are passed to the functional units and to the load / store unit (which may require index register values). At the same time, the index registers are updated depending on the current value read from the index update configuration memory. The key elements of this process are depicted in Figure 6.4.

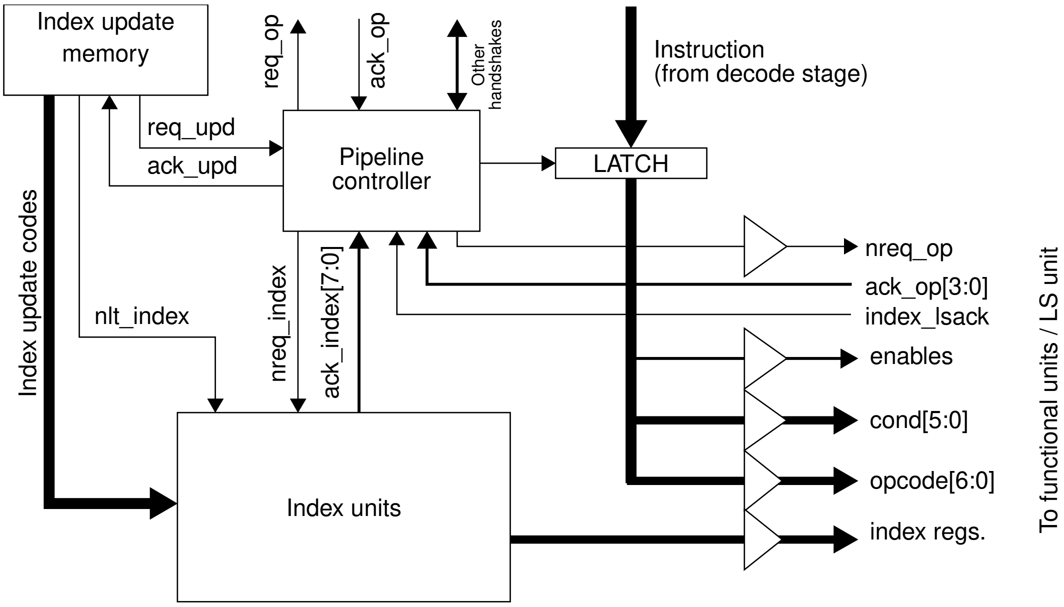


Figure 6.4 Passing of index registers for parallel instructions

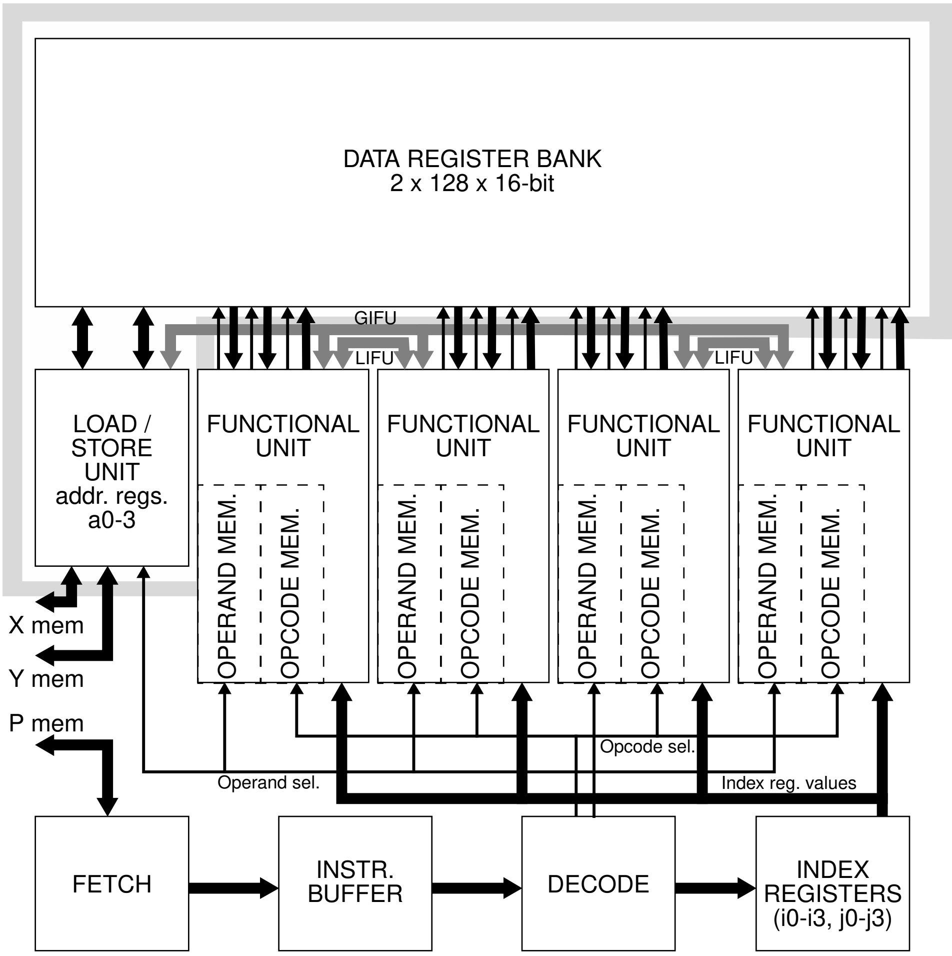
Within each index unit, the current index register value is passed out through a latch. The signal *nlt_index* from the index update memory controls this: when the new index update code has been read, the configuration memory sets *nlt_index* low to capture the current index register value, and issues a request on its output (*req_upd*). Since the output to the functional units is now captured, the selected index update can be requested by asserting *nreq_index*. At about the same time, the (possibly modified) instruction is also passed from the decode stage by the assertion of *req_op*. The pipeline latch is closed, and *ack_op* is asserted to indicate that the data has been captured.

Once both *req_op* and *req_upd* have arrived, both the instruction components and the index register values are known to be correct: these values are driven across to the functional units and the load / store unit, where the required values are captured before *ack_op[3:0]* are issued by the functional units and *ls_regack* is issued by the LS unit.

Once all the outputs have been captured and the acknowledges received, and the index update has indicated completion on *ack_index[7:0]*, an acknowledge is passed back on *ack_upd*, allowing a new index update code to be read. Similarly, the instruction latch can be reopened and any pending requests on *req_op* then be acknowledged.

The other functions managed by the pipeline controller are requests for access to all other operations in this and subsequent pipeline stages: writes / reads / immediate updates of the index registers, condition code checks in the functional units and writes to the configuration memories. These are routed through the pipeline controller, which blocks any request until the stage is cleared.

Chapter 7: Load / store operation and the register banks



The register file is at the centre of the CADRE architecture. During each instruction, up to eight reads can be requested from either the X or Y register bank, as well as a store operation from each bank which can read two further registers. Similarly, there can be up to four writes to either the X or Y register bank in addition to a load operation writing up to two registers in each bank. Clearly, the design of the register file can have a great deal of influence on the overall performance and power consumption of the system.

7.1 Load and store operations

Every parallel operation executed by CADRE can include load and store operations between the X and Y data memories and the register file, or a store operation from the GIFU to memory. These operations use address registers to identify the target of the operations in memory, which can be updated after each operation. A number of desirable features and constraints apply to the operation of loads and stores in relation to other accesses to the register file.

7.1.1 Decoupled load / store operation

Each parallel instruction can include load or store operations. However, when a load or store has been initiated it is undesirable to have to wait for these (potentially slow) operations to complete before another parallel instruction can take place. By decoupling the completion of load or store operations from the instruction stream, it is possible to place a load operation a few instructions before the point where the data is required, to prefetch the data and hide memory latencies. Similarly, store operations can be allowed to complete while the next result is being calculated. Processing is only paused if another load or store operation is requested while one is still pending.

7.1.2 Read-before-write ordering

To maximize the efficiency of code in terms of both number of instructions and speed, it is desirable to be able to execute as many operations as possible from within a parallel instruction. However, this brings about issues of how potential conflicts within an instruction are resolved.

Where a load from memory to a particular register occurs in parallel with an ALU operation accessing that register, the register must be read by the instruction before the load is allowed to complete. An example of code that requires this is shown in Figure 7.1: the MAC instruction uses the value that is in the register x:0 before the load operation overwrites it. This is a logical way of arranging events, since it is likely that the load operation will take significantly longer to complete than the register read and it allows data prefetches to be placed as early as possible in a sequence of instructions.

Similarly, when a store instruction occurs in the same parallel instruction as a writeback from a functional unit, it is necessary that the register is read for the store before being overwritten by the new data. An example of this is shown in Figure 7.2. Again, this is a natural arrangement: the writeback begins in the execution stage of the pipeline, while the register read occurs in the previous pipeline stage; so all that is necessary is a mechanism to ensure that the data has been captured before execution begins.

```
do #n
   {
      ; Reads current register x:0
      mac x:0,x:0,maca:a,maca:a
      ; Writes next value to register x:0
      load x:(r0)+,x:0
   }
enddo
```

Figure 7.1 Ordering for ALU operations and loads

```
{
   ; This writes to x:0...
   move maca:ah,x:0

   ; ... but x:0 is read first here
   store x:0,x:(r0)+
}
```

Figure 7.2 Ordering for ALU writebacks and stores

7.1.3 Write-before-read ordering

The decoupled nature of load operations makes it necessary to ensure that the requested data has arrived from memory before it is used by new instructions. Figure 7.1 is an example of code that requires this: the load from the previous iteration of the loop must have completed before the next iteration of the loop can be performed. This is performed by locking of registers. The X and Y register banks each have a single lock (as only one load can be in progress per bank at any time), and any attempt to read the register while the lock is in effect results in a stall until the load completes and the lock is removed.

As register reads occur in the previous pipeline stage to writebacks, a hazard exists when an instruction writes back a value to a register which is then read for the immediately following operation. It is the programmer's responsibility to ensure that an extra instruction is inserted in this case. The one exception to this rule is for store operations, where sequencing is enforced to ensure that the writeback has completed before the store commences. This allows loops to be written more concisely and stored in configuration memory without the need for special cases to deal with storing to memory, at the expense of a pipeline bubble being introduced for store operations. Examples of both cases are shown in Figure 7.3.

```
{
   ; Writeback to register x:0
   move maca:a,x:0
}
{
   ; Reads current register x:0
   ; ... Illegal, still being written.
   mac x:0,x:0,maca:a,maca:a
}
```

```
{
   ; Writeback to register x:0
   move maca:a,x:0
}
{
   ; Reads current register x:0
   ; Legal... waits for WB to complete
   store x:0,x:(r0)
}
```

Figure 7.3 Illegal and legal sequences of operations with writebacks

7.2 Load / store pipeline operation

The processor pipeline, shown in Figure 3.7 on page 103, combines parallel operations performed in a number of different physical blocks in the same logical pipeline stage. Other than the parallel arithmetic execution, the main area of parallelism is the load / store (LS) operations which are set up in parallel to the rest of the instruction. A highly simplified representation of the interactions data flow through the pipeline is shown in Figure 7.4: blocks which are grouped together physically are shown contained by grey rectangles. Operations outside the main pipeline sequence, such as load completions and writebacks, are indicated by a thick dashed grey border.

When a parallel operation has been identified in the decode stage of the pipeline, the first stage of configuration memory reads takes place, in the six separate operand configuration memories. Within each of the four functional units, the memories specify data sources and destinations and the choice of index registers to be used. Within the decode unit, a memory specifies how the index registers will be updated by the instruction. Finally, in the load / store unit, the LS configuration memory contains the register numbers or the index registers that specify the targets of the LS operations in the register bank, the address register selections for each operation, how the address registers will be updated, and the direction of both of the transfers. Full details of the encoding of the contents of each configuration memory is given in Appendix D.

The required index registers for each operation are known once the configuration memories have been read. While the current index register values are being sent to their destinations, the next set of values are calculated as per the instruction. In the functional units, the only operation during this pipeline stage is to receive and select the appropriate index register values. Within the load / store unit, index register values are also selected as needed. Also, the address registers used by the LS operation are read and the requested updates to their values are performed.

At this point, the functional units and LS units have obtained the details of the registers that they require from the register bank, and it is at this point that the pipelines converge. The interaction between the register reads and load / store operations is managed by the *lock unit*, which forms part of the register bank.

As read requests arrive from each of the functional units, they are compared with the targets of any pending load operations on the X or Y register bank. Should a match be found, the read request is delayed until the load has completed and unlocked the register. Each functional unit can request zero, one or two different registers.

Once all the read requests have passed the locking mechanism, and details of the current load / store operations have arrived, any requested load / store operations are initiated. If a load or store operation is required on a register bank where one is still pending, the process stalls here until pending operations have completed.

When a store to X/Y memory is requested from the register bank, it is necessary to wait for completion of any writebacks to the register bank from the functional units. This is done by waiting for the execution stage of the pipeline to signal completion, as this is when these writebacks are defined to happen. Registers required for the store operation are then fetched from the register bank, along with the registers requested by the functional units. Once the data has been supplied, the execution stage can commence operation, and completion of the store is decoupled from other read operations. Store operations therefore introduce a bubble into the execute stage. This could be avoided by additional locks to identify writeback targets, at the expense of substantial extra complexity. Alternatively, the programmer could be forced to insert an instruction between a writeback and a subsequent store, as is already required for conventional register reads after writebacks. However, it was felt that the benefit obtained from the denser packing of instructions outweighed the disadvantages of introducing a pipeline bubble, particularly as stores to memory are a relatively infrequent occurrence.

For the case of a load operation from X/Y memory to the register bank, the register lock on the X or Y register bank is updated with the new load target register. The load is initiated immediately, but an interlock within the register bank prevents the load from completing until register reads from the functional units for the current instruction have been completed, guaranteeing read-before-write in the instruction. This is only likely to affect operation when either the memory is very fast, or register reads are very slow.

Stores from the GIFU to memory are more complex, as they require the instruction initiating the store to have occupied the functional units, which in turn place the required value on the GIFU bus. As for a conventional store operation, the lock unit waits for the previous instruction to have completed before beginning the register read for the functional units. As soon as the required registers have been read, execution begins in the functional units and, when the functional units have indicated that valid data has been placed on the GIFU, the value on the GIFU is read and the store is initiated. Completion of the instruction is delayed until the value has been read (although in practice, the read will occur concurrently with execution of the instruction).

7.2.1 Address generation unit

The address generation unit consists of the four address registers (r0, r1, r2 and r3). Each of these has an associated update register (nr0-3) and modifier register (mr0-3). These groups of registers work together in a similar fashion to the index registers and their update and modifier registers. The update register is a 2s complement value which can be added or subtracted to the address register, while the modifier register either defines a circular buffer or, when zero, selects bit-reversed operation to assist with FFT operation.

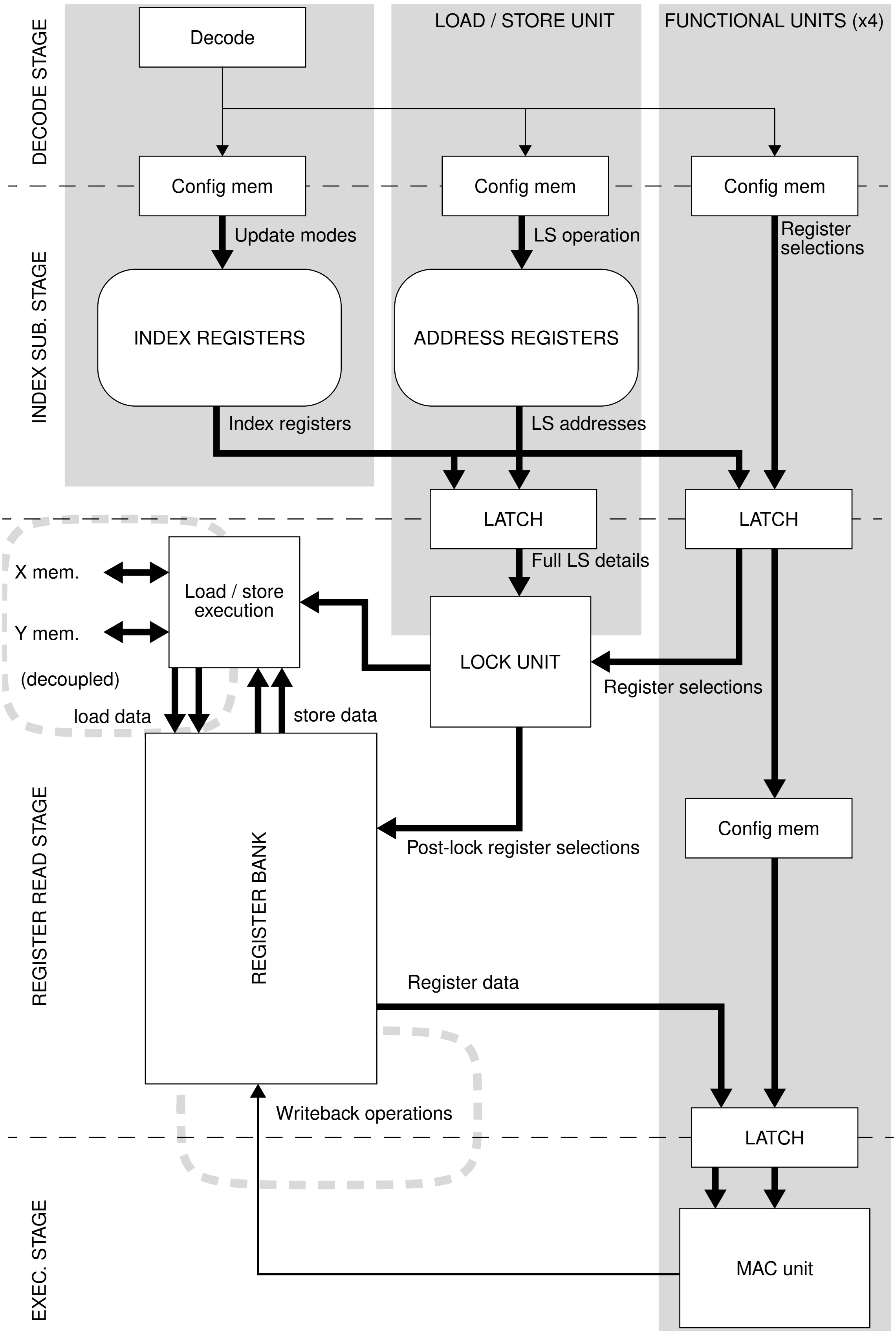


Figure 7.4 Load / store operations and main pipeline interactions

The main elements of the address generation unit datapath are depicted in Figure 7.5. For the sake of simplicity, the interfaces through which the address registers are set up are not shown, and neither is the control unit which manages the input and output handshakes for the pipeline stage. In contrast to the index registers, which have a separate ALU for each index register and can be updated simultaneously, only two address registers can be updated per instruction. Address register selection is performed using tri-state multiplexing: the X and Y register selections are decoded onto enable signals, which select one of the four register groups to be driven onto the buses to the X and Y ALUs respectively. Once the register selection has been made, the control unit closes the latches: the addresses are then ready for use by the next stage of the pipeline. At the same time, the selected address update begins in each of the X and Y ALUs. Once these updates have completed, write requests are made to the address registers. Only those registers indicated by the enable signals respond to the write request. The control unit is then ready to accept the next instruction and for the cycle to begin again, once the following stage has acknowledged the address and allowed the output latch to reopen.

Address ALU design

The specifications of the address arithmetic are virtually identical to those for the index units as specified in section 6.3.1 on page 146, except that they occur over 24 bits rather than 7. The extra width over which carries must propagate necessitates a somewhat different design approach when implementing circular buffering, although it was decided to use ripple-carry adders still due to their small size and low power consumption. The index register arithmetic was performed by adding the offset value, checking whether the result was within the bounds of the circular buffer, and then adding an offset value to correct the result if necessary. For the address unit, performing two such additions in series would take too long when using a simple ripple-carry adder. Instead, it was decided to evaluate both results simultaneously and select the one that fell within the appropriate bounds. An overview of the address ALU is shown in Figure 7.6. Shift operations are performed trivially through additional inputs to the output multiplexer, and the routing for this is not shown here.

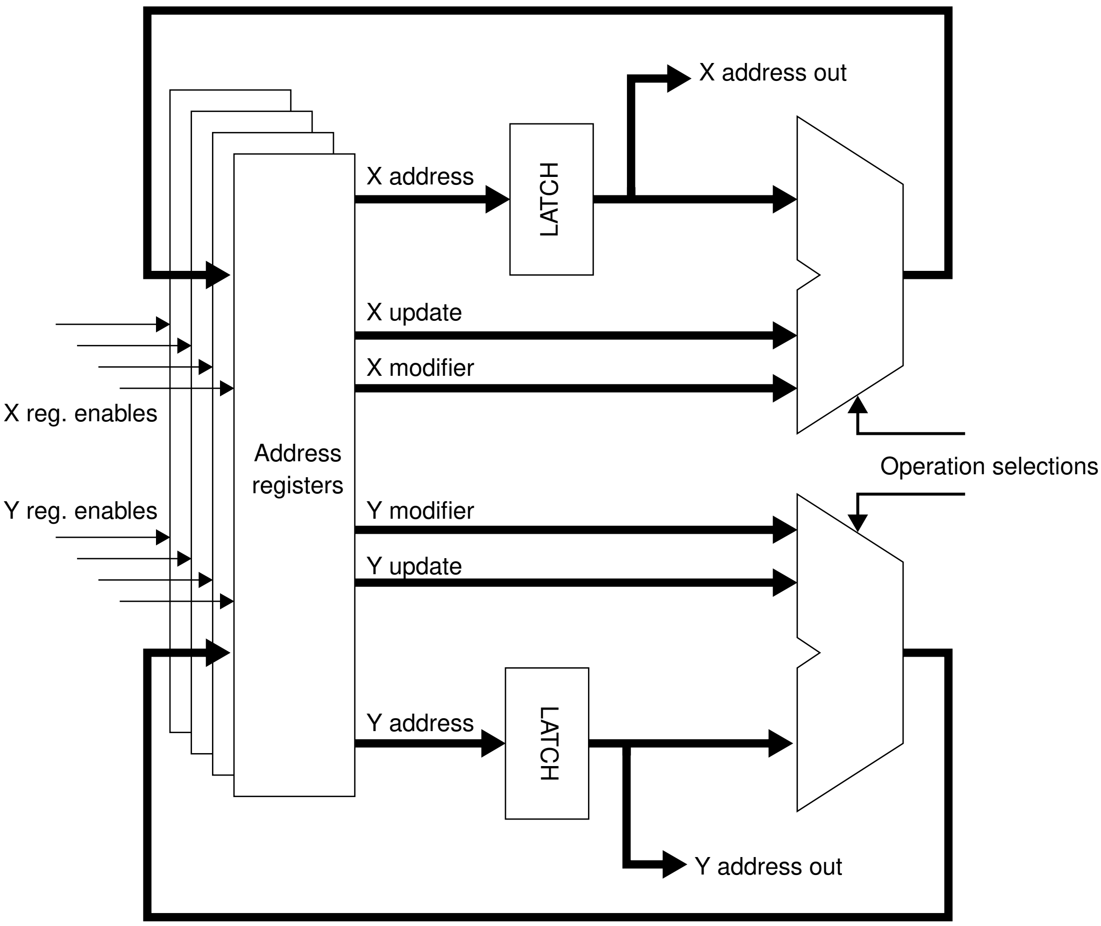


Figure 7.5 Structure of the address generation unit

The operation required can be a decrement, increment, subtraction, addition. The input conversion produces the appropriate offset value to implement the operation (e.g. by inverting the update value and generating a carry input for a subtraction), and produces the correct adjustment value to bring results back within the circular buffer bounds. For a subtraction, a positive value must be added to bring the result back within the bounds of the circular buffer while for an addition, a negative value must be added. The modifier value is also processed to determine the split point for the carry chain: for instance, if the modifier value were 100 decimal, then the carry chain would be split at the position corresponding to 128 decimal, the next power of 2. Arithmetic above the split position happens according to standard 2s complement arithmetic. A modifier value of zero bypasses the modulo arithmetic logic, and selects bit-reversed arithmetic which is performed only by the bottom adder circuit.

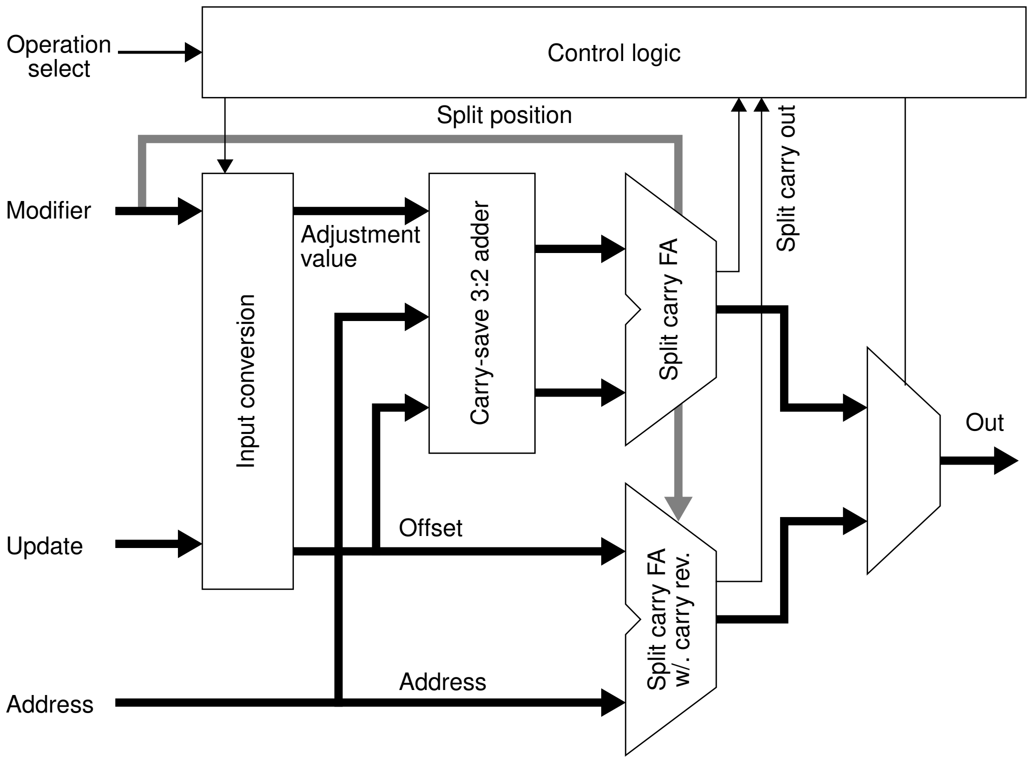


Figure 7.6 Address generator ALU schematic

When performing modulo arithmetic, the bottom adder circuit calculates the sum of the address and the offset. The carry-save adder (which has a critical path of 6 gate delays) adds the adjustment value, which passes to the upper adder to resolve the carries. The output is selected from either the adjusted or non-adjusted values by examining the carry output at the split point.

For an addition, the carry output from the adjusted value is studied: the adjustment is negative, so if a carry has been generated then the result of (address + offset - adjustment) is positive. This implies that (address + offset) was greater than the modulus, and the adjusted value should be selected; otherwise, the non-adjusted value should be selected.

For a subtraction, the carry output from the non-adjusted value is studied: the offset is negative, so if a carry has *not* been generated from the result of (address + offset) then the result is also negative, and the adjusted value should be selected to bring the result back into the positive modulus range. Otherwise, the non-adjusted value is passed.

7.2.2 Lock interface

The lock interface accepts the memory addresses, register selections and other load / store operation parameters, negotiates with the lock unit in the register bank and initiates the load / store operations. The schematic is shown in Figure 7.7, and consists of three components: the lock interface itself (*lock_if*) and the two execution units (*ls_execute*) which perform the load / store operations between X / Y memory, and the X / Y registers or the GIFU.

On receiving the load / store information from the previous pipeline stage, the lock interface latches the data and issues an input acknowledge. The next stage of operation is to perform the handshake with the lock unit but, if either of the execution units have an operation still pending (signalled by *x_ls_pending* and *y_ls_pending*), the lock interface waits until the operations have completed.

The exact sequence of events in the lock interface depends on the combination of loads or stores being performed. The simplest case is where no load or store operations are performed. In this case, the two enable signals (*x_en* and *y_en*) to the lock unit are low, and the lock handshake simply serves to synchronise the load / store pipeline with the main pipeline.

When a load from memory to one of the register banks is being performed, the appropriate enable signal is set high and *x_nload* / *y_nload* is set low. The target register for the operation is passed through *nx_reg[6:0]* / *ny_reg[6:0]* and the lock handshake is performed. This causes the target register to be locked in the register bank. Once this has happened, the load operation is initiated in the execution unit by asserting *x_lsinit_req* / *y_lsinit_req*. The execution unit commences the operation and asserts the pending signal, before responding with *x_lsinit_ack* / *y_lsinit_ack*.

When a store from a register bank to memory is being performed, both the enable signal and *x_nload* / *y_nload* is set high. For a store operation, it is necessary to ensure that any writebacks associated with the previous instruction have completed (write-before-read ordering), and it is also necessary to prevent the current instruction from executing until the data for the store has been read from the register bank (read-before-write ordering), as discussed earlier. The *hold_exec* signal to the lock unit prevents the current instruction from passing from the register bank to the execution stage, and is driven high when a store operation is detected. The lock handshake is then performed, and the store operation is initiated by asserting *x_lsinit_req* / *y_lsinit_req*. The execution units determine when the previous instruction has completed by monitoring the GIFU validity: during instruction execution, the GIFU valid signals are driven high and only return low at the end of execution. When all the GIFU valid signals are low, the execution unit reads the registers to be stored from the register bank. Only when this has completed does it respond with *x_lsinit_ack* / *y_lsinit_ack*. When both execution units have responded, *hold_exec* is removed and execution can continue.

A store from GIFU to memory begins in a similar fashion to a store from the register bank, with the *hold_exec* signal being set high to delay execution of the current instruction. However, this case is complicated by the need for the current instruction to enter the functional units and drive the GIFU correctly. Once the lock handshake has been performed, the store operation is initiated by asserting *x_lsinit_req* / *y_lsinit_req*. The execution units wait for the previous operation to have completed and the functional units to be empty, as for a normal store. This allows definite synchronization between the store operation and the current instruction which must drive the correct value onto the GIFU. However, before issuing *x_lsinit_ack* / *y_lsinit_ack*, *x_gifu_wait* / *y_gifu_wait* is driven high. This blocks the *op_done[3:0]* / *next_op* handshake, thereby preventing the functional units from releasing the GIFU once the current instruction has completed and ensuring that the value can be read by the execute unit. The corresponding *x_lsinit_ack* / *y_lsinit_ack* is then asserted, and *hold_exec* is released to allow the current instruction to enter the functional units. The GIFU will subsequently be driven, allowing the value to be read by the storing execution unit. Once the value has been read, the GIFU wait signal is removed and the instruction can complete.

7.3 Register bank design

A typical multiported register cell with n read and m write ports is shown in Figure 7.8. The data is stored by the cross-coupled weak inverters. Each of the read ports connects to one bit line (*Nop1...Nopn*, which go to all 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

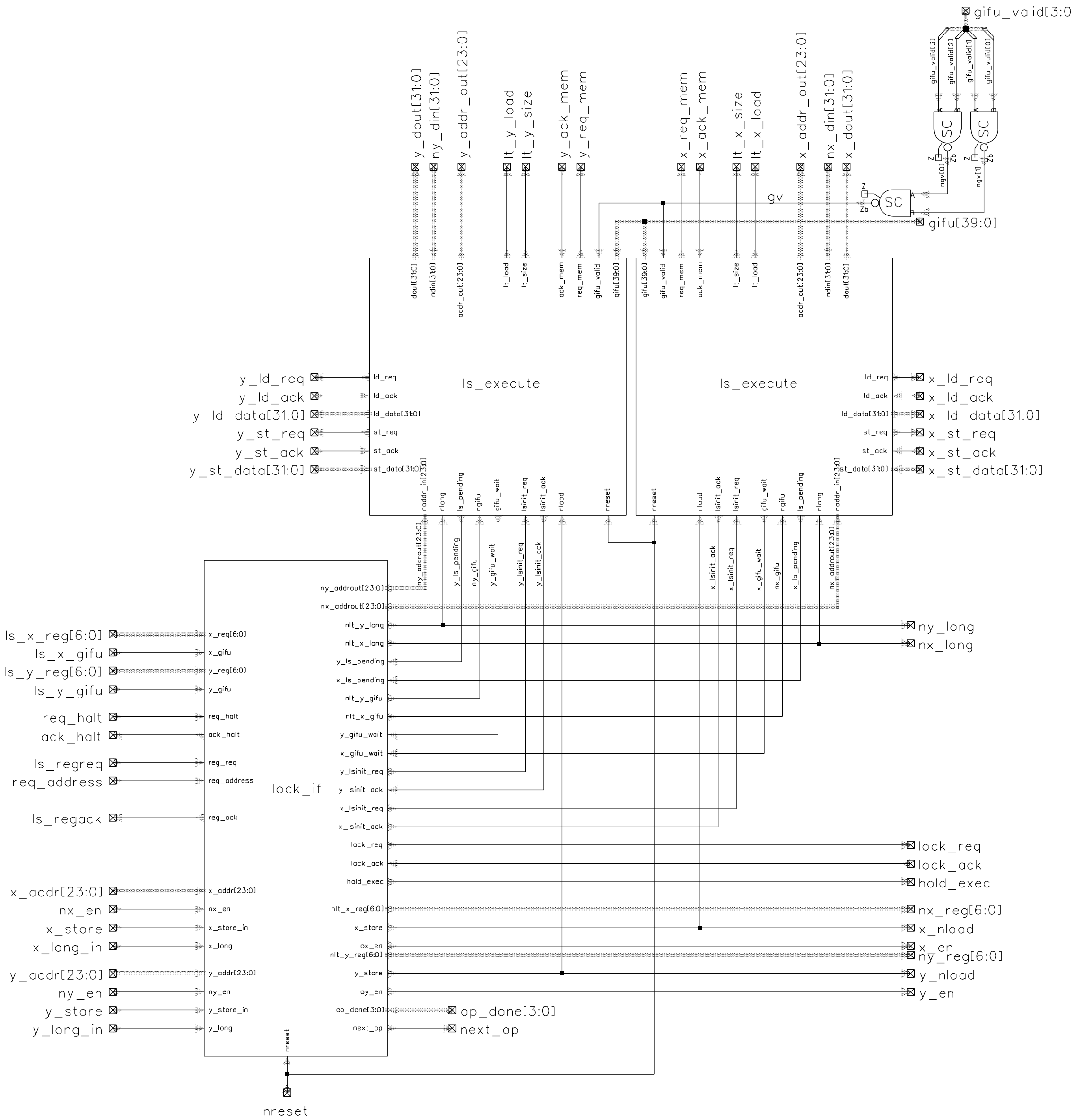


Figure 7.7 Lock interface schematic

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 7.9. Similarly, 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 area 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.

Is is claimed in [142] that the size of the register bank grows quadratically with the number of ports, which would be true if the size were limited by the wiring pitch of both the bit lines and word lines. It is suggested that, despite a number of power saving measures that can be employed, the register bank is likely to cause a major component of the power consumption.

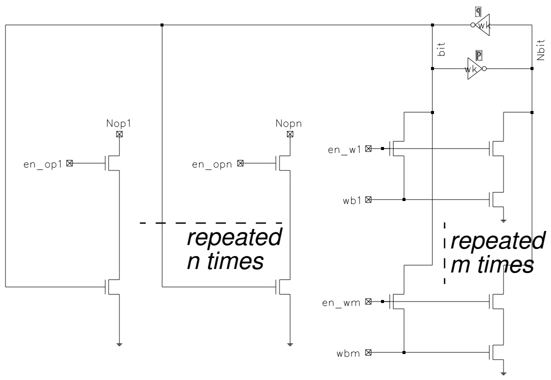


Figure 7.8 Multiported register cell

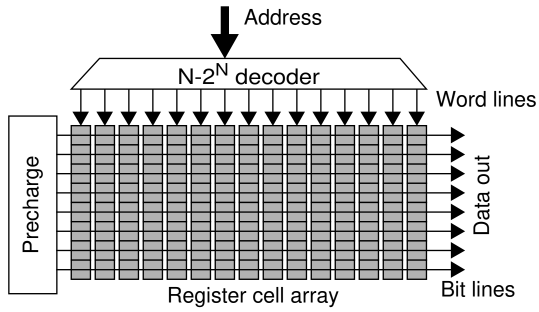


Figure 7.9 Word and bit lines in a register bank

One way of avoiding the energy and area cost of a large centralized 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 [143], but this adds hardware complexity and is not necessarily well suited to DSP algorithms where individual data values tend to be processed by many functional units.

The register bank for CADRE requires 10 read ports (2 reads by each functional unit, and data to be read for stores 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 loaded 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 of a conventional implementation, when using common data access patterns.

7.3.1 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 parallelised, this maps onto simultaneous requests to four consecutive addresses. Two important examples of this are the FIR filter algorithm and the calculation of autocorrelations (which is the dominant processing component of many speech codec algorithms).

FIR filter data access patterns

An N point FIR 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.

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.

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 7.1: Autocorrelation data access patterns

Where more than one autocorrelation value needs to be calculated, further optimisations 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. 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.

7.3.2 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 sub-banks, with the sub-banks containing sequential register numbers repeating every N^{th} digit. Given the mapping of operations 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 similar-sized 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 $n = 0\ldots7$) as shown in Figure 7.10.

When the code is written so that all the register accesses to each bank occur in different sub-banks, the power consumption and delay incurred will be that of an access to a single-ported 32-entry register file, with some overhead from the routing circuitry. Where contention for register sub-banks exists, a number of access cycles can be performed until

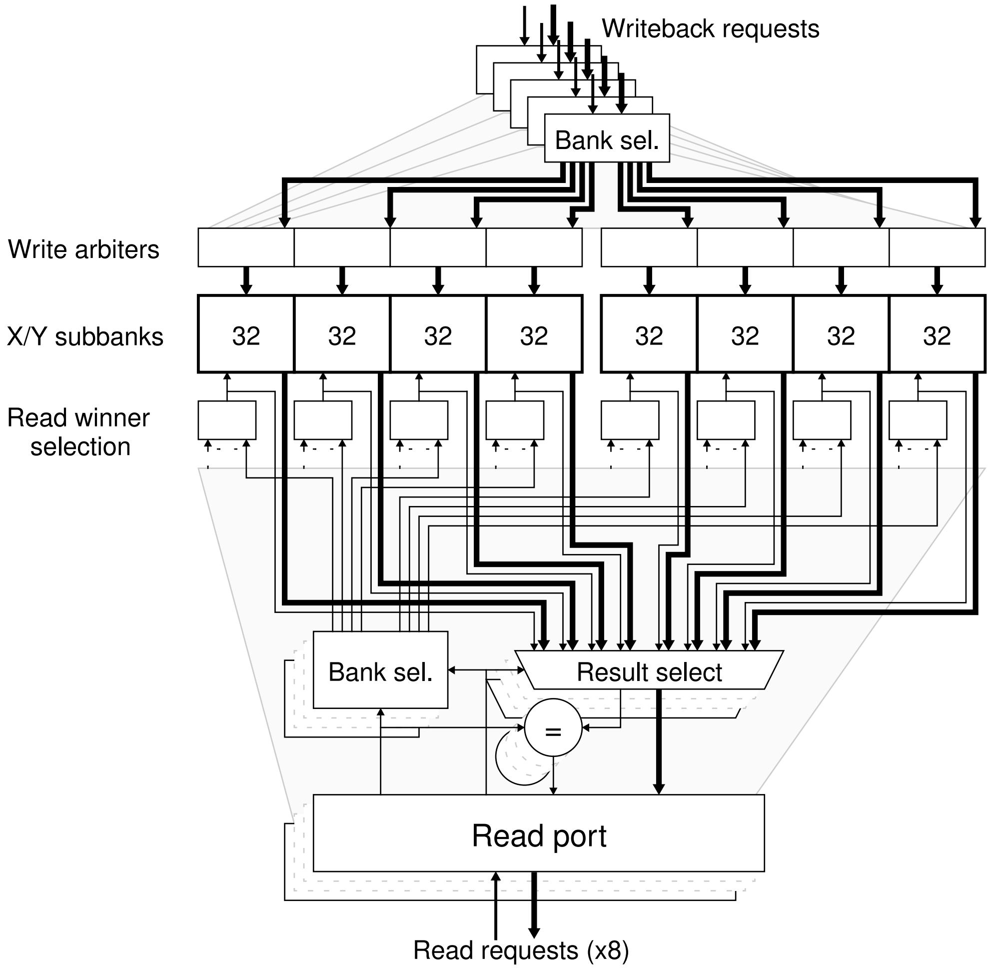


Figure 7.10 Register bank organization

all the accesses have been resolved. In the asynchronous domain, this represents no difficulty: surrounding stages will simply wait until the accesses have completed. The programmer need not be concerned with always maintaining optimal access patterns since, as long as the average access patterns are good, 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 access contentions occurred so that the operations are guaranteed to complete within the given clock period.

At the centre of the register bank in Figure 7.10 are the 8 X/Y sub-banks. Write and read requests are distributed to the various sub-banks, but the ways in which the write and read operations occur are very different.

7.3.3 Write organization

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 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 and writebacks, 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 being written back at the end of a pass through an algorithm.

The chosen mechanism for distributing writes is shown in Figure 7.11. 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.

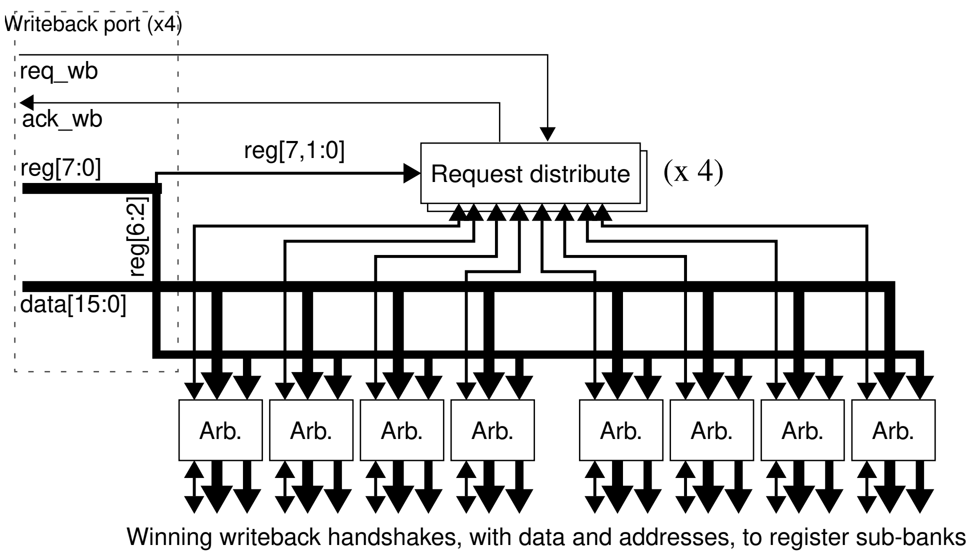


Figure 7.11 Write request distribution

At the input to each sub-bank, an arbiter block accepts possible write-requests from all 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 7.12 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 arbiter block is asymmetric: load completion is arbitrated after all 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 was 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 [144], to determine the winning request and then select the data and address corresponding to the winner (e.g. by using tri-state drivers). However, for this design speed was non-critical and the repeated tree structure gives a simple (and readily expandible) design.

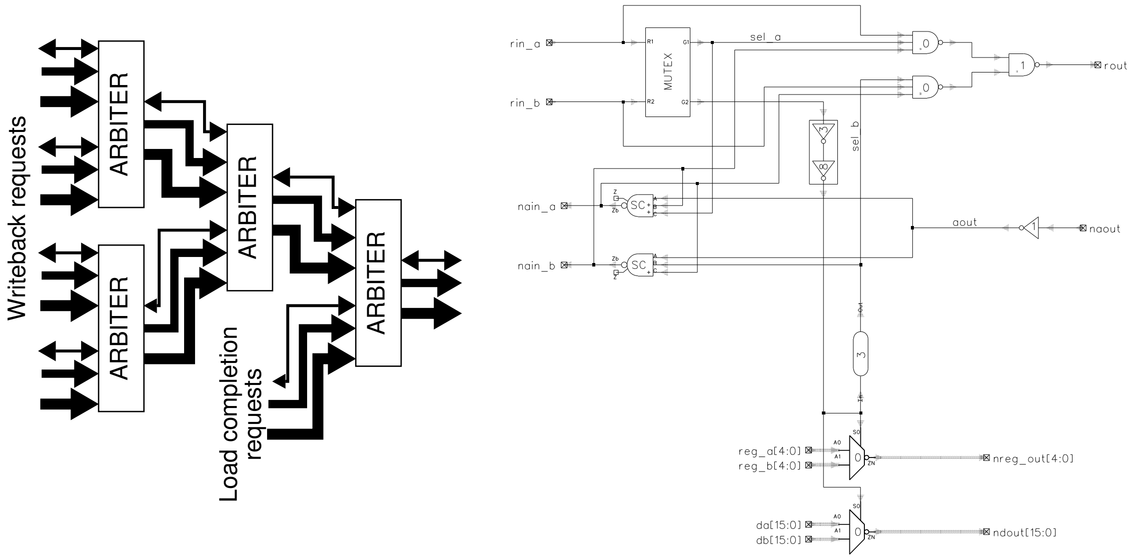


Figure 7.12 Arbitration block structure and arbitration component

7.3.4 Read organisation

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 register selections. 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 the close arrival of input requests. In addition, when a number of functional units all require access to exactly the same register (as occurs in the autocorrelation example in Table 7.1) it is undesirable that the same register should be read multiple times, for reasons of both performance and power consumption [146].

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.

The read mechanism is shown in more detail in Figure 7.13.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 connect to any sub-bank. The read ports operate semi-autonomously, passing requests for data across the switching network and capturing the data and sending it on to its destination when the request is satisfied. 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. The activity of the read ports is synchronised by two overall control elements: the lock unit, and the read controller.

Data being loaded from memory into the register bank can 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 by the completion of the load. If no conflict exists, the read request is passed on to the read port.

Read operation

When a read request arrives at a read port from the functional units, or a null handshake without an active request arrives, the read port asserts the go signal to the lock unit.

Each active 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 winning register selection is also passed back across the switching network to the requesting read ports, along with the output data, allowing them to determine when their register request has been satisfied.

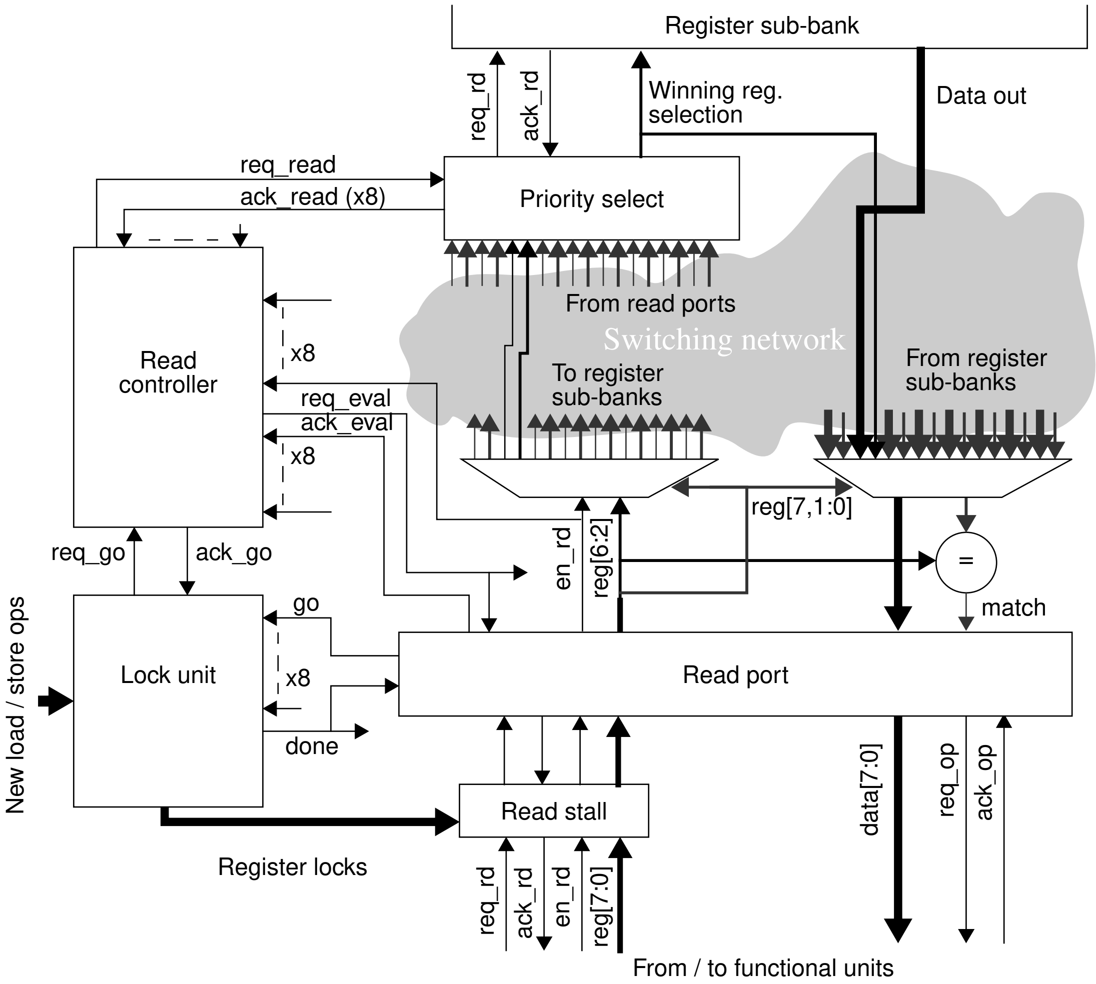


Figure 7.13 Read mechanism

Once go signals have been issued by all the read ports, the read process begins: this is the step where synchronisation occurs. First of all, new register locking information and details of loads and stores are accepted from the load / store unit. The new register locking information does not affect the state of any of the currently pending reads, allowing reads from a register and loads 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 performing read cycles as long as any read requests or stores are outstanding. Each read cycle begins by sending the req_read signal to all the sub-bank inputs. All 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. Matched delays in the control path ensure that changes to the read-requests pass across the switching network before the next read cycle begins.

Once the reads have completed, the read controller asserts req_eval to all 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 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. 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 complete their 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.

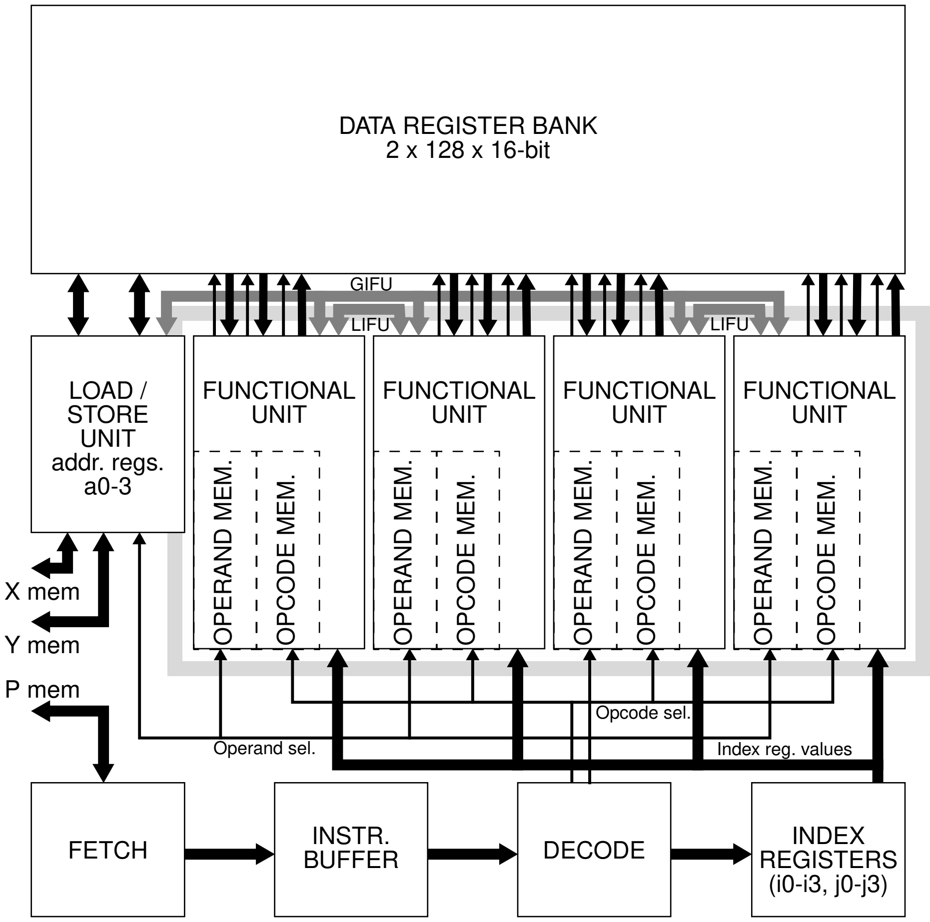
7.3.5 Register locking

To avoid complicating the description of the fundamental architecture of the register bank, only a small portion of the register locking / sequencing behaviour has been described so far. What has been excluded is the method by which read-before-write is guaranteed for both loads to the register bank and stores from the register bank.

When a load is being performed, it is required that any reads from the register bank in the same parallel instruction as the load will be completed before the load completes. To ensure that this happens regardless of the speed of the memory sub-system, a signal from the lock unit to the load port of the register bank is set high before the load instruction is accepted (and the loads themselves are begun). This signal prevents the load completion from writing to the register bank, and is cleared as soon as the read process of the current instruction is completed.

When a store is being performed, it is necessary that the data to be stored is read before any new writebacks, which may overwrite it, can occur to the register bank. The load / store unit ensures that the previous instruction has already completed, so the only source of danger is the writebacks that form part of the current instruction. To prevent these writebacks from occurring, the *hold_exec* signal from the load / store unit indicates to each read port that requests to the functional units should be stalled, although each read port collects the requested data in the usual fashion. Once the data for the store operations has been read, the *hold_exec* signal is removed, allowing execution to commence. Data requests from store operations are given priority at the register sub-banks, and are always serviced in the first cycle as they never contend with one another.

Chapter 8: Functional unit design



The only assumptions made about the nature of the functional units in the CADRE architecture are that they conform to the asynchronous interfaces at the various pipeline boundaries. The rest of the architecture can be viewed as simply a mechanism for feeding data to the functional units: to a great extent, the meaning of this data is left to the designer. This means that different units with radically different internal structures and functions can be selected for a particular application and, due to the clear asynchronous interfaces, these can be substituted for one another without great difficulties.

This chapter first describes the generic interfaces that must be implemented by all functional units. Secondly, the multi-purpose functional unit that was developed to evaluate the architecture is described. The assembler for the architecture currently supports only this type of functional unit: to allow different functional units to be interchanged easily, a more flexible framework would need to be developed whereby the assembler can be made aware of the mnemonics and characteristics of each functional unit.

8.1 Generic functional unit specification

The operations that make up the processor pipeline are, as has been mentioned previously, distributed in a number of separate physical units. This means that the functional units have a number of separate interfaces residing in different logical pipeline stages, with pipeline latches internal to the functional units as shown in Figure 8.1.

8.1.1 Decode stage interfaces

The primary interface to the functional units within the decode stage of the pipeline is *operand[6:0]*, bundled by *nreq_operand* / *ack_operand*. This is intended to specify the address in the operand configuration memory to be read for a parallel instruction, and is contained in bits 7-13 of the instruction word. The acknowledge is issued once the memory read has completed and the result has been latched by the following stage. However, the system designer is free to use a smaller memory, a combination of RAM and ROM, or indeed to dispense with a configuration memory altogether and treat the operand address as having some other arbitrary meaning.

8.1.2 Index substitution stage interfaces

During the index substitution stage, the current values of the index registers are transmitted to the functional units, along with the remaining fields of the parallel instruction. The intention behind this ordering is that the functional unit has determined which index registers it requires via the operand configuration memory's contents. The

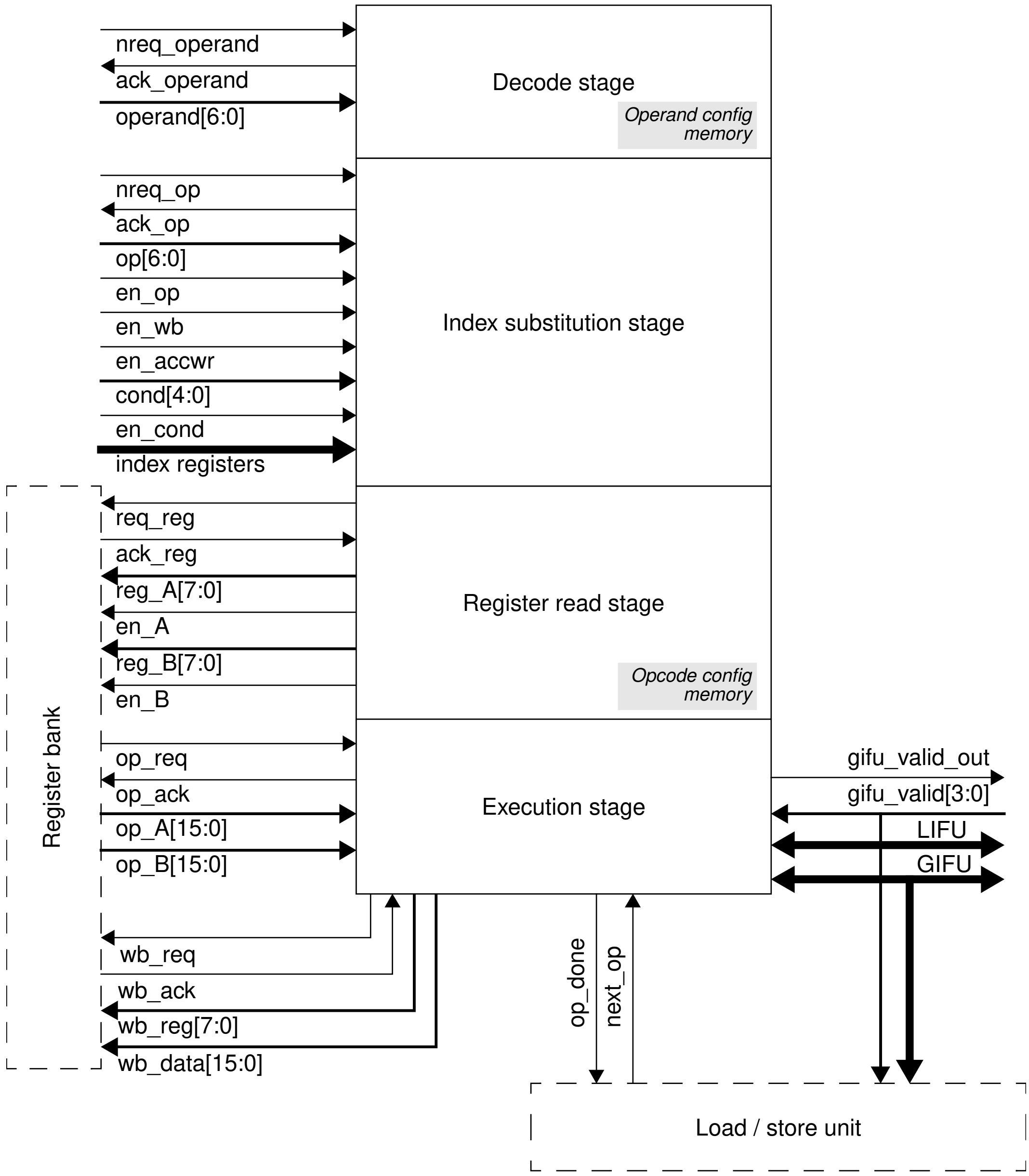


Figure 8.1 Primary interfaces to a functional unit

data is transferred through the *nreq_op* / *ack_op* handshake. Bundled by this handshake are the remaining instruction components and the eight 7-bit index register values (all of the index registers being passed to all of the functional units).

The first main component is the 7-bit opcode configuration memory address *op[6:0]*, contained in bits 0-6 of the instruction. This is intended to specify the opcode configuration memory location for the current operation but is open to other uses in a similar fashion to the operand address.

The second main component is a 5-bit conditional operation field *cond[4:0]*. This is contained in bits 18-22 of the instruction, and is intended to be used as a code specifying tests for operation against the functional units' internal state. Again, where this is appropriate, each functional unit can treat this as arbitrary data, but with certain restrictions: only values from 00000 to 01000 and from 10000 to 11000 binary may be used with impunity: other values are used for loop conditional operation.

Loop conditions are tested in the decode stage, and the value of the condition code transmitted onwards will be altered to either 00000 (intended to code for *always*) or 10000 (to code for *never*). This allows arithmetic operation in selected functional units to be conditional on the loop status. The loop condition may also cause changes in the writeback and load / store enable signals, in which case the condition code will be set to 00000.

The functional unit may only perform operations dependent on *cond[4:0]* when the associated *en_cond* signal is asserted. The *en_cond* signals are coded in bits 23-26 of the instruction, and allow each functional unit to perform conditional operations independently of the others. However, where the designer knows that all of the functional units have a common interpretation of the condition data, then these enable signals could be given an alternative meaning.

The final main group of bundled signals are a number of other enable signals: *en_op* is intended to activate or deactivate the main arithmetic / logical operation within the functional unit, with a separate bit for each functional unit contained in bits 23-26 of the instruction. Similarly, *en_accwr* is a single enable signal, intended as a global enable for parallel writes to the functional unit accumulators. This is coded in bit 15 of the instruction, and goes to all of the functional units. Both *en_op* and *en_accwr* could potentially be given different meanings. The final enable signal is *en_wb*, intended to enable writebacks from the functional unit accumulators to the register bank. This signal can be forced to zero by loop condition evaluations. Again, a functional unit for whom a writeback enable is unnecessary could give an alternative meaning to this signal.

Secondary interfaces

Two additional interfaces (not shown in Figure 8.1) are implemented at the index substitution stage: these allow tests on the condition codes in a particular functional unit (for conditional branch and break instructions), and perform writes to the configuration memories. Handshakes on the main interface and the two secondary interfaces are mutually exclusive.

8.1.3 Register read stage

Having received data through the interfaces in the previous pipeline stages, the functional units now take an active role, requesting the required register data from the register bank. The read request is made through the *req_reg* / *ack_reg* handshake signals. Bundled with this request are the register addresses *reg_A[7:0]* and *reg_B[7:0]*, and the associated enable signals *en_A* and *en_B* which indicate whether a read is required or not. As mentioned in the description of the register bank, a request must always be made even if no data is required, to allow synchronisation of read requests and the load / store process.

It is intended that, at the same time as the register read is being performed, the opcode configuration memory is read. The configuration data can then be passed locally to the next pipeline stage, to meet with the data arriving from the register read.

8.1.4 Execution stage

At the input to the execution stage, data arrives from the register bank on *op_A[15:0]* and *op_B[15:0]*, bundled by the handshake *op_req* / *op_ack*. Internally, other data from the register read stage such as configuration and enable signals will also arrive. Once the required data arrives, a number of different events are initiated in parallel, but only two have external interfaces.

Each functional unit may potentially drive either the GIFU or LIFU buses, and it is necessary to ensure that the bus has been correctly driven before any data is read from it. However, it is desirable that the processor should not deadlock if an incorrect configuration causes no functional units to drive the buses when one of them wishes to read it. To avoid this problem, each functional unit asserts its validity indication (*gifu_valid_out*) *either* when it has correctly placed a value on the buses, *or* if it will not be placing a value on either of the buses. A receiving device checks the state of all of these signals (*gifu_valid[3:0]*) and only proceeds once they have all been asserted. This means that in an error condition, an undefined value will be read from the bus (whatever value the weak bus keepers are currently maintaining) but the processor will not enter a deadlock condition.

As one part of instruction execution, the functional unit can request a writeback of data to the register bank. This is performed using *wb_req* / *wb_ack*, with the data and address bundled on *wb_data[15:0]* and *wb_reg[7:0]*.

Once all components of execution have completed, each functional unit indicates this fact by asserting *op_done*. These signals converge at the load / store unit, where another synchronising step is made: only when all of the *op_done* signals have arrived does it respond with *next_op* to allow the execution stage to proceed to the next instruction. This interaction is necessary to allow stores to be safely performed from the GIFU. Since the functional units are usually performing similar operations which are begun at similar times, this synchronisation only marginally reduces efficiency due to idle functional units.

8.2 Functional unit implementation

A top-level representation of the functional unit implemented for CADRE is shown in Figure 8.2. The functional unit is divided into four main components: the multiply-accumulate unit (*mac_unit*) where arithmetic and logical operations are performed, the operand decode stage which selects and processes incoming index register values and the data from the operand memory, the pipeline boundary between the index substitution and register read stages (*regrd latch*), and the two configuration memories which also contain an internal pipeline stage.

During the decode pipeline stage, the operand configuration memory is read by *nreq_operand* / *ack_operand*. Once read, the data is latched at the output and a request is issued on to the operand decode unit.

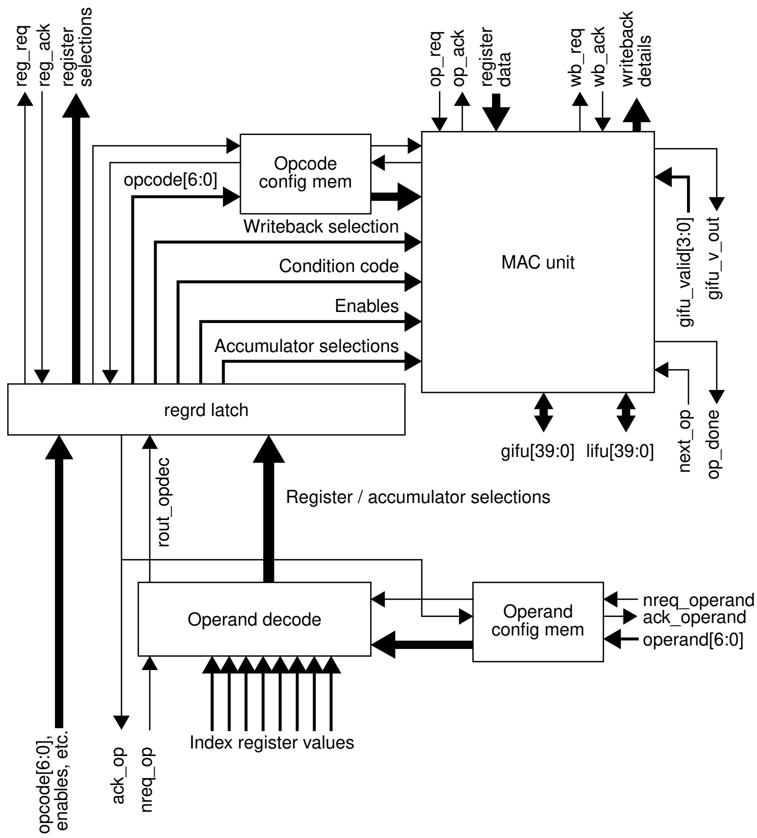


Figure 8.2 Top-level schematic of functional unit

When valid index registers arrive, signalled by *nreq_op*, the appropriate values are selected by the operand decode unit, and the register selections and various components of the operand configuration word are passed to *regrd latch* with a request on *rout_opdec*. When the data is captured at the register read boundary, *ack_op* is issued which allows the next instruction to enter the index substitution pipeline stage and the operand configuration memory.

From the latch at the entry to the register read stage, operation diverges. Firstly, the register request is sent to the register bank. Secondly, the opcode configuration memory is read. Along with the configuration data, the various enable signals and any immediate data are latched at the entry to the MAC unit. Once the requested data arrives back from the register bank, execution can begin. After execution has completed, the MAC unit asserts *op_done*. On acknowledgement by *next_op*, the functional unit may proceed to the next stage of operation.

8.3 Arithmetic / logical unit implementation

The arithmetic / logical unit (*mac_unit*) is made up of a number of blocks, which implement the various independent functions of the unit, as shown in Figure 8.3. Information required for the operation come bundled with handshakes from two separate sources: setup and immediate data come from within the functional unit, while register data comes from the register bank. It is anticipated that setup data would arrive first in an empty pipeline, but the unit must be designed to function correctly regardless of the order of arrival. A typical case when the pipeline is fully occupied will be that both sources of data will be simultaneously captured as soon as the unit becomes free.

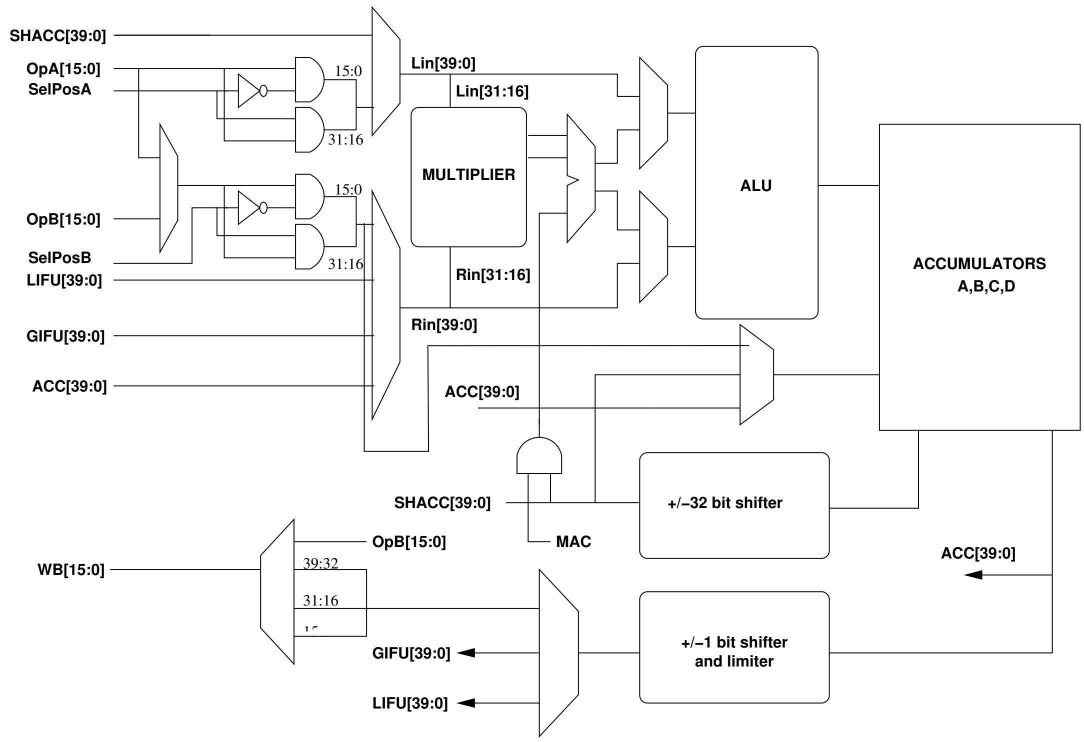


Figure 8.3 Internal structure of *mac_unit*

Three separate functions can occur within the unit: an arithmetic / logical operation with the result written to the accumulators, a writeback to the register bank, and a parallel write to the accumulators (e.g. an accumulator to accumulator move). Each of these operations can require data from a number of sources, and the functional unit is designed in such a way that each function can be performed as soon as the required data is available.

However, a number of constraints must also be applied, to ensure that the accumulators are read before they are overwritten and to allow store operations from the register bank to be completed safely. The sequencing of events that these constraints impose are summarised by Figure 8.4. Some orderings always hold, indicated by solid lines, while those indicated by dotted lines indicate possible sources of data required by a particular instruction: for example, an arithmetic operation using the shifted accumulator *shacc[39:0]* must wait until the shifted value has been produced.

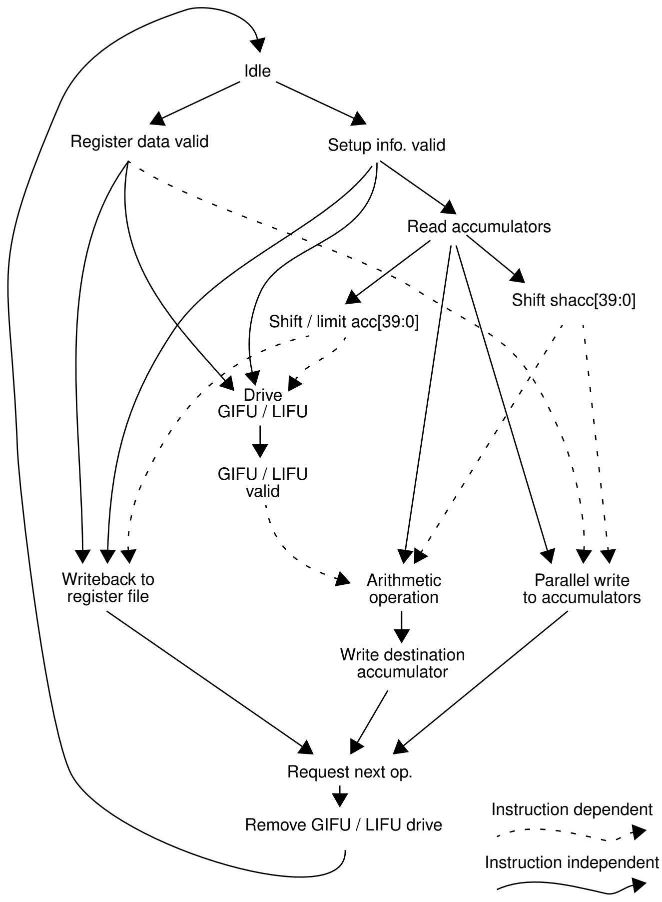


Figure 8.4 Sequencing of events within the functional unit

Before any operations may take place in the functional unit, the setup information specifying the operations must arrive. As soon as this has happened, the two accumulators specified in the instruction are read and latched at the *acc* and *shacc* ports of the accumulator bank (in the current implementation, two values are always read). This read must occur before any operation that could overwrite the contents of the accumulators.

Once the accumulators have been read, the shifted form of *shacc* may be generated: the time required is dependent on the shift being performed, with shifts of up to one place taking less time than all other shifts.

If the accumulators are the sources for the arithmetic / logical operation or the parallel write to the accumulators, these may now proceed. For any other source, it is necessary to wait for the data from the register bank to arrive. Writebacks to the register bank and driving of the GIFU / LIFU must also wait for the request to arrive from the register bank, to ensure the sequences of events required for store operations.

Once all three operations have completed, the functional unit requests to proceed to the next operation. Once this is granted, the GIFU / LIFU drive is removed and the functional unit re-enters the idle state.

8.3.1 Arithmetic / logic datapath design

A simplified diagram of the structure of the arithmetic / logic datapath is shown in Figure 8.5. The datapath consists of two separate pathways for arithmetic and logical operations.

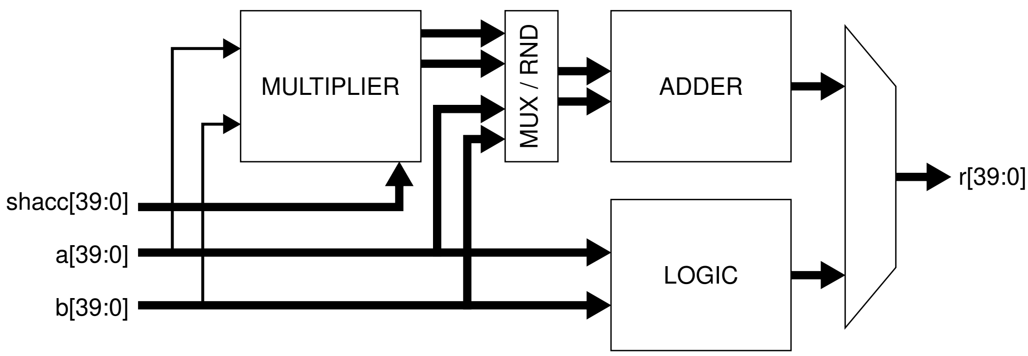


Figure 8.5 Arithmetic / logic datapath structure

Multiplication is always unsigned when using sign-magnitude number representation. The multiplier takes two 16-bit unsigned inputs and produces a redundant-representation output, and the sign is calculated separately from the sign bit of the two products. For a multiply-accumulate operation, the value of *shacc[39:0]* is added to or subtracted from the result of the multiplication depending on the relative signs of the result and the *shacc* value.

The redundant-representation result from the multiplier passes to the multiplexer / rounding stage. Depending on the operation, this selects either the outputs of the multiplier or the *a* and *b* inputs to be passed to the adder, with an appropriate offset added if rounding is to be performed.

The adder is used to convert the redundant result from the multiplier back to a positive binary value, or to perform addition and comparison operations. It is designed so that the result is always a positive binary, as is required for sign-magnitude representation. A negative result is indicated by a separate output.

The logic unit performs the standard bitwise logical operations (AND, OR, XOR). It also contains hardware to compute the Hamming distance between the two inputs, and to calculate a normalisation factor by which an input needs to be shifted to give a result whose magnitude is normalised between 0.5 and 1.0 (‘1’ in bit position 30).

At the output, the result from the arithmetic or logic function is selected. If the instruction indicates that the condition codes are to be updated, the result is evaluated to determine any changes required.

Multiplier Design

As mentioned in the introduction, multiplication can be thought of as a succession of shifts and adds. There are two basic approaches to speeding a parallel multiplier: reducing the number of additions that must be performed, and reducing the time taken to perform each addition.

- Reducing the number of additions

A 2s complement number can be written as:

$$A = (-2^{n-1})a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Each addition of a power of 2 corresponds to a shift and addition of the multiplicand. Booth [78] proposed an algorithm which reduces of the number of shifts and adds by replacing strings of 1s and 0s in the multiplier. A more practical form of the algorithm to implement in VLSI only looks at strings of three bits at a time, and is known as the Modified Booth algorithm. In this form, the multiplier value is rewritten as:

$$A = (-2^{n-1})a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i = \sum_{i=0}^{\frac{n}{2}-1} 2^{2i}(a_{2i-1} + a_{2i} - 2a_{2i+1}) = \sum_{i=0}^{\frac{n}{2}-1} 2^{2i} k_i$$

In this form, the number of additions is reduced by half. As well as being shifted, the amount to be added is multiplied by the value k_i which belongs to the digit set {-2, -1, 0, 1, 2}.

When dealing with unsigned numbers, the modified Booth algorithm may still be used. However, the assumed component $(-2^{n-1})a_{n-1}$ is incorrect: to counteract this, a value of $2^n a_{n-1}$ must be incorporated into the summation.

- Speeding the addition process

As discussed previously, the aspect of binary addition that requires the most time to complete is the resolution of carries: this is because the carry output of the most significant bit can depend on the state of the least significant and all intermediate bits. However, it is possible to defer the resolution of these carries by exploiting *redundant representations* for the intermediate values [68] [69]. These allow summations to be performed with carry propagation limited to a single bit position. Two main forms of redundant representation have been used in the design of multipliers: carry-save representation, and signed-digit representation.

Carry-save representation, as the name suggests, involves bringing the carry generated at each bit position of the adder out as a separate output. This effectively produces a redundant representation using two bits at each power of 2 which represent a value in the digit set {0,1,2}. A full adder circuit used in a carry-save adder has three inputs and two outputs, and a multiplier based around this type of carry-save array is known as a *Wallace tree multiplier*. By allowing one level of internal carry propagation, the number of inputs can be extended to produce a carry-save counter circuit with 4 inputs and 2 outputs. This has favourable properties for VLSI implementation, as it has a binary tree structure.

As with the carry-save representation, signed-digit representation uses two bits per power of two. However, in this case the digit set represented is {-1,0,1}, with one bit corresponding to +1 and the other corresponding to -1. Strictly speaking, the carry generated can propagate by two places: however, it is possible to design the addition circuit so that no further processing of the carry occurs after the first place [77]. High speed multipliers have been implemented using this type of representation, with 4:2 compression giving good layout properties.

- Choice of multiplier structure

A disadvantage of using the modified Booth algorithm when using 2s complement number representation is that generation of the negative multiples of each partial product require sign extension. This requires additional area to add the sign extension bits, and causes unwanted switching activity within the compression tree [147]. It is possible to reduce the number of sign-extension bits that must be generated using the modified sign-generate technique; however, Booth coding can still cause undesirable switching activity due to the race condition between the coding of the multiplier and arrival of the multiplicand value.

The difficulty in generating negative values for the Booth algorithm is eliminated when using signed-digit representation: generating positive or negative multiples is performed by routing the multiplicand to the positive or negative input of the signed-digit compressor, and setting the other input to zero. The circuit used to perform this function is shown in Figure 8.6.

The input signals *one* and *two* are mutually exclusive, and select either the shifted multiplicand bit *bsh* or the unshifted multiplicand bit *b* to perform the multiplication by two or one. To prevent activity on the input bus *b[39:0]* from causing power consumption in the compression tree and to exploit correlations between successive inputs fully, it is desirable to latch the partial product values at the input to the tree. This function is incorporated into the positive / negative multiplexing component of the circuit: when the

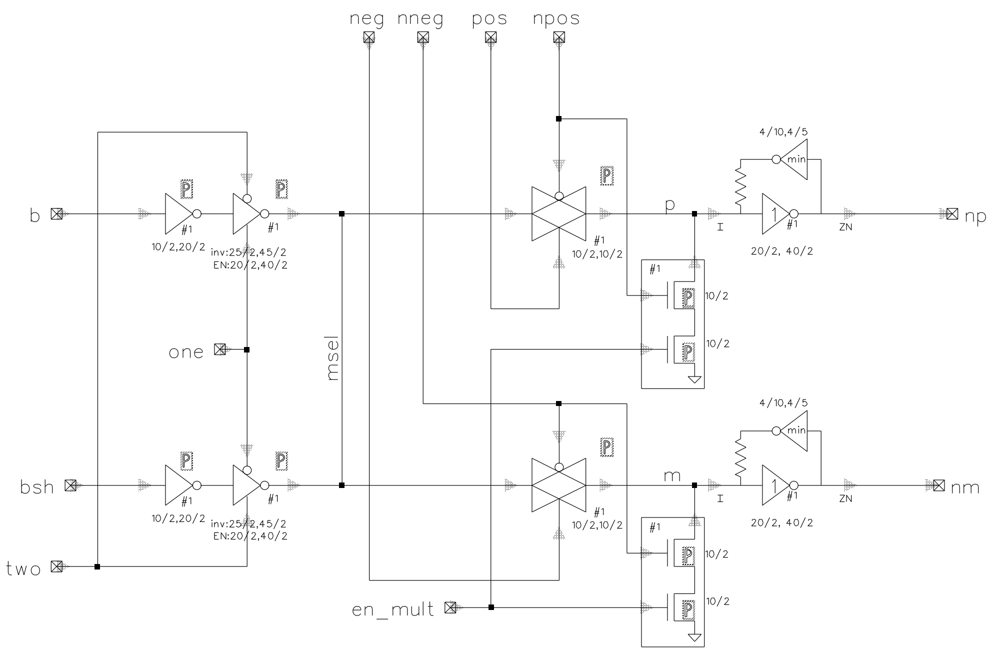


Figure 8.6 Signed digit Booth multiplexer and input latch

multiplier value has been processed, *neg* and *nneg* or *pos* and *npos* are asserted to switch on the appropriate transmission gate, along with *en_mult* to clear the other output. Between operations, the weak feedback inverter maintains the value stored at the output of the transmission gate. This method for generating the partial product values also avoids unnecessary activity caused by the race between the multiplier and multiplicand.

The signed digit adder circuit used to implement the compression tree was based on that proposed in [77]. However, instead of the proposed static CMOS implementation, a pass-transistor based implementation has been developed, with the aim of producing a more regular layout.

The compression tree of the multiplier has the structure shown in Figure 8.7. The first 4 stages combine the partial products produced by the Booth coding. The final stage combines this value with the offset required for unsigned operation and any accumulation value to be added to the product.

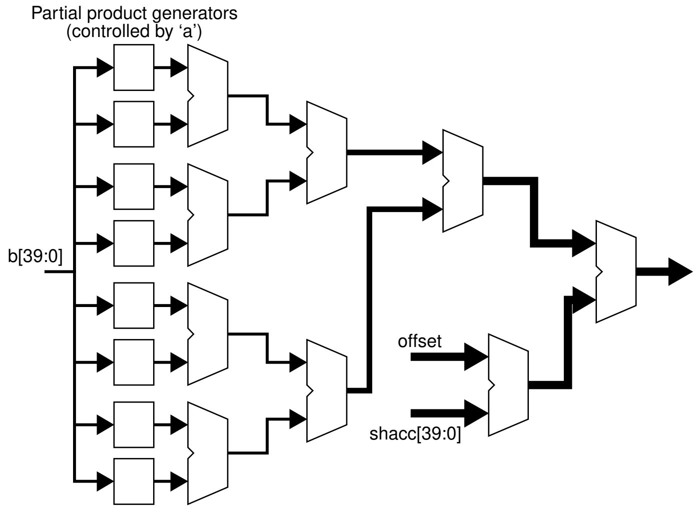


Figure 8.7 Multiplier compression tree structure

Input Multiplexer and Rounding Unit

An important part of DSP operation is rounding, to minimise the error when converting from the 40-bit extended precision accumulator quantities back to the 16-bit register and memory precisions. This is performed by adding 0.5 LSB, and truncating the result to 16 bits. All of the arithmetic operations (add, multiply and multiply-accumulate) support rounding. Since this is effectively another addition, it can be performed using the same type of redundant signed digit adders that are used for the multiplier, as a pre-processing step before the final adder.

One drawback of using sign-magnitude numbering is that it is necessary to make the sign of the value to be added the same as the sign of the final result: in 2s complement representation, the same value can be added regardless of the sign (although problems of bias in rounding exact half values do then occur). Since the sign of the result is not known before the result is calculated, rounding operations speculatively add a positive value. If the final result proves to be negative, the addition is repeated with a negative value. The extra addition should only be necessary in about half of the cases, and rounding is a relatively infrequent operation that is only performed at the end of processing a block of data.

Before the rounding is performed, the appropriate values are multiplexed onto the inputs. For a multiply or MAC operation, the redundant output is fed directly to the redundant adder performing the rounding. For an addition, the input values (*a[39:0]* and *b[39:0]*) are fed to the redundant adder. Because these numbers are in sign-magnitude form, different operations must be performed depending on the relative signs of the inputs. If the signs of the inputs are the same, then addition is performed. *a[39:0]* is fed to the positive input of the redundant adder, while *b[39:0]* is negated and fed to the negative input. If the signs of the inputs differ, then subtraction is performed. *a[39:0]* is again fed to the positive input of the redundant adder, while *b[39:0]* is this time fed *directly* to the negative input.

Add with carry is also implemented by this stage. Sign-magnitude representation makes the meaning of carry out differ from conventional meaning in 2s complement representation. Positive and negative carries are possible, with the decision being based on the sign of the result that set the carry flag. The redundant adder used to perform rounding also adds these carry values.

Adder Design

The adder takes the redundant value from the output of the rounding unit, and converts it back to binary form. Parhi [145] proposes a class of multiplexer-based adders which convert from this redundant form back to binary, and presents a methodology for selecting the architecture that consumes the least power for a given delay.

In this case the objective was to achieve the minimum delay since the multiply operation is on the critical path of the processor. A 3-way carry resolution circuit (considering carry signals from 3 bit positions) has been developed as part of the AMULET3 processor [125]. Not only is this circuit very fast, it also resolves 3 carry inputs per stage rather than 2. This means that only 4 carry resolution stages are required, rather than the 6 stages required if 2-way resolution is performed. The only drawback is that the circuit is pseudo-dynamic and requires a precharge phase, causing undesirable power consumption. However, since layout cells for the adder were available and time was limited, this was felt to be a reasonable compromise.

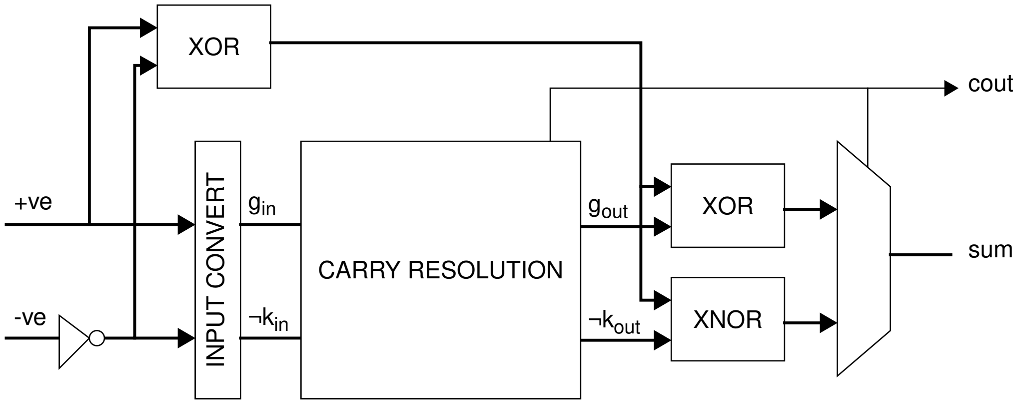


Figure 8.8 Late-increment adder structure

The redundant-representation input to the adder consists of the positive and negative components: the negative value is inverted to produce a 2s complement input to the adder. The input values are converted into carry generate and kill signals for carry resolution and at the same time the inputs are XORed to produce the sum at each bit position before carries are determined. The carry resolution tree then calculates carry generate and kill signals at each bit position.

The adder performs the operation $A - B$ by performing $A + \bar{B} + 1$, with the one being added using the carry input. The result of this operation may be a positive or negative 2s complement number. However, sign-magnitude numbering requires a positive result from the adder. This is achieved by 'late negation' of the result; inverting the sum without a carry input to give $\overline{A + \bar{B}} = -(A - B)$.

The generate and ¬kill (*not* kill) signals after the final stage of carry resolution correspond to the carry input at each bit position for zero and one carry input respectively. A high value of generate indicates that this bit has a carry in regardless of the carry into the least significant bit. A high value of ¬kill indicates that either a carry has been generated affecting this bit position, or that carries are propagated all the way from the least-significant bit. This means that the generate and ¬kill values at the end of carry resolution may be used to calculate the sum either with or without a carry input, as required by late negation.

A carry output from the most-significant position indicates that the result was positive. In this case the sum $A + \overline{B} + 1$ is performed as normal: the output is produced by XORing the sum at each bit position with the ¬kill values, the result corresponding to that with a carry into the LSB of 1. However, if no carry output is generated then the output of the adder is negative, and must be negated. In this case, the output is produced by XNORing the sum at each bit position with the carry generate signal.

In both cases therefore, the result is a positive value. The sign of the result is determined by considering the sign of the inputs and whether the output of the adder was negated or not.

Logic unit design

The structure of the logic unit is shown in Figure 8.9. Other than the conventional AND, OR and XOR operations, this unit can also calculate the Hamming distance between the inputs, and the shift required to normalise the *a* input.

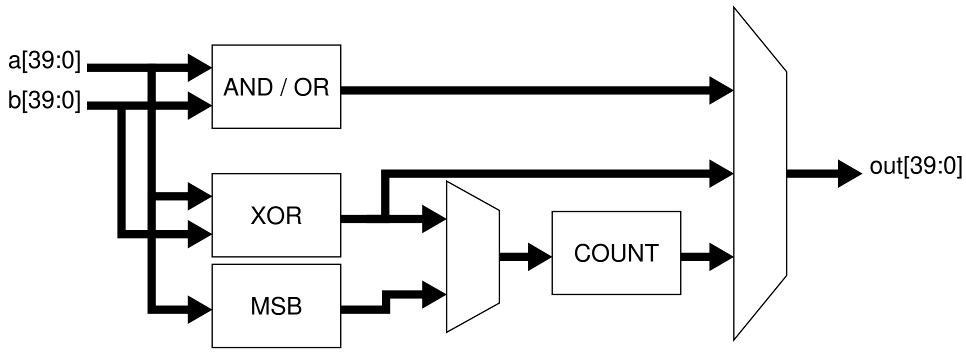


Figure 8.9 Logic unit structure

- Distance calculation

The distance metric is calculated by first XORing the two inputs together, to determine those bits which differ. The result then passes to a bit counter, implemented by a tree of 3:2 and 4:2 carry-save counters, which counts the number of 'ones'. The carry-save output of this counter tree is then converted to binary using a 5-bit ripple carry adder to give the total number of differing bits.

- Normalisation

The normalisation factor is simply the number of bits between the most-significant set bit of the input and the normalisation position (bit 30). To calculate the normalisation shift distance and direction, the *a* input is pre-processed to convert all of the bits between the most-significant bit of the input and the normalisation position to ‘1s’. All other bits are forced to zero by the pre-processing step. The result of this process is then passed to the same bit counter used for distance metric calculation.

The direction of shift is determined by whether the extension portion of the input (bits 31-39) are set or not. The direction is appended to the result as the sign, to be used in a shift instruction. To distinguish between the input cases of zero and an already-normalised value, which would otherwise both produce a zero result, a non-zero input is indicated by setting bit 6 of the result. This does not affect the subsequent shift operation, as this depends only on bits 0-5, but causes the zero flag to be cleared.

Chapter 9: Testing and evaluation

As discussed in Chapter 4, the design flow for CADRE involved the progressive replacement of C models with circuits. At the time of writing, all of the datapath components of the processor have been designed as gate or transistor-level schematics. The majority of the control circuits have also been mapped onto schematics. The control circuits in the functional units, register bank, configuration memories, index generation units, fetch unit and instruction buffer are all fully represented by schematics. In the decode stage, the control circuits in the first stage of decoding (involved with all operations) have also been mapped. The design in its current state contains over 750,000 transistors.

The control circuits that remain in the form of C models are those associated with control and setup instructions, and those in the load / store unit. It is felt that the absence of these control units will not affect the overall power consumption very much as these are used relatively infrequently. Furthermore, where a control circuit drives a significant load (and thereby may consume significant amounts of power), buffers are placed between the control circuit and the load. The power consumed by the buffers will be accurately reported, and the 'missing' power should be small in comparison.

9.1 Functional testing

Before performance could sensibly be evaluated, it was necessary to test that the processor was functioning as expected. To this end, a set of programs was developed to perform tests of incrementing complexity. These tests were not intended to be of production level, but were intended instead to give reasonable confidence that the processor was operating as intended; particularly for the tasks that would be required by later tests. The set of tests and their functions are listed in Table 9.1. Tests were run using the Timemill simulator on netlists extracted from the schematics. The environment of the DSP (program and data memories, and control signals) was emulated using C behavioural models. At the end of simulation, the contents of the memories were dumped to files, and the output checked with the expected results. For the more complex tests (fir20, mmfft64) the expected results were generated by using C implementations of the algorithms, designed to mimic the arithmetic precision and rounding functions of CADRE.

Name	Function
store0	Checks parallel execution and store to memory.
store1	More complex test of stores.
store2	Test of store long from registers.
store3	Test of store long from GIFU.
store4	Mixed GIFU / register store.
load0	Tests short load to registers.
load1	Tests long load to registers.
load2	Tests combined load and store.
areg0	Tests basic moves to address registers and immediate adds.
areg1	Tests basic address register updating.
ireg0	Tests move-multiple to index registers, and use of index registers as specification for writeback target and store source.
ireg1	Tests index register updating, and use of index register to specify load destination.
branch0	Basic test of JMP instruction.
branch1	Basic test of JSR and RTS.
branch2	Basic tests of BRACC, with NV / AL conditions.
branch3	Basic test of BSRCC with NV/ AL conditions.
do0	Tests simple immediate DO, use of circular buffers and *nfirst* condition for store.
add	Simple test of ADD operation and limiting.
mult	Simple test of MPY functions.
logsh	Test of logic functions and shifting.
minmax	Test of MAX and MIN functions, and condition code setting.
divide	Test of Newton-Raphson division algorithm used in the Schur recursion section of the GSM speech coder.
fir20	Twenty-point FIR filter run on a block of 80 random samples.
mmfft64	64-point complex FFT.
schur	Schur Recursion from GSM speech coder.

Table 9.1: Functional tests on CADRE

9.2 Power and performance testing

Once correct operation of the CADRE had been established, it was possible to perform tests to establish the performance of the DSP in terms of power consumption and processing throughput. This was performed using three test algorithms: a 20 point FIR filter, a 64-point complex FFT and the preprocessing and linear predictive coding (LPC) analysis section of the GSM full-rate speech compression algorithm. The FIR filter and FFT each processed 256 data samples, while the LPC analysis algorithm was performed on a GSM data frame of 160 samples. To evaluate the impact of data characteristics on power consumption, the FIR filter and FFT algorithms were run separately on random data and speech data (extracted from the ETSI speech test sequence used for testing GSM codecs). The LPC analysis algorithm was run only on speech data.

The Powermill circuit simulator was used to run the tests: this has the same timing accuracy as Timemill, and also records power consumption. Powermill is claimed to be close to SPICE in its accuracy, at a fraction of the computational load. Power consumption probes were assigned in a hierarchical manner, to provide a breakdown of the power consumed by the various segments of the design.

In a complete system, the memory power consumption may be a significant proportion of the total power consumption. In the simulations, the memories were implemented using C behavioural models. To estimate memory power consumption, the models were designed to report power consumption to the simulator during each read or write access, so as to consume a fixed amount of energy for each operation. The energy per operation was estimated at 0.67nJ, which was based on measurements of power consumed by the 8 kilobyte RAM block of the AMULET3i asynchronous embedded island [148].

9.2.1 Recorded statistics

A number of C models were included in the simulation for the purpose of monitoring various aspects of the operation of CADRE. The collected statistics were used as an aid to assessing performance of the device as a whole, and estimating the impact of the various architectural features.

Operating speed and functional unit occupancy

To monitor the rate of parallel instruction issue and level of activity of the functional units, a C model was designed to monitor the *req_op* / *ack_op* handshake to the functional units, along with the 4 bundled *en_op* signals. On the first handshake, the start time was recorded. On all subsequent handshakes, the number of parallel instructions per second were recorded, and the enable signals to the functional units were used to calculate and record the number of actual operations performed per second. These figures measure the *actual* performance including overheads such as setup instructions.

Memory and register accesses

The C model for the memory models was designed so that details of each memory access were recorded to a log file. Similarly, a C model was written to monitor register bank accesses and write details to a log file. Not only did this allow the number and type of accesses to be analysed after a simulation, it also allowed graphical monitors to be written (using Perl with Tk graphical extensions) allowing the contents of the registers and memory to be viewed *during* simulation, which aided debugging of algorithms.

Instruction issue

To allow the effect of the instruction buffer to be assessed, a C model was designed to monitor instructions arriving from the buffer at the decode stage. The number of decoded instructions was counted, and could be compared with the number of instructions fetched from program memory.

Address register and index register updating

C models were written to monitor the number and type of updates performed on the address and index registers. This allowed the relative number of address and index register updates to be assessed and, by combining these with the power consumption figures, the benefit gained from using the index registers to be estimated.

Register read and write times

To evaluate the effectiveness of the split register file architecture, timing information was collected for reads and writes to the register bank. 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. For the purpose of testing the register bank, two additional test programs were executed. These tests performed reads and writes respectively with varying degrees of contention for a single sub-bank.

9.3 Results

9.3.1 Instruction execution performance

Operating speed results for the three algorithms are shown in Table 9.2. This shows the rate of issue of parallel instructions, the operation rate within the functional units, and the average proportion of the functional units which are occupied for each parallel instruction.

Test	Instruction rate	Arithmetic operation rate	Occupancy
FIR filter	43MHz	163MOPS	95%
FFT	38MHz	141MOPS	93%
LPC analysis	34MHz	117MOPS	86%

Table 9.2: Parallel instruction issue rates and operations per second

The instruction rate is the measured rate of dispatch of parallel instructions to the functional units. This value depends on how many control / setup instructions had to be inserted between parallel instructions, and also on how quickly register reads and arithmetic operations completed. The arithmetic operation rate is the measured rate of arithmetic operations within the functional units, which depends on the instruction rate and the occupancy (how frequently each functional unit is used in parallel instructions).

It can be seen that the operation rate for the FIR filter exceeds the 160 MOPS target: the FIR filter kernel is extremely efficient, without any setup code required once the kernel is underway. The FFT algorithm is somewhat less efficient, requiring changes to the index and update registers between successive passes of the FFT kernel. Since the speed of arithmetic operations is not data dependent, the same operation rates were observed for both speech and random data. The GSM LPC analysis program is the least efficient, as the test involves a number of separate algorithms applied sequentially which require setup instruction between each pass. Also, some of the algorithms cannot be partitioned easily across the functional units. This is evident in the reduced utilisation figure.

9.3.2 Power consumption results

	FIR		FFT		GSM
	random	speech	random	speech	
Power consumption	**668mW**	**584mW**	**676mW**	**660mW**	**406mW**
Run time (μs)	**38.9**	**38.5**	**32.7**	**32.5**	**16.1**
Arithmetic ops.	**5888**	**5888**	**4100**	**4100**	**1288**

Table 9.3: Power consumption, run times and operation counts

Average power consumption for each of the algorithms is shown in Table 9.3. The run time over which power consumption is measured extends from the moment that the reset signal is removed to the time that the *nHalt* signal is asserted. The average power figures measured include the period when the configuration memories are being written to immediately after reset. The average power during the configuration process will be low, and will cause the reported average power to be less than the average power when actually performing arithmetic processing. Two different techniques have been used to calculate metrics for energy per arithmetic operation from the power consumptions, which deal with this error in different ways and allow bounds on the true figure to be set.

The first technique, used to calculate the bulk of the figures in Table 9.4, is to use the measured figures for operating speed, and to divide the power consumption by the number of operations per second. This measure incorporates the energy consumed by control and setup instructions during kernel execution, but does not take into account the reduced power consumption during the configuration period. This metric will therefore be an underestimate of the true energy consumed during operation, but will asymptotically tend toward the correct value as the run time is increased and the kernel power consumption comes to dominate. The figures displayed are rounded (some to zero), leading to a slight discrepancy for the totals which are calculated from unrounded values.

The second technique is to calculate the sum total of energy consumed during the simulation, by multiplying the average power consumption by the run time. The energy per operation can then be found by dividing the total by the number of operations. This will be an overestimate of the true energy per operation when running the kernel, as it includes the energy for the configuration process which would normally only be consumed once, and is marked as 'worst-case' in Table 9.4. However, this result also asymptotically tends toward the true value as the run time is increased. The LPC analysis algorithm has the shortest processing time compared to the configuration time, leading to a larger difference between the higher and lower estimates than observed for the other benchmarks.

	FIR		FFT		GSM	Avg.	% total
	random	speech	random	speech			
Instruction fetch	0.00	0.00	0.01	0.01	0.01	0.01	0.2%
Instruction decode	0.02	0.02	0.03	0.03	0.04	0.03	0.7%
Data memory	0.03	0.03	0.08	0.08	0.03	0.05	1.2%
Program memory	0.02	0.02	0.1	0.10	0.14	0.08	1.9%
Instruction buffer	0.06	0.06	0.1	0.10	0.11	0.09	2.1%
Index update	0.10	0.1	0.11	0.11	0.05	0.09	2.2%
Address update	0.17	0.17	0.22	0.22	0.33	0.23	5.4%
Register bank	0.34	0.32	0.5	0.50	0.15	0.36	8.7%
Config. memories	0.94	0.94	1.02	1.03	0.89	0.96	23.2%
MAC units	2.33	1.81	2.43	2.31	1.58	2.09	50.5%
Remainder	0.12	0.12	0.21	0.21	0.14	0.16	3.9%
Total	**4.2**	**3.6**	**4.8**	**4.7**	**3.5**	**4.2**	
Total (worst-case)	**4.4**	**3.8**	**5.4**	**5.2**	**5.1**	**4.8**	

Table 9.4: Distributions of energy (nJ) per arithmetic operation

It is easier to see the distribution of energy to the various portions of the processor from Table 9.4 when it is shown graphically as in Figure 9.1. It can be seen that the dominant sources of power consumption are the multiply-accumulate units, and a breakdown of the power consumption within one of them is depicted in Figure 9.2. The multiplier dominates, followed by the adder, as might be expected. The next most significant source of power consumption is the input multiplexer / rounding unit at the input to the adder: this is somewhat unexpected, as this implements only a small amount of functionality, and the adder does not present a very heavy load on this unit. This is due to the multiplier compression tree is producing a large number of spurious transitions when summing the partial products, causing the increased activity and relatively high power consumption within the rounding unit.

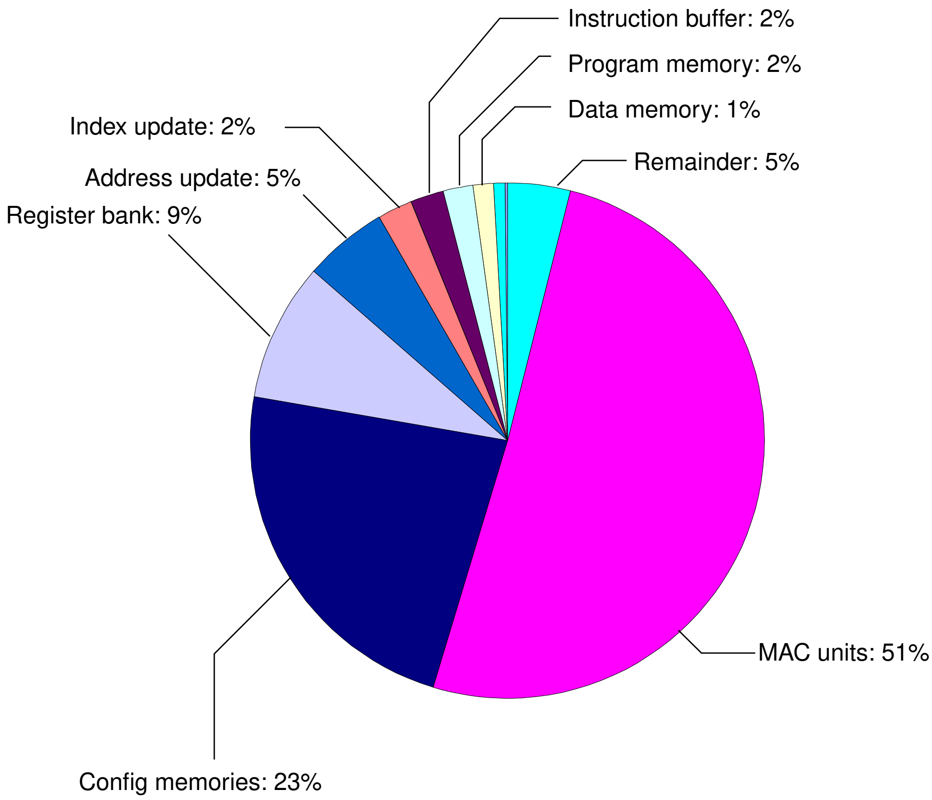


Figure 9.1 Average distribution of energy per operation throughout CADRE

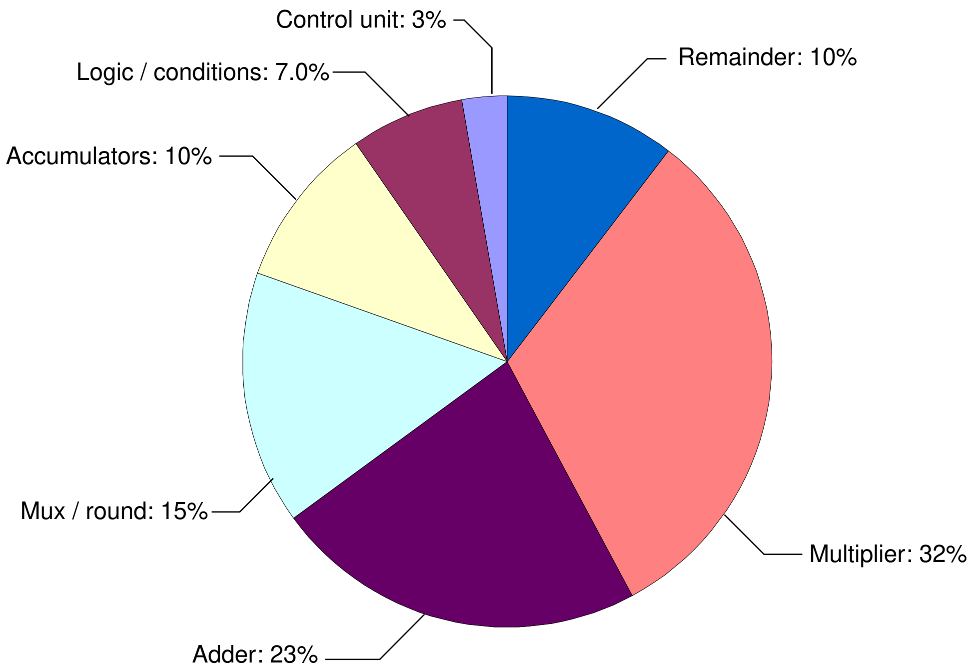


Figure 9.2 Breakdown of MAC unit power consumption

9.3.3 Evaluation of architectural features

Register bank performance

- Read timing

As stated earlier, tests were performed by varying the number of different contending read requests to a sub-bank. The maximum read times required to access data in each case is shown in Table 9.5. 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.

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	
9	69ns	

Table 9.5: Read and write times with different levels of contention

- Write timing

The measured worst case write cycle times for each level of conflict are shown in the right-hand column of Table 9.5. It can be seen that the time per write does not increase in proportion to the number of writes, with the incremental increase reducing somewhat. This is due to other requests propagating further through the arbiter tree while the first write requests are serviced, reducing subsequent write times.

- Performance for DSP algorithms

The average, minimum and maximum read and write cycle times for the different DSP algorithms are shown in Table 9.6. 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 read requests 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 despite the pathological cases having the highest read time.

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

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 9.6: Register access times for DSP algorithms

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.

- Power consumption results

Energy consumption figures per parallel instruction when running the test algorithms are given in Table 9.7. Figures are presented for the whole system and for just the register bank. 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.

The number of register bank accesses are measured at the sub-banks: this means that, where a number of read ports require access to a single register, only a single read is recorded. This can lead to an underestimate of the total number or reads required by a

Algorithm	Total energy	Register bank				Data memory
		Energy per instr.	Accesses	Accesses per instr.	Energy per access	Accesses
FIR filter (random)	4.15nJ	0.34nJ	11620	1.9	0.18nJ	556
FIR filter (speech)	3.59nJ	0.32nJ	11620	1.9	0.17nJ	556
FFT (random)	4.81nJ	0.5nJ	8032	1.8	0.28nJ	1096
FFT (speech)	4.84nJ	0.51nJ	8032	1.8	0.28nJ	1096
LPC analysis	3.47nJ	0.15nJ	1004	0.7	0.21nJ	180
averages	*4.17nJ*	*0.36nJ*	-	-	*0.22nJ*	-

Table 9.7: Energy per parallel instruction and per register bank access

particular algorithm, but does give a faithful indication of the energy cost of performing read cycles.

- Effect of split register architecture

It can be seen that, averaged over the different runs, the register bank consumes 9% of the total energy per operation. The register bank consumes decreasing amounts of energy per access for the FFT, LPC analysis and FIR filter algorithms respectively: this corresponds to how efficiently the algorithms make use of the register sub-bank interleaving.

If it is assumed that power consumption of register banks increases in proportion to the square of the number of ports as suggested in [142], 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 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.

When the number of accesses to memory are compared with the number of accesses to the register file, the benefit of using the register file becomes clear: the register bank is accessed between 6 and 21 times more frequently than the memory, and the register bank consumes on average 3 times less energy per access than the memory system.

A direct comparison of power consumption with and without the register bank is difficult: the lack of a register bank would require a radically different system architecture and programming style, but it is likely that the net effect would be for the memory system to consume still more energy per access. Also, the size and location (and hence energy consumption) of the memory system will be very dependent on the type of system into which CADRE is incorporated.

Overall, it is clear that the choice of a register bank and its architecture in CADRE contributes significantly to the power reduction. The DSP architecture is heavily optimised to reduce power consumption, so the fact that data accesses (including those to main memory) only make up around 10% of the total system energy per operation indicates how effective the register file architecture is. The proportion of the power consumption is half of the 20% of power dissipated by data accesses in the Hitachi DSP for which a similar breakdown is available [136]. In a full layout simulation, the effects of the interconnections will increase the proportion of power consumed by the register bank somewhat. However, the power consumed by the rest of the system will also increase, particularly the cost of accessing the data memories.

Use of indexed accesses to the register bank

The energy consumptions by each update operation within the index generation units and address generation units are presented in Table 9.8. The energy consumptions were calculated by determining the total energy for the runtime and dividing it by the number of updates.

The benefit of using the index registers can be seen clearly, from the number of updates alone: between 8 and 22 times more updates are performed using the small index generation units rather than the address generation units. An indication of the relative costs of each update can be seen from the calculated figures. However, some caution should be used when comparing these: the total energy calculated includes that for

Algorithm	Index registers		Address registers	
	Updates	Energy per update	Updates	Energy per update
FIR filter (random)	6178	0.10nJ	278	3.9nJ
FIR filter (speech)	6178	0.10nJ	278	3.9nJ
FFT (random)	2620	0.19nJ	348	2.9nJ
FFT (speech)	2620	0.19nJ	348	2.9nJ
LPC analysis	579	0.16nJ	50	12.5nJ

Table 9.8: Energy per index and address register update

instructions when no updates were required, which causes the energy per update to be overestimated. For the index registers, this effect is small since updates occur frequently. However, the address registers are only rarely updated, so the total energy requirement for each update is significantly overestimated.

Effect of instruction buffering

Results for the relative number of instructions passing from the instruction buffer and program memory are presented in Table 9.9, along with the calculated energy consumed per instruction issued by the buffer. To estimate the effect of the instruction buffer during kernel execution, the size of the configuration data block (which must be read from memory and passed through the instruction buffer) is given, and the number of executed instructions and instructions fetched from memory are presented both with and without this contribution. To assess accurately the energy consumed by an instruction passing through the buffer, the total number of issued instructions *including* the configuration data is used to calculate the energy.

The measured energy per instruction passing through the buffer is between 32% and 45% of the estimated energy required to fetch a word from program memory. The measured ratio of instruction issued by the buffer to those fetched from memory varies from between 2.7 and 22. This ratio depends on how efficiently a given algorithm makes use of the DO construct, and how many instructions are prefetched from a branch shadow and discarded. In order to give the most efficient parallel mapping of the instructions, the FFT

Algorithm	Config. data block size	Instruction buffer		Memory fetches
		Executed instructions	Energy per instr.	
FIR Filter (random)	112	1746 / 1634	0.22nJ	187 / 75
FIR Filter (speech)	112	1746 / 1634	0.22nJ	187 / 75
FFT (random)	188	1633 / 1445	0.27nJ	714 / 526
FFT (speech)	188	1633 / 1445	0.27nJ	714 / 526
LPC analysis	278	684 / 406	0.31nJ	399 / 121

Table 9.9: Instruction issue count and energy per issue for the instruction buffer

algorithm could only make limited use of DO loops. The same was true to a lesser extent of the LPC analysis algorithm. However, the FIR filter was sufficiently regular to allow efficient looping.

It can also be seen that the energy per instruction is higher when less efficient looping is possible: this is due to the increased energy to write and then read an instruction when compared with writing once and reading many times. Also, for the LPC analysis results, the number of configuration words passed is large in comparison to the total. This obscures the effect of the DO loops to some extent.

Effect of sign-magnitude number representation

The power consumption figures for the FIR filter algorithm show a total reduction of 13% when processing speech data rather than full-range random data. The figures for the FFT algorithm performed on comparably scaled speech data show a reduction of only 1%: the FFT algorithm is such that adjacent data points tend not to be processed sequentially, reducing the amount of correlation that can be exploited. The energy per operation performing the LPC analysis algorithm on speech are even lower than the energy for the FIR filter or the FFT. However, a direct comparison is impossible between speech and random data in this case, as the LPC analysis algorithm is ineffective for random data and exits early.

For the FIR filter, the greatest reduction in energy for each operation is, as could be expected, in the multiply / accumulate units where speech data causes 23% less energy consumption than random data. A reduction of 7.7% is caused in the register bank. For the FFT, the reductions are 6% and 2% respectively. However, the wiring capacitance is not incorporated in the simulations and the memory energy consumption is independent of the data pattern. The overall difference in power consumption for a full simulation incorporating these factors, or for tests run on a fabricated processor, is likely to be significantly greater.

9.4 Comparison with other DSPs

9.4.1 Detailed comparisons

Direct comparison with DSPs developed by other groups or commercial manufacturers is difficult: for a fair comparison, the same algorithms must be performed on the same data and, if architecture and circuit structures alone are to be compared, on the same process. At the very least, the same algorithms must be executed. One such comparison has been performed to evaluate the P1 test chip of the Pleiades reconfigurable processor architecture [149]. For this comparison, the chosen benchmarks were FIR filters, FFTs and IIR filters. It was decided to compare CADRE with these figures, on the same basis.

The P1 chip was fabricated using a 0.6µm process, and tested at 1.5v. To make a meaningful comparison, all tests in [149] were normalised to these conditions. Gate capacitance was assumed to represent circuit capacitance:

$$\text{capacitance} \propto \frac{A}{T_{ox}} \propto \frac{L^2}{T_{ox}} \tag{21}$$

where L is the minimum channel length, and T_{ox} is the gate oxide thickness assumed to be proportional to the native supply voltage of the process being considered.

Delay was normalised according to the gate capacitance and the saturation drive current:

$$\text{Delay} = \frac{CV}{I} \propto \frac{L^2 V}{(V - V_{th})^{1.3}} \tag{22}$$

where V is the supply voltage and V_{th} is the threshold voltage.

The process parameters presented in the paper, and those of the 0.35µm process on which CADRE is implemented, are shown in Table 9.10. Native V_{dd} is the standard supply voltage for the process technology, while test V_{dd} is that used to perform the tests on which the normalised figures are based. Energy and delay results for FIR and FFT benchmarks are compared in Table 9.11 and Table 9.12. Energy per tap or per FFT stage was calculated directly, by averaging the best and worst case figures for both speech and random data and determining the number of operations required in each case. Delay for each benchmark was calculated from the number of operations required and the average operation speed measured for that benchmark. Although it was not stated in the paper, the FFT benchmark appears to be for a single pass of a 16 point FFT (8 butterfly operations per pass). The optimised FFT kernel for CADRE performs 4 butterfly operations in 6 parallel instructions.

				V_{dd}		Cap. coeff.	Delay coeff.
Processor	L_{min}	T_{ox}	V_{th}	native	test		
Pleiades	0.6µm	9nm	0.7V	3.3V	1.5V	1.0	1.0
StrongARM	0.35µm	6nm	0.35V	1.5V	1.5V	1.96	4.7
TMS320C2xx	0.72µm	14nm†	0.7V†	5.0V	3.0V	1.1	1.37
TMS320LC54x	0.6µm	9nm†	0.7V	3.3V	3.0V	1.0	1.97
CADRE	0.35µm	9nm†	0.7V	3.3V	3.3V	2.9	6.2

Table 9.10: Fabrication process details from [149], and those for CADRE (estimated values marked with †)

The chosen metric for comparison is the energy-delay product. It is almost always possible to reduce energy by reducing speed (e.g. by reducing supply voltage), but to reduce both simultaneously requires improvements to the underlying design. For the results presented in Table 9.11 and Table 9.12, it is clear that CADRE is very much faster than the other processors, but does not have, on average, an advantage in terms of energy per arithmetic operation.

Processor	Strong-ARM	TMS320 C2xx	TMS320 LC54x	Pleiades	CADRE
Delay per tap	101ns	50ns	25ns	71ns	5.8ns
Energy per tap (nJ)	21.1	4.8	2.4	0.205	4.0
Capacitance / tap (pF)	9380	530	270	91	367
Capacitance / tap (pF) @ 0.6µm	16600	580	270	91	1064
Energy / tap (nJ) @ 0.6µm, 1.5v	37.4	1.3	0.6	0.2	2.4
Delay / tap @ 0.6µm, 1.5v, V_{th}=0.7v	475ns	68.5ns	49.3ns	71ns	36ns
Energy*Delay per tap (J.s.10^{-17}) @1.5v,V_{th}=0.7v	1760	8.9	2.9	1.5	8.6

Table 9.11: FIR benchmark results

Processor	Strong-ARM	TMS320 C2xx	TMS320 LC54x	Pleiades	CADRE
Delay per stage	4533ns	7600ns	1900ns	571ns	316ns
Energy / stage (nJ)	1040	478	197	13.3	245
Capacitance / stage (pF)	462	53.1	21.9	5.91	22.5
Capacitance / stage (pF) @ 0.6µm	831	58.4	21.9	5.91	65.3
Energy / stage (nJ) @ 0.6µm, 1.5v	1870	131	49.3	13.3	147
Delay / stage @ 0.6µm, 1.5v, V_{th}=0.7v	21µs	10.5µs	3.75µs	571ns	1.96µs
Energy.Delay per stage (J.s.10^{-14}) @1.5v,V_{th}=0.7v	3970	137	18.5	0.759	28.8

Table 9.12: FFT benchmark results

In the case of the FIR benchmark, the normalised energy-delay product for CADRE is a little lower than the Texas Instruments C2xx processors, but is approximately 3 times greater than the C54x processors and 5.7 times that of the Pleiades P1 chip.

In the case of the FFT benchmark, the normalised result for CADRE is only 1.6 times poorer than the Texas Instruments C54x processor, and is 4.8 times better than the C2xx.

This relative improvement indicates the advantage of the highly configurable architecture, which allows a very efficient partitioning of the FFT operations across the parallel resources: the FIR filter results are less good in comparison, as it is a simpler algorithm which under-utilises the capabilities of CADRE.

The results presented for the Pleiades architecture are somewhat better than CADRE, which is to be expected as Pleiades is more akin to a reconfigurable ASIC than a true processor and is significantly less programmable.

The normalisation to 0.6µm severely affects CADRE's results: when this normalisation is not performed, the results of the comparison are very different. CADRE has an energy-delay product which is 2.6 times lower than the TMS320LC54 for the FIR filter benchmark and 4.8 times lower for the FFT benchmark.

9.4.2 Other comparisons

Sources of data about other DSPs are very much less complete: usually, all that are quoted are headline figures for peak rate of operation and power consumption. CADRE is unlikely to appear favourably in such a comparison, since its performance is best when exploiting complex algorithms as shown by the FFT benchmark.

The optimal energy delay product for CADRE, as calculated from (21) and (22), occurs when operating at approximately 1.2v. At this voltage, the operating speed is reduced by a factor of 3.1 and the energy per operation is reduced by a factor of 7.6 from the values measured at 3.3v. Using the figures from the FIR filter as the basis for comparison gives a peak operating speed of 55 MOPS (43MHz / 3.1 over 4 functional units), at a power consumption of 29mW (based on the average of the energy per operation results for random and speech data). The equivalent to the energy-delay metric with these figures is milliwatts per MOP^2: energy per operation is power divided by rate of operation, while delay per operation is the reciprocal of rate of operation. For CADRE, this results in a figure of 9.6×10^{-3} mW/MOP^2. There follows a comparison with the data available about other 16-bit fixed point DSPs.

OAK / TEAK DSP cores

The datasheets for the OAK and TEAK DSP cores [151], produced by DSP Group Inc., are implemented on 0.6μm and 0.25μm technologies respectively. The OAK datasheet claims a headline current consumption of 25mA at 3.3v and 80MHz, corresponding to 82.5mW. This gives an energy-delay metric of 12.8×10^{-3} mW/MOP^2. The TEAK datasheet claims a headline current consumption of 0.45mA/MHz, a peak performance of 130MHz, and a minimum supply voltage of 1v: however, it does not state at which voltage the current and performance were measured and therefore no power consumption information can be inferred.

Texas Instruments TMS320C55x DSP

The new Texas Instruments processor range [129] claims headline performance figures of 0.05mW/MIP and speeds up to 400MHz. No details of the process technology are given, but it is presumably 0.18μm or below. The real meaning of the figures presented in the technical brief are far from clear, and product data sheets indicate that power consumption measurements from a fabricated product are yet to be made, but an energy-delay metric of 0.17×10^{-3} mW/MOP^2 is suggested.

Cogency ST-DSP

The Cogency ST-DSP [152] was a commercial self-timed DSP intended for low power and low noise operation. This was implemented on a 0.6μm CMOS process operating at 5v. Current consumption running a fax/modem application with a nominal operation rate of 30MHz was 81mA, corresponding to a rather poor energy-delay metric of 450×10^{-3} mW/MOP^2.

Non-commercial architectures

A number of DSP architectures are presented in the academic literature that are either not developed by established companies or are research architectures.

Lee et al. [153] present a 1v programmable DSP for wireless communications that uses a variety of circuit techniques and clock gating to minimise power consumption. The device was built in 0.35μm technology using dual threshold voltages to enable fast low voltage operation without excessive static power consumption. Figures reported in the paper give an energy-delay metric of 4.2×10^{-3} mW/MOP^2.

Igura et al. of NEC's ULSI Research Lab present a 1.5v 800 MOPs parallel DSP [154], intended for mobile multimedia processing. This architecture uses 4 independent DSP cores with both local and shared memories, and was built in 0.25μm technology. When run at 1.5v and 800MOPs, a power consumption of 110mW is reported, corresponding to an energy-delay metric of 0.17×10^{-3} mW/MOP^2.

Recently, Ackland et al. presented a multiprocessor DSP core that could perform a prodigious 1.6 billion 16 bit multiply accumulate operations per second [155]. This was built in 0.25μm technology, and a power consumption of 4W was reported when running with a supply voltage of 3.3v. This corresponds to an energy-delay metric of 1.6×10^{-3} mW/MOP^2. However, it is likely that this figure could be improved by operating at a reduced supply voltage.

9.5 Evaluation

CADRE has performed reasonably in the overall comparisons: detailed comparison with the Texas Instruments TMS320LC54x core (which was their current low-power device at the time this project began) shows that CADRE's energy delay product can be within a factor of 1.6 of the commercial product, depending on the algorithm being executed. CADRE's greatest benefit stems from the fact that complex algorithms can be executed efficiently by the parallel architecture through the use of compressed instructions and the register file.

The less detailed comparisons, based on headline figures, show the energy-delay metric of CADRE to be 75% of the figure for the OAK DSP. The OAK was also a current low-power product at the time the research started, and was used in the GEM301 baseband processor IC. CADRE performs around 3 times less well than a contemporary research architecture built using 0.35μm technology [153]. However, this architecture uses dual threshold voltage technology to enable efficient low-voltage operation, which is a technique that could be directly applied to CADRE to improve its low-voltage performance. Those processors which have a consistently better energy-delay metric are the more modern devices built using 0.25µm technology or better: the Texas Instruments TMS320C55x and the 1.5v 800MOPs parallel DSP. These have figures that are a factor of 50 better.

When making comparisons of results, the fact that simulations of CADRE do not include wiring capacitances should be considered. A simulation based on full layout would exhibit increased power consumption and reduced throughput. However, the parts of CADRE which currently consume the most power are the functional units, and the multiplier in particular. The multiplier circuit only uses short local interconnections between neighbouring cells, so the capacitance of these should not unduly affect the results. The same is true to a greater or lesser extent of the other parts of the functional units. The greatest wiring loads in CADRE are driven in the transmission of the index register values to the functional units, driving of the GIFU, and accesses to memory. Accesses to memory are kept to a minimum by the architecture, and the GIFU is driven relatively infrequently. The power consumption involved in transmitting the index register values should be reasonably small, since only a few bits tend to change from instruction to instruction. The delay inherent in driving the signals will not affect overall performance, since the index substitution pipeline stage within which the signals are driven is much faster than the critical path of the processor.

Design for low power requires correct decisions to be made at all levels. Most of the work on CADRE has been at an architectural level; due to constraints of time, circuit level designs could not be heavily optimised, although power consumption was clearly borne in mind when choosing circuit structures. The architectures that perform better than CADRE are based on more advanced process technologies and are produced by commercial organisations or larger research groups. It would be expected that many more man-years have been devoted to the low-level optimisation of their products than was possible with CADRE, where the entire architecture was conceived and implemented in a little over two man-years. Furthermore, the headline figures for power consumption and maximum operating speed do not reflect the ability of the architectures to execute complex algorithms efficiently: the difference between the FIR and FFT benchmark results demonstrates how much difference an efficient parallel mapping can make to CADRE's energy-delay figures.

The evaluation of CADRE has been performed on the basis of performing the benchmark algorithms at maximum speed. This hides the effect of an important aspect of the operation of CADRE: the ability of asynchronous circuits to halt and restart virtually instantaneously. The advantage of this ability could only be assessed with CADRE operating in a variety of real applications. However, significant power savings would appear likely, which would provide a substantial advantage over synchronous architectures which require programmer intervention to gate or shut down the clock. The future for mobile telephones appears to involve the integration of more and more functions, including a wide variety of user applications such as speech recognition and multimedia streaming. These user applications will cause processing demand to vary even more than is experienced already from very low demand when idle to very high when streaming high bandwidth compressed multimedia data across a broadband link. An asynchronous system can cope with this varying load most effectively, either by halting when idle or, ideally, through the use of a variable power supply responding to demand.

Chapter 10: Conclusions

10.1 CADRE as a low-power DSP

The results show the effectiveness of the architectural features implemented in CADRE. The 4 parallel functional units allow high throughput to be maintained with the minimum of power consumption. However, this alone is insufficient, since these processing elements must be kept fully occupied to provide efficient operation.

The configuration memories within the functional units allow the parallel archiecture to be used efficiently in complex DSP algorithms, while minimising the power consumed by instruction fetch and decoding. The power consumed in fetching instructions is reduced still further by the use of the instruction buffer to eliminate large numbers of program memory accesses and PC updates.

The use of a large register bank allows data to be supplied to the functional units at a sufficiently high rate, and simplifies program design. The data access patterns of typical DSP algorithms are exploited to simulate a highly ported large register bank through the use of a number of smaller single ported register banks, with the asynchronous design allowing common-case data access patterns to be fast without abandoning support for worst-case patterns. The results show that the register bank design allows the average energy cost of a data access to be around 12 times less than if the data were fetched from main memory.

Having data located within the register bank allows address generation units to be replaced with smaller index generation units to refer to the data required by the algorithms. These index registers can be updated more quickly and at much lower power cost.

The choice of asynchronous design for CADRE offers low electromagnetic interference, and enormously simplifies power management since asynchronous circuits shut down automatically when no processing is required and can restart instantaneously. The simple interrupt structure allows CADRE to perform sequences of tasks or to process blocks of data with the minimum of control overhead, with automatic shutdown once the current task is completed.

Finally, the choice of sign-magnitude representation for data offers some reduced switching activity. However, the savings in power consumption due to this feature are not now felt to be sufficient to justify the additional complexity incurred in the arithmetic elements.

The results show that, individually, each of the architectural features had a considerable effect on the power consumed by that aspect of the architecture. However, the overall performance of CADRE in comparison with other processors was not as good as was hoped for. The power consumption of CADRE was dominated by that of the arithmetic processing elements, and it is now clear that further optimisation of these components is required before the full benefits of the CADRE architecture can be realised.

10.2 Improving CADRE

10.2.1 Scaling to smaller process technologies

The choice of parallel architecture for CADRE was based on the assumption that die area could be traded for reduced power consumption. Clearly, therefore, it would be beneficial to migrate the design to more advanced technologies such as 0.25µm, 0.18µm or smaller.

As indicated by the results for the commercial DSPs operating at 0.25µm, reduced feature size does not only improve system integration. A simple analysis of the effects of scaling ideal MOS transistors [39] also suggests dramatic improvements in energy-delay product: for a scale factor S, intrinsic gate delay reduces by 1/S, and energy per operation (power-delay product) decreases by a factor of $1/S^3$. Were this ideal scaling to hold, the energy-delay product of CADRE would scale by a factor of $1/S^4$; reducing the energy-delay metric by a factor of 50 to only 0.67×10^{-3} mW/MIP^2 on 0.18µm technology, comparable with the other DSPs. However, this simple analysis over-estimates the benefits that may be obtained.

In practice, the ideal behaviour does not hold entirely due to the inability to scale certain parameters. Short-channel effects reduce the maximum theoretical drive current of each transistor, and threshold voltage cannot be scaled ideally due to sub-threshold leakage current (although the leakage can be reduced by the use of a dual threshold process, where the leakage is prevented through the use of high V_t devices where they do not impact on performance). Both of these effects cause the intrinsic delay of each gate to decrease by less than predicted.

An extremely important physical element of the circuits that cannot be scaled linearly are the wires. Wire resistance is dependent on the area of the wires and scales according to the square of the linear scale factor. To allow wires to be packed as closely as possible while maintaining adequate conductance requires the wires to have a tall and thin profile. This means that adjacent wires have a significant capacitance between them. The increased inter-wire capacitance leads to increased crosstalk, and the capacitances of the wires come to dominate the operating speed and power consumption of the gates driving them. The increased resistance and capacitance of the wires causes them to have a significantly increased inherent RC transmission delay, which limits operating speed over longer wires regardless of the strength of the driving gate.

Many of the design features that make up the design of CADRE, such as the use of configuration memories, instruction buffering and the register bank, are intended specifically to minimise the average distance which data must be moved by allowing access to local copies of data. This fact should mitigate the impact of wire loads on both average performance and power consumption. The only points in the CADRE pipeline where signals must potentially travel across the entire width of the core are the decode and index substitution stages. In the decode stage, the only other activities are a very simple logic function to check the instruction type, and a read of the configuration memories. Within the index substitution stage, the only other activity in series with the wire delay is a multiplexing function. In both cases, significant wire delay could be borne without approaching the critical path delay within the multiply-accumulate units.

An increasingly important problem for clocked circuit designers is the managing of clock skew across chips in deep sub-micron technologies. A large design such as CADRE, if synchronous, would require effort in the clock tree design to balance delays to all parts of the circuit and to ensure that data setup and hold times at latches are met. This would be made even more difficult by the need to provide clock gating. The intention for CADRE is to allow the use of heterogeneous functional units with capabilities matched to the requirements of the application. A synchronous system would require that operation was re-verified every time a significant change was made to a functional unit, although the operating frequency of CADRE is sufficiently low that reasonable margins could be included which would make this task easier.

By using asynchronous interfaces to pipeline stages, the timing problem is reduced to ensuring that the delay of the bundled data is matched by the delay in the control signal. It is clearly easier to guarantee the timing relationship of two signals generated within the same circuit than of two signals generated separately.

10.2.2 Optimising the functional units

As has been discussed, the design of CADRE has not been heavily optimised at the circuit level: thus there is likely to be scope for significant improvement in both the speed and power consumption of parts of the circuit. The part of the circuit where the most improvement could be gained is the functional units: these consume over 50% of system power, and represent the critical path of the device for many operations.

Multiplier optimisation

Breakdowns of MAC unit power consumption show that the multiplier tree consumes the greatest part of the power. This appears to be due to the large number of spurious transitions generated within the compression tree by the 2:1 signed digit adders used: the analysis of multiplier structures in [59] suggests that this may be an inherent problem for this number representation. These adders also switch significant internal capacitance for changes in the input values. This style of multiplier was chosen due to the elegance of partial product generation with the sign-magnitude number system. However, sign-magnitude numbering appears to be of much less benefit should 1-of-4 coding be implemented for module interfaces. It would therefore be beneficial, for reduced complexity and power consumption and increased operating speed, to reimplement the functional units to use 2s complement numbering. This change would not affect anything outside of the functional units.

Pipelined multiply operation

The longest critical path for the processor is for multiply-accumulate operations, with other parts of the system operating at twice this speed or more. A straightforward way to increase the speed of the processor would therefore be to pipeline the multiply operation. The first stage of the multiply pipeline would be partial product generation and compression. The second stage would be accumulation and summing of the redundant result, avoiding pipeline dependencies except when the input to the multiplier was from the accumulator registers.

Two different strategies could be adopted to achieve this. The simplest strategy would be to have an intermediate register between the multiplier and adder, so that multiplication would effectively become a two instruction operation. This technique would put pipeline dependencies under programmer control, and allow the external interfaces to the functional units to remain the same.

The more complicated strategy would be to split execution into two pipeline stages. This would require different strategies to be adopted for dealing with pipeline dependencies, particularly relating to store operations and writebacks, and drive of the GIFU/LIFU. However, this technique would give the best performance since the latency of a multiply-accumulate operation would only be slightly increased by the addition of a pipeline latch but the throughput could be approximately doubled.

Adder optimisation

The second greatest source of power consumption is the adder. While the 3-input carry resolution tree is extremely fast, the circuits are pseudo-dynamic and therefore undergo activity regardless of the input data characteristics. If pipelining were employed, the adder could be designed so that the critical paths of the multiply stage and addition stages were matched. A lower power static adder design could then be chosen with the appropriate performance, by a method such as that proposed in [145].

Improving overall functional unit efficiency

Due to time constraints, the functional units as currently designed are identical and implement the same functions, including some rarely used operations such as normalisation and scaling or distance calculation. Complexity of each functional unit could be improved by designing functional units with different capabilities such that the required algorithms could be efficiently mapped onto them without excess functionality. This would improve the power consumption and possibly the speed of each unit, reduce the area, and potentially allow fewer bits to be used in the configuration memories. The use of delay-insensitive interfaces would allow a library of functional units to be maintained and used in a given application with ease.

10.2.3 Optimising communication pathways

In the FIR filter algorithm shown in Table 3.2 on page 88, it can be seen that each value from the data 'moves across' the functional units in subsequent instructions; for example, data point x_{n-3} begins in MAC D, and is then processed by MAC C, B and A in successive instructions. Currently this requires each value to be read from the register bank four times. This data reuse could be readily exploited by including a 16-bit pathway between adjacent functional units through which the data could pass with significantly reduced energy cost.

10.2.4 Optimising configuration memories

The configuration memories represented the second greatest source of power consumption. The greatest number of configuration memories used in any of the test programs was 29, for the LPC analysis code. This suggests that a benefit could be obtained by splitting the configuration memories into two smaller sub-banks each containing 64 entries, allowing a number of different algorithms to be located within each sub-bank. By only driving the bit- and word-lines of one half at a time, the power consumption could be reduced and access speed increased.

Currently, configuration occurs as part of instruction execution, so other execution must stop to write new data to the configuration memories. Splitting the configuration memories would enable configuration to be taken out of the execution stream, and instructions for new algorithms could be loaded while the previous algorithms complete.

With configuration taken out of the execution stream, more complex configuration mechanisms could be easily employed. For example, different functional units could demand different bit widths for their configuration from the host microprocessor, to reflect the complexity of the internal functions. This data could then be transmitted in packets across a delay-insensitive interface.

Configuration data is amenable to compression (e.g. by 'gzip' or similar), and it would be possible to maintain configuration data for the DSP in compressed form with the host microprocessor or a dedicated circuit extracting the information required for an algorithm. This would reduce both the cost of main memory and the amount of information fetched from it.

10.2.5 Changes to the register bank

Currently, the size of the register bank is easily sufficient to contain a single frame of GSM speech data. However, future applications may require more storage to be executed most efficiently.

An increase of register bank size by a factor of 2 or 4 could easily be accommodated simply by increasing the size of the register sub-banks: further increases could be accommodated by using a RAM-like design using sense amplifiers. The number of sub-banks could also be increased, increasing both the size and the number of accesses to sequential registers that could be accommodated, at the expense of increased area and power consumption in the switching network between the ports and the register sub-banks.

An increase in the size of the register bank would require changes in the surrounding architecture to allow them to be addressed. A minimal change would be to increase the width of the index registers, allowing only indexed accesses to address the full range of registers. Changes would then be required to instructions that set up the index registers; in particular MOVEM instructions which could no longer hold 4 full index register values. To allow full access to an enlarged register bank for all types of addressing would require the width of configuration memories to be increased to contain the extra address bits.

Currently, no register locking is enforced for writebacks to the register bank from the functional units. It is left to the programmer to ensure that an instruction is inserted between a writeback and an instruction referencing the written data, except for the case of a store operation which causes a pipeline stall regardless of any possible hazards.

Should the pipeline depth be increased, to implement pipelined multiply-accumulate, the impact of this stall would be greater and more inserted instructions would be needed to prevent hazards. This would make a full register locking implementation more attractive.

10.3 Conclusions

The EPSRC ‘Powerpack’ project, through which my PhD studentship was funded, set as its goal to reduce power consumption by an order of magnitude in a number of key applications. Initial schematic simulations show that CADRE has already reduced the mW/MIPS^2 figure by 25% compared to the the OAK DSP, which formed part of the mobile phone chipset example presented at the beginning of the project, and the energy-delay metric is also somewhat improved. Given the discussed optimisations to improve the performance of the arithmetic circuitry, and transfer of the design onto a modern process technology, CADRE looks set to provide new directions for the next generation of DSP architectures. The architectural features allow the design to be scaled onto very deep sub-micron processes, to execute the complex high-performance algorithms required by the mobile phone applications of the future.

References

[1] “The world in your pocket”, *The Economist,* 9 October 1999

[2] “Mobile-phone licences - Giddy bidding”, *The Economist*, 15 April 2000

[3] *Official Bluetooth web site,* http://www.bluetooth.com/

[4] *GEM301 GSM Baseband Processor Preliminary Information*, Mitel Semiconductors 1997

[5] P.S. Laplace, *Theorie Analytique des Probabilities*, 2 vols., available from Editions Culture et Civilisation, 115 Avenue Gabriel Lebron, 1160 Brussels, Belgium

[6] E.T. Jaynes, “Bayesian Methods: General Background”, *Maximum Entropy and Bayesian Methods in Applied Statistics*, ed. J.H. Justice, Cambridge University Press 1985, pp. 1-25

[7] W.D. Stanley, G.R. Dougherty, R. Dougherty, *Digital Signal Processing (2nd edition)*, Reston Publishing Company Inc., Reston, Virginia, 1984

[8] Boddie, J. “On the 20th anniversary of DSP1”, After dinner speech at DSP World, Orlando, Florida November 2, text available on-line at http://www.lucent.com/micro/starcore/doc.html

[9] R.J. Higgins, *Digital Signal Processing in VLSI*, Analog Devices Technical Reference, Prentice Hall, Eaglewood Cliffs NJ, 1990

[10] Intel Corp. *2920 Analog Signal Processor Design Handbook,* Intel Corp. Santa Clara California, 1980

[11] Texas Instruments Inc. “TMS32010 and TMS320M10 High-Performance 16/32 Bit Microcomputers,” *Microcomputer Series Preliminary Data Manual,* Texas Instruments Inc., June 1982

[12] Analog Devices Inc. *ADSP-2100 User's Manual - Architecture*, Analog Devices Inc., Norwood MA, 1988

[13] J.G. Proakis, D.G. Manolakis, *Digital Signal Processing: Principles, Algorithms and Applications (3rd edition)*, Prentice Hall, Upper Saddle River, NJ, 1996

[14] *DSP56000 24-bit Digital Signal Processor Family Manual*, Motorola Inc., 1990

[15] J.W. Cooley, J.W. Tukey, "An Algorithm for the Machine Computation of Complex Fourier Series", *Math. Comp.*, vol. 19, pp. 297-301, April 1965

[16] S.H. Redl, M. Weber, M.W. Oliphant, *An Introduction to GSM*, Artech House 1995, ISBN 0890067856

[17] M. Rahnema, "Overview of the GSM System and Protocol Architecture", *IEEE Communications Magazine*, April 1993

[18] GSM Technical Specification 06.10 "Full rate speech transcoding", European Telecommunications Standards Institute 1998, http://www.etsi.org/

[19] GSM Technical Specification 06.20 "Half rate speech transcoding", European Telecommunications Standards Institute 1998, http://www.etsi.org/

[20] GSM Technical Specification 06.60 "Enhanced Full Rate (EFR) Speech transcoding", European Telecommunications Standards Institute 1997, http://www.etsi.org/

[21] GSM Technical Specification 05.03 "Channel coding", European Telecommunications Standards Institute 1995, http://www.etsi.org/

[22] GSM Technical Specification 05.01 "Physical layer on the radio path", European Telecommunications Standards Institute 1996, http://www.etsi.org/

[23] A. Gatherer, T. Stetzler, M. McMahan, E. Auslander, "DSP-Based Architectures for Mobile Communications: Past, Present and Future", *IEEE Communications Magazine*, January 2000, pp. 84-90

[24] J.G. Proakis, *Digital Communications (3rd edition)*, McGraw-Hill Inc. 1995

[25] B. Sklar, "Rayleigh Fading Channels in Mobile Digital Communication Systems Part II: Mitigation", *IEEE Communications Magazine*, July 1997, pp. 102-109

[26] R.E. Blahut, *Theory and Practice of Error Control Codes*, Addison-Wesley Publishing Company Inc. 1983

[27] A.J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm", *IEEE Transactions on Information Theory*, vol. IT13, April 1967, pp. 260-269

[28] A.M. Kondoz, *Digital Speech: Coding for Low Bit Rate Communications Systems*, John Wiley & Sons Ltd., 1994

[29] B. Kim et al., "MDSP-II: A 16-Bit DSP with Mobile Communication Accelerator", IEEE Journal of Solid-State Circuits, Vol. 34 No.3, March 1999, pp. 397-404

[30] W. Honcharenko, J.P. Kruys, D.Y.Lee, N.J. Shah, "Broadband Wireless Access", *IEEE Communications Magazine*, January 1997, pp. 20-26

[31] M. Zeng, A. Annamalai, V.K. Bhargava, "Recent Advances in Cellular Wireless Communications", IEEE Communications Magazine, September 1999, pp. 128-138

[32] U. Varshney, "Recent Advances in Wireless Networking", *IEEE Computer Magazine*, June 2000, pp. 100-103

[33] *International Telecommunication Union IMT-2000 web pages,* http:// www.itu.org/imt

[34] W. Namgoong, S. Reader, T.H. Meng, "An All-Digital Low-Power IF GPS Synchronizer", *IEEE Journal of Solid-State Circuits*, vol. 35 no. 6, June 2000, pp. 856-864

[35] Y Chang, H. Suzuki, K.K. Parhi, “A 2-Mb/s 256-State 10-mW Rate-1/3 Viterbi Decoder”, IEEE Journal of Solid-State Circuits, vol. 35 no. 6, June 2000, pp. 826-834

[36] L. Brackenbury, M. Cumpstey, S. Furber, P. Riocreux, “An Asynchronous Viterbi Decoder”, *European Low Power Initiative for Electronic System Design 3rd International Workshop*, July 2000, pp. 8-21

[37] N. Weste, K. Eschragian, *Principles of CMOS VLSI Design: A Systems Perspective*, Addison-Wesley, 1988

[38] A.P. Chandrakasan, R.W. Brodersen, “Minimizing Power Consumption in Digital CMOS Circuits”, *Proceedings of the IEEE*, Vol. 83 No. 4 April 1995

[39] H.B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley, 1990

[40] A.P. Chandrakasan, R.W. Brodersen, *Low Power Digital CMOS Design*, Kluwer Academic Publishers, 1995

[41] A.J. Bhavnagarwala, B.L. Austing, K.A. Bowman, J.D. Meindl, “A Minimum Total Power Methodology for Projecting Limits on CMOS GSI”, *IEEE Transactions on VLSI Systems*, Vol. 8 No. 3, June 2000, pp. 235-251

[42] M.R. Stan, “Low Threshold CMOS Circuits with Low Standby Current”, *Proc. International Symposium on Low-Power Electronics and Design,* 1998, pp. 97-99

[43] J.T. Kao, A.P. Chandrakasan, “Dual-Threshold Voltage Techniques for Low-Power Digital Circuits”, *IEEE Journal of Solid-State Circuits*, Vol. 35 No. 7, July 2000, pp. 1009-1018

[44] M.M. Khella, M.I. Elmasry, “Power Minimization of High-Performance Submicron CMOS Circuits Using a Dual-V_{dd} Dual-V_{th} (DVDV) Approach, *Proc. International Symposium on Low-Power Electronics and Design,* 1999, pp. 106-108

[45] A.P. Dancy, R. Amirtharajah, A.P. Chandrakasan, "High-Efficiency Multiple-Output DC-DC Conversion for Low-Voltage Systems", *IEEE Transactions on VLSI Systems*, Vol. 8 No. 3, June 2000, pp. 252-263

[46] D.J. Frank, P. Solomon, S. Reynolds, J. Shin, "Supply and Threshold Voltage Optimization for Low Power Design", *Proc. International Symposium on Low-Power Electronics and Design,* 1997, pp. 317-322

[47] L. Nielsen, C. Nielsen, J. Sparsø, K. van Berkel, "Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage", *IEEE Transactions on VLSI Systems,* Vol. 2. No. 4, December 1994, pp.391-397

[48] L. Nielsen, J. Sparsø, "A Low Power Datapath for a FIR Filter Bank", *Proc. International Workshop Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996, IEEE Computer Society Press

[49] T.R. Halfhill, "Transmeta breaks x86 low-power barrier: VLIW chips use hardware-assisted x86 emulation", *Microprocessor Report*, Vol. 14, No. 2, February 2000, pp.1, 9-18

[50] D. Somasekhar, K. Roy, "LVDCSL: Low Voltage Differential Current Switch Logic, a Robust Low Power DCSL Family", *Proc. International Symposium on Low-Power Electronics and Design,* 1997, pp. 18-23

[51] A.M. Fahim, M.I. Elmasry, "SC^2L; A Low-Power High Performance Dynamic Differential Logic Family", *Proc. International Symposium on Low-Power Electronics and Design,* 1999, pp. 88-90

[52] H. Zhang, V. George, J.M. Rabaey, "Low-Swing On-Chip Signalling Techniques: Effectiveness and Robustness", *IEEE Transactions on VLSI Systems*, Vol. 8 No. 3, June 2000, pp. 264-272

[53] W.C. Athas, W-C Liu, L."J". Svensson, "Energy-Recovery CMOS for Highly Pipelined DSP Designs", *Proc. International Symposium on Low-Power Electronics and Design,* 1996, pp. 101-104

[54] J. Lim, D-G Kim, S-I Chae, “A 16-bit Carry-Lookahead Adder Using Reversible Energy Recovery Logic for Ultra-Low-Energy Systems”, IEEE Journal of Solid-State Circuits, Vol. 34 No. 6, June 1999, pp. 898-903

[55] K-Y Khoo, A.N. Wilson Jr., “Charge Recovery on a Databus”, *Proc. International Symposium on Low-Power Electronics and Design,* 1995, pp. 185-189

[56] B. Bishop, M.J. Irwin, “Databus Charge Recovery: Practical Considerations”, *Proc. International Symposium on Low-Power Electronics and Design,* 1999, pp. 85-87

[57] I. Sutherland, B. Sproull, D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann Publishers Inc., 1999

[58] C. Svensson, A. Alvandpour, “Low Power and Low Voltage CMOS Digital Circuit Techniques”, *Proc. International Symposium on Low-Power Electronics and Design,* 1998, pp. 7-10

[59] G. Keane, J. Spanier, R. Woods, “The Impact of Data Characteristics and Hardware Topology on Hardware Selection for Low Power DSP”, *Proc. International Symposium on Low-Power Electronics and Design,* 1998, pp. 94-96

[60] Y-C Tseng, S.C. Chin, J.C.S. Woo, “The Impact of SOI MOSFETs on Low Power Digital Circuits”, *Proc. International Symposium on Low-Power Electronics and Design,* 1997, pp. 243-246

[61] D.A. Antoniadis, “SOI CMOS as a Mainstream Low-Power Technology: A Critical Assesssment”, *Proc. International Symposium on Low-Power Electronics and Design,* 1997, pp. 295-300

[62] F. Assaderaghi, “Circuit Styles and Strategies for CMOS VLSI Design on SOI”, *Proc. International Symposium on Low-Power Electronics and Design,* 1999, pp. 282-287

[63] A.G. Aipperspach, D.H. Allen, D.T. Cox, N.V. Phan, S.N. Storino, "A 0.2-µm, 1.8V, SOI, 550-MHz, 64-b PowerPC Microprocessor with Copper Interconnects", *IEEE Journal of Solid-State Circuits*, Vol. 34 No. 11, November 1999, pp. 1430-1435

[64] S.B. Park, Y.W. Kim, Y.G. Ko, K.I. Kim, I.K. Kim, H-S Kang, J.O. Yu, K.P. Suh, "A 0.25-µm, 600-MHz, 1.5-V, Fully Depleted SOI CMOS 64-Bit Microprocessor", *IEEE Journal of Solid-State Circuits*, Vol. 34 No. 11, November 1999, pp. 1436-1445

[65] M. Favalli, L. Benini, "Analysis of glitch power dissipation in CMOS ICs", *Proc. 1995 International Symposium on Low Power Design,* 1995, pp. 123-128

[66] J.H. Satyanarayana, K.K. Parhi, "Theoretical Analysis of Word-Level Switching in the Presence of Glitching and Correlation", *IEEE Transactions on VLSI Systems*, Vol. 8 No.2, April 2000, pp. 148-159

[67] H. Kapadia, L. Benini, G. De Micheli, "Reducing Switching Activity on Datapath Buses with Control-Signal Gating", *IEEE Journal of Solid-State Circuits*, Vol. 34 No. 3, March 1999, pp. 405-414

[68] A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic", *IRE Transactions on Electronic Computers,* Vol. 10, September 1961, pp. 389-400

[69] B. Parhami, "Generalized Signed-Digit Number Systems- A Unifying Framework for Redundant Number Representations", *IEEE Transactions on Computers*, Vol. 39, No.1, January 1990

[70] S. Arno, F. S. Wheeler, "Signed-Digit Representations of Minimal Hamming Weight", *IEEE Transactions on Computers*, Vol. 42, No. 8, August 1993

[71] N. S. Szabó, R. I. Tanaka, "Residue Arithmetic and its Applications to Computer Technology", *McGraw-Hill Series in Information Processing and Computers*, 1967

[72] M. A. Hitz, E. Kaltofen, "Integer Division in Residue Number Systems", *IEEE Transactions on Computers* Vol. 4 No. 8, August 1995

[73] R. Sivakumar, N. J. Dimopoulos, “VLSI Architectures for Calculating X mod *m*”, *IEE Proceedings on Circuits, Devices, and Systems*, Vol. 142, No. 5, October 1995

[74] W. A. Chren Jr, “One-Hot Residue Coding for Low Delay-Power Product CMOS Design”, *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Vol. 45, No. 3, March 1998

[75] M.R. Stan, W.P. Burleson, “Low-Power Encodings for Global Communication in CMOS VLSI”, *IEEE Transactions on VLSI Systems,* Vol. 5, No.4, pp. 444-455, 1997

[76] N. Takagi, S. Yajima, “High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree”, *IEEE Transactions on Computers*, Vol. C34, No. 9, September 1985

[77] W. Balakrishnan, N. Burgess, “A Very-high-speed VLSI 2s-Complement Multiplier Using Signed Binary Digits”, *IEE Proceedings-E*, Vol. 139, No. 1, January 1992

[78] A. D. Booth, “A Signed Binary Multiplication Technique”, *Computer Arithmetic- Benchmark Papers in Electrical Engineering and Computer Science Vol. 21* (ed. E.E. Swartzlander), Dowden Hutchinson and Ross Inc. 1980, ISBN 0-87933-350-2

[79] H. Kojima, A. Shridhar, “Interlaced Accumulation Programming for Low Power DSP”, *Proc. International Symposium on Low-Power Electronics and Design,* 1996, pp. 213-216

[80] A. T. Erdogan and T. Arslan, “Data Block Processing for Low Power Implementation of Direct Form FIR Filters on Single Multiplier CMOS Based DSPs”, *IEEE International Symposium on Circuits and Systems*, June 1998, pp. D441-D444

[81] V. Sundararajan and K.K. Parhi, “A Novel Multiply Multiple Accumulator Component for Low Power PDSP Design”, *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, June 2000

[82] A.T. Erdogan, T. Arslan, “Low power multiplication scheme for FIR filter implementation on single multiplier CMOS DSP processors”, *IEE Electronics Letters*, Vol. 32 No. 21, October 1996, pp. 1959-1960

[83] K. Masselos, P. Merakos, T. Stouraitis, C.E. Goutis, “Novel Techniques for Bus Power Consumption Reduction in Realizations of Sum-of-Product Computation”, *IEEE Transactions on VLSI Systems*, Vol. 7 No. 4, December 1999, pp. 492-497

[84] P. Hicks, M. Walnock, R.M. Owens, “Analysis of Power Consumption in Memory Hierarchies”, *Proc. International Symposium on Low-Power Electronics and Design,* 1997, pp. 239-242

[85] 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

[86] L. Benini, A. Macii, E. Macii, M. Poncino, “Selective Instruction Compression for Memory Energy Reduction in Embedded Systems”, *Proc. International Symposium on Low-Power Electronics and Design,* 1999, pp. 206-211

[87] C. Lefurgy, T. Mudge, “Code Compression for DSP”, presented at Compiler and Architecture Support for Embedded Computing Systems (CASES) 1998

[88] I.E. Sutherland, “Micropipelines”, *Communications of the ACM*, Vol. 32 No. 6, June 1989, pp. 720-738

[89] P. Day, J. V. Woods “Investigations into micropipeline latch design styles”, *IEEE Transactions on VLSI systems*, Vol. 3 No. 2, June 1995, pp. 264-272

[90] S. B. Furber, P. Day, “Four-phase micropipeline latch control circuits”, *IEEE Transactions on VLSI systems*, Vol. 4 No. 2, June 1996, pp 247-253

[91] A.J. Martin, “Asynchronous Datapaths and the Design of an Asynchronous Adder”, *Formal Methods in System Design*, Vol. 1 No. 1, July 1992, pp.119-137

[92] K. van Berkel, M. Rem, "VLSI Programming of Asynchronous Circuits for Low Power", *Asynchronous Digital Circuit Design*, Springer-Verlag, 1995, pp.152-210.

[93] J. Haans, K. van Berkel, A. Peeters, F. Schalij, "Asynchronous Multipliers as Combinational Handshake Circuits", *Asynchronous Design Methodologies*, IFIP Transactions, Vol. A28, Elsevier Science Publishers, 1993, pp.149-163

[94] C. D. Nielsen, A. J. Martin, "Design of a Delay-Insensitive Multiply-Accumulate Unit", *Integration,* Vol.15 No.3, 1993, pp.291-311

[95] B. W. Hunt, K. S. Stevens, B. W. Suter, D. S. Gelosh, "A Single-Chip Low Power Asynchronous Implementation of an FFT Algorithm for Space Applications", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp.216-223, IEEE Computer Society Press.

[96] A. Bardsley, D.A. Edwards, "Compiling the Language Balsa to Delay Insensitive Hardware", *Proc. CHDL'97*, April 1997, published in *Hardware Descriptions Languages and their Applications*, IFIP & Chapman Hall, 1997, pp. 89-91, ISBN 01412 78810 1

[97] T. Ono-Tesfaye, C. Kern, M. Greenstreet, "Verifying a Self-Timed Divider", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 146-158, IEEE Computer Society Press.

[98] M. Ligthart, K. Fant, R. Smith, A. Taubin, A. Kondratyev, "Asynchronous Design Using Commercial HDL Synthesis Tools", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000, pp. 114-125, IEEE Computer Society Press

[99] T. E. Williams, M. A. Horowitz, "A 160ns 54 bit CMOS Division Implementation Using Self-Timing and Symmetrically Overlapped SRT Stages", *Proc. 10th IEEE Symposium on Computer Arithmetic*, 1991, pp. 210-217

[100] M. Singh, S.M. Nowick, "High-Throughput Asynchronous Pipelines for Fine-Grain Dynamic Datapaths", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000, pp. 198-209, IEEE Computer Society Press

[101] J Sparsø, C. Nielsen, L. Nielsen, J. Staunstrup, "Design of Self-Timed Multipliers: A Comparison", *Asynchronous Design Methodologies*, IFIP Transactions, Vol. A28, Elsevier Science Publishers, 1993, pp. 165-179

[102] O. Salomon, H. Klar, "Self-timed Fully Pipelined Multipliers", *Asynchronous Design Methodologies*, IFIP Transactions, Vol. A28, Elsevier Science Publishers., 1993, pp. 45-55

[103] G. Matsubara, N. Ide, "Low-power Zero Overhead Self-Timed Division and Square-Root Unit Combining Single-Rail Static with Dual-Rail Dynamic Circuits", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997, pp. 198-209, IEEE Computer Society Press

[104] K. Lee, K. Choi, "Self-Timed Divider Based on RSD Number System", *IEEE Transactions on VLSI Systems*, Vol. 4 No. 2, June 1996, pp.292-295

[105] J. D. Garside, "A CMOS VLSI Implementation of an Asynchronous ALU", *IFIP Working Conference on Asynchronous Design Methodologies*, April 1993

[106] D. J. Kinniment, J. D. Garside, B. Gao, "A Comparison of Power Consumption in Some CMOS Adder Circuits", *Power and Timing Modelling, Optimisation and Simulation (PATMOS)*, 1995

[107] D.J. Kinniment, "An Evaluation of Asynchronous Addition", *IEEE Transactions on VLSI Systems*, Vol. 4 No. 1, March 1996, pp.137-140

[108] J. Escriba, J. Carrosco, "Self-Timed Manchester-Chain Carry Propagate Adder", *IEE Electronics Letters*, Vol. 32 no. 8, 1996, pp. 708-710

[109] D. Johnson, V. Akella, "Design and Analysis of Asynchronous Adders", *IEE Proceedings, Computers and Digital Techniques*, Vol. 145, No. 1, 1998, pp. 1-8

[110] D. Kearney, N, Bergmann, “Bundled Data Asynchronous Multipliers with Data Dependent Computation Times”, *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997, pp. 186-197, IEEE Computer Society Press

[111] S. M. Nowick, “The Design of a Low-Latency Asynchronous Adder Using Speculative Completion”, *IEE Proceedings, Computers and Digital Techniques*, Vol. 143 No. 5, September 1996, pp. 301-307

[112] S. Nowick, K. Yun, P. Beerel, A. Dooply, “Speculative Completion for the Design of High Performance Asynchronous Dynamic Adders”, *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997, pp. 210-223, IEEE Computer Society Press

[113] K. Van Berkel, “Beware the Isochronic Fork”, *Integration*, Vol. 13 No. 2, June 1992, pp. 103-128

[114] A.J. Martin, “Compiling Communicating Processes into Delay-Insensitive VLSI Circuits”, *Distributed Computing* 1:226-234, 1986

[115] S.H. Unger, *Asynchronous Sequential Switching Circuits*, Wiley-Interscience, John Wiley & Sons Inc., 1969

[116] S.M. Nowick, K.Y. Yun, D.L. Dill, “Practical Asynchronous Controller Design”, *Proc. International Conference on Computer Design (ICCD),* 1992, pp. 341-345, IEEE Computer Society Press

[117] 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

[118] D. W. Dobberpuhl et al, “A 200Mhz 64-b Dual Issue CMOS Microprocessor”, *IEEE Journal of Solid-State Circuits*, Vol. 27 No. 11, 1992, November 1992, pp. 1555-1565

[119] S.B. Furber, J.D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, N.C. Paver "AMULET2e: An Asynchronous Embedded Controller", *Proceedings of the IEEE*, Vol. 87 No.2, February 1999, pp. 243-256

[120] K.Y. Yun, A.E Dooply, "Pausible Clocking-Based Heterogeneous Systems", IEEE Transactions on VLSI Systems, Vol. 7 No. 4, December 1999, pp. 482-488

[121] J. Muttersbach, T. Villiger, W. Fichtner, "Practical Design of Globally-Asynchronous Locally-Synchronous Systems", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 52-59, IEEE Computer Society Press

[122] M. Roncken, K. Stevens, R. Pendurkar, S. Rotem, P. Pal Chaudhuri, "CA-BIST for Asynchronous Circuits: A Case Study on the RAPPID Asynchronous Instruction Length Decoder", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 62-72, IEEE Computer Society Press

[123] M. Lewis, J. Garside, L. Brackenbury, "Reconfigurable latch controllers for low-power asynchronous circuits", *Proc. International Workshop Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1999, pp. 27-35, IEEE Computer Society Press

[124] P.A. Riocreux, M.J.G. Lewis, L.E.M. Brackenbury, "Power reduction in self-timed circuits using early-open latch controllers", *IEE Electronics Letters*, vol.36, January 2000, pp.115-116 (ISSN 0013-5194)

[125] J. Liu, "Arithmetic and control components for an asynchronous microprocessor", PhD thesis, Department of Computer Science, University of Manchester, 1997

[126] T. Arslan, A.T. Erdogan, D.H. Horrocks, "Low Power Design for DSP: Methodologies and Techniques", *Microelectronics Journal*, vol. 27 no. 8 pp. 731-744, Nov. 1996

[127] A. Bindra, "Flexible, modular process packs more than half a billion transistors on a tiny silicon chip", *Electronic Design*, vol.48 no.10, pp.26, May 2000

[128] *An Overview of the ZSP Architecture*, white paper available at http://www.zsp.com/, LSI Logic Inc.

[129] *TMS320C55x DSP Core Technical Documentation*, Texas Instruments Inc.

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

[131] M.S. Schlansker, B. Ramakrishna Rau, "EPIC: Explicitly Parallel Instruction Computing", *COMPUTER*, vol.33 no.2, February 2000, pp.37-45

[132] P. Kievits, E. Lambers, C. Moerman, R. Woudsma, "R.E.A.L. DSP Technology for Telecom Baseband Processing", *Proc. 9th International Conference on Signal Processing Applications and Technology*, Miller Freeman Inc., 1998

[133] *Carmel DSP Core Technical Overview Handbook,* Infineon Technologies, 2000

[134] J.M. Rabaey, "Reconfigurable Processing: The Solution to Low-Power Programmable DSP", *Proc. International Conference on Acoustics, Speech and Signal-Processing*, April 1997

[135] L. Hwang Lee, B. Moyer, J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops", *Proc. International Symposium on Low-Power Electronics and Design,* 1999, pp. 267-269

[136] R.S. Bajwa, M. Hiraki, H. Kojima, D.J. Gorny, K. Nitta, A. Shridhar, K. Seki, K. Sasaki, "Instruction Buffering to Reduce Power in Processors for Signal Processing", *IEEE Transactions on VLSI Systems*, vol. 5 no. 4 pp. 417-423, Dec. 1997

[137] M. Lewis, L.E.M. Brackenbury, "An Instruction Buffer for a Low-Power DSP", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp. 176-186, IEEE Computer Society Press

[138] P.B. Endecott, S.B. Furber, "Modelling and Simulation of Asynchronous Systems using the LARD Hardware Description Language", *Proc. 12th European Simulation Multiconference*, June 1998, Society for Computer Simulation International, pp. 39-43

[139] C.L. Seitz, "System Timing", *Introduction to VLSI Systems*, ed. C.A. Mead, L.A. Conway, Addison-Wesley Inc. 1980, Chapter 7

[140] E. Brunvand, "Low-latency self-timed flow-through FIFOs", in *16th Conference on Advanced Research in VLSI,* Chapel Hill, NC, IEEE Computer Society Press, pp. 76-90, 1995

[141] K. Yi, "The Design of a Self-Timed Low Power FIFO using a Word-Slice Structure", M.Phil. thesis, Department of Computer Science, University of Manchester 1998

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

[143] 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

[144] 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

[145] K.K. Parhi, "Low-energy CSMT carry generators and binary adder", *IEEE Transactions on VLSI Systems*, Vol. 7 No. 4, Dec. 1999, pp. 450-462

[146] 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"

[147] R. Fried, "Minimizing Energy Dissipation in High-Speed Multipliers", *International Symposium on Low-Power Electronics and Design,* 1997, pp. 214-219

[148] 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

[149] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, J. Rabaey, “Evaluation of a Low-Power Reconfigurable DSP Architecture,” *Proceedings of the Reconfigurable Architectures Workshop*, Orlando, Florida, USA, March 1998

[150] R. Gonzalez, M. Horowitz, “Energy Dissipation in General Purpose Microprocessors”, *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 9, Sep. 1996 pp. 1277-1284

[151] *DSP Group Inc. website*, http://www.dspg.com/

[152] N.C. Paver, P. Day, C. Farnsworth, D.L. Jackson, W.A. Lien, J. Liu, “A Low-Power, Low Noise, Configurable Self-Timed DSP”, Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, 1998, pp. 32-42

[153] W. Lee et al. “A 1v Programmable DSP for Wireless Communications”, *IEEE Journal of Solid-State Circuits*, vol. 32 no. 11, Nov. 1997, pp. 1767-1776

[154] H. Igura, Y. Naito, K. Kazama, I. Kuroda, M. Motomura, M. Yamashina, “An 800-MOPS, 110mW, 1.5-V, Parallel DSP for Mobile Multimedia Processing”, *IEEE Journal of Solid-State Circuits*, vol. 33 no.11, Nov. 1998, pp. 1820-1828

[155] B. Ackland et al., “A Single-Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP”, *IEEE Journal of Solid-State Circuits*, vol. 35 no.3, March 2000, pp. 412-424

Appendix A: The GSM full-rate codec

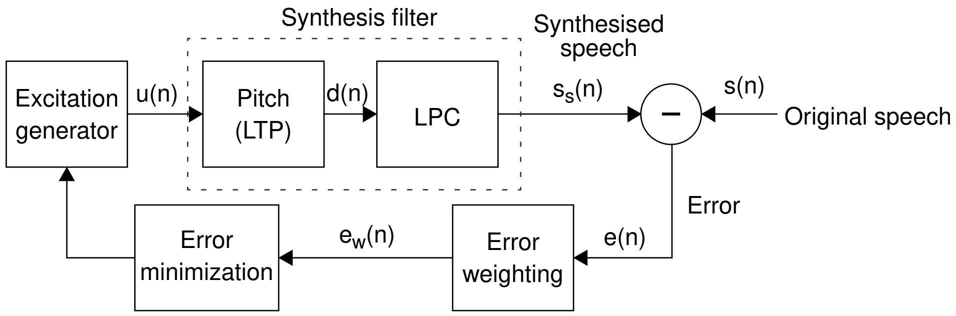


Figure A.1 Analysis-by-synthesis model of speech

A.1 Speech pre-processing

The input to the GSM speech encoder is a block of 160 13-bit samples. These are scaled appropriately, and a DC blocking (notch) filter is applied to remove any offset in the input signal from the analogue to digital converter. This filter has the equation

$$s_{of}(k) = s_o(k) - s_o(k-1) + 32765 \times 2^{-15} s_{of}(k-1) \tag{23}$$

In the Z-transform domain this filter has the form

$$S_{of}(z) = \frac{z-1}{z - 32765 \times 2^{-15}} \times S_o(z) \tag{24}$$

The filter has a pole very close to the unit circle at $z=0.999$, and to guarantee stability it is necessary to use a double precision 31x16 bit multiply for the recursive part. Overall, the preprocessing section requires one subtract, two multiplies and two adds per data point (in addition to a number of shifts required for scaling of the input signal and to perform the double-precision multiplication).

Following the offset removal, the signal is passed to a first-order high-pass FIR pre-emphasis filter with the equation:

$$s(k) = s_{of}(k) - 0.860 s_{of}(k-1) \tag{25}$$

This part of the speech preprocessing requires one multiply-accumulate per data point.

A.2 LPC Analysis

The next processing stage in the GSM full-rate codec is to estimate the parameters of the linear predictive coding filter. LPC models a signal $x(k)$ as the output of an IIR filter of order *P* driven by an excitation signal $e(k)$:

$$x(k) = e(k) + \sum_{i=1}^{P} a(i)x(k-i) \tag{26}$$

For the GSM full-rate codec, the number of model parameters *P* used is 8. An estimate of the parameters is obtained by solving the set of simultaneous linear equations

$$\begin{bmatrix} r(0) & r(1) & \dots & r(8) \\ r(1) & r(0) & \dots & r(7) \\ \dots & \dots & & \dots \\ r(8) & r(7) & \dots & r(0) \end{bmatrix} \begin{bmatrix} 1 \\ a(1) \\ \dots \\ a(8) \end{bmatrix} = \begin{bmatrix} r(0) \\ 0 \\ \dots \\ 0 \end{bmatrix} \tag{27}$$

where *r(k)* is the autocorrelation function of the signal defined by $r(k) = \sum_{i=k}^{N} x(i)x(i-k)$. To calculate these autocorrelation values requires 1249 multiply-accumulate operations (160 for *r(0)*, 159 for *r(1)*, etc.). Prior to this, the absolute maximum value of the signal must be found, requiring 160 subtract operations, and the entire signal normalized to avoid overflow, requiring 160 shift operations (although if not implementing a bit-exact GSM codec and the accumulators have sufficient guard bits to prevent overflow during the autocorrelation calculations, it is possible to only normalize the resulting autocorrelation values).

The autocorrelation matrix is Toeplitz, and is solved using the Schür recursion: this calculates the reflection coefficient form of the AR parameters K_i, rather than the direct form given by (26). The main advantage of the reflection coefficient form is that they can be used to calculate an inverse filter which is guaranteed to be stable. A good explanation of the mechanics of the Schür recursion is given in [13], p870. For the case of the 8 parameter LPC analysis considered here, the algorithm requires 8 divisions, and 64 multiply-accumulate operations. When implemented using Newton-Raphson iteration, a division to 16 bit accuracy requires 5 multiply and 5 multiply-accumulate operations.

The reflection coefficient values *r(i)* are in the range

$$-1 \le r(i) \le 1 \tag{28}$$

To improve the quantisation characteristics, these are converted into approximate log-area ratios LAR(i), by the following set of rules:

$$\mathrm{LAR}(i) = r(i)\,;\ |r(i)| < 0.675 \tag{29}$$

$$\mathrm{LAR}(i) = sign(r(i)) \times (2|r(i)| - 0.675)\,;\ 0.675 \le |r(i)| < 0.950 \tag{30}$$

$$\mathrm{LAR}(i) = sign(r(i)) \times (8|r(i)| - 6.375)\,;\ 0.950 \le |r(i)| \le 1.000 \tag{31}$$

The conversion requires up to 2 comparisons and potentially one shift-and-subtract for each of the 8 reflection coefficients. The 8 log-area ratios have different distributions and dynamic ranges, and for this reason are encoded differently and with a different number of bits. The general formula for the encoding is

$$\mathrm{LAR}_C(i) = \mathrm{round}(A(i) \times \mathrm{LAR}(i) + B(i)) \tag{32}$$

with A(i) and B(i) varying to give quantisation of between 3 and 6 bits per parameter. These conversions require a total of 8 multiplications and 8 additions with rounding.

A.3 Short-term analysis filtering

Once the parameters of the LPC filter have been determined, the speech signal is passed through the inverse of the filter, to determine the signal at the output of the pitch filter (d(n) in Figure A.1). This is known as short-term analysis, as it removes local correlation between samples and produces either a pitch residual, or a noise-like signal for unvoiced speech.

To ensure that the rest of the encoding process is mirrored by the decoder, the same quantised log-area ratios sent to the receiver are used to form the inverse filter. Also, interpolation is performed between the current log-area ratios and those of the previous frame to prevent audible artefacts due to sudden changes in the estimated values from frame to frame. This interpolation requires 48 shift and addition operations. After this, the log-area ratios are converted back to reflection coefficient form, prior to being used in the short-term analysis filter. The short term analysis filter has the lattice structure shown in Figure A.2. Processing the signal through the filter requires sixteen multiply-accumulate operations per sample, with 2560 operations in total.

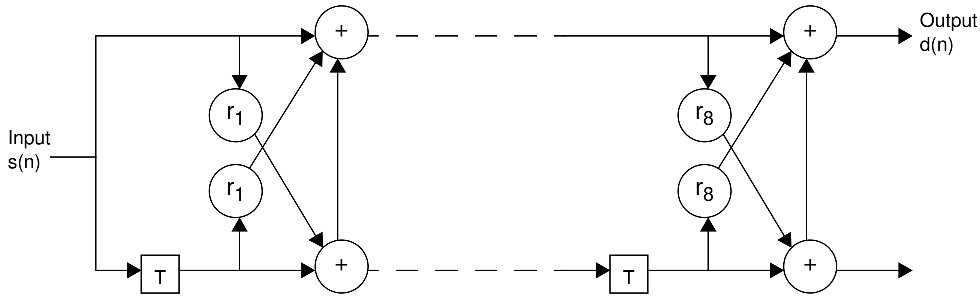


Figure A.2 Short-term analysis filter structure

A.4 Long-term prediction analysis

The short-term analysis filter is assumed to have removed the correlations corresponding to the frequency response of the vocal tract: the remaining signal is then the periodic excitation signal produced by the vocal chords. The long-term prediction filter has the equation in the Z-transform domain

$$P(z) = \frac{1}{1 - \beta z^{-\tau}} \qquad (33)$$

where the parameters β and τ describing the gain and period of the pitch filter. The period or *lag* τ of the signal is found by searching for a peak in the autocorrelation function of the signal (i.e. looking for the point of most self-similarity), after which the *gain* β can be calculated.

The signal is segmented into 4 blocks of 40 samples prior to LTP analysis, and values for the lag and gain are calculated and transmitted for each block. For each block *j=0...3*, the first step is to calculate the autocorrelation values of the block $d(k_j)$ with the reconstructed short-term residual signal from previous blocks $d'(k_j)$:

$$R_j(\lambda) = \sum_{i=0}^{39} d(k_j + i) \times d'(k_j + i - \lambda); \lambda = 40, 41, \ldots, 120 \qquad (34)$$

This process requires 3200 multiply-accumulate operations per block. The value λ which gives the maximum value of $R_j(\lambda)$ gives the estimate of the lag τ_j; the search for the maximum requires 80 compare operations per block.

Finally, the pitch gain value β_j for the block is calculated, using the equation

$$\beta_j = \frac{R_j(\tau_j)}{\sum_{i=0}^{39} d'^2(k_j + i - \tau_j)} \qquad (35)$$

This requires another 40 multiply accumulate operations, followed by a division operation: the pitch gain is subsequently quantised into 2 bits, so fewer iterations of the Newton-Raphson division algorithm (or other method) would be required. For each block, the lag is transmitted directly as a 7-bit value, while the pitch gain is quantised by a non-linear rule into two bits. The quantising requires a maximum of 3 comparisons per block.

The residual excitation $e(k_j)$ for the block *j* is calculated as the difference between the current signal and the result of the previously reconstructed short-term residual samples being passed through the LTP filter

$$e(k_j + k) = d(k_j + k) - d''(k_j + k) \qquad (36)$$

$$d''(k_j + k) = \beta_j \times d'(k_j + k - \tau_j) \qquad (37)$$

This operation requires 40 multiply operations and 40 subtractions per block.

A.5 Regular pulse excitation encoding

The final stage of the full rate speech coder attempts to find the sequence of 13 pulses, spaced at regular intervals, which produces the best match for the residual excitation sequence $e(k_j)$ for each block. Before this, a 'perceptual weighting' filter is applied to the residual excitation, to emphasise those components of the signal deemed most important. The impulse response $H(i)$ of the filter is convolved with the excitation:

$$x(k) = \sum_{i=0}^{10} H(i) \times e(k+5-i) \qquad (38)$$

This requires 11 multiply-accumulate operations per sample, with a total of 440 operations.

The filtered residual $x(k)$ is then split into 4 interleaved sub-sequences $x_m(i)$ of length 13, according to the rule

$$x_m(i) = x(k+m+3i); \; i = 0\ldots12, \; m = 0\ldots3 \qquad (39)$$

The optimum subsequence $X_M(i)$ is the one with the maximum energy E_M:

$$E_m = \sum_{i=0}^{12} x_m^2(i) \qquad (40)$$

The search for the maximum energy requires 52 multiply-accumulate operations and 4 comparisons. The optimum grid position *M* is encoded directly using 2 bits.

The 13 samples in the selected subsequence (the *RPE sequence*) are quantised using adaptive pulse-code modulation. First, the maximum value is found (requiring 13 comparison operations). The maximum value is quantised in base 2 logarithmic form with 6 bits, requiring up to 6 additions. The 13 samples are then divided by the quantised value, which can be done with a multiplication followed by a shift due to the logarithmic encoding of the maximum value. These results are quantised using 3 bits each.

The final stage of processing for each block is to use the quantised values of the lag and the RPE sequence to generate the excitation signal as received by the decoder, to be used by the encoder for subsequent blocks. Dequantization of the RPE sequence requires 13 multiply-accumulate operations and some shifting. The reconstructed long-term residual signal is then produced by inserting zeros between the 13 RPE samples according to the grid position M. Finally, the reconstructed short-term residual signal for this block $d''(n)$ must be produced by adding the reconstructed long-term residual to the estimated pitch signal. This requires 40 additions.

Appendix B: Instruction set

- Notation

(X | Y | Z)... either X, Y, or Z (without brackets)

[X | Y | Z]... optional X, Y, or Z (without square brackets)

`exec #operation`

`0000 0000 0000 0000 0000 0000 0000 0000`

This instruction causes the stored parallel instruction specified by **operation** to be executed. The encoding for **operation** is shown in Table B.1.

Bit position	Function
0-6	Operation select
7-13	Operand / load-store / index select
14	Load/store enable
15	Global enable parallel accumulator write
16	Global enable writeback
17	Enable index register update
18-22	Condition code bits
23-26	Enable operations 1-4
27-30	Conditional operation 1-4

Table B.1: Parallel instruction operation specification

`MOVEM #a,#b,#c,#d,(i|mi|ni|j|mj|nj)`

`1jnm dddd dddc cccc ccbb bbbb baaa aaaa`

Move-multiple of immediate constants a,b,c,d to the index / update / modifier registers specified according to the code `jnm`:

`jnm`	Target	
`000:`	i0,i1,i2,i3	Index registers
`100:`	j0,j1,j2,j3	
`010:`	ni0,ni1,ni2,ni3	Update registers
`110:`	nj0,nj1,nj2,nj3	
`001:`	mi0,mi1,mi2,mi3	Modifier registers
`101:`	mj0,mj1,mj2,mj3	

MOVE #d,(rn|mrn|nrn)

`1011 umnn dddd dddd dddd dddd dddd dddd`

Move 24-bit immediate value **d** to the address / update / modifier register specified by **n**. Bits **u**,**m** select the update or modifier register **nrn** or **mrn** as the destination (these are mutually exclusive).

JMP #dddddd

`1111 0000 dddd dddd dddd dddd dddd dddd`

Unconditional jump to 24-bit address **dddddd.**

JSR #dddddd

`1111 0001 dddd dddd dddd dddd dddd dddd`

Unconditional jump to subroutine with 24-bit address **dddddd**, with the value PC+1 being placed on the internal stack within the branch unit.

BRAcc #offset,#n

`1111 0010 0ccc ccnn oooo oooo oooo oooo`

Conditional branch: add 16-bit 2's complement value **offset** to PC if the condition codes of functional unit **n** meet the condition specified by **cc**.

BSRcc #ooooxx

`1111 0011 0ccc ccnn oooo oooo oooo oooo`

Conditional branch to subroutine: if the condition codes of functional unit **n** meet the condition specified by **cc**, then push PC+1 onto the stack and branch.

RTS

`1111 0100 xxxx xxxx xxxx xxxx xxxx xxxx`

Restore PC from stack. (xxxx's are don't cares)

HALT #mask

`1111 1111 0000 0000 0000 0000 0000 00MM`

Stop processing until interrupted: the mask bits `MM` specify which cooperative interrupts (int0, int1) the processor will respond to.

`DO #i, #m`

`1111 1100 000m mmmm iiii iiii iiii iiii`

Zero-overhead hardware DO loop. Execute the following m instructions i times. On entering the loop, the current loop status is put onto the stack, to allow for nested DO loops. On exiting the loop (either through BREAK or by the loop count reducing to zero) the loop status is restored. Branches (either software or interrupt) cause the DO status to be flushed.

`DO (i|ni|mi|j|nj|mj|a)r, target`

`1111 1101 000o oooo mna0 0000 0000 0jrr`

As for previous zero-overhead DO, except that the loop counter comes either from one of the 7-bit index / update / modifier registers or the least significant 16 bits of one of the address register according to bits `mna`:

`mna`	
`000`	Index register
`100`	Modifier register
`010`	Update register
`001`	Address register

`BREAKcc #n`

`1111 0111 0ccc ccnn 0000 0000 0000 0000`

If condition code **cc** within funtional unit FU **n** is met, then restore the loop status from the loop stack and continue at the end of the current loop.

`ADD #m,rn`

`1111 10nn mmmm mmmm mmmm mmmm mmmm mmmm`

Add the 24-bit 2's complement immediate value **m** to address register **rn** (affected by the contents of the modifier register **mrn**).

`MOVE (rk|nrk|mrk),(rl|nrl|mrl)`

`1111 0110 0kkn mlln m000 0000 0000 0000`

Move the source address / update / modifier register **k** into the destination address / update / modifier register **l**.

`add #n,(id|jd)`

`1111 0110 1ddd 0110 0000 0000 0nnn nnnn`

Add the immediate value **n** to the index register value **id / jd** (affected by the current modifier **mid / mjd**).

`sub #n,(id|jd)`

`1111 0110 1ddd 0100 0000 0000 0nnn nnnn`

Subtract the immediate value **n** from the index register value **id / jd** (affected by the current modifier **mid / mjd**): ddd = 0-3->i0-i3, 4-7->i0-i7

`lsl (id|jd)`

`1111 0110 1jdd 1100 0000 0000 0XXX XXXX`

Shift the the index register value **id / jd** left by one position (X's are don't cares).

`lsr (id|jd)`

`1111 0110 1jdd 1110 0000 0000 0XXX XXXX`

Shift the the index register value **id / jd** right by one position (X's are don't cares).

`MOVE #immed,(id|nid|mid|jd|njd|mjd)`

`1111 0101 0jdd nm00 0000 0000 0iii iiii`

Move the 7-bit immediate value into index / update / modifier register d.

`MOVEM #immed,(i|ni|mi|j|nj|mj)`

`1111 0101 0j00 nm01 0000 0000 0iii iiii`

Move the 7-bit immediate value into index / update / modifier registers: bit `j` selects j registers, bit `n` selects update registers, bit `m` selects modifier registers.

`MOVEM (is|nis|mis|js|njs|mjs),(i|ni|mi|j|nj|mj)`

`1111 0101 1j00 nm01 0jss nm00 0000 0000`

Move single index / update / modifier register **s** into multiple index / update / modifier registers. Bits `jssnm` select source register, bits `j(00)nm` select destination registers.

`MOVE (is|nis|mis|js|njs|mjs),(id|nid|mid|jd|njd|mjd)`

`1111 0101 1jdd nm00 0jss nm00 0000 0000`

Move index / update / modifier register **i/js** into index / update / modifier register **i/jd**.

`config FUNCTION/OPERAND #start,#count`

`1111 1111 0000 0000 00sss ssss fccc cccc`

Reads the subsequent **count** x [4/6] words and stores them in successive configuration memory locations from **start**. Choice of memories given by f:

```
0: Functional unit opcode memories (4)
1: Operands, load-store and index update (6)
```

Appendix C: The index register units

C.1 Index unit structure

The top-level schematic of the index unit is shown in Figure C.1. Operation of the index unit at this level is managed by the *index_unit_ctl* module. This controls the datapath signals for writes to and reads from the index registers, and requests updates from the ALU (*index_alu*) when appropriate. The index unit controller supports 5 different operations: index register updates requested by *nreq_index*, write-multiple to index registers requested by *nreq_wrm*, writes to a single index register requested by *nreq_indwr*, and ALU operations with immediate data requested by *nreq_indop*. All of these operations are acknowledged by *ack_index*.

The request signals are common to all 8 of the index register units, and enable signals indicate which index units should respond to them. Single writes, reads and immediate ALU operations are enabled by the *index_sel* signal, and only the single enabled index unit performs the update and issues an acknowledge. In the case of write-multiple and index update operations, all units issue an acknowledge; but only those that are enabled actually perform an operation. Write-multiple operations are enabled by *wrmsel[2]*, which selects whether the i or j registers are the target, while index update operations are enabled by *upd[0]*, from the index update configuration memory.

The register values themselves are stored in the three latches (*std_svensson* and *dffr*) to the left of the ALU. The update and modifier registers are stored in the level-sensitive svensson latches, while the index register is stored in an edge-triggered register.

Writes to the registers are handled similarly whether triggered by a write-multiple or single write instruction. The only differences between the two cases are the source of the immediate data and the specification of update or modifier registers as the target. For a write-multiple operation, the data enters on *wrm[6:0]* and the update / modifier specification comes from *wrmsel[1:0]*. For a single write, data enters through *immed[6:0]* and the update / modifier specification is made by *index_update* and *index_mod*. The signal *nsel_wrm* selects the appropriate source at the start of the operation. The input

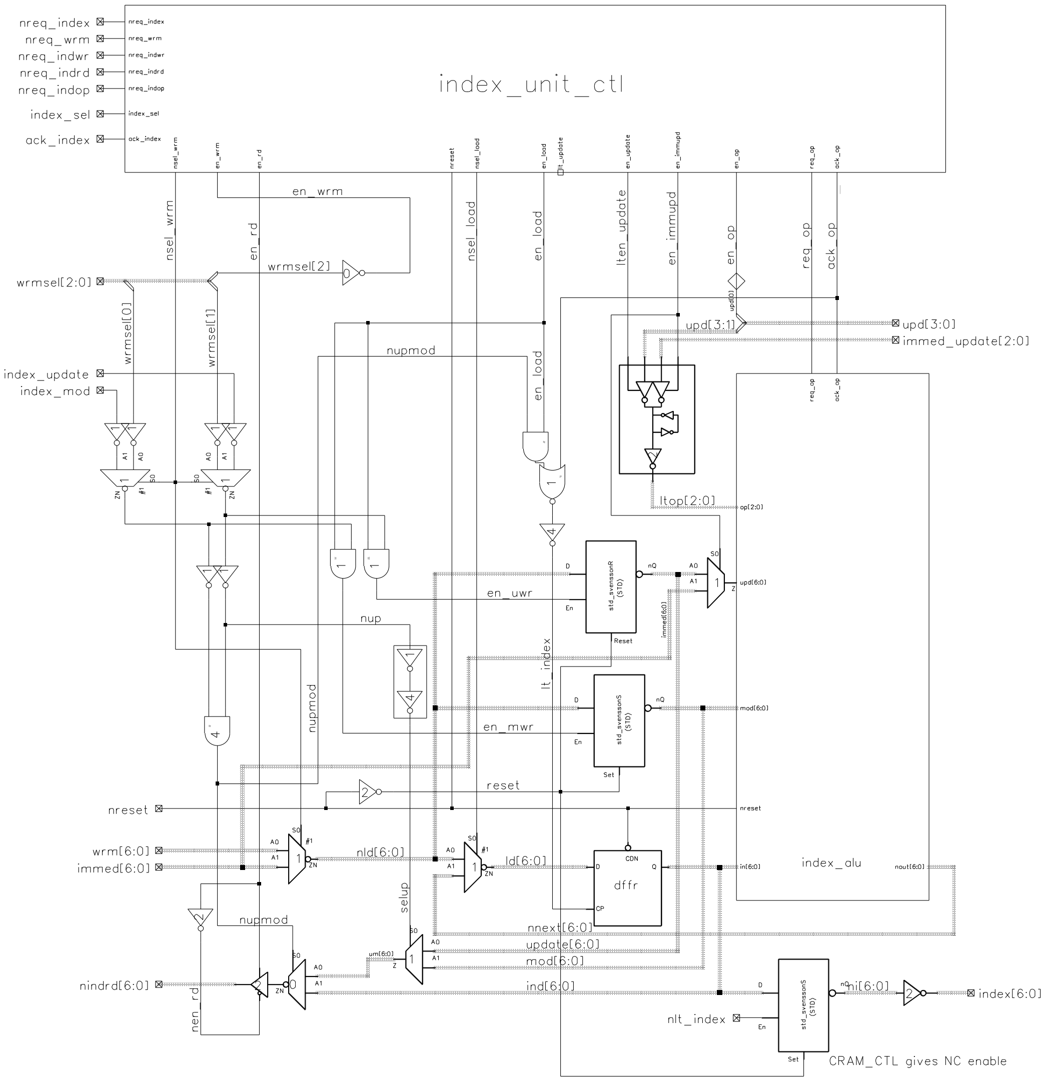


Figure C.1 Index unit schematic

signals select either the index, update or modifier register to be enabled to respond to *en_load*, which is driven high by the control unit to capture the data.

Reads from the index registers (performed when moving one register value to another register) are performed across a shared bus. The index, update or modifier value required is selected by *index_update* / *index_mod* and passed to a tristate driver. On receiving *nreq_indrd*, the enabled index unit asserts *en_rd* which causes the selected value to be driven onto the output bus.

Updates to index registers as part of a parallel instruction and updates using immediate data occur in a similar manner, differing only in the source of their data: in a parallel instruction, the operation to be performed is selected by *upd[3:1]* from the configuration memory and the operands are the index and update register values. For an immediate update, the operation to be performed is specified in the instruction on *immed_update[2:0]* and the operands are the index register and immediate data. In both cases, the modifier register value may affect how the operation is performed.

To prevent large numbers of spurious transitions within the index units due to unrelated instructions, the operation selection value is latched before passing into the ALU. When an index update is to be performed, either *lten_update* (for parallel instructions) or *en_immupd* (for immediate updates) is driven, which passes the appropriate value to the ALU. The operation is then requested through the *req_op* / *ack_op* handshake with the ALU, and the result is captured when *ack_op* goes high.

C.2 Index ALU operation

The schematics of the index register ALU is shown in Figure C.2. This forms a separate asynchronous module and has its own control circuit (*index_aluctl*). The remainder of the circuit is the datapath, and consists of four main elements: the adder / comparator (*index_add*), input selection logic for the adder, a carry-save adder that adds together the two operands and an optional adjustment value, and a circuit to determine the split point based on the modifier register value (*index_modmask*). The modifier register is only changed by specific writes, and the outputs of *index_modmask* are guaranteed to have stabilised before index updates occur. The operation to be performed on the index register is selected by *op[2:0]*, which affects how the input is set up and how the datapath responds to signals from the control unit. The encoding for the various operations is given in Appendix D on page 267.

The basic sequence of events is the same for all arithmetic operations. Initially, only the index register and the value to be added are presented to the carry-save adder (*add_off* is low). The sum and carry values are passed to the main adder, which resolves the carries and calculates the sums above and below the split point. The timing of this is managed by a matched delay from *req_add* to *ack_add*. The result of the operation below the split

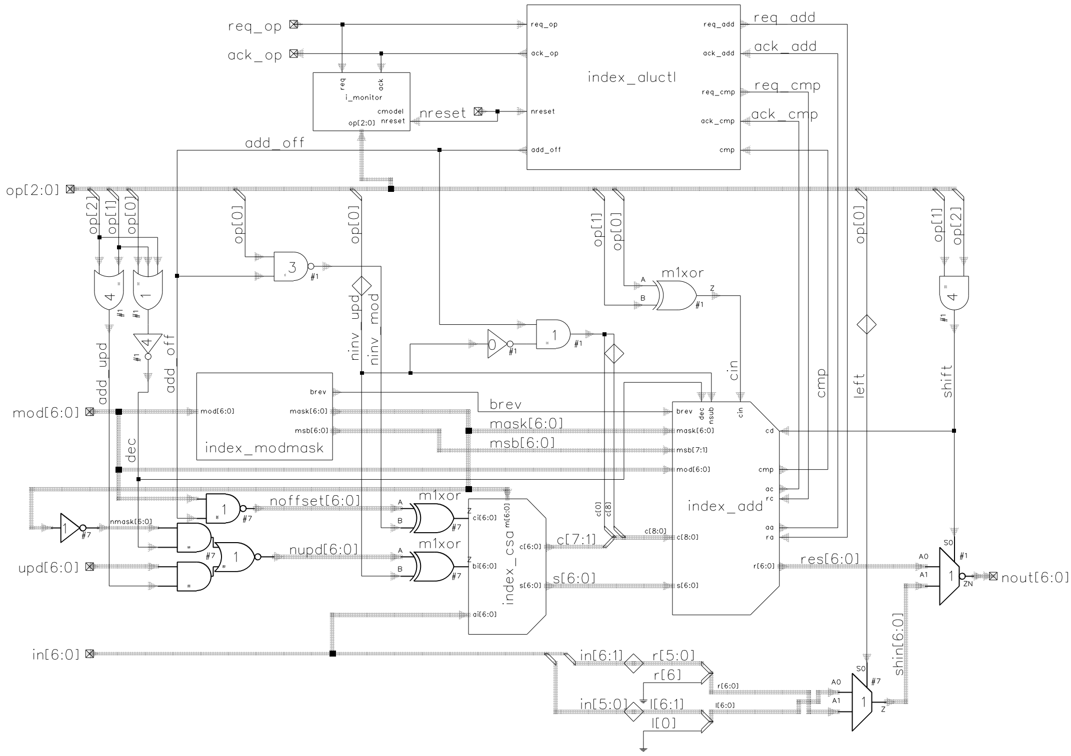


Figure C.2 Index ALU schematic

point is then compared to the modifier value (circular buffer bound), timed by *req_cmp* / *ack_cmp*. An overflow is indicated by the signal *cmp*.

If no overflow is detected, then the update process is complete and the control unit issues an acknowledge. However, should an overflow be detected then *add_off* is set high. This causes the appropriately-signed offset to be presented to the carry-save adder to bring the result back within the limits of the circular buffer. The carries are resolved again in the main adder, after which the result is available.

Bit-reversed addressing is indicated by the signal *brev* being set by *index_modmask*. This causes the carry chain in the main adder to be reversed. Since all bits in the modifier are clear, the result of the comparison always indicates that no overflow has occurred (the split point is below the least-significant bit).

For shift operations, the appropriately shifted input is simply multiplexed onto the output. The control unit still issues *req_add* and *req_cmp*, but the comparison result is disabled so another cycle is never started. The time taken by the two delays is excessive for a shift operation. However, it was felt that there would be insufficient benefit gained from faster shift operations to justify the extra complexity of designing the control circuit to deal with shifts differently.

C.2.1 Split adder / comparator design

The circuit for the split carry chain adder and comparator unit, *index_add,* is shown in Figure C.3. The signal *msb[7:0]* indicates the split point of the carry chain, while *mask[6:0]* is used to select only those bits below the split point. The carry out at the split point is passed out on tc[6:0], and this is used along with the output of the comparator to determine whether the circular buffer range has been exceeded. The input *dec* indicates a decrement operation while *nsub* indicates a subtraction, which alters the sense of the carry out. Bit reversed addressing is selected by *brev*. Since the input comes from a carry-save adder, it is necessary to shift the carry input right by two places when performing bit-reversed addressing to reverse the direction of carries.

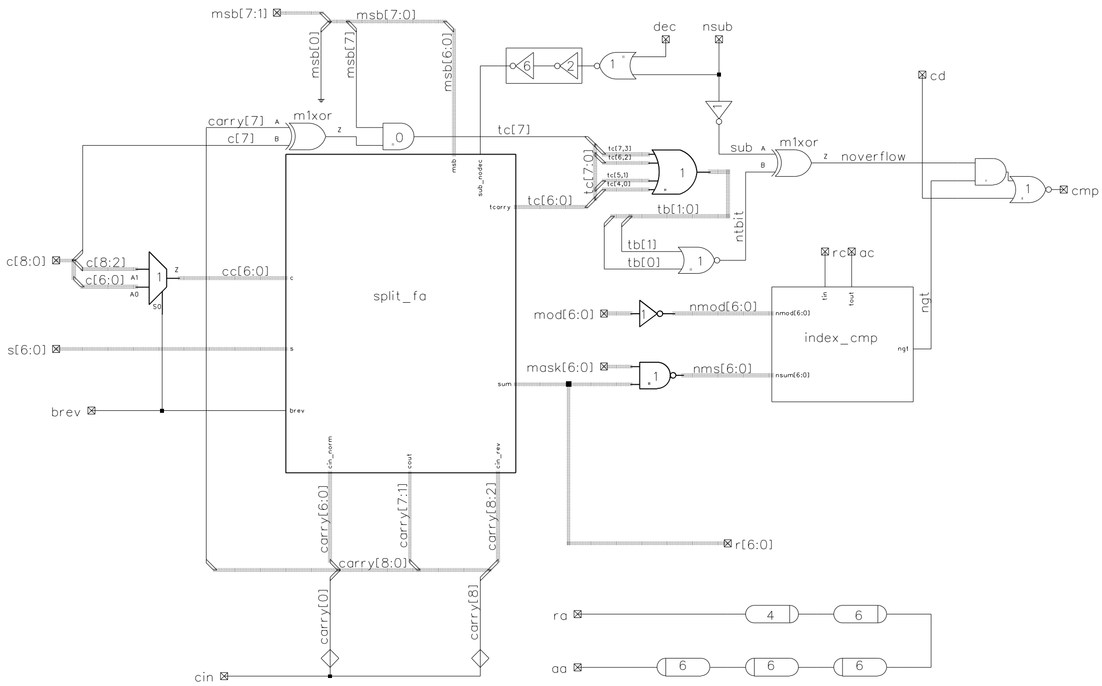


Figure C.3 Split adder and comparator schematic

The circuit used to implement the split adder is depicted in Figure C.4. The brev signal controls whether the forward or backward carry signal is to be selected to form the carry input *cin*. The *s* and *c* inputs are the sum and carry inputs from the carry-save adder, while *sub_nodec* is used to ensure that the correct outputs are generated from the bit position directly above the split point when either a subtraction or a decrement is being performed.

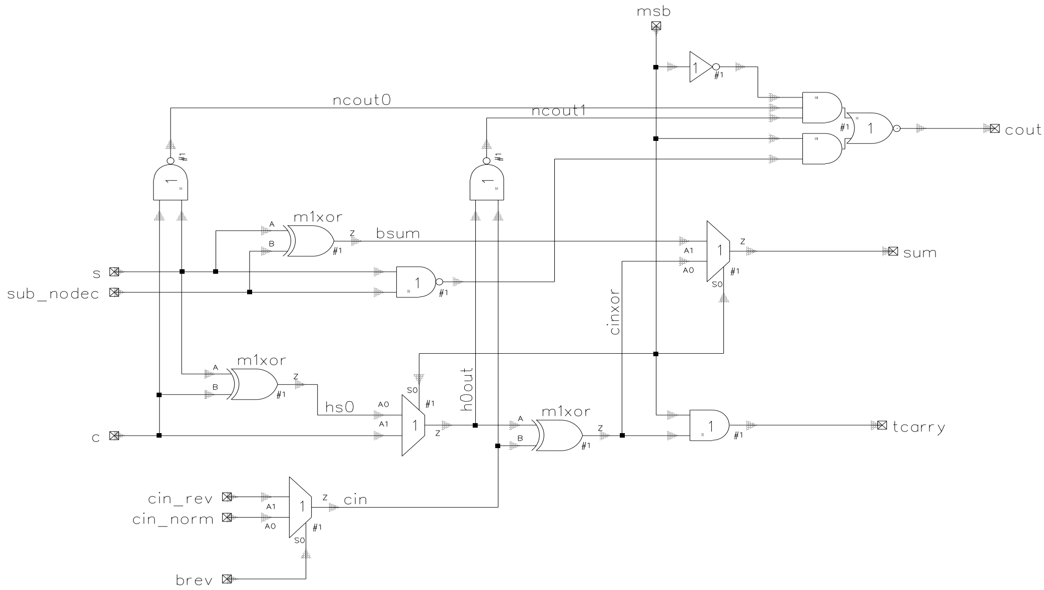


Figure C.4 Full adder with bidirectional split carry chain

When *msb* is low, the circuit behaves as a conventional full adder: *c* and *s* are XORed to produce the sum result from the first half-adder, *hs0*. This is XORed with the carry input *cin* to produce the final sum. Similarly, the carry out (*cout*) is produced by the combination of the carries produced by the two half-adders (*ncout0* and *ncout1*).

A high value on *msb* indicates that the carry splits at this position. The carry-save adder at the input means that the carry input *c* is from the most-significant bit beneath the split point, while the sum input *s* is from the least-significant bit above the split point.

In this case, *sum* and *cout* are formed from the result of a half-adder between the sum input and the subtraction / decrement adjustment value; i.e *sum* and *cout* are the least significant outputs of the result above the split point. The output *tcarry* is the most significant bit of the result below the split point. This is formed from *c* and *cin* being XORed together, and is enabled by *msb* so that only the result at the split point affects detection of circular buffer overruns.

C.2.2 Verification of index ALU operation

The arithmetic for circular buffering is complex, so to give confidence in the correctness of the design an extensive set of tests were performed. A simulation test harness was produced for the index ALU to feed in random index, modifier and update values (with the index and update values within the proper ranges for each chosen modifier value). Random operations were selected in each case, and correctness of the result was checked. No errors were found in 100,000 different operations, giving reasonable confidence that the design is correct.

Appendix D: Stored opcode and operand configuration

D.1 Functional unit opcode configuration

The configuration words for the functional units are entirely dependent on the implementation of each functional unit: the rest of the processor makes no assumptions about how this data will be interpreted. However, the functional unit implemented for this work has the structure shown in Figure D.1.

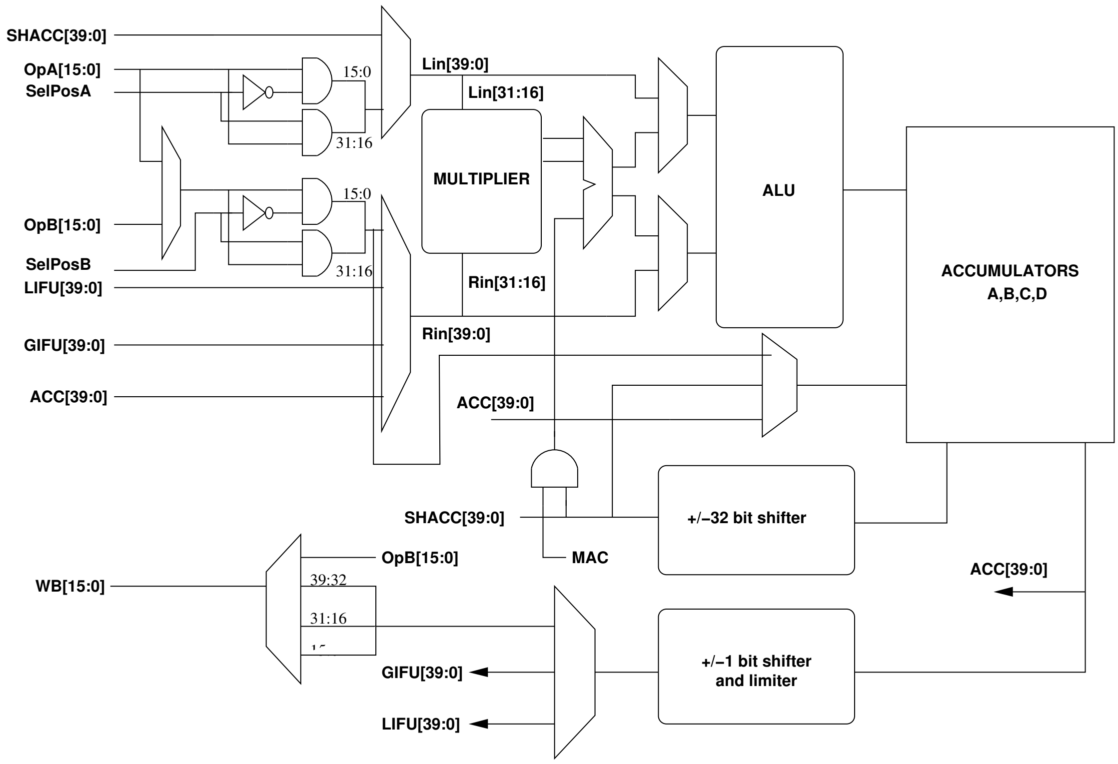


Figure D.1 Functional unit internal structure

Bit Postions	Function
0	Left input select (SHACC / OpA)
1	Left OpA bus position (OpA[15:0] -> Lin[31:15]/Lin[15:0])
2-3	Right input select (OpA.B/LIFU/GIFU/ACC) 00 ACC 01 GIFU 10 LIFU 11 OpA.B
4	Right OpA/OpB select
5	Right Op bus position (OpA/B[15:0] -> Rin[31:16]/ Rin[15:0])
6-10	Opcode... see Table D.3
11	Set condition code
12	Left input sign invert
13	Right input sign invert
14	SHACC shifter direction (1 = left, 0 = right)
15-19	SHACC shift distance
20	SHACC invert
22-21	ACC shifter control 00 No shift 10 Shift Left 01 Shift Right 11 Conditional Shift
23	ACC limiter on/off
24-25	ACC Shift/limiter output 00 NONE 01 Writeback 10 LIFU 11 GIFU
26-27	Writeback Source 00 OpB 01 ACC[15:0] 10 ACC[31:16] 11 ACC[40:32]
28-29	ACCWR source: 00 No write 01 Op 10 ACC 11 SHACC
30	Enable Writeback
31	Unused

Table D.1: Functional unit opcode configuration encoding

SM0,SM1	Scaling mode	Set by SCLNONE, SCLUP, SCLDOWN: affects the way that rounding, E/U bit, and automatic ACC shifting works: 00 No scaling 01 Scale up 10 Scale down
S	Scaling bit	Set when data growth is detected, according to the scaling mode.
L	Limit bit	Set when the ACC limiter produces a limited result.
E	Extension bit	Set when the last result written to the accumulators has a non-zero extension section (dependent on scaling).
U	Unnormalized bit	Set when the MSP bit (bit 30,31 or 32 depending on scaling mode) is not set.
Z	Zero bit	Set if the result is zero.
C	Carry bit	Set if a carry is generated out of the result, or a borrow occurs.
N	Negative bit	Set if the result is negative

Table D.2: Functional unit condition codes

00000	MPY	10000	DISTANCE
00001	MAC	10001	AND
00010	ADD	10010	OR
00011	ADC	10011	XOR
00100	MPYR	10100	NORM
00101	MACR	10101	ASHIFT
00110	ADDR	10110	LSHIFT
00111	ADCR	10111	Reserved
01000	CMP	11000	SCLNONE
01001	CLIP	11001	SCLUP
01010	ABSMAX	11010	SCLDOWN
01011	ABSMIN	11011	Reserved
01100	MAX	11100	Reserved
01101	MIN	11101	Reserved
01110	SIGN1	11110	Reserved
01111	SIGN2	11111	NOP

Table D.3: Opcodes

D.1.1 Arithmetic operations

The inputs to these operations are treated as sign-magnitude numbers, and the SHACC shifter performs arithmetic shifts.

`mpy / mpyr lin,rin,dest`

Multiply / multiply with rounding the left input **lin** by right input **rin**, writing result to accumulator **dest**.

`mac / macr lin,rin,shacc,dest`

Multiply-accumulate / MAC with rounding

`add / addr lin,rin,dest`

Add / add with rounding

`adc / adcr lin,rin,dest`

Add with carry / with rounding: an offset of +1 / 0 / -1 is set depending on the state of the C and N flags. This allows extended-precision 40 bit signed digit arithmetic.

`cmp lin,rin`

Compare left and right inputs, and set the flags according to the result (does not perform a subtraction if the signs differ).

`clip lin,rin,dest`

If magnitude of right input is greater than the magnitude of the left input, then write the left input to the destination, otherwise clip the magnitude of the left input to that of the right input.

`absmax / absmin lin,rin,dest`

Write the destination with whichever input has the absolute maximum / minimum value, and set the condition codes accordingly.

`max / min lin,rin,dest`

Write the destination with whichever input has the signed maximum / minimum value, and set the condition codes accordingly.

`sign1 lin,rin,dest`

Write the right input to the destination, with its sign set to be the same as that of the left input.

`sign2 lin,rin,dest`

Write the right input to the destination, with its sign set to be the same as that of the left input, unless the left input is zero in which case write zero to the destination.

D.1.2 Logical operations

The inputs to these operations are treated as unsigned binary numbers (i.e. no special treatment of the sign bit is made) and the SHACC shifter performs logical shifts (with the exception of the ASHIFT instruction).

`distance lin,rin,dest`

The lower 6 bits of the destination are written with the Hamming distance between the two inputs.

`and / or / xor lin,rin,dest`

Standard 40-bit logical operations.

`norm lin,dest`

The right shift that needs to be performed to normalise the left input (i.e. put a '1' in bit 30) are written to bits 0-4 of the destination, with bit 15 and the sign bit of the result being set if the result is negative (a left shift is needed). If the input is non-zero, a 'one' is written into bit 14 of the result, while otherwise the result is zero.

`ashift shacc,rin,dest`

Perform an arithmetic shift of the value on SHACC by 'rin' places to the right (or left if 'rin' is negative) and write the result to the destination. This overrides the SHACC shift value specified in the opcode.

`lshift shacc,rin,dest`

Perform a logical shift of the value on SHACC by 'rin' places to the right (or left if 'rin' is negative) and write the result to the destination. This overrides the SHACC shift value specified in the opcode.

sclnone maca-d

Set the scaling mode in the selected functional unit to 'no scaling'.

sclup maca-d

Set the scaling mode in the selected functional unit to 'scale up'.

scldown maca-d

Set the scaling mode in the selected functional unit to 'scale down'.

D.1.3 Conditional execution

The encodings for conditions for conditional execution is partially independent of the implementation for the functional units: codes between 01001-01111 and 11001-11111 are loop conditionals and are never seen by the functional units as they are interpreted earlier on. The interpretation of the other codes depends on the implementation of the functional units, and are as shown in Table D.4.

00000	AL: always	10000	NV: never
00001	CC: Carry clear (C=0)	10001	CS: Carry set (C=1)
00010	EC: Extension clear (E=0)	10010	ES: Extension set (E=1)
00011	NC: Normalize clear (N=0)	10011	NS: Normalize set (N=1)
00100	LC: Limit clear (L=0)	10100	LS: Limit set (L=1)
00101	SC: Scale clear (S=0)	10101	SS: Scale set (S=1)
00110	GT: Greater than (Z+N=0)	10110	LE: Less-equal (Z+N=1)
00111	PL: Plus (N=0)	10111	MI: Minus (N=1)
01000	NE: Not equal (Z=0)	11000	EQ: Equal (Z=1)
01001	LOAD/STORE nfirst	11001	LOAD/STORE first
01010	Writeback nfirst	11010	Writeback first
01011	Arithmetic op. nfirst	11011	Arithmetic op. first
01100	Reserved	11100	Reserved
01101	LOAD/STORE nlast	11101	LOAD/STORE last
01110	Writeback nlast	11110	Writeback last
01111	Arithmetic op. nlast	11111	Arithmetic op. last

Table D.4: Condition encoding for conditional execution

D.2 Stored operand format

The format of stored operands are partly dependent on the implementation of the functional units as bits 15-22 are interpreted by the functional unit. The remaining bits are interpreted in other portions of the architecture and are therefore fixed.

Bit position	Function
0-3	A Index, X/Y
4-7	B Index, X/Y

8-10	Immediate select
	`000: AB both index`
	`001: 8 bit immediate opA`
	`010: 8 bit immediate opB`
	`011: AB long immediate`
	`100: AB both index, Writeback immed.`
	`101: 8 bit direct reg opA`
	`110: 8 bit direct reg opB`
	`111: 8 bit direct reg both`

11-14	Writeback Index, X/Y
15-16	ACC src
17-18	SHACC src
19-20	Op destination
21-22	ACCWR destination
23	Enable register file reads
24-31	Immediate value

Table D.5: Operand Format stored in Operand configuration memory

The value defined in bits 24-31 can be used either as an immediate value for one of the inputs, as a direct register specification for one input, or as a direct register specification for writeback. Alternatively, bits 24-31 and bits 0-7 can be combined to form a 16-bit immediate value, which is used for both inputs, or two separate 8-bit direct register specifications can be made.

D.3 Index update encoding

The updates for the 8 index registers i0-i3 and j0-j3 are encoded in 32 bits, with the update code for i0 in bits 0..3, i1 in bits 4..7, etcetera. The meanings for the codes are given in Table D.6.

0	Enable update

1-3	Op select
	`000 Postdecrement`
	`001 Postincrement`
	`010 Postdecrement by n`
	`011 Postincrement by n`
	`100 Postdecrement by (n+1)`
	`101 Postincrement by n+1`
	`110 Shift left`
	`111 Shift right`

Table D.6: Index register update codes

D.4 Load / store operation

The load / store configuration memory contains the selection of the data register to be the destination or source and the address register to be used for each of the X and Y operations. The register selection can be either a 7-bit immediate value, an indirect reference through an index register, or a store can be performed from the GIFU (bypassing the register bank and simplifying stores of long accumulator values). Also, update codes are specified for both of the selected address registers: it is the programmers responsibility to avoid simultaneous updates to the same address register. Details of the encodings are shown in Table D.7.

Bit position	Function

0-6	X index register / register select
	`When indexed / GIFU:`
	`0..2  Index register select`
	`3     GIFU select`

7-13	Y index register / register select
	`When indexed / GIFU:`
	`0..2  Index register select`
	`3     GIFU select`

14	X indexed / GIFU
15	Y indexed / GIFU
16	X long
17	Y long
18	Xdir: 0=load,1=store
19	Ydir: 0=load,1=store
20	X enable
21	Y enable
22-23	X address reg select
24-26	X address reg update mode 000 Decrement 001 Increment 010 Ri-nRi 011 Ri+nRi 100 ASL 101 ASR 111 NOP
27-28	Y address reg select
29-31	Y address reg update mode

Table D.7: Load/store operation format