Processor Architectures for Power Efficiency and Asynchronous Implementation

A thesis submitted to the University of Manchester for the degree of Master of Science in the Faculty of Science

Philip Brian Endecott

Department of Computer Science

1993

Contents

List of Figures

List of Tables

Abstract

Portable battery-operated computing equipment requires high processing performance and low power consumption. Other computer systems may need low power consumption for other reasons such as overcoming the problem of heat transport away from the processor chip. This thesis investigates ways in which processor architecture can influence power efficiency.

One implementation technique which may lead to lower power consumption is the use of asynchronous logic. In particular, asynchronous logic can be more power-efficient in a system with a rapidly-changing computational load. Because of the differences between synchronous and asynchronous logic, certain architectural features are more suited to asynchronous implementation than others. This thesis proposes a number of features that are more suitable for asynchronous implementation. Important areas include the branch mechanism and the way in which data dependencies are dealt with.

Other architectural factors that influence power consumption are investigated. Increasing code density will lead to increased power efficiency because the power consumed in many parts of the computer system is proportional to the rate of instruction fetch. Code density is investigated and ways in which the density of the SPARC architecture could be increased are proposed. The most important improvements are found to result from using a Huffman encoded opcode field and reducing the length of some immediate fields. Other factors are the use of 2-address instructions, load and store multiple instructions and explicit last result re-use.

It is also noted that the power efficiency of the memory system can be increased through the use of multiple level caches and by using a copy-back write policy.

By combining these features, an architecture with a power efficiency double that of a conventional processor could be constructed.

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 institute of learning.

Acknowledgements

Over the last year I have received a great deal of help from very many people, without which the work described in this thesis would have been impossible.

My supervisor Prof. Steve Furber has been a great source of inspiration. I would like to thank him for his constant interest in my ideas and encouragement.

The other members of the AMULET research group have provided me with a stimulating environment in which to work. I am grateful to them all for helping me to understand asynchronous logic and for listening to my ideas.

Much of my data has been collected using the SPARC simulator ‘Shade’ and the GNU C Compiler. These programs have been made available free of charge, and I would like to thank those people who have made this possible, including everyone at the Free Software Foundation and Bob Cmelik at Sun.

I am also grateful for help and information given by GCC users, computer architects and others through the USENET NEWS system. I am especially grateful to Jim Wilson for his assistance with GCC, and John Fitch for providing information about the DEC ECL BIPS processor.

Several people have helped by reading and commenting on drafts of this thesis. I must extend my thanks to Viv Woods for his useful comments, and to my father Richard Endecott and to Janette Taylor for proofreading.

Thanks, everyone.

The Author

Philip Endecott obtained a B.Sc. (I) degree in Computer Science from the University of Manchester, England, in 1991. This thesis is the result of the first year of research as a member of the AMULET research group at the University of Manchester.

AMULET (Asynchronous Microprocessor Using Low Energy Techniques) comprises four projects looking at different areas where asynchronous logic techniques can be applied.

Chapter 1 : The Increasing Importance of High Power Efficiency

This thesis is about designing computer architectures whose implementations can have low power consumption. This introductory chapter examines the reasons for wanting low power consumption and the techniques that can be applied to achieve it.

The body of the work concentrates on two techniques that can be used to improve power efficiency. Firstly the idea of using asynchronous logic is considered, and secondly the influence of processor architecture on power efficiency is studied. The objective is to define a set of architectural features that permit an asynchronous power-efficient implementation.

1.1 Why Low Power?

The most well-known motivation for the design of power-efficient processors is for battery operated portable applications. As well as this, power efficiency is important or will become important in some applications because of the problems of thermal management and power supply. A third motivation is the issue of electricity cost and the environmental impact of power consumption.

These three issues are investigated in this section.

New Application Areas Need Low Power

Increased performance from physically small computer systems has opened up a number of application areas for portable battery powered equipment. Examples include:

- **Portable personal computers.** It is not unusual to find portable personal computers with processing performance, storage capacity and screen resolution closely comparable to desktop machines.
- **Personal Digital Assistants (PDAs).** This new class of portable computer demands a high-performance processor for functions such as handwriting recognition.

- **Hand-held video games.** Again, performance can be similar to non-portable equipment.
- **Personal Digital Hi-Fi.** The processing power needed to control a compact disk or digital cassette player is significant.
- **Portable Radio Telephones.** The demand to accommodate more channels within the same bandwidth for cellular telephones means that sophisticated signal compression techniques are being applied, with considerable processing requirements.
- **Global Positioning by Satellite (GPS) systems.** Previously used only by the military and mariners, GPS is now becoming cheap and small enough to use in cars; soon perhaps by mountaineers as well.
- **Hand-held photo and video equipment.** Successful digital photo and video cameras need good image sensors, high density storage and sufficient processing power to apply the necessary image compression algorithms.

Users of these products demand low weight and size and long operating periods between battery changes or recharging. Often, batteries make up a significant proportion of the size and weight and yet users are not satisfied with the battery life. To improve the situation, either the energy density of the battery must be increased, or the power consumption of the system must be reduced. Battery technology will improve, and in the period 1946 to 1980 battery energy density doubled approximately every ten years [2], but the growth in demand is likely to exceed the growth in battery capacity.

Approximate values for the energy density and cost of some batteries used in portable equipment are given in table 1.1 [2] [3]. Note that the battery with the highest energy density, the lithium battery, has a capacity considerably greater than the alkaline battery but the cost grows even faster. Note also that rechargeable batteries have an energy density around three times smaller than that of non-rechargeables. In order to keep the cost of the batteries needed at a reasonable level, the most advanced technology must be avoided.

The only alternative to increasing battery capacity is to reduce the power consumption of the equipment. This includes building more energy-efficient processors.

Some might expect that in the applications listed above components such as motors, speakers, RF transmitters etc. would consume far more power than the control electronics. However in practice the digital electronics actually consumes a significant proportion. For example, in a portable compact disk player approximately 50% of the power consumption is in the integrated circuits [1].

Battery	Energy density $\mathrm{kJ\,kg^{-1}}$	Cost $\mathrm{pence\ kJ^{-1}}$
Primary (non-rechargeable) cells		
Zinc carbon	230	2.6
Zinc chloride	270	2.2
Alkaline	340	3.0
Lithium	1200	12.7
Secondary (rechargeable) cells		
Lead acid	108	
Nickel cadmium	111	

Table 1.1: *Batteries used in portable equipment (approximate)*

Thermal and Electrical Issues

Although the obvious applications needing power-efficient processors are the battery operated ones described above, it is probable that it will soon be necessary to apply the same low-power techniques to mains-powered desktop systems.

Since the beginning of 1992, DEC has announced two processors with very high power consumption: a CMOS implementation of the Alpha architecture that consumes 30W [4], and an ECL implementation of the MIPS architecture that consumes 115W [5]. Although this power consumption is currently considered to be high, future processors will inevitably be announced with still higher power consumption.

How fast will power consumption increase? If current trends continue, the feature size used in integrated circuit fabrication will continue to decrease by about 12% per year [6]. If this decrease in feature size is not accompanied by a decrease in supply voltage (as was the case until recently when all systems used 5V supplies), it would lead to an increase in power consumption per unit area of about 45% per year[1] [2]. However it seems that there is now a move to lower supply voltages as components operating at 3.3V are increasingly common. If supply voltage is scaled down linearly with the feature size there will be no increase in power consumption per unit area, although current consumption per unit area would increase by about 12% per year.

---

1. With constant voltage, power density increases by α^3 as feature sizes decreases by α. So if $100\% - \frac{1}{\alpha} = 12\%$, $\alpha^3 = 1.47$. See [6], table 4.12.
2. Assuming that operating frequency is always increased to the greatest possible value.

As integrated circuit processing techniques improve, defect density will decrease and larger devices will be feasible with economic yields. The trend in increasing device size is not as regular or as well understood as the trend in decreasing feature size; however the capacity of DRAM memory chips has typically increased by around 60% per year[7] which is about 24% more per year than is explained by the decrease in feature size. This suggests that area is increasing by about 24% per year.

If these trends apply to processors in the future, we can extrapolate the figures in table 1.2, based on the DEC 21064 Alpha in 1992 using 30W from a 3.3V supply as a starting point.

Year	Constant voltage	Voltage scales with feature size	
	Power consumption / W	Operating voltage / V	Power consumption / W[a]
1992	30	3.3	30
1994	95	2.6	45
1996	315	2.0	70
1998	1015	1.5	110
2000	3275	1.2	170

Table 1.2: *Projected power consumption (approximate)*

a. Note that because the supply voltage is decreasing, supply current will grow more quickly than power consumption. It will increase 15 fold from 9A in 1992 to 140A in 2000.

Although supply voltages are likely to decline to some extent, there are three reasons why they may not decline as fast as the feature size:

- In a system it is difficult to operate different components with different supply voltages, so the system supply voltage will drop only as quickly as the slowest individual components.
- Maintaining a higher operating voltage permits operation at a higher operating frequency, and hence to higher performance. Designers will find ways to avoid scaling down supply voltage in order to maintain performance.
- Transistor threshold voltages must scale down with supply voltage. It will become increasingly difficult to make transistors with small enough thresholds.

Whatever happens to supply voltage, in a few years processor chips will be consuming far more power than they are now. This will face designers with problems including the following:

- The capacity of power supplies has to be increased and distribution of power on the circuit board becomes more difficult.

- Power supply from the board to the dies becomes more difficult: the number of bond wires and pins used for power and ground has to be increased. The DEC 21064 Alpha chip uses a total of 138 power and ground pins to supply its 30W requirement. The current that can be passed through a gold bond wire before it melts and the closest pitch at which they can be bonded also imposes a limit on the amount of power that can be transferred to the chip [8].

- Power distribution on the chip becomes more difficult. The DEC ECL BIPS chip has to use a layer of 25µm thick gold power distribution rails over the top of the rest of the chip for power distribution [8].

- The very high currents on the chip can cause electromigration; that is, the force of the moving electrons hitting metal atoms leads to deformations and breaks in the metal [9].

- Removing the heat from the chip becomes increasingly difficult. There is an upper limit on the amount of heat that can be carried away by a passive heat sink of a few hundred watts[1]. At this point, active devices such as the Peltier-effect heat pump which cools the package to below ambient temperature [13] and the heat pipe or thermosiphon (as used by the DEC ECL BIPS chip [8]) become necessary. In practice, at high power dissipations the most difficult stage is moving the heat the first millimetre or so from the die to the heat sink. A good material for this is diamond, which is an excellent thermal conductor and also an electrical insulator. Cray computers have used diamond-filled epoxy compounds to attach dies to heatsinks. A more extreme solution is to etch channels on the back of the die and pump coolant through the channels. It has been estimated that by using this technique power dissipation up to 4kW per device can be dealt with [10].

- Removing waste heat from enclosures requires larger fans and design effort has to be spent on air-flow design. The noise level from fans has to be kept low enough to reduce noise pollution.

1. This calculation is based on the thermal resistance of an infinite hemisphere of copper, which is around 0.1 °C/W when connected to a heat source of size $1\text{cm} \times 1\text{cm}$.

Not only are these problems difficult to overcome, but the solutions can be expensive. For example, the thermosiphon used by DEC [8] would cost $125 in large quantities. The cost of gold power distribution rails and diamond adhesive prohibits their use in all but a few cost-insensitive applications.

To avoid problems like these, high-performance systems will need to apply similar power efficiency techniques to the ones used by battery operated equipment.

Environmental Issues

It has been estimated that 5% of the USA's non-domestic electricity consumption is used by computing equipment, and this is set to rise to 10% by the end of the decade [11]. Reasons for this increase include the increased power consumption of individual computers, the increasing number of computers, and the increasing tendency to leave equipment permanently switched on. In the US, the Environmental Protection Agency has proposed a standard for a low-power personal computer. They estimate that adoption of this standard could lead to a reduction in CO_2 emissions equivalent to 5 million cars.

Adoption of the low-power standard would also lead to significant financial savings on individual electricity bills. The electricity to operate a personal computer consuming 300W continuously for a year currently costs around £200 in the UK.

In the case of the personal computer, the majority of the power savings can be made in areas other than the processor; the most important areas are in the monitor and in increasing the efficiency of the power supply [12]. However the contribution made by the processor is not insignificant, and if neglected it will become more significant as the efficiency of other parts of the system increases. Processor manufacturers have started to realise this; for example Intel has recently announced that it will incorporate power-saving logic previously used only in processors for portable computers into all of its i486 processors[11].

Conclusions

It has been shown that high power efficiency is or will soon become an important factor in computer system and processor design in all types of equipment. In low-performance applications, the driving force is increased battery life for portable equipment. In medium performance applications, the cost of electricity and the environmental results of electricity use are most important. For the highest performance applications, the high thermal dissipation and power distribution may become limits.

The next section studies the approaches that exist for reducing power consumption.

1.2 How Can Power Consumption Be Reduced?

Power reduction techniques can be applied in many of the levels of the computer system, from underlying silicon techniques at the bottom level to compiler optimisations at the highest level.

Approaches Based on the Fundamental Technology

At the lowest level, power consumption is proportional to the capacitance of the wires and transistors on the chip and the square of the supply voltage [6]. Running the chip at a reduced voltage will reduce power consumption, but it will also reduce performance as maximum operating frequency is proportional to supply voltage.

The capacitance of the wires is proportional to the feature size, λ. Decreasing the feature size will give a proportional reduction in power consumption. The limit on feature size is imposed by the available technology for fabrication and is constantly decreasing. This decrease in feature size will certainly lead to further increases in power efficiency.

Circuit Design Approaches

In CMOS circuits, power consumption is proportional to the number of signal transitions[1]. Power consumption can therefore be reduced by reducing the number of signal transitions that occur. Consider for example a state machine whose state is represented by four bits, which spends 99% of its time alternating between two of these states. If these states were encoded as 0000 and 1111, there would be 4 times as much power dissipation in the state machine than if they were encoded as 0000 and 0001, because there would be four times as many signal transitions. It should be possible to incorporate this sort of optimisation into automatic logic synthesis programs, which could balance the trade-off between the most energy-efficient encoding and the encoding with the smallest or fastest implementation.

In another case, there can be a trade-off between power consumption and logic size. Sometimes pre-charge logic [6] is used, where nodes are charged to one state through a pre-charge transistor, and then possibly discharged to the other state through a transistor tree. This is in contrast to the simpler static logic approach where transistor trees are used in the pull-up and pull-down paths. The pre-charge approach saves transistors, but it means that on each cycle the output nodes make twice as many transitions[2].

1. In other technologies such as ECL, power consumption is static, i.e. independent of the number of transitions. Hereafter CMOS is implied unless otherwise stated.
2. Assuming an equal probability that the bit is a zero or a one. It can be worse; in a circuit such as a content addressable memory (CAM), the ‘hit lines’ may be the output of pre-charge logic. All but one of the bit lines is charged and discharged, making two transitions, every cycle.

The use of asynchronous logic rather than clocked synchronous logic may also lead to improved power efficiency. This effect results from various possible factors, the most important of which is the ability of asynchronous systems to perform well in situations with varying computational load.

Parallelism versus Speed

In some applications it is possible to replace a circuit with another that has greater parallelism but a lower operating frequency and lower power consumption [14]. If a circuit consists of n parallel sub-circuits, all operating at frequency f and supply voltage v, its throughput is proportional to nf and its power consumption is proportional to nv^2f. If the number of sub-circuits was doubled to $2n$ and the operating frequency was reduced to $\frac{1}{2}f$, the throughput would remain nf. However since maximum operating frequency is proportional to supply voltage [6], at the reduced operating frequency the supply voltage could be reduced to $\frac{1}{2}v$. The power consumption would then be $\frac{1}{4}nv^2f$; that is, doubling the silicon area reduces the power consumption four-fold.

Designs rarely if ever make use of parallelism with the objective of reducing power consumption. When parallelism is used in processors it is done to increase the performance that is possible at the supply voltage and operating frequency being used. Examples of this type of parallelism are multiple ALUs (superscalar processors) and multi-word wide buses between internal blocks [7].

Sleep Modes

At a higher level, systems can partially disable themselves when inactivity is detected. This feature is frequently found on portable computers. The simplest approach is to reduce the clock frequency. To obtain a further reduction in power consumption, once the clock frequency has been reduced the supply voltage can be reduced to give an even greater power saving. Reducing the supply voltage to a chip whilst it is enabled may not be a simple procedure, particularly for memory chips which may contain internal nodes that can store voltages across the change. The limit on the power efficiency improvement that can be obtained with a sleep mode depends on the workload pattern of the computer; if the computer is expected to be idle for a large proportion of the time, a sleep mode will be more beneficial than if it is almost permanently active.

Architectural Approaches

Certain aspects of a processor architecture can indirectly affect power consumption, in particular the energy used to fetch instructions is inversely proportional to the code density of the instruction set. Another example of an architectural feature influencing power consumption is the width of the processor's datapath; a wide datapath wastes energy when it is used to operate on narrow data.

Software and Compiler Techniques

If the algorithms used by the programs in use or the optimisations applied by the compiler to those programs can be improved so that they do the same work in fewer instructions, they will consume less energy as they are executed. These optimisations also lead to increased speed, and are constantly being improved.

1.3 Introduction to this Work

All of the areas mentioned above are important, and a successful power-efficient processor will combine the best techniques from each area to obtain the optimum power consumption.

This thesis concentrates on two of these areas, namely the use of asynchronous logic and the influence of the processor architecture.

Chapter 2 introduces asynchronous logic and argues why it may have better power efficiency than traditional synchronous logic in some applications.

Chapter 3 describes why a processor using asynchronous logic can exploit different architectural features from a conventional synchronous processor. The AMULET1 processor design, which is an asynchronous implementation of an architecture originally implemented synchronously, is considered and features that an architecture needs for efficient asynchronous implementation are proposed.

Chapter 4 looks at the power consumption in a processor system and finds that it is influenced by code density and other features. The issue of code density is investigated further and the densities of various processors are compared.

Chapter 5 describes various experimental studies to find how the code density of an architecture can be increased. The code density of the Sparc architecture is examined, and various ways in which its code density could be increased are proposed.

Chapter 6 draws together the architectural features that have been proposed in chapters 3 and 5, and proposes the basis of an architecture that is suitable for a power-efficient asynchronous implementation. The chapter concludes with a mention of possible future work.

There are three appendices, concerned principally with the experimental data used in Chapter 5. Appendix A studies the pattern of data accesses made by a program to its register bank and to main memory and gives a mathematical model for an aspect of this behaviour. This data is useful for evaluating the size of register banks, caches etc. Appendix B presents data relating to the frequency of occurrence of instructions in the Sparc architecture, the distribution of immediate values, and other data. Appendix C describes the benchmark programs used.

Chapter 2 : Asynchronous Logic

Currently most digital design including processor design is done using synchronous globally-clocked techniques. Synchronous design has been successful but some believe that asynchronous logic has the potential to perform better in some applications.

There are several different areas where asynchronous logic may have advantages. These include:

- Speed.
- Power efficiency.
- Formal verification and synthesis

This chapter starts by introducing some of these the proposed advantages of asynchronous logic over synchronous logic. Secondly, it describes some of the basic asynchronous implementation techniques.

The reason why asynchronous logic is of interest in this work is its potential for improved power efficiency. The third section covers the arguments for why asynchronous logic may have better power efficiency than synchronous logic.

2.1 Asynchronous Logic versus Synchronous Logic

Synchronous designs make use of a global clock signal, distributed to all parts of the system, to control timing. Communication between blocks is all relative to this clock. Blocks sample their inputs on an edge of the clock signal, and arrange for their outputs to be stable on some future edge of the same signal.

In contrast, in an asynchronous design there is no such global signal. All inter-block communication occurs at a local level. When a block wishes to communicate with its neighbour, it does so by means of some form of local request-acknowledge signalling.

The types of asynchronous circuits discussed here fall into the category of being self-timed. This means that circuits generate their own timing signals by means of matched paths and delays. This is one of the weakest forms of asynchronous logic. Other asynchronous designs make use of stronger methodologies, such as delay-insensitive logic which does not rely on matched delays. [19] [20]

What advantages does the asynchronous technique have over the synchronous? There are 4 possible arguments in favour of asynchronous design:

- The speed of an asynchronous design is dependent on the time taken in the typical case, whereas for a synchronous design the speed is always limited by the worst case.
- Asynchronous circuits do not suffer from clock skew and may not suffer from ground bounce.
- Asynchronous design is easier than synchronous design.
- Asynchronous circuits consume less power than synchronous circuits.

These points are discussed in the following sections.

2.1.1 The Typical-Case / Worst-Case Timing Argument

The clock frequency in a synchronous system has to be slow enough to allow all blocks to propagate changes from their inputs to their outputs in one clock cycle in all conditions. The worst case will occur for:

- The slowest block.
- Its slowest operation.
- Its worst-case input data.
- Its worst operating conditions (i.e. supply voltage, fabrication speed and temperature).

In the case of an asynchronous system, the delay from data being read into a block to data being available on the output is variable and need never be longer than is necessary. It may sometimes be as slow as the synchronous clock period, but for a typical block doing a typical operation on typical data in typical conditions the delay will be smaller. So an asynchronous system will *typically* operate faster than a synchronous system.

Designers of synchronous systems go to some lengths to minimise the worst case delay. They apply techniques such as the following:

- The delays in blocks are made as close to equal as possible. This means positioning the divisions between blocks so that the numbers of levels of logic in each block are as close to equal as possible. Synchronous designers try to find the critical path in their designs and speed it up. This means that even rarely-used pieces of logic have to be analysed in detail. In asynchronous systems, the governing principle is that “the typical case must be fast, and the unusual case must be correct”. For example, figure 2.1 shows a pipeline where two alternative blocks B and C can occupy the second stage. If this pipeline was for a synchronous processor, the clock would have to be as slow as the slowest of any of the four blocks. However, for an asynchronous processor, the unusual case where block B is used could be allowed to be slow, and only the typical case where block C is used needs to be optimised.

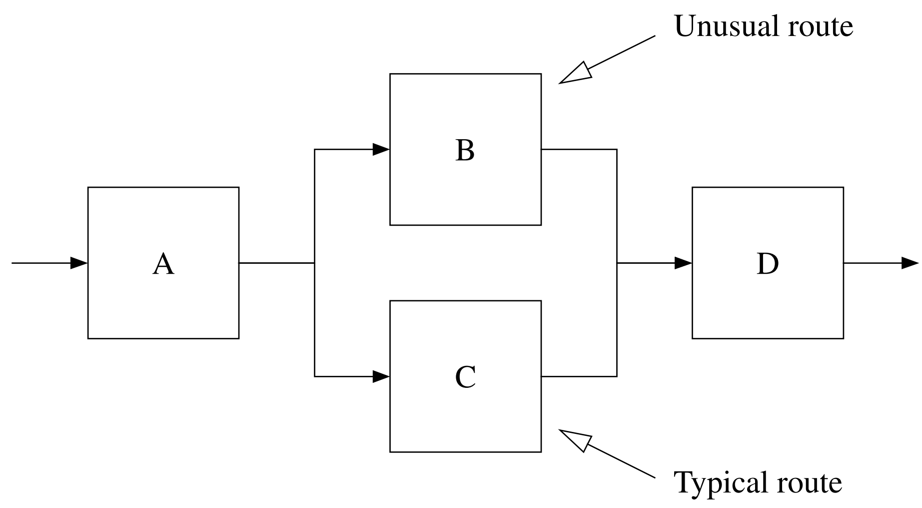


Figure 2.1: *Hypothetical pipeline arrangement*

- Techniques are applied to increase the speed of the logic for the worst case data input. The best example of this is for an adder unit. The simplest construction of an adder uses ripple carry, where the carry output of each adder cell is fed to the carry input of the next most significant cell. This circuit is fast for typical cases where the carry only has to propagate between a few cells, but in the rare cases where a carry ripples all the way from one end of the circuit to the other, the circuit is relatively slow. In a synchronous system, the clock speed would have to be adjusted to this worst case. To avoid this, synchronous designers devise much more complex adder circuits using techniques such as carry look-ahead, carry skip and carry select [7]. These circuits are much larger than the simple ripple-carry adder and work little if at all faster for typical operands. However they do work faster in the worst case, allowing the clock frequency to be increased. In an asynchronous system, this type of circuit is unnecessary. In [15], the asynchronous ALU used in the AMULET1 processor is described. The adder in this design has a worst-case delay that is approximately twice the typical-case delay. However, the worst-case input data occurs only about 5% of the time.

- It may seem that there is nothing that a designer can do to make his circuit work better in the worst case operating conditions. However, it is possible to redefine what the worst case operating conditions are. For a synchronous designer there is a trade-off between what he specifies as acceptable operating conditions and the maximum clock frequency that he allows. For example, the speed of a circuit decreases with increasing temperature. A designer may have a choice between specifying that his design is either "Maximum temperature 50˚C, Maximum frequency 30 MHz" or "Maximum temperature 85˚C, Maximum frequency 20 MHz". He has a similar choice in the case of operating voltage, and possibly also for process speed. On the other hand, the asynchronous designer can specify the widest possible margins for these parameters, on the understanding that achieved performance will be best in certain conditions.[1]

For many microprocessor applications, having a performance that is dependent on operating conditions is acceptable. An example is the personal computer, where users would be happy to accept improved typical performance with variation depending on operating conditions. On the other hand, there are application areas where worst case performance is the important measure. For example consider a microcontroller in an engine management system whose purpose is to collect data from various sensors and operate the spark plugs at the appropriate times. In this application, it is essential that even in the worst operating conditions the controller is able to process the input data before the time when the spark has to be generated. In applications such as this, the benefit of asynchronous logic is limited.

There may be fewer applications that fall into the same category as the engine controller than one might think. Although the specifications for most signal processing and real-time applications include maximum response times and minimum throughputs, this is just because the specification was drawn up with synchronous implementation in mind. Here are two examples:

Example 1: A Modem Application

Currently modems operate at fixed data rates, e.g. 9600 bits/second. The controller inside a modem has to do a significant amount of processing to decode the output bit stream from the input analogue samples. In a synchronous design, the controller would be clocked sufficiently fast so that it could do the required processing even for the worst case input data. In an asynchronous design, at first sight it seems that the "engine controller" situation applies, and that the asynchronous controller has to keep up with the data rate even in worst case conditions. However, this is just a result of the specified fixed data rate. There is no fundamental reason why modems need to use a fixed data

1. This makes writing data sheets for asynchronous processors a challenging proposition!

rate. If the modem standard was redefined, the speed could be agreed between the sending and the receiving modems (and continuously adjusted) so that both could keep up with it.

Consider for example a modem in a battery operated portable computer. With a synchronous controller, as the battery voltage drops the controller will stop working. With an asynchronous controller, as the battery voltage drops the controller will continue to function correctly but more slowly. As it does so the communication speed will drop. Whilst the battery is fully charged (i.e. in typical conditions) the asynchronous modem will operate more quickly than the synchronous design[1].

Example 2: A Compact Disk Player

Consider the data processing in a compact disk player. This is a similar problem to the modem application; the controller has to process the digital data from the disk and generate an analogue signal. The majority of the processing required is for the error detection and correction algorithms. However, unlike the modem, it is not possible for an asynchronous design to respond to reduced battery voltage or other adverse conditions by reducing the throughput, as this would slow the sound output. Instead as an alternative the controller can reduce the amount of error correction that it carries out, or it can discard some of the samples. To the user of a portable CD player, this means that as batteries run out, the synchronous design will stop functioning, whereas the asynchronous design will just lose some sound quality.

2.1.2 Clock Skew and Ground Bounce

Two communicating blocks in a synchronous design may see slightly different versions of the clock signal. This is because the different blocks may place unequal loads on it and the clock distribution tracks may be longer to reach some blocks than others. This is known as clock skew. The result is that some versions of the clock will be changing before other versions. When one block communicates with another block that has a slightly earlier version of the clock, it must set up its output signals in time for that early version of the clock. This means that the actual time available for computation within a block is less than the period of the clock, as shown in figure 2.2.

1. Provided that other factors such as line bandwidth do not limit the speed.

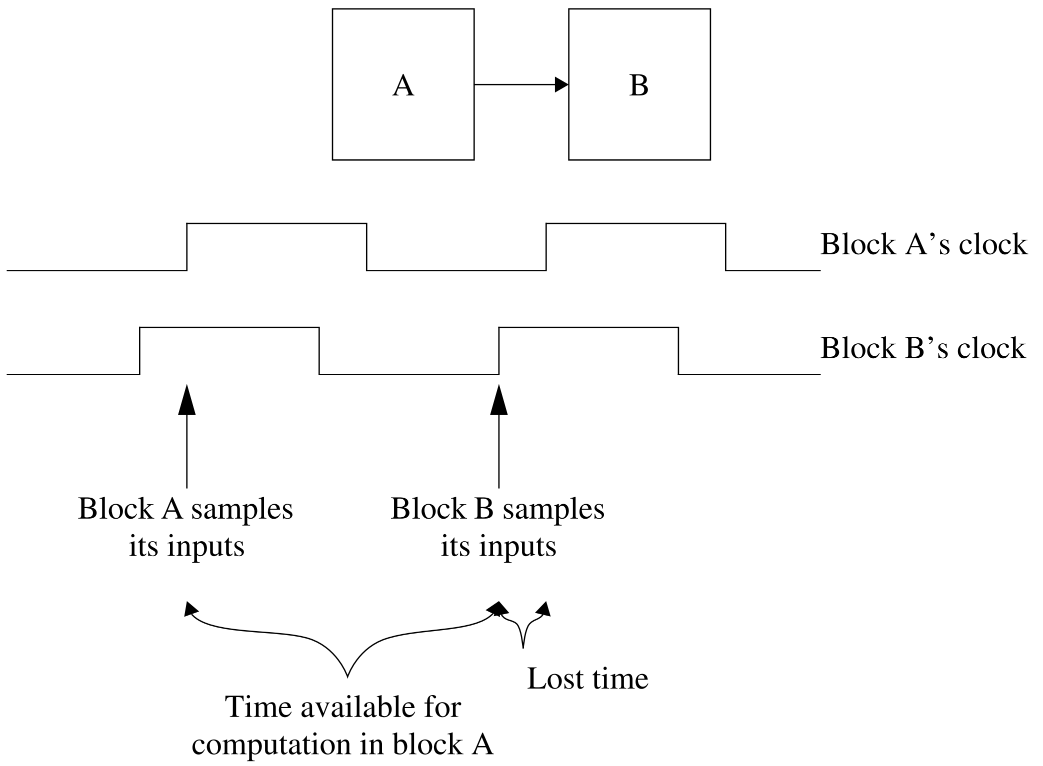


Figure 2.2: *Clock skew*

Synchronous designers do what they can to reduce clock skew. This includes careful analysis of the load on the clock signal and careful design of the clock drivers. In the case of the DEC Alpha 21064 chip [16], the designers published a three-dimensional figure showing the simulated clock skew across the chip. This processor's clock driver uses a tree of five levels of buffers.

In asynchronous design, the absence of a global clock completely eliminates this design problem, and also removes the need for a clock driver of the type used in the Alpha implementation.

In CMOS circuits, almost all power is consumed when switching occurs. In a synchronous design, this switching occurs when the clock switches. The power consumed in a synchronous design therefore occurs mostly around the clock edges. These peaks of power consumption lead to voltage drops due to the inductance of the power supply

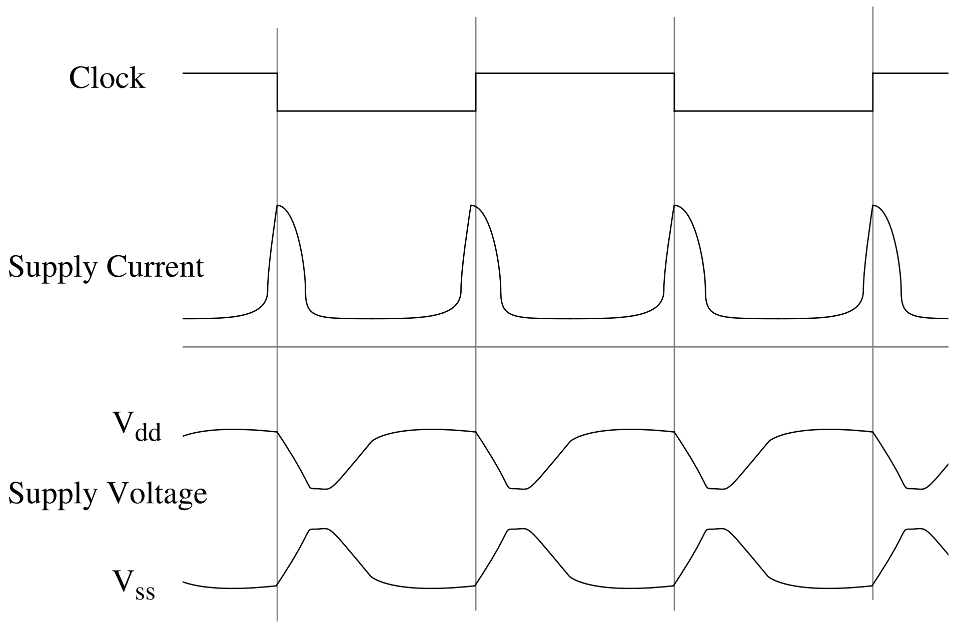


Figure 2.3: *Ground bounce*

path. This effect is known as ground bounce, and is illustrated in figure 2.3. This variation in supply voltage can lead to malfunction if it is too extreme as signals may no longer represent valid logic levels with respect to the power supply.

Synchronous designers try to avoid the problem by reducing the inductance of the power supply and in the case of the DEC Alpha 21064 chip [16] by constructing power supply decoupling capacitors on the chip.

In asynchronous circuits, because there is no global clock signal power consumption will be randomly distributed over time, so typically the spikes shown in figure 2.3 will not occur. However there is a probability that by chance a large number of nodes will change simultaneously, and a power surge will occur[1]. This is particularly possible if a group of connected elements such as adjacent stages in a pipeline operate at close to the same natural frequency.

If a significant surge did occur, the resulting voltage drop may have a less severe impact on an asynchronous circuit than it would on a synchronous circuit. Provided that the

1. The probability per unit time that n nodes in a circuit of m nodes, changing at random with average frequency f, would change within time δt is ${}^{m}C_{n}(f\delta t)^{n}$. Because of the factorial nature of this expression, the chance of all nodes in a circuit switching simultaneously is very low (e.g. of the order of the age of the universe), but the probability of perhaps 90% of the nodes changing may be relatively high and should be allowed for. 50% of the nodes may change simultaneously very frequently.

supply voltage did not change by more than the transistor's threshold voltage, the asynchronous circuit would simply slow down briefly but it would continue to behave correctly. In a synchronous circuit this slowing down would mean that the clock may occur before the next data was ready, and the circuit would fail. In both asynchronous and synchronous cases, if the supply voltage change is too great logic levels will be miss-interpreted, causing the circuit to fail.

2.1.3 Ease of Design

Several points have already been mentioned that make asynchronous design easier:

- The speed of infrequently used blocks does not affect overall speed, so in these cases costly sophisticated design techniques may be avoided.
- Simpler designs may be used for blocks with data-dependent delays (e.g. the ripple-carry adder).
- Problems with clock distribution, skew and ground bounce may not exist, so the designer does not need to spend time resolving them.

The other important reason is that asynchronous design can be more modular than synchronous design.

An asynchronous designer can take a "plug-and-play" approach. Provided each block in an asynchronous design is internally correct and meets the simple timing constraints of its external interface (see section 2.2), the design will be correct in terms of timing. The designer can therefore simply replace one block by another with different characteristics, and evaluate any change in performance. The synchronous designer does not have this flexibility.

Another way of looking at this is by analogy with high-level languages. The global clock in a synchronous design is like a global variable in a program. The self-contained local timing of an asynchronous design is more like the limited scope local variable of a program. Local variables are considered "better" than global variables in high-level language programs because they help to hide information in each function from the others.

The benefit of this modularity may be most significant at the level of systems made up of large components. For smaller designs, keeping track of the timing in a synchronous system is not too difficult and the asynchronous advantage may not be significant to the designer.

Another reason why asynchronous logic leads to easier design is the potential for synthesis from high-level descriptions. Various research groups have shown that it is possible to translate from a high-level description of a system written in a CSP-like language to an asynchronous circuit. The Tangram language [29] for example is translated into so-called handshake circuits, which are four-phase asynchronous circuits. It has also been suggested that the same properties that support automatic synthesis can be used to allow the automatic verification of designs.

2.2 Micropipelines and Other Styles of Asynchronous Logic

This section introduces some asynchronous design techniques. Firstly the design style used in the AMULET1 asynchronous processor is explained, and then various alternatives are considered.

In reality all timing styles are instances in a multi-dimensional space; synchronous design and the style used in AMULET1 are simply points in this space. There are very many other possibilities.

The technique used in the AMULET1 processor is known as *micropipelines* [17]. It uses two fundamental ideas:

- Two-phase event signals.
- Bundled data.

Two-Phase Event Signalling

Signals in a micropipeline design can be divided into two categories: data signals and event signals. Event signals control timing, and do so by signalling events. Events are occurrences at points in time; events have no “duration”.

In the micropipeline design style, events are represented by two-phase transition signalling. Transition signalling means that the events are represented by a transition on the wire. In two-phase transition signalling, both low-to-high and high-to-low transitions are meaningful and equivalent (figure 2.4).

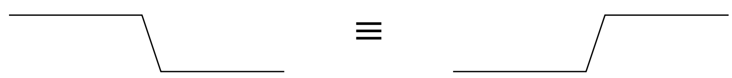
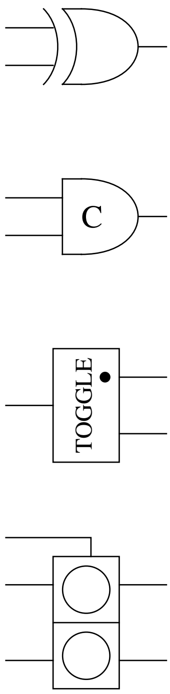


Figure 2.4: *Two-phase transition signalling*

Contrast the concept of an event signal with a level-sensitive signal. A level-sensitive signal communicates the state of the driver to the receiver at all times. On the other hand, an event signal does not communicate any information except when the driver has changed state. If the receiver of an event signal is “not looking” when the event arrives, it cannot tell that it has missed the event, as levels have no absolute meaning. On the other hand, the receiver of a level-sensitive signal can tell the state of the transmitter at all times.

Various logic gates are used to build circuits that process event signals [17]. Some of these are shown in figure 2.5.



Exclusive Or. Merges events on the two inputs to the single output.

Muller C Element. Wait for events on both inputs, then signal an event on the output.

Toggle. Steer events from the input alternately to the two outputs, starting with the one marked with a dot.

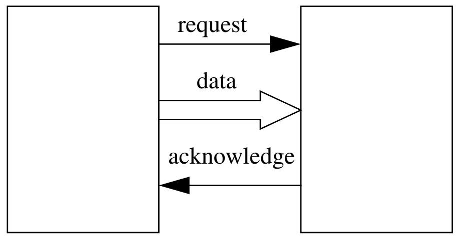
2x1 Decision-wait. Wait for an event on the x-axis input and an event on one of the y-axis inputs, then signal an event on the corresponding output.

Figure 2.5: *Example two-phase event logic elements*

Bundled Data

Data in a micropipeline is represented using a normal binary encoding in the same way that would be used by a synchronous design (this contrasts with the alternative of dual rail encoding, discussed later). This data is ‘bundled’ along with event signals that indicate its validity. Normally there is one event signal in each direction. The signal from the sender to the receiver is called request and the signal from the receiver to the sender is called acknowledge. Contrast the approaches used by the bundled data method and the synchronous method in figures 2.6 and 2.7.

1. Computation in sending block is complete and data is ready.
2. Data is driven onto the data signals.
3. Sending block signals an event on request.
4. When receiving block has used the data, it signals an event on acknowledge.
5. Sending block can remove the data.



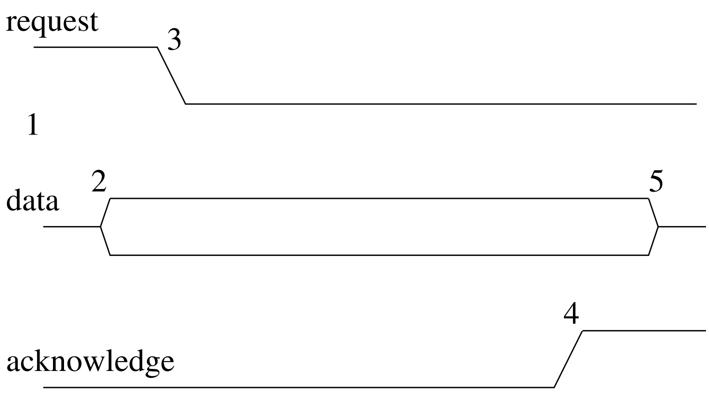
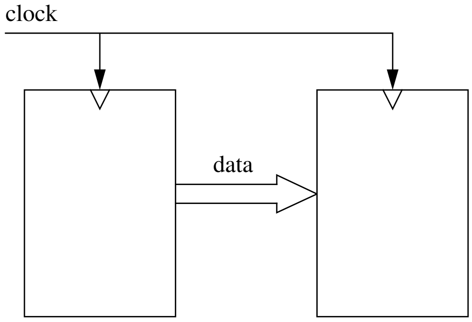


Figure 2.6: *Micropipeline inter-block communication*

A micropipelined circuit typically consists of a number of stages each of which communicates with its two neighbours using a request-acknowledge protocol. If the problem to be solved can be represented by a sequence of units with unidirectional communication, then a micropipelined implementation is simple.

When the problem requires less structured communication, the implementation may become more difficult. In a clocked system, the progress of data between all blocks is in lockstep. In an asynchronous system, this lockstep does not occur and blocks do not know the progress of data through other blocks, so when a rendezvous is required extra control is needed. As a general rule it is therefore a good idea to avoid structures that require non-local communication.

1. Calculation in sending block *must* have completed.
2. Sending block puts data on the data signal t_{su} (set-up time) before clock.
3. Receiving block *must* be ready for new data.
4. Clock edge occurs.
5. Sending block can remove data after t_h (hold time).



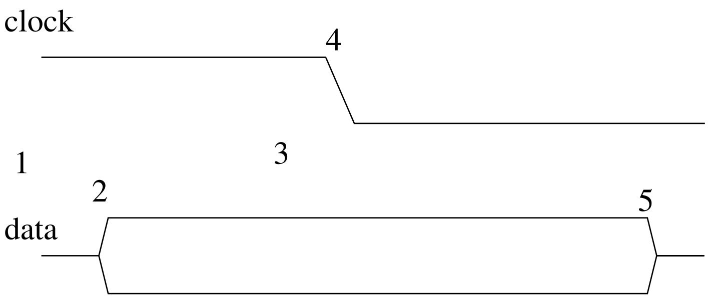


Figure 2.7: *Synchronous inter-block communication*

There are various alternative asynchronous schemes apart from two-phase event signals and bundled data. These include:

- Four-phase event signalling.
- Dual-rail encoding.

These two techniques are now considered.

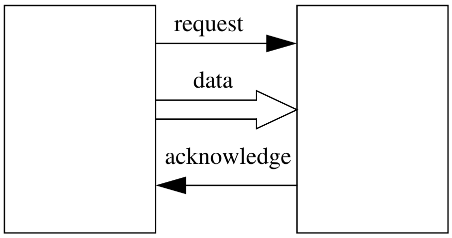
Four-Phase Signalling

Four-phase event signalling is an alternative to two-phase signalling. Unlike two-phase signalling, the meaning of rising and falling transitions are not the same. An exchange of data using a request-acknowledge signal pair involves not two transitions but four. An example of this is shown in figure 2.8[1]. The reason for this apparent additional complexity in the protocol is that the control circuitry needed can be less complex and/or faster for the four-phase protocol. The elements shown in figure 2.5 have to respond to both

1. Other arrangements where the data is transferred on other edges of the request and acknowledge signals are possible.

rising and falling transitions on their inputs. Corresponding elements for four-phase signalling only need to respond to one edge, and so may be simpler.

1. Computation in sending block is complete and data is ready.
2. Data is driven onto the data signals.
3. Sending block raises request.
4. Receiving block raises acknowledge.
5. Sending block lowers request
6. When receiving block has used the data, it lowers acknowledge.
7. Sending block can remove the data.



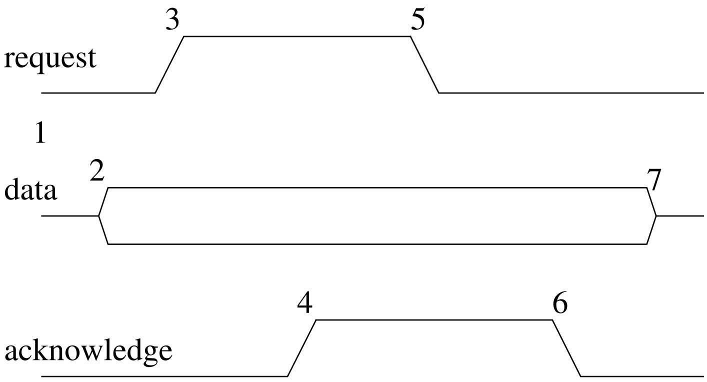


Figure 2.8: *Four-phase transition signalling inter-block communication*

Dual Rail Encoding

Dual rail encoding is an alternative representation for data signals. A dual-rail encoded signal uses two wires to represent each bit of the binary value. One of the wires is used to communicate a one and the other is used to communicate a zero.

When dual-rail encoded data is used, there is no need for an external timing signal such as the clock in a synchronous system or the request signal in a bundled data system. The timing data is contained within the data itself. Furthermore, in a multi-bit signal, the different bits can become valid at different times.

There are two-phase and four-phase versions of dual-rail encoding. In the four-phase version, the two wires can be decoded to give four possible meanings, as shown in table 2.1.

Zero	One	Meaning
0	0	Value is not yet computed
0	1	Value is One
1	0	Value is Zero
1	1	Illegal

Table 2.1: *Four-phase dual-rail encoded data representation*

Some sort of acknowledge mechanism is needed to allow the wires to return to the 0,0 state; this can be on a per-bit or per-signal basis.

In the two-phase version, a transition on the zero wire indicates that the value has become zero and a transition on the one wire indicates that the value has become one. An acknowledge signal is necessary to indicate to the sending block when it is free to send another value on the same wire.

There are two particular applications where dual-rail encoding has an advantage over bundled data:

- Certain control circuits, where the bundled data representation would be internally converted to dual-rail anyway.
- Datapath operations where it is advantageous to start processing before all bits have arrived.

These two cases are now considered.

Dual Rail Control Circuits

Consider the two-phase bundled data circuit shown in figure 2.9[1]. Here, a one-bit bundled data signal is used to produce an event on one of two output signals

1. The rectangular circuit element in figure 2.9 is a select block [17]. It steers an event on its left-hand input to one of the right-hand outputs, depending on the state of the boolean input at the top.

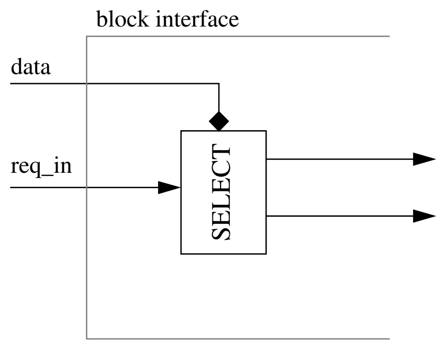


Figure 2.9: *Bundled data circuit eliminated by the use of dual-rail signals*

In this circuit, if the input was instead represented using dual rail encoding, this circuit could be entirely removed and the two dual rail wires could be connected in place of the outputs from the select block, reducing silicon area, power and delay. An example of this occurs in the AMULET1 processor where the ABORT signal enters the chip as a dual-rail encoded signal.

Dual Rail Datapath Functions

In many cases, datapath functions must wait for all the input bits to be available before acting. For example, a register write operation will always write all bits at the same time because the input data has to be synchronised with the register address. If dual rail encoded data was used in this case, a circuit would be needed to detect that all bits had arrived; that is, a dual-rail to bundled-data converter.

However there is one particular case where dual-rail encoded data demonstrates an advantage. This draws on ideas from the ALU of the superscalar SuperSparc processor [38] (which is synchronous).

Consider a circuit that adds three operands A, B and C together in two stages. Figure 2.10 shows the timing of the bundled-data implementation of this circuit.

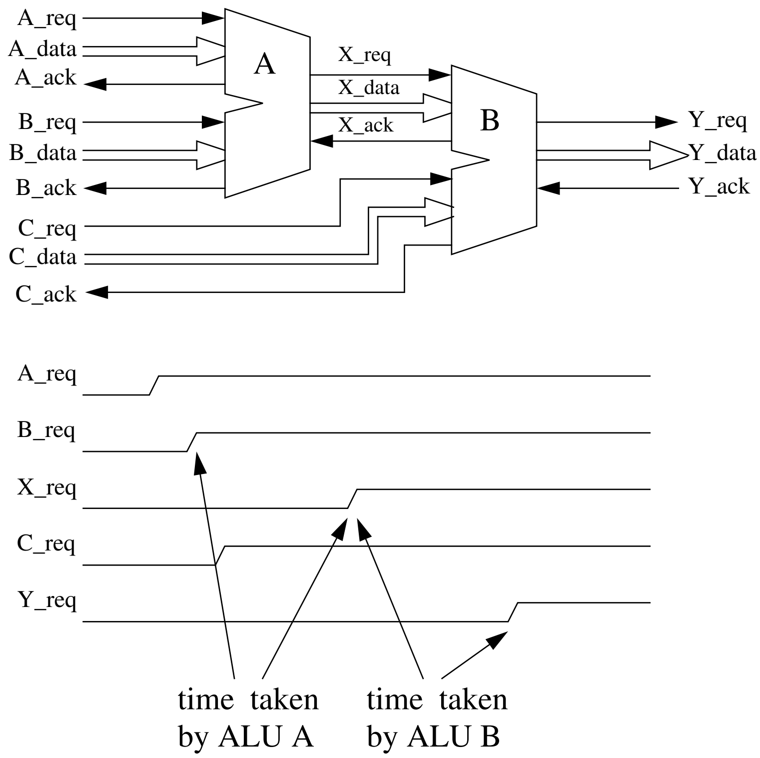


Figure 2.10: *Bundled data pipelined ALUs*

In the bundled-data implementation, the request output of the first ALU is connected to a request input of the second ALU, so the total time taken by the circuit is the sum of the total time taken by each of the two sub-circuits.

Using a dual-rail encoding for the intermediate signals between the two sub-circuits, a particular property of the adder circuit can be exploited; this is that the least significant bits are evaluated before the most significant bits. This is because of the nature of the carry signals that join the individual bit-wide adders[1]. Furthermore, the most significant input bits are not required until the carry signals have propagated to them. Figure 2.11 shows how this can be used. The time saving of this circuit over the one using bundled data for the intermediate value is equal to the time between the least significant bit of the output of the ALU becoming valid and the most significant bit becoming valid.

The number of cases where this technique can be applied may be fairly limited, but where it can be used it may lead to significant performance improvements.

1. This is particularly true in the case of a simple ripple-carry adder, but does also apply to other more sophisticated adders. See the description of adders for asynchronous designs on page 27.

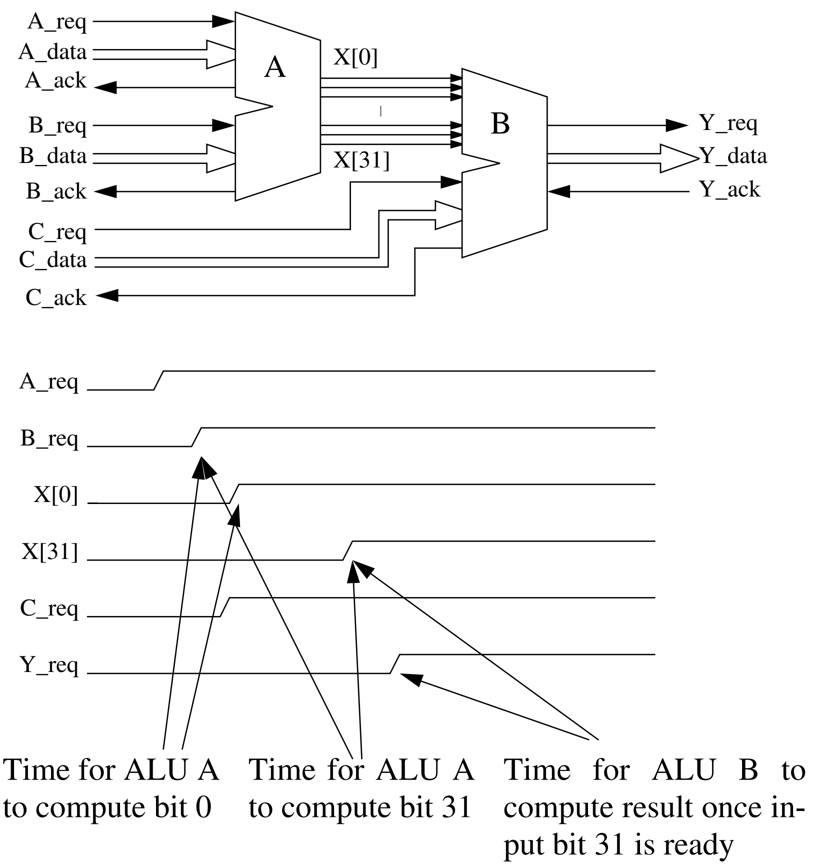


***Figure 2.11:** Pipelined ALUs using dual-rail encoding*

2.3 Asynchronous Logic and Power Efficiency

As has been previously stated, in CMOS energy is principally used when nodes switch from one voltage level to another. Power consumption is therefore proportional to the rate of node switching. In fact the equation governing power consumption is:

$$P = \frac{1}{2}CV^2fn$$

Where P is the power consumption, C is the average capacitance per node, V is the supply voltage, f is the switching rate in transitions per second and n is the number of nodes.

The use of asynchronous logic leads to lower power consumption because asynchronous circuits can switch fewer nodes per second with less capacitance than equivalent synchronous ones doing the same task.

The most important factor leading to this reduced power consumption is the fact that asynchronous systems can potentially have fewer “wasted” transitions than a synchronous system. This point is considered shortly. Firstly, consider the power used in the synchronous and asynchronous systems for each transaction.

Transitions Per Cycle

Consider the power used in the inter-block communication schemes outlined in figures 2.6, 2.7 and 2.8. Table 2.2 summarises the number of transitions that occur per cycle in each of the systems.

System	Transitions in control signals	Transitions in data signals
Synchronous	clock: 2	$\frac{n}{2}$ [a]
Two-phase bundled data	request: 1 acknowledge: 1	$\frac{n}{2}$ [a]
Four-phase bundled data	request: 2 acknowledge: 2	$\frac{n}{2}$ [a]
Two-phase dual-rail	-	n
Four-phase dual-rail	-	2n

***Table 2.2:** Comparison of design styles by number of transitions per cycle*

a. This assumes that on average half of the data signals change per cycle. For some situations fewer signals will change; for example if the data is the output of a counter on average 2 bits change. When a precharged logic system is used, the average value will be doubled.

From table 2.2, the following observations can be drawn:

- When the number of data bits n is very small, dual-rail encoding is advantageous.
- Two-phase transition signalling is more power-efficient than four-phase.
- In terms of number of transitions per cycle, no asynchronous design style does better than clocked design.

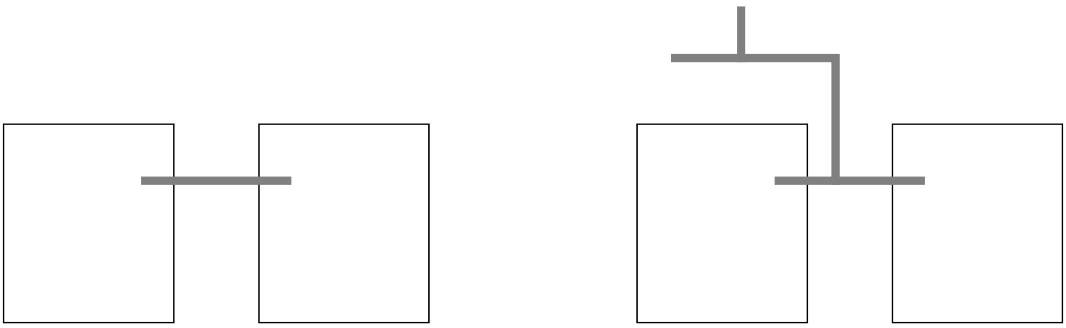
Despite the final point above, there are reasons to believe that two-phase bundled data designs can consume less power than synchronous designs:

- In a synchronous pipeline, two non-overlapping versions of the clock signal may be distributed[1], doubling the effective number of transitions.

1. Some recent published designs do make use of a true single-phase clock. [16] [18]

- Synchronous designs may even distribute the true and inverse of the clock signal or signals [6], doubling again the effective number of transitions per cycle[1].
- Asynchronous designs can have smaller loads on control signals

The basis for the final assertion above is that asynchronous control signals have a more local distribution than the global clock in a synchronous design. Although there are more control signals in an asynchronous design, their total capacitive load should be less than the total capacitive load on a clock signal, because they do not have the overhead of distribution from a central clock generation circuit



Local communication in an asynchronous system.

Global clock distribution in a synchronous system. The load on the clock will be greater than the total load on the asynchronous system's local signals.

Figure 2.12: *Load on a clock signal compared to load on asynchronous control signals*

The clock generation circuit typically makes use of a tree of drivers to produce the final clock signal. Power is also dissipated within this tree.

Asynchronous Designs Have Fewer Wasted Transitions

The most important argument for the reduced power consumption of asynchronous logic is that an asynchronous system can respond better to fluctuating levels of processing demand.

1. It should however be noted that some micropipelined units also distribute or generate locally the true and complement of the timing signals.

In all practical designs, there will be sections of the circuit that are not active at certain times. In asynchronous implementations, these circuits will see no events on their request inputs and will consume no power.

In synchronous systems, there are two approaches to this situation:

- The clock to the unused circuit can be disabled. When this is done, no power is consumed. However there are disadvantages resulting from the need to insert a gate between the global clock and the local gated clock to the block in use. This gate will introduce a delay, causing clock skew (see section 2.1.2). Adding the gate also adds to the design size and increases complexity as logic is required to control the input to the gate.

- The clock to the unused circuit can remain enabled, but a control input is used to indicate that no action should occur. In this case, the same power may be consumed as when the circuit is enabled.

In practice, the use of gated clocks is considered as something of a “black art” by many synchronous designers; for example, it is not mentioned in two of the standard textbooks [6] [19]. The natural power saving of asynchronous design is an improvement over the synchronous approach in this area.

The power saving resulting from the use of asynchronous logic can occur on any scale; from a gate-by-gate level when particular bits in a signal are not being used, to a block-by-block level when for example some of a processor’s functional units are not busy, to the highest level when the processor stops and waits for an interrupt before continuing. In an asynchronous system this power-down occurs quite automatically and instantaneously, in contrast to the power-down modes of processors used in portable computers which require software intervention to enter and are generally less flexible.

2.4 The Disadvantages of Asynchronous Logic

There are certainly some areas in which the synchronous design style has advantages over the asynchronous. At the low level, limited experience suggests that asynchronous circuits may be larger and slower than synchronous ones. A a higher level, the non-deterministic nature of asynchronous design introduces problems such as deadlocking which are not present in synchronous design. A solution to the problem of the test of asynchronous circuits has yet to be demonstrated.

Size and Speed of Control Circuits

Experience with the AMULET1 processor suggests that the use of the micropipelined design style leads to an increase in the area taken up by the control logic compared with a synchronous implementation [23]. Possible reasons for this include:

- The asynchronous design process is less mature than the synchronous process. Synchronous designers have the advantage of design tools, standard cells and experience that has not yet evolved for asynchronous implementations.

- The two-phase event signals used need extra logic to react to both rising and falling edges, whereas a synchronous system needs to operate on one edge only. The use of four-phase event signalling can help this, but it may lead to an increase in power consumption.

The AMULET1 control circuits also have lower speed compared to clocked circuits. This can be attributed to the increase in area and complexity, and also to a more fundamental aspect of the design style; asynchronous control circuits tend to be more sequential than synchronous circuits which can be more parallel. Circuit diagrams for event logic circuits have the appearance of flow charts, where wires communicate events representing the flow-of-control within the circuit. At any time only one element is active. Introducing parallelism into an asynchronous control circuit may involve arbitration, which is also slow.

The crucial design objective must be to reduce the delay through the control circuit to no more than the delay through the datapath. Datapath design for micropipelines is very similar to datapath design for synchronous logic as both use a straightforward binary representation of the data. In the AMULET1 processor the datapath was implemented using custom cells and hand layout and the control logic used standard cells and automatic layout, yet the speed of operation is limited in most cases by the delays through the control logic.

There are two ways in which the control delay and the datapath delay can be equalised: the delay through each stage of the control logic can be reduced, or the delay through each stage of the datapath can be increased.

Possible ways in which the control logic speed can be increased include:

- The use of four-phase event signalling.

- The use of more sophisticated synthesis procedures.

- The replacement of sequential actions by parallel actions.

- The reduction of the required complexity, by implementing a different architecture.

The idea of increasing the delay through the datapath logic is only sensible if it results in an increase in the functionality per stage. An example of an area where this technique has been applied is in the AMULET1's multiplier.

The AMULET1 multiplier is an asynchronous iterative multiplier. It consists of a control circuit and a number of multiplier stages, as shown in figure 2.13.

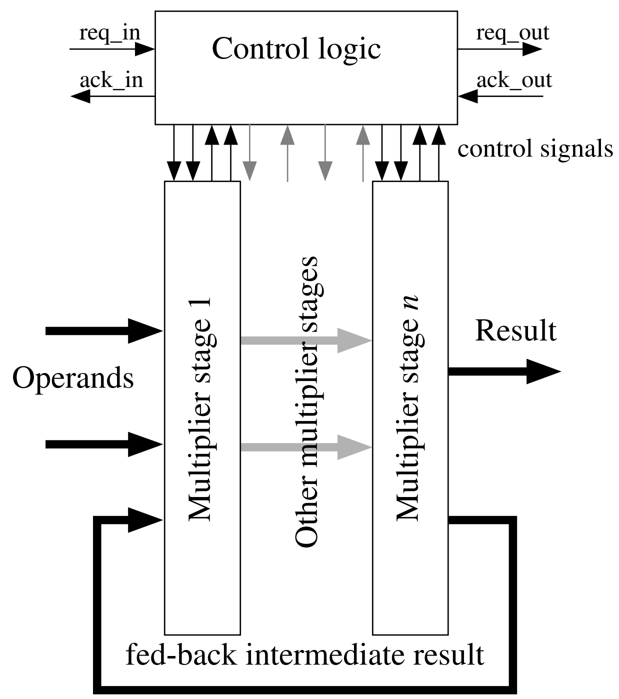


Figure 2.13: *AMULET1 multiplier construction*

The total number of stages that the data needs to pass through is fixed by the width of the multiplication. However there is a trade-off between implementing a small number of stages and feeding back the intermediate results a large number of times to produce the final result, and implementing more stages and feeding back the data fewer times. The trade-off was chosen by implementing enough stages to match the delay through the stages implemented and the control logic. The number chosen was 3.

Adding more stages would be a waste of silicon, as it would perform no faster than the 3 stage solution. Having fewer stages would reduce the silicon area, but the critical path would then be through the control logic.

It should be possible to implement the idea used here in other areas. Two applications are immediately apparent:

- The number of pipeline stages in the datapath can be reduced, but to maintain throughput extra parallel datapaths can be added (i.e. multiple functional units).
- The rate of data transfer in, for example, the instruction fetch mechanism can be reduced, but the width of the buses can be increased to yield the same throughput.

Interestingly, the argument that speed should be reduced and parallelism should be increased was proposed in section 1.2 to allow a reduction of operating voltage and hence an increase in power efficiency. The limit on the success of this approach is determined by the amount of parallelism that can be exploited successfully.

Deadlocks

As with systems of communicating processes, micropipelined units can enter a state of deadlock where all units are waiting for some other unit to act before continuing. In the design of the AMULET1 processor, several interesting and not immediately obvious potential deadlocks were discovered. It is hoped that some mathematical formal techniques may be possible to detect or avoid the possibility of deadlock at a high level in the design process, but at present no such technique is known.

Test

The action of an asynchronous system is inherently non-deterministic. The presence of arbiter circuits means that for a particular series of inputs there may be more than one acceptable series of outputs. The conventional approach to synchronous test where values are applied to inputs and then the outputs are compared with the expected outputs is clearly not applicable.

Furthermore, in an asynchronous circuit response time is variable. The problem of detecting a fault which leads to an increased delay in the chip is practically insoluble. However the problems of test are being addressed.

2.5 Conclusions

This chapter has given an outline of the asynchronous design style used in the AMULET1 processor, and has explained some of its strengths and weaknesses. Importantly, the use of asynchronous logic has some potential benefit for power consumption.

The following chapter looks at the question of processor architectures for asynchronous implementation. To exploit the power efficiency potential of asynchronous logic, it is first necessary to have an architecture that suits asynchronous implementation.

Chapter 3 : The Architecture of Asynchronous Processors

The previous chapter introduced asynchronous logic, and compared its properties with synchronous logic which has previously been used to implement processors. This chapter investigates the influence of the use of asynchronous logic on the architecture to be implemented.

Processor architecture has evolved to its current state through experience with synchronous implementations. Some features of current synchronous processor architectures can be attributed directly to properties of synchronous logic. For example, one of the arguments of the RISC approach is that complex instructions that slow down the cycle time should be removed from the architecture. As has been shown an asynchronous architecture can be tolerant of variable latency functional units, so this architectural principle need not be applied.

This chapter considers the organisation of a processor in terms of the properties of asynchronous logic, in order to identify those features that are more or less suitable for an asynchronous implementation. The basis for this analysis is the AMULET1 processor, which is an asynchronous implementation of the ARM architecture [15] [21] [22] [23]. This chapter looks at various architectural ideas and concludes with a summary of features that are most suitable for asynchronous implementation.

The ARM architecture has previously been implemented in a family of synchronous processors. Its principle features are summarised in figure 3.1.

The AMULET1 processor was implemented using the “Micropipelines” asynchronous design style (see section 2.2). The objective was to show that possibilities exist for improved power efficiency and performance resulting from asynchronous implementation.

The ARM architecture was chosen for the AMULET1 implementation to show that asynchronous techniques can be applied to a complex and successful existing design. Other asynchronous processor implementations have chosen simpler architectures omitting important features such as pipelining and exceptions [24] [25].

- Fixed length 32-bit instructions.
- 15 general purpose registers.
- Full set of standard 32-bit integer arithmetic and logical instructions including multiply.
- Arithmetic and logical instructions include an optional shift of one of the operands, allowing up to 3 source operands per instruction.
- Only load and store instructions access memory.
- 16^{th} register stores the program counter, and can be used as source or destination by nearly all instructions.
- Load and store have optional address pre- and post-incrementing and decrementing.
- Load and store multiple instructions transfer any subset of the 16 registers to or from memory.
- All instructions may be conditional on the condition codes.
- Precise exceptions including data abort on load and store.

Figure 3.1: *ARM architecture features*

Because the ARM instruction set was originally designed with synchronous implementation in mind, it can be expected that some features would not suit an asynchronous implementation as well as a synchronous implementation.

3.1 Data Dependencies

The basic organisation of a simple asynchronous processor's datapath pipeline is shown in figure 3.2. This shows a pipeline which has the following properties:

- Instructions are issued and complete strictly in order.
- All input state (operand registers) is read at the same time.
- All result state (destination registers) is written at the same time.

This simple pipeline is relatively straightforward and efficient to implement as a micropipeline.

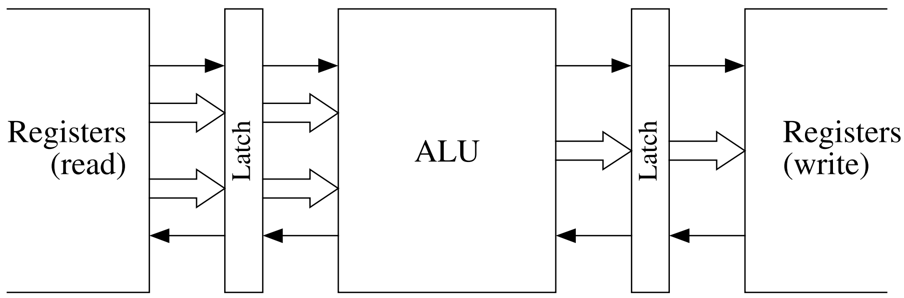


Figure 3.2: *A simple processor datapath pipeline*

There are however many features of the ARM instruction set which require a more complex structure for the AMULET1 processor. Similar features also exist in most or all synchronous architectures. The first of these features, data dependencies, is now considered.

Consider the following instruction sequence:

```
R3 := R2 + R1
R5 := R3 + R4
```

In the normal interpretation of this instruction sequence, the value read from R3 by the second instruction is the value written by the first instruction. However in a pipelined processor (synchronous or asynchronous), because of the overlap of instructions the value is not written to the register by the first instruction until after the second instruction has read it. This would result in the second instruction getting an old value.

Various techniques have been used to overcome or avoid this problem:

- Do not have pipelining in the execute unit. This is adopted by current implementations of the synchronous ARM processor. Such an approach may lead to low performance however if the pipeline throughput is limited by this stage. This is the ideal solution if it provides the necessary performance as no extra control logic is required, but most synchronous processors find that it does not provide enough performance. Section 2.4 suggests that for a micropipelined implementation it may be appropriate to have fewer levels of pipelining and greater parallelism, so this may be a good choice.

- Define the semantics of the instructions so that the result of an instruction is not available to the n following instructions, where n is the number of pipeline stages between register reading a writing. It is necessary to have the compiler interleave instructions as much as possible and insert NOP instructions where necessary. Unfortunately it is often not possible to order the instructions avoiding NOPs because very many instructions wish to use the result of the previous instruction[1], so this approach leads to inefficiency and low code density. It was used by the original MIPS architecture in the case of load instructions only, but this was dropped in later versions [7]. In the case of an asynchronous processor the number of following instructions n is a variable depending on how full or empty the pipeline happens to be, so implementing this scheme is virtually impossible.

- Limit the issue of instructions so that following instructions wait for preceding ones, on which they are dependent, to complete. This requires extra logic in the instruction issue stage that compares the register numbers of source operands with the register numbers of the destination operands of outstanding instructions. This is the approach used in the AMULET1 processor; the register lock fifo [22] keeps track of which registers will be written to by outstanding instructions, and following instructions are not allowed to read their source registers until outstanding writes to those registers have occured. This scheme has the disadvantage that when adjacent instructions are dependent on each other, the pipeline will contain only one instruction at a time, resulting in reduced throughput. Compiler optimisations can be applied to interleave instructions as much as possible, but there is a limit on how effective this can be. This is recognised as a problem with the AMULET1 processor.

- Modify the pipeline to insert forwarding paths. These are buses that connect the output of the ALU to the inputs of earlier stages in the pipeline, so that results can be forwarded to following instructions that want them. This technique is used by many synchronous processors. In a synchronous pipeline all pipeline stages produce their outputs simultaneously, and only multiplexers are required at the inputs to stages (figure 3.3). In the case of an asynchronous pipeline, there is a problem because the outputs are produced with no fixed timing relationship to each other. It would be very difficult for the control logic to tell what data should be forwarded from what stage to what other stage, and synchronisation would be required at the input to each stage.

In all cases, it is harder for an asynchronous processor to deal with data dependencies than for a synchronous one. To support efficient asynchronous implementation, the instruction set architecture should allow the compiler to schedule instructions with the minimum of data dependencies. To do this, the compiler must interleave multiple threads of execution.

Figure 3.4 shows how a program can be re-arranged to remove some dependencies. However code that is reorganised in this way will tend to need more registers for tempo-

1. See the measurements of last-result use in section B.2.

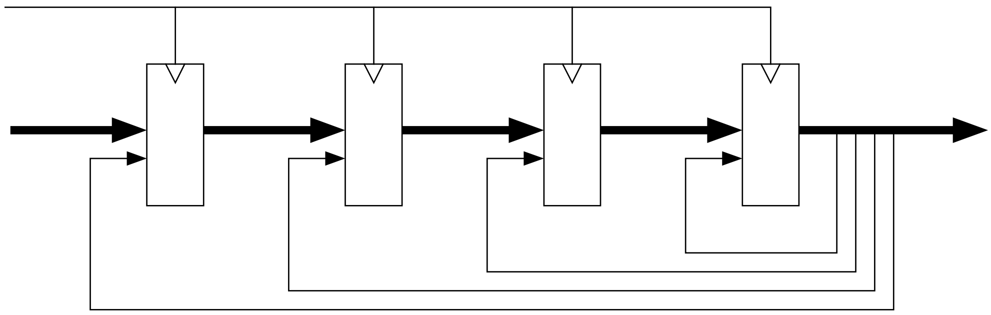


Figure 3.3: *Forwarding paths in a synchronous pipeline*

```
a=b+c+d+e+f+g
```

```
a=b+c
a=a+d
a=a+e
a=a+f
a=a+g
```

No temporary registers. All instructions are dependent on the previous instruction.

```
a=b+c
x=d+e
y=f+g
z=a+x
a=y+z
```

Three temporary registers. Only one instruction is dependent on the previous instruction; average dependency length is 2 instructions.

Figure 3.4: *Trade-off between number of registers and data dependencies*

rary local variables. This suggests that to minimise data dependencies, an instruction set should have a generous number of general purpose registers. The ARM architecture has 15 general purpose registers; most other RISC processors have 31 or 32.

Another factor limiting the amount of interleaving that the compiler is able to do is the use of shared processor resources such as condition codes (figure 3.5). Architectures such as the Alpha and the MIPS, that can test the value in any register to evaluate a condition, or the Sparc V9 architecture, which has multiple condition code registers, are advantageous here.

```
if (a>b) a++;
if (c>d) c++;
```

```
CMP a,b
if_greater INC a
CMP c,d
if_greater INC c
```

One condition code register. The instructions cannot be interleaved.

```
CMP a,b -> CC1
CMP c,d -> CC2
if_greater(CC1) INC a
if_greater(CC2) INC c
```

Multiple condition code registers. Interleaving results in longer data dependencies.

Figure 3.5: *Impact of a contended resource (condition codes) on data dependencies*

Perhaps the most important shared resource that limits the number of threads that the compiler can generate is the program counter; instructions can be interleaved only in sequential code between branch instructions, because the non-branching threads must continue to execute whether the branch is taken or not. To overcome this problem requires a radical shift in the processing model from a sequential instruction stream to a set of parallel instruction streams. This idea is not explored further.

3.2 Order of Completion and Precise / Imprecise Exceptions

In the simple pipeline model, the instructions all follow the same path, take approximately the same time to execute and complete in the same order that they were issued. In a real pipeline this is unrealistic because different instructions will take different amounts of time to complete.

Examples of slower instructions are the complex arithmetic instructions such as multiply. In an asynchronous processor it is reasonable to have such instructions because the pipeline speed is not limited by these slow instructions in the typical case - see section 2.1.1.

The most important slow instruction is the load instruction. Load takes longer than the simple arithmetic operations because calculating the address will take as long as an ordinary addition, and this is followed by an access to the data memory. There are two ways in which a pipeline can deal with this increased latency:

- In-order completion. Non-load instructions wait behind any load instructions and instructions complete in the same order that they were issued. A pipeline arrangement for in-order completion is shown in figure 3.6.

- Out-of-order completion. Faster non-load instructions may overtake any load instructions that are accessing the memory and complete before them. This is shown in figure 3.7.

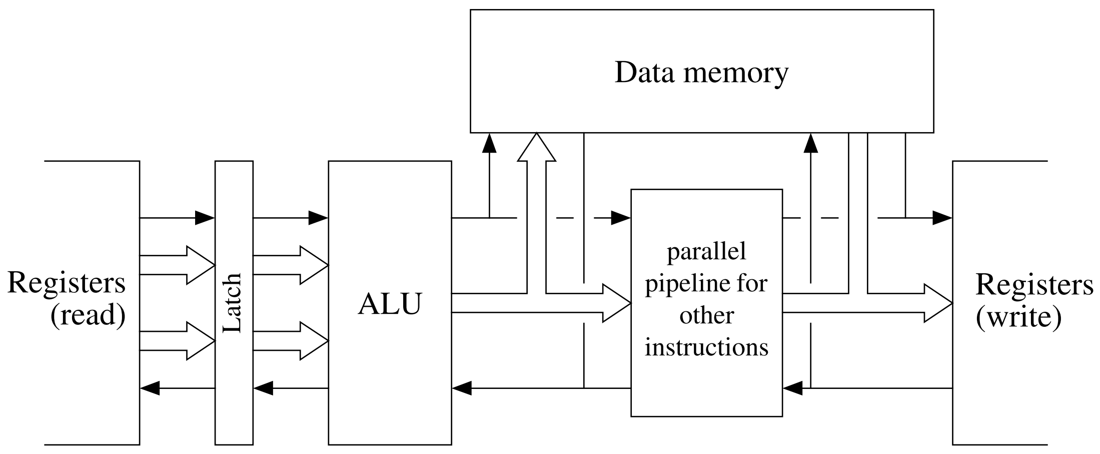


***Figure 3.6:** Pipeline arrangement for in-order completion*

For out-of-order completion in an asynchronous processor, an arbiter is used to merge the load results and the non-load results before register write occurs. In a synchronous processor, one scheme is to add an extra write port to the register bank.

Out-of-order completion is an attractive idea because it gives non-load instructions a lower latency through the pipeline. This means that subsequent instructions that are dependent on these instructions can be issued sooner.

However there is a disadvantage to out-of-order completion. Sometimes it is necessary to cancel an instruction due to some sort of exception; for example a load operation may cause a protection violation. In the case of in-order completion, when this occurs it is a relatively simple matter to cause the faulting instruction and any following instructions that have been issued to be cancelled. When out-of-order completion is used, the instruction following the load may already have completed by the time the load fault is signalled, and it is too late to cancel it. This is a difficult problem on a synchronous processor, but it is even more complex on an asynchronous processor as the relative timing

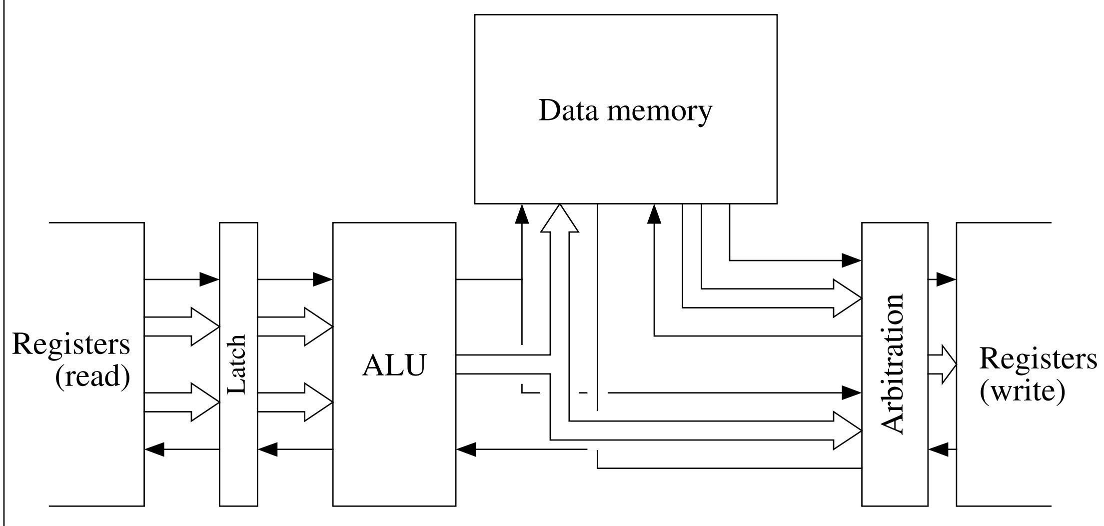


Figure 3.7: *Pipeline arrangement for out-of-order completion*

of the fault signal and the progress of the following instructions may be non-deterministic.

The ARM architecture specifies that exceptions must be precise; that is when an exception occurs the processor state must not advance beyond the instruction that causes the exception. This makes out-of-order completion difficult to implement.

The alternative to precise exceptions is to have imprecise exceptions, where processor state is allowed to advance to some extent after an exception. The DEC Alpha architecture [26] uses an imprecise exception model to deal with arithmetic exceptions, because these events are typically either ignored or signal an error. Because it is not generally necessary to restart after an arithmetic exception they can be made imprecise to allow out-of-order completion of these instructions. When the effect of precise exceptions is required, the Alpha architecture provides 'barrier instructions' which wait for any potentially-faulting outstanding instructions to complete.

In the case of load instructions, it may or may not be necessary for the process to be restartable after an exception. In the case of the Alpha architecture, an exception may indicate that the required memory access cannot be dealt with until a page has been brought in from a swap disk. This implies that code must be restartable so the Alpha implements precise load exceptions.

In a system which uses hardware to deal with MMU translation buffer misses[1] and does not store unused memory pages on disk, the only reason for an exception during a load operation is a memory error or a protection violation. Such a situation may exist on a portable system with no disk storage. In this case, a load exception always indicates an error and it may be acceptable to the operating system for the user-visible part of the processor state to become undefined when it occurs. This allows the use of imprecise load exceptions.

A trade-off between in-order completion with precise exceptions and out-of-order completion with imprecise exceptions is to have in-order execution up to a point where instructions commit to completing without fault. Beyond this point, completion can be out-of-order. This is the scheme used in the AMULET1 processor. The instructions after a load are not allowed to pass beyond the ALU to the register write stage until the preceding load has committed to completing without fault.

For this scheme to work efficiently, the load instruction must typically commit to completing correctly quickly. This suggests the use of a translation cache (TLB) closely coupled to the ALU.

3.3 State Changing Actions

In the simple pipeline model, each instruction reads all its input state as it is issued and changes all its result state as it completes. This arrangement fits well with the simple pipeline described. However in the ARM instruction set there are various instructions that make multiple changes to the result state.

All single load and store instructions can optionally write a modified value back to their base register. This mechanism is used to implement stacks and other structures. Most other RISC architectures do not have this facility.

On the synchronous ARM, this base-register write-back occurs ‘for free’ during a cycle when the register write port would otherwise be idle. In an asynchronous implementation it is not as easy. The AMULET1 processor implements this by changing the single ‘bubble’ instruction into multiple ‘bubbles’, one for each of the writes, in mid-pipeline. This causes more complexity in the pipeline control logic.

The other group of instructions that are converted into multiple ‘bubbles’ and change multiple registers is the load and store multiple instructions. The control logic becomes more complicated when combined with the problems of dealing with precise exceptions, described previously.

---

1. This is known as table walking. The alternative is to cause an exception when the translation buffer misses and to use software to perform the translation.

As an illustration of the problem with these complex instructions, in the AMULET1 processor, a load multiple of less than about 4 registers is slower than an equivalent sequence of single-register loads.

The result of this complication is an increase in the amount of control logic required, and an increase in the delay through this control logic, leading to lower performance and lower power efficiency for all instructions. This suggests that the philosophy of removing unnecessary instructions is at least as appropriate for an asynchronous processor as it is for a synchronous one.

3.4 Condition Codes and Conditional Instructions

Like other architectures the ARM processor has a condition code register. The flags in this register are optionally set by all arithmetic and logical instructions.

Unlike other architectures, all ARM instructions can be executed conditionally on the basis of the condition codes; that is, all instructions contain a four-bit field indicating a condition such as 'less-than', 'not-equal', 'always' which must be true for the instruction to execute. In most architectures, only branch instructions can be conditional in this way.

The synchronous implementations of the ARM have a non-pipelined execute stage, so at the end of one instruction's decode stage the value of the condition codes resulting from the previous instruction is known. This allows an instruction to set the condition codes and for the following instruction to test them without any delay.

In the AMULET1 processor, the execute stage is an asynchronous pipeline. The condition codes are stored in the ALU, where they are generated. There are then two choices of how to implement conditional instructions:

- Instructions whose condition code is other than 'always' are delayed at the issue stage until any preceding instruction that may set the condition codes has done so. If the condition code fails, the instruction is discarded. This is simple to implement but reduces performance, as it is common for instructions to test the condition codes immediately after they have been set. This is the data dependency problem; see figure 3.5.

- All instructions are issued immediately and the condition codes are tested before the instruction completes. If the condition codes fail, the result is not written to the registers. This is more complex to implement because the control for this function has to interact with the control for instructions that make multiple changes to the result state, as described above. It also has a performance impact as instructions whose condition codes have failed stay in the pipeline, slowing down following instructions, until they are removed at completion; for example a non-executed multiply instruction will be executed, taking non-trivial time, before its result is thrown away.

The AMULET1 processor implements a scheme that is a combination of the above. Most instructions have their condition codes tested as they leave the ALU, but the load and store multiple instructions are treated specially. This can only add to the complexity of the control logic.

It is possible that much of the benefit of the conditional instructions could be obtained from only conditional branches and conditional moves - the only instructions that are made conditional in any other architecture. This limited conditionality is simpler to implement than making all instructions conditional. The implementation of conditional branches is considered in the next section. Conditional moves can be implemented very simply in the ALU as it is not necessary to cancel the instruction if the condition fails; the old value is simply written back.

3.5 Branches and Flow-of-Control

In the ARM architecture, register 15 is the program counter. It can be read by any instruction just like any other register, and can also be written to which causes a branch to occur. The most useful application of this function is to perform procedure entry and exit using the load and store multiple instructions.

This free availability of the program counter is relatively simple to implement in the synchronous ARM. This is because the main datapath ALU is used to perform branch calculations, and so the program counter has to be available at the input to the ALU.

Because of the fixed relationship between the pipeline stages in the synchronous processor, whenever a program reads from R15 it gets the address of the instruction currently being read from memory; as there are three pipeline stages it will be PC+8. In the asynchronous processor, there is no such fixed relationship, and the AMULET1 has to use a 'PC pipeline' to feed PC+8 values to the register bank where they are substituted if R15 is read.

Figure 3.8 shows the synchronous ARM processor's organisation. The program counter is closely coupled to the register bank. This is useful for a synchronous processor that has a single memory port. Some implementations of the ARM processor have no cache and so a single memory port is essential. However most processors now implement separate caches for data and instructions, and give the processor core two memory ports. This helps to remove the potential bottle-neck caused at the processor-memory interface[1], and allows load and store instructions potentially to execute in a single cycle.

1. In a synchronous processor, the cycle time is likely to be limited by the speed of the cache [28].

The use of separate memory ports for data and instructions leads to the idea of a remote program counter (figure 3.9). This is a unit that generates sequential memory addresses and feeds them to the instruction memory. The datapath is still responsible for executing branch instructions and the main ALU is used for this purpose.

As was observed in the last chapter, micropipelines are most suited to systems where communication is by means of regular unidirectional paths without global communication. This makes the idea of a remote program counter attractive as it has more regular communication than the ARM organisation. This model is somewhat at odds with the idea of making the program counter available as a general register.

A continuation of the remote program counter idea is a remote branch unit (figure 3.10). In this scheme, branch instructions are interpreted locally to the program counter, and are not passed down the pipeline to the execution unit. This is attractive in an asynchronous implementation because the elastic pipelines mean that the datapath can make use of the extra bandwidth obtained from not having to perform branch calculations. Furthermore, the reduced amount of communication between the datapath and the instruction fetch (which are operating with no fixed timing relationship) means that the control overhead is reduced.

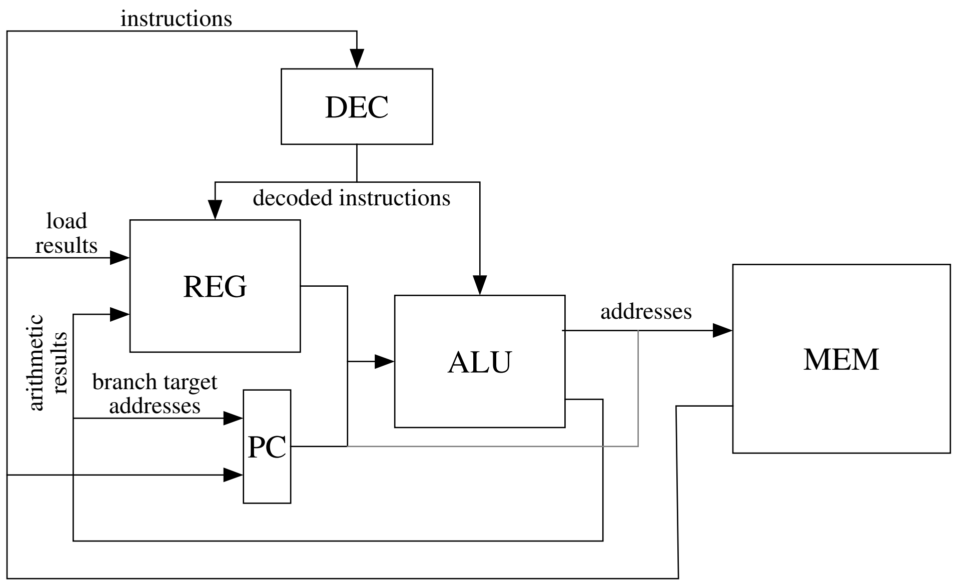


Figure 3.8: *Processor arrangement for ARM-style branches*

One feature that does require communication between a remote branch unit and the datapath is the resolution of conditional branches (this is not shown in figure 3.10). There are various models for branch conditions which work more or less well with a remote branch unit:

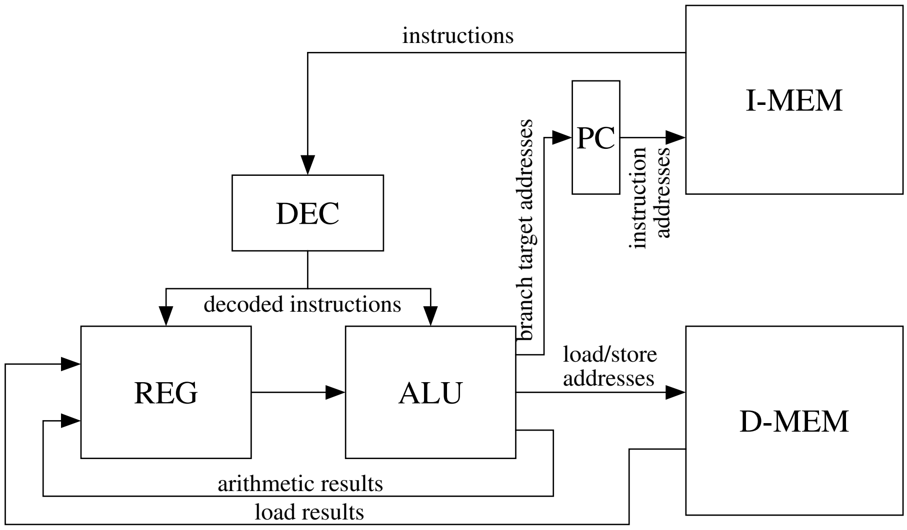


Figure 3.9: *Processor arrangement for remote program counter*

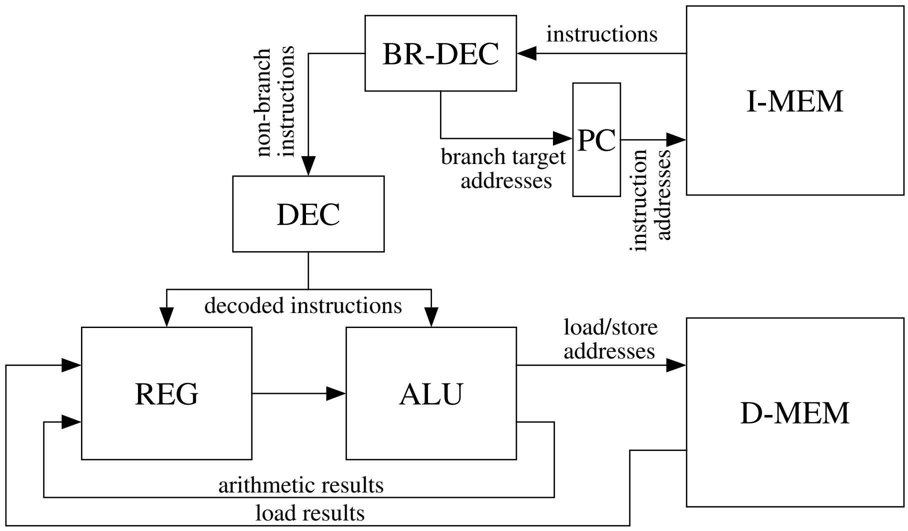


Figure 3.10: *Processor arrangement for remote branch unit*

- Condition codes. To use condition codes, the branch unit has to know when the last instruction that was going to change the condition codes has completed, and it then has to be able to test the value of the condition codes. This makes it important that instructions that can set the condition codes are easily identifiable. In the ARM architecture, all arithmetic and logical operations can potentially set the condition codes, but in practice in the majority of cases only the compare instructions are so used. Because the branch unit would have to track the progress of instructions that may set the condition codes, their use is unattractive.

- Specify the comparison or test in the branch instruction. This is the approach used in the MIPS and Alpha architectures [7] [26]. In a remote branch unit scheme, the comparison to be computed would be passed to the execute unit like any other instruction, but with the destination set as ‘branch unit’. The branch unit would then wait for the result of the comparison.

- An alternative idea which suits asynchronous implementation is to connect the output of the ALU to the branch unit by means of a micropipeline. Compare instructions complete by writing a boolean value into this pipeline. A subsequent branch instruction reads the first value from the pipeline to determine whether it is taken or not.

The last of these ideas appears the most attractive, because it allows the branch condition to be sent to the branch unit as soon as it is computed. The first two schemes would lead to greater latency. The last idea does have its disadvantages; in particular it is possible that a branch instruction may be executed when no previous compare instruction has been fetched. In this case the processor could enter a state of deadlock. It is however possible to detect this condition.

Branch instructions are very frequent and their performance is important; however the time taken to resolve the direction of a conditional branch can be large. Processors try to overcome this latency by employing speculative execution. Instructions are speculatively executed in the hope that they are the right ones; when the branch is resolved they may be cancelled. In a synchronous pipeline where the relationship between pipeline stages is fixed, it is relatively simple for a controller to know which pipeline stages contain instructions that need to be cancelled. In the case of an asynchronous pipeline where there is no fixed relationship, this is more complicated. It compares with the problem of cancelling instructions in a precise exception system after an exception. It is attractive therefore to consider not implementing speculative execution in an asynchronous system.

Speculative execution is also unfavourable from the point of view of power consumption, as doing work that is not ultimately productive is wasteful.

If speculative execution is not implemented, it is even more important that branch instructions execute as quickly as possible[1]. This suggests that a simple set of branch

1. Note that branch prediction outside the branch unit is still possible - it is only speculative execution within the execution unit that is discouraged.

instructions is preferable to the ARM instruction set where any arithmetic or logical or load instruction can write its result to the program counter.

3.6 Conclusions

This chapter has looked at the asynchronous implementation of architectural features including the following:

- Data dependencies, forwarding and its alternatives.
- Exception models and order of completion.
- Branch mechanisms and condition codes.

Features that have been identified as preferable for asynchronous implementation include the following:

- It is likely that data dependencies between instructions will slow the processor down. The compiler can reduce the impact of this effect by interleaving small groups of unrelated instructions. To do this, the processor will need enough registers to support more than one thread's temporary storage.
- Also to aid threading, the number of shared resources needed must be minimised. This suggests using multiple condition code registers or having no condition code register and using general registers to store condition values.
- Allow the most relaxed exception model possible, so that out-of-order completion can be used. This means that if a particular exception doesn't need to be recovered from, it shouldn't be necessary to preserve the process state. If exceptions sometimes need to be recovered from and sometimes don't it may be worthwhile indicating whether a precise or imprecise exception should be generated using an instruction bit or a status register flag.
- Avoid instructions that disrupt the pipeline organisation by executing multiple cycles or by requiring global communication, as these instructions will complicate and hence slow down the control of the pipeline.
- Define a set of branch instructions that can be executed by a remote branch unit; this means that a minimum of interaction between the branch unit and the datapath should be required, and that the instructions should be easily decodable by the branch unit.
- Define a conditional branch mechanism that suits the idea of a micropipelined remote branch unit. One possible scheme is to implement a fifo between the comparison output of the ALU and the branch unit to communicate the result of branch instructions.

With these observations made, the following two chapters go on to look at other aspects of processor architecture that lead directly to increased power efficiency. In Chapter 6 these ideas are revisited in order to define a general set of features for a power-efficient processor architecture.

Chapter 4 : Architectural Factors Influencing Power Efficiency

The previous two chapters have explored one way in which architectural choices can affect power efficiency: by incorporating certain features in the architecture, an asynchronous implementation is possible, and the use of asynchronous logic may give increased power efficiency. This chapter will consider other aspects of processor architectures and will show how they influence power efficiency.

This chapter starts by investigating where the power consumption in a computer system occurs, and determines what factors affect the scale of the power consumption. These factors are then related to architectural features.

Figure 4.1 illustrates one possible arrangement of the components of a computer system. This shows the memory systems for instructions and data, the instruction sequencing and decoding blocks and the register bank and ALU where the computations are carried out.

All blocks consume power. The amount of power consumed by a block depends on the rate at which it carries out its operations and by the amount of internal work that it needs to do for each of those operations. The power efficiency of a block can be increased by decreasing the number of actions that it needs to carry out per unit of computation or by reducing the amount of work that it needs to do for each of those actions.

In the following section, each of the blocks shown in figure 4.1 will be considered from this perspective. The power consumption will be discussed in terms of the characteristics of the blocks and their throughput. Once each block has been discussed, the architectural factors that have been considered are summarised.

In section 4.3 the most important architectural factor identified, code density, is studied in more detail.

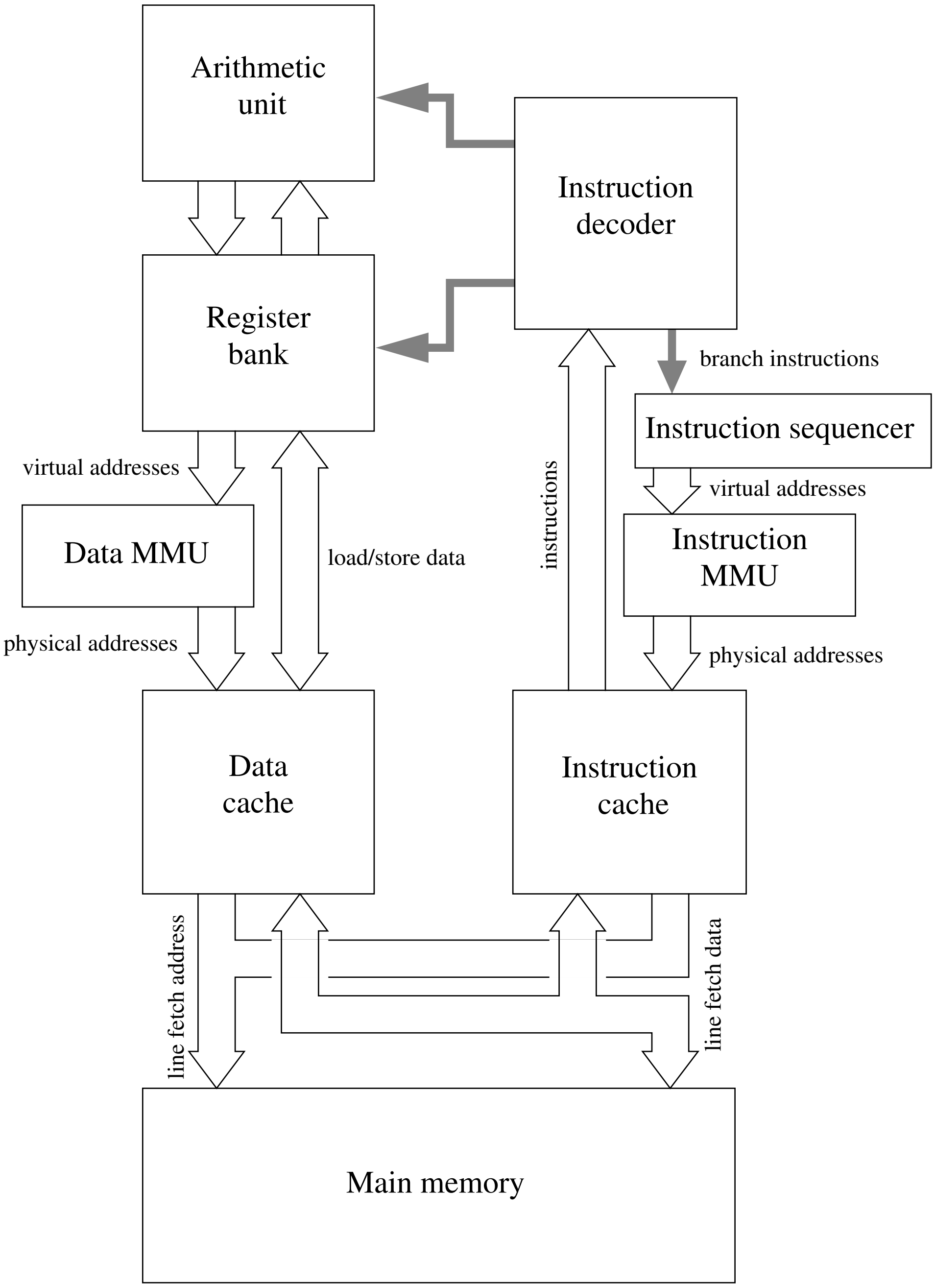


Figure 4.1: *Possible processor arrangement*

4.1 Power Consumption Block-by-Block

4.1.1 Main Memory

The power consumed by the main memory is proportional to the number of read and write operations that it has to carry out per second and to the number of bits read or written by each of those operations.

Because the main memory is not on the same chip as the processor, for each access the processor has to drive PCB tracks. The load of a PCB track is considerably greater than the load of a track within the processor chip[1], so the power dissipated by a main memory access is greater than the power dissipated by an internal operation. It is therefore preferable to access internal memory whenever possible. Typically accessing main memory is also slower than accessing internal memory, so for performance reasons it is also desirable to reduce the frequency of main memory accesses.

The frequency of main memory accesses depends on the following factors:

- The rate at which instruction fetches occur.
- The rate at which load and store operations occur.
- The proportion of instruction fetches and loads and stores that result in a main memory access.

Instruction fetches and load operations result in a main memory access when the required data cannot be found in the internal cache. The proportion of the time when this occurs depends on the size and organisation of the cache and on the behaviour of the program. A large cache or a small working set will give a low cache miss rate, and hence high power efficiency.

The effect of increasing cache size on power consumption is one of diminishing returns. Even an infinitely large cache will miss sometimes[2] and a main memory access will result. There is a trade-off between the cost of a larger cache and the power efficiency and performance improvements that result.

1. An approximate values for the capacitance of a track on a chip connecting a processor core to its cache is around 0.5 pF, whereas the combined capacitance of pads, packages and a PCB track connecting two physically close chips is approximately 15 pF. This indicates a thirty-fold difference.
2. Because of conflicts between different addresses for the same cache location, and because of loads after cache flushing.

Reducing the size of the program's working set means increasing the instruction set's code density, or applying higher-level modifications to the program to reduce the number of instructions executed.

The proportion of store operations that result in a main memory access is a function of the write policy in use. If a write-through cache or a write buffer is used, all write operations cause a main memory access. If a copy-back cache is used, write operations only cause a main memory access when the cache entry has to be copied back to the main memory. With a large write-through cache the majority of main memory accesses are likely to be writes, so changing to a copy back cache will lead to a significant improvement in power efficiency.

A reduction in the number of load and store operations can be obtained by increasing the processor's number of registers. With more registers, the program will need to use main memory to store values less often.

A reduction in the rate of instruction fetches can be obtained by increasing the processor's code density. With a higher code density, the processor can do more computational work using the same number of instruction fetches.

4.1.2 Caches

Caches carry out two basic operations: they are accessed by the processor core, and they are loaded from the main memory. The frequency with which they are loaded is the same as the frequency with which the main memory is accessed as discussed in the previous section, and the power used is proportional to the same factors.

Accessing the cache involves two operations:

- A content-addressable memory or an address tag RAM is used to find which cache line contains the required data (or if it is not present).
- The data is read from or written to the selected line of the data RAM.

The frequency of the look-up operation can be reduced by exploiting the sequentiality of accesses. Between branches, instructions come from sequential addresses. Caches are organised into lines storing sequential addresses. So it is necessary to access the address store only when a branch instruction is executed or when the end of a cache line is reached. The power consumed can therefore be reduced by increasing the line size and by reducing the frequency of branch instructions.

Increasing the line size has the disadvantage of potentially reducing the effectiveness of the cache, as ‘dead areas’ can occur at the ends of lines after branch instructions. Compiler techniques can reduce this effect by suitable code ordering [26].

The power used when the cache RAM is accessed depends on the frequency of the accesses, the number of bits read or written in each access, and the energy used by each bit read or written.

In the case of instruction fetch accesses, the number of bits accessed per unit of computation depends on the code density. Increasing the code density will therefore lead to greater power efficiency.

In the case of load and store accesses, a reduction in the frequency of load and store operations will increase power efficiency. Also, a reduction in the number of bits in a load or store will increase power efficiency. On processors with 32 bit registers, it is common to load and store variables whose values are known to cover only a small range in 32 bit words. This is inefficient from the point of view of power consumption.

The energy used by each bit read or written depends on the size of the cache. A larger cache will have longer tracks running through it, which will have larger capacitance. The energy used is proportional to the capacitance, so the energy used per access is proportional to the cache size.

There is therefore a conflict between increasing the cache size, which increases power consumption in the cache and reduces power consumption in the main memory, and reducing the cache size which reduces the power consumption in the cache and increases the power consumption in the main memory. Power used per access will be larger for the main memory, so the balance will tend to favour a larger cache.

Adding an intermediate second-level cache between a small low-power primary cache and the main memory may be the best solution, as this will minimise power consumption in the primary cache and in the main memory. Perhaps even a large number of small caches, each activated only when the previous level misses.

The idea of multiple levels of cache is particularly attractive in the context of asynchronous implementation, as an asynchronous processor can tolerate a wide variation in access time in a way that a synchronous processor cannot. In a synchronous processor, the access time has to be an integer number of clock cycles. In an asynchronous system, an arbitrary time can elapse between request and acknowledge signals.

4.1.3 Memory Management

The power used by the MMUs depends on their frequency of use and their complexity. The frequency of use depends on the frequency of instruction fetches and loads and stores, and on the proportion of these addresses that need to be translated.

The frequency of instruction fetch and load and store operations has been mentioned before and is dependent on the code density and the number of registers.

There are two ways in which the proportion of addresses that needs to be translated can be reduced:

- The same sequentiality arguments as were mentioned above for caches can be applied, and an instruction fetch address translation is only performed when a branch instruction is executed, or when a page boundary is crossed.

- The position of the MMU with respect to the cache can be changed. If the cache stores virtual addresses, the MMU only needs to be used when the cache misses. This has the disadvantage that when the virtual-to-physical address mapping changes, the cache has to be flushed. The other practical disadvantage of a virtual cache is that the time between a memory access being initiated and its success being confirmed is increased, because the address translation does not occur until after the virtual cache has been searched. If a hierarchy of caches is used (as discussed above), the address translation may occur at some point within the hierarchy.

The power used by the MMU for each translation depends on whether the translation required is present in the translation cache. Like the main cache, the power efficiency of the translation cache can be increased by using multiple levels of caching. This technique is used for the purpose of power saving in the MIPS R4200 processor [30].

The power used when a TLB miss occurs can be significant if software is used to find the necessary translation, as the software handler can execute a large number of instructions. If hardware is used to load a new translation from a table in memory, more silicon area is required for the memory management unit but less power is dissipated.

4.1.4 Datapath

The processor's datapath consists of the register bank, ALU and associated logic. The power consumed here per unit of computation depends on the number of operations needed per unit of computation and on the power consumed for each of those operations.

There are various factors that the instruction set architecture controls that determine how much power is consumed:

- The number of registers.
- The complexity of the functional units.
- The width of the datapath.

Having a larger number of registers or more complex functional units will increase the length of the datapath. This will lead to increased capacitance and hence to larger power consumption per operation.

On the other hand, having more registers reduces the number of load and store operations that are required, which saves power. Having functional units that are capable of complex operations replacing a number of simple operations gives a reduction in the total number of instructions required. For example, removing the multiply instruction will reduce the capacitance of the datapath but every multiply instruction has to be replaced by a sequence of add and shift instructions.

Using a wide (e.g. 32 bit) datapath to carry out operations on narrow (e.g. 8 bit) operands is inefficient because the remaining width of the datapath consumes energy without doing any useful computational work. On the other hand having a narrow datapath means that when wide operands are being used a sequence of instructions is required.

Various architectural features can be implemented to increase the efficiency of wide and narrow operations.

With a narrow datapath:

- Add-with-carry instructions make the implementation of wider arithmetic easier. For example, the MIPS R3000 and the ARM architectures both have 32 bit datapaths. The MIPS architecture has no add-with-carry instruction and performing a 64 bit addition requires 5 instructions. The ARM architecture does have add-with-carry and the 64 bit addition takes only 2 instructions.
- Wide arithmetic instructions can be implemented using a single multi-cycle instruction. This technique is common in small 8 bit microprocessors which are able to carry out 16 bit arithmetic on pairs of registers. It is also used in the floating point units of some processors.

With a wide datapath:

- Instructions can be provided to carry out narrower operations that activate only a portion of the datapath. This facility is provided in the 68000 processor family.
- It is possible to operate on multiple narrow operands at the same time. This is possible on any processor if the compiler is good enough for move and logical operations, but the behaviour of carries means that it cannot be used for arithmetic operations. The DEC Alpha architecture implements some interesting instructions for processing 8 8-bit operands simultaneously in one 64-bit word, including an 8-byte parallel compare operation.

An architecture for low power consumption should include one of these features to increase the power efficiency of its datapath.

4.1.5 Instruction Decoding and Sequencing

The power used by the instruction decoding logic depends on the instruction execution rate, the instruction length and the decoding complexity. The instruction execution rate and the instruction length are dependent on the code density; higher code density will tend to decrease the power consumption. On the other hand, more dense instruction encodings tend to require more complex decoding logic, which leads to greater power consumption in this block.

Some choices in the control logic will affect power efficiency. For example, some processors increase the performance of branch instructions by means of speculative execution. Instructions after a branch are issued although it is not known whether the branch was taken or not. When the direction taken by the branch is known, the instructions may either complete or be cancelled. Speculatively executing instructions consumes power which may turn out to be wasted. If the necessary performance can be obtained without speculative execution, removing it would save power.

4.2 Summary of Factors Affecting Power Consumption

Summarising the preceding section, the following architectural factors influence power consumption:

Code Density

A high code density will lead to increased power efficiency in many blocks of the processor, as the energy used in these blocks is proportional to the instruction fetch bandwidth. Most importantly high code density will reduce the power consumed accessing external memory, which is a highly power-consuming action.

In some of these blocks such as the memory management unit and the cache address store the energy used is proportional to the number of instructions executed. For the maximum power efficiency these blocks favour higher code density through fewer longer instructions. In other blocks such as the main memory and the cache data store, the energy used is proportional to the total code size in bits that is executed. In this case, power efficiency can be increased either through fewer instructions or shorter instructions.

If an instruction encoding with a greater density is used, the function that the instruction decoder has to carry out is made more complex. This means that the instruction decoder will consume more power for an instruction set with a higher code density.

There are other strong arguments in favour of high code density. Increasing code density leads to a reduction in the amount of main memory and backing storage required. This leads to reduced system cost, and in the case of portable equipment to reduced size and weight. Code density and the ways in which architectural features affect it are considered further in section 4.3.

"Cache Friendliness"

It is possible for some programs to exhibit better cache behaviour than others as a result of their memory access pattern. Programs that have a worse cache behaviour will access main memory more often and use more power. Building compilers that generate programs with good cache behaviour is an active research area [7].

It is likely that some instruction set features will lead to better cache behaviour than others. However it is not currently clear what these features are.

Number of Registers

An increased number of registers leads to a reduced number of load and store operations, which saves power because load and store operations can lead to main memory accesses. On the other hand, more registers lead to greater datapath length and more power consumption in the datapath per instruction. The appropriate number of registers will be a balance between these two factors.

Datapath Functions Provided

Datapath functions such as multiply should only be included if the increase in power consumption resulting from the greater datapath length can be offset by a reduction in the total number of operations.

Width of Datapath

It is inefficient to have a wide datapath such as the ARM's 32 bit datapath and not provide some sort of mechanism for saving power when narrower quantities are being used.

Speculative Execution

Speculative execution after branches results in wasted energy when the speculation is wrong. To remove the need to use speculation, the architecture must allow for branch instructions whose target address and condition can be computed quickly.

Memory Hierarchy Size and Organisation

The following features will lead to a power-efficient memory hierarchy:

- Multiple levels of cache, with a small low-power primary cache and larger intermediate levels.
- A large translation cache.
- Translation miss processing (table walking) in hardware.
- A write-back rather than a write-through cache.
- Long cache lines (to reduce the number of cache address tag look-ups).

However the organisation of the caches and memory management units is an implementation issue, not part of the architecture. There are few factors that the instruction set architecture specifies that restrict the choice of implementations. There are however some factors that can affect it:

- If a multiprocessor system is possible, cache consistency between the processors is required. This may make some cache options such as the write-back cache more difficult to implement.
- The exception model will influence the choice of translation mechanism (see section 3.2).

4.3 Code Density

Code density has been identified as significant in the power efficiency of a processor. The remaining part of this chapter investigates the way in which code density is affected by architectural choices.

4.3.1 Defining and Measuring Code Density

The code density of a processor can be defined as the average semantic content per bit of instruction for average programs in execution on that processor. This in turn depends on the average number of bits per instruction and on the number of instructions required for each unit of semantic meaning or computational work.

Let d be the code density, n be the number of instructions per unit of computational work and l be the average instruction length. They are related by the following equation:

$$d = \frac{1}{nl}$$

These variables d, n and l will be used in the following sections to describe the characteristics of the architectures and features considered.

For the purpose of evaluating power efficiency it is the dynamic code density that must be measured; that is the number of instructions that must be *executed* per unit of work, not the number that must be *stored*. The latter measure is evaluated to find the memory requirement of a processor. In general the dynamic and static code density of a processor will be similar.

Figure 4.2 plots estimates of n and l values for various processors. This data is approximate and is derived from various sources [7] [27] [31] [39], between which there is some disagreement. However the general trend is clear.

The following sections consider the properties of the average instruction length and the number of instructions per unit of computation, taking some of the processors illustrated in figure 4.2 as examples.

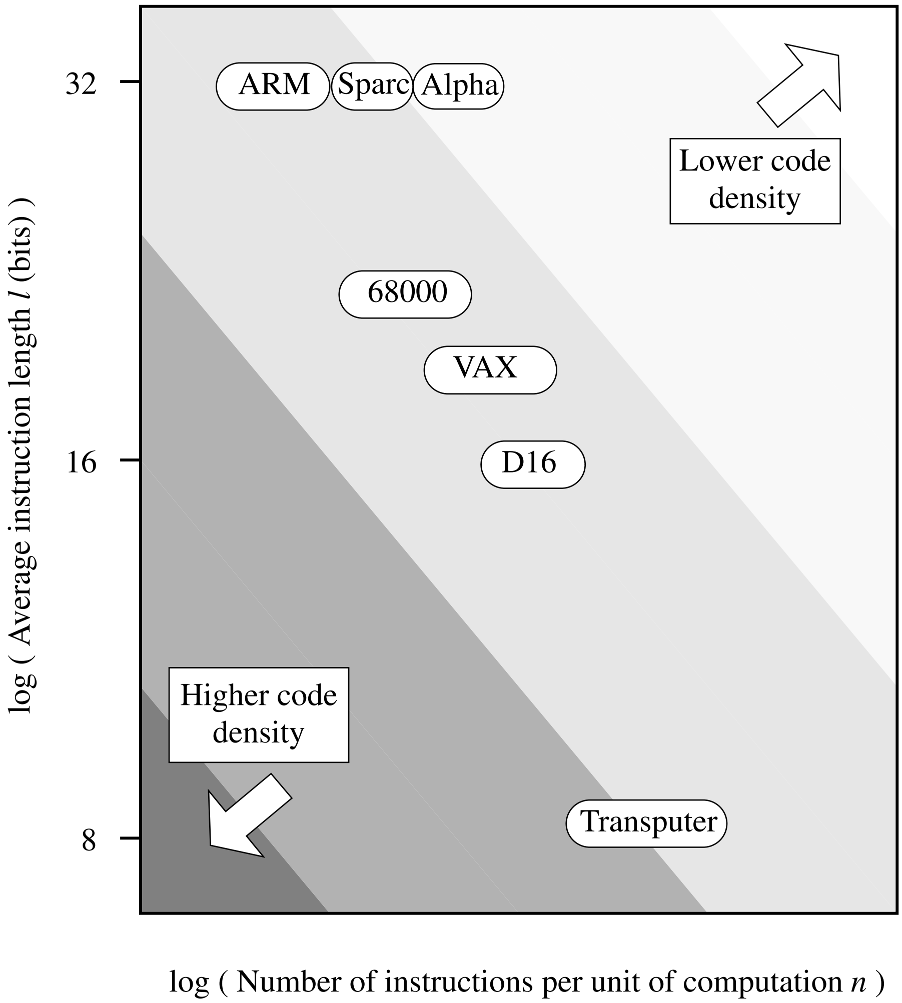


Figure 4.2: *Code density*

4.3.2 Fixed and Variable Instruction Lengths

Architectures can be divided into two categories with respect to the length of their instructions:

- Architectures with a fixed instruction length.
- Architectures with a variable instruction length.

In the former case, the average instruction length must be exactly the fixed instruction length. In the latter case, the average instruction length is a weighted average of the possible instruction lengths.

Most RISC architectures such as the Alpha, Sparc and ARM have fixed instruction lengths of 32 bits. There are two principle reasons why a fixed instruction length is advantageous:

- A fixed instruction length makes instruction formats more regular and hence instruction decoding is simpler.
- A fixed instruction width (of 32 bits) means that instructions are always aligned on word boundaries, which simplifies the process of fetching instructions and computing branch targets.

32 bit instructions are used by these processors because a 32 bit instruction is long enough to encode useful numbers of five-bit register specifiers, opcode bits and immediate fields, and because other aspects of the architecture such as the datapath are also 32 bits wide.

CISC architectures, including the 68000 and the VAX, typically have variable instruction lengths, based on multiples of 8 or 16 bits. This reflects the increased range of complexity present in the CISC instruction sets.

Another reason for the difference between the RISC and CISC instruction encodings is that the CISC architectures predate the RISC architectures, and over time the cost of memory has fallen. The objective of increasing code density in order to reduce the amount of memory required has become less important, so the RISC architectures are more prepared to ‘waste’ some instruction bits in order to reduce complexity.

There are a few examples of RISC processors that have a variable instruction length. The ROMP processor is an early RISC processor, and at the time of its development the benefit of reducing memory use was considered important. The ROMP therefore has 16 or 32 bit instructions.

In general the use of fixed instruction lengths leads to lower code density, as fixed length instructions will tend to include unused bit fields and variable length instructions can assign shorter instruction formats to more frequently used instructions. The benefit of this effect is studied further in chapter 5.

4.3.3 The Semantic Content of Instructions

The number of instructions required to carry out a unit of computational work is a complicated function of the architecture, and is also related to the sophistication of the compiler used. A compiler can be thought of as translating higher-level operations into sequences of machine code instructions. The length of these sequences depends on what can be done by a single instruction.

The Sparc, Alpha and MIPS Architectures

These conventional RISC architectures are very similar in many respects and so have similar code densities. In general the code density is low because the instructions contain unused fields and fields that are sparsely encoded.

Some of the differences between the architectures do lead to differences in the code density. The most important case is the register windows in the Sparc architecture. By including register windows, the Sparc reduces the number of loads and stores that have to be carried out in the program.

The ARM Architecture

The ARM architecture has a fixed length 32 bit instruction format similar to the Sparc, Alpha, etc. However it has a higher code density than many of the other RISC processors because it encodes more semantic content into each instruction. Examples include:

- A single instruction can specify an action for both the arithmetic unit and the shifter, for example `ADD R1,R2,R3 LSL R4`.
- Load multiple and store multiple instructions replace up to 16 separate load and store instructions.
- Load and store instructions have optional post incrementing and decrementing modes.
- Conditional instructions remove the need for some short branches.

The ARM adds this extra functionality at the expense of having only 16 registers, compared with the 32 registers of the Sparc, Alpha, etc. It also leaves fewer unused bit fields in its instructions than some of these architectures. The ARM is believed to have a code density similar to that of the 68000 processor, which is approximately 1.6 times that of the conventional RISC processors.

The D16 Architecture

Like the ARM, the D16 architecture [31] is similar to the conventional RISC architectures in many ways. However in contrast to the ARM it achieves higher code density not by increasing the semantic content of the instructions but by reducing their length to 16 bits.

D16 is a hypothetical variation on the DLX processor, which is in turn very similar to the MIPS architecture. D16 has 16 registers, two-address operations (i.e. one of the source registers is used as the destination), and a more complex instruction encoding than MIPS. D16 has a code density 1.5 times that of MIPS, which indicates that its semantic content per instruction is about 0.75 times that of MIPS.

The SH7000 architecture from Hitachi has 16-bit instructions that are similar to D16. The SH7000 is targeted at portable applications requiring low power, so the increased code density is beneficial.

The Transputer

The Transputer is unique in the processors studied here in that it has fixed-length 8-bit instructions. In many other ways the transputer is also an unusual architecture; for example it has a stack based datapath rather than a general purpose register bank.

Although one view is that the instructions are of fixed lengths, it should be noted that some of the instructions take no direct action themselves but act as prefix instructions that modify the action of the following instruction. The most important case of this is an instruction that appends a four bit value to an internal constant register, so if a 12 bit addition is wanted three of these instructions are used to generate the required 12 bit value. In other instruction sets, these groups of bytes would be defined as a single instruction, rather than a series of instructions. This is however only a question of terminology as the code density is independent of where the lines between instructions are drawn.

The Transputer's stack architecture rather than a register bank exploits locality of reference in a way that other processors cannot. Transputer instructions automatically use the result of the last instruction because it is at the top of the stack. In a RISC processor, a register specifier is needed to indicate what register every operand should come from, even in the common case where it is the result of the last operation. See section 5.6.1 for a discussion of the frequency of last result reuse.

4.4 Conclusions

In this chapter, the power consumption of a processor has been studied and architectural features that influence it have been identified. Code density has been considered in most detail as its effect is believed to be of most significance.

Code density depends on average instruction length and the number of instructions required per unit of computation. Fixed length and variable length instruction encodings have been compared, and the features of a number of architectures that give them high or low code density have been examined.

In the following chapter, the code density of the Sparc architecture is examined in more detail and in a quantitative fashion, to find the potential for increasing its code density.

Chapter 5 : The Potential For Increased Code Density

The previous chapter has suggested that code density is an important influence on power efficiency, and that code density depends on the average instruction length and the number of instructions required to perform each unit of computational work.

This chapter goes on to study the potential for increasing the code density of an instruction set. In the first section, the experimental technique used is described. In subsequent sections, the instruction set of the Sparc architecture is considered and ways in which its code density can be increased are proposed. The techniques considered include reducing the instruction length, making the instruction length variable, and changing the number and size of register specifiers.

The Sparc architecture is similar to other RISC architectures such as the Alpha and MIPS in various important ways; many of the measurements made here will be generally applicable to this category of processors. The Sparc architecture has been studied because a high-quality simulator, Shade, has been available.

At various points, as alternative features are proposed, the possibility of implementing the instruction set asynchronously is considered.

The chapter concludes by considering how the ideas presented can be reconciled with the need to maintain a relatively simple instruction encoding with a small number of different instruction lengths.

5.1 Experimental Procedure

Most of the measurements described in this chapter are based on a series of simulations carried out using a simulator called Shade [36]. Shade executes a Sparc program compiled on a Sun computer, and allows the user to accumulate various statistics as the program executes. Much of the data is presented in appendix B. In this chapter, code density results are calculated from that data.

The benchmark programs used were chosen to represent a cross-section of those available on the Unix system used; they include a compiler, a compression program and a graphics file manipulation program. In practice, for the statistics being measured there was little variation between the programs. The actual programs used are listed in appendix C.

In some sections data obtained in other ways is used, and in these sections the source of the data is indicated.

The effect of the code density changes is described in terms of the variables d, n and l that were introduced in section 4.3.1. These values are measured relative to the standard Sparc architecture; so, for example, if a particular change increases d to 1.2, then it has increased the code density relative to the standard Sparc by 20%.

5.2 The Sparc Architecture

The principle features of the Sparc instruction set are illustrated in figure 5.1.

- Fixed length 32-bit instructions.
- 31 general purpose integer registers visible at a time.
- Standard set of 32-bit 3-operand arithmetic and logical instructions.
- Load/Store architecture: the only instructions that can access data in memory are load and store. All other instructions operate on data in registers.
- Multiple overlapping register windows.

Figure 5.1: *Characteristics of the Sparc processor*

Sparc instructions are made up from bit fields of up to four different types:

- Unused fields. Because the instructions are always 32 bits long and some instructions do not need that many bits, some are unused.
- Opcode fields, which specify which operation is to be carried out.
- Immediate fields, which specify constant values used in the instruction.
- Register specifiers, which indicate which registers are to be used as sources and destinations.

The potential for increased density in each of these fields is considered in turn in the following sections.

5.3 Unused Fields

The most important example of an unused field is an 8 bit field that occurs in all arithmetic and logical and load and store instructions that specify three registers rather than two registers and an immediate value.

Data described in section B.1 shows that the above instruction types make up 13.1% of all instructions in dynamic execution for the benchmark programs. If these fields were eliminated, the length of these instructions would be reduced from 32 bits to 24 bits. This would reduce the relative average instruction length l to 0.96. Since the relative total number of instructions n would be unchanged, the relative code density d would increase to 1.03.

5.4 Opcode Fields

The majority of Sparc instructions use a total of 9 bits to specify the opcode. Some instructions that require more bits for other purposes have fewer opcode bits; for example the call instructions has only 2 opcode bits and the SETHI instruction has 5 opcode bits. Branch instructions have 10 opcode bits.

Some instructions are used to implement multiple functions. The most important example is the add immediate instruction, which is used for at least three functions:

- To add a register and a constant.
- To move a constant into a register: the source register is set to R0, which always reads as zero, and the immediate value to load is placed in the immediate field.
- To move one register into another: the immediate field is set to zero.

Another example is the subtract instruction, which is also used to perform the compare function.

Each of these functions is considered separately in the following, and the length of the opcode is the total number of bits required to specify the function. For example, to carry out a move immediate the total opcode length is 14 bits, made up from 9 bits for the add immediate opcode and 5 bits for the register specifier for R0. The lengths of opcodes under this definition vary from 2 bits to 22 bits.

Data presented in section B.1 gives the relative frequency of each of the Sparc instructions. The data shows that quite a small number of instructions dominate the dynamic instruction counts. By computing a weighted average, the mean opcode length has been found to be around 11.2 bits.

Although an 11 bit opcode can specify 2048 different instructions, only 16 instructions make up about 80% of those executed. There is clearly a significant amount of redundancy in this instruction encoding. Reducing this redundancy could lead to increased code density.

There are two ways to reduce the redundancy:

- A large number of the less often used opcodes could be removed entirely from the instruction set. When the compiler wants to carry out one of these operations, it is then forced to use a sequence of the more common instructions. This involves a trade-off of increasing n (number of instructions) against decreasing l (average instruction length).

- The length of the opcode field could be made variable, and shorter codes allocated to more frequent instructions. This involves a decrease in l with n remaining constant.

The benefits of each approach are considered in the following sections.

5.4.1 Reducing the Number of Instructions

The idea of reducing the number of instructions in a processor's instruction set by removing the less useful ones is one of the principles of the RISC philosophy.

Infrequently executed instructions can be divided into a number of categories, some of which may be candidates for removal:

- Instructions that are simple to replace with a small number (e.g. less than around 5) of common instructions.

- Instructions that need a longer sequence of common instructions to replace them, or perhaps a call to a library function.

- Irreplaceable instructions.

Table 5.1 lists some examples of these instructions.

Instruction	Possible replacement sequence
Short replacement sequences	
Add double	Add single, carry out Add single, carry in

`A = B XOR C`	`x = B OR C`
	`y = B AND C`
	`z = NOT y`
	`A = x AND z`

Long replacement sequences	

`A = B << C`	`    A = B`
	`    x = C`
	`L1: IF x == 0 STOP`
	`    x = x - 1`
	`    A = A+A`
	`    GOTO L1`

`A = B / C`	`CALL DIV`
Irreplaceable	
enable interrupts	
flush cache	

Table 5.1: *Replacement sequences for infrequent instructions*

The number of instructions that cannot be replaced depends on the amount of special state the processor has, as these instructions typically carry out privileged functions or modify operating modes. The number of instructions of this type can be minimised by using a simple set of control registers.

The first instructions to be removed from the instruction set are those most infrequently used and which have the shortest replacement sequences.

Consider the possibilities of implementing four and five bit fixed-length opcode fields. If the number of infrequent irreplaceable instructions is zero these fields can specify the 16 and 32 most frequent instructions respectively. From the measurements taken in section B.1, 16 opcodes can specify 80% of all instructions and 32 opcodes can specify 98%.

If the average number of instructions in the replacement sequence for a removed instruction is r, then with a four-bit opcode field the relative total number of instructions n

would be $80\% \times 1 + 20\% \times r$. With a five-bit opcode field n would be $98\% \times 1 + 2\% \times r$.

A four bit opcode field would save 7 bits compared with the standard Sparc's average of 11 bits to specify the opcode, so the relative instruction length l would be $\frac{32-7}{32} = 0.78$. A five bit opcode would save 6 bits, so l would be $\frac{32-6}{32} = 0.81$.

The relative code densities are given be $d = \frac{1}{nl}$. Values of d for various values of r are given in table 5.2.

average replacement sequence length r	relative density d	
	opcode field length = 4	opcode field length = 5
2	1.07	1.21
4	0.80	1.16
8	0.53	1.08
16	0.32	0.95

Table 5.2: *Relative density resulting from 4-bit and 5-bit opcode fields*

These results show that a fixed opcode field length of 4 bits could only lead to increased code density if the average replacement sequence length was 2 instructions or fewer. This seems unlikely. On the other hand, a fixed opcode field length of 5 bits would give an 8% increase in code density even if the average replacement sequence length was 8 instructions. Using a five bit fixed length opcode field is an interesting possibility.

However these results are only valid if the number of irreplaceable instructions is nil. If, for example, there were 16 irreplaceable instructions, the benefits of the 5 bit opcode field would be reduced to those of the 4 bit field.

5.4.2 Variable Length Opcode Fields

The objective of using variable length opcode fields is to give short opcodes to the most frequent instructions and long opcodes to infrequent instructions. By making the opcode length completely variable, i.e. allowing it to take any integer length, the highest code density can be achieved.

The optimum encoding can be found by using a Huffman encoding [37]. The Huffman encoding involves repeatedly subdividing the ordered list of opcodes into blocks with equal frequency until the blocks contain only one instruction. The technique is illustrated in figure 5.2.

Instruction	Cumulative Frequency
load word 1 reg	0.111
move register	0.207
move immediate	0.294
add immediate	0.371
branch if equal	0.424
branch if not equal	0.472
load byte 1 reg	0.518
compare registers	0.562
shift left immediate	0.604
compare immediate	0.645
store word 1 reg	0.681
add registers	0.716
subtract immediate	0.746
unconditional branch	0.769
or immediate	0.788
. . . .	. . . .

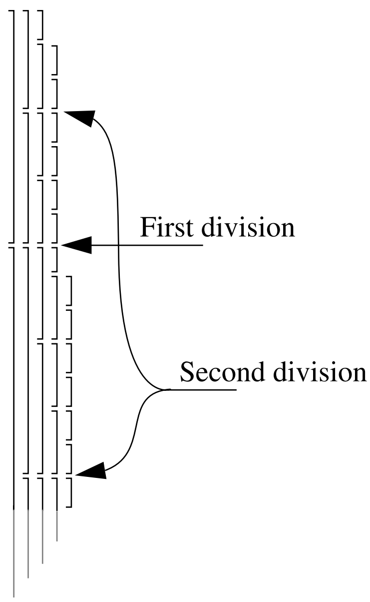


Figure 5.2: *Assigning divisions for the Huffman encoding*

Under this encoding, the most frequent instruction (load word register + immediate) has a three bit opcode and the next 7 most frequent instructions have four bit opcodes. The average opcode length is 4.7 bits.

Since the standard Sparc encoding has an average opcode length of 11.2 bits, it must have a redundancy of $1 - \frac{4.7}{11.2} = 58\%$.

This encoding reduces the average instruction length from 32 bits to 25.5 bits, representing a relative average instruction length l of 0.80. Since the total number of instructions remains unchanged, this increases the relative code density d to 1.25.

5.4.3 Asynchronous Opcode Decoding

The Huffman encoding naturally associates frequently used instructions with short bit fields. Following the principle of "making the common case fast and the rare case correct", it is reasonable to use a decode mechanism whose latency is proportional to the length of the opcode.

Figure 5.3 illustrates how an asynchronous decoder could be built using this principle. At the left hand side is the opcode bundle, consisting of a request/acknowledge pair and the opcode data itself. At the right hand side is a set of output signals, one corresponding to each opcode. Each of the rectangles in the figure is a select element. These take a boolean control input on the top input (indicated by a solid diamond), and when a transition arrives on the left-hand input, it is steered to either the top or the bottom right-hand output, depending on the state of the boolean control input.

This circuit decodes two opcodes of length 2 bits, one opcode of length 3 bits, and 6 opcodes of length 4 bits. The latency of the decoder is proportional to the number of select elements that the signal passes through, and as can be seen exactly one select element is used for each significant bit in the opcode. The latency is therefore proportional to the number of bits in the opcode, and if the opcodes are Huffman encoded, the latency will be inversely proportional to the frequency of the opcodes.

It should also be noted that since power dissipation is proportional to the number of signal transitions, the power consumed in this circuit is also minimum for the most frequent opcodes.

5.5 Immediate Fields

Section B.3 presents data showing the distribution of the values used in immediate fields. This data and the summary in table 5.3 show that often the number of bits provided for the immediate value by the Sparc architecture is larger than is necessary.

For some instructions, the potential for increased density is greater than for others. Referring to table 5.3, it is apparent that load, store and the logical operations are poor candidates for increased density, whereas the branch instructions and subtract[1] have significant redundancy.

1. The reason for the typically small subtract immediates is the high frequency of subtracting 1 in a loop control variable. Note similarly that around 60% of add operations add 1 (see figures B.3 and B.4).

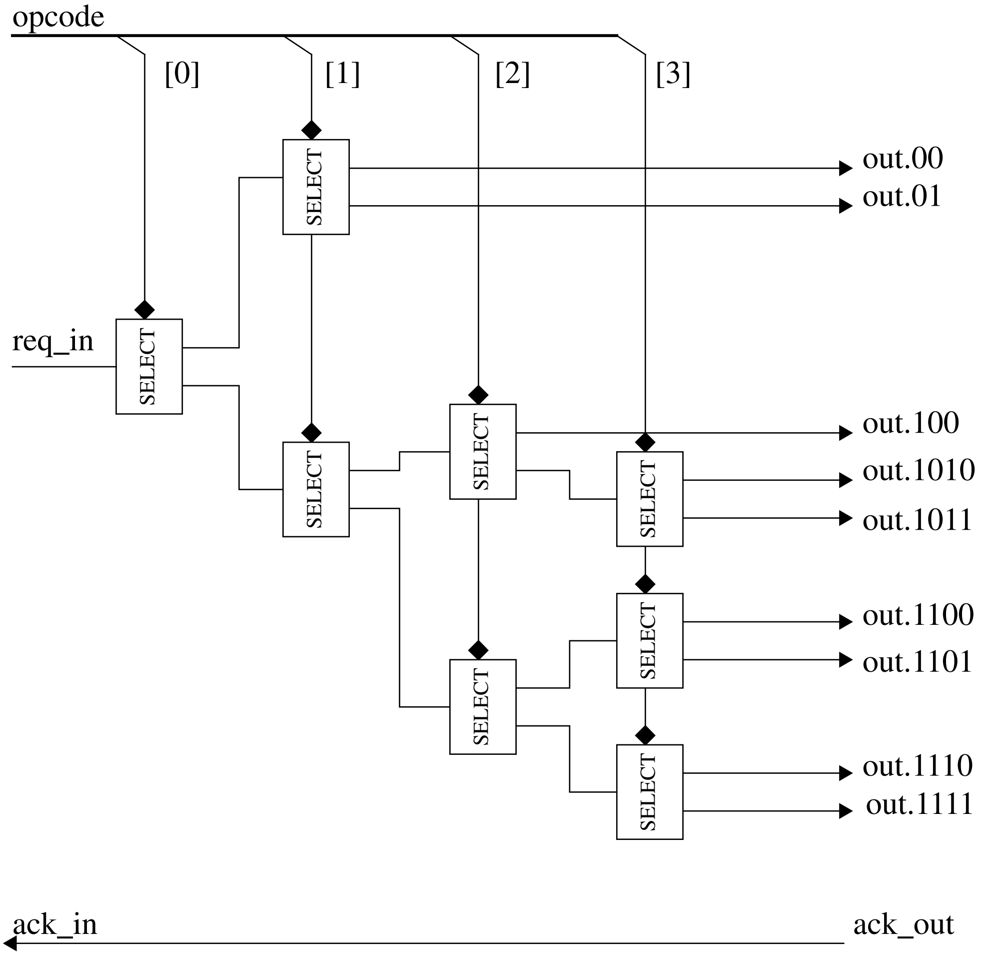


Figure 5.3: *Asynchronous variable-latency instruction decoder*

In the following sections, possible increases in code density by changing the length of the immediate fields for branch instructions and add and subtract instructions are considered. Addition is included because although table 5.3 does not show great potential for improvement, it is a very common instruction and any increase in code density would have an important influence on the overall density of the instruction set.

Reducing the length of an immediate field reduces the range of values that can be represented by the field. When the compiler wishes to use a value outside the range that can be represented by the immediate field it is forced to use an alternative mechanism. For arithmetic and logical operations, this will typically mean computing the value in a temporary register. For branch operations, the compiler must compute the branch target in a register and then move the register to the program counter. The use of these mechanisms results in an increase in the relative total number of instructions n. This must be bal-

Instruction	Number of bits provided in the Sparc encoding	Number of bits sufficient for 90% of cases
add	13	7
and	13	8
andn	13	8
call	30	16
compare	13	7
conditional branch	22	8
load word	13	10
move immediate	13 or 22[a]	10
or	13	10
store word	13	10
subtract	13	3
unconditional branch	22	10

Table 5.3: *Potential for shorter immediate fields*

a. 13 bits for normal move immediate, 22 bits for the SETHI instruction

anced against the reduction in the relative average instruction length l resulting from reducing the length of the immediate field to find the optimum length.

5.5.1 Immediate Fields in Branch Instructions

Figure 5.4 is derived from data shown in figures B.11 and B.12 and shows how code size is affected by the length of branch instructions. The lower dark area represents the change in the average instruction length resulting from a branch immediate field of the length indicated. The upper light area includes the effect of the extra computed long branches that are required when the immediate field is not long enough. This figure assumes that two instructions are needed to prepare an arbitrary branch target address and a third instruction is needed to jump to this address.

The figure shows that the best code density can be obtained with an immediate field of 10 bits for branch instructions. This means that branch instructions would be 20 bits long, and that the relative average instruction length l would be reduced to 0.93. The extra instructions required when the branch immediate is not long enough would increase the relative total number of instructions n slightly to 1.008. This would increase relative code density d to 1.07.

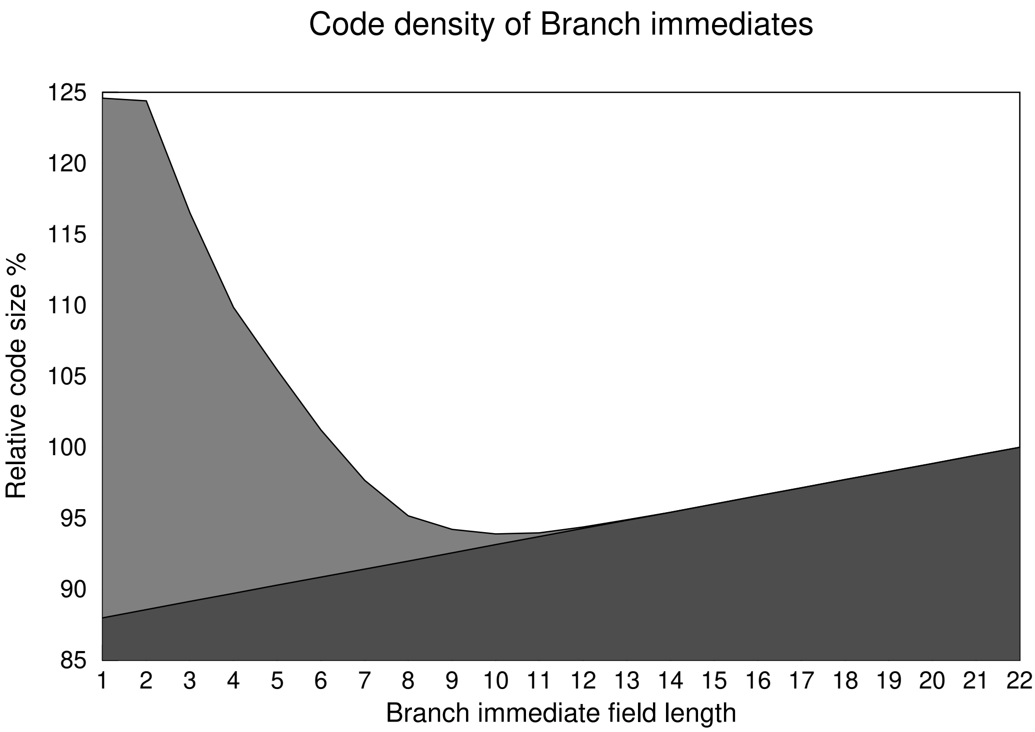


Figure 5.4: *Branch immediate field length and code density*

5.5.2 Immediate Fields in Addition and Subtraction Instructions

Using the same technique as in the previous section, figure 5.5 shows that the optimum length for add and sub immediates is also 10 bits.

However, comparing figures 5.4 and 5.5 note that the behaviour around shorter immediate field lengths is significantly different. It seems that the arithmetic operations could also benefit from a shorter immediate field length of around 6 bits, as the curve shows a local minimum at this point. Also, as has been previously noted, adding and subtracting 1 are extremely common operations. It is therefore worth considering using a selection of different immediate field lengths for these instructions. Figure 5.6 shows the effect on code density of adding a second shorter immediate field once a 10 bit immediate field and add and subtract 1 instructions are in existence.

This result shows that the optimum code density can be obtained by using instructions with immediate field lengths of 3 and 10 bits, and instructions that add and subtract one.

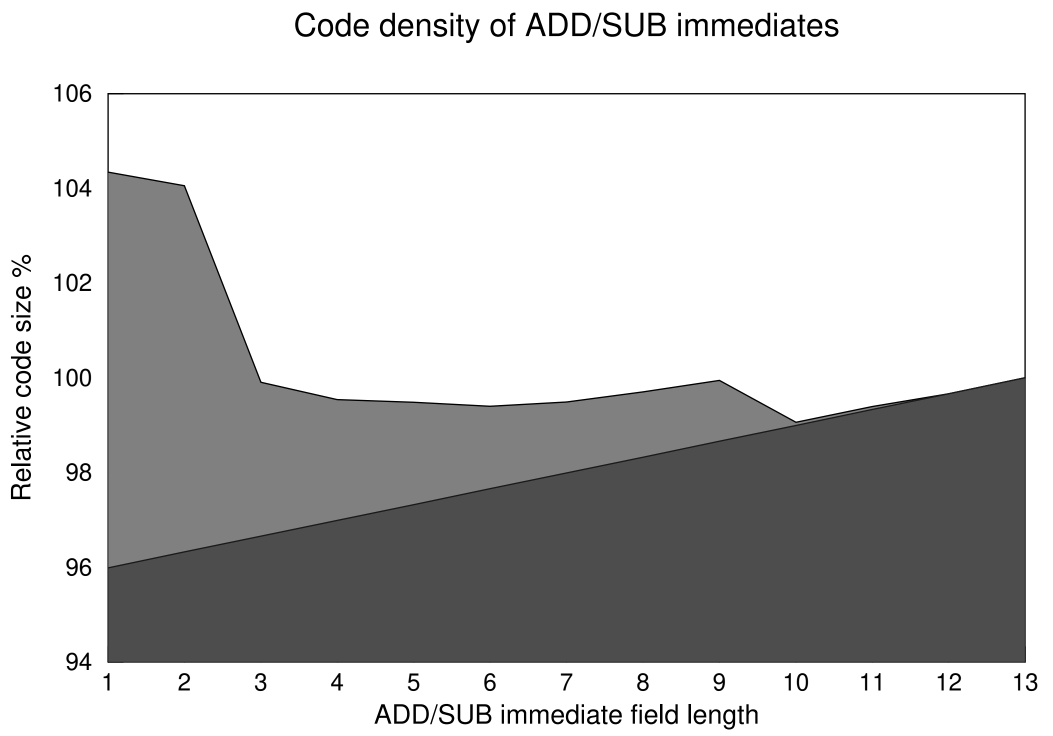


Figure 5.5: *Arithmetic immediate field length and code density*

Table 5.4 shows that with these instructions the average length of an add or subtract instruction can be reduced from 32 bits to 21.2 bits.

Instruction	Percentage of arithmetic instructions	Instruction length / bits
add or subtract 1	63.0	19
3 bit immediate field	22.1	22
10 bit immediate field	14.9	29
Weighted average		21.2

Table 5.4: *Length of arithmetic immediates*

This results in an overall relative average instruction length l of 0.96. The small number of extra SETHI instructions increase the relative total number of instructions n to 1.001. The relative code density d is therefore increased to 1.04.

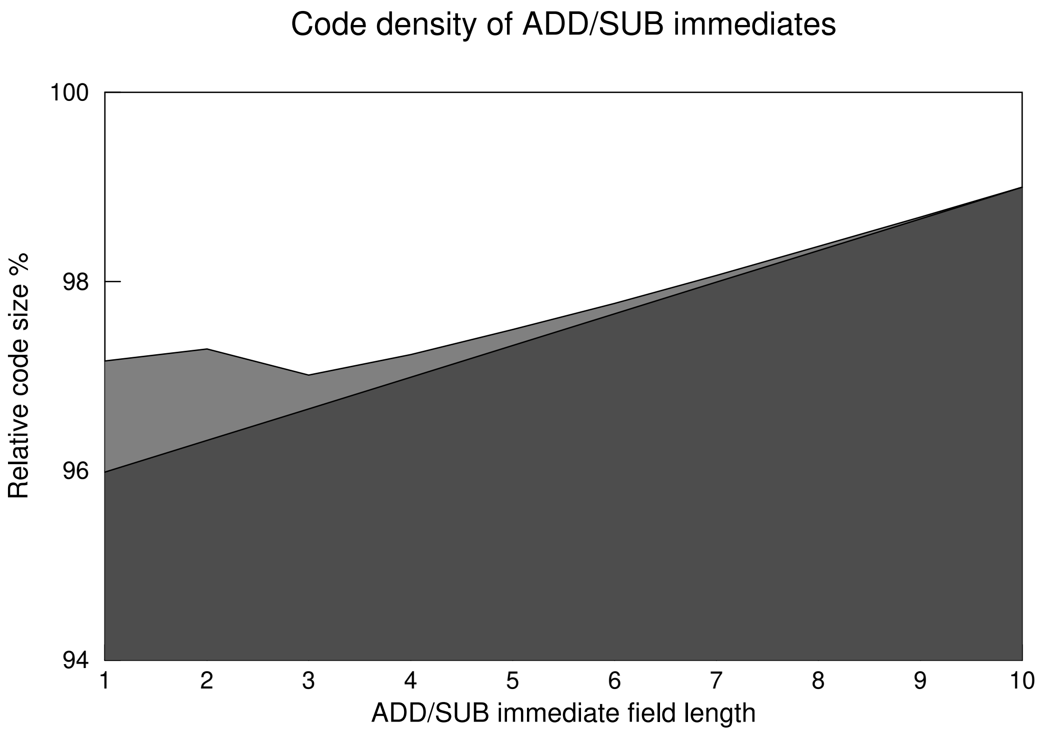


Figure 5.6: *Adding a second arithmetic immediate field*

5.6 Register Specifiers

The sparc architecture has 31 general-purpose registers, and a 32nd pseudo-register that reads as zero. It therefore requires five bits per register specifier.

Like most RISC architectures, the Sparc has three-address instructions. This means that each operand and result can use a different register. In contrast, other architectures have two-address instructions so that only two registers are specified in the instruction and the destination is one of the source registers.

There are therefore several approaches to increasing the code density of the register specifiers:

- The number of registers can be reduced, making the register specifiers shorter.
- The number of register specifiers per instruction can be reduced.
- The number of register specifiers needed can be reduced through the use of special-purpose registers for some functions.

These three approaches are considered in the following sections.

5.6.1 Number of Register Specifiers Per Instruction

2-Address Instructions

As mentioned above the Sparc instruction set has 3-address instructions. However it is very common for instructions to specify the same register in more than one of their register specifier fields; this corresponds to high level language operations such as `x=x+1`.

If 2-address instructions were used, the length of many instructions could be reduced by 5 bits from 32 bits to 27 bits. Data in section B.4 indicates that 57.8% of instructions could have their length reduced in this way.

There are then two alternative ways to deal with the remaining instructions that do have distinct register specifiers:

- Three-address instructions can be used along with the new two-address instructions.
- The three-address instructions can be transformed into a pair of two-address instructions; for example the operation `A=B+C` can be converted into the sequence `A=B; A=A+C`.

In the former case, the relative total number of instructions n is not increased. The relative average instruction length l is reduced to 0.95 because 31.4% of instructions have duplicate register specifiers. This results in a relative code density d of 1.05. However, the number of opcodes needed is increased which will reduce the benefit.

With the latter approach, the relative total number of instructions n is increased to 1.26 because of the extra move instructions required. The relative average instruction length l is reduced to 0.90 because 57.8% of the existing instructions and the 26.3% of new move instructions are 5 bits shorter. This gives a relative code density d of 0.88; this reduces the code density.

Explicit Last Result Re-use

The previous sub-section has investigated the potential for increasing code density by exploiting the fact that multiple register specifiers in the same instruction may be the same. This section investigates the related idea of register specifiers in adjacent instructions being the same.

The most common case where the same register is used by two instructions in sequence is when one register reads the result written by the previous instruction. This is referred to as last result re-use, and its frequency is also important from the point of view of implementations; see section 3.1.

To exploit re-use, new instructions are required that have appropriate register specifiers omitted but which use the result of the last instruction. The presence of these instructions will lead to an increase in the number of opcodes needed, and this increase must be offset against the decrease in instruction length resulting from the removed register specifiers.

Section B.2 studies the frequency of last result re-use, and finds that around 29% of instructions use the result of the last instruction. If the length of these instructions was reduced by 5 bits, the average instruction length would be 30.5 bits and the relative average instruction length l would be 0.95. Since the total number of instructions is unchanged, the relative code density d is 1.05.

Section B.2 also shows that the results of around 21% of instructions are used by the following instruction and are then never used again. In this case, it is not necessary for either the first or the second instruction in the re-use pair to give a register specifier. This means that five bits can be saved from 50% of instructions, giving an average instruction length of 29.5 bits and a relative code density d of 1.08.

5.6.2 Number of Bits Per Register Specifier

The number of bits per register specifier is the logarithm to base 2 of the number of processor registers. Doubling the number of registers increases the length of a register specifier by 1 bit. Data from table B.3 indicates that on average there are about 1.6 register specifiers per instruction, so each doubling of the number of registers increases the average instruction length by 1.6 bits.

Adding more registers can however reduce the number of operands that need to be stored in memory, so the number of load and store instructions required is reduced. Reducing the number of load and store instructions leads to a lower total number of instructions.

For some number of registers, these two factors will balance to give the best relative code density. To find this balance, it is necessary to find the relationship between the number of registers and the frequency of loads and stores. The model for the pattern of data accesses presented in appendix A provides the basis for this relationship. Appendix A finds a relationship between the number of storage locations used and the frequency of access to those storage locations for benchmark programs.

Assume that a processor with r registers will use those registers to store the r most frequently used variables at any time, and will use memory to store all the other less frequently used variables. Let the threshold frequency f_{th} be the access frequency above which variables are stored in registers. From equation 3 in appendix A, r and f_{th} are related as follows: (b and c are constants)

$$r = \int_{f_{\mathrm{th}}}^{\infty} a(x)dx = \int_{f_{\mathrm{th}}}^{2} a(x)dx = b\left(\left(\frac{1}{f_{th}}\right)^{c} - \left(\frac{1}{2}\right)^{c}\right)$$

Rearranging: (assuming $\frac{r}{b} » \left(\frac{1}{2}\right)^{c}$)

$$f_{\mathrm{th}} = \left(\frac{b}{r}\right)^{\frac{1}{c}}$$

The frequency of access to all memory locations, f_{m}, is the sum of all access frequencies less than f_{th}:

$$f_{\mathrm{m}} = \int_{0}^{f_{\mathrm{th}}} xa(x)dx = b\frac{c+1}{1-c}\left(\frac{1}{f_{\mathrm{th}}}\right)^{c-1}$$

Substituting:

$$f_{\mathrm{m}} = b^{\frac{1}{c}}\left(\frac{c+1}{1-c}\right)r^{1-\frac{1}{c}}$$

Since the value of c measured in appendix A is 0.81, we can derive the following relationship between the number of registers and the frequency of memory accesses:

$$f_{\mathrm{m}} \propto r^{-0.23}$$

Hence doubling the number of registers will reduce the frequency of memory accesses to $2^{-0.23} = 0.85$ of the previous level.[1]

1. This model ignores some significant factors. The most important two are that the compiler is not always able to use all of the registers present, and that in order to access operands in memory locations, some registers must be used to compute addresses and store the value once it is loaded.

It is now possible to find the effect on code density of increasing or decreasing the number of registers. Each time the number of registers is doubled, the average number of bits per instruction is increased by 1.6 because of the increased length of the register specifiers and the number of load and store instructions is reduced to 0.85 of its previous number. Table 5.5 computes the magnitude of these effects for various numbers of registers.

bits per register specifier	number of registers	average instruction length	relative average instruction length l	proportion of instructions that are load or store	relative total number of instructions n	relative code density d
3	8	28.8	0.9	0.32	1.09	1.02
4	16	30.4	0.95	0.27	1.04	1.01
5	32	32.0	1.00	0.23	1.00	1.00
6	64	33.6	1.05	0.20	0.97	0.99
7	128	35.2	1.10	0.17	0.94	0.97

Table 5.5: *Number of registers and code density*

The results shown is this table indicate that from the starting point of 32 registers, code density can be increased by decreasing the number of registers; the effect of the reduced instruction length is greater than the effect of the increased number of load and store instructions. However the size of the increase is rather small - only 1% for the move from 32 to 16 registers.

It should be remembered that reducing the number of load and store instructions is doubly good for power efficiency, as each load or store instruction that is executed consumes power both for instruction fetch from memory and for data transfer. This point is considered further in section 6.1.

5.6.3 Special Purpose Registers

The use of special purpose registers is unusual in RISC processors, which apply the idea of a homogeneous general-purpose register bank. There are however a few significant exceptions to this that are worth investigating.

Link Address Register

The Sparc's CALL instruction, which is used for subroutine entry, places a copy of the old program counter into register 15. Similarly, the ARM processor copies the old program counter into register 14. These register numbers are hard-wired.

Using this mechanism, when a function calls another function it must always save the previous value from this link register. On the other hand, some other local variables may not need to be saved if the compiler is able to carry out inter-procedural register allocation.

It would be possible to add a register specifier to CALL instructions, indicating which register should be used for the link address. This would increase the length of the CALL instruction by five bits, but it may be offset by the reduced number of store instructions required in the called function.

It has not been possible to measure the benefit or disadvantage of a special link register because no machine with a general-purpose link register has been available. However, using data from the experiments described in appendix B it seems that any benefit either way would be very small, since the frequency of procedure entry is low.

Frame Pointer or Workspace Pointer

In a program compiled from a high-level language such as C, many accesses to main memory are to variables that are located relative to a frame pointer or workspace pointer of some sort. Because this register is used more often than other registers in load and store instructions, it is possible to have special load and store instructions that imply the use of this register and require one less register specifier. Architectures such as the Transputer and the Hitachi SH7000 use registers of this type.

Load and store instructions make up 23% of Sparc instructions overall, so if in half[1] of these cases the base register specifier could be removed and replaced with an implicit reference to a frame pointer register, the relative average instruction length l would be reduced to 0.96 and the relative code density d would be increased to 1.03.

The disadvantage of using a special-purpose frame pointer register is that it is not possible to use different registers in different procedures and to apply inter-procedural register allocation. The effect of this has not been measured.

1. Half is an arbitrary proportion, this factor has not been measured.

5.7 Features of the ARM Instruction Set

The ARM instruction set has a few interesting features, helping to give it higher code density than the Sparc, that are worth evaluating at this point.

Firstly, the ARM has load and store multiple instructions. These contain a 16-bit field that indicates which of the processor's 16 registers should be loaded or stored. This instruction increases code density and also increases performance on the ARM because the single memory port can operate sequentially during the transfer.

Measurements have found that the ARM typically transfers 46 registers to and from memory using 25 load, store, load multiple and store multiple instructions for each 100 instructions executed [27]. If no multiple instructions were present, these 46 register transfers would each need a separate load or store instruction. This would increase the ARM's code size by 21%.

As has been noted elsewhere, the ARM has only 16 registers and as a result needs to transfer more data to and from memory than the Sparc[1]. Note that the Sparc typically transfers around 23 registers to and from memory for each 100 instructions executed.

On average the ARM transfers about 1.8 registers per load, store, load multiple or store multiple instruction [27]. If the Sparc instruction set was extended to include load and store multiple and if it could exploit it as successfully as the ARM does, the number of load or store instructions would decrease by this same factor. This would mean a decrease in the relative total number of instructions n to 0.90, and an increase in relative code density d to 1.11.

In practice it is unlikely that the Sparc could exploit load and store multiple as well as the ARM can. The reasons include the following:

- On the ARM, because register 15 is the program counter, a load multiple instruction can be used to carry out a subroutine return that restores the necessary registers and returns control to the calling function.
- Since the Sparc has 32 registers, an ARM-like scheme of one bit per register to indicate which are to be transferred could not be used. Any alternative encoding would reduce the effectiveness of the instruction.

As was noted in section 3.3, the asynchronous implementation of load and store multiple instructions is not simple. This discourages their use in an asynchronous low-power processor.

1. The Sparc's register windows also reduce the frequency with which it executes load and store instructions.

The second interesting ARM feature is its conditional instructions. All ARM instructions have a condition code which is tested before the instruction is executed. This feature reduces the number of very short branches.

Measurements indicate that removing the condition codes from the instructions (except branches) would increase the total number of instructions required by around 9% [27]. If the same feature could be added to the Sparc processor with the same benefit, the relative total number of instructions n would be reduced to 0.92, and for a 4-bit condition field the relative average instruction length l would be increased to 1.125. The resulting relative code density d would be 0.97; code density would be decreased.

5.8 Conclusions

Table 5.6 summarises the potential code density increases that have been identified in this chapter.

Section	Feature	d
5.3	Remove unused fields	1.03
5.4.1	Fixed length 5-bit opcode field, assuming $r = 8$	1.08
5.4.2	Huffman encoded variable-length opcodes	1.25
5.5.1	10-bit branch displacements	1.07
5.5.2	1, 3 and 10-bit add and subtract instructions	1.04
5.6.1	2- and 3-address instructions	1.05
5.6.1	Explicit use of last result	1.05
5.6.1	Explicit generation and use of last result	1.08
5.6.2	16 registers	1.01
5.6.3	Workspace pointer register[a]	1.03
5.7	Load and Store multiple	1.11

Table 5.6: *Summary of code density increases*

a. If 50% of loads and stores are relative to this register.

The most beneficial changes are the use of a Huffman encoded opcode field, the load and store multiple instructions, last result re-use and shorter branch displacement fields.

The total increase in code density resulting from applying all of these changes cannot be found simply by multiplying the individual benefits together, as they are all interdependent. For example, whenever it is stated that removing x bits from $y\%$ of instructions leads to a relative average instruction length of l, it is assumed that the other $(100 - y)\ \%$ have length 32 bits. If this is not true, the value of l will be wrong.

To estimate the combined effect of the proposed changes, a calculation was carried out based on the following parameters:

- Instructions can have any length.
- Huffman-encoded opcode fields are used.
- 2-address and 3-address versions of all appropriate instructions are used.
- There are 32 registers.
- Immediate fields are generally 10 bits, except for call which is large and for the add and subtract instructions, where ± 1, 3-bit and 10-bit immediates are allowed.
- Explicit use of the result of the previous instruction without a register specifier.
- Generation of a result that is used only by the following instruction without a register specifier.
- Load and store multiple instructions[1].
- No unused fields

This composite architecture has a relative code density of 2.0; it is twice as dense as the Sparc instruction set.

Unfortunately this high code density comes at the cost of a large number of possible instruction lengths; this encoding has instructions of 14 different lengths, from 14 bits to 29 bits, and a far larger number of different formats of instruction. For a practical design, it is essential that the number of instruction formats can be simplified. The criteria are that the instruction lengths should be multiples of a basic size such as a byte, and that the position of fields within the instructions should be regular.

Consider the format shown in figure 5.7. This format uses instructions of length 16 bits or 32 bits. 32-bit instructions are made up from two halves with a similar format. The 16 bit instructions are divided into a 6-bit opcode field and either two 5-bit register specifier fields or a 10 bit immediate field.

1. It is simply assumed that each load or store instruction has 1.8 register specifier fields and transfers 1.8 registers.

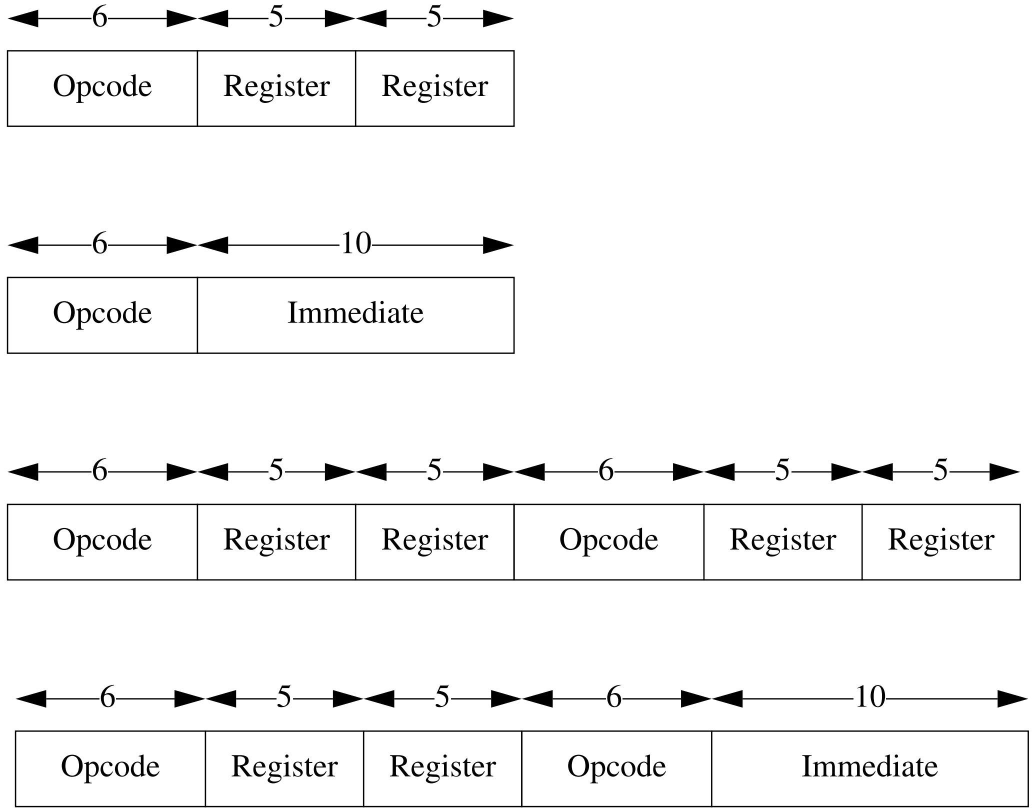


Figure 5.7: *A possible instruction format*

With this encoding, instructions such as branch, `A=A+B`, compare registers and `A=B` have relatively efficient encodings, whereas others such as load register+immediate and move immediate are a little too large for the 16 bit format yet waste space in the 32 bit format. Using this encoding, a relative code density of 1.6 is obtained.

The best way to improve on this encoding is to add an intermediate format of 24 bits. This is particularly useful for the move immediate instruction. However the alignment on byte boundaries means that implementation is more complicated.

So in conclusion the code density increase obtained depends on the flexibility of the instruction format. An upper limit of twice the density is possible, but would require very complex decoding within the processor. Using 16 or 32-bit instructions, a code density increase of 60% can be obtained. By using instructions whose length is a multiple of 8 bits, an increase between these two limits is likely.

The following final chapter reviews these results along with those from chapter 3.

Chapter 6 : Summary and Conclusions

This work has investigated the influence of processor architecture on power consumption on two fronts:

- Chapter 3 investigated architectural features that permit asynchronous implementation, which may lead to increased power efficiency.
- Chapter 5 investigated architectural features that lead to increased power efficiency through increased code density.

In the first sections of this final chapter, these architectural features are reviewed. In later sections, other influences on the choice of an architecture are considered, and finally possible areas for future work are mentioned.

6.1 Summary of Architectural Features

In this section, the architectural features proposed in chapters 3 and 5 are reviewed. The areas considered are the register file size and organisation, the branch mechanism, datapath operations and load and store operations. Figure 6.4 shows a possible block-level organisation of a processor including the features described in this section.

Register File

Section 5.6.2 found that a slight increase in code density could be made by decreasing the number of registers from the SPARC's 32. However it was noted that the power consumption of a load or store instruction is greater than that of other instructions because it involves two memory accesses; one to fetch the instruction and another to transfer the data. Other instructions access the memory only once.

With this broader perspective, the benefit of increasing or decreasing the number of registers can be reconsidered. Table 6.1 shows the relative power consumption for various different numbers of registers.

number of registers	relative code density d	relative power consumption[a]
8	1.02	1.06
16	1.01	1.03
32	1.00	1.00
64	0.99	0.98
128	0.97	0.96

Table 6.1: *Power efficiency and number of registers*

a. Assuming that load and store instructions consume twice as much power as any other instruction.

The result shown in this table is that increasing the number of registers will lead to a small increase in power efficiency; doubling from 32 to 64 registers saves 2% of the power.

It has also been suggested (section 3.1) that a larger number of registers can help the compiler to interleave more threads on an instruction-by-instruction basis, which avoids any delay associated with data dependencies.

On the other hand, an upper limit on the number of registers may be imposed by the compiler's ability to use them. In compiled C programs for example, it may be difficult to store variables in registers if they are array elements or if there are pointers to them. In dynamically bound languages, the system may be less able to apply interprocedural register allocation which again limits the number of registers that can be used. The compiler may also have difficulty in identifying which variables have the greatest frequency of access and hence are appropriate for register storage.

Weighing up these various aspects, 32 registers and 5-bit register specifiers may be the best compromise.

Not having a register file at all as the transputer does may lead to the highest code density (section 4.3.3); however the stack structure may become a bottleneck. Further investigation of alternatives such as this are worthwhile.

Branches

The following factors influence the choice of branch instructions:

- Branch instructions are common, so their density and their performance is important.
- Branch displacements are typically small; 10 bits is sufficient in most cases (section 5.5.1).
- For efficient asynchronous implementation, branches need to be decoded easily in a remote branch unit (section 3.5).
- The preferred branch condition mechanism for asynchronous implementation is the branch pipe; the results of compare instructions are directed to the branch unit through a pipeline stage.
- Conditional move instructions can reduce the number of short branches needed.

A possible set of 16-bit branch and conditional move instructions is shown in figure 6.1. In addition to these instructions, a computed branch instruction and an instruction to allow branches with longer displacements are required.

The computed branch instruction may be implemented using a pipe in a similar way to the branch condition pipe. Instructions must then be provided to put values into this pipe, such as move register to branch pipe or load to branch pipe, or alternatively one of the registers can be a pseudo-register, writing to which places a value into this pipe. The computed branch instruction itself then needs no register specifier or immediate field, and may be an 8-bit instruction.

The long branch instruction may be a 32 bit instruction, with as long an immediate field as is possible.

The action of the branch and link instruction is to write the old program counter value into a dedicated link register.

A halt instruction may also be implemented in the branch unit. When the halt instruction is executed, the processor will stop until an interrupt occurs. This instruction is useful for implementing a sleep mode.

Branch prediction leads to wasted power when the prediction is wrong; for optimum power efficiency, branch prediction should not be used and instructions should only be fetched once it is known that they will be executed. It is therefore not necessary to include branch hinting bits in the branch instruction. Not including these bits also improves code density.

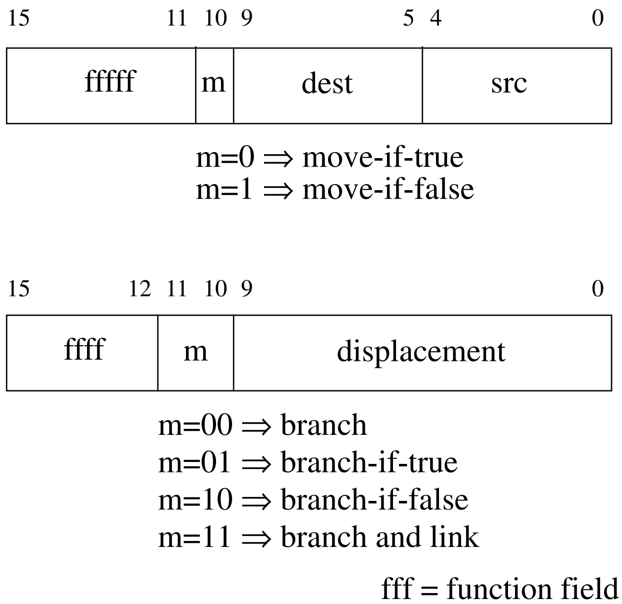


Figure 6.1: *Branch and conditional move instructions.*

Datapath Operations

There is little flexibility in the choice of the basic arithmetic and logical operations provided; AND, OR, EXOR, ADD, SUB etc. are essential.

Compare instructions are required, as described above in respect of conditional branch instructions. The result of the comparison is to place a true or false value into the branch condition pipe. Comparisons needed are therefore equal, less than and less than or equal, plus any signed/unsigned variants. The opposite condition is detected by using branch-if-false.

The width of the provided operations is important (section 4.1.4). If a 32-bit datapath is provided, for power efficiency it is essential to provide either a set of narrow operations, such as 8-bit and 16-bit addition, subtraction and move, or a set of instructions that process multiple narrow values simultaneously. The effectiveness of the latter would depend on the ability of the compiler to find pairs of operands that can be processed together, and may not work very well in general.

The inclusion of shift and multiply instructions must be justified by evaluating their frequency of occurrence and their implementation cost. The best solution may be to provide a subset of shift instructions, such as 8, 16 and 24 bit shifts for carrying out byte manipulations and 1,2,4,8 and 16 bit shifts which can be combined to give shifts by arbitrary amounts. In the case of multiplication, it is tempting to include a general-purpose multiply instruction simply because it is possible to implement a variable-latency multiplier asynchronously, but the benefit must be considered against the implementation cost. The frequency of shift and multiply operations is very dependent on the program in use.

The most common instructions should be given shorter formats making use of last result re-use and 2-address formats. Depending on the use of 2 or 3 operands and the use of the last result, these instructions may use between 0 and 3 register specifiers. The number of immediate lengths available should also be flexible. Some possible instructions and their encodings are shown in figure 6.2.

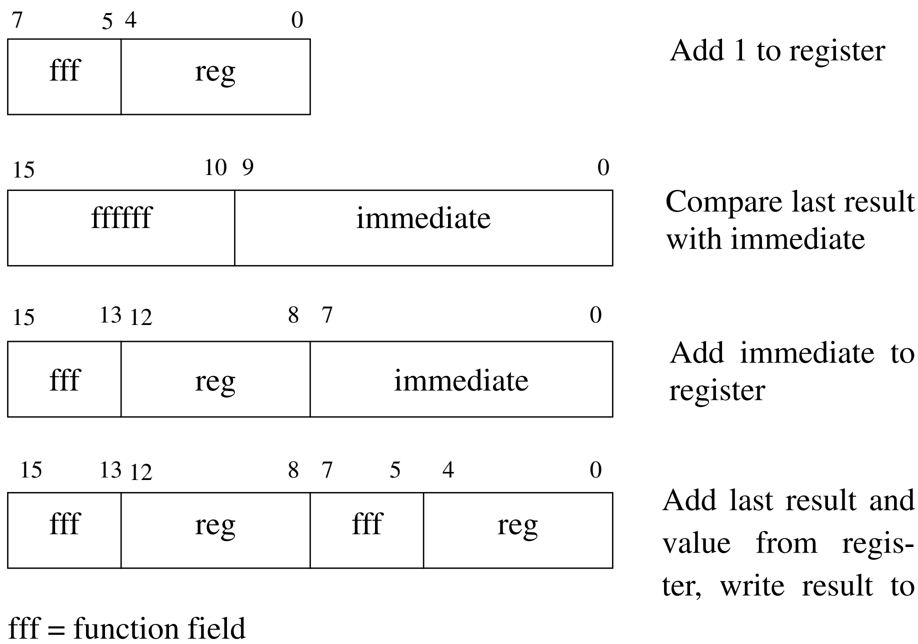


Figure 6.2: *Common instructions with variable-length encodings*

Less common instructions can be given in a simpler general format, more like the format used by SPARC and the other RISC processors, to ease decoding and to reduce the number of opcodes required (figure 6.3).

Signalling of overflow can be through imprecise exceptions and precise barrier instructions. Overflow detection may be enabled or disabled for each instruction. The less common case (overflow detection if the source program is in C) may use a longer instruction encoding.

31–25	24–20	19–15	14–10	9–0
fffffff	func	dest	src	immediate

31–25	24–20	19–15	14–10	9–5	4–0
fffffff	func	dest	src1	-	src2

fff = function field

Figure 6.3: *Regular instruction formats (32-bit) for less common operations*

In summary, the datapath instructions may be divided into two groups: the more common ones where the use of variations such as two-address formats is justified, and the less common ones where a simpler general format is used. In the former category are instructions such as add, subtract, compare and move. In the latter category are instructions such as the logical operations, shifts and multiply. The variations possible for the common instructions include the following:

- With and without causing an exception on overflow.
- 8, 16 and 32 bit operands.
- Use the result of the last operation as one operand.
- Send the result to only the following instruction.
- A variety of immediate field sizes.
- 2-address format.

Memory Operations

Load and store multiple are bad for asynchronous implementation (section 3.3) but are good for code density (section 5.7). It may be worth implementing a simple form of load and store multiple; for example prohibiting loading over the base register or loading to the PC would make dealing with an exception during the transfer easier. The way in which the registers to be transferred are specified is important, and depends on how many registers are transferred by each instruction. One solution is to specify a first and last register field and transfer a contiguous sequence of registers.

Transferring 8- and 16-bit values using 32-bit operations is wasteful of power, and so 8 and 16 bit load and store instructions should be provided.

It has been noted that the use of parallelism can improve power efficiency. To support shared memory multiprocessing it is necessary to provide some form of semaphore instruction such as swap, load-locked and store-conditional, or load-and-increment [7]. These instructions involve an atomic read-modify-write cycle which is not very suitable for an asynchronous implementation as it may disrupt the pipeline flow. The load-locked and store-conditional operations[1] on the other hand do not involve atomic read-modify-write, and so may be more suitable. This approach may also lead to less unnecessary overhead in a non-multiprocessing system.

Non Instruction-Set Issues

The organisation of caches is not related to the instruction set. However it does have a significant impact on the power efficiency.

The idea of employing multiple levels of cache, described in section 4.1.2, is attractive because it leads to increased power efficiency and because the variable response time can be dealt with easily in an asynchronous environment.

The data caches should also be organised as copy-back caches, to reduce the frequency of writes to the main memory.

6.2 Quantitative Evaluation of the Proposed Architecture

The architecture described in the proceeding sections is intended to have greater power efficiency than conventional architecture. This section briefly evaluates the possible relative power efficiency and also the relative performance.

1. Load-locked performs a normal load, but tells the memory system to try and lock the location read from. Store conditional performs a normal store, except in the case when the lock has been broken by a load-locked from another processor, in which case the store fails.

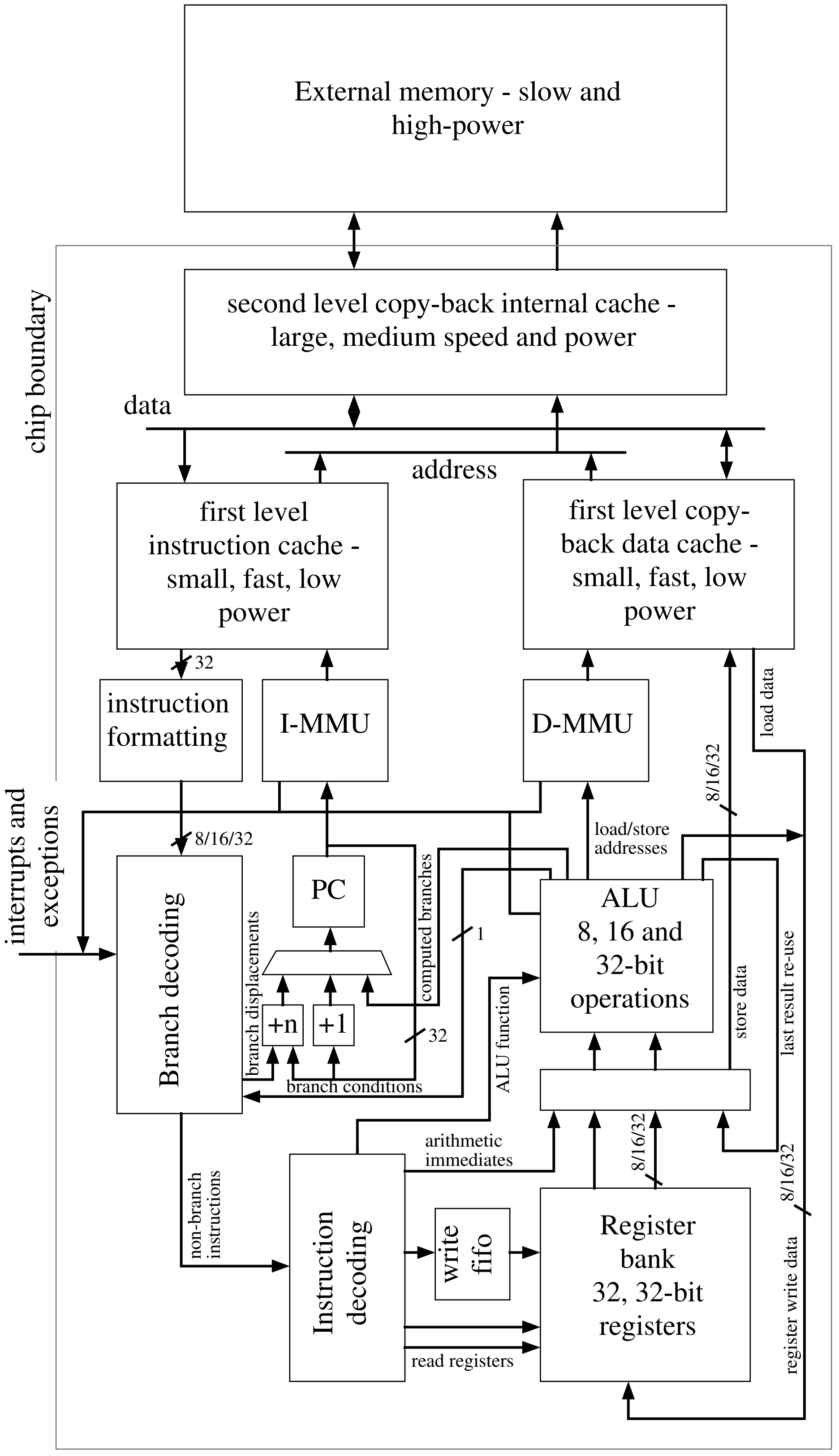


Figure 6.4: *Processor Organisation*

6.2.1 Power efficiency

External memory

The number of external memory accesses required per unit of computational work is reduced as a result of the following factors:

- The direct result of increased code density. Based on the results from section 5.8, the code density of the architecture should be between 50% and 100% greater than a conventional RISC processor such as the SPARC. If 80% of the memory accesses are instruction fetches, this will directly reduce the power consumed in the external memory to 60% -75%.
- The effect of the increased code density on the performance of the internal cache. Doubling the code density may be equivalent to doubling the cache size. Data [7] suggests that by doubling the size of a cache from the miss ratio may be reduced to around 70% - 80%[1]. This means that by doubling the code density, the number of external memory accesses may be reduced to 70% - 80%.
- Compared to a processor with a similar sized internal cache using a write-through cache organisation or a write buffer, this architecture reduces the power consumed in the main memory because fewer writes take place. If 90% of writes are not passed to the main memory, if 10% of cache accesses are writes, and if the caches have a hit rate of 90%, this will reduce the total number of external memory accesses to 55%.

The combined effect of these factors may be to reduce the power consumed in the external memory to 25% - 35% of that of a processor without these features.

Caches

If the multiple levels of caches work well, the power consumed by the caches should approach that of a single cache of the size of the first level cache whilst retaining the overall hit ratio of a single cache of the size of the second level cache. If for example the first level cache makes up a quarter of the total cache, the power consumed by the caches should be reduced to one quarter of the power consumed by a single level cache. In practice the benefit would be smaller, but it may be possible to halve the power consumed in the caches in this way.

The increased code density will also lead to reduced power consumption in the cache, by the same factor of 60% - 75% that applies to the main memory. The total power con-

1. For a 1-way or 2-way set-associative cache, of between 1kbytes and 32 kbytes.

sumed in the caches may then be reduced to as little as 30% of that of a comparable processor without these features.

Datapath

The introduction of instructions to operate on 8 and 16 bit data as well as 32 bit data will lead to a reduction of the power consumed in the ALU and register file. The magnitude of this saving is hard to quantify as the proportion of instructions that could use these narrow operations is not known.

Use of Asynchronous Logic

The contribution to power efficiency from the use of asynchronous logic is not known. Hopefully the power consumption of the AMULET1 processor will soon be evaluated in sufficient detail to derive some useful conclusion. The most important aspect of its contribution is likely to be the power saving that occurs during intervals of inactivity. The magnitude of this effect is highly dependent on the processor's workload.

Considering all of these factors, it is possible that the total overall power consumption of the processor may be reduced to less than half. Referring to table 1.2, this is equivalent to the likely increase in power consumption over 2 to 4 years.

6.2.2 Performance

This work has reversed the importance of power efficiency and performance compared with previous architectural studies. Previously performance has been considered as the most important aspect of an architecture. There is a trade-off between the two factors, and since this architecture has increased power efficiency it is likely that it will suffer from reduced performance.

The performance of the processor depends on the speed of the asynchronous logic used and on the effect of the various instruction set features proposed. The speed of the asynchronous logic is not well understood, though this will be improved when the AMULET1 processor has been fully evaluated. It is hoped that the performance of asynchronous logic will be at least as good as that of synchronous logic for suitable designs. The performance of this architecture should be at least as good as that of the AMULET1 processor.

The performance may be limited slightly by the increased instruction decoding complexity. However with an asynchronous implementation of the decode logic the more common instructions can be decoded more quickly, so the penalty of the complex instruction encoding should be less than for a synchronous implementation.

The performance will also be influenced by the pipeline dependencies, such as the case where a comparison is followed by a branch or a load operation is followed by an instruction that uses the loaded data, and by the absence of branch prediction. The importance of these effects may be reduced slightly through the use of an optimised compiler.

6.3 Other Considerations for Architectures

To complete an architectural specification, a great many factors that have little or no influence on power efficiency must be taken into consideration.

If it is successful, a processor architecture can exist in use for many years. The DEC Alpha architecture [26] for example has a 25-year projected lifetime. Others are more conservative with their claims, but experience with the 8086 and 68000 architectures suggests that lifetimes of above 10 years are not unreasonable[1].

Although the architecture will remain frozen (or at best ‘backward compatible’) over this lifetime, the compiler technology available will continue to advance. It is therefore important to design the original architecture with an awareness of possible future advances in compiler technology.

It is also likely that over time the type of programs being used and the programming languages in which they are written will change. A good architecture will incorporate features that give a positive benefit to all programs, not just those on which it was evaluated. The RISC principle of providing primitive instructions from which the compiler can construct programs rather than complex instructions targeted at particular language structures leads to such features.

6.4 Conclusions

For power efficiency and asynchronous implementation, an architecture should include various features including the following:

- To give a high code density: variable instruction lengths, variable and short immediate field lengths, explicit last result re-use, 2-address instructions, load and store multiple.
- Narrow (i.e. 8 and 16-bit as well as 32-bit) datapath operations and load and store instructions.

1. This situation could change dramatically if systems became less dependent on binary code compatibility than they are now.

- A simple branch mechanism, using a branch condition pipe for conditional branches.
- Multi-level copy-back caches.

By using these features, it should be possible to at least double power efficiency.

6.5 Future Work

It is hoped that a detailed architectural specification for an asynchronous power-efficient processor can be drawn up and an implementation made.

This process is likely to be iterative. In particular, the only basis on which to judge the suitability of an architectural feature for asynchronous implementation to date has been experience with the AMULET1 processor. It is probable that some features will prove more or less suitable for implementation during the implementation process, and revised ideas will be used.

Ultimately it is hoped that a processor with significantly high power efficiency can be demonstrated.

It is possible that the most appropriate architecture for power-efficient asynchronous implementation is a totally novel architecture unrelated to the variations on conventional themes considered here. Some ideas have occurred during the work for this thesis which deserve more detailed investigation. Elements of these ideas may be incorporated into the final design.

Appendix A : Patterns of Access to Data Operands

At several times during the architectural studies the question of the pattern of accesses to data items arose. Questions to be answered included, “How frequently are data values accessed? What is the distribution of these frequencies like? How many storage locations contain values being accessed with this frequency? How does this behaviour vary between programs?”

Some examples of situations in which this data was required include the following:

- For the analysis of the benefit of last result forwarding it was necessary to find the proportion of data accesses that use the result of the previous instruction.
- To find the optimum number of registers for the architecture the number of storage locations whose access frequency is less than some threshold is needed.
- Similarly, to find the ideal size for a data cache.

This appendix proposes a model for the pattern of access to data operands, from which answers to these questions can be derived.

Two approaches to the problem were considered, a theoretical approach and an experimental approach. The former was to build a model for program behaviour that would predict the pattern of accesses. The latter was to measure the access patterns of real programs and to find an approximation to these results.

The experimental work produced a simple model for the behaviour which is explained in section A.2. Firstly however the theoretical approach is reviewed.

A.1 Theoretical Models of Program Behaviour

Previous work by Denning, Spirn and others has investigated program behaviour for the purpose of evaluating page replacement strategies in demand-paged virtual memory systems [32] [33]. This work shows how program behaviour can be approximated to by statistical models with constants derived from the study of real programs, and by other techniques. One result is Belady's Lifetime Function, which relates the frequency of paging to the amount of available memory.

The frequency of paging in a virtual memory system is orders of magnitude lower than the frequency of access to registers, and it is not clear that this previous work scales across all types of memory accesses.

More recent work has applied the concept of fractal geometry to the study of program behaviour [34] [35]. This work models the pattern of instruction fetch accesses of the program as a fractal random walk. Because of the well-known property of fractals that the observed behaviour is the same at any level of magnification, this model can be applied across a range of memory types from page store to cache memory. The work was intended for the generation of synthetic address traces for cache simulation and the results produced by the fractal algorithm are similar to real traces in many important respects.

Unfortunately none of this work concentrates on the issue of data accesses. The behaviour of data accesses varies in important ways from the general pattern of memory accesses, which is dominated by instruction fetch activity. Instruction fetch is essentially sequential and interrupted by branches. Any sequential pattern in data references is less important.

These results are therefore not directly relevant to the questions of interest.

In the absence of any inspiration for a more appropriate model, the experimental approach was considered.

A.2 Experimental Studies of Program Behaviour

The basis of the experimental work is as follows:

Data accesses are considered in terms of lives. A life is a value in a storage location. A life starts when a value is written to a location and ends when it is read for the last time before the next write to that location. The life may be accessed more times between the initial write and the final read.

A life is neither a variable nor a storage location. Variables store many lives and storage locations store many variables over the execution time of a program.

In a processor with a load-store architecture lives exist in the registers and in the memory. In the memory lives are started by store operations and other accesses occur through load operations. In the registers lives are started by writing the results of instructions and are accessed as operands to other instructions.

Two experiments were therefore conducted. In the first, the access patterns of benchmark programs to the processor registers were studied. In the second, the access patterns to data memory by load and store instructions were studied.

The results of these experiments showed that the behaviour of all the benchmark programs closely follows a well-defined pattern.

The experiments conducted and their results are now presented.

Register Operand Access Patterns

A simulator for the Sparc architecture, Shade [36], was modified to record each register read and write operation. For each register a record was kept indicating the time since the most recent write to and most recent read from that register and the number of reads that have occurred since that write. Each time a register write occurred the lifetime of the life (that is the time from the first write to the last read) and the number of events that occurred during the life were noted. At the end of the simulation these statistics were output.

The simulator was used to run a number of benchmark programs. The programs used are described in appendix C. The raw output data from the simulator consists of the distribution of the lives with respect to the lifetime and number of accesses.

Memory Operand Access Patterns

Memory access traces intended for cache simulation for various benchmark programs were obtained. The programs used are described in appendix C. These traces were processed to extract only the data accesses and to discard the instruction fetches. This data was then analysed by a program that noted the times and count of accesses to each memory location[1] in a similar fashion to the register access analysis program. Once again the output of the program consists of the distribution of the lives with respect to the lifetime and the number of accesses.

Analysis of the Results

The objective of the analysis is to answer questions such as “How many storage locations are accessed more than once every 1000 instructions?”. Let the total number of

1. Because of the large size of the memory space a hashing technique was used to study only a subset of the total memory accesses.

storage locations that are accessed with frequency f be $a(f)$. If $a(f)$ is known, then questions such as that posed can be answered by integration; for example:

$$\text{Number of locations storing values that are accessed more than once every 1000 instructions} = \int_{\frac{1}{1000}}^{\infty} a(f)df$$

The problem is to derive an approximation to the $a(f)$ function from the experimental data.

Let the total number of lives with n accesses and lifetime l be $d(n, l)$. This is the distribution recorded by the experiments.

The frequency of access to a life f is given by $f = \frac{n}{l}$. Using this a new distribution $d(f, l)$ can be derived from the experimental results, giving the number of lives during the execution of the program with access frequency f and lifetime l.

Let the total duration of the program be p. p is also recorded in the experiments.

At any instant during the execution of the program, the probability that a particular life with lifetime l is alive is $\frac{l}{p}$, since this is the fraction of the total time for which the life is alive. It follows that the total number of lives alive at any instant with lifetime l and access frequency f is $d(f, l)\frac{l}{p}$.

From this, $a(f)$ can be derived by summation over l:

$$a(f) = \sum_{l} d(f, l)\frac{l}{p} \tag{1}$$

This operation was applied to the experimental data to obtain values for $a(f)$.

To see the characteristics of this data, it was plotted as shown in figures A.1 and A.2. The following points should be noted:

- The graphs show cumulative $a(f)$ on the y-axis against $\frac{1}{f}$ on the x-axis.
- A cumulative y-axis is used because when the discrete n and l values are divided to obtain f a ‘spiky’ distribution concentrated at the common fractions would occur.

- The x-axis represents the period of the accesses, rather than the frequency, i.e. it shows the reciprocal.
- The axes are logarithmic.

The last two points were found to give graphs on which the trend could be seen most easily.

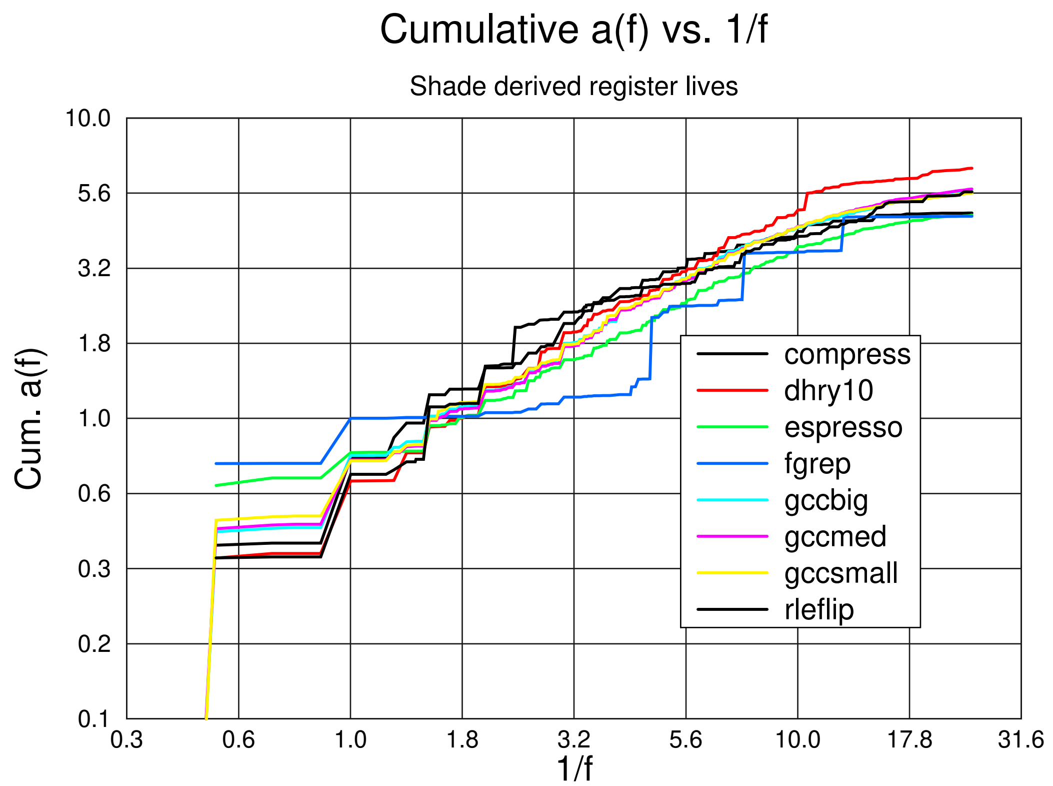


Figure A.1: *a(f) distribution from register-access analysis*

The following observations can be made from the graphs shown in figures A.1 and A.2:

- The behaviours of the programs are very similar. There is no great deviation from the centre-line by any of the benchmarks, except for the memory accesses of the tex program[1].

1. This discrepancy is attributed to the behaviour of the particular period of execution that was studied, which contained many iterations of a loop of a fixed size. It was found that the length of this loop corresponds to the step seen at $\frac{1}{f} \approx 30$.

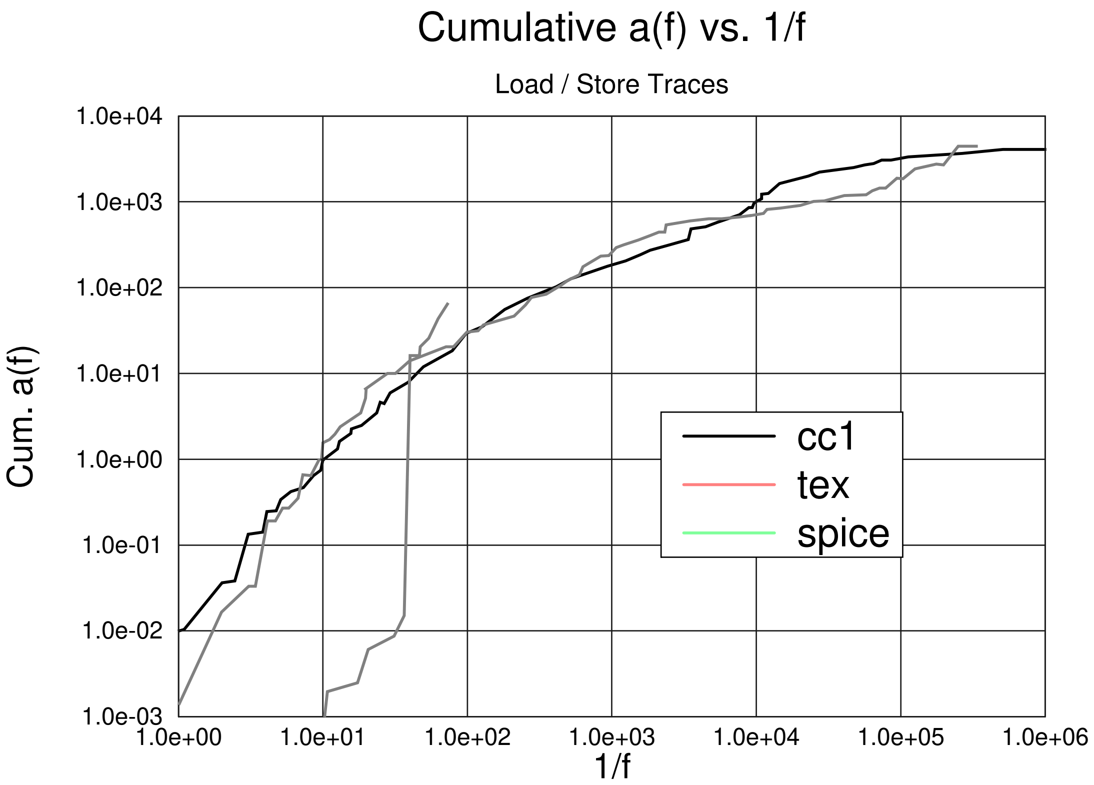


Figure A.2: *a(f) distribution from memory access analysis*

- The register accesses have relatively high frequencies and the memory accesses have relatively low frequencies.

- The highest access frequency obtained is 2 accesses per instruction. This corresponds to instructions such as `ADD R1,R1,#3`.

- The register access graph can be approximated to by a straight line on these axes up to an access frequency of around 1 access every 10 instructions, where it starts to tail off.

- The number of memory locations storing lives accessed more often than about once every 30 instructions is negligible.

- The memory access graph can be expected to finish around $\frac{1}{f} = 10^5 \ldots 10^6$ because of the finite length of the data input.

The relationship between the register access patterns and the memory access patterns can be seen in figure A.3, which shows the two sets of results superimposed.

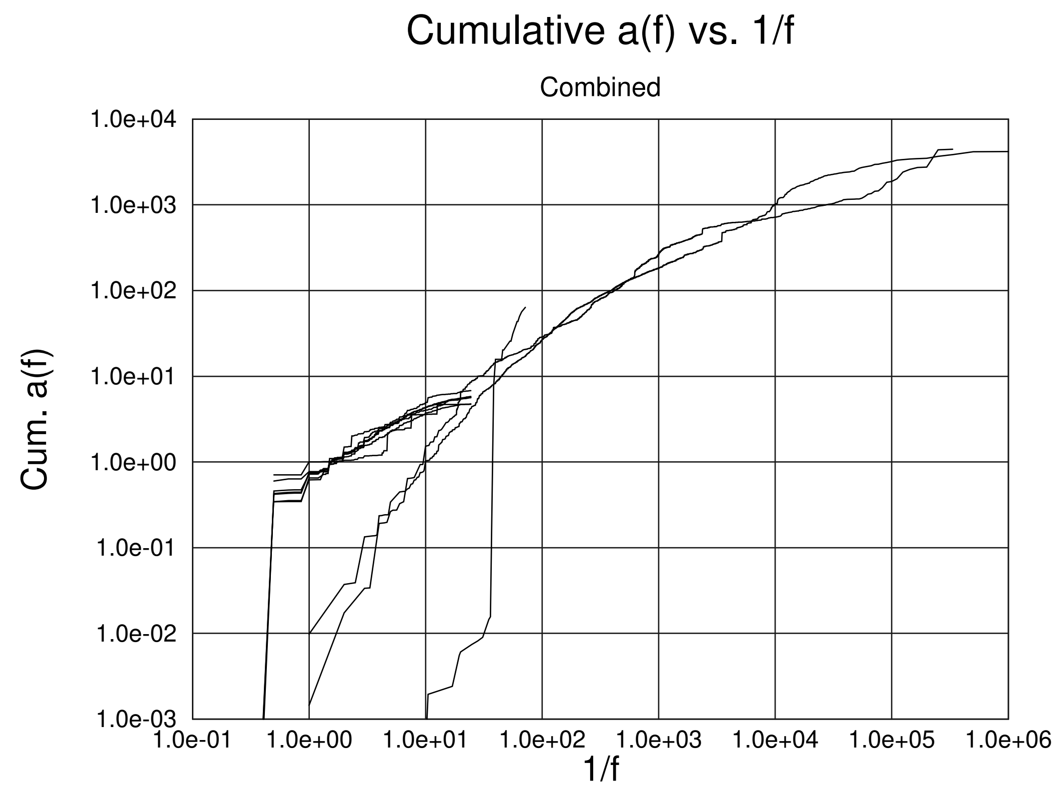


Figure A.3: *Combined a(f) distributions from both analyses*

From this figure it is clear that the memory and register accesses are unified by a single pattern of accesses. Above a threshold of approximately one access in every 20 instructions, memory is used. Below this threshold, registers are used.

On these axes, a straight-line approximation is possible for $\frac{1}{2} \leq \frac{1}{f} \leq 10^5$. Remembering that the axes are unusual the straight line indicates that a relation of the following form holds:

$$\int_0^{\frac{1}{f}} a\left(\frac{1}{x}\right) d\frac{1}{x} = b\left(\frac{1}{f}\right)^c$$

where b and c are constants measured from the graph to be 0.60 and 0.81 respectively.

It is possible to derive the following expression for $a(f)$:

$$a(f) = \begin{cases} 0 & f > 2 \\ b\,(c+1)\left(\dfrac{1}{f}\right)^{c+1} & f \le 2 \end{cases} \tag{2}$$

However it may be more useful to retain the integrated form:

$$\int_{f_{\min}}^{f_{\max}} a(x)dx = b\left(\left(\frac{1}{f_{\min}}\right)^{c} - \left(\frac{1}{f_{\max}}\right)^{c}\right) \qquad \begin{matrix} f_{\max} \le 2 \\ f_{\min} \le 2 \end{matrix} \tag{3}$$

Using (3) it is possible to answer the question set out as an objective, “How many storage locations are accessed more than once every 1000 instructions?”. The answer is:

$$b\left((1000)^{c} - \left(\frac{1}{2}\right)^{c}\right) = 161$$

Appendix B : Analysis of the Code Density of the Sparc Processor

Chapter 5 considers the potential for increasing the power efficiency of a processor by means of increasing its code density. This appendix presents experimental results to support the arguments made in that chapter.

The experimental work was carried out on a Sparc processor. Various benchmark programs were executed on a simulator which was modified to record statistics of interest during the execution. The simulator used was shade [36], and the benchmark programs used are described in appendix C.

The statistics that were measured include the following:

- The relative frequency of occurrence of each instruction.
- The frequency of occurrence of last result re-use and which instructions are most likely to use it.
- The distribution of immediate values.
- The proportion of instructions where one source register is equal to the destination register.

These results are presented in the following sections.

B.1 Instruction Frequencies

Figure B.1 shows the relative frequency of each of the most frequent instructions. The instruction with the highest frequency, the conditional branch, is sub-divided into each possible condition in figure B.2.

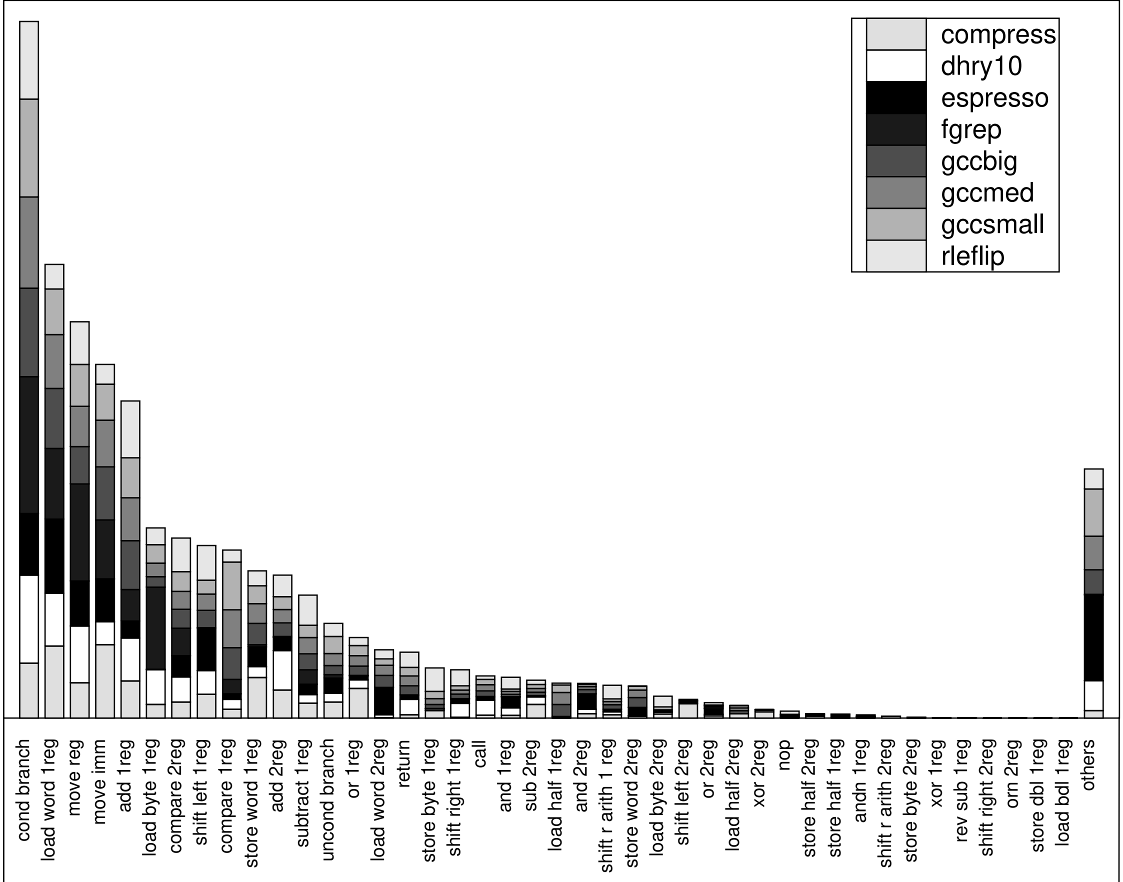


Figure B.1: *Relative instruction execution frequencies*

B.2 Frequency of Last Result Re-use

When an instruction writes its result into a register and the immediately following register reads from the same register, last result re-use has occurred. The frequency with which this occurs and the instructions that act as source and destination are important for the evaluation of forwarding and other architectural features. Table B.1 shows the frequency with which last result re-use occurred in the various benchmark programs.

The first two data columns in table B.1 show the frequency of last result re-use inclusive and exclusive of the condition codes. Re-use of the condition codes is very common as comparison instructions are often followed by conditional branch instructions. The second data column counts only re-use involving a general register.

The third data column measures the occurrence of last result re-use when the value that is re-used is never accessed again; that is the result of one instruction is used as an operand by the next instruction and then discarded.

Note that these results are consistent with the data shown in figure A.1.

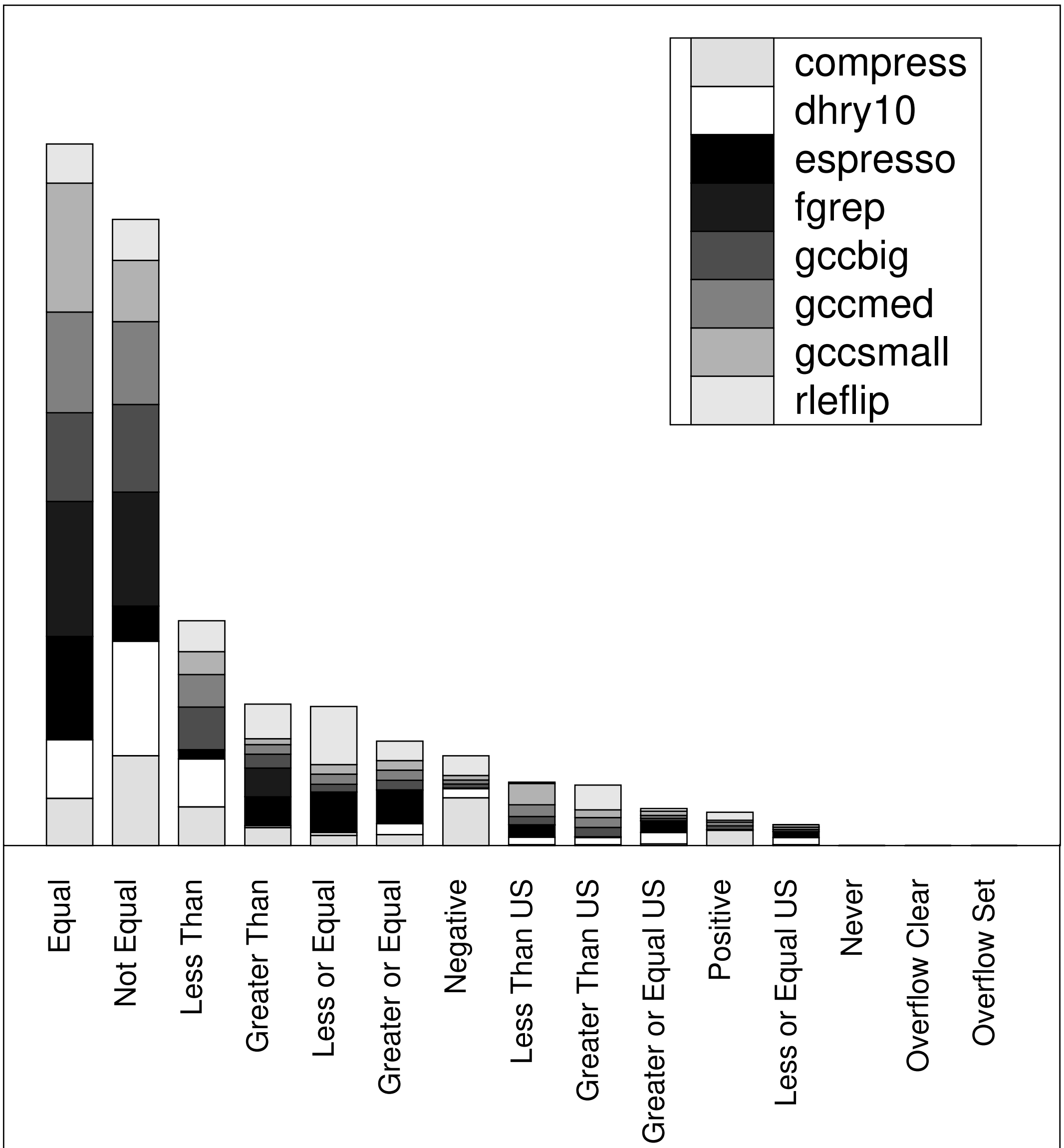


Figure B.2: *Execution frequencies for conditional branches*

B.3 The Distribution of Immediate Operands

Sparc instructions have immediate fields of length 5, 13, 22 and 30 bits. Most arithmetic and logical operations and loads and stores use 13 bit fields. Shift instructions use 5 bit fields and branch and call instructions use 22 and 30 bit fields respectively.

Often the most significant bits of these fields are unused. The most common operands for many of these operations could fit into smaller fields. This is of interest because using shorter fields would lead to increased code density.

To investigate the distribution of these immediate values, the Shade simulator was used to record the values of the immediate fields during the execution of the benchmark pro-

Program	Proportion of instructions that use the last result		
	Including condition codes	Excluding condition codes	Excluding condition codes Never used again
ccreg	0.445	0.300	0.187
compress	0.453	0.361	0.215
dhry10	0.414	0.265	0.155
espresso	0.487	0.375	0.316
fgrep	0.538	0.287	0.252
gccbig	0.422	0.284	0.198
gccmed	0.418	0.269	0.186
gccsmall	0.408	0.239	0.161
rleflip	0.353	0.264	0.180
Mean	0.438	0.294	0.206

Table B.1: *Last result re-use*

grams. The results are presented as cumulative graphs for each instruction measured in figures B.3 to B.14.

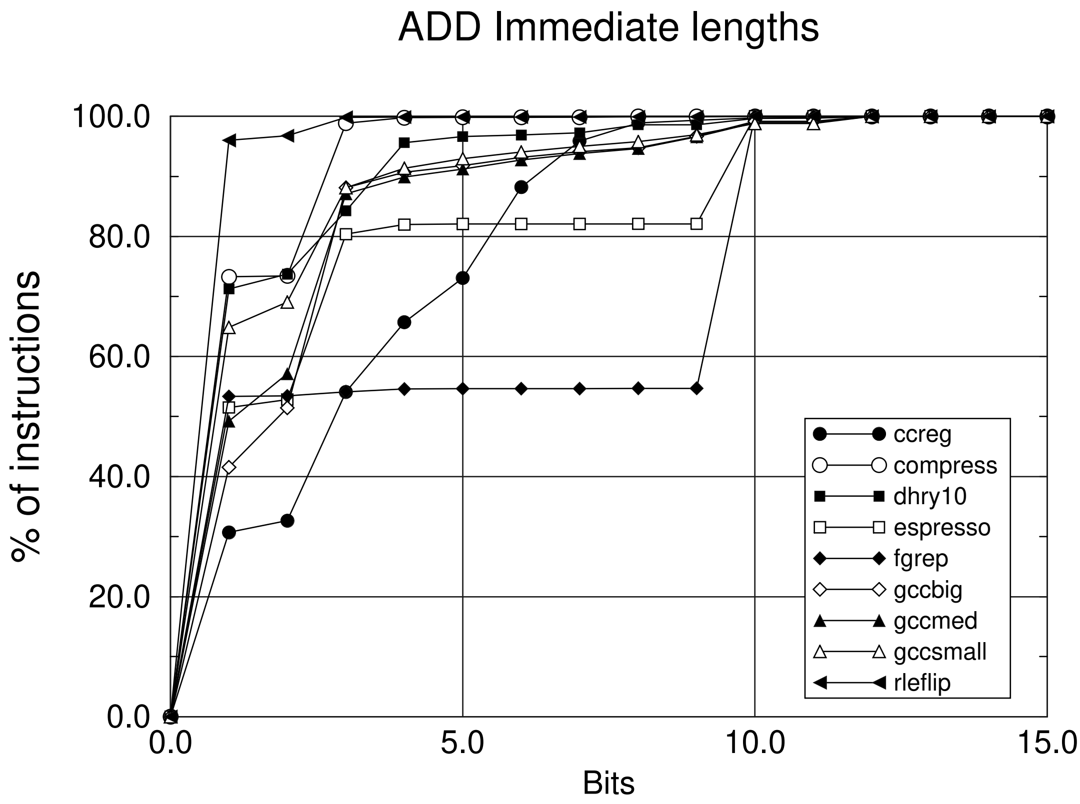


Figure B.3: *Add immediate lengths*

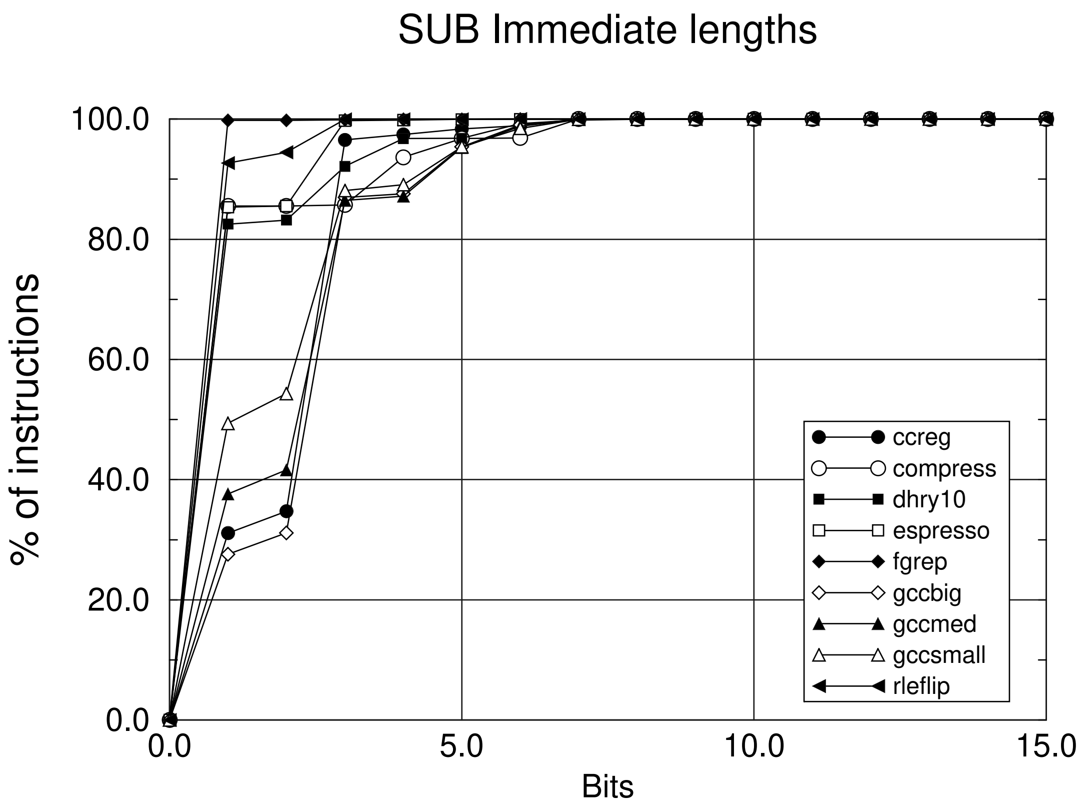


Figure B.4: *Subtract immediate lengths*

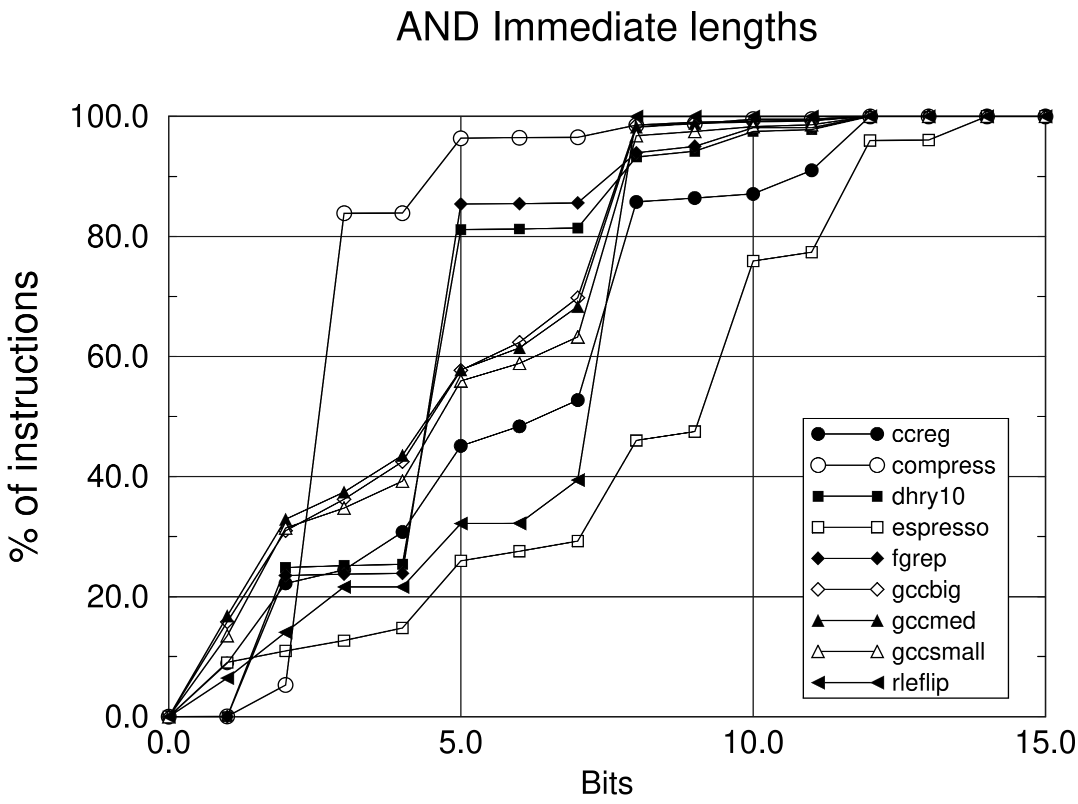


Figure B.5: *And immediate lengths*

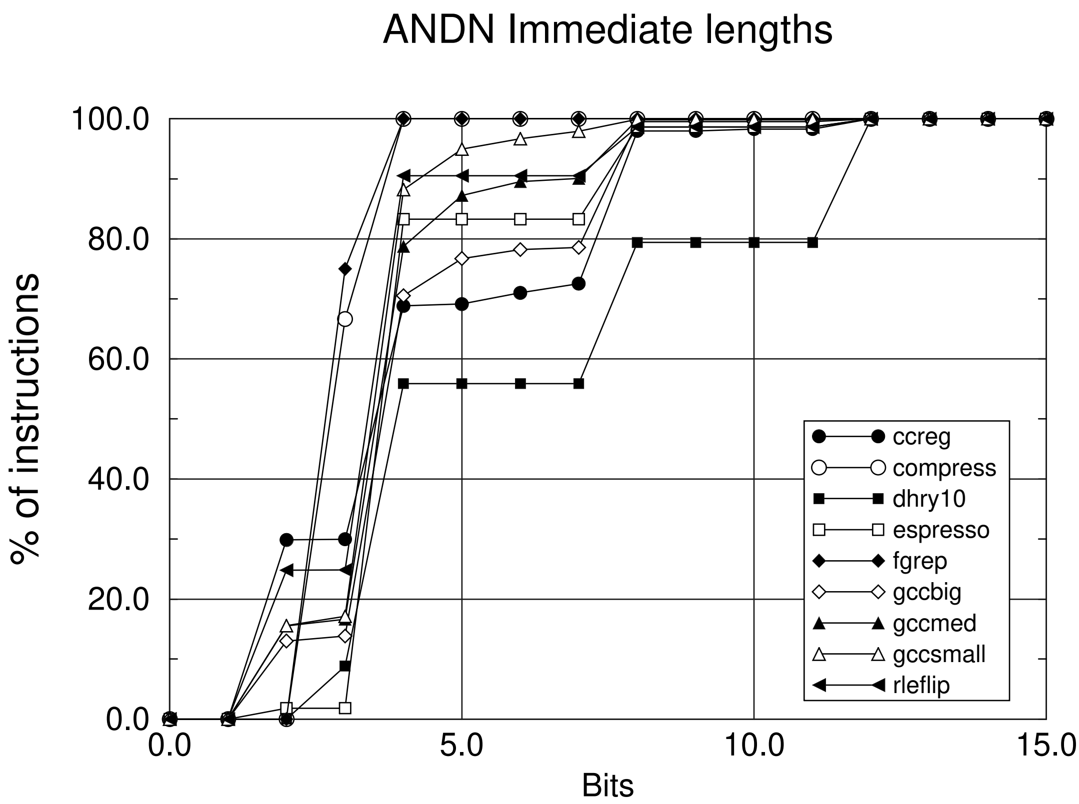


Figure B.6: *And Not immediate lengths*

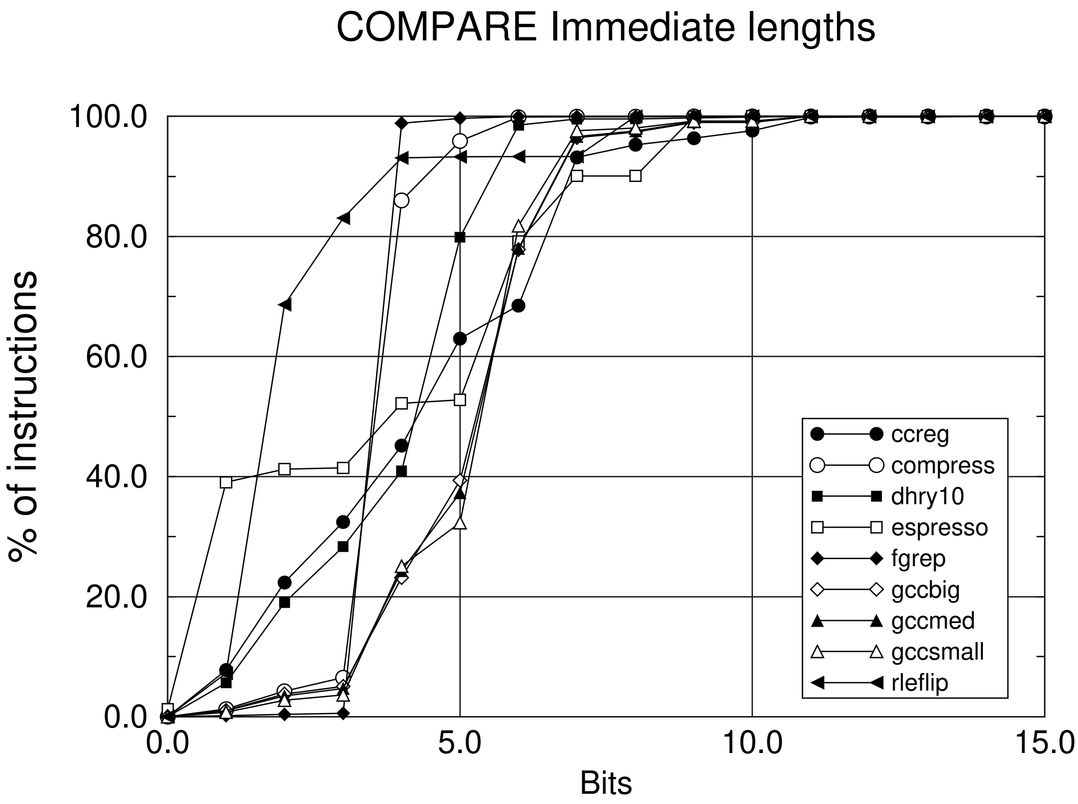


Figure B.7: *Compare immediate lengths*

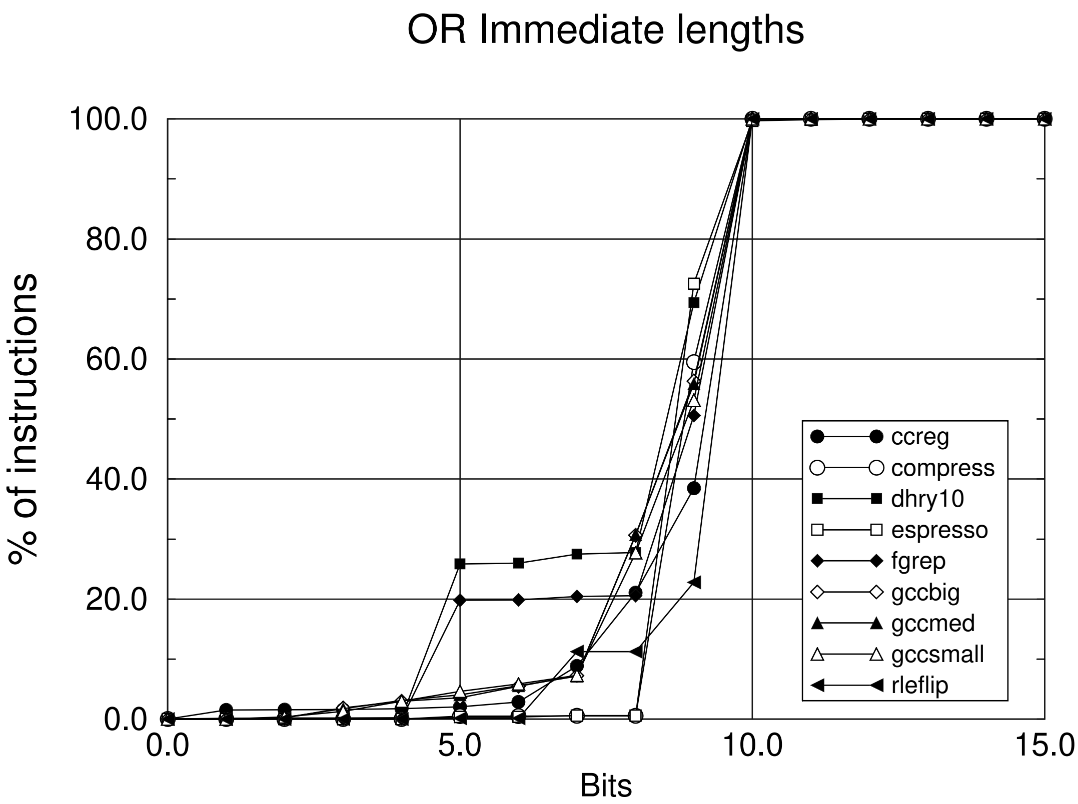


Figure B.8: *Or immediate lengths*

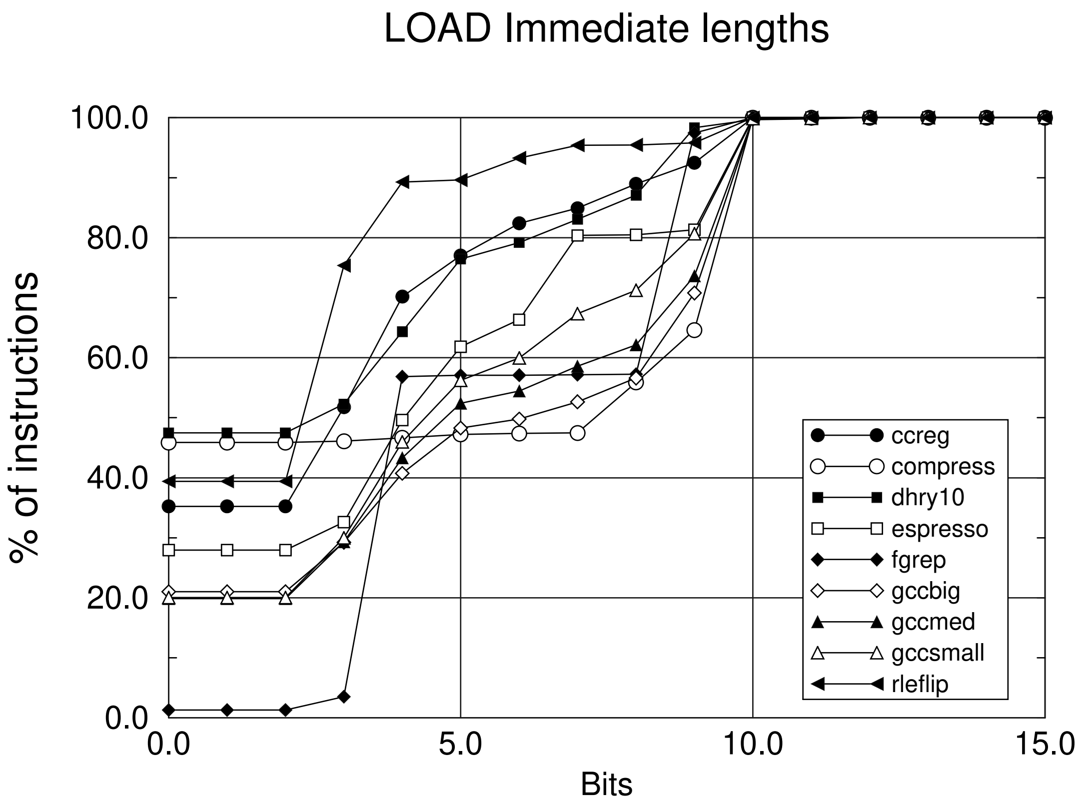


Figure B.9: *Load immediate lengths*

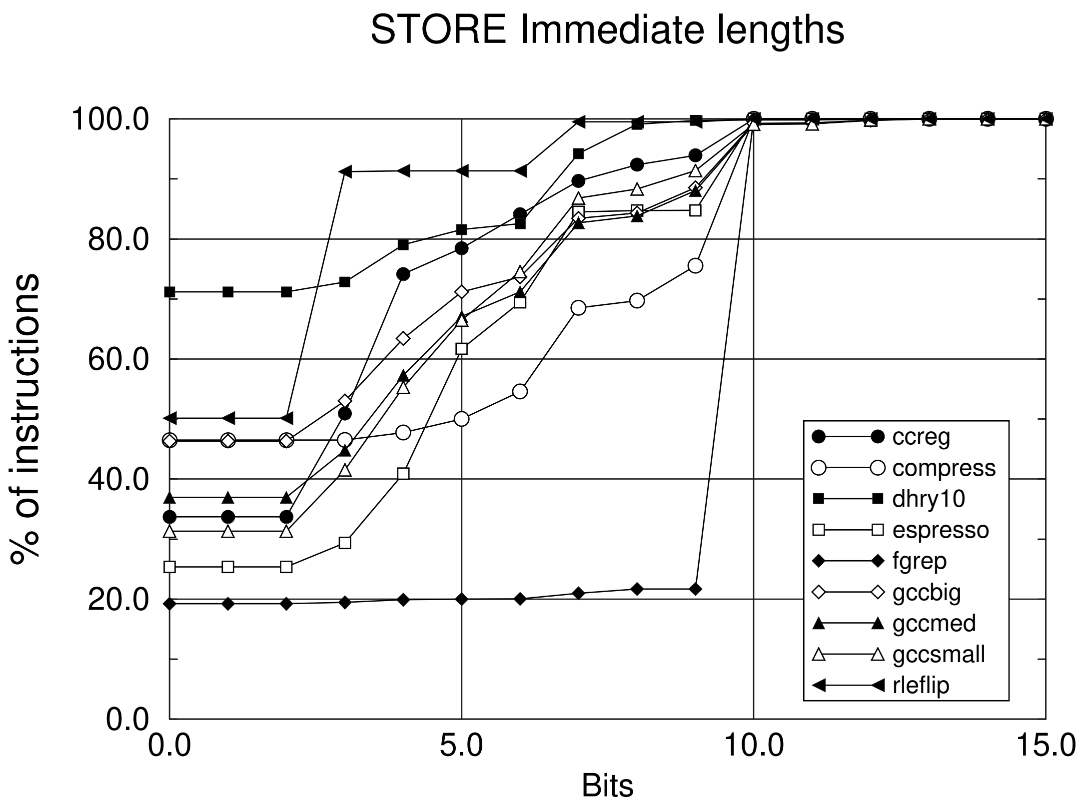


Figure B.10: *Store immediate lengths*

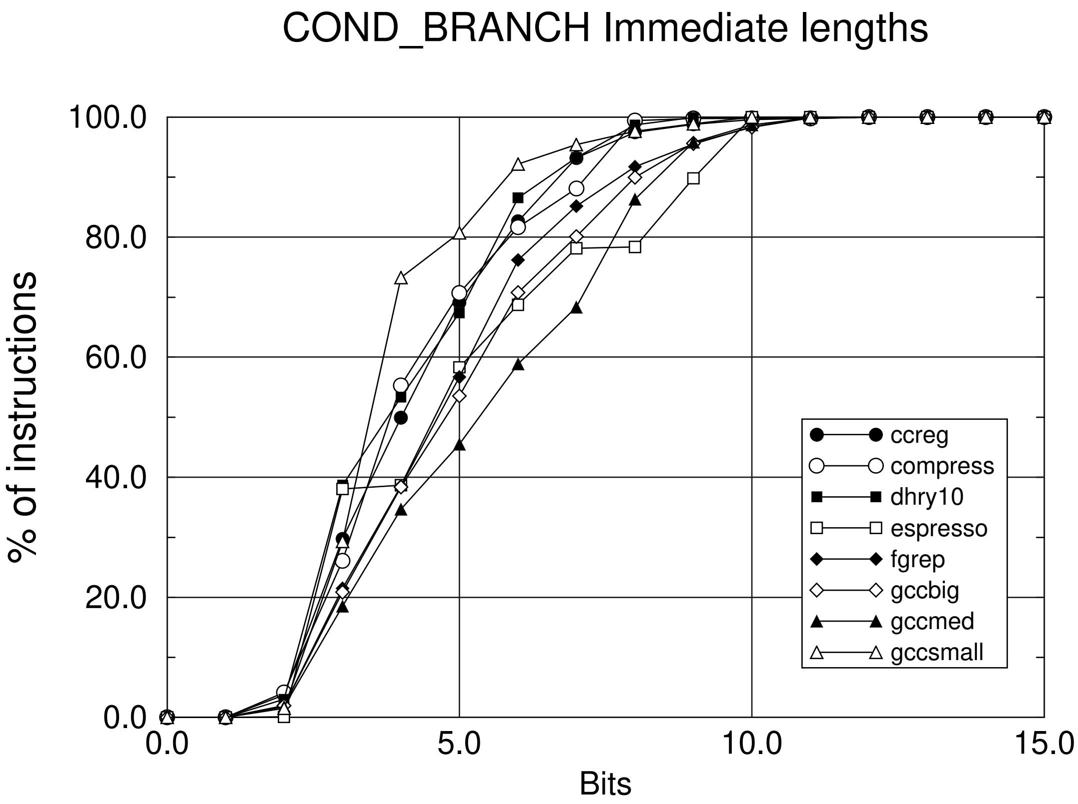


Figure B.11: *Conditional Branch immediate lengths*

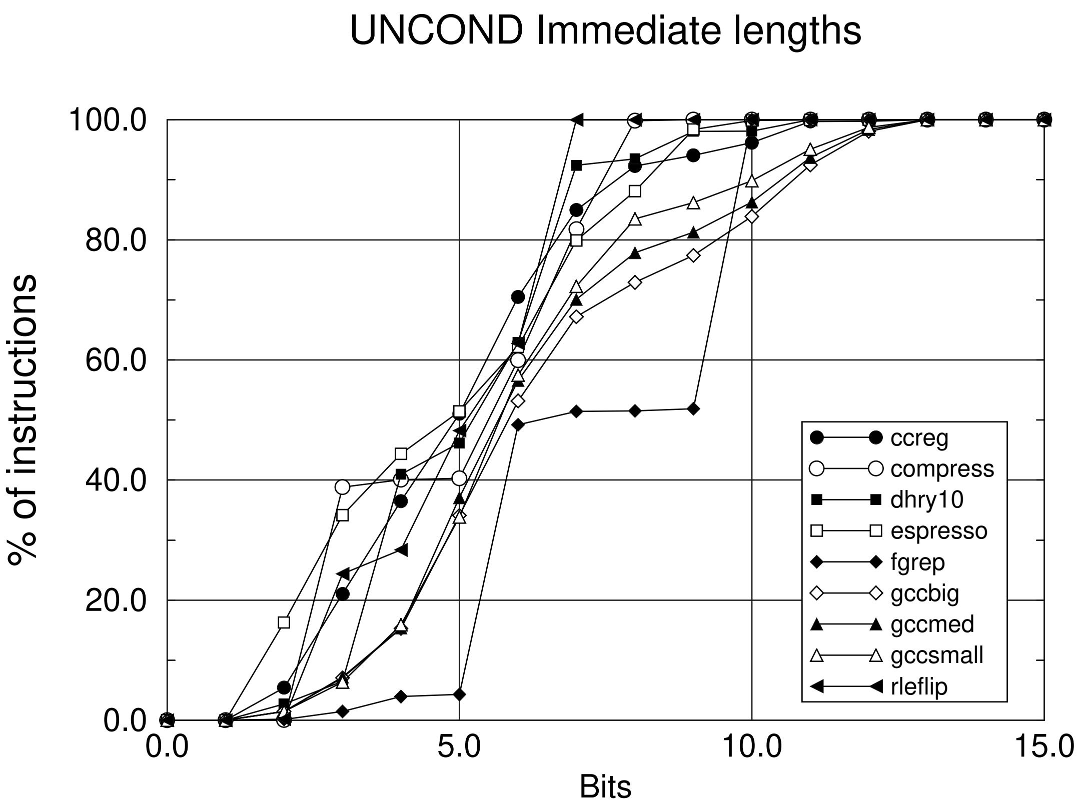


Figure B.12: *Unconditional Branch immediate lengths*

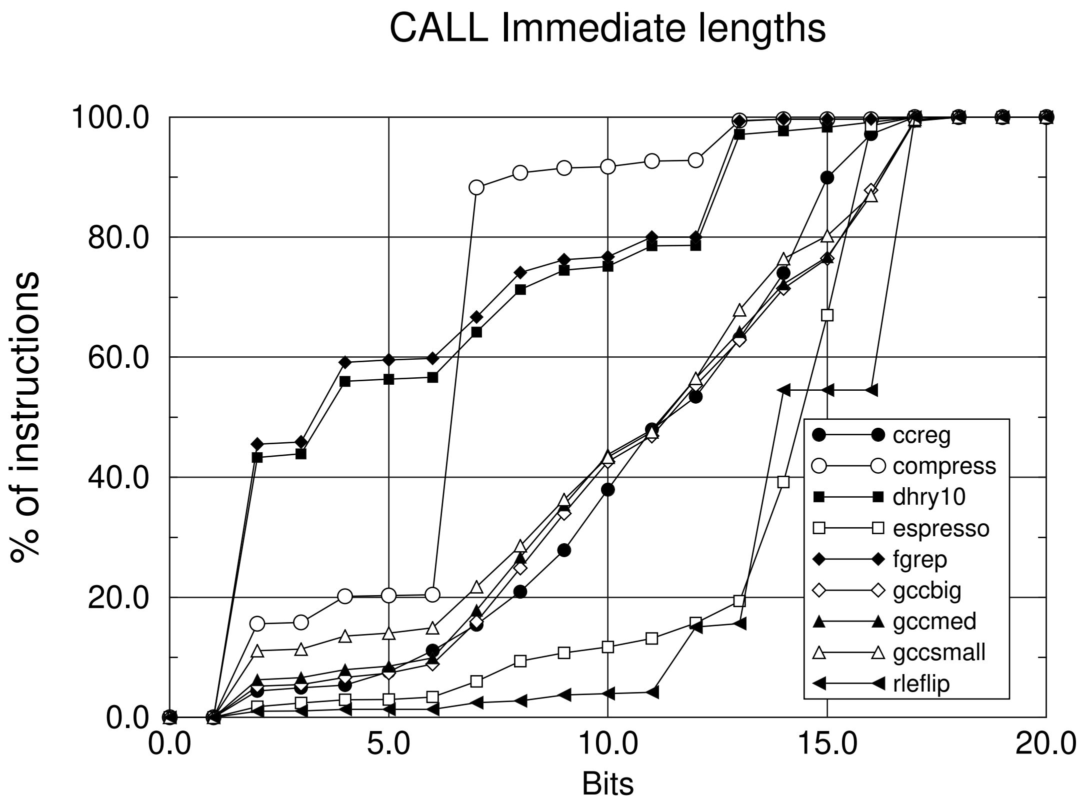


Figure B.13: Call immediate lengths

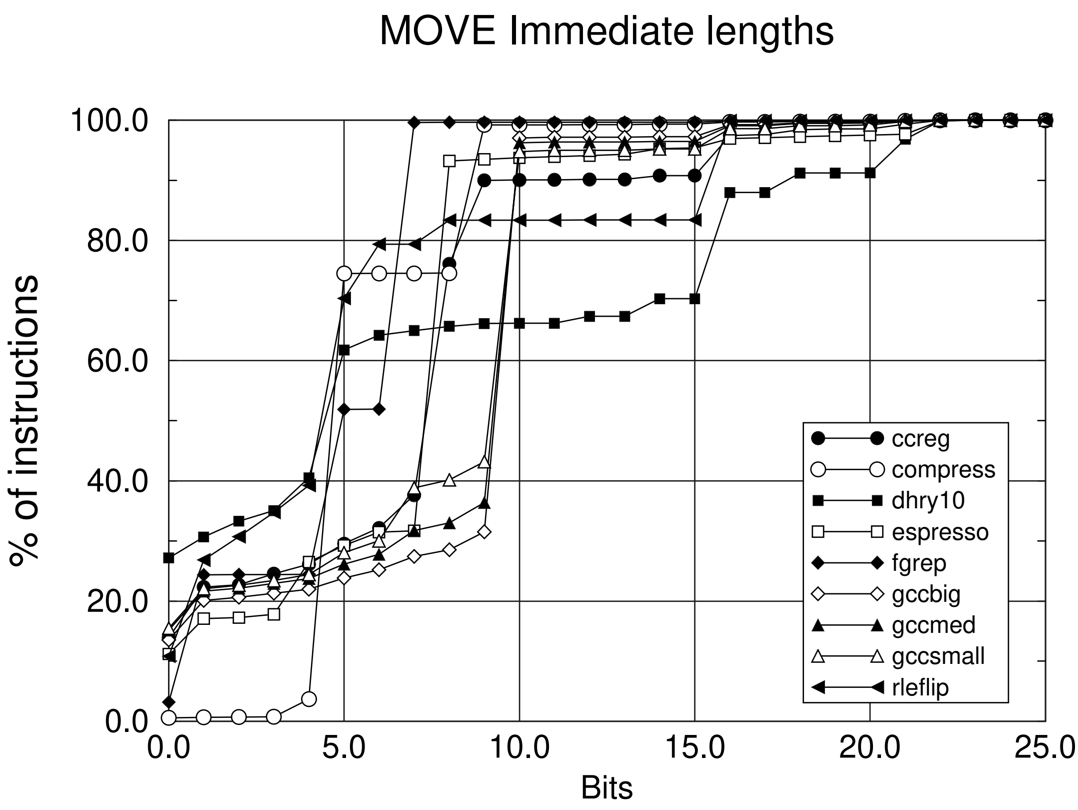


Figure B.14: Move immediate lengths

Table B.2 summarises some of the information presented in figures B.3 to B.14.

Instruction	Immediate field length / bits			
	5	10	15	20
add	86.9	99.6	100.0	100.0
and	59.7	95.0	100.0	100.0
andn	84.2	97.1	100.0	100.0
call	19.8	47.4	82.5	100.0
compare	65.9	99.4	100.0	100.0
conditional branch	63.1	99.4	100.0	100.0
load word	62.9	99.9	100.0	100.0
move immediate	43.9	91.2	92.0	98.4
or	6.8	100.0	100.0	100.0
store word	65.3	99.7	100.0	100.0
sub	97.5	100.0	100.0	100.0
unconditional branch	38.5	94.9	100.0	100.0

Table B.2: *Immediate lengths summary*

B.4 Two and Three Address Instructions

In order to evaluate the potential code density increase resulting from a change from 3-address to 2-address instructions, it was necessary to measure how frequently a three-address instruction could be mapped directly to a two-address instruction; that is, how often one of the source registers was the same as the destination register.

The shade simulator was used to measure this statistic. The results are presented in table table B.3.

By adding the two sub-totals given in the table it can be see that 57.8% of instructions fall into the categories where a 2-address format would be possible. 31.4% of instructions are in one of these categories and have their destination register equal to one of the source registers, allowing a 2-address instruction to be used.

Instruction category		Example	Percentage of instructions
0-read 0-write instructions		Branch	19.7
0-read 1-write instructions		Move immediate	8.2
1-read 0-write instructions		Compare Immediate	3.8
1-read 1-write instructions	2-address	`ADD R1,R1,#1`	25.6
	3-address	`ADD R1,R2,#1`	22.5
	total	Add Immediate	48.2
2-read 0-write instructions		Compare Register	9.9
2-read 1-write instructions	2-address	`ADD R1,R1,R2`	5.8
	3-address	`ADD R1,R2,R3`	3.8
	total	Add Register	9.6
3-read 0-write instructions		Store Register+Register	0.6

Table B.3: *Potential for 2-address instructions*

Appendix C : Benchmark Programs

This appendix describes the benchmark programs that were used to obtain the statistics given in appendices A and B and elsewhere.

C.1 Programs Executed on the Shade Simulator

The statistics presented in sections A.2, B.1, B.2 and B.3 were obtained by extracting data from a simulator during the execution of a set of benchmark programs.

The programs used were executed on a simulator of a Sparc processor called Shade [36] running on a Sun workstation. The execution that has been studied includes the main program and its calls to run-time libraries, but not the operating system activity in response to a system call.

The programs used are as follows:

ccreg. The Sun C compiler, assembler and linker. The benchmark involves compilation, assembly and linkage of the modified simulator using the Sun C compiler and associated tools.

compress. The standard unix data compression program. The benchmark involves applying compress to a text file containing the first 5000 lines of the unix dictionary file, `/usr/dict/words`.

dhry10. The dhrystone synthetic benchmark. The benchmark involves executing the dhrystone loop ten times.

espresso. The logic minimisation program from the University of California at Berkeley. The benchmark involves applying espresso to the description of a PLA from the asynchronous ARM design.

fgrep. The standard unix text search program. The benchmark involves searching for the string `end` in the standard unix dictionary file `/usr/dict/words`.

gccbig, gccmed, gccsmall. These benchmarks are taken from the tape distribution accompanying Hennessy and Patterson's book [7]. They are benchmarks of the C compiler from the Free Software Foundation, GCC. The version used is GCC v1.26, which is configured to generate assembly code for the 68000 processor. The input to the compiler is various files from gcc v1.26 itself. The files used for each of the benchmarks are shown in table C.1. The names small, med and big refer to the size of the input files.

rleflip. A program for manipulating graphics files in the rle (run-length-encoded) format. The program flips the graphic in the $y = x$ axis. The input file used is an image of Jupiter's red spot at a resolution of 505 by 480 pixels with 24 bits of data per pixel.

The total number of instructions executed by each of the benchmarks is shown in table C.2.

The C compiler used by ccreg and the programs compress and fgrep are supplied as compiled programs with the Sun computer installation. They are believed to be compiled using the Sun C compiler at optimisation level -O2.

The dhrystone program was compiled using GCC2 with optimisation enabled.

The GCC benchmark programs were compiled using the Sun C compiler with maximum optimisation[1].

The compiler used for the rle program is not known.

gccsmall	gccmed	gccbig
`version.c`	`regclass.c`	`expr.c`
`genflags.c`	`stor-layout.c`	
`gencodes.c`	`recog.c`	
`genconfig.c`	`rtl.c`	
`genextract.c`	`genrecog.c`	
`genpeep.c`	`global-alloc.c`	
`genemit.c`	`final.c`	
`obstack.c`	`local-alloc.c`	

Table C.1: *GCC benchmark input files*

1. One part of GCC has to be compiled with optimisation off because of a fault in the Sun compiler.

Benchmark	Total number of instructions
ccreg	187 422 583
compress	3 681 440
dhry10	256 821
espresso	35 959 927
fgrep	8 025 076
gccbig	173 787 819
gccmed	365 934 487
gccsmall	93 665 456
rleflip	51 913 246

Table C.2: *Sizes of benchmark programs*

C.2 Address Traces

For the memory access pattern analysis in section A.2, a set of three address traces was used. These traces were supplied on the tape accompanying Hennessy and Patterson's book [7].

The three traces were obtained from the following programs:

cc1. The main part of the GCC compiler.

spice. The circuit simulation program.

tex. The text formatting program.

The total number of data memory accesses in each file is around 200 000 - 250 000.

Unfortunately the compiler used, the input data supplied to the programs and the period of execution that was monitored are not known.

References

[1] van Berkel K.,
personal communication, August 1993,
and unpublished comments at the presentation of “VLSI programming of a modulo-N counter with contant response time”, proc. IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England, March - April 1993.

[2] Linden D.,
“Handbook of batteries and fuel cells”,
McGraw-Hill, 1984, ISBN 0-07-037874-6.

[3] ESD Electronic Services,
“The Electronics Book 1993”,
Edinborough Way, Harlow, Essex, CM20 2DF, U.K.

[4] Digital Equipment Corporation,
“DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet”,
Maynard, Massachusettes, U.S.A., 1992.

[5] Jouppi et. al.,
“A 300MHz 115W 32b Bipolar ECL Microprocessor”,
Symposium Record of Hot Chips V, Stanford Universtiy, California, U.S.A., August 1993

[6] Weste N. H. E., Eshraghian K.,
“Principles of CMOS VLSI Design”,
Second edition, Addison-Wesley, 1993, ISBN 0-201-53376-6.

[7] Hennessy J. L., Patterson D. A.,
“Computer Architecture a Quantitative Approach”,
Morgan-Kaufman, 1990, ISBN 1-55860-069-8.

[8] Hamburgen W. R., Fitch J. S.,
“Packaging a 150W Bipolar Microprocessor”,
DEC Western Research Laboratory Research Report 92/1, 1992.

[9] Sze S. M.,
“VLSI Technology”,
McGraw Hill, 1983, ISBN 0-07-Y66594-X.

[10] Tuckerman D.,
"Heat Transfer Microstructures for Integrated Circuits",
Livermore Labs Report UCRL-53515, 1984.

[11] Gwennap L.,
"Intel Adds Low-Power Features to Every i486",
Microprocessor Report, Vol. 7, No. 8, June 21 1993, p. 1 and pps. 7-8.

[12] Feibus M.,
"Energy Star PCs Debut",
Microprocessor Report, Vol. 7, No. 9, July 12 1993, pps. 23-25.

[13] Antonettie V. W., Simons R. E.,
"Bibliography of heat transfer in electronic components",
IEEE transactions on components, hybrids and manufacturing technology, Vol. 8, No. 2, pps. 289-295, 1985.

[14] Lyon R. F.,
"Cost Power and Parallelism in Speech Signal Processing",
Proceedings of the IEEE 1993 Custom Integrated Circuits Conference, San Diego, California, May 1993.

[15] Garside J. D.,
"A CMOS VLSI Implementation of an Asynchronus ALU",
proc. IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England, March-April 1993.

[16] Dobberpuhl et. al.,
"A 200-MHz 64-b Dual-Issue CMOS Microprocessor",
IEEE J. of Solid State Circuits, Vol. 27, No. 11, November 1992, pps. 1555-1565.

[17] Sutherland I. E.,
"Micropipelines",
Commun. ACM, Vol. 32, No. 6, June 1989, pps. 720-738.

[18] Yuan J., Svensson C.,
"High-speed CMOS circuit techniques",
IEEE J. Solid-State Circuits, Vol. 24, No. 1, pps 62-70, February 1989.

[19] Mead C., Conway L.,
"Introduction to VLSI Systems",
Addison Wesley, 1980, ISBN 0-201-04358-0.

[20] Gopalakrishnan G., Jain P.,
"Some recent asynchronous system design methodologies",

Technical report, Dept. of Computer Science, University of Utah, U.S.A., UU-CS-TR-90-016, October 1990.

[21] Paver N. C.,
"The design and implementation of an asynchronous microprocessor",
Ph.D. thesis, University of Manchester, U.K., to be submitted.

[22] Paver et. al.,
"Register locking in an asynchronous microprocessor",
proc. International Conference on Computer Design, 1992.

[23] Furber et. al.,
"A Micropipelined ARM",
proc. VLSI '93, Grenoble, France, September 1993.

[24] David I., Ginosar R., Yoeli M.,
"Self-timed architecture of a reduced instruction set computer",
proc. IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England, March-April 1993.

[25] Martin et. al.,
"The design of an asynchronous microporocessor",
Advanced research in VLSI: proceedings of the decennial caltech conference on VLSI, MIT Press, pps. 351-373, 1989; also as technical report number Caltech-CS-TR-89-02, Computer Science department, California institute of technology, U.S.A., 1989.

[26] Digital Equipment Corporation,
"Alpha Architecture Handbook",
Maynard, Massachusettes, U.S.A., 1992.

[27] Furber S.B.,
"VLSI RISC Architecture and Organization",
Marcel Dekker, New York, 1989, ISBN 0-8247-8151-1.

[28] Asprey et. al.,
"Performance Features of the PA7100 Microprocessor",
IEEE Micro, June 1993, pps. 22-35.

[29] van Berkel et. al.,
"The VLSI-programming language Tangram and its translation into handshake circuits",
proc. European Design Automation Conference, pps. 384-389, 1991.

[30] Rowen C.,
"MIPS R4200, an innovation in low-power microprocessor design",
presentation at Microprocessor Forum Europe, May 1993.

[31] Bunda J., Fussell D., Jenevin R.,
"16-Bit vs. 32-Bit Instructions for Pipelined Microprocessors",
Technical report TR92-39, Department of Computer Science, University of Texas, 1992.

[32] Masuda T., Fin T.-H.,
"Program Behavior and Its Models",
Lecture Notes in Computer Science, Vol. 143, 1982, pps. 80-103.

[33] Spirn J.R.,
"Program Behavior : Models and Measurement",
Elsevier/North-Holland, 1977.

[34] Thiebaut D., Wolf J.L., Stone H.S.,
"Synthetic Traces for Trace-Driven Simulation of Cache Memories",
IEEE Trans. on Computers, Vol. 41, No. 4, 1992, pps. 388-410.

[35] Thiebaut D.,
"On the Fractal Dimensions of Computer Programs and its Application to the Computation of the Cache Miss Ratio",
IEEE Trans. on Computers, vol. 38, pps. 1012-1026, July 1989.

[36] Cmelik R.F., Keppel D.,
"Shade: A Fast Instruction-Set Simulator for Execution Profiling",
Dept. of Computer Science, University of Washington, U.S.A., Technical Report UWCSE 93-06-06, 1993.

[37] Usher M.J.,
"Information theory for information technologists",
Macmillan, London, 1984, ISBN 0-333-36703-0.

[38] Sun Microsystems Inc.,
"The SuperSPARC Microprocessor",
Technical white paper, 1992, Mountain View, California, U.S.A.

[39] Katevenis M. G. H.,
"Reduced Instruction Set Computer Architectures for VLSI",
MIT Press, 1985, ISBN 0-262-11103-9.