SCALP: A Superscalar Asynchronous Low-Power Processor

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

Philip Brian Endecott

Department of Computer Science

1996

Contents

List of Figures

List of Tables

Abstract

The design of low power microprocessors is an important research area because of the increasing demand for high-performance portable computers with long battery life. Most previous low power microprocessor designs have been constrained by the need to maintain compatibility with existing instruction sets and so have been able to apply power efficiency techniques only at the implementation level and below. Furthermore conventional design approaches have prevented processor implementations from exploiting the power saving potential of asynchronous logic. This thesis describes a processor, SCALP, whose architectural design and implementation are free from these constraints.

SCALP is motivated by three objectives: high code density, highly parallel operation, and asynchronous implementation. These three objectives should all lead to increased power efficiency and also dictate the nature of the architecture.

Conventional instruction sets indicate the flow of data between instructions by means of register numbers; each instruction gives register numbers for its operands and result. This technique leads to complexity in pipelined superscalar implementations: register numbers from many instructions must be compared to identify dependencies and activate appropriate forwarding paths. In asynchronous systems these forwarding paths cause further inefficiency due to the additional synchronisation that they impose. Furthermore the register specifier bits in a typical instruction set take up around half of the total instruction, making them crucial to code density.

SCALP's main architectural innovation is its use of “explicit forwarding”. SCALP does not use a global register bank but rather indicates for each instruction to where the result should be sent. This takes the form of a destination queue identifier. One such queue is

associated with each operand required by each functional unit. Using this scheme the need for many register specifier comparators is eliminated. The arrangement also suits asynchronous implementation and increases code density.

SCALP has other features aimed at increased code density or other ways of increasing power efficiency: it has variable length instructions and operations need only activate part of the datapath for byte operations.

After describing the background to and features of the proposed architecture the implementation is described. SCALP is the first known asynchronous implementation of a superscalar architecture. It operates using the four-phase bundled data protocol like the AMULET2 processor. Much of the design uses a “macromodule” design approach, and the total size of the design is around 9,500 components. The design of some of the more interesting parts of the implementation, such as the parallel asynchronous instruction issuer, are described in detail.

The implementation has been taken to gate level using the hardware description language VHDL. The implementation is extensively evaluated in comparison with a conventional instruction set and conclusions about its effectiveness are drawn.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning, except as indicated in section 2.1.

Copyright and Intellectual Property Rights

Acknowledgements

During my three years as a postgraduate student in Manchester I have received a great deal of support and encouragement from many students and staff in the Department of Computer Science. I would like to thank everyone for their help.

I am especially grateful to my supervisor Prof. Steve Furber who has encouraged my creativity, and to the other members of the AMULET research group.

Thanks are also due to those people who have helped by proof reading and commenting on this thesis, in particular Richard York and David Gilbert.

The Author

Philip Endecott obtained a B.Sc. (I) degree in Computer Science from the University of Manchester in 1991. After a year employed by Advanced RISC Machines Ltd in Cambridge he returned to Manchester as a postgraduate student. In 1993 his M.Sc. thesis entitled “Processor Architectures for Power Efficiency and Asynchronous Implementation” was submitted. This thesis represents the result of a further two years work in this area.

Chapter 1: SCALP: A Superscalar Asynchronous Low-Power Processor

To design a superscalar microprocessor is a complex and challenging task. To do so using asynchronous logic - the subject of this thesis - makes it doubly difficult. So what is it that makes such a processor desirable? The answer is that superscalar and asynchronous operation, along with a number of other factors, should help to make SCALP a highly power efficient processor.

Power efficiency is becoming an increasingly important factor in digital design, particularly as a result of the proliferation of portable battery-operated applications. Proposed design techniques for low power systems are discussed later in this chapter. These techniques have been applied with particular success in special purpose signal processing applications such as the Philips Digital Compact Cassette Player [BERK95] and the Berkeley InfoPad [CHAN94]. In the case of general purpose processors however designers have been less adventurous.

Many constraints influence the designer of a general purpose microprocessor. Often it is necessary to maintain code compatibility with a previous design. If absolute code compatibility is not necessary then the processor is expected to follow a model that assembly language programmers and compiler code generators are used to. At the very least an execution model that is a suitable target for common high level languages is needed.

Other constraints limit the choice of implementation technology. Compatibility with existing memory and interface circuits requires that the processor operates in a conventional synchronous fashion at a fixed standard supply voltage. All these restrictions limit the designer’s ability to apply novel and radical ideas that may have benefits in terms of power efficiency or other factors.

This work has suffered no such impediment. SCALP, the processor described here, deviates significantly from the conventional idea of a microprocessor in order to try architectural and implementation ideas that may contribute towards increased power efficiency.

This first chapter starts by providing a background explaining why power efficiency is becoming increasingly important. Proposed techniques for improving power efficiency are reviewed, and finally an overview of the remainder of the thesis is presented.

1.1 The Importance of Power Efficiency

There are several reasons why power efficiency is becoming increasingly important. Most importantly portable systems powered by batteries are performing tasks requiring increasing computational performance. At the same time these systems are becoming physically smaller and battery weight is becoming more significant. Users demand longer battery life and this can only be obtained by increasing the capacity of the battery or by increasing the efficiency of the logic. The rate of progress in battery technology is slow, so the onus is on the digital designer to improve efficiency.

There are other reasons why power is becoming important. As the heat output of chips increases it becomes more difficult and expensive to provide sufficient cooling in the form of special packages, heat sinks and fans. Furthermore increased temperatures lead to greater stress on the component and reduced reliability.

Electrical issues are also important. Providing a supply of sufficient capacity requires a large number of bond wires between the chip and the package, and a large proportion of the potential signal routing space is occupied by power distribution. High current densities can lead to electromigration. At the system level, increased power demands larger and more expensive power supplies.

Even the cost of the electricity used can be important. It is estimated that at the end of the decade 10 % of U.S. non-domestic electricity consumption will be by personal computers. Reducing this contribution would have significant economic and environmental benefits.

These factors, and others described in [ENDE93], combine to make power efficiency an increasingly important factor in the design of digital systems.

1.2 Low Power Design Techniques

This section reviews techniques that have been suggested for improving power efficiency. These techniques apply at a number of levels from transistors through to high level design. The savings obtained at each of these levels tend to combine multiplicatively, so a successful low power system must employ techniques at all levels.

The descriptions in the following sections apply to CMOS logic unless otherwise indicated.

1.2.1 Low-Level Power Efficiency

Process Technology

The ever-diminishing feature size possible with successive generations of fabrication processes is greatly beneficial to power efficiency. As feature sizes shrink capacitances are reduced and gates become faster. As a first approximation the energy per gate transition is proportional to the cube of the feature size and the gate delay is directly proportional to the feature size if the supply voltage is scaled with the feature size [WEST93].

As feature sizes continue to scale interconnect delays become increasingly significant. Interconnect delays do not scale with feature size or with supply voltage and so eventually limit the available performance.

Improving process technology also allows for the construction of increasingly large circuits on single chips. Inter-chip connections are far more power consuming and slower than on chip connections, so as the number of inter-chip connections necessary is reduced the power efficiency of the system is improved.

Transistor Level Design

The size of the transistors used to build basic logic gates affects their speed and power consumption. One objective is to obtain fast output changes as short-circuit current flows in the driven gate at intermediate voltages during the transition. On the other hand it is desirable to minimise transistor sizes in order to reduce capacitances. Recent work [FARN95] has investigated the ideal transistor sizes for greatest power efficiency.

In some cases asymmetric gates can be constructed; some input to output paths are faster and are used for critical paths while others are more power efficient and are used for less critical paths. The scope is greatest in the case of complex gates where the order of the transistors can be chosen based on known input transition orders and probabilities [GLEB95].

Charge Recovery and Adiabatic Systems

Conventional circuits charge nodes that must become high by connecting them to the positive supply and discharge nodes that must become low by connecting them to the negative supply. This mode of operation leads to the conventional CMOS power consumption equation:

$$P \propto CV_{\text{supply}}^{2}$$

Other schemes use different techniques. Adiabatic systems [SOLO94] use inductors, capacitors and sinusoidal clock / power signals to charge nodes in a more efficient fashion such that

$$P \propto V_{\text{supply}} V_{\text{threshold}}$$

More readily implementable schemes use charge sharing between signals; signals that are high and wanting to become low are connected to signals that are low and wanting to become high, moving all nodes to an intermediate level. Subsequently the power supply is used to complete charging or discharging the nodes [KHOO95].

Logic Optimisation

For conventional CMOS the majority of the power consumption is dynamic; that is power is consumed only when signals change. Power efficiency can therefore be improved by reducing the probability that signals will change.

In many cases the most common mode of operation of a block is known, for example a typical sequence of states for a state machine or a typical set of inputs for a combinational block. Low power consumption in this mode can be an objective of the design process. Algorithms have been proposed that provide state encodings and implementations for optimum power efficiency for state machines based on typical state sequences [OLSO94].

Supply Voltage Adjustment

Optimum power efficiency occurs when a circuit is powered from the lowest supply voltage at which it can provide the necessary performance. Recent work [USAM95] has suggested providing multiple supply voltages to a system. Gates on the circuit’s critical path are operated from the higher voltage and those not on the critical path are operated from the lower voltage. Other work [NIEL94] [KESS95] proposes that the power supply to the whole circuit is dynamically adjusted in response to the varying workload (see section 4.1.4).

Parallelism

Increasing the parallelism in the system permits individual gates to operate more slowly while maintaining the same overall throughput. Slower operation may be performed at a lower supply voltage, increasing power efficiency [CHAN94] (see chapter 3).

Precomputation

Precomputation may be used to prevent unnecessary operations from being performed; for example in a comparator it is possible to initially compare only a few of the bits. Only if these bits match need the remaining bits be compared. In the case where the first comparison gives a false result power has been saved [ALID94].

Clocking Schemes and Asynchronous Logic

In synchronous systems significant power is consumed by the clock signal, its driver and the driven latches. Clock gating may be employed to disconnect the clock from parts of the circuit that are inactive at a particular time [BENI95].

A more radical approach is to eliminate the global clock altogether. In asynchronous systems, local timing signals control communication between blocks which occurs only when useful work has to be performed (see chapter 4).

1.2.2 Higher-Level Power Efficiency: Microprocessors

At higher levels of abstraction methods of increasing power efficiency are dependent on the particular application. In the case of microprocessors, this means the architectural properties of the processor and its instruction set.

Previous work by the author identified three particularly important architectural features: external memory bandwidth, cache characteristics, and datapath arrangement [ENDE93].

External Memory Bandwidth

The power consumed operating an external signal is substantially larger than the power used by an internal transition; consequently it is important to minimise number and size of external memory accesses. This can be effected by:

- Increasing the code density; a processor with a higher code density will require fewer or smaller instructions to be fetched for a given workload.

- Reducing the frequency of load and store operations. This may be achieved by for example incorporating a larger register bank.

- Improving the on-chip cache characteristics. A higher cache hit rate will reduce the frequency of external memory accesses. Cache write policy is also important; write-through caches will cause more external memory accesses than copy-back caches.

Cache Characteristics

Because the cache is a large regular structure its power efficiency properties are both important and easy to study. Larger caches will consume more power than smaller ones; this must be traded off against the corresponding reduction in external memory access frequency mentioned above.

Greater power efficiency can be obtained with multiple levels of on-chip cache. In this case the power consumption should be determined by the small size of the more frequently used first level cache, whereas overall performance should be determined by the large size of the second or further levels.

Datapath Arrangement

Typical RISC processors have a 32-bit datapath which is used for all operations. Many of those operations will be operating on smaller quantities such as characters or small loop counters that could be processed using a small fraction of the datapath width. Power efficiency will be improved if the instruction set provides datapath operations of various widths.

1.3 Previous Low Power Processors

Most previous low power microprocessors make use of only low level power efficiency techniques in order to maintain code compatibility with previous high power versions of the processors. This section describes the low power techniques used by these conventional processors.

The ARM processor [FURB89] is considered to be one of the most power efficient microprocessors available [ARM95]. Its power efficiency is due primarily to its simplicity and consequent small size. The processor was designed for simplicity in order to reduce both design complexity and cost; the resulting low power consumption was a bonus. While most microprocessors are available only as single integrated circuits the ARM is available as a VLSI macrocell which can be incorporated into a larger chip. In this way the number of off chip interconnections required is reduced, increasing power efficiency.

Other processors that are inherently more complex must apply other mechanisms to increase their power efficiency. The most common approach is to introduce a number of power saving modes in which parts of the processor are disabled. Typically there would be several modes such as “standby”, “sleep”, “hibernate” etc. As deeper levels are entered the number of active blocks reduces. Generally these modes are controlled by operating system software. As one example the NEC 4100 processor [TURL95b] has standby, suspend, and hibernate modes that reduce power consumption to 10 %, 5 %, and virtually zero respectively. This processor is targeted at the portable “personal digital assistant” market. Another example is the Hitachi SH7708 processor [TURL95a] which allows the clock frequency to be dynamically changed to 1, 2, or 4 times the input frequency.

Some processor designs have made use of transistor sizing to minimise power consumption. The PowerPC 400 series processors [CORR95] have been designed using a large library of basic gates with many different combinations of transistor sizes. The choice of which gate to use is made by a synthesis tool; the result is that relatively few gates on critical paths use large transistors for high speed and the majority use smaller transistors for low power consumption.

Some other processors do incorporate architectural features that benefit power efficiency, though often these features were not incorporated with power efficiency in mind. Specifically section 2.1.3 considers a number of processors with high code density. Generally though no current microprocessors make high-level architectural choices such as changing the underlying execution model in order to improve power efficiency.

1.4 Overview of the Thesis

Whereas the previous work described above has considered primarily the low level power efficiency techniques, this work concentrates primarily on three of the higher level techniques described in section 1.2, namely parallelism, asynchronous logic and high code density.

The thesis is arranged as follows:

Chapters 2 to 4 consider each of the areas of interest in turn. Chapter 2 considers how the number of transitions required to execute a program can be reduced by increasing code density and reducing the datapath activity. Chapter 3 considers parallelism, particularly how parallelism is obtained in pipelined and superscalar processors. Chapter 4 considers asynchronous logic, investigating how its use can improve power efficiency and considering how the parallel structures described in chapter 3 can be implemented asynchronously.

Chapter 5 describes the SCALP architecture and explains how its novel organisation helps several of the objectives identified in the previous chapters. Chapter 6 gives an introduction to asynchronous design styles before describing the implementation of SCALP as a gate level VHDL model. It gives an overview of the design and then concentrates on some of the more interesting blocks. Chapter 7 presents an evaluation of the processor and studies how well it meets its objectives, identifying strengths and weaknesses.

Chapter 8 gives a summary of the work and then considers ways in which the architecture and implementation could be improved. Ways in which the successful SCALP techniques could be applied to more conventional processors are also considered.

Chapter 2: Reducing Transitions

One way in which power efficiency can be increased is to reduce the amount of circuit activity required to carry out a computation. This chapter considers two particular ways in which the number of signal transitions that occur while executing a program can be reduced.

The first section is concerned with increasing code density. If code density can be increased then the number of signal transitions in the instruction memory and instruction fetch logic is reduced. The second section considers how datapath activity can be reduced and proposes that when operands are smaller than the whole width of the datapath only the required part of the datapath should be activated.

2.1 Code Density

Code density is a measure of the compactness of encoding of a processor's instruction set. In this section the relationship between code density and power efficiency is shown, and ways in which code density can be altered through changes to the instruction set architecture are investigated. Much of this section is based on previous work by the author in [ENDE93].

The code density of a processor can be defined as the reciprocal of the amount of program code required to perform some standard benchmark task. There are two variants of this measure:

- Static code density measures the total amount of memory required to store the code for the benchmark program.

- Dynamic code density measures the total amount of processor - memory code communication required for the execution of the benchmark program.

Static code density and dynamic code density tend to be correlated, but factors such as compiler optimisation can lead to differences. For example, if the compiler unrolls loops in the program the static code density will decrease but the dynamic code density will increase.

Static code density is important when questions such as the size of cache memories are considered; a processor with a higher static code density can obtain equivalent performance with a smaller cache. For questions related to power efficiency it is generally the dynamic code density that is most important.

As a first approximation power consumption in the memory and related parts of the system is inversely proportional to the dynamic code density. If dynamic code density could be doubled then the activity in the memory would be halved, and consequently its power consumption would be halved. The power consumption associated with the main memory system is particularly important as the main memory is typically on a different chip from the processor itself and inter-chip communication is less power efficient than on-chip communication.

Compared with previous generations, modern RISC processors have a relatively low code density. This has occurred for a number of reasons:

- In order to minimise the complexity and increase the speed of the processor's instruction decoding logic, the instruction encoding is simple and hence redundant. Older designs used a microcode ROM for decoding which made more arbitrary encodings possible.

- All instructions are the same length, and the length is determined by the longest required instruction. Other instructions have unused bits. Previous designs used variable length instructions.

- RISC processors have a large number of registers. Typically five bits are used to specify one of 32 registers, and instructions may specify three registers, requiring a total of 15 bits. In earlier systems compilers were unable to make use of so many registers and fewer were provided.

Previous work [ENDE93] measured the potential improvement in code density that could be achieved for a SPARC processor by changing certain aspects of its instruction set architecture. Please refer to appendix A for details of the benchmark programs used. The possible improvements are summarised in table 2.1; following sections interpret many of the values in this table. It can be seen that there is no single way in which the code density can be dramatically improved, though taken together a combination of these changes could have a substantial effect.

	Change	Relative code density
1	Remove unused bit fields	1.03
2	Fixed length 5-bit opcode field	1.08
3	Huffman encoded variable-length opcodes	1.25
4	10-bit branch displacements	1.07
5	1, 3 and 10-bit add and subtract immediates	1.04
6	2- and 3-address instructions	1.05
7	Explicit use of last result	1.05
8	Explicit generation and use of last result	1.08
9	16 registers	1.01
10	Workspace pointer register	1.03
11	Load and store multiple	1.11

Table 2.1: Improvement in SPARC code density resulting from instruction set changes

This section investigates two of these areas where SCALP makes improvements over conventional architectures. These areas are the use of variable length instructions and the code density of register specifiers. The final subsection describes a number of other processor designs that have higher than usual code density.

2.1.1 Variable Length Instructions

There are two ways in which the use of variable length instructions can improve code density:

- Opcodes can be encoded with longer codes for less common operations and shorter codes for more frequent operations. Line 3 of table 2.1 above shows that this technique can increase SPARC code density by 25 %; however to achieve this increase it is necessary to allow opcodes to be any number of bits in length. Realistic variable length instruction formats will allow only a small number of instructions lengths, reducing the benefit of this approach.

- The lengths of immediate fields can be varied. Lines 4 and 5 of table 2.1 shows that if a branch instruction with an immediate field of 10 bits and arithmetic operations with immediate fields of 1, 3 and 10 bits are added SPARC code density can be increased by 11 %.

The reason why RISC processors avoid variable length instructions is that they are more difficult to decode than fixed length instructions. This is especially true when a superscalar processor needs to decode more than one instruction at once.

Figure 2.1 shows how two fixed length instructions can be decoded in parallel. In contrast figure 2.2 shows how five variable length instructions in the same total number of bits must be decoded (the solid lines indicate instruction boundaries, the dotted lines indicate boundaries between parts of the same instruction). Because it cannot tell in advance where the boundaries between instructions lie the decoder must consider the instruction parts serially, making the operation significantly slower.

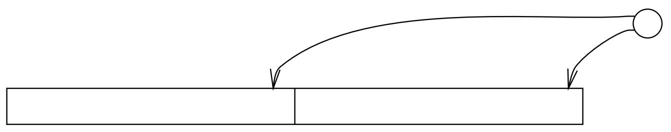


Figure 2.1: Decoding Fixed Length Instructions

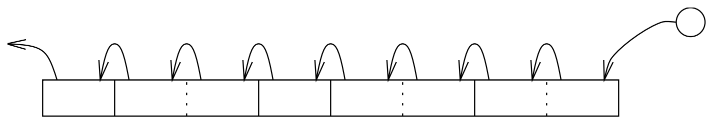


Figure 2.2: Decoding Variable Length Instructions

There are ways in which decoding variable length instructions can be made easier. The simplest modification is to ensure that the first part of each instruction contains all of the control information to indicate the total length of the instruction. In this case the decoder can operate more quickly as shown in figure 2.3.

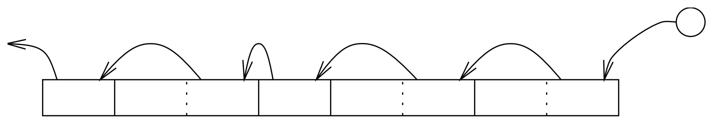


Figure 2.3: More Efficient Variable Length Instruction Decoding

A more radical approach is indicated in figure 2.4. Here a “control field” is associated with each group of instructions. The control field is decoded first and indicates the layout of the instructions within the group to the decoder. The decoder is therefore able to decode all instructions in parallel once the control field has been decoded, making the decoding process nearly as fast as the fixed length instruction decoding. This is the technique used by SCALP.

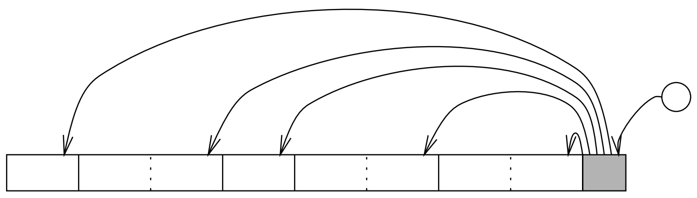


Figure 2.4: Variable Length Instruction Decoding with a Control Field

2.1.2 Register Specifiers

RISC processors typically use up to about half of the bits in their instructions to represent register numbers. When other techniques such as less redundant opcode encodings and variable length instructions are used this fraction grows more significant. It is important to be able to reduce this factor in order to increase overall code density.

Simply reducing the number of registers is not a solution. Compilers are able to use 32 registers effectively and if the number is reduced the number of load and store instructions is increased.

Reducing the number of register specifiers per instruction is a strong possibility. It is common to find that the destination register is the same as one of the source registers; this was quantified in [ENDE93] (see appendix A) and is summarised in table 2.2. This form of encoding is known as a 2-address encoding rather than a 3-address encoding. Table 2.2 shows that in around one third of instructions one register specifier can be eliminated and a 2-address instruction used. Line 6 of table 2.1 indicates that this can lead to a 5 % code density increase.

Using 2-address instructions is a way of exploiting the locality of register specifiers: if a particular register is given by one register specifier in an instruction then there is an increased chance that it is given by another register specifier in the same instruction. This idea of locality can be taken further. It can be observed that if a register is used in one instruction it has an increased chance of being used by preceding or following instructions.

Instruction category	Example	% of instructions
No register specifiers	Branch	19.7
One register specifier	Move immediate	12.0
Two register specifiers, equal	ADD R1,R1,#1	25.6
Two register specifiers, not equal	ADD R1,R2,#1	32.4
Three register specifiers, two equal	ADD R1,R1,R2	5.8
Three register specifiers, not equal	ADD R1,R2,R3	4.4
Total potential 2-address instructions		31.4

Table 2.2: Frequency of 2-address instructions in SPARC code

It is very common for the result that has been computed by one instruction to be used by the immediately following instruction. By referring to “the result of the last instruction” in shorthand rather than by giving its register number a significant code density increase can be obtained. Table 2.3 summarises measurements of this property made in [ENDE93] (see appendix A).

	Instruction category	% of instructions
A:	Use the result of the previous instruction	43.8
B:	As A, but excluding branch instructions that use the boolean result of a preceding compare	29.4
C:	As B, and the value used is not used by any subsequent instructions	20.6

Table 2.3: Frequency of Last Result Re-use

Line 7 of table 2.1 indicates that code density can be increased by 5 % by using an instruction encoding which indicates last result reuse in shorthand.

Line C of table 2.3 suggests how this idea can be taken further. Many instructions use the result of the previous instruction and that result is subsequently never used again. In this case it is not necessary for either instruction to provided a register specifier to refer to this value; both may use shorthand. Line 8 of table 2.1 indicates that both forms of shorthand together provide an 8 % code density increase.

It is possible to go further and to allow instructions to refer to the results of other preceding instructions, leading to further code density improvements. This idea forms the basis of the "explicit forwarding" mechanism used by SCALP which is described in chapter 5.

2.1.3 Previous High Code Density Processors

Conventional RISC processors have relatively poor code density due to their regular fixed length instruction format. In comparison the preceding generations of CISC processors made use of a more dense variable length instruction encoding.

In addition to the CISC processors a number of other architectures have increased code density. This section considers three processor designs; D16 and Thumb use 16-bit RISC-like instructions whereas the transputer uses a variable length stack based instruction set.

D16

[BUND94] proposes a 16-bit RISC-like instruction set called D16. This is based on DLXe which is in turn very similar to the MIPS architecture. Comparisons are made between the code density and power efficiency of D16 and DLXe.

D16's instruction set is relatively simple with only five instruction formats. The instructions specify at most two register specifiers and can access 16 registers.

The dynamic code density of D16 is found to be around 70 % higher than that of DLXe. However as the 16 bit instruction set contains less redundancy than the conventional instruction set, more bits on the instruction fetch bus change per cycle. For D16 on average 3.88 bits change per byte of instruction; for DLXe 2.85 bits change. This means that the 70 % code density increase leads to a power efficiency increase on the instruction fetch bus of only around 30 %.

Thumb

Thumb [ARM95] is an extension to the ARM architecture that extends the instruction set to include a set of 16-bit instructions. These 16 bit instructions are dynamically translated to ARM 32 bit instructions during decoding.

The Thumb instruction set is complex with 19 different instruction formats. Most operations use a two address format and can access 8 registers. The standard 32 bit ARM instruction set is available when necessary; for example the Thumb instruction set contains no floating point instructions.

The motivation for Thumb is to increase code density so that ROMs in embedded applications may be smaller, and so that performance can be increased when the processor is connected to a byte or 16 bit wide instruction fetch bus. The reduced instruction fetch requirement also leads to increased power efficiency. Thumb’s code density is around 50 % higher than the standard ARM processor.

Transputer

The transputer [MITC90] is an unusual processor with two key features that lead to increased code density.

Firstly the execution model is based on a stack rather than a register bank. Most instructions receive their operands from and write their results to a three entry stack. Other data must be stored in memory, though a small memory is integrated with the processor making this relatively efficient. Memory accesses may be relative to a workspace pointer.

Secondly short variable length instructions are used. Instructions are multiples of one byte long. The most common instructions using only implicit stack addressing without immediate values are encoded in a single byte; less common instructions and immediate values are encoded using “prefix instructions”.

2.2 Datapath Activity

Conventional RISC processors normally have a 32 or 64 bit wide datapath consisting of register bank, ALU, shifter, multiplexors, latches etc. This datapath works efficiently when it is processing quantities of this full width such as addresses but many of the values dealt with by a program are significantly narrower than this. Examples include characters which are encoded by only 8 bits and booleans which require only one. When a 32 bit datapath is used to operate on these smaller quantities much of the power used is wasted.

The solution is to add operations to the instruction set that operate on small quantities. When executing these instructions the processor need only activate a part of the datapath.

The power benefit of the reduced datapath activity must be balanced against the reduced code density resulting from the additional width information in the instructions. SCALP chooses the simplest possible scheme with each instruction using a single bit to indicate whether it operates on byte or word quantities.

Many earlier CISC processor architectures included datapath operations of more than one width. In some cases these features lead to complications when advanced implementations of these architectures were considered. Consider the following example:

- An early instruction writes a 32-bit value into a register.
- A later instruction starts execution and will write an 8-bit value into the same register.
- The immediately following instruction wishes to read a 32-bit value from this register.

In a pipelined implementation, a forwarding path would be used to send the result of the second instruction to the third. However the value that the third instruction should receive should contain 8 bits from the second instruction and 24 from the first instruction. It is therefore necessary for the third instruction to get part of the value from the forwarding path and part of it from the register bank. This is complex and undesirable.

This problem has recently been reported in the Intel P6 processor which is a superscalar implementation of the old 8086 architecture [GWEN95].

To avoid this type of problem SCALP defines the most significant bits of a value to be undefined when an instruction has operated on only the least significant byte.

Chapter 3: Parallelism

This chapter investigates the idea that increased parallelism can be beneficial for power efficiency, and then describes forms of parallelism that can be used in microprocessors and other systems. Firstly the relationship between power, computational throughput, parallelism, and supply voltage is established.

3.1 Parallelism and Power

Let

V be the supply voltage

P be the power consumption

S be the speed at which gates operate

N be the extent of parallelism

and T be the computational throughput

For CMOS circuits, over a reasonable range of supply voltages the gate speed depends directly on the supply voltage [WEST93]:

$$S \propto V \tag{1}$$

If the potential parallelism is fully utilised, the throughput depends on the gate speed and the parallelism:

$$T \propto SN \qquad (2)$$

If there is no overhead associated with the parallelism, for conventional CMOS power consumption depends on the square of the supply voltage and the throughput [WEST93]:

$$P \propto TV^2 \qquad (3)$$

Let us derive the relationship between P and N for constant T and variable V:

Substituting from (1) and (2) into (3):

$$P \propto T\left(\frac{T}{N}\right)^2 \qquad (4)$$

Since T is constant, from (4)

$$P \propto \frac{1}{N^2} \qquad (5)$$

So power decreases with the square of the parallelism. For example, if a processor with no parallelism is replaced by another with parallelism of two, the power consumption is reduced by a factor of four if the supply voltage is adjusted to maintain the same throughput. The overhead of the parallelism will eat in to this advantage to some extent; the additional control logic will consume power, and the utilisation of the parallelism may be incomplete. Although these effects may be quite significant in some cases they are unlikely to exceed the factor of four mentioned here.

Figure 3.1 plots the relationship derived above for parallelism between 1 and 5.

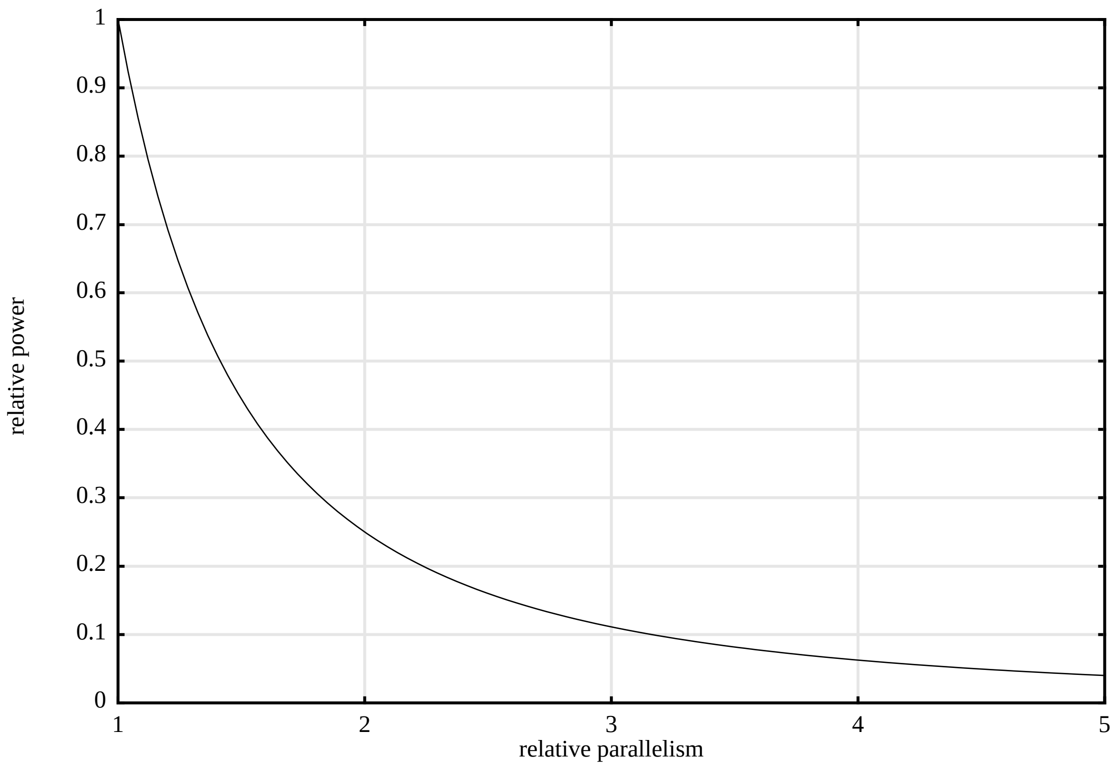


Figure 3.1: The Relationship between Parallelism and Power Efficiency

3.2 Power, Parallelism and Cost

One practical disadvantage of an increase in parallelism is the resulting increase in cost, particularly as many power-sensitive embedded applications are also cost sensitive.

For most forms of parallelism circuit area increases at least linearly with the degree of parallelism (the important exception is pipelining which is discussed later). Cost depends on area, and as area increases yield decreases further multiplying the cost.

In most cases the question of what additional cost can be sustained for a particular increase in power efficiency is a matter of marketing. In some cases however a quantitative argument can be made.

Consider a system comprising a processor and a battery. If the power efficiency of the processor can be increased then a cheaper lower capacity battery could be used with the same overall system performance. If

C_{p1} is the cost of the non-parallel processor

and C_{b1} is the cost of the battery for the non-parallel processor

then the optimum degree of parallelism, N_{opt}, to minimise total cost is given by

$$N_{\text{opt}} = \sqrt[3]{\frac{2C_{b1}}{C_{p1}}} \qquad (6)$$

This result is unappealing because it suggests that in this case even two-way parallelism is attractive only when $C_{b1} = 4C_{p1}$. Other factors for desiring low power, such as increased battery life or reduced battery weight, are less quantifiable but may be more compelling.

3.3 Parallelism in Special Purpose Processors

The use of increased parallelism to reduce power consumption has been applied in a number of digital signal processing applications with good results.

An excellent example is the Berkeley InfoPad work [CHAN94] [RABA94] which has implemented a number of signal processing functions for a portable multimedia terminal including video decompression. These circuits incorporate substantial parallelism, allowing them to operate at reduced supply voltages and hence with increased power efficiency. Table 3.1 compares two implementations of a video decompression circuit.

These results show that in this special purpose processing application the use of parallelism and reduced supply voltage can lead to very great power savings. It would be a great success to achieve the same sort of savings in general purpose processors.

Design Style	Clock Frequency / MHz	Supply Voltage / V	Power / mW	Area / mm^2
Uni-processor	8.3	3.3	10.8	72
Multi-processor	1.04	1.1	0.25	112

Table 3.1: Video Decompression Circuits from [RABA94]

3.4 Parallelism in General Purpose Processors

In the past parallelism has not been applied to general purpose processors with the objective of improving power efficiency, but it has been extensively used to improve performance and exactly the same techniques may be applied. These techniques and their applicability to low power processors are described in the remaining sections of this chapter.

Parallelism in processors may be divided into two categories:

- Parallelism that is explicitly visible to the programmer, for example the multiple communicating processors of a transputer network.

- Parallelism that is not visible to the programmer, for example using pipelining to overlap the execution of several instructions.

Making parallelism visible places the duty of exploitation on the programmer rather than the processor designer. In the case of a transputer network the hardware overhead is low; the only requirement is for a number of communication channels between the processors. On the other hand the programmer's overhead is large: he must re-write his algorithms to use message-passing parallelism.

When parallelism is invisible to the programmer, code that has been written for a conventional processor can be executed unchanged on the parallel machine. The hardware takes responsibility for detecting and exploiting the available parallelism; consequently the processor design is more complex.

Some examples of programmer-visible and invisible parallelism are show in table 3.2.

Fully Visible	Visible only to compiler	Invisible
Message passing multi-processors	Vector processors	Pipelined processors
Shared memory multiproc-essors	VLIW processors	Superscalar processors

Table 3.2: Programmer-visibility of parallelism

With all of these approaches the potential for increased power efficiency is determined in the same way by the degree of parallelism achieved, as explained in section 3.1. SCALP uses parallelism that is generally invisible to the programmer and so this chapter focuses on pipelining and superscalar execution.

The following two sections describe the implementation of conventional pipelined and superscalar processors, and it will be seen that particularly in the case of superscalar processors the implementation complexity is high. SCALP finds a compromise that can maintain the programming simplicity of programmer-invisible parallelism and bring some of the hardware simplicity of programmer-visible parallelism.

3.5 Pipelining

Pipelining is a particularly attractive form of parallelism because it does not carry the same area overhead that other forms do. This section seeks to find out what degree of pipeline parallelism can be obtained reasonably in a microprocessor and what cost is associated with its implementation.

The degree of parallelism depends on two factors:

- The number of pipeline stages.
- The extent of utilisation of the stages.

Pipelining is possible in RISC microprocessors as a result of their regular instruction sets. In these processors all instructions follow virtually the same sequence of micro-operations[1]:

- Generate the address of the instruction.
- Fetch the instruction from the memory.
- Decode the instruction.
- Read the source registers.
- Perform the ALU operation.
- For load and store instructions, access the memory.
- Write the result to the destination register.

There are a number of ways in which these operations can be organised into pipeline stages. The particular organisation chosen will depend on the relative speeds of each of the operations. Here we consider a six-stage organisation where each of these operations is carried out in a separate pipeline stage, with the exception of instruction decode and register read which can be carried out together[2]; other pipeline organisations are considered later.

1. branch instructions excepted.

2. The positions of the source register specifiers in a RISC instruction are fixed so these registers can be read while the rest of the instruction is decoded.

The structure and timing of this pipeline are shown in figures 3.2 and 3.3. In figure 3.3 each column represents a clock period and each row a pipeline stage; the numbers indicate which instruction is active in each stage at each time. It would appear that this organisation offers 6 fold parallelism which has a potential 36 fold power saving. In fact a non pipelined processor would not have to execute exactly 6 times as many cycles as the pipelined processor because many instructions do not need to perform every operation; only loads and stores access the memory, compares and stores do not write a result to the register bank, branch instructions do not access the register bank etc.

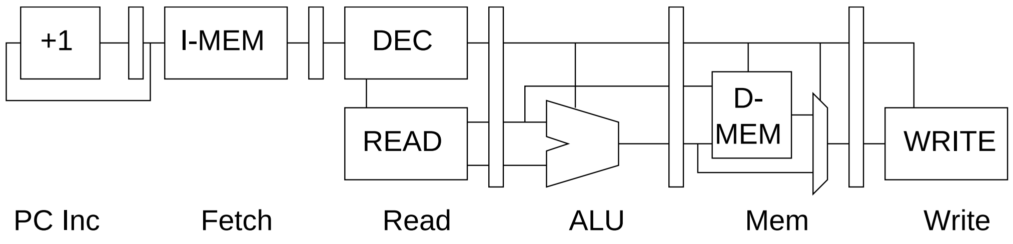


Figure 3.2: A 6-Stage Pipeline

New PC	1	2	3	4	5	6	7	8	9
Fetch		1	2	3	4	5	6	7	8
Read			1	2	3	4	5	6	7
ALU				1	2	3	4	5	6
Mem					1	2	3	4	5
Write						1	2	3	4

Figure 3.3: Timing of a 6-Stage Pipeline

Furthermore there are two particular problems that could reduce the performance of the pipelined processor: branches and dependencies. The following sections explain these problems and the mechanisms used to mitigate their effects; afterwards table 3.3 quantifies the parallelism that is obtainable.

3.5.1 Branch Instructions

Branch instructions do not work well in simple pipelines because until they have computed their target address instructions from the target address cannot be fetched and begin execution. In a processor with a single adder branch target addresses take the path highlighted in figure 3.4. The timing of this organisation is shown in figure 3.5; instruction 3 is a branch to instruction 10. Two instructions (shaded) must be discarded after the branch is fetched.

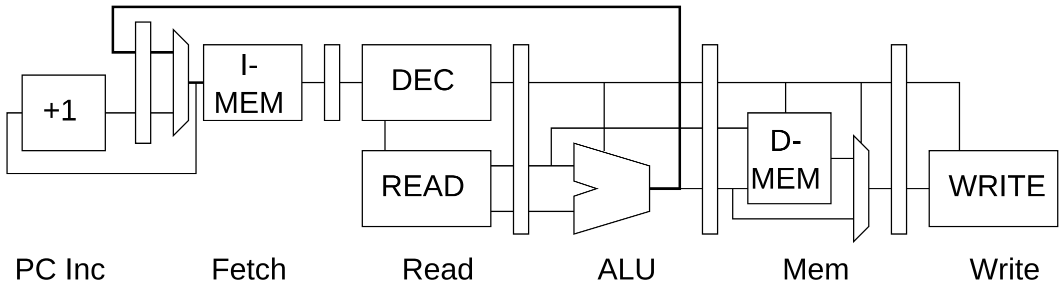
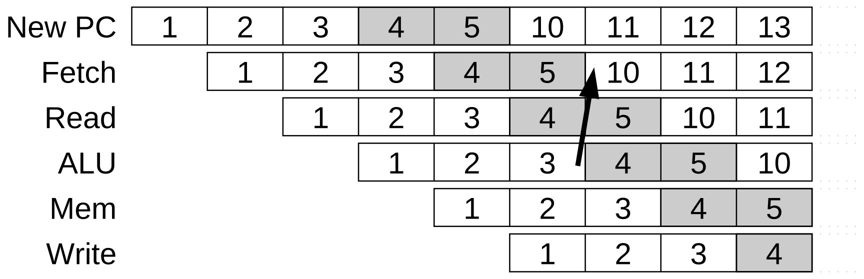


Figure 3.4: Branches in a 6-Stage Pipeline



New PC	1	2	3	4	5	10	11	12	13
Fetch		1	2	3	4	5	10	11	12
Read			1	2	3	4	5	10	11
ALU				1	2	3	4	5	10
Mem					1	2	3	4	5
Write						1	2	3	4

Figure 3.5: Timing of Branches in a 6-Stage Pipeline

If a dedicated adder is available for branch target computation and branch instructions can be decoded quickly the approach shown in figures 3.6 and 3.7 can be used. In this case only one cycle is lost.

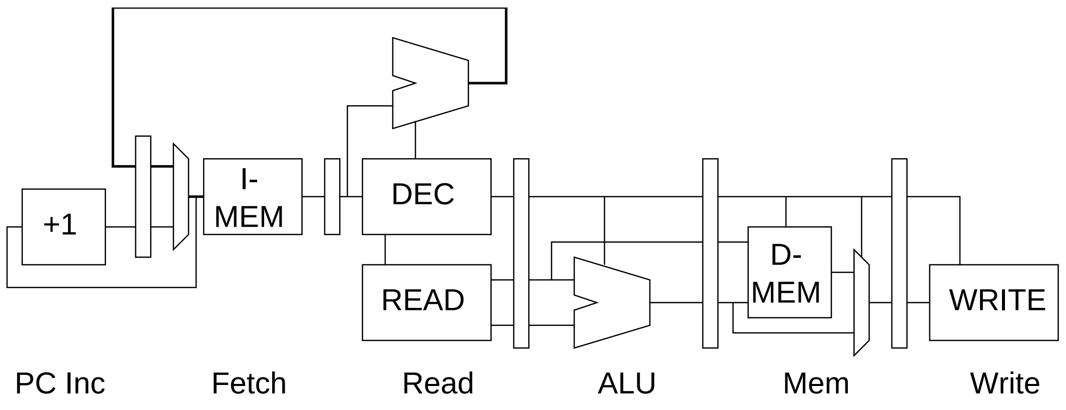


Figure 3.6: A Pipeline with a dedicated branch adder

New PC	1	2	3	4	10	11	12	13	14
Fetch		1	2	3	4	10	11	12	13
Read			1	2	3	4	10	11	12
ALU				1	2	3	4	10	11
Mem					1	2	3	4	10
Write						1	2	3	4

Figure 3.7: Timing of Branches with a dedicated branch adder

Two principle techniques may be used to reduce the impact of this effect on the pipeline's performance. Firstly branch delay slots can be used. In this system the instructions in the locations following the branch instructions are fetched and executed irrespective of whether the branch is taken. The compiler has to find instructions that can be placed here to do useful work. [HENN90] claims that it is able to do so usefully about half of the time for the first slot and significantly less of the time for any second or subsequent slots (see appendix A).

The second approach is to use branch prediction. In this system the branch target is guessed and instructions from the target are fetched and executed. The true branch target is computed concurrently. If the branch turns out to have been wrongly predicted the speculatively fetched instructions are cancelled. The prediction scheme may be a simple one such as "predict not taken" where instructions are always fetched from beyond the branch target or a more complex dynamic scheme based on the instruction's previous behaviour. The effectiveness of branch prediction depends on the complexity of the branch history mechanism but correct prediction rates of around 90% have been reported [HENN90].

3.5.2 Dependencies

Consider the following instruction sequence:

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

Note that the second instruction uses the result of the first instruction as an operand. This is referred to as a “read after write” or RAW dependency.

In the simple pipeline shown in figure 3.2 this instruction sequence would execute wrongly; when the second instruction read R1 from the register bank it would read the value of R1 existing before the first instruction executed.

There are three solutions to this problem:

- The anomalous behaviour described here could be defined to be the correct behaviour and the problem of ensuring that the correct value is ready be assigned to the compiler.
- The dependency could be detected at the register read stage by means of some sort of locking mechanism and the second instruction could be stalled until the first had completed.
- The dependency could be detected at the register read stage and a mechanism could be used to “forward” the result from the first instruction directly to the second.

In practice only the third of these techniques is used in synchronous processors with this type of pipeline; the others are mentioned here because they are of interest later in the cases of asynchronous and superscalar processors.

The necessary forwarding busses for this pipeline are shown in figure 3.8. In terms of control, the processor has to record the destination of each previous instruction that is still in the pipeline (two in this case). These destination register numbers are compared with the source register numbers of the instruction being issued. Each possible match activates one of the forwarding paths (figure 3.9).

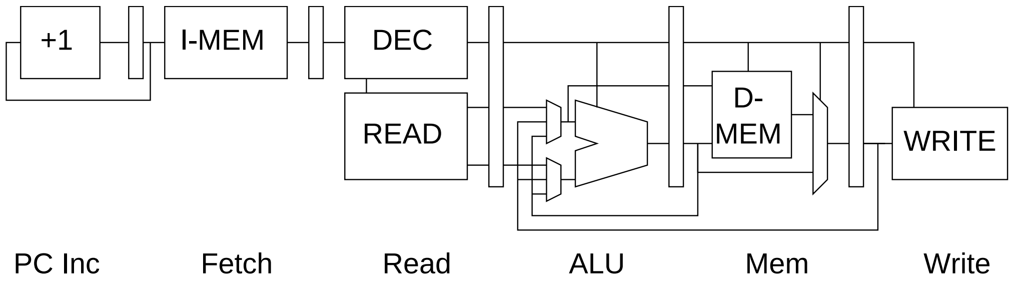


Figure 3.8: A 6-stage Pipeline with Forwarding

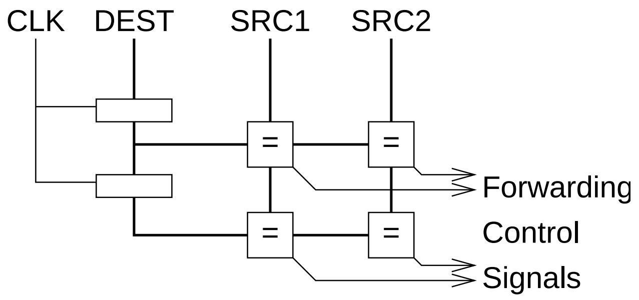


Figure 3.9: Control for Forwarding

Note that when the result of a load is used by the following instruction forwarding is not possible; in this case the pipeline must be stalled for one cycle or the next instruction must be prohibited from using this register.

With the exception of the load instruction, in this simple synchronous pipeline the addition of forwarding paths means that RAW dependencies have no effect on performance. This is in contrast to superscalar processors, described in the next section, and most existing asynchronous processors.

The effective parallelism obtained in the 6 stage pipeline with each of these factors taken into account is shown in table 3.3[1].

	Execution Cycles	Relative Parallelism
Non-pipelined processor	473	1.00
Perfect pipeline, no stalls	100	4.73
As above but stalling when a load result is used by the following instruction (23 % of loads)	104	4.54
As above and allowing for stalls due to branches:		
Single adder	132	3.58
Dedicated branch adder	118	4.00
Dedicated branch adder and branch delay slot, filled usefully 48 % of the time	111	4.24
Dedicated adder and branch prediction, 90 % of predictions correct, 90 % of branches cached	109	4.34

Table 3.3: Effective parallelism in a 6 stage pipeline

Table 3.3 shows that the greatest parallelism achievable with the pipeline described is 4.34; this could be used to give a power saving of about 19 times compared with the non-pipelined processor.

1. The instruction mix used is shown in table 3.4. This and other statistics from [HENN90] (see appendix A).

Type	Count
Branch	14
Load	18
Store	8
Compare	3
Others	57
Total	100

Table 3.4: Instruction Mix

3.5.3 Different Pipeline Arrangements

Shorter Pipelines

Shorter pipelines are worthwhile when some operations take much longer than others. The ARM2 processor was designed for use without a cache memory and its cycle time matches the dynamic RAM to which it was interfaced. The cycle time of this RAM is relatively long so the processor merges the register read, ALU and register write operations into one pipeline stage. As a consequence there is no need for any forwarding mechanism.

Longer Pipelines

When the desired performance or power efficiency exceeds that which can be achieved with the type of pipeline described above it is necessary to consider using a longer pipeline where each stage performs a simpler operation. To build a longer pipeline it is necessary to find a way of subdividing the internal structure of the stage that limits overall pipeline throughput. This probably means either the memory access cycles (instruction fetch and load/store) and the ALU.

Subdivision of these stages is likely to be difficult and costly in terms of area. More importantly the longer pipeline will worsen the problems caused by branches and dependencies; for example if the ALU is pipelined then forwarding is not possible between one instruction and the next so the pipeline will have to stall more often.

At this point the designer must consider other forms of parallelism. One possibility is superscalar execution which is considered next.

3.6 Superscalar Parallelism

The evolution of conventional RISC processors progressed from pipelined processors to so-called Superscalar processors. Superscalar processors contain multiple parallel pipelined datapaths sharing a single global register bank. They aim to fetch several instructions at a time from memory and issue one to each of the pipelines. They do this in a way that does not change the programmer’s model of the processor: the result of the program must be the same as would be obtained with a single pipeline. Interactions between instructions must be detected and dealt with by the hardware.

There are many ways in which superscalar processors can be organised. One structure is shown in figure 3.10. This is a symmetrical arrangement where two pipelines of the sort shown in figure 3.2 are combined. A global register bank is shared by the two pipelines; it has 2 write ports and 4 read ports.

Ideal code could execute on this processor with twice the degree of parallelism that it would obtain on a single pipeline machine. This parallelism could be used to provide a four-fold power saving. Unfortunately real code does not match this ideal.

The realisable parallelism in a superscalar processor is limited by the same two factors that limit pipeline performance: branches and dependencies.

This section focuses on how these two problems affect this processor and how their effects can be mitigated. The answer is that good performance is possible but complex logic is required to control how and when instructions are issued.

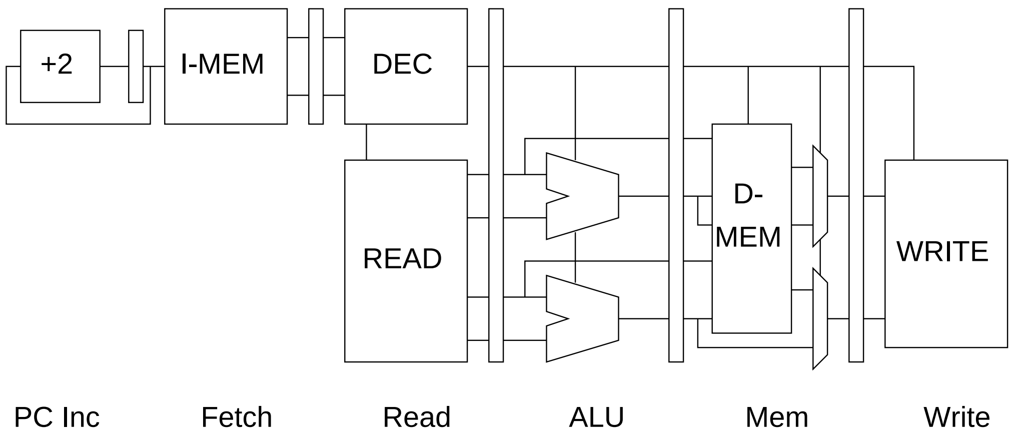


Figure 3.10: Symmetric 2-way superscalar processor

3.6.1 Dependencies

As in the pipelined processor forwarding mechanisms can be used to move results within and between the pipelines; an arrangement of forwarding paths for the processor of figure 3.10 is shown in figure 3.11.

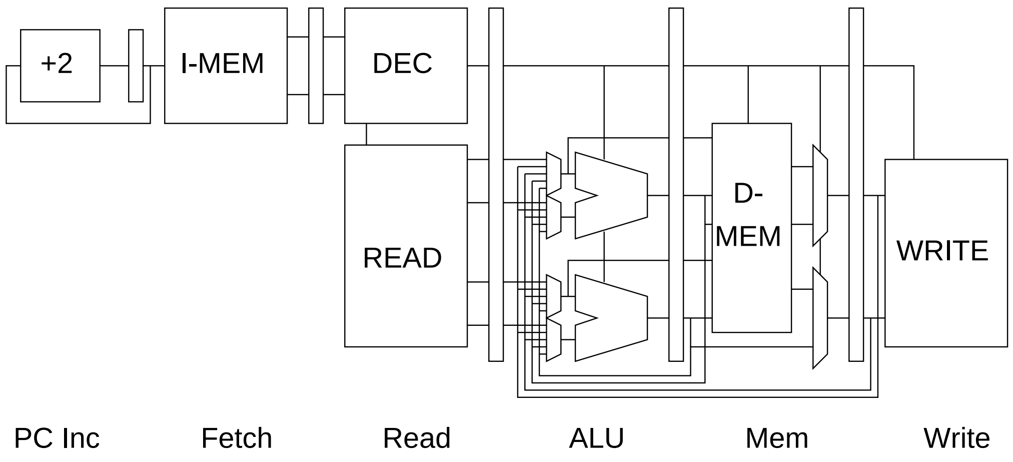


Figure 3.11: Forwarding in a Superscalar Processor

Because of the increased complexity of the processor the number of forwarding paths is increased. Each ALU input must select between the register bank output and four forwarding paths. To control these forwarding paths the destination of the last two instructions in each pipeline must be stored and compared with each operand of each instruction being issued. This requires 16 register specifier comparators compared with the 4 required in the single pipeline.

Unfortunately despite the increased complexity of the forwarding logic, whereas in the pipelined processor the forwarding paths solved nearly all RAW dependencies, in the superscalar processor many dependencies have to lead to pipeline stalls. Consider this sequence:

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

Because of the RAW dependency between these instructions they cannot be issued at the same time. The first may be issued but the second cannot be issued until the next cycle[1]. Detecting these dependencies requires that the destination register of the first instruction in the decoder is compared with the source registers of the second instruction. Any match indicates that the pipeline must be stalled.

This restriction significantly limits the effective parallelism of a superscalar processor of this type. This limit may be overcome but it requires yet more complexity in the instruction decoding and issuing. One technique that may be used is out of order issue. Consider this sequence:

```
R1 := R2 + R3
R4 := R1 + R5
R6 := R7 + R8
R9 := R6 + R10
```

Here the second instruction depends on the first instruction and the fourth instruction depends on the third instruction. In order to fully occupy two pipelines these instructions

1. If the ALU is not the processor’s critical path, then it may be possible for results computed by an ALU in one pipeline to be forwarded to another pipeline to be used in the same cycle. This technique is applied in the Sun SuperSPARC processor [SUN92] but is not considered further here.

can be issued out of order: in the first cycle the first and third instructions are issued, and in the second cycle the second and fourth instructions are issued.

By using out of order issue the utilisation of the available parallelism in a superscalar processor is significantly increased. Unfortunately another form of dependency introduces another limit. The following code sequence illustrates a write after read (WAR) dependency:

```
R1 := R2 + R3
R4 := R1 + R5
R5 := R7 + R8
R9 := R5 + R10
```

The first and second instructions cannot be issued together because of a RAW dependency between them. However the first and third instructions are independent and so it would be desirable to issue them together. Unfortunately the third instruction modifies R5, one of the registers that is used as an operand by the second instruction, and so it cannot be issued before it. This is a WAR dependency.

WAR dependencies can be avoided using a technique called “register renaming”. Register numbers from instructions are considered as “logical” register numbers and are mapped to the “physical” registers during decoding. In this code sequence, the references to R5 in the third and fourth instructions are made to a “renamed” version of R5 which is actually a different physical register:

```
R1  := R2  + R3
R4  := R1  + R5
R5a := R7  + R8
R9  := R5a + R10
```

With this re-arrangement the first and third instructions can be issued together, and the second and fourth instructions can be issued together in the next cycle.

To support out of order issue and register renaming the organisation of the superscalar processor must be significantly changed. One possible organisation from [JOHN91] is shown in figure 3.12. The operation of this processor is as follows:

- Instructions are decoded in order and distributed to “reservation stations” (RS), one per functional unit. Each reservation station contains a number of entries storing an instruction and its operands.

- As each instruction is issued an empty entry is allocated in the reorder buffer (ROB) for its result. This entry contains the instruction’s destination register number and a “tag”.

- As instructions are placed into the reservation stations their source operands are fetched from the register bank (REG) or reorder buffer. If the registers concerned are subject to update by currently executing instructions then a tag is obtained rather than an actual operand value.

- As preceding instructions complete, each reservation station entry containing a tag compares its tag value with the tag value of the completing instruction. If the tags match the result value is copied into the reservation station.

- When an instruction in a reservation station has obtained all of its operands it is removed from the reservation station and put into the functional unit (FU) pipeline.

- Results are written into any matching reservation station entries and the reorder buffer.

- Reorder buffer entries are copied into the register bank in issue order. This is so that “in order state” is maintained in the register bank so that branch prediction failures and exceptions can be handled correctly.

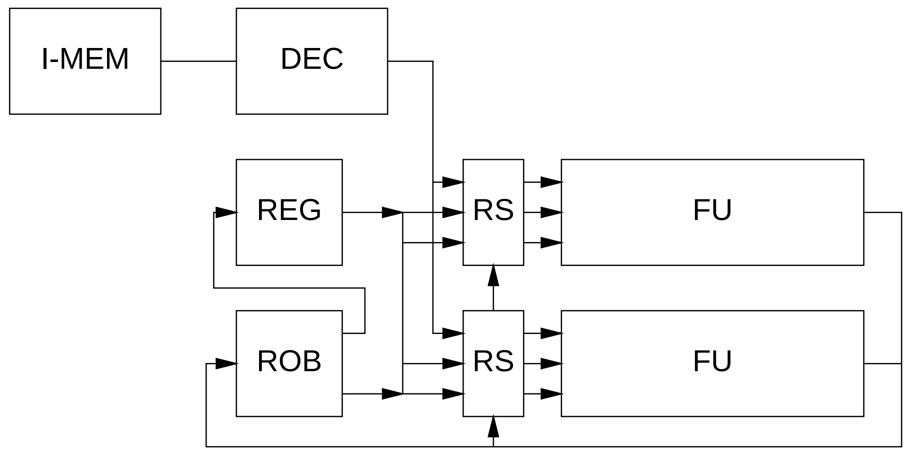


Figure 3.12: Superscalar Processor with Out of Order Issue and Register Renaming

3.6.2 Branch Instructions

The same fundamental problems affect branch instructions in a superscalar processor as in a pipelined processor: during the branch latency useful instructions cannot be fetched. In a superscalar processor this problem is multiplied: the number of cycles of the branch latency remains the same, but the number of potential instructions that have been lost grows.

Branch prediction may be used to ensure that useful work is done during the branch latency. When a branch has been mispredicted it is necessary to undo the effect of all speculatively executed instructions. This is one of the functions of the reorder buffer; results are not released from the reorder buffer to the register file until their execution has been confirmed.

3.6.3 Observations

[JOHN91] makes a very thorough study of the performance of superscalar processors for general purpose applications with and without various architectural features including out of order issue, register renaming, and branch prediction (see appendix A). Table 3.5 summarises his conclusions about the importance of the various features.

Feature	Performance advantage resulting from adding the given feature to a processor that already has all of the other features
Out of order issue	52 %
Register Renaming	36 %
Branch Prediction	30 %
Four instruction decoder (as opposed to a two instruction decoder)	18 %

Table 3.5: Performance Advantage of Major Processor Features (from [JOHN91])

It can be seen that in order to obtain reasonable parallelism from a superscalar processor it is necessary to implement most or all of these complex features; without them the potential parallelism simply is not achieved. Yet their implementation is complex and consequently highly power consuming.

It is interesting to quote Johnson's observations about the complexity of superscalar processor control logic:

> *" 9.1.4 The Painful Truth*
>
> *For brevity, we have examined only a small portion of the superscalar hardware: the portion of the decoder and issue logic that deals with renaming and forwarding. This is a very small part of the overall operation of the processor. We have not begun to consider the algorithms for maintaining branch-prediction information in the instruction cache and its effect on cache reload, the mechanisms for executing branches and checking predictions, the precise details of recovery and restart, and so on. But there is no point in belaboring the implementation details. We have seen quite enough evidence that the implementation is far from simple.*
>
> *Just the hardware presented in this section requires 64 5-bit comparators in the reorder buffer for renaming operands: 32 4-bit comparators in the*

reservation stations for associatively writing results: 60 4-bit comparators in the reservation stations for forwarding: logic for allocating result tags and re-order-buffer entries: and a reorder buffer with 4 read ports and 2 write ports on each entry, with 4 additional write ports on portions of each entry for writing register identifiers, results tags, and instruction status. The complexity of this hardware is set by the number of uncompleted instructions permitted, the width of the decoder, the requirement to restart after a mispredicted branch, and the requirement to forward results to waiting instructions.

If the trend in microprocessors is toward simplicity, we are certainly bucking that trend. ”

3.6.4 Simpler forms of Multiple Functional Unit Parallelism

The preceding section shows that the parallelism obtainable from superscalar execution is desirable but its cost is too high. This section considers ways in which the superscalar control problem can be simplified through changes to the instruction set architecture.

The role of the instruction decoding and issuing logic and the reservation stations in a superscalar processor can be thought of as translation. In the input language the flow of data between instructions is described by means of register identifiers; in the output the same information is described by renamed register specifiers and forwarding control information. It is interesting to consider encoding some form of this translated information directly into the instructions.

There are a number of problems with this idea. If the renamed registers and forwarding information are encoded directly it is not possible to use the same code on processors with different organisations. It should also be noted that the output of the translation process in the conventional processor is not a function of only the static program but also its dynamic behaviour: a particular sequence of instructions may be translated into different forwarding information depending on which branch instruction lead to its execution.

Despite these problems it is possible to find compromise encodings that require less processing by the decoder and issuer than the conventional system using register numbers alone. SCALP uses such a technique called explicit forwarding which is described in chapter 5.

An alternative method for simplifying the superscalar organisation is to adopt the VLIW (very long instruction word) approach where each instruction specifies one operation for each of the functional units. This technique has not been considered for SCALP because it leads to a substantial decrease in code density; whenever the compiler is unable to find an operation for a functional unit that part of this instruction is unused. Chapter 2 explained that code density is important to SCALP for low power operation.

3.7 Speculative Execution

The preceding sections have mentioned the importance of branch prediction to obtain high levels of parallelism in pipelined and superscalar processors. The parallelism obtained through branch predication can be used to increase power efficiency.

On the other hand it can be argued that when branch prediction is wrong energy has been wasted in partially executing the wrongly predicted instructions. To obtain the maximum power efficiency there must be a balance between these two factors.

This is an example of a general question concerning the balance between speculation and concurrency. Another example occurs in the design of an equality comparator: the minimum number of transitions occur if the comparator operates serially, stopping as soon as a mismatch is detected. On the other hand a parallel implementation that compares all bits simultaneously will operate more quickly allowing for lower voltage operation.

Whether branch prediction is power efficient can be computed as shown in table 3.6[1]. This table relates to the pipelined scheme described in section 3.5.

1. Note that this excludes the power costs of implementing the branch prediction function itself.

	With Branch Prediction	Without Branch Prediction
Relative parallelism	4.34	4.00
Power due to parallelism	0.053	0.063
Relative throughput	1.03	1.00
Power due to throughput	1.03	1.00
Overall Power	0.055	0.063

Table 3.6: Branch Prediction and Power Efficiency

It can be seen that the increased parallelism due to branch prediction leads to a power saving of 16 %, and on the other hand the power wasted on speculatively executed instructions amounts to 3 %. There is therefore a net power saving of 13 % due to branch prediction.

Chapter 4: Asynchronous Logic

The previous chapter has studied the organisation of superscalar pipelined synchronous processors. This chapter considers if and how this type of processor can be implemented using asynchronous logic. The first section explains the motivation for using asynchronous logic in low power systems. Subsequent sections consider the implementation of pipelines and superscalar parallelism using asynchronous logic, and considers ways in which instruction set features can make implementation simpler. The final section reviews previous asynchronous processors.

4.1 Asynchronous Logic and Power Efficiency

The SCALP processor uses asynchronous logic because this approach is believed to have benefits in terms of power efficiency and other factors. This section explains these benefits.

4.1.1 Asynchronous Circuits can be Faster

All systems, both synchronous and asynchronous, can be divided into regions of combinational logic separated by storage. In the two cases the speed of the combinational logic and of the storage elements will be approximately equal. The claim for increased performance for asynchronous systems comes from the observation that in synchronous systems the speed of the global clock must match the speed of the slowest logic block with its slowest input data. The consequence is that faster blocks will be idle for part of the clock cycle.

This is illustrated in the upper part of figure 4.1. The light areas represent processing delays in logic blocks; the dark areas represent storage delays. The cycle time of the global clock signal is set according to the slowest logic block. For the other blocks there is an idle period shown in white.

In an asynchronous system there is no global clock; local request and acknowledge signals indicate when a result is ready and it may advance immediately. This is shown in the lower part of figure 4.1; there are no idle periods between cycles. Consequently the overall speed of the system is increased[1].

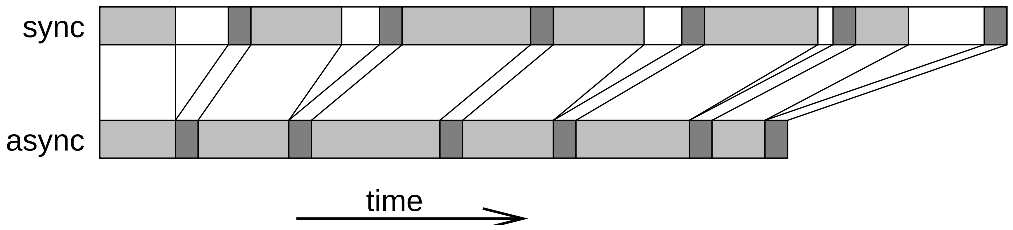


Figure 4.1: Speed of Synchronous and Asynchronous Circuits

This increase is especially significant when the speed of the logic is data dependent; the clock speed in a synchronous system would have to allow for the worst case data which may be very rare.

If asynchronous circuits can be faster than equivalent synchronous circuits then a power saving can be achieved; by adjusting the supply voltage the circuits can be made to operate at the same speed, but because of the reduced supply voltage the asynchronous circuit will do so using less power.

1. Asynchronous pipelines are subject to problems of starvation and blocking which can cause blocks to be idle at some points. This proviso is discussed in section 4.2.

4.1.2 Asynchronous Circuits can be Smaller

In some circumstances asynchronous circuits may be smaller than equivalent synchronous circuits. By reducing the size of the circuit a power saving can be made as fewer wires of smaller capacitance must be charged and discharged for each operation.

This type of advantage is best demonstrated by two datapath elements, adders and multipliers. In the case of an adder the main design problem is dealing with the carry signals. With a simple ripple carry design in the worst case the carry has to propagate across the whole width of the adder, though in typical cases it only needs to propagate for a few bits. In an asynchronous system this typical case delay is the important measure and the relatively small ripple carry adder is satisfactory. In a synchronous environment on the other hand it is the worst case delay that is important, and adder designs such as carry select adders are used. These designs provide better performance in the rare worst case but at a significant cost in terms of area. [GARS93] compares the ALUs from the synchronous ARM6 processor and the asynchronous AMULET1 processor. The ARM6 carry select adder is 2.5 times larger but operates at about the same speed for typical operands.

Asynchronous multipliers can be smaller than synchronous multipliers for a different reason. Multiplication using carry save adders is an example of an algorithm that performs a simple step a number of times in order to compute its result. In an asynchronous multiplier the simple carry save adder logic is exercised as many times as is necessary at high speed by local request and acknowledge signals. In a synchronous design on the other hand the global clock speed is typically limited by some other part of the design, and is many times slower than the carry save adders. In order to maintain a reasonable speed the synchronous designer makes a number of copies of the adders so that several stages of the computation can take place within one cycle. The result is a circuit that is no faster than the asynchronous design but is much larger. The increased size means that wires both within the multiplier and outside it will be longer and hence consume more power when they transition.

These examples of the area benefit of asynchronous logic must be balanced against any change in the the complexity and size of asynchronous control structures compared with equivalent synchronous circuits.

4.1.3 Asynchronous Logic in Variable Demand Systems

Perhaps the strongest case for the power efficiency benefit of asynchronous logic comes from the study of systems with variable processing demand. This sort of system is very common; most computers that operate interactively experience considerable and rapid fluctuations in processing demand as the user performs different tasks. In embedded applications it is also common; processors are activated in response to particular events, operate for a short time and are then suspended.

Synchronous processors take a coarse-grained approach to power saving in this situation. Portable computers for example monitor user activity and enter various types of sleep mode by slowing or stopping clocks as activity slows. They use this coarse-grained approach because switching between clocks is a complex procedure, and typically some interaction with the system software is needed.

In asynchronous systems the behaviour is quite different. In the absence of demand for processing, none of the request or acknowledge signals will be asserted. Consequently there is no circuit activity and no power is consumed. This provides an instantaneous response to changes in the computational demand and consequently a greater power saving than can be obtained with a synchronous system.

Figure 4.2 illustrates this. The lower line indicates processor activity. In an asynchronous implementation power consumption would be proportional to this activity level. In a synchronous implementation the clock speed would be adjusted more coarsely as shown by the upper line and the power consumption would be greater. The area between the two lines represents the power saving that the asynchronous processor would obtain.

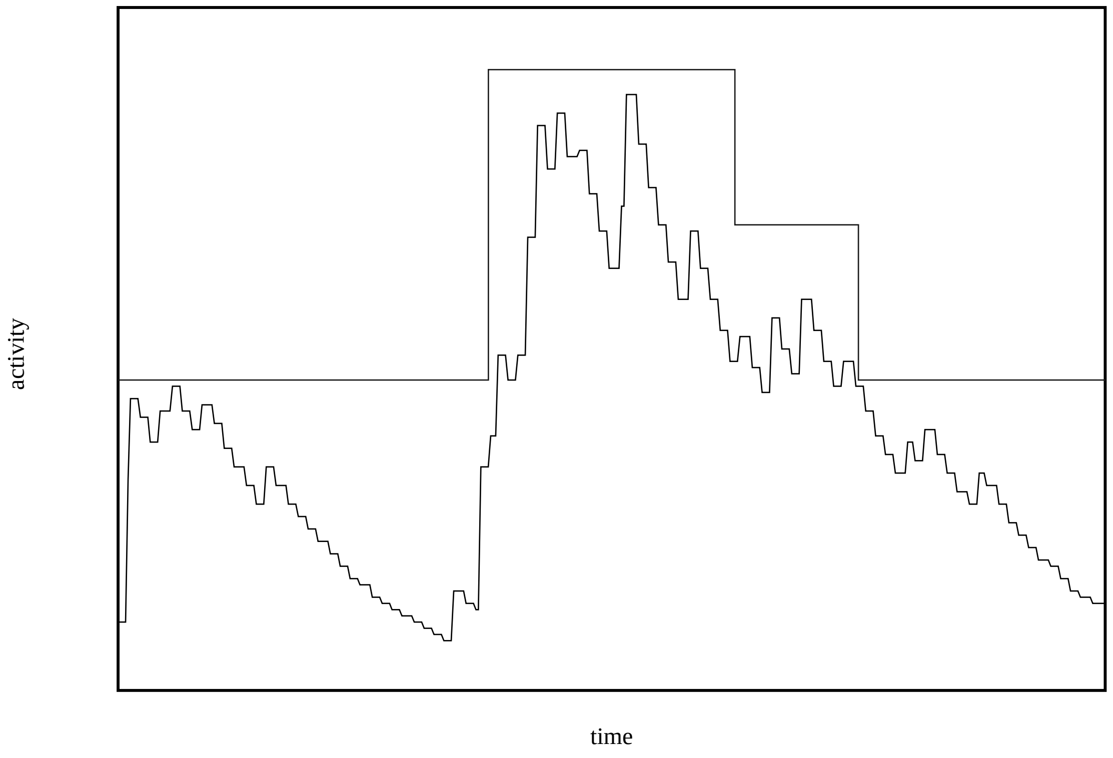


Figure 4.2: Power Saving in Variable Demand Systems

4.1.4 Asynchronous Logic and Dynamic Supply Voltage Adjustment

Greater power savings can be obtained in a variable demand system if the system supply voltage is reduced when activity is low. In synchronous systems as with clock speed adjustment this approach can again only be applied at a coarse level, and the designer has to ensure that the clock rate and the supply voltage maintain a safe relationship. In an asynchronous system on the other hand feedback techniques can be used to continuously maintain the supply voltage at the minimum necessary level.

One possible scheme for digital signal processing applications is described in [NIEL94] and is shown here in figure 4.3. In this example incoming data is placed into a fifo. The occupancy of the fifo is measured and is used to set the supply voltage; as the fifo fills the supply voltage is increased so the processor operates faster, emptying the fifo.

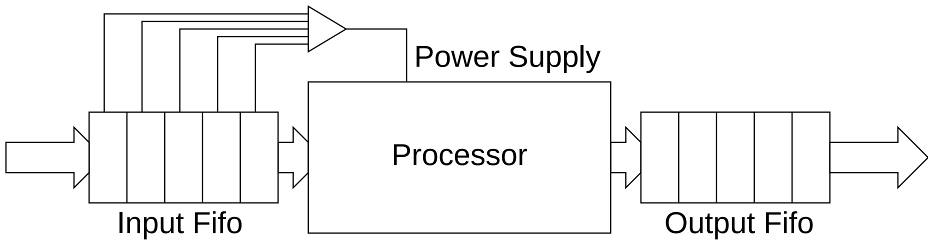


Figure 4.3: Dynamic Supply Voltage Adjustment controlled by Fifo Occupancy

Another scheme suitable for a general purpose processor such as SCALP is shown in figure 4.4. This case relies on the program executing a HALT instruction when it has no useful work to do; in the case of SCALP, HALT means “wait for next interrupt”. When the processor is halted it asserts an output. This output is integrated to derive an analogue level indicating the system load. This is used to adjust the supply voltage.

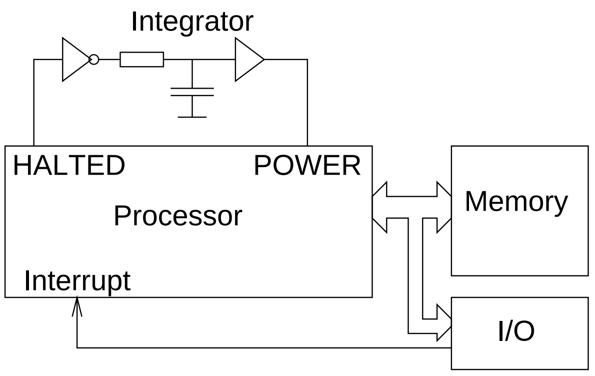


Figure 4.4: Dynamic Supply Voltage Adjustment controlled by a HALT instruction

4.1.5 Asynchronous Logic Doesn’t Waste Transitions

A common mode of operation for a synchronous circuit is as follows: in response to a clock edge, a number of signals change. As these changes propagate through the circuit outputs change possibly several times before reaching a stable value. These glitches do not cause the circuit to fail because the next piece of logic will not sample their value until the next clock edge; however every transition, useful or not, consumes power.

In an asynchronous circuit[1] on the other hand every transition has a meaning. Any spurious transitions caused by hazards will cause the circuit to malfunction, so circuits must be designed to not create glitches. Consequently the asynchronous circuit may make fewer transitions and hence consume less power than an equivalent synchronous circuit.

On the other hand it can be suggested that because these asynchronous circuits must be hazard free they will be more complex, which will in turn lead to greater power consumption. These two factors must be balanced for each particular application.

4.1.6 Other benefits of Asynchronous Logic

As well as the power efficiency issues discussed above there are other benefits of asynchronous logic that motivated its use in SCALP:

- Asynchronous systems lend themselves better to the application of formal techniques for verification or synthesis. Although these techniques are not yet mature they offer the prospect of more rigourous or rapid validation of complex designs in the future.

- The nature of the interfaces between blocks in asynchronous systems means that they can be designed in a very modular fashion. In a synchronous system the presence of a global clock speed requirement means that existing components may not be reusable if their clock speeds do not match. In an asynchronous system this restriction does not apply and the designer is free to exchange modules for other blocks with the same functional characteristics irrespective of their timing properties.

1. This applies to many though not all classes of asynchronous circuits; see section 6.1. Specifically it applies to delay insensitive and speed independent circuits but may not apply to bounded delay circuits.

4.1.7 Tangram and the Philips DCC Error Corrector

As an example of the potential power efficiency benefits of asynchronous logic, this section describes work carried out by Philips Research Laboratories using the VLSI Programming language Tangram. Tangram is an Occam-like high level language which is compiled into asynchronous circuits. The mathematical approach used makes formal reasoning about the circuits for purposes such as testability possible, and its high level nature allows designers to concentrate on architectural factors affecting the power efficiency.

Recent publications [KESS95] [BERK95] describe the implementation of an error detection circuit for a digital compact cassette (DCC) player. The nature of the application requires a constant data throughput, though the work required for each block of data is variable. The algorithm has two parts:

(i) error detection, which is applied to all input data,

(ii) generation of error correction information, which is only necessary when part (i) detects errors.

The solution used is as follows: the chip has two power inputs, one providing 1.5 V and the other 5 V. During part (i) the 1.5 V supply is used. During part (ii) if necessary the 5 V supply is used. In this way the system uses the lowest possible supply voltage. Interestingly no communication is required with the environment during part (ii), eliminating the need for voltage level shifters at the interface.

A synchronous implementation of the part already existed which uses a fixed 3 MHz clock which is gated to some parts of the chip when not needed and a fixed 5 V supply. In comparison the Tangram design is 20 % larger (due to the increased size of the control logic components) but gains a factor 5 power efficiency improvement through the use of asynchronous logic and a further factor of 20 through the use of dynamic supply voltage scaling.

Though the DCC chip is not a general purpose processor the techniques that it uses could be applied to one, with substantial power efficiency benefitis.

4.1.8 Conclusion

This section has presented a number of arguments suggesting that the use of asynchronous logic can provide power efficiency improvements, and mentioned a recent real design that seems to justify them.

For the SCALP design, the use of asynchronous logic is more important than just an implementation technology. Because SCALP is an entirely new design, the use of asynchronous logic can be taken into consideration throughout the design process including in the design of the instruction set. This issue is considered further in the following sections.

4.2 Asynchronous Pipelines

Figure 4.5 shows the familiar structure of a synchronous pipeline with latches (L) and combinational logic blocks (CL). All latches are controlled by a single global clock signal and so operate simultaneously.

This form of pipeline is readily implemented asynchronously as shown in figures 4.6 and 4.7. The latches (L) and combinational logic blocks (CL) are the same as in the synchronous pipeline but the timing is controlled quite differently. Each latch has an associated latch control circuit (LC). The latch control circuit opens and closes the latch in response to request (Req) signals from the previous stage and acknowledge (Ack) signals from the following stage.

The request signal from the latch control circuit must be delayed by an amount greater than the corresponding data delay in the combinational logic. This may be done either using a matched path delay element (D) as shown in figure 4.6 or by using some form of completion detection as shown in figure 4.7.

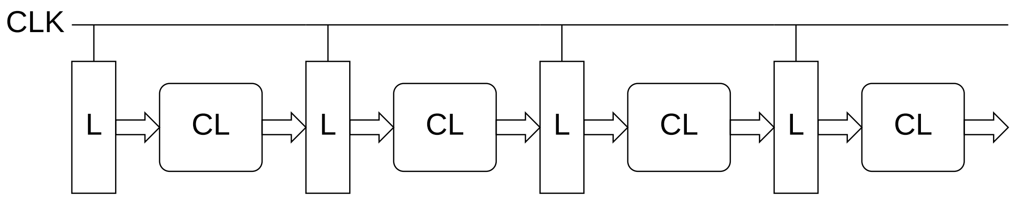


Figure 4.5: A Simple Synchronous Pipeline

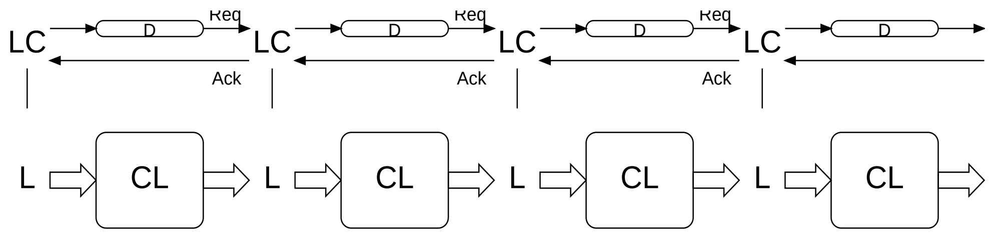


Figure 4.6: A Simple Asynchronous Pipeline with Matched Delays

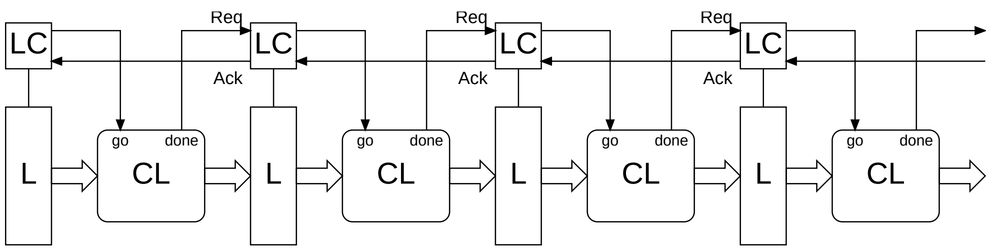


Figure 4.7: A Simple Asynchronous Pipeline with Completion Detection

The performance advantage of the asynchronous pipeline was mentioned in section 4.1.1 and is summarised in table 4.1. Note however that table 4.1 gives only an upper bound on the possible performance of asynchronous pipelines; performance can be decreased by starvation and blocking [KEAR95]. A pipeline stage is said to be starved if it cannot start an operation because it is waiting for the result of the previous stage to be available. Similarly a stage is said to be blocked if it cannot transfer its result to the next stage because that stage is still processing the result of the previous operation. Starvation and blocking are less significant in pipelines with small variations among the possible stage delays.

The effect of starvation and blocking on performance can be reduced by introducing decoupling queues between the stages. When one stage has to perform a particularly slow operation the queue preceding it will fill, preventing the previous stage from blocking. At the same time the queue following it will empty preventing the following stage from starving.

The performance of an asynchronous pipeline may also be reduced by delays introduced by the latch control circuit.

	Latency determined by:	Throughput determined by:
Synchronous pipeline	Slowest block, worst data, worst conditions	Slowest block, worst data, worst conditions
Asynchronous pipeline with matched delays	Typical block, worst data, typical conditions	Slowest block, worst data, typical conditions
Asynchronous pipeline with completion detection	Typical block, typical data, typical conditions	Slowest block, typical da-ta, typical conditions

Table 4.1: Performance Of Synchronous and Asynchronous Pipelines

4.2.1 Forwarding in Asynchronous Pipelines

The simple linear pipelines described above benefit from asynchronous implementation. Unfortunately the presence of forwarding paths as described in section 3.5.2 in processor pipelines makes efficient asynchronous implementation more difficult.

In a synchronous pipeline forwarding is simple to implement as shown in figure 4.8. Stage 2 uses as input the results of stages 1 and 3. Because all stages use a single clock these inputs are available at the same time.

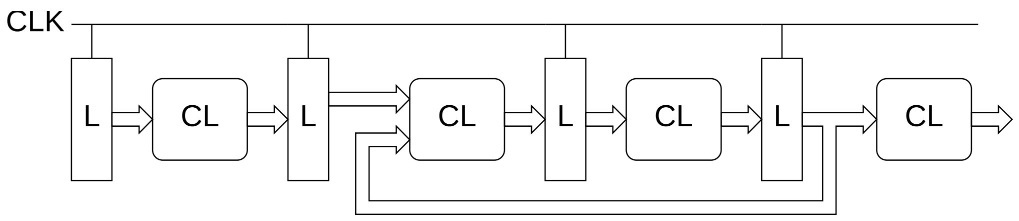


Figure 4.8: A Synchronous Pipeline with Forwarding

Figure 4.9 shows how this can be implemented asynchronously[1]. The datapath is the same as in the synchronous pipeline; however the control is modified so that a request is sent to stage 2 when data from stages 1 and 3 are both available. A Muller C element[2] is used to perform this synchronisation. Similarly the acknowledge signal from stage 2 is sent to both stages 1 and 3. Stage 3 waits for acknowledgements from stages 2 and 4 using a second Muller C element.

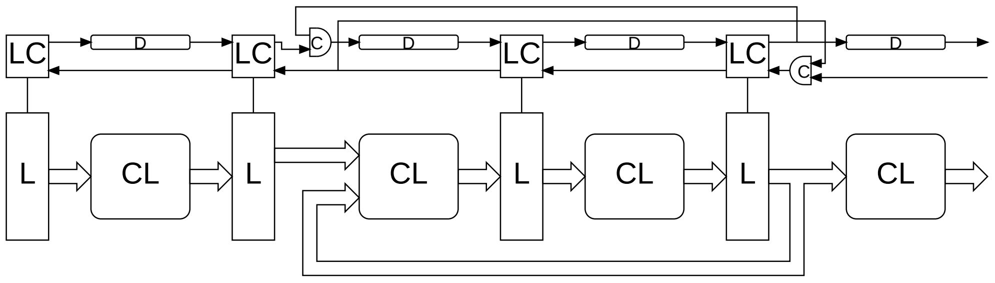


Figure 4.9: An Asynchronous Pipeline with Forwarding

1. Some additional logic is required to ensure correct initialisation.

2. A Muller C element behaves as follows: when both inputs are high, the output becomes high. When both inputs are low, the output becomes low. When the inputs differ the output retains its previous level.

The additional timing constraints in this circuit make it adopt a form of locally synchronous behaviour, constraining the progress of instructions along the pipeline. This leads to a reduction in performance compared with the simple asynchronous pipeline.

The extent of this performance reduction has been found for an example pipeline by simulation. The results are shown in figure 4.10. The figure shows performance for synchronous and asynchronous pipelines with and without forwarding. The pipeline concerned is a four stage pipeline with random stage delays of 1 [PLUSMINUS] v time units where v is the variance shown on the x axis. The y axis shows performance relative to the synchronous implementation.

The top line shows the performance of a perfect asynchronous pipeline. This perfect performance is not achieved because of starvation and blocking within the pipeline. The second line indicates the performance of a pipeline allowing for starvation and blocking. The third line indicates the performance of a pipeline with an unconditional forwarding path between the third and second stages.

When the variance is 0.2 (for example stage delays are equiprobably 0.8 and 1.2 time units) the perfect asynchronous pipeline’s throughput is 20 % higher than the synchronous pipeline. Allowing for starvation and blocking this advantage is reduced to 14 %. When the forwarding path is introduced it is further reduced to only 6 %.

4.2.2 Conditional Forwarding

Operands provided via forwarding paths are not always used by the stages to which they are sent. Typically the receiving stage uses a multiplexor to select between the forwarded value and a value received from the previous stage. This multiplexor is controlled by a bit generated during instruction decode that indicates whether forwarding should be used (see figures 3.8 and 3.9).

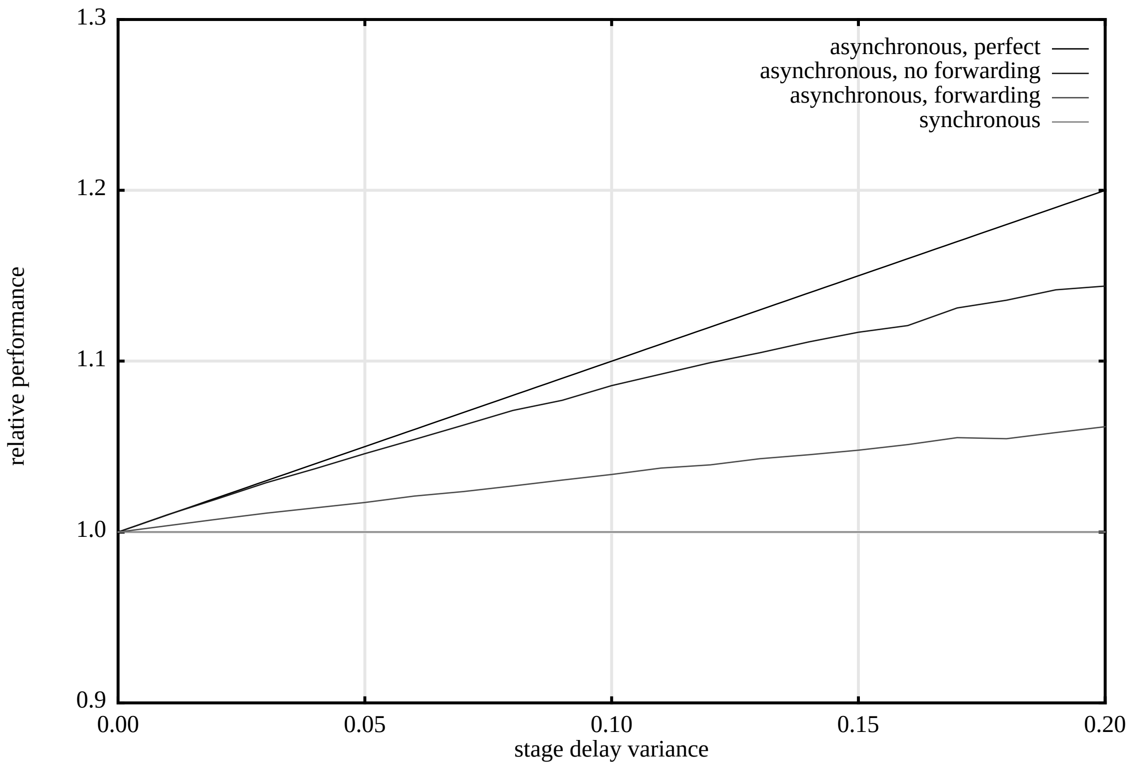


Figure 4.10: Performance of Synchronous and Asynchronous Pipelines with and without Forwarding

In principle this conditional nature of forwarding could be used to improve the performance of asynchronous forwarding pipelines. When forwarding is not to be used no additional synchronisation is carried out and performance is similar to that of a simple asynchronous pipeline. When forwarding is required temporary additional synchronisation can be introduced to allow the forwarded data to be transferred.

Such a scheme could be implemented as shown in figure 4.11: the instruction decode generates two bits for each instruction, one (USE_FWD) indicating that the instruction must use a forwarded value produced by a preceding instruction and another (GEN_FWD) indicating that the instruction must send its result to be used by a following instruction. At the pipeline stage where the forwarded data is used a “conditional pipeline merge” (CPM) element synchronises with forwarded data only when the USE_FWD bit is asserted. At the pipeline stage where the forwarded value is created a “conditional pipeline fork” (CPF) element sends the forwarded value only when the GEN_FWD bit is asserted.

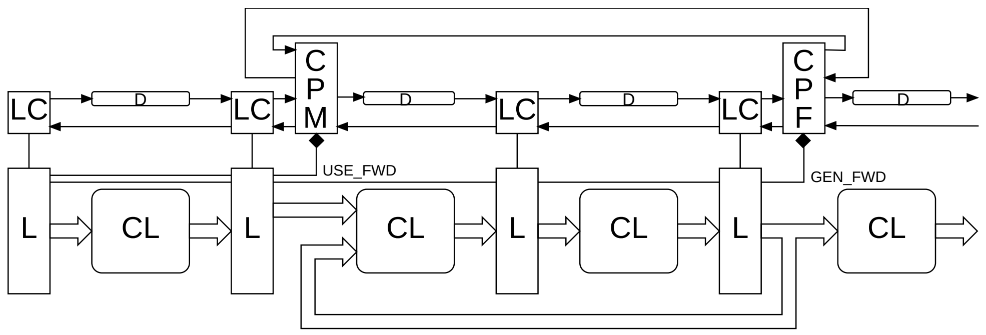


Figure 4.11: An Asynchronous Pipeline with Conditional Forwarding

The flaw with this proposal is that with conventional instruction sets it is impossible for the instruction decoder to generate the GEN_FWD bit. The USE_FWD bit can be generated as shown in figure 3.9, but to generate the GEN_FWD bit it would be necessary to know the operand registers of future instructions before they are issued.

4.2.3 Explicit Forwarding

One solution to this problem is to change the nature of the instruction set so that the need to forward a value can be detected by the instruction decoder. This may be done by giving more information about the way in which a result will be used in the instruction that produces that result. This technique is referred to here as “explicit forwarding”.

One example of explicit forwarding is the scheme described in section 2.1.2 for using shorthand to transfer values between adjacent instructions without specifying register numbers.

Other systems can take the idea further. SCALP entirely replaces the conventional idea of accessing operands by means of register numbers with a system based on explicit forwarding. This is described further in chapter 5.

4.3 Asynchronous Superscalar Parallelism

4.3.1 Simple Superscalar Structures

This section briefly considers the implementation and performance of general asynchronous superscalar structures in comparison with equivalent synchronous structures.

Figure 4.12 shows a synchronous parallel structure comprising an “issuer” (I) and a number of functional units (F1 to F3). Each of these blocks operates from the same global clock and so the performance of the system is limited by slowest block.

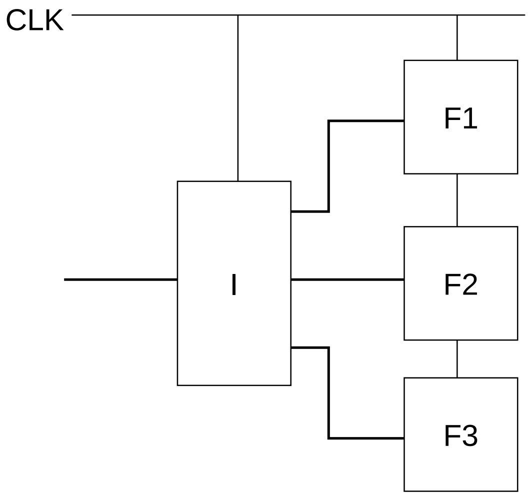


Figure 4.12: Synchronous Parallel Functional Units

Figure 4.13 shows an equivalent asynchronous structure. In this case there is no global clock and performance is determined by the typical block speed. In fact the advantage of this structure over the synchronous circuit is greater than the advantage of an asynchronous pipeline over a synchronous pipeline. Even in an asynchronous pipeline throughput is limited by the slowest block. With this parallel arrangement this is not the case and throughput as well as latency is determined by the typical block speed. Furthermore for this parallel structure there is no equivalent of the starvation and blocking problem that affects asynchronous pipelines. This is summarised in table 4.2 which can be compared with table 4.1 for pipelines.

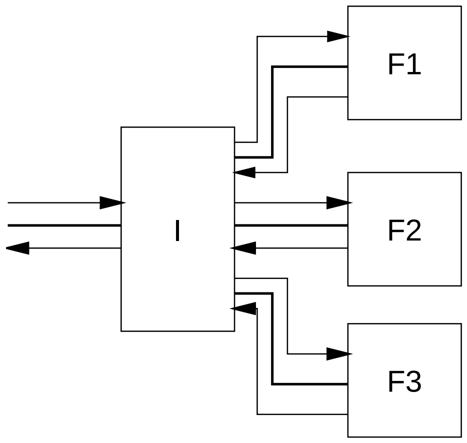


Figure 4.13: Asynchronous Parallel Functional Units

	Latency determined by:	Throughput determined by:
Synchronous superscalar processor	Slowest block, worst data, worst conditions	Slowest block, worst data, worst conditions
Asynchronous superscalar processor with matched delays	Typical block, worst data, typical conditions	Typical block, worst data, typical conditions
Asynchronous superscalar processor with completion detection	Typical block, typical data, typical conditions	Typical block, typical data, typical conditions

Table 4.2: Performance Of Synchronous and Asynchronous Superscalar Processors

The disadvantage of the asynchronous parallel implementation of figure 4.13 is that the issuer which distributes work to the various functional units must be more complex. In the synchronous implementation the issuer must make a decision about what to issue once per cycle; it studies the incoming instructions, looks at the state of the functional units and decides to issue one or two instructions. In the asynchronous implementation the task is more complex as the functional units can become ready to accept the next instruction at any time. Is is also possible that both units can become ready at the same time; in this case the issuer must arbitrate between them to decide which will receive the next instruction.

4.3.2 Superscalar Forwarding

As in the single asynchronous pipeline forwarding makes implementation more difficult, but in this case forwarding is required both within and between the pipelines. To provide unconditional forwarding between all necessary points virtually every stage in every functional unit has to be synchronised with every other, leading to global lockstep behaviour and substantially reduced performance.

As for pipelines instruction set extensions may be used to make conditional forwarding and conditional synchronisation possible. This will lead to more independence between the pipelines and better performance.

The idea of explicit forwarding can be extended to superscalar processors; each functional unit output is connected by a conditional forwarding path to each functional unit input. Instructions explicitly indicate to where their results must be sent to so that only the necessary forwarding paths need be activated.

Unfortunately the non-deterministic allocation of instructions to functional units makes this form of explicit forwarding impossible. If an instruction’s result is used by a particular following instruction then the first instruction must indicate that its result is sent to the functional unit that will execute the following instruction. That functional unit must therefore be known to the compiler.

This is not a problem when the functional units are asymmetric. If one functional unit executes loads and stores, another executes floating point instructions and another executes integer instructions, for example, then it is statically known which functional unit must execute each instruction, allowing explicit forwarding. When some instructions can be executed by more than one functional unit then the allocation must be known by the compiler. This can be achieved by performing the allocation of instructions to functional units in the compiler and encoding the appropriate functional unit identifier in the instruction.

This is essentially the system used by SCALP. Each instruction indicates the destination functional unit input to which its result should be sent. At the destination functional unit inputs forwarded data is paired with instructions arriving from the instruction issuer in much the same way that a reservation station matches instructions with data.

The SCALP architecture is described in the following chapter.

4.4 Previous Asynchronous Processors

A number of asynchronous microprocessors have been proposed or built recently. This section describes the architecture and asynchronous design style of each[1]. Table 4.3 summarises these characteristics.

The Caltech Asynchronous Processor

The first recent asynchronous processor was built by Alain Martin at Caltech [MART89]. The processor was described using a CSP-like notation which was compiled to a circuit description by means of program transformations. The resulting circuit is delay insensitive, using dual rail signalling.

The processor has a "RISC-like" load/store instruction set with 16 registers. It consists of a number of concurrent processes responsible for instruction fetch, operand read, ALU operate etc. The processor has been implemented in a 1.6 μm CMOS process, and operates at 18 million instructions per second (MIPS) at room temperate and 5 V. The circuit continues to function at very low supply voltages, with optimum energy per

1. The asynchronous design styles are defined in section 6.1.

Processor	Design Style	Instruction set	Organisation
CAP	4-phase, Dual rail, Delay insensitive	Own 16-bit RISC-like	Fetch-execute pipeline
AMULET1	2-phase, Bundled data	ARM	Pipelined, No forwarding
AMULET2	4-phase, Bundled data	ARM	Pipelined, Forwarding
NSR	2-phase, Bundled data	Own 16-bit RISC-like	Pipelined, No forwarding, Decoupled load/store and branch
Fred	2-phase, Bundled data	based on 88100	Pipelined, Multiple functional units, Single Issue, No forwarding, Decoupled load/store and branch
CFPP	2-phase, Bundled data	SPARC	Pipelined, Multiple execution stages, Single Issue, Forwarding using counter-flow result pipeline
Hades	unspecified	Own	Pipelined, Multiple functional units, Multiple issue, Forwarding
ECSTAC	Fundamental mode	Own Variable length	Pipelined, No forwarding
TITAC	2-phase, Dual rail, Quasi delay insensitive	Own 8-bit	Non-pipelined
ST-RISC	Dual rail, Delay insensitive	Own	Fetch-execute pipeline
FAM	Dual rail, Delay insensitive, 4-phase	Own RISC-like	Pipelined.
STRiP	Variable clock synchronous	MIPS-X	Pipelined, Forwarding
SCALP	4-phase, Bundled data	Own	Pipelined, Multiple functional units, Multiple Issue, Explicit forwarding

Table 4.3: Recent Asynchronous Microprocessors

operation at around 2 V. It has also been tested in liquid nitrogen at 77 K when its performance reaches 30 MIPS.

AMULET1 and AMULET2

The AMULET group led by Steve Furber at the University of Manchester built AMULET1, the first asynchronous implementation of a commercially important instruction set [PAVE94]. AMULET2 is a similar processor still under development which also implements the ARM instruction set.

The ARM instruction set was designed for synchronous implementation [FURB89], and some of its features were included because they were convenient in the synchronous system. AMULET1 and AMULET2 implement this instruction set “warts and all”, with some of these features leading to complexity and overheads. AMULET1 and AMULET2 implement the difficult areas of interrupts and exceptions.

AMULET1 was designed using a 2-phase bundled data design style based on [SUTH89]. It has a 5-stage execute pipeline (register read, multiply/shift, ALU, empty stage, register write) but does not provide for result forwarding. A locking mechanism is used to stall instructions at the register read stage until their operands have been written by previous instructions. Consequently the pipeline throughput is low.

Similar problems affect branch instructions. A branch instruction has to pass through 10 pipeline or fifo stages between being fetched and the target address being sent to memory. This results in large numbers of prefetched instructions being discarded and significant periods during which the pipeline is stalled.

AMULET1 permits out of order completion of load instructions relative to normal ALU instructions, though the instructions must commit to completing without fault in order.

Comparison of AMULET1 with the synchronous ARM processor is shown in table 4.4 (from [PAVE94]).

It should be noted when considering the figures in table 4.4 that the ARM6 is the fourth generation of this synchronous microprocessor and is considered to have particularly high power efficiency and small size.

	AMULET1	ARM6
Process	1.2 µm 2-layer metal CMOS	1.2 µm 2-layer metal CMOS
Cell core area	5.5 mm x 4.1 mm	4.1 mm x 2.7 mm
No. of transistors	58,374	33,494
Performance	9 K Dhrystones	14 K dhrystones @ 10 MHz
Dissipation	83 mW	75 mW @ 10 MHz
Design Effort (approx.)	5 man years	5 man years

Table 4.4: Characteristics of the AMULET1 compared with ARM6; slow-slow silicon

The AMULET2 processor is being designed using a 4-phase bundled data design style because this is believed to have benefits in terms of speed, size, and power. AMULET2 has a slightly shorter pipeline than AMULET1 and employs both forwarding and branch prediction to increase the pipeline utilisation. Forwarding is performed by means of a "last result" register at the ALU. Because this forwarding is around only a single pipeline stage the performance degradation mentioned in section 4.2.1 is not a problem. Other forwarding mechanisms are used to allow the result of a load instruction to be used in a following instruction. These enhancements are expected to give AMULET2 significantly better performance than AMULET1.

NSR

The NSR (Non-synchronous RISC) processor was built using FPGA technology by Erik Brunvand at the University of Utah [BRUN93] [RICH92]. It is implemented using a 2-phase bundled data protocol.

NSR is a pipelined processor with pipeline stages separated by fifo queues. The idea of the fifo queues is that they decouple the pipeline stages so that an instruction that spends a long time in one stage need not hold up any following instructions - see section 4.2. The disadvantage of this approach is that the latency of the queues themselves is significant and because of the dependencies within a processor pipeline the increase in overall latency is detrimental.

Like AMULET1 NSR uses a locking mechanism to stall instructions that need operands produced by previous instructions; it has no forwarding mechanism.

NSR has a 16-bit datapath and 16-bit instructions. The instructions include three 4-bit register specifiers and a 4-bit opcode - much like a half-length RISC instruction. Some aspects of its instruction set are specialised for the asynchronous implementation: load, store and branch instructions make the fifos that interconnect the functional units visible to the programmer.

Load instructions have two parts. One instruction specifies a load address. A subsequent instruction uses the load result by reading from a special register r1. There may be an arbitrary separation between the two instructions, and it is possible to have several load operations outstanding at one time. Store instructions work similarly by writing the store data to r1.

Conditional branch instructions are decoupled by a boolean fifo from the ALU that executes comparison instructions. Computed branch instructions also use a fifo to store computed branch addresses.

Fred

Fred is a development of NSR built by William Richardson and Erik Brunvand at the University of Utah [RICH95]. Like NSR, Fred is implemented using 2-phase bundled data logic. It is modelled using VHDL.

Fred extends the NSR to have a 32-bit datapath and 32-bit instructions, based on the Motorola 88100 instruction set.

Fred has multiple functional units. Instructions from the functional units can complete out of order. However the instruction issuer can only issue one instruction at a time, and it seems that the register bank is only able to provide operands for one instruction at a time. This allows for a relatively straightforward instruction issue and precise exception mechanism, but limits the attainable level of parallelism.

There is no forwarding mechanism; instructions are stalled at the instruction issuer until their operands have been written to the register bank. There is no out of order issue.

Like the NSR, Fred uses programmer-visible fifo queues to implement decoupled load/ store and branch instructions. This arrangement has the possibility of deadlock if the program tries to read from an empty queue or write to a full one. Fred chooses to detect this condition at the instruction issuer and generate an exception.

Counterflow Pipeline Processor

The Counterflow Pipeline Processor (CFPP) is being developed by Ivan Sutherland, Robert Sproull, Charles Molnar and others [SPRO94]. Its implementation is based on extensions of the techniques proposed in [SUTH89].

The CFPP executes SPARC instructions. Its novel contribution is the way in which it solves the problem of result forwarding in an asynchronous pipeline.

CFPP has two pipelines as shown in figure 4.15. In one pipeline instructions flow upwards; in the other results flow downwards. As instructions flow upwards they watch out for results of previous instructions that they must use as operands flowing downwards. If they spot any such operands they capture them; if not they eventually receive a value that has flowed down from the register bank which is at the top of the pipelines. When an instruction has obtained all of its operands it continues to flow upwards until it finds a pipeline stage where it can compute its result (there can be several functional units associated with different stages of the pipeline). Once computed the result is injected into the result pipeline for use by any following dependent instructions and is also carried forward in the instruction pipeline to be written into the register bank.

The counterflow pipeline neatly solves the problem of result forwarding. However it seems that the throughput of the processor is ultimately limited by the rate at which instructions can progress, and this is not likely to be particularly fast in view of the amount of comparison and arbitration logic required at each stage.

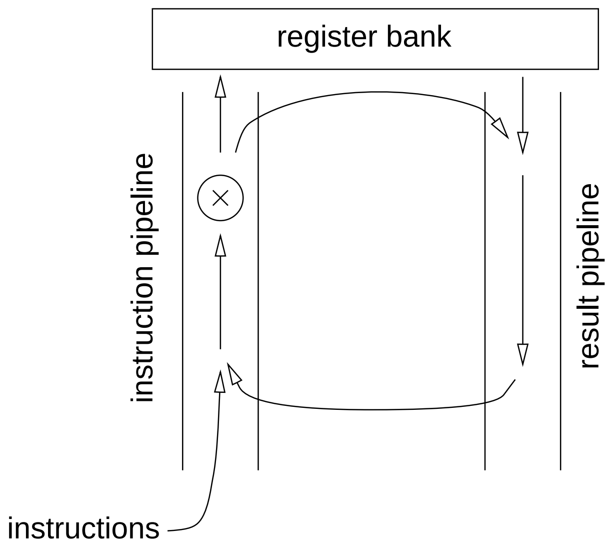


Figure 4.14: Counterflow Pipeline Processor

Hades

Hades is a proposed superscalar asynchronous processor in the early stages of design from Corrie Elston at the University of Hertfordshire [ELST95]. It is in many ways similar to a conventional synchronous superscalar processor; it has a global register bank, forwarding, and a complex (though in-order) instruction issuer. Its forwarding mechanism uses a central scoreboard to keep track of which result is available from where.

ECSTAC

ECSTAC is an asynchronous microprocessor designed by Shannon Morton, Sam Appleton and Michael Liebelt at the University of Adelaide [MORT95].

ECSTAC is implemented using fundamental mode control circuits.

It is deeply pipelined with a complex variable length instruction format. It has 8 bit registers and ALU. The designers report that the variable length instructions and the mismatch between the address size and the datapath width made the design more complex and slower.

There is no forwarding mechanism within the datapath, and a register locking scheme is used to stall instructions until their operands are available.

TITAC

TITAC is a simple asynchronous processor built by a group at the Tokyo Institute of Technology [NANY94]. It is based on the quasi delay insensitive timing model and so has to use dual rail signals to encode its datapath; this results in about twice as many gates in the datapath compared to an equivalent synchronous datapath.

Architecturally TITAC is very straightforward with no pipelining and a simple accumulator-based instruction set.

ST_RISC

ST-RISC is an architecture proposed by a group from the Israel Institute of Technology [DAVII93]. It is delay insensitive and uses a dual-rail datapath.

ST-RISC has a two stage fetch-execute pipeline and a 3-address register based instruction set.

FAM

FAM [CHO92] is a dual rail asynchronous processor with a RISC like load-store instruction set. It has a four stage pipeline but register read, ALU, and register write occur in a single stage eliminating the need for any forwarding.

STRiP

STRiP was built by Mark Dean at Stanford University [DEAN92]. Its instruction set is that of the MIPS-X processor.

STRiP is included here even though it has a global clock signal and could be considered synchronous. It is unusual in that the speed of the global clock is dynamically variable in response to the instructions being executed, giving much of the advantage of an asynchronous system. The performance of STRiP is typically twice that of an equivalent synchronous processor.

By maintaining global synchrony STRiP is able to implement forwarding in the same simple way that synchronous processors do.

4.4.1 Observations

The processors described here can be divided broadly into two categories:

- Those built using a conservative timing model, suitable for formal synthesis or verification, but with a simple architecture: CAP, TITAC, ST-RISC.
- Those built using a less cautious timing model using an informal design approach, but with a more ambitious architecture: the AMULET processors, NSR, Fred, the Counterflow Pipeline Processor, Hades, ECSTAC and STRiP.

From the point of view of this work it is the second category that is more interesting. The lessons that each of these processors teaches us are

AMULET1: Asynchronous implementation of a commercially important processor is possible. Exceptions and interrupts are possible. The lack of forwarding and branch prediction are significantly detrimental.

AMULET2: Forwarding is possible in a simple pipeline.

NSR: Decoupling with fifos is easy, but perhaps too easy as it is detrimental to performance. An instruction set with explicit queues and decoupling is possible.

Fred: Multiple functional units can be used with a single instruction issuer, but the throughput will be limited by the issuer.

CFPP: An interesting though expensive general implementation of asynchronous forwarding.

Hades: Superscalar control of the style used by synchronous processors is applicable in principle to asynchronous processors.

ECSTAC: Variable length instructions are unpleasant to deal with.

STRiP: Global synchrony does not have to mean fixed frequency clock signals.

Chapter 5: The SCALP Architecture

Having given the motivation for SCALP in the previous chapters, this chapter describes the SCALP architecture and explains how it meets the objectives that have been set out.

To summarise these objectives:

For high power efficiency we desire high code density, parallelism and asynchronous implementation.

To obtain high code density:

(1) Use variable length instructions (section 2.1.1).

(2) Have the minimum possible redundancy in the instructions (section 2.1).

(3) Reduce the contribution to code size of register specifiers by exploiting the fact that the result of one instruction is typically used very soon by a following instruction (section 2.1.2).

To obtain parallelism:

(4) Employ pipelining (section 3.5).

(5) Use multiple functional units (section 3.6).

(6) Issue multiple instructions out of order (section 3.6.1).

(7) Try to reduce the complexity of the superscalar instruction issuer (section 3.6.3).

To permit asynchronous implementation:

(8) Adapt the architecture so that conditional forwarding may be used within and between the pipelines (section 4.2.2).

(9) Provide decoupling between functional units (section 4.2).

Also:

(10) Have a HALT instruction (section 4.1.4).

(11) Have instructions that operate on quantities narrower than the full width of the datapath (section 2.2).

SCALP's main architectural innovation is its lack of a global register bank and its result forwarding network. This approach is motivated by points (3), (7), (8), and (9) above.

5.1 Explicit Forwarding and Non-Global Registers

Most SCALP instructions do not specify the source of their operands and destination of their results by means of register numbers. Instead the idea of explicit forwarding is introduced. Each instruction specifies the destination to which its result should be sent. That destination is the input to another functional unit where the instruction that uses that result will subsequently execute, consuming the value.

Instructions do not specify the source of their operands at all; they implicitly use the values provided for them by the preceding instructions.

SCALP does have a register bank; it constitutes one of the functional units. It is accessed only by read and write instructions which transfer data to and from other functional units by means of the explicit forwarding mechanism.

Figure 5.1 shows the overall organisation of the processor.

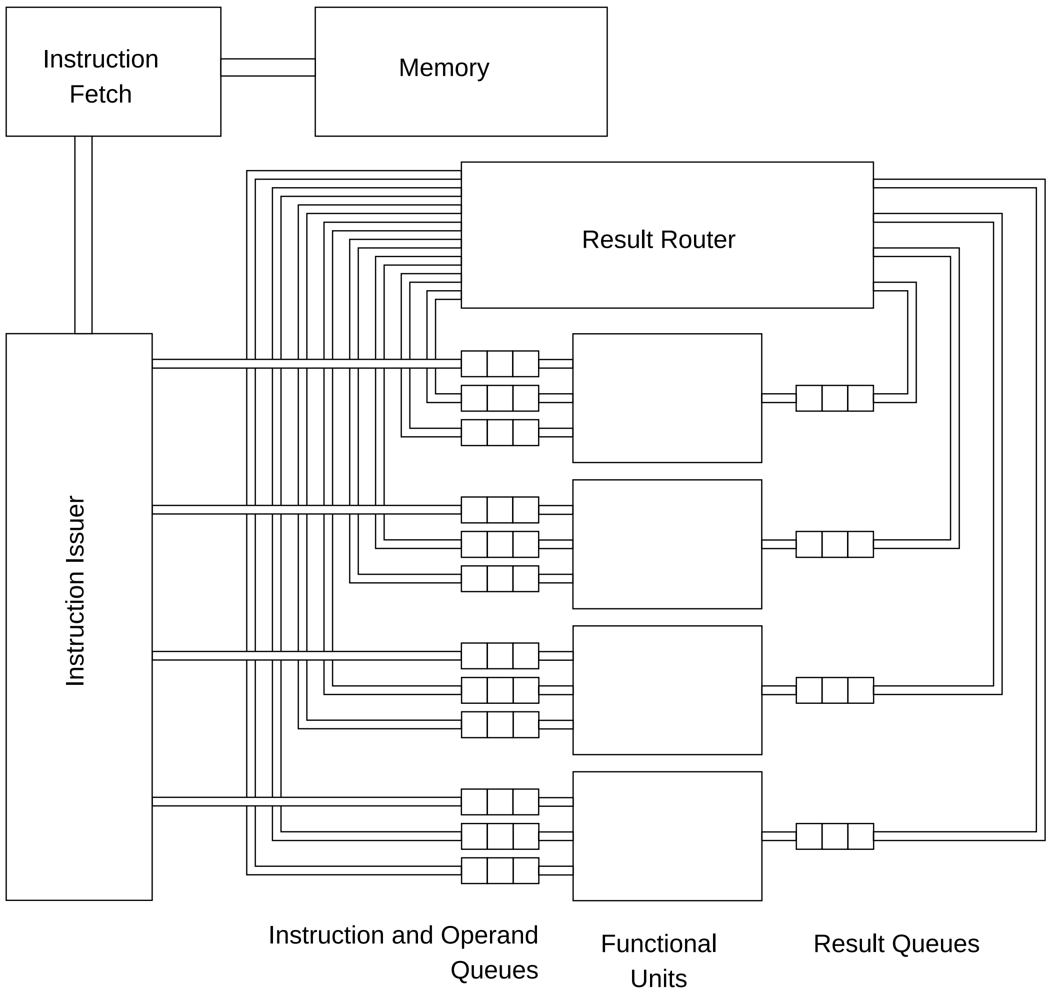


Figure 5.1: SCALP Processor Organisation

Several instructions are fetched from memory at a time. Each instruction has a small number of easily decoded bits that indicate which functional unit will execute it[1]. The instruction issuer is responsible for distributing the instructions to the various functional units on the basis of these bits.

1. Note that the instructions are statically allocated to functional units. If there is more than one functional unit that is capable of executing a particular instruction, one must be chosen by the compiler. This helps to simplify the instruction issuer and is essential to the explicit forwarding mechanism. It may be slightly detrimental to performance if the compiler is not able to predict accurately which functional units will be most busy.

Each functional unit has a number of input queues: one for instructions and one for each of its possible operands. An instruction begins execution once it and all of its necessary operands have arrived at the functional unit.

The functional unit sends the result along with the destination address to the result routing network. This places the result into the appropriate input queue of another functional unit.

Consider the following code:

```
a = b+c;
d = 10-a;
```

If we assume that variables b and c are initially in memory, a conventional architecture would compile this to the following code:

```
load rb,[b_addr]
load rc,[c_addr]
add  a,b,c
sub  d,#10,a
```

SCALP would treat the code as follows: it observes that the loaded values are used only by the following addition and that the result of the addition is used immediately by the subtraction. The compiler will be able to tell whether variable a will be read again before it is next written to. If it is not then the following code is possible:

```
load [b_addr] -> adder_input_a
load [c_addr] -> adder_input_b
add           -> subtracter_input_a
sub  #10      -> ...
```

This code uses the same number of instructions as the conventional architecture, yet each instruction can be significantly smaller. The instructions do not need to specify any input operands and the destinations are specified simply by a functional unit input code, which can be encoded in only a few bits.

The advantages of explicit forwarding are as follows:

- Code density is substantially improved. In the SCALP instruction set described in section 5.3, the 4 functional units have a total of 8 operand input queues, so a 3 bit field is needed to specify the destination of an instruction. This is small compared with the 15 bits needed to specify 3 registers in a conventional RISC architecture (though SCALP will tend to execute more instructions).

- The complexity of the superscalar instruction issue problem is substantially reduced. SCALP has no need to check for dependencies at the instruction issuer. If an instruction is issued before its operands have been computed it will wait at the input to the functional unit until they arrive.

- By explicitly indicating where each result must be sent to, an implementation using conditional forwarding is made possible.

- The operand and instruction queues provide decoupling between the functional units, so a slow instruction in one functional unit may not hold up instructions in other functional units.

One way of viewing the result routing network and the associated queues is as an additional level in the processor’s memory hierarchy. In a conventional processor the register bank is used for both short term storage from one instruction to the next[1] and for medium term storage of local variables. In SCALP, the register bank is still used for medium term storage of local variables but the result routing network and associated queues are used for short term storage between instructions.

The fifo queues associated with each functional unit input make it possible to use explicit forwarding to transfer values between instructions that may not be immediately adjacent, as was the case in the earlier example. A simple example is the following code that increments two memory locations:

1. In fact in conventional processors the register bank is not used for short term storage between instructions; the forwarding busses are. However the programming model makes it appear that the register bank is being used, and the values will be written into the register bank even when it is not necessary.

```
LOAD -> alu
LOAD -> alu
ADD #1 -> store
ADD #1 -> store
STORE
STORE
```

Here a form of software pipelining has been applied to reduce the number of data dependencies in the code, permitting faster execution.

There are some superficial similarities between the SCALP approach and dataflow computing. In particular it is possible to describe SCALP programs by means of dataflow graphs such as figure 5.2. This similarity is only superficial however; flow of control in SCALP is determined by a conventional control driven mechanism, not a data driven mechanism.

Figure 5.2 shows a dataflow diagram for a simple SCALP program to sum the values in an array. The left hand loop generates a sequence of addresses; in the middle a load instruction reads the array value; at the right hand side a loop sums the loaded values. The code listing is shown alongside.

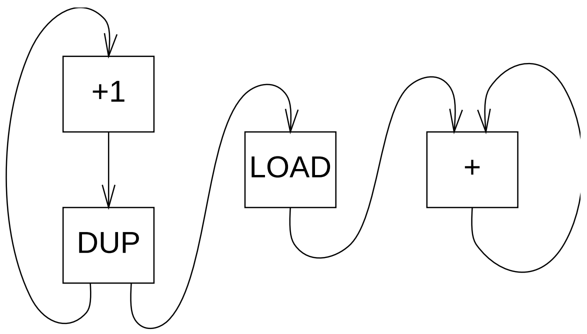


Figure 5.2: Scalp Program Dataflow Representation

```
L1:     ADD 1 -> movea
        DUP -> alua, mema
        LOAD -> alub
        ADD -> alua
        BR L1
```

The basic execution model allows each result to be used by only one following instruction. This is sufficient in very many cases. When a result needs to be used more than once two possibilities are available to the programmer:

- The value can be written into the register bank, and read as many times as necessary.
- A duplicate operation can be used to send two copies of the value to two different destinations.

The program shown in figure 5.2 makes use of a duplicate instruction.

Analysis of SPARC code carried out for [ENDE93] (see appendix A) measured the number of times that each instruction result is used. This is summarised in table 5.1 and shows that around two thirds of instruction results are used exactly once, and hence could make use of explicit forwarding without using a duplicate instruction.

Number of uses	Proportion of results
0	17 %
1	64 %
2	11 %
3	4 %
4	2 %
>4	2 %

Table 5.1: Number of times each instruction result is subsequently used (from[ENDE93])

One way of viewing the instruction and operand queues is as reservation stations. The difference is that in SCALP the instructions are issued from the queues in the same order that they arrive. Furthermore operands must arrive in the correct order to match their instructions; this is guaranteed by the structure of the code.

[JOHN91] (see appendix A) considers the possibility of in-order issue to functional units from reservation stations and concludes that it has only a small detrimental effect on performance compared with out of order issue from the reservation stations:

> *“ 7.1.3 A Simpler Implementation of Reservation Stations*
>
> *...each reservation station can issue instructions in the order that they are received from the decoder. Instructions are issued in order at each functional unit, but the functional units are still out of order with respect to each other, and thus the processor still performs out of order issue. The advantage of this approach is that each reservation station need only examine one instruction to issue - the oldest one. .... Also, ... the reservation station can be managed as a first in first out queue.*
>
> *The performance advantage of out of order issue at the reservation stations is negligible in some cases. Across the entire range of reservation station sizes [1 to 16 entries], the biggest difference in average performance between the two approaches is 0.6 % for a two instruction decoder and 2 % for a four instruction decoder. ”*

5.2 Deadlocks and Determinism

A number of questions may be asked about the safety of the proposed architecture:

- What happens when an instruction needs to fetch an operand from an input queue to which no preceding instruction has sent a result?

- What happens if instructions send many results to an operand queue without any instructions consuming them?
- What happens if two functional units send results to the same operand queue at about the same time?

The answers are that in the first two cases the processor will deadlock. In the third case the behaviour will be nondeterministic.

Note that all three cases represent programming errors. Correctly constructed programs should not exhibit any of these code sequences.

In the case of the deadlock problems, the Fred architecture (see section 4.4) has the same situation in the case of its memory access decoupling. Fred chooses to detect any illegal instruction sequences in the instruction issuer and raise an exception. Using this approach in SCALP would require a number of up/down counters and comparators in the instruction issuer and would both complicate and slow down the issuing process. In view of this SCALP chooses another approach. Deadlocks are allowed to occur and are detected by means of a watchdog timer associated with for example the instruction fetch logic. In the event of a watchdog timer timeout the processor is reset and restarts from an exception vector with empty queues. If necessary a mechanism could be provided to store a copy of the program counter value at the time when the reset occurred so that the system software could identify the faulty code.

The non-deterministic instruction sequences are more interesting. At first sight it may seem that these sequences also represent programming errors and should be illegal. Unfortunately some apparently useful program constructs lead to this form of non-determinism. Consider the following code sequence:

```
LOAD -> alu
ADD -> xyz
READ REGISTER -> alu
SUB -> pqr
```

The effect that is wanted here is that the result of the load instruction is sent to the ALU where it is used by the following ADD instruction, and that the result of the read register instruction is sent to the ALU where it is used by the SUB instruction.

Unfortunately one possible sequence of events in an asynchronous implementation is the following:

- The load instruction is issued and enters the load unit instruction queue. The load unit starts to access the memory which is slow.
- The add instruction is issued and enters the ALU instruction queue. It cannot start execution because its operand has not arrived.
- The read register instruction is issued and enters the register bank instruction queue. It executes quickly and sends its result to the ALU.
- The add instruction is matched with the result of the read register instruction, executes and sends its result to “xyz”.
- The sub instruction is issued and enters the ALU instruction queue. It cannot start execution because its operand has not arrived.
- The load instruction completes and sends its result to the ALU.
- The sub instruction is matched with the result of the load instruction, executes and sends its result to “pqr”.

This is not the desired behaviour, yet the original code sequence seems to be a useful one.

It should be noted that there are many similar situations where this problem does not occur. For example instructions that are executed by the same functional unit may not overtake each other. The routing network also guarantees that two values sent between the same pair of points will reach their destination in the same order that they left their source. Consequently the following code sequence is legal:

```
LOAD -> alu
ADD -> xyz
LOAD -> alu
SUB -> pqr
```

Often the order of the instructions will be enforced by data dependencies between the instructions. Consider the following sequence:

```
LOAD -> alu
ADD -> regd
WRITE REGISTER
READ REGISTER -> alu
SUB -> pqr
```

Here the register read must execute after the register write (even if a different register is being accessed). The register write is dependent on the result of the add which in turn depends on the load. It is therefore impossible for the result of the register read to reach the ALU before the result of the load.

It can be argued that when this problem occurs it indicates that the code is not written in a particularly parallelisable way, or that the processor does not have the right balance of functional units. For example in the faulty example presented above the code makes greater use of the ALU than of the other functional units, making it a bottleneck. If two ALUs were present the code would operate more efficiently and without any problems of determinism. In this case the program would be as follows:

```
LOAD -> alu_1
ALU_1:ADD -> xyz
READ REGISTER -> alu_2
ALU_2:SUB -> pqr
```

Despite this there are situations where non-determinism needs to be eliminated. There are two approaches.

Firstly a sequencing instruction is provided. Two operands are sent to the move unit where they arrive on different input ports. The seq instruction, which is executed by the move unit, sends the two values to their ultimate destination in a defined order. For example:

```
LOAD -> movea
READ REGISTER -> moveb
SEQ -> alu
ADD -> xyz
SUB -> pqr
```

Alternatively the code may be re-ordered to introduce new dependencies, ensuring a fixed execution order. For example this code

```
READ REGISTER -> load address
READ REGISTER -> alu
LOAD -> write data
ADD -> write data
WRITE REGISTER
WRITE REGISTER
```

may be re-written as

```
READ REGISTER -> load address
LOAD -> write data
WRITE REGISTER
READ REGISTER -> alu
ADD -> write data
WRITE REGISTER
```

The disadvantage of this second approach is that it tends to serialise the code, reducing the potential parallelism.

To help the programmer, the scaSIM simulator (see section 6.2) detects any non-deterministic sequences and produces an error message.

5.3 Other SCALP Features

There are two other features of the instruction set that increase power efficiency: variable length instructions and variable width operations.

The format of a SCALP instruction is shown in figure 5.3. The instruction comprises 4 or 5 parts: a functional unit identifier, a destination address, an opcode whose meaning is specific to the functional unit, a width bit, and an optional immediate value.

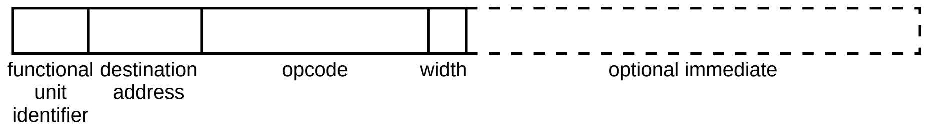


Figure 5.3: SCALP Instruction Format

The width bit indicates whether the instruction operates on 32 bit or 8 bit data. Power is saved by activating only a quarter of the datapath for 8 bit operations.

Making the immediate value optional increases the code density as many instructions do not need an immediate operand. This makes the instruction format variable length. As was mentioned in section 2.1.1 variable length instructions potentially make instruction issue more difficult in a superscalar processor. SCALP tries to reduce this effect as follows:

- There are only two sizes of instruction.
- The additional bits in the longer format contain only the immediate value and not any information that may be needed early in decoding.
- Instructions with and without immediate values are easily distinguished early in decoding.

5.4 A SCALP Instruction Set

This section describes a proposed instruction set for a SCALP processor. This instruction set is used for the examples in this chapter and is implemented in the next chapter. However it should be noted that the SCALP principles described in the preceding sections need not apply only to this instruction set; in fact this instruction set has been chosen as just about the simplest SCALP instruction set possible and so has some weaknesses.

There are five functional units:

- The ALU. This performs a standard set of arithmetic, logical, and comparison operations. It has two operand queues, alua and alub. Normal arithmetic results can be sent to any of the other functional units' input queues, but comparison results are handled separately and can be sent only to the move unit for conditional move instructions and the branch unit for conditional branch instructions.

- The register bank. This executes read and write instructions, operating on the 32 general purpose registers. It has one input queue, regd, for write data.

- The load/store unit. This performs memory accesses. It has two input queues, mema and memd, for addresses and store data respectively. The unit includes an adder which can add a displacement in the instruction to the address from the mema queue.

- The move unit. This performs the seq and dup instructions described in sections 5.2 and 5.1. It also performs a conditional move instruction and the mvlink instruction which is used in subroutine calls. It has four input queues; two normal ones (movea and moveb), a special boolean input queue for conditional move conditions and a special input for subroutine return addresses.

- The branch unit. This provides conditional and unconditional relative branches, computed branches for case statements and subroutine returns, and a branch and link instruction for subroutine entry. There are two input queues; a normal one for computed branch addresses and a boolean queue for conditional branch conditions. The branch unit also executes the HALT instruction.

There is a total of 8 operand queues (alua, alub, regd, mema, memd, movea, moveb, brc) which can be encoded using 3 bits.

Move unit and register bank instructions have a fixed length of 12 bits, as shown in figures 5.4 and 5.5.

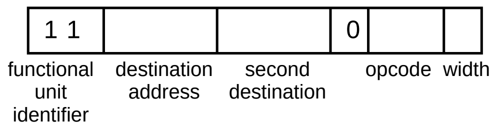


Figure 5.4: Move unit instruction format

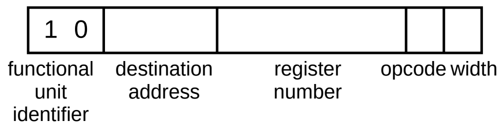


Figure 5.5: Register bank instruction formats

Other instructions can be either 12 or 24 bits long. In the case of 24 bit instructions the additional bits form an immediate value used by the instruction; all of the control information is contained in the first 12 bits. The formats of the ALU, load/store and branch instructions are shown in figures 5.6, 5.7, and 5.8.

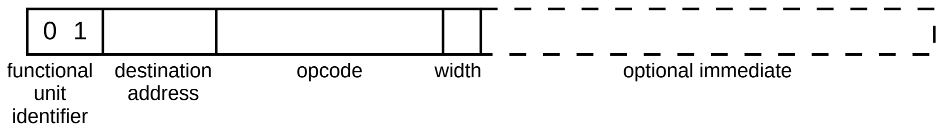


Figure 5.6: ALU instruction format

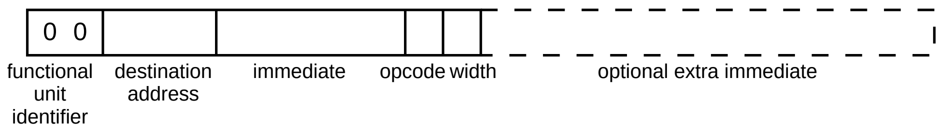


Figure 5.7: Load/Store instruction format

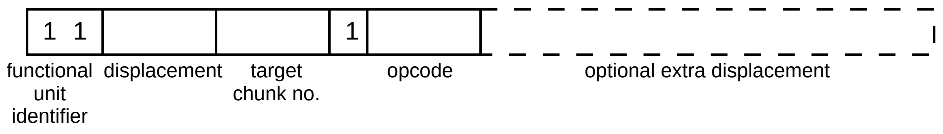


Figure 5.8: Branch instruction format

The instruction stream is divided into blocks of 12 bits, referred to here as “chunks”, each of which is either an instruction chunk or an immediate chunk. Five of these chunks are then placed together into a 64 bit “FIG” (Five Instruction Group) (figure 5.9). The five chunks use 60 of the 64 bits. The additional 4 bits form a control field that encodes which of the five chunks is an immediate and which is an instruction[1]. As was noted in section 2.1.1, when carrying out superscalar instruction issue with variable length instructions it is essential to be able to tell where each instruction starts and finishes. The control field has this effect.

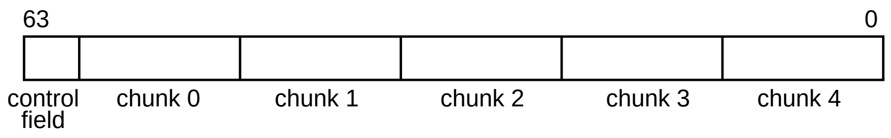


Figure 5.9: Five Instruction Group (FIG) format

1. Only four bits are needed because not all possible combinations of immediate and instruction chunks are possible within the FIG; two immediate chunks may never be adjacent to each other.

Branch target addresses are specified by giving the address of the FIG containing the target and then giving the chunk number of the target instruction within the FIG.

There are two particular reasons why this scale of organisation has been chosen. Typically approximately one instruction in five is a branch and one instruction in five is a load or store. If more than five-fold parallelism is required then it becomes necessary to execute more than one load or store instructions or more than one branch instruction concurrently. This adds extra complexity which this simpler architecture avoids.

The instruction set is summarised in table 5.2. d indicates a result destination (i.e. a functional unit input queue), bd indicates a boolean destination, [i] an optional immediate value and i a compulsory immediate value. t indicates a branch target address.

Instruction	Functional Unit	Operation
`DUP -> d1, d2`	MOVE	Duplicate
`SEQ -> d`	MOVE	Sequence
`CMOVE -> d`	MOVE	Conditional Move
`MVLINK -> d`	MOVE	Move result of last branch and link
`READ n -> d`	REG	Read register
`WRITE n`	REG	Write register
`ADD [i] -> d`	ALU	Addition
`SUB [i] -> d`	ALU	Subtraction
`RSB i -> d`	ALU	Reverse subtract
`NEG -> d`	ALU	Negate
`OR [i] -> d`	ALU	Logical OR
`AND [i] -> d`	ALU	Logical AND
`XOR [i] -> d`	ALU	Logical XOR
`CMPEQ [i] -> bd`	ALU	Compare for equality
`CMPLT [i] -> bd`	ALU	Compare for less than

Table 5.2: Instruction Set Summary (Part 1 of 2)

Instruction	Functional Unit	Operation
`CMPLTEQ [i] -> bd`	ALU	Compare for less than or equal
`MOV i -> d`	ALU	Move immediate
`MKIMM i -> d`	ALU	Build large immediate value
`LOAD [i] -> d`	LD/ST	Load
`STORE [i]`	LD/ST	Store
`BR t`	BRANCH	Unconditional branch
`BRT t`	BRANCH	Branch if true
`BRF t`	BRANCH	Branch if false
`BRL t`	BRANCH	Branch and Link
`BRC`	BRANCH	Computed Branch
`HALT`	BRANCH	Halt

Table 5.2: Instruction Set Summary (Part 2 of 2)

5.5 The SCALP Programming Model

The previous sections have described the SCALP architecture and instruction set in terms of their hardware-related characteristics such as code density and suitability for simple asynchronous implementation. The purpose of this section is to consider the architecture from a software viewpoint and to ask whether SCALP makes a good programming target.

Specifically there two questions that must be resolved:

- How many extra instructions such as duplicate and sequence operations does SCALP need in comparison with a conventional architecture?

- What effect does the presence of these instructions have on the parallelism possible between the functional units?

This section tries to address these questions by considering an example program and showing how it can be translated to the SCALP instruction set. The program is small but it illustrates the necessary points. It is a function from a graphics program that takes the (x,y) coordinates of the vertices of a triangle abc. It executes the following function

```
if (bx-ax)*(cy-ay) > (cx-ax)*(by-ay)
then
        /* do nothing, order OK */
else {
        swap(bx,cx);
        swap(by,cy);
}
```

which first checks whether the triangle is constructed in a clockwise or anti-clockwise direction. If the triangle is constructed anti-clockwise the coordinates of b and c are exchanged to make it clockwise. An implementation of the function in a conventional register-based instruction set (based on the ARM instruction set) is shown below.

```
MK_CLOCKWISE:
        LOAD RBX,[bx]
        LOAD RAX,[ax]
        SUB RP,RBX,RAX          # compute bx-ax
        LOAD RCY,[cy]
        LOAD RAY,[ay]
        SUB RQ,RCY,RAY          # compute cy-ay
        MUL RR,RP,RQ            # compute (bx-ax)*(cy-ay)
        LOAD RCX,[cx]
        SUB RS,RCX,RAX          # compute cx-ax
        LOAD RBY,[by]
        SUB RT,RBY,RAY          # compute by-ay
        MUL RU,RS,RT            # compute (cx-ax)*(by-ay)
        CMP RR,RU               # (bx-ax)*(cy-ay) > (cx-ax)*(by-ay) ?
        BRGT Rlink              # return if order is OK
        STORE RBX,[cx]          # store b and c swapped
        STORE RCX,[bx]
        STORE RBY,[cy]
        STORE RCY,[by]
        BR Rlink                # return
```

Note that it is assumed that the coordinates are initially in memory and must be loaded into registers. The function consists of 19 instructions totalling 608 bits.

One simple scheme for SCALP code generation is to take a conventional program such as the above and to replace each instruction with a sequence of register read instructions to fetch operands, an operate instruction, and a register write instruction to store the result. This scheme is clearly inefficient since it makes no use of the explicit forwarding mechanism, but it has the advantage of being able to encode any program without needing duplicate or sequence operations or other transformations to ensure determinism. This scheme can be used to give a worst case measure of SCALP code complexity. In the case of this example the SCALP program would have 50 instructions including 12 writes and 19 reads, totalling 640 bits - more than the conventional program.

A more efficient translation will make use of the explicit forwarding mechanism when appropriate dataflow exists between the instructions. When explicit forwarding cannot be used the translation can fall back to the register-based mechanism described above.

An initial dataflow-based translation to SCALP code gives the following as yet unsatisfactory program (note that for the purpose of this example the SCALP instruction set has been extended to include a multiply instruction):

```
MK_CLOCKWISE:
        MVLINK -> brc           # store return address
        LOAD [bx] -> alu_a
        LOAD [ax] -> alu_b
        SUB -> mul_a            # compute bx-ax
        LOAD [cy] -> alu_a
        LOAD [ay] -> alu_b
        SUB -> mul_b            # compute cy-ay
        MUL -> alu_a            # compute (bx-ax)*(cy-ay)
        LOAD [cx] -> alu_a
        LOAD [ax] -> alu_b
        SUB -> mul_a            # compute cx-ax
        LOAD [by] -> alu_a
        LOAD [ay] -> alu_b
        SUB -> mul_b            # compute by-ay
        MUL -> alu_b            # compute (cx-ax)*(by-ay)
        CMPGT -> br_cond        # (bx-ax)*(cy-ay) > (cx-ax)*(by-ay) ?
        BRT DONE
        LOAD [bx] -> st_data    # swap bx, cx
        LOAD [cx] -> st_data
        STORE [cx]
        STORE [bx]
        LOAD [by] -> st_data    # swap by, cy
        LOAD [cy] -> st_data
        STORE [cy]
        STORE [by]
DONE:   BRC
```

This program suffers from the following problems:

- It contains more LOAD instructions than are necessary. Unlike the conventional program a new load is used each time an operand is required as previously loaded values are not kept in local storage.

- There are problems with the order in which operands reach functional units. For example note that the result of the first multiply is supposed to reach the ALU for the comparison, yet it will probably arrive much earlier in the middle of some other calculation.

Both of these problems are addressed in the following version of the code (| indicates changed lines). This reorders the early part of the calculation so that the ax and ay operands are duplicated after they have been loaded, avoiding two loads. In addition the multiply instructions are repositioned so that their results go to the correct destination instructions.

```
MK_CLOCKWISE:
        MVLINK -> brc           # store return address
        LOAD [bx] -> alu_a
|       LOAD [ax] -> move_a
|       DUP -> alu_b, alu_b
        SUB -> mul_a            # compute bx-ax
|       LOAD [cx] -> alu_a
|       SUB -> mul_a            # compute cx-ax
        LOAD [cy] -> alu_a
|       LOAD [ay] -> move_a
|       DUP -> alu_b, alu_b
        SUB -> mul_b            # compute cy-ay
        LOAD [by] -> alu_a
        SUB -> mul_b            # compute by-ay
|       MUL -> alu_a            # compute (bx-ax)*(cy-ay)
```

```
        MUL -> alu_b            # compute (cx-ax)*(by-ay)
        CMPGT -> br_cond        # (bx-ax)*(cy-ay) > (cx-ax)*(by-ay) ?
        BRT DONE
        LOAD [bx] -> st_data    # swap bx, cx
        LOAD [cx] -> st_data
        STORE [cx]
        STORE [bx]
        LOAD [by] -> st_data    # swap by, cy
        LOAD [cy] -> st_data
        STORE [cy]
        STORE [by]
DONE:   BRC
```

The remaining question is that of determinism. A simple check for possible nondeterminism is to study the source of values for each operand queue. If the source is always the same functional unit there cannot be a problem as values taking the same path through the result routing network must arrive in the same order that they left. For the move_a, mul_a, mul_b and st_data queues the source is always the same functional unit so the operand order is deterministic.

For the alu_a queue the source is normally the load unit with the exception of the first multiply instruction. For the alu_b queue the source is normally the move unit with the exception of the second multiply instruction. Can the results of the multiply instructions arrive at the ALU sooner than they should?

In the case of the alu_b queue there is no problem because the multiply instruction is itself dependent on the previous ALU result, and so cannot generate its result out of order.

In the case of the alu_a queue there is a problem. The first multiply instruction and the preceding load instruction can execute concurrently, and either could be first to write to the alu_a queue. The solution to the problem is to reorder the code slightly and introduce a SEQ instruction that guarantees the order of the resulting values, as shown in the following.

```
MK_CLOCKWISE:
        MVLINK -> brc           # store return address
        LOAD [bx] -> alu_a
        LOAD [ax] -> move_a
        DUP -> alu_b, alu_b
        SUB -> mul_a            # compute bx-ax
        LOAD [cx] -> alu_a
        SUB -> mul_a            # compute cx-ax
        LOAD [cy] -> alu_a
        LOAD [ay] -> move_a
        DUP -> alu_b, alu_b
        SUB -> mul_b            # compute cy-ay
|       LOAD [by] -> move_a
|       MUL -> move_b           # compute (bx-ax)*(cy-ay)
|       SEQ -> alu_a
        SUB -> mul_b            # compute (by-ay)
        MUL -> alu_b            # compute (cx-ax)*(by-ay)
        CMPGT -> br_cond        # (bx-ax)*(cy-ay) > (cx-ax)*(by-ay) ?
        BRT DONE
        LOAD [bx] -> st_data    # swap bx, cx
        LOAD [cx] -> st_data
        STORE [cx]
        STORE [bx]
        LOAD [by] -> st_data    # swap by, cy
        LOAD [cy] -> st_data
        STORE [cy]
        STORE [by]
DONE:   BRC
```

This final code contains 27 instructions, 42% more than the conventional code. Three of the extra instructions are DUP and SEQ instructions required because of the explicit forwarding mechanism. Four are extra load instructions in the swap code required because of the lack of a global register bank for short term storage. One is an extra instruction required because of the operation of the subroutine return mechanism.

The increase is substantially offset by the smaller size of the SCALP instructions. The total program size for the SCALP code is 346 bits, only 57% of the conventional program’s size.

The code reordering and other changes to the program have affected the way in which it can be executed on a superscalar processor. Table 5.3 shows a possible occupation of functional units for the conventional program. Loads, stores and multiplies are assumed to take 3 time units, subtracts, compares and untaken branches 2 time units; these values are arbitrary but may approximate to the relative speeds of these operations in an asynchronous processor.

This shows that the total execution time for the program is 39 time units.

Table 5.4 is a similar representation for the SCALP code. Move operations take 1 time unit.

This shows that the total execution time for the program is 52 time units, 13 more than the conventional architecture. This is largely explained by the additional LOAD instructions in the swap code. Apart from this there has been virtually no loss of concurrency resulting from the introduction of the SEQ and DUP instructions or the code reordering.

While any conventional program can be translated to SCALP code using an inefficient scheme, this example has shown that an approach based on dataflow can give substantially better results. The problems of result ordering and determinism that require the introduction of DUP and SEQ instructions and other transformations are solvable and do not substantially affect the efficiency of the program in terms of code size or available concurrency between the functional units.

This example illustrates well the problems encountered in the translation of straight line code within basic blocks; however it has a relatively simple control flow structure which does not demonstrate how branches can disrupt the dataflow. In the presence of more branches there is likely to be greater reliance on the register bank.

LOAD/ STORE	SUB/ CMP	MUL	BRANCH
LOAD RBX,[bx]			
*			
*			
LOAD RAX,[ax]			
*			
*			
LOAD RCY,[cy]	SUB RP,RBX,RAX		
*	*		
*			
LOAD RAY,[ay]			
*			
*			
LOAD RCX,[cx]	SUB RQ,RCY,RAY		
*	*		
*		MUL RR,RP,RQ	
LOAD RBY,[by]	SUB RS,RCX,RAX	*	
*	*	*	
*			
	SUB RT,RBY,RAY		
	*		
		MUL RU,RS,RT	
		*	
		*	
	CMP RR,RU		
	*		
			BRGT Rlink
			*
STORE RBX,[cx]			BR Rlink
*			*
*			
STORE RCX,[bx]			
*			
*			
STORE RBY,[cy]			
*			
*			
STORE RCY,[by]			
*			
*			

Table 5.3: Functional Unit Use for Conventional Program (Example 1)

LOAD/ STORE	SUB/ CMP	MUL	BRANCH	MOVE
LOAD -> alu_a				MVLINK -> brc
*				
*				
LOAD -> move_a				
*				
*				
LOAD -> alu_a				DUP -> alu_b, alu_b
*	SUB -> mul_a			
*	*			
LOAD -> alu_a	SUB -> mul_a			
*	*			
*				
LOAD -> move_a				
*				
*				
LOAD -> move_a				DUP -> alu_b, alu_b
*	SUB -> mul_b			
*	*			
		MUL -> move_b		SEQ -> alu_a
	SUB -> mul_b	*		*
	*	*		*
		MUL -> alu_b		*
		*		
		*		
	CMPGT -> br_cond			
	*			
			BRT DONE	
			*	
LOAD -> st_data			BRC	
*			*	
*				
LOAD -> st_data				
*				
*				
STORE				
*				
*				
STORE				
*				
*				
LOAD -> st_data				
*				
*				
LOAD -> st_data				
*				
*				
STORE				
*				
*				
STORE				
*				
*				

Table 5.4: Functional Unit Use for SCALP Program (Example 1)

The effect that branches can have is shown in this second example. The problem is simply to find the greatest of four values a, b, c and d:

```
        max4(a,b,c,d) = max2(max2(a,b),max2(c,d));
        max2(x,y) = if x>y then x else y
```

One possible conventional implementation is as follows (note that it overwrites its operands during its operation):

```
        CMP Rb,Ra
        BGT L1
        MOV Rb,Ra
L1:     CMP Rd,Rc
        BGT L2
        MOV Rd,Rc
L2:     CMP Rd,Rb
        BGT DONE
        MOV Rd,Rb
DONE:   # result is in Rd
```

On average each execution of this code will execute 7.5 instructions using 240 bits.

A SCALP implementation suffers from the problem that the pattern of forwarding needed depends on the outcome of the comparisons, by which time it is too late to specify the destination in the source instructions. The most efficient solution is to use the register bank for all operand communication (except compare result to branch unit) as shown in the following listing:

```
        READ Rb -> alu_a;  READ Ra -> alu_a; CMPGT -> br_cond
        BRT L1
        READ Ra -> regd; WRITE Rb
L1:     READ Rc -> alu_a;  READ Ra -> alu_a; CMPGT -> br_cond
        BRT L2
        READ Rc -> regd; WRITE Rd
L2:     READ Rd -> alu_a;  READ Rb -> alu_a; CMPGT -> br_cond
        BRT DONE
        READ Rb -> regd; WRITE Rd
DONE:   # result is in Rd
```

On average each execution of this code will execute 15 instructions using 192 bits. This is twice as many instructions and 80% of the code size of the conventional code's size.

Tables 5.5 and 5.6 show functional unit usage in the same style as for the first example. For illustration the first branch is taken, the second not and the third is taken again.

SUB/ CMP/MOV	BRANCH

`CMP Rb,Ra`
`*`	

	`BGT L1`
	`*`

`CMP Rd,Rc`
`*`	

	`BGT L2`
	`*`

`MOV Rd,Rc`
`*`	

`CMP Rd,Rb`
`*`	

	`BGT DONE`
	`*`

Table 5.5: Functional Unit Use for Conventional Program (Example 2)

The execution time for the conventional program is 14 time units and for the SCALP program is 20 time units. Neither version exhibits any functional unit parallelism due to the tight data and control dependencies.

This second example shows that in unfavourable conditions such as the presence of many data dependent branches SCALP does best by falling back on its global register bank. In this case its code density still exceeds that of the conventional processor, and SCALP does no worse than the conventional processor at exploiting functional unit parallelism.

General results should not be implied from these very limited examples, but they are sufficiently encouraging to suggest that proceeding with the design is worthwhile.

SUB/ CMP	BRANCH	REG
		READ Rb -> alu_a
		READ Ra -> alu_a
CMPGT -> br_cond *		
	BRT L1 *	
		READ Rc -> alu_a
		READ Ra -> alu_a
CMPGT -> br_cond *		
	BRT L2 *	
		READ Rc -> regd
		WRITE Rd
		READ Rd -> alu_a
		READ Rb -> alu_a
CMPGT -> br_cond *		
	BRT DONE *	

Table 5.6: Functional Unit Use for SCALP Program (Example 2)

5.6 Interrupts and Exceptions

Providing support for interrupts and exceptions is often a challenging part of the design of a processor. SCALP's asynchronous implementation and the explicit forwarding scheme make exception handling somewhat different from exception handling in a conventional architecture.

No exception handling is incorporated in the SCALP implementation described in chapter 6. This section serves to show that exception handling is not an insurmountable difficulty with the architecture.

The greatest difficulty is that the exception handler must be able to save the transient state of the main program and restore it later. In a conventional processor this is not difficult as the transient state is contained within the register bank; the exception handler simply copies the contents of the register bank to a safe area of memory. For SCALP it is necessary to save and restore the contents of the operand queues which is more difficult for the following reasons:

- The exception handler will not know how many values are stored in each of the queues. If it tries to read more values than are present the processor will deadlock.

- The connectivity of the queues is limited. A value in the ALU queue cannot be saved to memory unless it can be transferred through the ALU unchanged and into the store data queue.

There are several possible solutions:

- Additional hardware can be provided to assist in saving the state. One solution would be to modify the operand queue registers to allow a parallel read out of both data values and empty/full information. This solution is the most general but it has significant hardware cost.

- The interrupt handler’s knowledge of the contents of the queues can be increased in some way. One possibility is to maintain counters in the instruction issuer that count how many values are in each queue. The interrupt handler could use this information to read only the correct number of values from each queue.

- Restrict the occasions on which interrupts can be taken to times when the operand queues are empty. This can either be detected by means of counters in the instruction issuer or by software control. This is similar to the transputer approach [MITC90] where interrupts occur only when branch instructions are executed, and the contents of the stack are not preserved over branches.

In practice some combination of these approaches would be most practical with some higher priority forms of exceptions using more additional hardware than others.

5.7 Related Work

This section describes other work that proposes an alternative to the global register bank model. The transputer uses a stack model; NSR, Fred, PIPE and WM use various forms of architecturally visible queues and “Transport Triggered” processors use a form of explicit forwarding.

5.7.1 Stack Processors

The stack processor is the only alternative to a global register bank that has been used in a commercial processor, namely the transputer [MITC90]. In a stack processor instruction sources and destinations are not normally specified explicitly by the instruction; rather they are implicitly taken from and returned to the top of a stack. Special operations must be provided to duplicate, delete, and reorganise the stack items. The code density advantage of this approach is discussed in section 2.1.3.

The disadvantage of the stack model is that it does not work particularly well in a parallel environment. The stack itself becomes a bottleneck that must be shared by the functional units in the same way that a global register bank would, but with the added problem of maintaining the stack pointer. Every instruction implicitly changes the stack pointer; it is hard to conceive of an implementation where these changes do not occur serially.

5.7.2 Architectural Queues

Some previous designs have made use of programmer-visible queues to communicate between some functional units. Typically this form of communication is restricted to the results of load operations, store data, and branch conditions as it is in these cases that the decoupling provided by the queues is most useful. Synchronous processors that have used this technique include PIPE [GOOD85] and WM [WULF88]. The technique is more natural for asynchronous processors and it is used by the NSR and Fred Processors (section 4.4) [BRUN93] [RICH92] [RICH95]. In all of these designs a global register bank is retained and used for the majority of communication. Special register numbers are reserved to refer to the queues.

5.7.3 Explicit Forwarding

Transport triggered architectures (TTAs) [CORP93] [POUN95] share with SCALP the idea of transferring values between functional units by explicitly stating the functional units concerned, rather than by using a global register bank. The Transport Triggered Architecture is also referred to as the MOVE processor because it has a single instruction which moves a value from the output of one functional unit to the input of another.

Each functional unit has a number of input registers for the various operands. One of these is designated the “trigger register”. When a value is written into the trigger register the functional unit is activated and after a known number of cycles its result is written into its result register.

The principle differences between the TTA idea and SCALP are:

- TTAs do not associate queues with the operands.
- TTAs require the compiler to be aware of the latencies of the functional units.
- TTAs allow the results of operations to be used many times without explicit duplicate operations.

The designers of the TTA consider it to be an example of a processor where the compiler takes responsibility for a greater proportion of the preparation for execution than in conventional architectures. Figure 5.10, from [HOOG94], illustrates this.

Like the TTA, SCALP makes many additional tasks the responsibility of the compiler. The compiler (or assembly language programmer) must be aware of dependencies

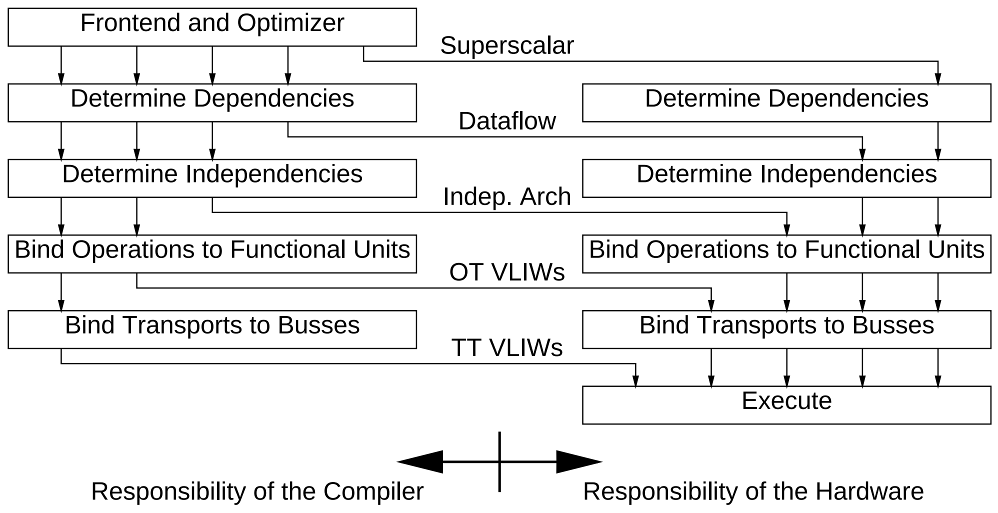

Frontend and Optimizer
Superscalar
Determine Dependencies
Determine Dependencies
Dataflow
Determine Independencies
Determine Independencies
Indep. Arch
Bind Operations to Functional Units
Bind Operations to Functional Units
OT VLIWs
Bind Transports to Busses
Bind Transports to Busses
TT VLIWs
Execute
Responsibility of the Compiler
Responsibility of the Hardware

Figure 5.10: The division of responsibilities between hardware and the compiler (from [HOOG94])

between instructions to write SCALP code. SCALP code makes a static allocation of instructions to functional units. Like TTAs and unlike any of the other processor types in figure 5.10 SCALP code makes a static allocation of result and operand movement to particular paths within the processor.

The transport triggered architecture work is significant because a compiler has been ported to the processor and significant benchmark programs have been simulated. This is encouraging because it suggests that SCALP's execution model may also make a reasonable compiler target.

Chapter 6: Implementation

A gate-level implementation of the architecture and instruction set described in the previous chapter has been carried out using the hardware description language VHDL. This design is at a sufficiently detailed level that it could be used for a silicon implementation, but this has not been done and instead the correctness and performance of the design have been established by simulation.

The first sections of this chapter describe the choice of asynchronous design style used and the overall design methodology. Later sections describe the implementation of selected parts of the architecture: the instruction issuer, the result routing network and the functional units. These areas have been chosen as they illustrate the more interesting parts of the processor in terms of asynchronous design. The final section concludes by measuring the size of the complete design.

6.1 Asynchronous Design Styles

The term “asynchronous design” is very broad; it simply means not synchronous. Within this category there are design styles that differ from each other as much as they differ from synchronous design. This section briefly describes some of these styles and investigates their properties.

Underlying each design style is a timing model. The timing model specifies what assumptions the designer is allowed to make about the relative timing of events in the system.

There are three basic timing models [SEIT80]:

- In the bounded delay model an upper and a lower bound is known for each gate delay.
- In the speed independent model gate delays are assumed to be unbounded but finite, but wires are assumed to have zero delay.
- In the delay insensitive model both gate delays and wire delays are assumed to be unbounded but finite.

In the progression from bounded delay to speed independence to delay insensitivity the designer is given less and less freedom in what he may build; all delay insensitive circuits are speed independent and all speed independent circuits work under bounded delay, but the converse is not true.

On the other hand this progression gives the designer increasing freedom in how the circuits may be constructed. In a delay insensitive circuit any interconnection of functionally correct components will operate as desired. For speed independence any collection of gates will function but wires must act as equipotential regions, that is all gate inputs driven by the wire must change together. Under the bounded delay model the designer must use gates whose delays are known to be within the limits of his delay assumptions.

This trade-off is an important one. A pragmatic designer may choose to use some combination of the different timing models in different parts of his system. For example when using full-custom tools to design at the layout level the delays in gates and wires can be accurately predicted. In this case it is reasonable to use a bounded delay timing model which allows the greatest choice of circuits. When designing at a more abstract level with less control over circuit geometry and capacitances accurate knowledge of gate delays may not be possible. Here the speed independent or delay insensitive model may be applied.

6.1.1 Timing Models for Control Circuits

The structure of control circuits is often quite random in nature and automatic placement and routing tools are generally used. This gives the designer little control over propagation delays and consequently control circuits may be designed to be speed independent or delay insensitive.

Asynchronous control circuits are often built up from simple subcircuits that perform operations such as sequencing, merging, and selecting. Because these macromodules will be used many times the designer may choose to make a bounded delay internal implementation. This requires more effort and may require manual placement and routing (or at least manual verification after automatic placement and routing), but the bounded delay circuit may be significantly simpler than the alternative speed independent or delay insensitive circuit.

6.1.2 Timing Models for Datapaths

Datapaths are important for two reasons: firstly they comprise a large part of many designs so their efficient implementation is important. Secondly they have a very regular structure which can sometimes be exploited for timing purposes.

Datapath busses comprise a number of bits whose values are used together. When each bit of the bus has been computed by one stage of the datapath the next stage may start its operation.

Under the speed independent and delay insensitive timing models the individual bits may become valid at quite different times. Consequently it is necessary for each bit of the bus to encode three possible states: logical 0, logical 1 and “not yet ready”. This is typically done using a “dual rail” encoding where two wires represent each bit.

A number of designs[1] including the Caltech asynchronous processor, TITAC, and the earlier Tangram work [BERK91] used this dual rail encoding. These designs demonstrated the substantial overhead of a dual rail encoding in terms of area, power and possibly performance compared with a single rail datapath.

Under the bounded delay timing model a different approach known as “bundled data” is possible. In a bundled data system the datapath uses a single wire for each bit like a synchronous datapath and provides a single timing signal. This timing signal is delayed by an amount guaranteed to be greater than the longest delay of any of the data bits. In this way the area is reduced compared with the dual rail implementation. However it is necessary to calculate and implement the necessary delays for the timing signal.

The previous section suggested that the bounded delay model is most appropriate for small circuits where the delays can be accurately predicted. Datapaths do not meet this criterion. However datapaths do have a very regular structure and so delays in all bits should be well matched. The timing signal’s delay path may be implemented safely by adding an extra bit to the datapath and adding some suitable safety margin.

6.1.3 Asynchronous Signalling Protocols

Communication between blocks in an asynchronous system typically uses pairs of wires called “request” and “acknowledge”. Request indicates that an action should be initiated; acknowledge indicates that the operation is complete. In bundled data systems this request-acknowledge handshake also indicates the validity of associated data signals. This exchange is sometimes referred to as handshaking.

1. See section 4.4

Two alternative signalling protocols have been proposed for handshaking using the request and acknowledge signals; these are the 2-phase and the 4-phase protocols.

The 4-phase protocol is a return to zero protocol. The behaviour of the request and acknowledge signals and any associated data bus is indicated in figure 6.1. The rising edge of request indicates the validity of the data and the desire for the action to begin. The rising edge of acknowledge indicates that the data may become invalid and that the action has completed. The falling edges of request and acknowledge comprise the recovery phase of the protocol and perform no useful work[1].

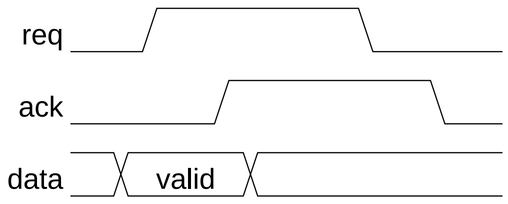


Figure 6.1: Four Phase Asynchronous Signalling

The 2-phase protocol is a non return to zero protocol. The rising edges of request and acknowledge indicate the start and end of data validity and the start and end of the required operation as before. However the handshake now finishes with the request and acknowledge signals both high. The subsequent falling edges of request and acknowledge indicate the next handshake as shown in figure 2.

The combination of 2-phase signalling and bundled data is proposed by Sutherland in his paper Micropipelines [SUTH89].

Arguments in favour of 2-phase signalling are that fewer signal transitions are required which reduces power consumption and that the lack of a recovery phase gives potential

1. Other 4-phase protocols are possible; for example in the “long hold” protocol the data remains valid until after the falling edge of acknowledge. These alternative protocols may sometimes permit implementation optimisations.

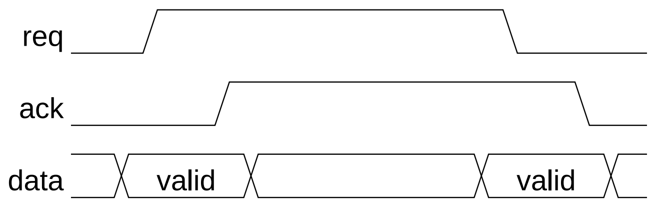


Figure 6.2: Two Phase Asynchronous Signalling

for faster operation. On the other hand advocates of 4-phase signalling suggest that 4-phase macromodule components are simpler and smaller than equivalent 2-phase components, giving both speed and power benefits.

The AMULET1 microprocessor was implemented using 2-phase signalling. Subsequent work [DAY95] has investigated the possibility of using the 4-phase protocol. The conclusion is that 4-phase is superior in terms of speed, power, and area; the AMULET2 microprocessor is being implemented using 4-phase signalling.

Interestingly the opposite conclusion has been drawn by the TAM-ARM project at Manchester which has implemented part of the asynchronous ARM using an ECL process. In this technology, which uses differential voltage signalling, the 2-phase protocol is preferred.

The SCALP design choices have been made with conventional CMOS implementation of the same type used by the AMULET processors in mind. Consequently the SCALP implementation uses 4-phase signalling[1] with bundled data datapaths.

6.2 Implementation Methodology

The overall design flow was as follows:

- Instruction set design.

1. Section 6.3 describes one case where the 2-phase protocol was found to be superior.

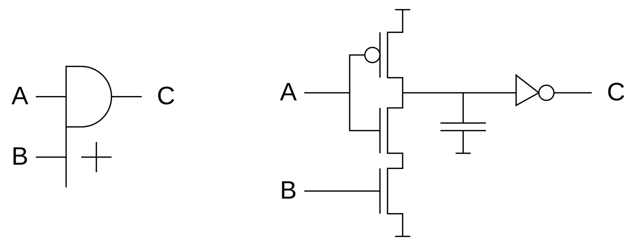
- Assembler and instruction set simulator written in ‘C’.
- High-level behavioural simulation written in VHDL.
- High-level simulation verified against ‘C’ simulation using general test programs.
- Simple gates and basic asynchronous elements modelled using behavioural VHDL.
- More complex asynchronous macromodules designed in terms of basic elements by hand or with simple asynchronous synthesis tools.
- Gate level implementations of functional units and other blocks designed.
- Blocks verified against ‘C’ simulation using block level test programs.
- Blocks combined to form complete gate level processor model.
- Processor model verified against ‘C’ simulation using general test programs.
- Processor model evaluated using “real” programs and specially developed test sequences.

The primary tool used in the implementation of SCALP has been the hardware description language VHDL. VHDL has a number of characteristics which make it ideal for this task:

- Good range of high-level types and constructs suitable for abstract behavioural modelling.
- Good low-level features for gate-level modelling.
- Good modularity suitable for a large project.
- Event based simulation model suitable for asynchronous logic.
- Fast and reliable simulators.

- Good support from other CAD software.

The set of basic gates used is based on the AMULET Low Power Cell Library [FARN95]. It includes conventional gates such as and, or, nand, nor, and xor as well as special asynchronous gates such as Muller C gates. An example C gate's symbol, simplified implementation using dynamic logic, and truth table are shown in figure 1; 17 different types of C gate are used in the design. C gates are the main state holding elements in this type of asynchronous circuit.



A	B	C
0	0	0
0	1	0
1	0	C
1	1	1

Figure 6.3: An Example Muller C Gate

More complex asynchronous macromodules were built from these basic gates. The designs come from three sources:

- The AMULET2 design.

- Designs synthesised from choice free signal transition graphs using the program FORCAGE [KISH94]. FORCAGE is less sophisticated than some of the other synthesis programs and cannot be applied to many of the macromodules because they include choice.

- Designs carried out by hand.

The macromodules used are described in table 6.1.

The majority of the macromodules carry out quite simple functions, such as sequencing or selection of events. Undesirable complexity results from two things:

Name	Function
once_or_twice	For each input handshake perform either one or two output handshakes depending on the value of a condition input.
twice	For each input handshake perform two output handshakes.
cond	For each input handshake perform either zero or one output handshake depending on the value of a condition input.
cond_undec	As cond, but input acknowledge is not deasserted until output acknowledge is deasserted i.e. there is no decoupling between input and output.
seqj2	For each input handshake perform a handshake on each of the two outputs in turn.
seqj3	For each input handshake perform a handshake on each of the three outputs in turn.
seqs2	As seqj2 but second output request is not asserted until first output acknowledge is deasserted.
seqs3	As seqj3 but second and third output requests are not asserted until first and second output acknowledges (respectively) are deasserted.
cond_pipe_merge	When condition input is asserted, wait for both input requests to be asserted before asserting output request. When condition input is not asserted, wait only for first input request. See figure 4.11.
pipe_merge	Wait for both input requests to be asserted before asserting output request.
sel2	Send each input handshake to one of the two outputs depending on the value of the condition input.
steer2	As sel2 but without acknowledge signals.
call2	Handshakes on either of two mutually exclusive inputs cause a handshake on the output.

Table 6.1: Macromodules used in the SCALP implementation

- What to do with the acknowledge signals. Often the logic required to generate the output request signal is straightforward, but more complexity is associated with storing some internal state so that the correct acknowledge signals are subsequently generated.

- What to do with the return to zero phases of the 4-phase protocol. Generally a simple approach to the return to zero phases is possible but it has poor performance. The alternative is to add additional components to “decouple” the return to zero phases. These additional components may have the effect of slowing down the forward logic. Which solution is best in a particular application is hard to evaluate and depends on the speed of the environment.

6.3 Overview of the Design

Figure 6.4 shows the top-level of the SCALP implementation. This section presents a brief overview of the function of each of the blocks in the design. The following sections consider the more detailed implementation of some of the more interesting areas.

Memory Interface

The SCALP model has a single memory port which is shared by instruction fetch and load/store accesses. The memory interface is responsible for arbitration and multiplexing between the load/store unit and the fetcher for access to the memory.

Fetcher

The fetcher contains the program counter. When a free space is present in the FIG fifo the fetcher reads the next FIG[1] from memory using the memory interface. The 64 bit FIG is fetched in two cycles as the memory interface is only 32 bits wide.

Branch target addresses are sent to the fetcher by the branch unit. If a space is available in the FIG fifo but no new address has been sent from the branch unit then the fetcher increments the last program counter value and sends this to the memory. The block has to

1. FIG: Five Instruction Group

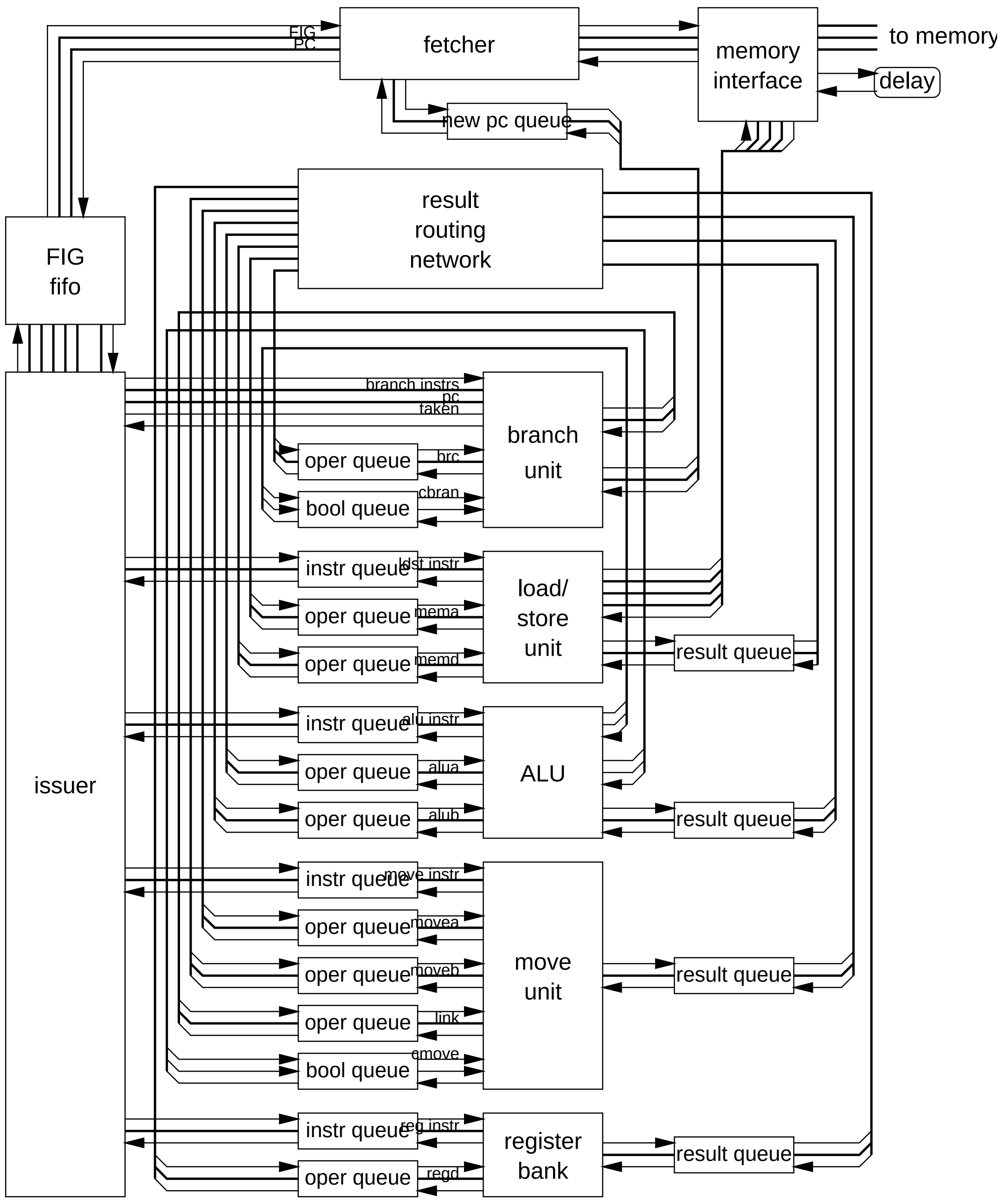


Figure 6.4: SCALP Implementation Overview

arbitrate between the arrival of a new address from the branch unit and the dispatch of an incremented program counter value.

FIG Fifo

The FIG fifo is a three stage queue that stores FIGs between the fetcher and the issuer. Its purpose is to provide decoupling between the two units so that neither unit suffers blocking or starvation; for example if the load/store unit gains access to the memory interface then the fetcher will be unable to supply new FIGs, but due to the FIG fifo the issuer will be able to continue to operate.

Issuer

The issuer decodes the functional unit identifier bits in the instructions in incoming FIGs and distributes them to the appropriate functional units via the instruction queues.

Instruction Queues

The two deep instruction queues provide decoupling between the instruction issuer and each of the functional units.

Operand Queues

The operand queues store values that have been computed by previous instructions and sent to their next instructions by explicit forwarding. The values remain in the queues until they are matched with instructions. The queues are each two deep.

Branch Unit

The branch unit executes branch instructions. For conditional branch instructions it receives comparison results from the ALU via the CBRAN boolean queue. For the computed branch instruction it receives a target address via the BRC queue. It sends target addresses to the fetcher via a single stage queue.

Most branches are relative to the current program counter value. The branch unit contains an adder which performs this calculation. The current program counter value is provided to the branch unit for this purpose with each instruction from the issuer.

The branch unit has no instruction queue. This is so that the branch unit can acknowledge to the issuer to indicate whether each branch has been taken or not. It is the responsibility of the instruction issuer to skip over any incorrectly fetched instructions between a taken branch and the branch target.

The branch and link instruction causes the unit to send the address of the instruction following the branch to the move unit’s LINK queue. This is used to effect a subroutine call.

ALU

The ALU executes arithmetic, logical and comparison instructions. Arithmetic operations are carried out by a completion indicating ripple carry adder. Logical operations are carried out by a simpler piece of logic with a matched delay path. The results of arithmetic and logical operations are sent to the result network.

Comparisons are performed by either subtracting or exclusive-ORing the operands using the normal arithmetic and logical blocks. The result is then tested for equality to zero or the adder’s carry out is tested. This means that comparison for equality is faster than a magnitude comparison as the later requires only an exclusive OR rather than a subtract. Comparison results are sent to either the branch unit’s CBRAN input or the move unit’s CMOVE input.

Load/Store Unit

The load/store unit executes load and store instructions. It accesses the memory via the memory interface. The block contains an adder which adds a displacement provided in the instruction to an address provided by the MEMA queue. Store data is provided by the MEMD queue. Load results are sent to the result routing network.

Move Unit

The move unit executes four special purpose instructions. DUP duplicates its operand, sending the two copies to any two operand queues. SEQ sequentialises two operands, sending them to a single destination in a deterministic order. CMOVE is a conditional move instruction; either one of its input operands is sent to the destination under the control of a boolean input from the result of a comparison. The MVLINK instruction transfers a value from the LINK queue to some destination, for example a leaf subroutine can return by using MVLINK to send the link value back to the branch unit.

Register Bank

The register bank contains a 32 entry register file and executes READ and WRITE instructions that access it. WRITE data is provided by the REGD input. READ results are sent to the result network.

Result Queues

The result queues provide a single stage of decoupling between the functional units and the result routing network. This decoupling helps to prevent functional units from blocking when the result routing network is temporarily busy.

Result Routing Network

The result routing network transfers the results of instructions from the result queues to the operand queues. Each functional unit copies the destination address from its instruction to the result output to accompany the result data. The result routing network uses this address to select one of the eight operand queues.

6.4 Implementation of the Instruction Issuer

The instruction issuer is one of the key parts of the SCALP design. It accepts five instruction groups (FIGs) that have been fetched from memory and sends the individual instructions to the five functional units. Each interface uses a 4-phase bundled data protocol. The issuer is decoupled from the fetcher and the functional units by fifo queues.

Each instruction within the FIG contains three bits that indicate to which functional unit it must be sent (figure 5.3). The issuer must study these bits for each instruction and select the appropriate output.

The operation is made more complex by the variable length instructions and the special treatment of branch instructions. To clarify this description a simplified design excluding these features is presented first and these additional aspects are added later.

6.4.1 The Need for Parallelism

One functionally correct though unsatisfactory implementation of the instruction issuer is shown in figure 6.5.

This circuit works as follows: when a FIG arrives from the fetcher, the count5 block and multiplexor split it into its five constituent instructions which are considered in turn. The three bits that indicate which functional unit the instruction is for are split off from the rest of the instruction and control a five way select element. The outputs of this select are requests to each of the functional unit queues.

The problem with this solution is that it is too sequential. In a pipelined processor it is essential that no part of the system becomes a consistent bottleneck; the throughput of each stage must be balanced. The memory interface can fetch several instructions at once and the five functional units can operate in parallel. This instruction issuer can issue only

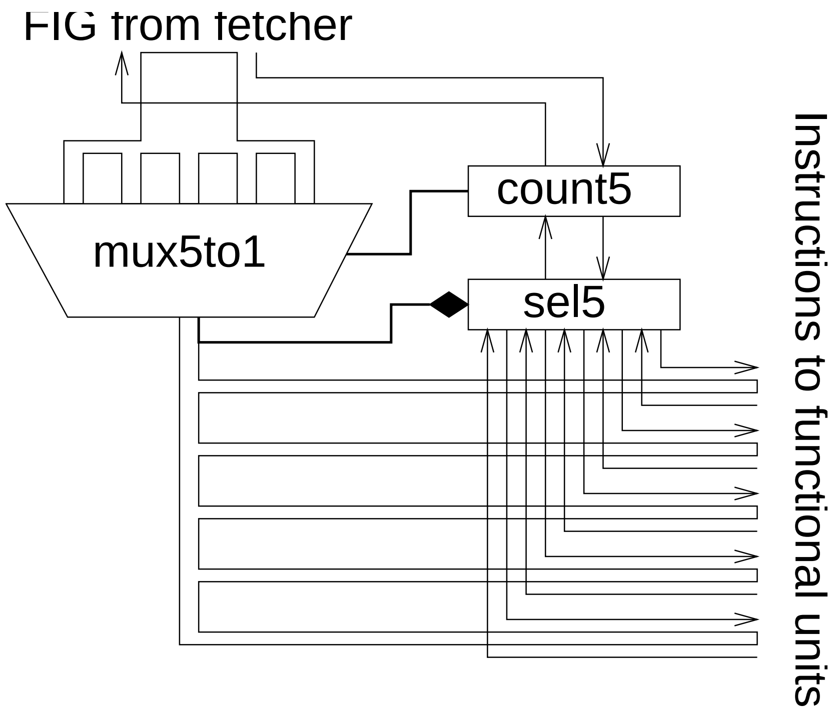


Figure 6.5: An Unsatisfactory Issuer Architecture

one instruction at a time and unless it can be made to operate five times as fast as the other parts of the system - which is unlikely - it will become a bottleneck.

The design of the instruction issuer must therefore incorporate parallelism, allowing several incoming instructions to be considered and issued to different functional units simultaneously.

Note that although the order in which instructions for one particular functional unit must arrive at that functional unit is fixed, the order in which instructions are issued between the functional units may vary. For example, if a FIG contains five instructions for these functional units:

```
 0   1   2   3   4
F0  F0  F2  F2  F1
```

then initially instructions 0, 2 and 4 can be issued to functional units F0, F2 and F1 respectively. Subsequently instructions 1 and 3 can be issued.

The boundaries between FIGs are also unimportant. For example if two FIGs contain instructions for these functional units:

```
 0   1   2   3   4
F0  F0  F0  F0  F0
F1  F1  F1  F1  F1
```

then the issuer may choose to issue instruction 0 in the first FIG to functional unit F0 and simultaneously instruction 0 in the second FIG to functional unit F1.

A good solution will trade off the performance benefits of these forms of out-of-order issue against their implementation cost. This trade-off is considered further in section 6.4.6.

6.4.2 The Proposed Solution

The proposed solution makes use of a 5 by 5 crossbar switch to allow unrestricted connections between each instruction in the FIG and each functional unit. The incoming FIGs are stored in a “forked fifo”; this is an asynchronous fifo queue whose input is controlled by a single request/acknowledge pair and whose output has separate request and acknowledge signals for each instruction. This arrangement means that once an instruction has been issued then it can be acknowledged and removed from the output of the fifo making the instruction behind it in the next FIG visible and eligible for issue.

Control is then divided up into one unit corresponding to each of the functional units. These units “watch” the instructions at the front of the FIG fifo and when they see an instruction for their functional unit they activate the appropriate switch in the crossbar and send a request to the functional unit.

This arrangement is shown in figure 6.6 (connections between the controllers and the crossbar are omitted for clarity).

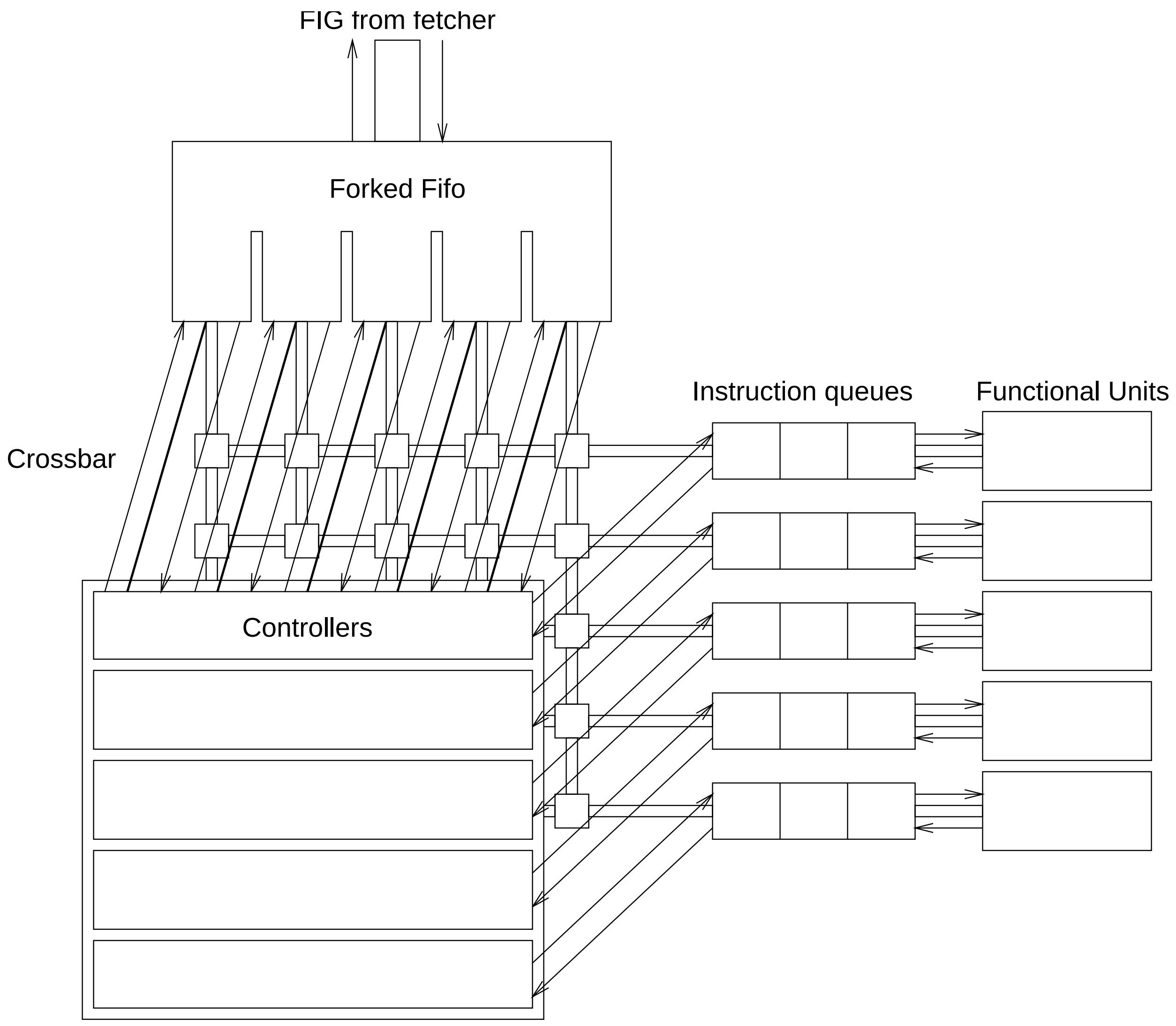


Figure 6.6: Instruction Issuer Architecture

Each functional unit controller can see the following information about each instruction at the front of the FIG fifo:

- From the state of the request and acknowledge signals, whether any valid instruction is present.
- From the decoded functional unit bits in the instruction (the “for me” signals) whether this instruction is for this functional unit.

On the basis of this information the functional unit controller must choose which if any of the instructions it may issue at this time:

- If none of the instructions is “for me” and “valid” then it can do nothing.

- If one of the instructions is “for me” and “valid” then that one may be issued, unless a preceding instruction is not valid in which case the controller must wait to see if it is “for me” first.

- If more than one instruction is “for me” then the first one in program order may be issued.

Unfortunately the information described above is not quite sufficient because of the way that the FIG fifo works. Consider the following two cases:

```
 0   1   2   3   4
F0  F1  F0  F1  F2
F0  F1  F2  F3  F0

 0   1   2   3   4
        F0  F1  F2
F0  F1  F2  F3  F0
```

In the first case the front row of the FIG fifo is full. In the second case the first two instructions from the front FIG have already been issued making the first two instructions from the second FIG visible.

Note that in both cases the instructions visible to the controllers are the same: F0, F1, F0, F1, F2. Consider the controller for functional unit 0. In the first case the correct behaviour is for it to issue instruction 0. In the second case the correct behaviour is for it to issue instruction 2. With the information described above it is not able to distinguish these two cases.

Here is another example:

```
 0   1   2   3   4
F0  F1  F0  F1  F2
F0  F1  F0  F2  F2
```

```
 0   1   2   3   4
F0  F1  F0  F1
F0  F1  F0  F2  F2
```

Once again the instructions visible to the controllers are the same: F0, F1, F0, F1, F2. Consider the controller for functional unit 2. In the first case the correct behaviour is for it to issue instruction 4. In the second case the next instruction in program order for functional unit 2 is instruction 3 of the second FIG, an instruction which is not yet available for issue, so the controller must do nothing.

To solve these problems further information must be made available about the instructions at the head of the FIG fifo. A modulo-3 counter controlled by the request and acknowledge signals is associated with each column of the FIG fifo. The controllers see the states of these counters and are able to tell which instructions belong logically before others. The letters A, B, and C are used to indicate the state of the counter. This is referred to as the "layer" in which the instruction belongs; this is orthogonal to the rows and columns of the crossbar.

The algorithm that each controller has to implement is then as follows:

- At reset, the "current column" is column 0 and the "current layer" is layer A
- If the instruction in the current column is not valid (i.e. request is not asserted; it has not arrived) then wait.
- If the instruction in the current column is for the previous layer then wait.
- If the instruction in the current column is for the current layer and it is for this functional unit then issue it.
- If the instruction in the current column is for the current layer but it is not for this functional unit then consider the next instruction.
- If the instruction in the current column is for the next layer then consider the next instruction.

- When the instruction in column 4 has been considered, go on to column 0 of the next layer.

The algorithm is described here in a sequential fashion, but it could be implemented in parallel using priority encoders and similar structures. Several possible approaches were considered and eventually one based on token passing was chosen.

The principle of operation is this: for each column and each layer a block of logic precomputes what it would do if this column and layer were “current”. The possible actions are to wait, to consider the next instruction, and to issue this instruction. These precomputations can be carried out for all columns and layers in parallel. A “token” is then passed between the blocks. When a block holds the token the column and layer that it represents are the current column and layer. When the token arrives the block performs one of three possible actions:

- If the action is to wait, the token is kept by the block. Eventually conditions will change (for example the input will become valid) causing the action to change to one of the others.

- If the action is to consider the next instruction then the token is passed to the next block.

- If the action is to issue this instruction then the appropriate crossbar switch is activated and a request is sent to the functional unit. When the functional unit sends an acknowledge then an acknowledge is sent to the FIG fifo and the token is passed to the next block.

The organisation of the blocks and the token’s path between them is shown in figure 6.7. Note that there is one token and associated structure of this type for each functional unit.

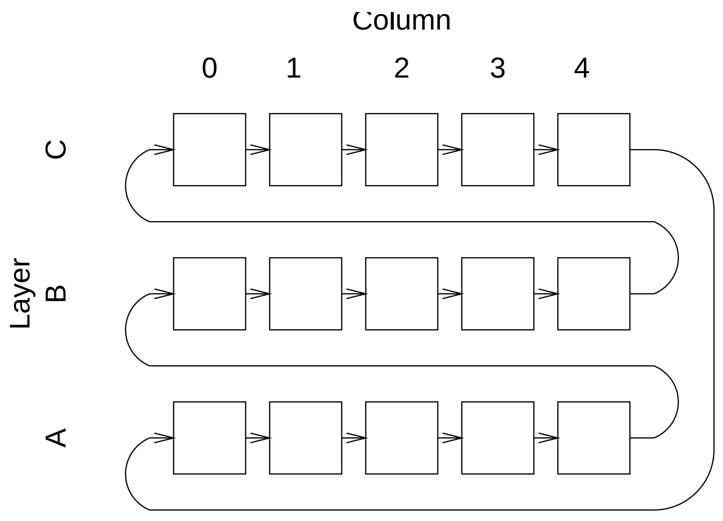


Figure 6.7: Issuer Token Path

6.4.3 The Implementation

Internally each block is divided into three sub-blocks as shown in figure 6.8. The decode block performs a combinational function to decide which of the three actions wait, pass, or issue should be carried out. The sequence block is an asynchronous sequential circuit that operates according to the required action in response to the arrival of a token, driving the functional unit and crossbar outputs when necessary. The token block is responsible for receiving tokens from the previous stage and sending them to the next stage under control of the sequence block.

Token Block

For reasons described in section 6.1.3 the majority of the SCALP design has been carried out using the 4-phase signalling protocol. In the case of the token block a 4-phase implementation was carried out, but it was subsequently found that a 2-phase implementation can be both smaller and faster[1]. The design is shown in figure 6.9. Note

1. One reason why the 4-phase implementation was more complex is that it needed acknowledge signals for the tokens. The 2-phase circuit manages without them.

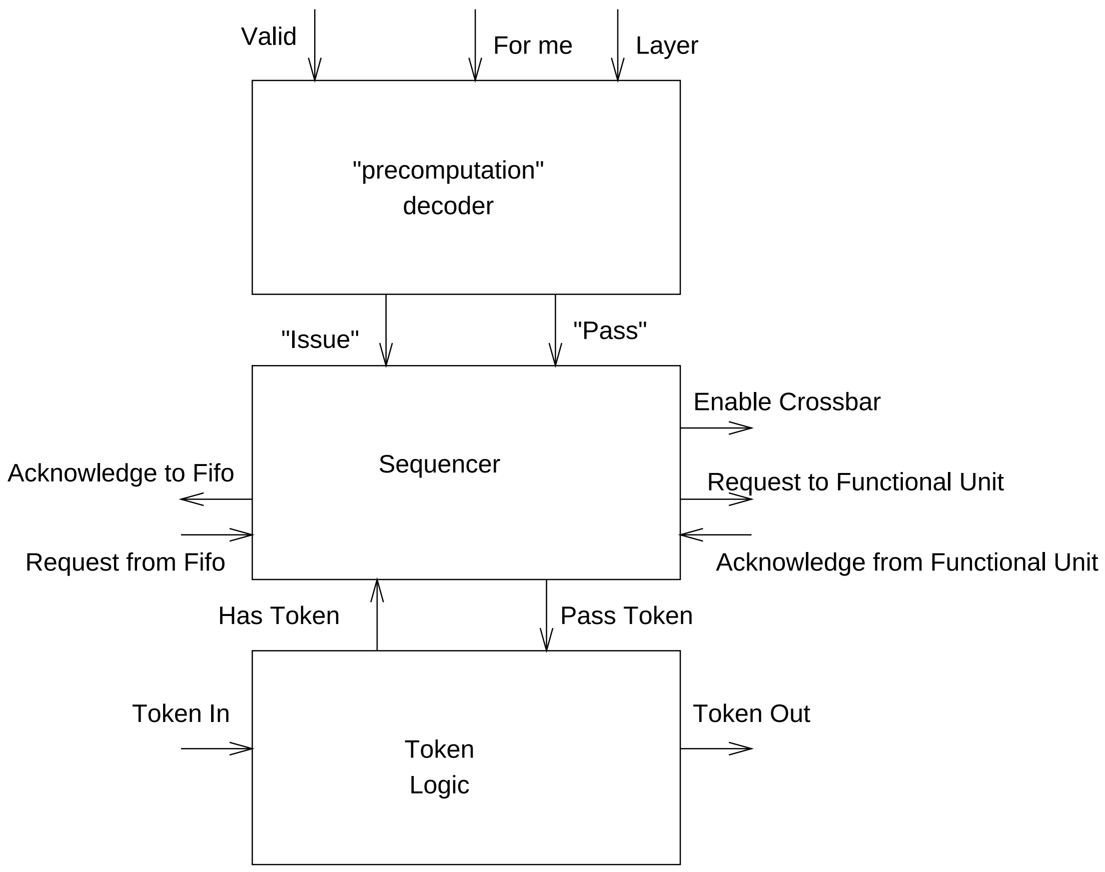


Figure 6.8: Issuer Cell Internal Organisation

that it is only the token_in and token_out signals that are 2-phase; the has_token and pass_token signals are 4-phase.

Operation is as follows: when no token is present, the input and output are the same and the output of the XOR gate is 0. When a token arrives on the input, the XOR gate output becomes high asserting has_token. When pass_token is asserted the latch is opened and the transition representing the token is passed to the output. Both inputs to the XOR gate are equal once more and has_token returns to zero.

Note that with this circuit pass_token may be asserted before a token has arrived, indicating that the token should be passed on as soon as it arrives. In this case the token may pass from block to block with each block imposing only a single gate delay.

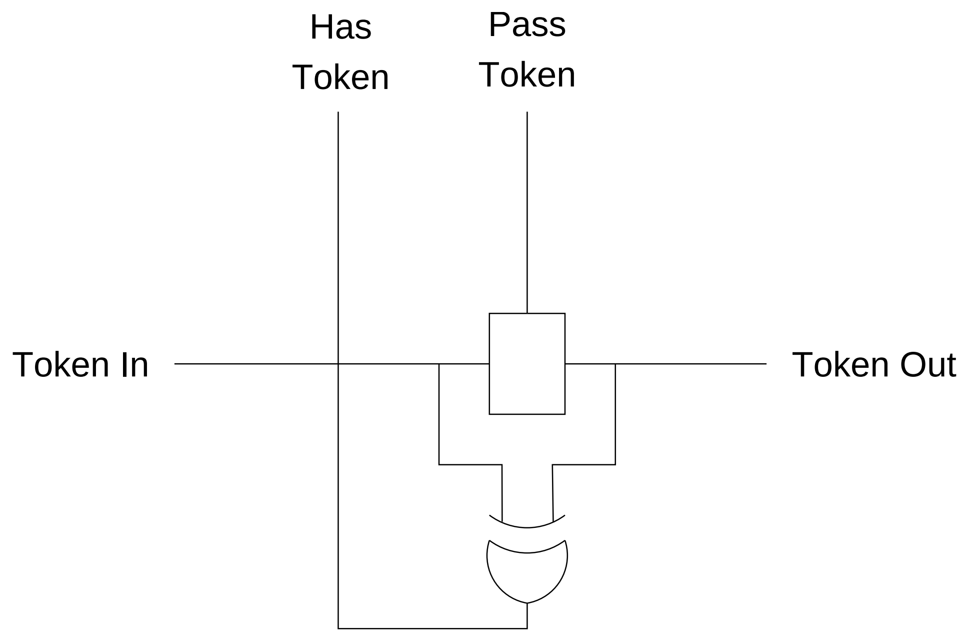


Figure 6.9: Token Block Implementation

Decode Block

The purpose of the decode block is to decide on the basis of the for_me, layer, and valid signals what should happen when the token arrives. It has two outputs, “issue” and “pass”. “issue” indicates that when the token arrives the instruction in this column should be issued and the token passed on. “pass” indicates that when the token arrives it should be passed on without issuing anything. If neither is asserted, when the token arrives nothing should happen i.e. the block waits. The function that the block has to perform is described in table 6.2.

This function must be implemented so that there are no glitches on the issue and pass outputs for any valid input change sequences. An implementation is shown in figure 6.10.

Inputs			Interpretation	Outputs	
Valid	Layer	For_me		Issue	Pass
X	Previous	X	Can’t see the instruction we want; wait	0	0
0	This	X	Must wait until this instruction is valid to see if it is for me; wait	0	0
1	This	0	This instruction not for me; pass	0	1
1	This	1	For me; issue	1	0
X	Next	X	The instruction that was here has been issued by another controller; pass	0	1

Table 6.2: Issuer Decode Block Function

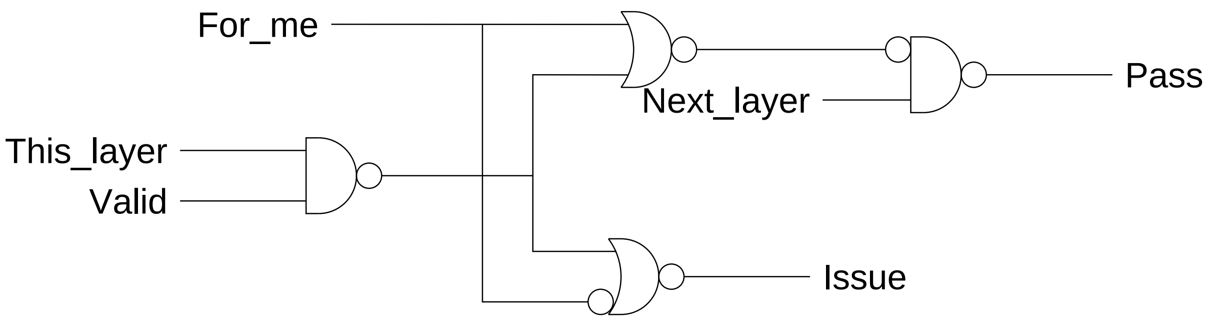


Figure 6.10: Decode Block Implementation

Sequence Block

The function of the sequence block is to carry out the action indicated by the decode logic when the token arrives. This means either passing the token on by asserting pass_token or issuing an instruction.

Passing is straightforward: pass_token is asserted in response to pass. Issuing the instruction involves more steps and some choices. For example, at what point should the token be passed on? Can it be passed when issuing starts or must it wait until it has completed? If the token is passed when issuing starts performance will be improved but subsequent blocks must check that this issue has finished before starting to issue themselves. This solution takes a simple approach as shown in figure 6.11.

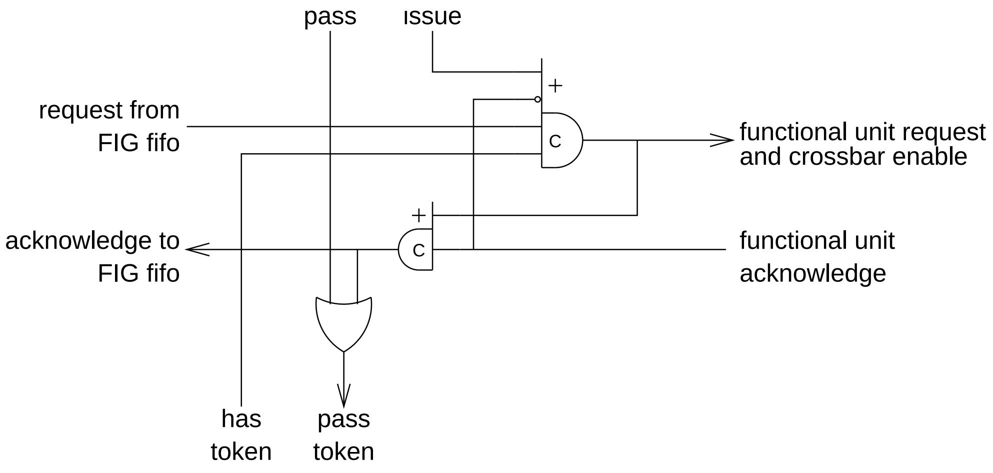


Figure 6.11: Sequence Block Implementation

The four input Muller C element[1] generates the request signal to the functional unit. It is asserted when the following four conditions hold:

- The issue signal from the decode block is asserted.
- The acknowledge signal from the functional unit is not asserted, i.e. any previous issue to this functional unit has completed.
- This block has the token.
- The request signal from the FIG fifo is asserted (this will always be the case when the issue signal from the decode block is asserted).

When the functional unit’s instruction fifo has accepted the instruction it will assert its acknowledge signal. The role of the second Muller C element is to select only those acknowledge signals that result from requests from this block. When its output is asserted two things happen:

1. Asymmetric Muller C elements are represented here by a gate symbol with extensions to one or both sides. Inputs joining the body of the gate take part in both rising and falling transitions of the output. Inputs joining a side marked with a + or - take part in only the rising or falling output transitions respectively.

- The acknowledge signal to the FIG fifo is asserted.

- The token is passed on.

The top Muller C element's output is deasserted when the acknowledge to the FIG fifo and the token passing have completed. This is detected by the request from the FIG fifo and has_token both being deasserted.

6.4.4 Adding Branch Instructions

Branch instructions complicate the instruction issuer in two ways:

- When a branch instruction is taken, instructions between the branch and the branch target must be disposed of. This task is the responsibility of the instruction issuer. Branch targets are identified as such by means of an additional bit per instruction introduced by the fetcher. The branch unit indicates whether a branch has been taken when it acknowledges the instruction.

- In order to not issue instructions that should have been disposed of, controllers for other functional units must not consider instructions beyond the instruction currently being considered by the branch unit's controller.

The first of these complications is dealt with as follows:

- The branch unit controller has an additional token signal between each block. The existing signal transfers "look" tokens and the additional signal transfers "delete" tokens.

- When a "look" token is received the block considers the current instruction in the normal way. If the instruction is issued and the branch unit acknowledges indicating that the branch was not taken then a "look" token is passed to the next block as normal.

- If the branch unit acknowledges indicating that the branch was taken then a “delete” token is passed to the next block.

- When a “delete” token is received the block studies the bit indicating whether this instruction is a branch target. If it is not a branch target then the instruction is discarded by asserting acknowledge to the FIG fifo. A “delete” token is then passed to the next block.

- If a “delete” token is received and the current instruction is a branch target then the instruction is considered as normal and a “look” token is passed to the next block.

The second complication of preventing other controllers from overtaking the branch controller is dealt with as follows: the token block in each block contains additional logic that delays the arrival of a new token when the branch unit controller has a token in the same column and layer. At reset the branch unit controller is given a “head start”. In this way the other controllers all follow behind the branch controller.

6.4.5 Adding Variable Length Instructions

Each of the five chunks of a FIG may be either an instruction or an immediate value associated with the previous chunk. Which instructions have immediate chunks is indicated by the FIG’s control field (see figure 5.9).

Three of the functional units (ALU, load/store, and branch) accept instructions with optional immediates. These functional units have double-width instructions queues that can accept one or two chunk instructions.

There are three main ways in which the instruction issuer must be extended to deal with the variable length instructions:

- The crossbar is extended to allow each of the input chunks to be directed to either the instruction or the immediate part of each instruction queue.

- The control field is decoded to provide an additional bit for each column of the FIG fifo indicating if the instruction in this column has an associated immediate. Each decode block is then extended to decode two new actions: issue with immediate and pass with immediate (in addition to the existing issue, pass and wait). The sequence blocks are extended to deal with these extra actions.

- The token logic is extended to allow tokens to be passed to either the next block or to the block after next. This is required so that immediate fields are skipped over and not treated as instructions. Additional token signals are required between the blocks. Incoming tokens are merged with an XOR gate. Outgoing tokens can be steered to either of the next two blocks.

The effect of these changes is significant both in terms of the size and the performance of the issuer. This is studied further in the next section.

6.4.6 Performance of the Instruction Issuer

As with many types of design the performance of the instruction issuer is determined by its architectural performance and by its implementation performance. Architectural performance refers to the number of instructions that can be issued per cycle. Implementation performance refers to the time taken per cycle.

Architectural Performance

The sequential issuer of figure 6.5 operates at 1 instruction per cycle. The parallel issuer described here can operate at up to 5 instructions per cycle. However this performance can only be achieved when the instructions within each FIG are evenly distributed between the functional units. When the instructions are not evenly distributed the performance will be limited by the controllers for the more heavily used functional units.

Other issuer architectures are possible; in particular it is possible to make the issuer wider or narrower so that more or fewer than five instructions can be considered at a time.

A simulation has been used to measure the performance of the issuer described here for different instruction mixes and for different widths of instruction issuer. The results of the simulation are shown in table 6.3 and figure 6.12.

Organisation	Instructions	Performance / chunks per cycle
Sequential issuer (figure 6.5)	Any	1.00
As described: 5 columns	Best case	5.00
	Evenly distributed random instructions	2.92
	qsort program (see section 7.1.1)	1.86
2 columns	Evenly distributed random instructions	1.67
3 columns		2.23
4 columns		2.63
5 columns		2.92
10 columns		3.78
15 columns		4.27
20 columns		4.58
25 columns		4.83

Table 6.3: Instruction Issuer Architectural Performance

It can be seen from these results that the issuer’s performance could be substantially increased by increasing the number of instructions that are considered at a time, though this would be at the cost of substantially increasing its size. If the instruction issuer was a bottleneck in the processor as a whole then this may be an attractive option. Subsequent results (section 7.3) will show that this is not the case and that other parts of the processor limit its performance.

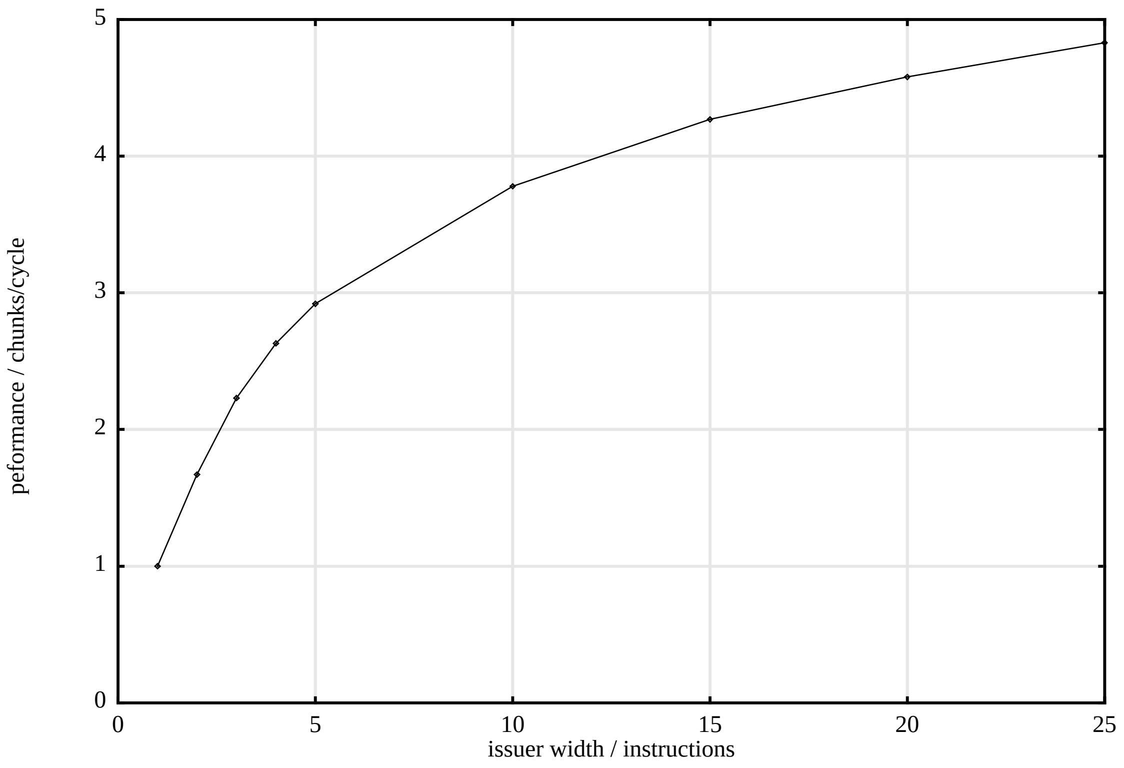


Figure 6.12: Instruction Issuer Architectural Performance

Implementation Performance

This section briefly compares the cycle times of the sequential issuer of figure 6.5, the simplified issuer described in sections 6.4.2 and 6.4.3 and the issuer as implemented with additional logic for branch instructions and variable length instructions.

The sequential issuer has not been modelled at gate level but an estimate of the cycle time is around 8 gate delays (GD) per instruction. This is equivalent to 40 GD per instruction per functional unit. This does not allow for branch or variable length instructions.

The simplified parallel issuer’s cycle time would be as little as 12 GD per instruction per functional unit if its performance was limited by the speed at which tokens are passed from block to block. In practice the critical path lies from the output stage of the FIG fifo, through the modulo-3 counters to the decode blocks, through the sequence blocks in issue mode to the input stages of the functional unit instruction queues, and back through the sequence blocks to the FIG fifo acknowledge. This limits the cycle time to around 24 GD per instruction per functional unit.

The performance of the issuer with support for branch instructions and variable length instructions is measured in detail in section 7.3.1. Its cycle time is at best 54 GD per instruction per functional unit.

The difference between the 24 GD cycle time of the simplified issuer and the 54 GD cycle time of the real issuer can be attributed to the additional complexity of dealing with branches and variable length instructions.

6.5 Implementation of the Result Routing Network

The result routing network transfers values from the functional unit outputs to the operand queues of other functional units. Each value has associated with it a bit indicating its width, byte or word, and three bits indicating its destination. The routing network has four inputs - one from each of the functional units except the branch unit - and eight outputs.

As with the instruction issuer an important initial question is the degree of parallelism that is required. A simple sequential design (figure 6.13) could be implemented that processes only one value at a time. At the other extreme a fully populated crossbar (figure 6.14) could be used to provide the greatest performance but with significant hardware cost. Initial analysis suggested that a balanced solution would lie between these two extremes. The proposed structure is shown in figure 6.15.
The datapath comprises two shared busses. Each of the four network inputs is connected to both busses by a tri-state buffer, and each bus is connected to each output in the same way.

Control is divided into two parts associated with the input side and the output side respectively. The input side is responsible for arbitrating between the four inputs and granting access to the shared busses. The output side is less complex; it studies the

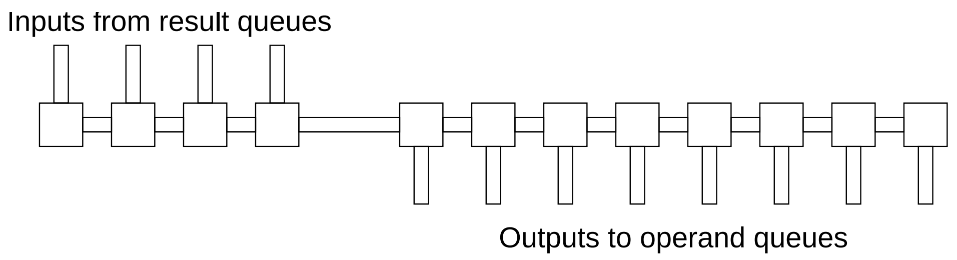


Figure 6.13: Single Bus Network

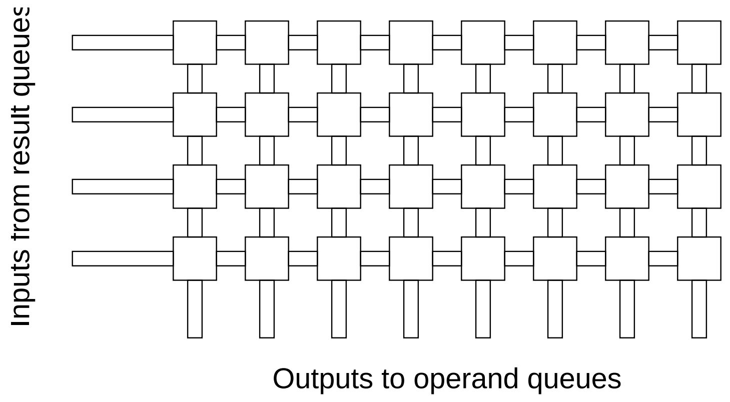


Figure 6.14: Fully Populated Crossbar Network

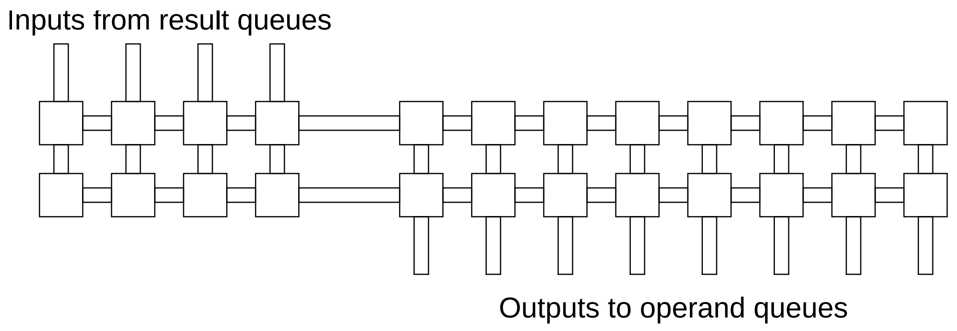


Figure 6.15: Two Bus Network

destination specifiers on the busses and activates tristate connections and output requests as necessary.

Arbitration

A complex arbiter is required to control the input side of the controller. It has four request inputs corresponding to each of the inputs to the network. Associated with each request input are two grant outputs. In response to a request input being asserted one of the grant outputs will eventually be asserted. This indicates that that input may drive the associated bus. The grant output will remain asserted until the request is deasserted. No more than one input may be granted use of each bus at any time.

The basic arbitration element available for the design is the two input mutual exclusion element. This element has two request inputs and two grant outputs. When one request input is asserted the corresponding grant output is subsequently asserted. When both request inputs are asserted simultaneously the element resolves any metastability internally and eventually asserts exactly one grant signal. It is guaranteed that no metastable activity will be observed on the output signals, though the arbiter may take an arbitrarily long time before asserting either signal.

From these two input mutual exclusion elements it is necessary to construct a four input mutual exclusion element. One such element will be associated with each bus in the controller. The circuit used for this purpose is shown in figure 6.16.

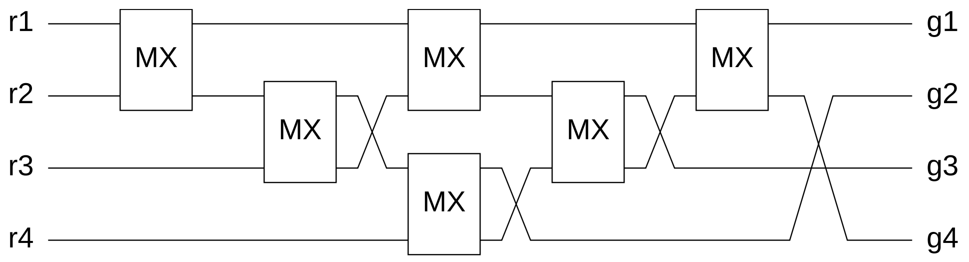


Figure 6.16: Four way mutual exclusion circuit built from two way mutual exclusion elements

The operation of the overall controller is as follows: When a request is received, the corresponding requests to both four way mutual exclusion elements are asserted.

Eventually one or other (or both) will assert the corresponding grant signal. A two way mutual exclusion element is used to arbitrate between the two grant signals so that only one of the grant inputs is asserted. The request to the other four way mutual exclusion element is then released.

The circuit used is shown in figure 6.17.

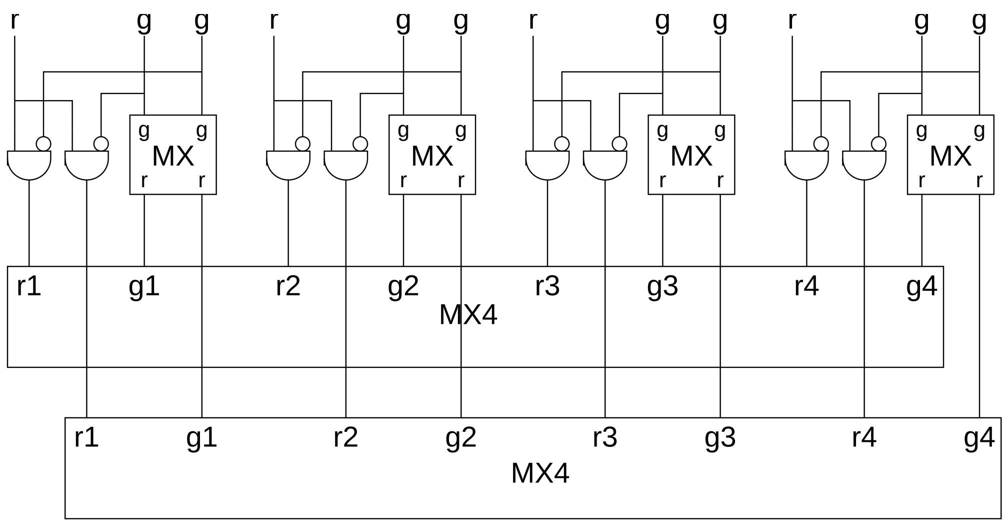


Figure 6.17: Arbiter for Result Network

Other Organisations

The network organisations considered here are all symmetric, i.e. all outputs have equal access to all busses. More efficient solutions may be provided by networks that are biased towards some patterns of communication and against others. Figure 6.18 shows such an arrangement. One input has exclusive use of a bus. Two other busses are shared by two other inputs, with the fourth input given access to only one of them. On the output side two of the busses are connected to all outputs but the other has limited connectivity.

The interconnections in a scheme of this sort would have to be based on analysis of network usage by benchmark programs.

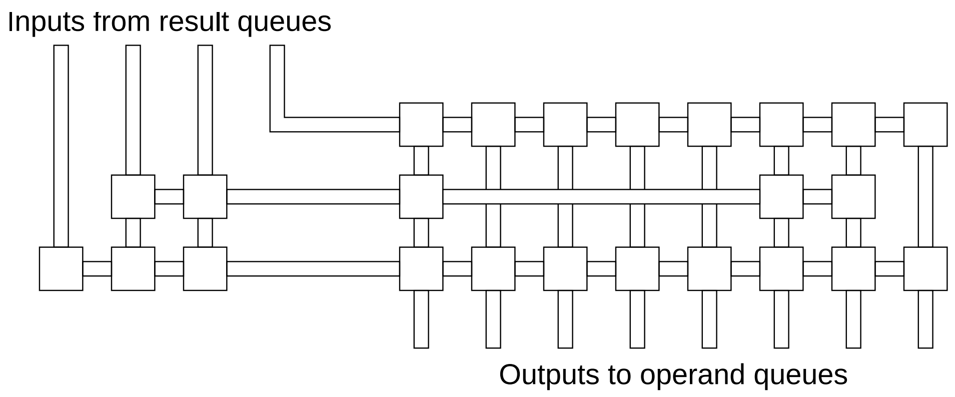


Figure 6.18: An Asymmetric Result Network

6.6 Functional Units

Although each of the five functional units performs a different function, in terms of structure they have much in common.

The interfaces to the units are similar. Each has an input for instructions from the issuer and for one or more operands from the result network. They also have similar outputs to send results to the result network.

Internally, the functional units are divided into datapath and control sections as shown in figure 6.19. The datapaths comprise multiplexors, adders and other logic connecting the inputs and outputs under the control of various signals from the control logic.

The structure of the control logic in each functional unit is similar; it comprises instruction decoding, input synchronisation and sequencing.

The decoder studies the instruction and generates a number of control signals; some of these go to the datapath. Other signals indicate which of the various operands are needed by this instruction.

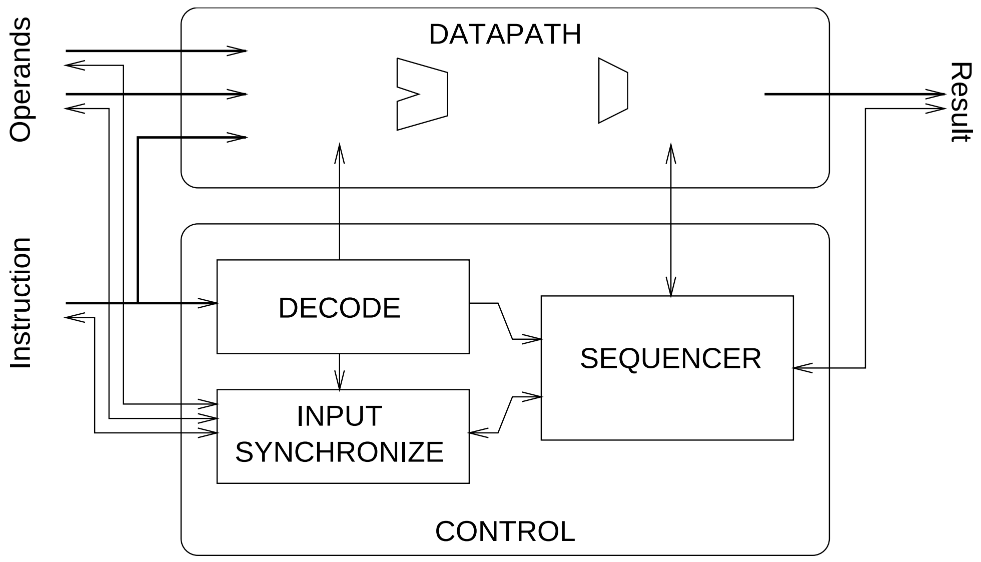


Figure 6.19: Function Units General Structure

The role of the input synchroniser is to establish when all of the necessary operands and the instruction have arrived. When this occurs it indicates the fact to the sequencer.

The sequencer is responsible for activating any sequential logic in the datapath such as an adder and for sending the result to the output; this part of the design varies substantially from one functional unit to another.

As an illustration, figure 6.20 shows the arrangement of the sequencer for the branch unit.

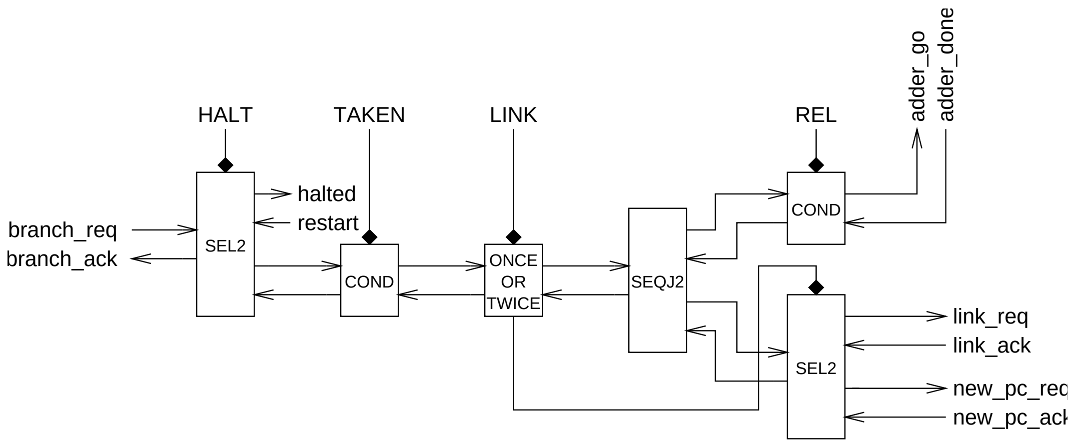


Figure 6.20: Branch Unit Sequencer

Its operation is as follows: the instruction is decoded to provide the HALT, TAKEN, LINK and REL inputs which indicate that the branch is a halt instruction, is taken, is a branch and link instruction and is a relative branch respectively. When any necessary operands have arrived the branch_req input from the input synchroniser is asserted.

The first block is a two-way select block that deals with the HALT instruction. For halt instructions the external output “halted” is asserted and the sequencer awaits the “restart” input before continuing.

For all other instructions the request is passed to the next block. For untaken branches the conditional block returns the request directly to the acknowledge. For taken branches the request proceeds to the next block.

Most branch instructions execute in a single “cycle”. The exception is the branch and link instruction which uses one cycle to compute the target address and a second cycle to compute the link address (either the address of the current instruction or the address of the current instruction plus one). The “once or twice” block cycles its output twice for link instructions and once for others.

The next block is a simple sequencer that activates each of its two outputs in turn. The purpose of the upper (first) output is to activate the adder when necessary. The adder is only required by relative branches so a second COND block enables it only in this case.

The second output of the sequencer is used to send the result from the datapath to its destination. This may be either the link address queue for the link cycle of a branch and link instruction or the new PC queue of the fetcher for other cycles. An additional output from the earlier once or twice block controls a select block that steers the signal to the appropriate output.

6.7 Size of the Implementation

The number of basic gates modelled in the VHDL code has been counted and is broken down by gate type in table 6.4. The total number of components is around 9,500. The number of gates used by each block is shown in table 6.5[1].

Count	Gate	Description
220	and2	AND gates
131	and3	
1	and4	
181	dffr	D-type flip-flop
1334	inv	Inverter
7	mc101	Muller-C elements; first digit is number of symmetric inputs, second is number of positive-only inputs, third is number of negative-only inputs. R suffix indicates global reset. In a VLSI implementation some of the larger gates would be implemented from several smaller components.
6	mc101R	
6	mc102	
3	mc102R	
1	mc103	
174	mc110	
1	mc110R	
10	mc111	
10	mc112R	
195	mc120	
12	mc120R	
98	mc2	
24	mc201R	
90	mc220	

Table 6.4: Total Gate Counts (Part 1 of 2)

1. In table 6.5 the RAM part of the register bank is counted as only two components. To provide a fairer idea of the register bank's relative size in table 6.6 an estimate of the equivalent number of components is given.

Count	Gate	Description
25	mc2R	
1	mc3	
75	mc320	
18	mutex	Mutual exclusion (arbitration) element
2397	nand2	NAND gates
551	nand3	
42	nand4	
497	nor2	NOR gates
66	nor3	
59	nor4	
454	or2	OR gates
70	or3	
15	or4	
1	ram32x24	RAM blocks (register bank)
1	ram32x8	
1365	TL	Transparent latches
82	TLR	
5	TLsetR	
838	tridrv	Tri-state driver
32	xnor2	Exclusive OR gates
402	xor2	

Table 6.4: Total Gate Counts (Part 2 of 2)

The relative sizes of the various blocks are indicated in figure 6.21.

The total number of transistors in the design is approximately 64,000. This is compared with the sizes of AMULET1 and the ARM6 processor in table 6.6.

Block	Gates	Proportion
Issuer (of which Switches Control	4165 550 3615	39 % 5 % 34 %)
Result Network	1194	11 %
Register Bank	1158	11 %
ALU	1090	10 %
Operand Queues	624	6 %
Branch Unit	548	5 %
Load/Store Unit	488	5 %
Fetcher	422	4 %
Move Unit	215	2 %
FIG Fifo	228	2 %
Instruction Queues	176	2 %
Result Queues	168	2 %
Memory Interface	106	1 %

Table 6.5: Gates per Block

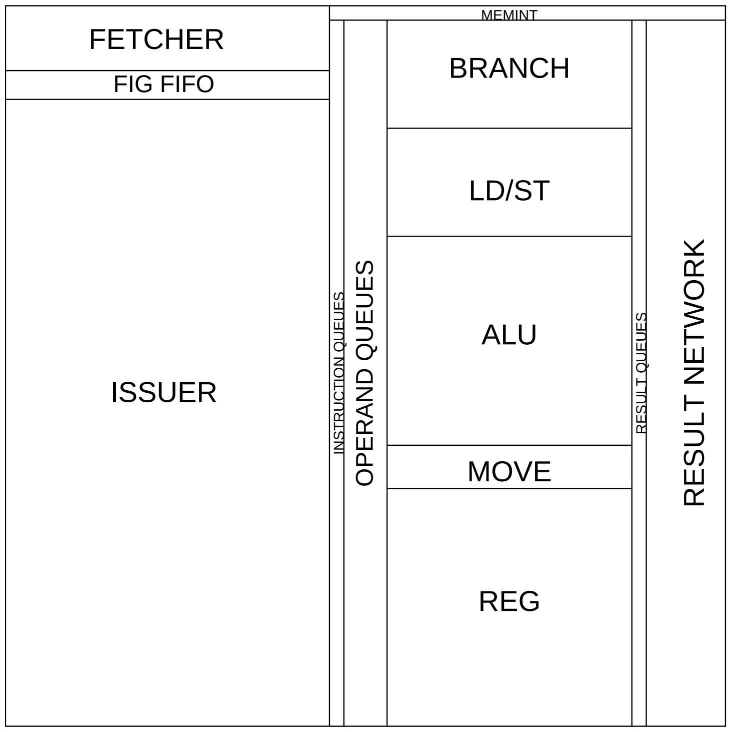


Figure 6.21: SCALP Block Sizes

Processor	Transistors
SCALP	64000
AMULET1	54000
ARM6	34000

Table 6.6: SCALP, AMULET and ARM sizes

Chapter 7: Evaluation

This chapter investigates the SCALP architecture and implementation to see how well it meets its stated objective high power efficiency through high code density, parallelism and asynchronous implementation. The first sections introduce tools and example programs that have been used to carry out the evaluation. Subsequent sections describe the results of the evaluation.

7.1 Evaluation Procedure

The gate level VHDL model was evaluated as follows: a number of forms of test stimulus were developed to exercise the model in interesting or representative ways. The model was then extended by the addition of instrumentation code that reports internal activity during execution of the stimulus. The activity trace thus obtained was subsequently analysed by a number of evaluation programs to report statistics about the behaviour of the processor.

7.1.1 Test Stimuli

The test stimuli used take the form of example programs and instruction sequences.

Instruction Sequences

Six instruction sequences have been written with different properties as described below. The sequences each execute approximately 1000 instructions. They are not “real programs” in the sense that they do not carry out any useful work.

(i) Distributed independent instructions (dis-ind)

An instruction sequence comprising instructions well distributed between the functional units with the minimum of data dependencies. There are no taken branches.

(ii) Non-distributed independent instructions (ndis-ind)

An instruction sequence comprising instructions that concentrate on one functional unit for some time making it a bottleneck before moving to the next. Few data dependencies exist between the instructions. There are no taken branches.

(iii) Distributed dependent instructions (dis-dep)

An instruction sequence comprising instructions well distributed among the functional units with many data dependencies. There are no taken branches.

(iv) Non-distributed dependent instructions (ndis-dep)

An instruction sequence comprising instructions that concentrate on one functional unit for some time making it a bottleneck before moving to the next. There are many data dependencies between the instructions. There are no taken branches.

(v) Independent branches (ind-br)

An instruction sequence comprising instructions well distributed among the functional units with few data dependencies. One instruction in five is an unconditionally taken branch.

(vi) Dependent branches (dep-br)

An instruction sequence comprising instructions well distributed among the functional units with many data dependencies. One instruction in five is a taken branch which is dependent on the preceding instructions.

Example Programs

The lack of a high level language compiler for SCALP makes evaluation with real programs difficult. Five simple example programs have been written in SCALP assembly language. Brief descriptions of the programs follow.

(i) Shading

A short piece of code that implements the inner loop of the Gouraud shading algorithm: a number of sequential memory locations are loaded with values that differ by a constant amount.

(ii) Word Count

A loop whose function resembles the unix “wc” program: the number of words and lines in an area of memory are counted.

(iii) String Tree Search

A binary tree with string key values is searched recursively for values that may or may not be present.

(iv) Quicksort

The O(n log n) recursive sorting algorithm is applied to an array of integers.

(v) Sumlist

Integer values in a null-terminated linked list are summed.

For comparison the same programs have also been written in assembly language for two conventional processors, ARM and “ARMLESS”. ARMLESS is a hypothetical processor whose instruction set is a subset of the ARM similar to the SPARC processor. Its properties are

- All instructions are 32 bits long.
- 14 general purpose registers.
- Instructions specify three registers or two registers and an immediate for sources and destinations.
- Only load and store instructions access memory.
- Branch instructions may be conditional on the value of the condition codes.

The ARM instruction set [FURB89] also features:

- All instructions may be conditional on the value of the condition codes.
- Load and store instructions may write incremented base register values back to the register bank.
- Load and store multiple instructions may transfer any group of registers to or from memory in a single instruction.
- Load and load multiple instructions may transfer the program counter to effect a branch.

7.1.2 Evaluation Programs

A number of programs have been written to analyse the output of the instrumented VHDL model.

Pipeline Visualisation

Being able to see graphically the flow of data within a processor can be a great help to understanding limits on performance such as bottlenecks and dependencies. This is especially true in asynchronous systems where time domain behaviour is less constrained than in a synchronous design.

Other work has applied visualisation to asynchronous pipelines executing SPARC code [CHIE95] and to the AMULET processor [SWAI95].

A visualisation program, xpipevis, has been developed to show SCALP's internal data flow graphically. Using this tool it is possible to see where bottlenecks occur and what the processor's typical behaviour is. The program displays a block diagram of the processor on which each bus is illuminated using a colour corresponding to its state: idle, busy, or reseting. The visualisation can run either at a constant rate relative to the simulated real time or in "single step" mode where the user presses a key to advance the display. A snapshot of the typical output from the program is shown in figure 7.1.

xpipevis has proved useful for optimising the SCALP implementation, for example when choosing the number of stages in the instruction and operand queues and the configuration of the result network.

Resource Utilisation

A common objective in parallel system design is to obtain a high utilisation of the available resources. If some resources are rarely or never being used the design may be optimised by removing them; on the other hand if a particular resource is often fully used it may limit the performance of the system as a whole.

A program has been written to measure the utilisation of the functional units, the result network busses and the queue stages in the instruction fifo, functional unit instruction and operand queues, and the functional unit result queues.

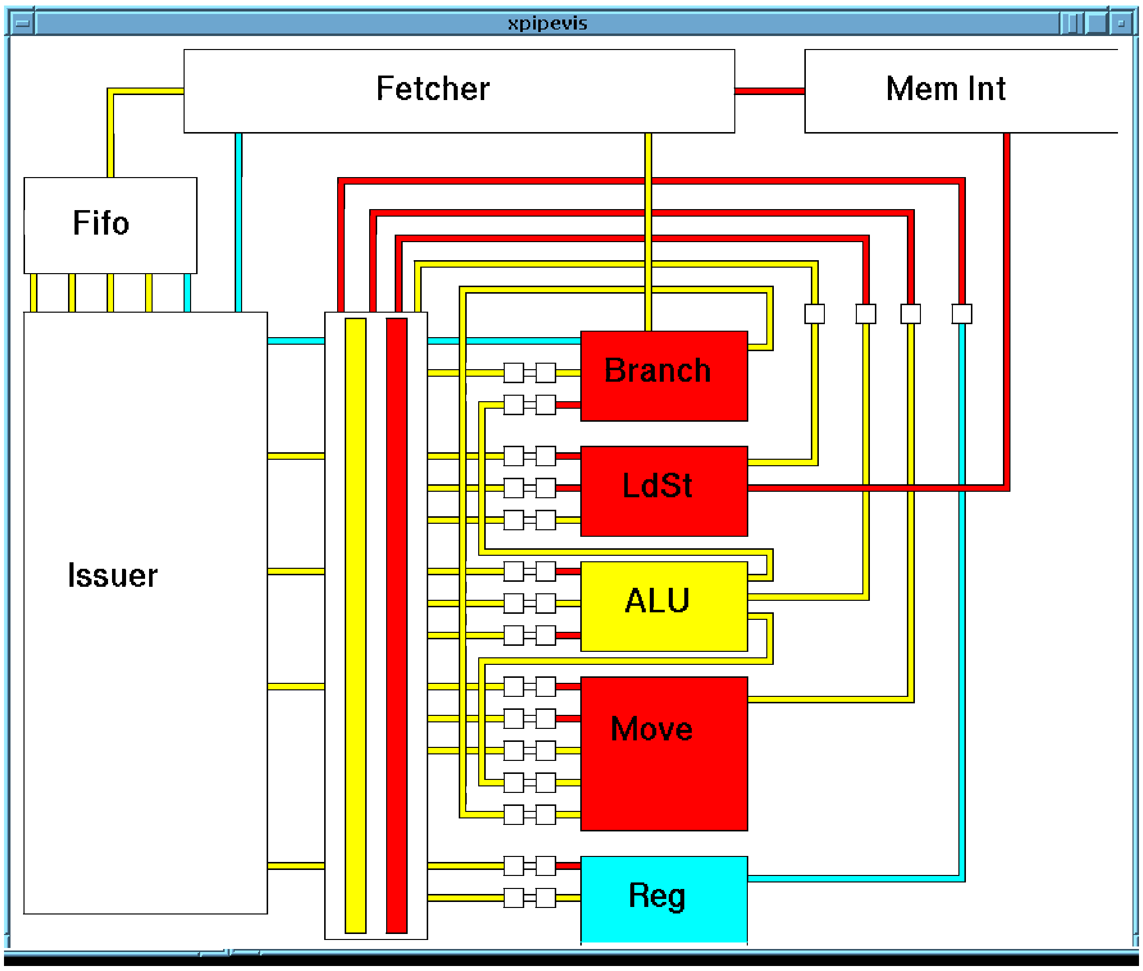


Figure 7.1: Pipeline Visualisation

Instruction Rate

A simple program has been written to measure the processor’s instruction execution rate in terms of instructions per second.

7.1.3 Use of the Geometric Mean

Where a set of benchmark programs is used to evaluate the performance of a processor it is useful to be able to compute a summary value indicating overall performance. If the benchmark programs constitute a representative workload for the processor then the arithmetic mean of the individual execution times can be used to predict the total execution time for the workload.

In some cases the benchmarks used are artificial or do not constitute a real workload. In this case no measure can indicate real performance but some sort of summary value is still useful. The arithmetic mean has the particular disadvantage that the benchmarks whose execution times are greatest will dominate the result. In contrast the geometric mean gives equal importance to all benchmarks.

Consider the data in table 7.1, which shows execution times for two programs A and B on two machines X and Y. X does twice as well as Y on A, but Y does twice as well as X on B, so in some sense they are equally good overall. This is indicated by their equal geometric means. The arithmetic mean on the other hand is biased by the fact that A takes less time than B on both machines.

	Machine X	Machine Y
Program A	10	20
Program B	100	50
Arithmetic Mean	55	35
Geometric Mean	31.6	31.6

Table 7.1: Illustrating the use of the Geometric Mean

[HENN90], pages 50-53, discusses the relative merits of the arithmetic and geometric means for this type of analysis.

7.2 Code Density

In this section the code density achieved by SCALP is evaluated. High code density was an objective of the design because the reduced memory bandwidth resulting from high code density is beneficial to power efficiency.

Tables 7.2 and 7.3 present the static and dynamic code sizes measured for SCALP and the conventional processors ARM and “ARMLESS”, using the example programs described in section 7.1.1.

Program	SCALP	ARM	ARMLESS
Shading	37	44	48
Word Count	54	48	56
String Tree Search	126	116	136
Quicksort	189	168	200
Sumlist	27	32	36
Geometric Mean	66.3	66.7	76.6

Table 7.2: Static Code Size / bytes

Program	SCALP	ARM	ARMLESS
Shading	1024	1024	1280
Word Count	3290	3128	3656
String Tree Search	382	420	500
Quicksort	13194	5908	6648
Sumlist	315	380	464
Geometric Mean	1398	1247	1485

Table 7.3: Dynamic Code Size / bytes

In terms of static code density, SCALP does about as well as the ARM processor and about 13% better than “ARMLESS”.

In terms of dynamic code density, SCALP does about 12% worse than ARM and about 6% better than “ARMLESS”.

These results are worse than expected and result from a greater than expected overhead of register bank and move unit instructions. These instructions are required to move operands when the explicit forwarding mechanism is insufficient. Table 7.4 shows the number of register bank or move unit instructions required per “useful” instruction.

Benchmark	Instructions
Shading	0.99
Word Count	0.72
String Tree Search	1.06
Quicksort	1.34
Sumlist	0.61
Geometric Mean	0.91

Table 7.4: Register Bank and Move Unit Instructions per Useful Instruction

The large number of these instructions can be attributed to a flaw in the explicit forwarding mechanism that makes it less useful to the programmer than the results concerning the frequency of forwarding cited in table 2.3 might lead one to expect. The consequence is that many results are written to the register bank immediately after being computed and are read from there before each use.

This failing of the instruction set explained further in section 7.8.

7.3 Pipeline Performance

This section investigates the throughput of SCALP's pipeline and the parallelism that it is able to achieve between stages. Firstly the cycle times of the individual pipeline stages are considered and then the processor as a whole is examined using the instruction sequences and example programs described in section 7.1.1.

Times in this section are generally expressed in terms of standard gate delays (GD). These can be related to real execution times if a particular technology is considered. Section 7.5 performs this comparison.

7.3.1 Performance of Isolated Stages

FIG Fetch

The FIG fetch stage comprises three components: the fetcher, the memory interface and the memory itself. In the test environment the fetched FIGs are fed to the FIG fifo, but the output of the fifo is "short circuited" so that its rate of consumption of FIGs is maximised. No load or store activity is present.

The cycle time of the FIG fetcher in the absence of branch instructions has been measured and is shown in table 7.5.

Cycle time / GD
117

Table 7.5: FIG fetch performance, no branches

Instruction Issue

The performance of the instruction issuer has been measured by connecting it between the FIG fifo and the functional unit instruction queues as normal but with the other ends of these queues short circuited to provide maximum performance. The performance is dependent on the instructions being issued; when the instructions are evenly distributed among the functional units the performance will be greater than when the instructions are clustered within a subset of the functional units. The cycle times for various representative FIGs are shown in table 7.6. All branch instructions are not taken.

The cycle time when deleting instructions skipped over after a taken branch has also been measured and is also shown.

Instructions (A-E=functional units, I=immediate)					Cycle time / GD
Chunk 0	Chunk 1	Chunk 2	Chunk 3	Chunk 4	
A	B	C	D	I	54
A	A	B	C	D	66
A	A	A	B	C	93
A	A	A	A	B	124
A	A	A	A	A	155
Deleting skipped over instructions					75

Table 7.6: Instruction issuer performance

These results are plotted in figure 7.2. It can be seen that when two or more instructions in a single FIG are for the same functional unit the issuer’s cycle time is approximately 31 GD per instruction for that functional unit. This suggests that in this case the cycle time is limited by the issuer’s output speed. When no functional unit receives more than one instruction from a single FIG the cycle time is 54 GD, presumably limited by the issuer’s input speed.

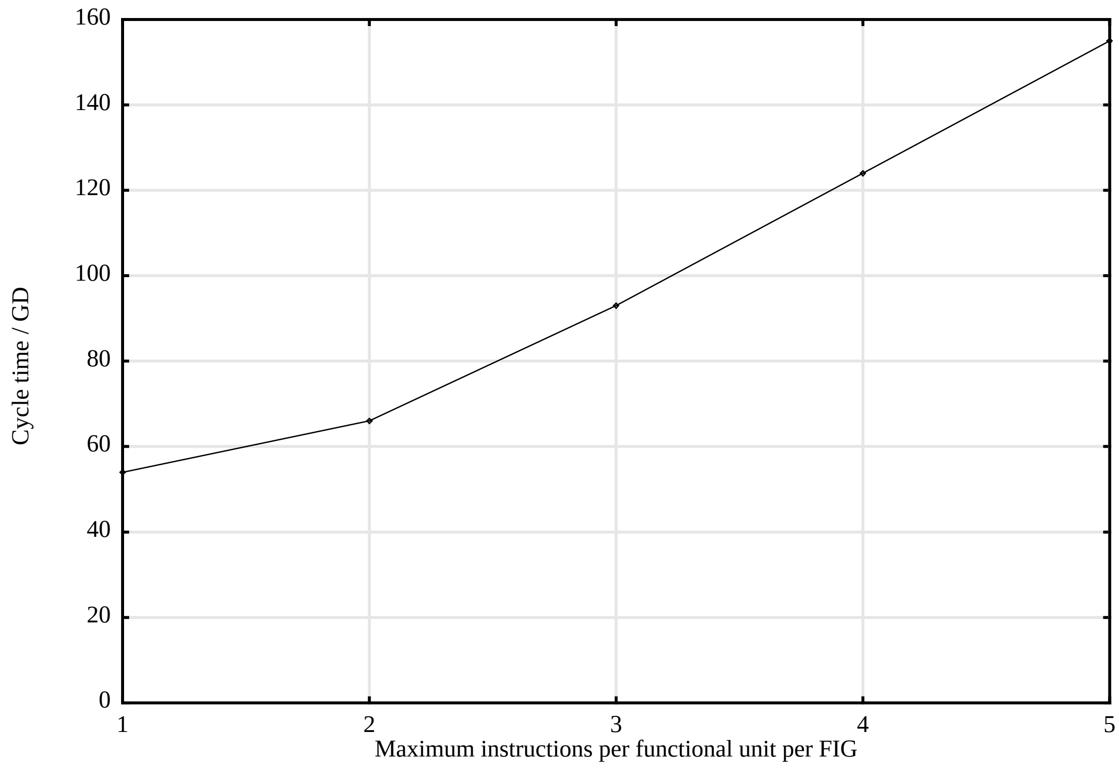


Figure 7.2: Instruction Issuer Performance

Functional Units

The performance of each functional unit has been measured by connecting it between instruction, operand, and result queues as normal but with the other ends of the queues short circuited to provide maximum performance. The load/store unit was connected to the memory via the memory interface but no instruction fetch activity was present to compete for the memory bandwidth. The branch unit's connection to the fetcher was also short circuited. The cycle times for each of the functional units for various representative instructions are shown in table 7.7. For those instructions which involve an addition or similar operation the cycle times were measured for a number of different operands which cause different carry propagation lengths.

Functional Unit	Instruction	Carry Length	Cycle time / GD
ALU	Logical function	-	38
	Logical comparison	-	41
	Arithmetic function	0	47
		4	52
		10	65
		32	107
	Arithmetic comparison	0	49
		4	54
		10	67
		32	109
Move	Duplicate, Sequence	-	70
	Conditional Move, Move Link	-	44
Register Bank	Read	-	42
	Write	-	35
Load / Store	Load	0	79
		32	123
	Store	0	72
		32	116
Branch	Non-taken conditional	-	26
	Taken conditional or unconditional	0	69
		4	69
		10	70
		28	110
	Computed branch	-	48
	Branch and link	0	77

Table 7.7: Functional Unit Performance

Result Network

The performance of the result network has been measured by connecting it between the functional unit result queues and the operand queues as normal but with the ends of these queues short circuited to maximise performance. The result is given in table 7.8.

Cycle Time / GD
47

Table 7.8: Network Performance

7.3.2 Overall Performance

The performance of each of the isolated stages in the absence of taken branches is summarised in table 7.9. This data is also shown in figure 7.3.

Stage	Cycle Time / GD
Instruction fetch (no branches) per FIG	117
Instruction issuer (input side) per FIG	54
Instruction issuer (output side) per instruction, each output	31
ALU (representative arithmetic operation)	54
Move unit (two cycle instruction)	70
Register read	42
Register write	35
Load (representative operation)	84
Store (representative operation)	77
Untaken branch	26
Result network	47

Table 7.9: Performance of Isolated Pipeline Stages

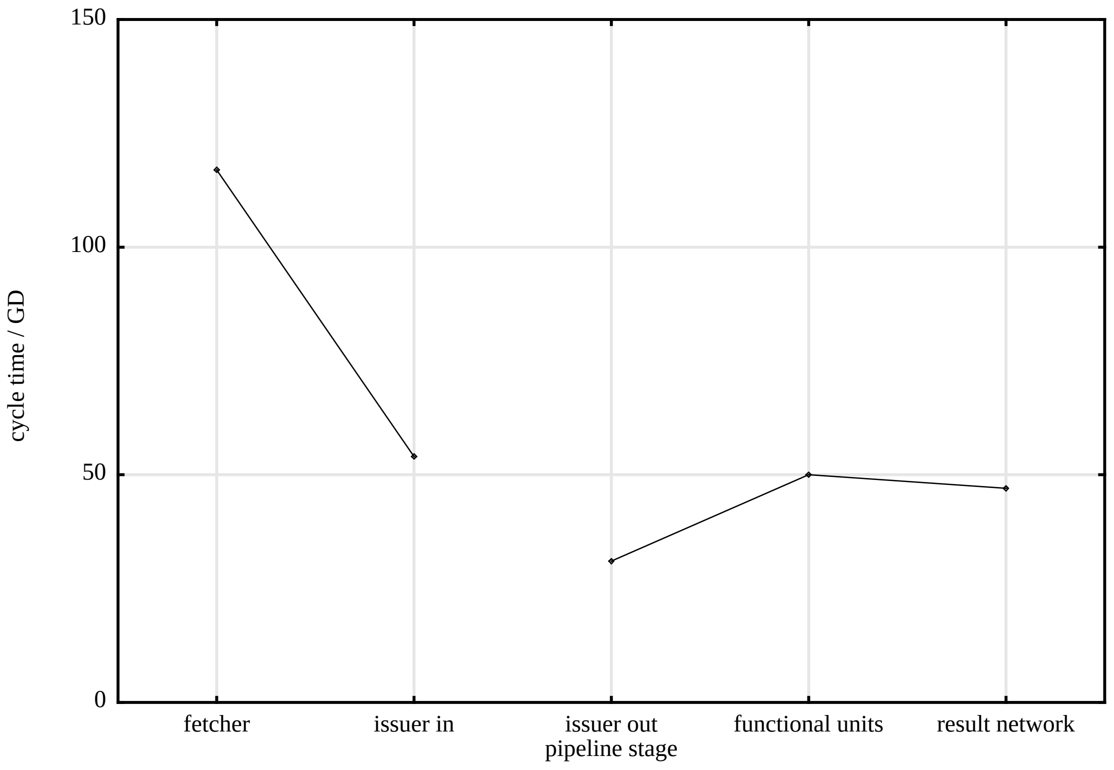


Figure 7.3: Performance of Isolated Pipeline Stages

In the first part of the pipeline which deals with FIGs the throughput is limited by the cycle time of the instruction fetcher of 117 GD.

In the second part of the pipeline which deals with individual instructions throughput is generally limited by the functional units. The issuer output generally does not limit performance because its cycle time of 31 GD is less than nearly all functional unit cycle times. The result network speed is faster than nearly all instructions that send results to it[1].

When the pipeline is considered as a whole the throughput may be limited either by the rate at which the fetcher can provide FIGs or by the rate at which the functional units consume instructions. The particular instruction mix determines where the limit is imposed. As the typical functional unit cycle time is roughly half of the fetcher's cycle time the throughput of the whole pipeline is limited by the functional units if each FIG

1. The exceptions are taken branch instructions which are faster than the instruction issuer output and register read instructions which are faster than the result network.

contains more than two instructions for any functional unit. If each FIG contains two or fewer instructions for each functional unit then performance will be limited by the fetcher.

It is possibly significant that generally the instruction issuer and the result forwarding network do not limit performance as these are the parts of the system that received the greatest design effort. Instead performance is limited by either the instruction fetcher or the functional units whose design was less involved.

Note that this analysis does not consider the performance effect of data dependencies between instructions or branch instructions which are considered later.

Starvation and Blocking

Asynchronous pipeline performance may be degraded by the occurrence of starvation and blocking; that is when individual stages are prevented from operating because the preceding or following stages are not ready to provide or accept data. This problem may be reduced by introducing fifo queues between functional units to decouple them; when one stage performs a slow operation the queue preceding it will fill up, preventing the previous stage from blocking, and the queue following it will empty, preventing the following stage from starving.

SCALP's FIG fifo, instruction queues and result queues perform this decoupling function. [KEAR95] claims that the ideal length for such queues is approximately four times the coefficient of variation of the isolated stage delays. Table 7.10 applies this calculation to the SCALP pipeline. True mean and standard deviation values are dependent on a particular instruction mix; the values used here are rough approximations.

The length of the queue between two stages should be determined by the greater of the ideal queue lengths in the preceding and following stages. Table 7.11 compares these ideal queue lengths with the queue lengths actually implemented.

Pipeline Stage	Mean Cycle Time / GD	Standard Deviation / GD	Coefficient of Variation	Ideal Queue Length
FIG Fetch	119	7	0.1	0
Issuer Input	54	2	0.0	0
Issuer Output	31	4	0.1	0
ALU	57	21	0.4	2
Move Unit	60	13	0.2	1
Load / Store unit	95	20	0.2	1
Register unit	40	3	0.1	0
Result Network	47	25	0.5	2

Table 7.10: Pipeline Decoupling Queue Lengths

Queue	Ideal Length	Actual Length
FIG queue	0	3
ALU instruction queue	2	2
Move instruction queue	1	2
Load / Store instruction queue	1	2
Register bank instruction queue	1	2
ALU result queue	2	1
Move result queue	2	1
Load / Store result queue	2	1
Register bank result queue	2	1

Table 7.11: Ideal and Actual Queue Lengths

These ideal queue lengths must be balanced against other considerations. The large FIG queue helps the instruction issuer to find independent instructions to send to the functional units by considering parts of up to three FIGs simultaneously. On the other hand the shorter than ideal result queues help to reduce the latency of sending results between functional units which is important for code whose performance is limited by data dependencies.

Generally it can be claimed that in view of these results SCALP will not be subject to significant performance degradation resulting from starvation or blocking.

Pipeline Performance with Branches

The foregoing results do not consider the effect on performance of taken branches. When a branch is taken the target PC value must be sent to the fetcher and the target instructions processed by the issuer before any further instructions may be executed by the functional units.

The performance of the fetch and issue logic in the presence of branch instructions was measured by connecting the memory, fetcher, instructions fifo, issuer, functional unit instruction queues and branch unit as normal but with no instructions issued to other functional units. The result is given in table 7.12.

Time between taken branches / GD
241

Table 7.12: Branch Latency

This branch latency is large compared with the cycle times of the functional units: each functional unit could execute four or more instructions in this time. This means that for each taken branch instruction as many as 20 or even more potential instruction execution times are lost. The effect of this on throughput is illustrated in figure 7.4.

This figure shows that even for very long intervals between taken branches the processor’s performance is severely degraded.

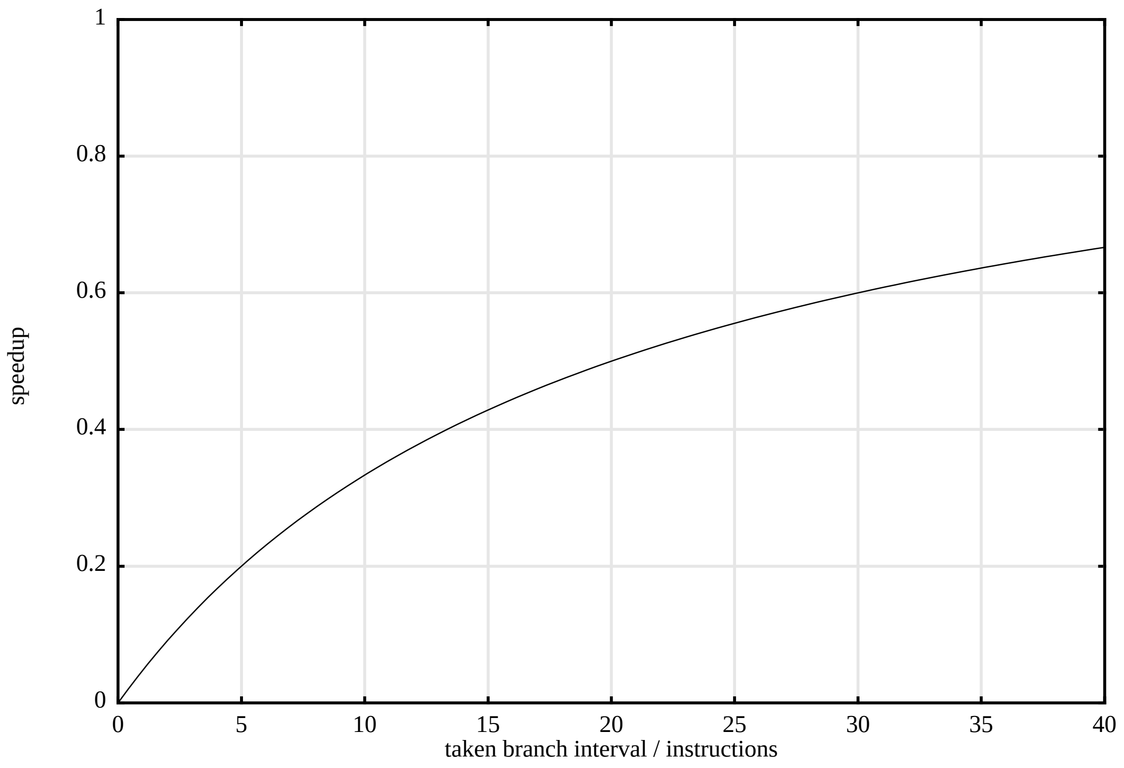


Figure 7.4: Performance and Branch Interval

There are a number of factors that reduce the significance of this problem:

- Some of the execution times after the taken branch may be occupied by independent instructions waiting in other functional units' instruction queues. At most 8 such instructions could be waiting.

- Throughput may be limited by data dependencies or by an imbalance in the distribution of instructions among the functional units. In this case the number of potential instructions lost during the branch latency may be lower.

- As with any superscalar processor, branch prediction could be added to reduce the effect of the branch latency. The benefit of branch prediction is considered in section 8.3.1.

7.3.3 Measured Performance

To validate the results in the previous section and to measure the effect of other factors including data dependencies between instructions the throughput of the processor as a whole was investigated using the examples described in section 7.1.1.

Instruction Sequences

The performance of each of the instruction streams has been measured and the results are shown in table 7.13.

Sequence	Time per instruction / GD	5 x Time per instruction / GD
dis-ind	29.9	150
ndis-ind	56.5	282
dis-dep	48.1	240
ndis-dep	65.8	329
ind-br	48.3	242
dep-br	87.8	438

Table 7.13: Instruction Sequence Throughput

As would be expected the greatest performance is achieved by dis-ind, the sequence with instructions evenly distributed among the functional units, few data dependencies and no branches. Performance is reduced when either the distribution is unbalanced, data dependencies are introduced, or branches are taken.

The measure 5 x cycle time can be compared with the cycle times measured in the previous section - the factor of 5 corresponds to the 5 functional units and to the 5 instructions per FIG. This is valid because these sequences contain very few immediate chunks. Only the speed of the best case sequence comes close to being limited by the instruction fetcher. The cycle times of all other sequences are significantly greater than the limits of the fetcher or functional unit throughputs, so they must be limited by unbalanced functional unit use, data dependencies or branch instructions.

ndis-ind is intended to be limited by individual functional unit throughput. Its cycle time of 56.5 GD is close to the typical functional unit cycle times of table 7.7.

dis-dep is intended to be limited by data dependencies between instructions. Its cycle time might be expected to be the sum of the functional unit and result network cycle times (i.e. around 100 GD) yet the measured cycle time is 48.1 GD. This difference can be attributed to the difference between the latency of a block (request in to request out) and the cycle time (request in to next request in); this is clear from pipeline visualisation.

ind-br is intended to be limited by the branch latency and has one taken branch in each five instructions. The cycle time for five instructions closely matches the branch latency measured in table 7.12, indicating that the branch latency is limiting the performance.

Example Programs

The performance of the example programs described in section 7.1.1 has been measured and is shown in table 7.14.

Program	Time per instruction / GD
qsort	61.7
shading	47.2
strtree	66.2
sumlist	62.9
wc	57.5
Arithmetic mean	58.1

Table 7.14: Example Program Throughput

Generally the times measured for these programs are about the same as the times recorded for the instruction sequences with dependences, unbalanced use of the functional units, and branches.

7.3.4 Conclusion

Limits may be imposed on SCALP's performance by a number of different factors depending on the particular code being executed:

- The high branch latency means that frequent taken branches substantially reduce performance. This could be alleviated by means of branch prediction.
- Concurrency among the functional units can be limited by data dependencies between instructions executing in separate functional units.
- Concurrency among the functional units can also be limited by an imbalance in the distribution of instructions between the functional units.
- When other factors do not limit performance the cycle time of the FIG fetcher determines the throughput.
- The instruction issuer and result network do not generally limit performance.

7.4 Functional Unit Parallelism

This section evaluates the parallelism that SCALP is able to achieve between its five functional units. Up to five fold parallelism is possible between the functional units, but how much of this parallelism is actually achieved is dependent on the extent of their utilisation.

Figures 7.5 and 7.6 show the number of functional units active at any time for each of the example instruction sequences and programs.

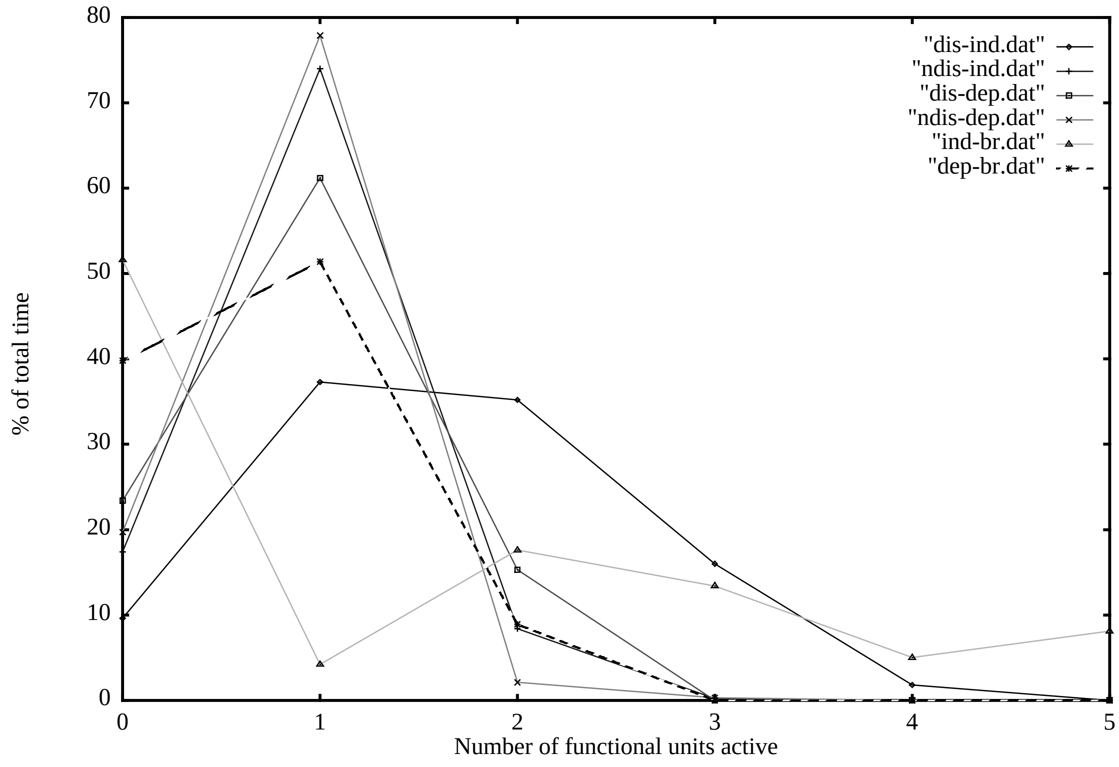


Figure 7.5: Instruction Sequence Functional Unit Parallelism

These figures can be interpreted as follows:

- When no functional units are active the processor is likely to be experiencing a branch latency. Observe that the instruction sequences with no branches spend relatively little time (less than 25%) with no functional units active; the sequences with branches spend 40-50% of their time thus.

- When exactly one functional unit is active the processor is likely to be limited by either data dependencies between instructions or by severe resource contention over one functional unit. The instruction sequences with many data dependencies spend around 50 % to 80 % of the time with one functional unit active; the instruction

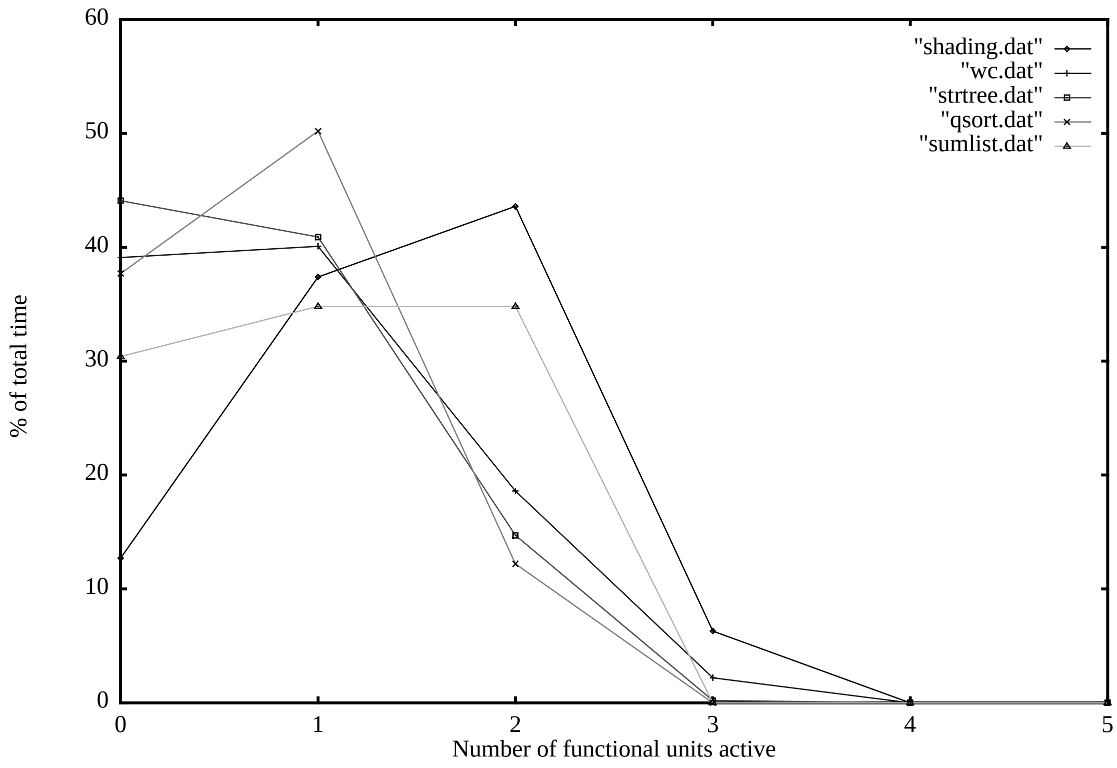


Figure 7.6: Example Program Functional Unit Parallelism

sequences with severe functional unit contention spend around 75 % to 80 % of the time with one functional unit active. The sequences that have neither contention nor dependencies spend the least time with only one functional unit active.

- When more than one functional unit is active the processor is operating free from severe data dependencies or resource contention. Most of the sequences spend less than 15 % of the time with more than one functional unit active. The sequences which are free from resource contention and data dependencies spend around 40 % - 50 % of the time with more than one functional units active.

With these points in mind it can be seen that the example programs spend between around 15 % and 45 % of their time experiencing branch latencies, between around 30 % to 50 % of the time experiencing data dependencies or functional unit contention, and between around 10 % to 50 % of the time free from these impediments.

The programs themselves impose a limit on the amount of functional unit parallelism that is possible. Figure 7.7 shows the amount of time that each functional unit must be active for each example program. From this data it is possible to calculate the maximum possible functional unit parallelism for the programs. This is presented in table 7.15.

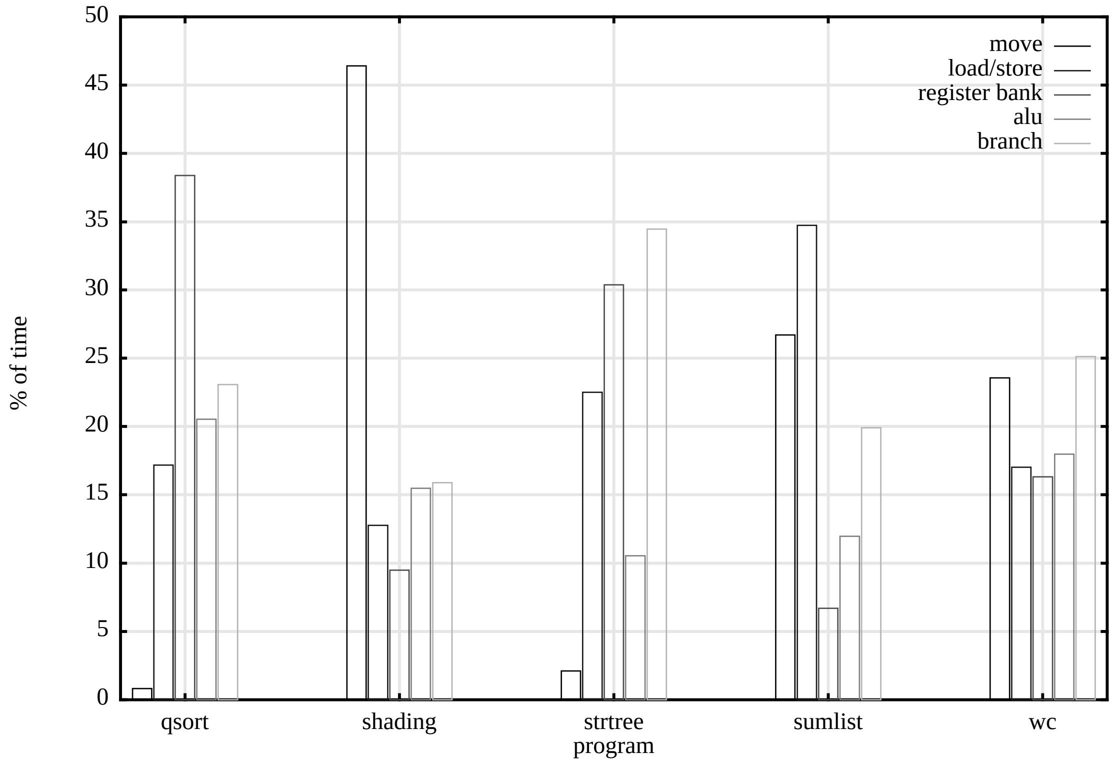


Figure 7.7: Functional Unit Use for Example Programs

Program	Maximum Parallelism
qsort	2.60
shading	2.15
strtree	2.90
sumlist	2.88
wc	3.98

Table 7.15: Maximum Potential Functional Unit Parallelism for Example Programs

Table 7.16 summarises the average number of functional units actually active for the programs.

The raw values in this table are misleading if compared directly with measures of functional unit parallelism for a synchronous processor. When a synchronous processor's functional unit parallelism is measured it is assumed that the functional units are active for all of the clock cycle; in contrast the asynchronous functional unit is considered inactive as soon as its operation is complete. Table 7.16 also provides a measure which allows for this factor.

Benchmark	Functional Unit Parallelism	Equivalent Synchronous Functional Unit Parallelism
dis-ind	1.63	2.34
ndis-ind	0.91	1.24
dis-dep	0.92	1.45
ndis-dep	0.83	1.06
ind-br	1.40	1.45
dep-br	0.69	0.80
shading	1.44	1.48
wc	0.84	1.22
strtree	0.71	1.06
qsort	0.75	1.13
sumlist	1.04	1.11

Table 7.16: Average Functional Unit Parallelism

In comparison [JOHN91] (see appendix A) proposes a number of conventional synchronous superscalar architectures for which the equivalent parallelism measure falls between 1 and 2, typically around one third higher than SCALP.

7.5 Comparative Performance

This section compares the performance of SCALP with the ARM and ARMLESS processors described in section 7.1.1.

Table 7.15 gives the SCALP execution time in gate delays for each of the benchmark programs. Alongside are measurements for ARM and ARMLESS in numbers of cycles.

All ARMLESS instructions take one cycle except taken branches which have a one cycle penalty. For ARM the branch penalty is two cycles; in addition load and store instructions take extra cycles because ARM has a single memory interface.

In order to compare the ARM and ARMLESS results with the SCALP results the cycle times of ARM and ARMLESS have been estimated in terms of gate delays. The values taken are 33 gate delays for ARM and 25 gate delays for ARMLESS. These are approximations based on the clock speeds of ARM and of ARMLESS-like processors and estimates of typical gate delays.

	SCALP	ARM		ARMLESS	
	GD	Cycles	GD	Cycles	GD
Shading	31062	448	14784	384	9600
Word Count	84683	1239	40887	1221	30525
String Tree Search	15822	222	7326	156	3900
Quicksort	459655	2627	86691	1797	44925
Sumlist	10944	217	7161	136	3400
Geometric Mean	46150		19401		11179

Table 7.17: Example Program Performance

These results show that on average SCALP operates 2.4 times more slowly than ARM and 4.1 times more slowly than ARMLESS.

This result may be attributed to the following factors:

- The poor code density reported in section 7.2 means that SCALP must execute more instructions than ARM and ARMLESS.

- The cycle times of the individual functional units and other blocks are slower than the cycle times of the synchronous processors. This is considered in section 7.6.

7.6 Evaluating the performance of Asynchronous Logic

The poor performance results reported in the previous section can be attributed in part to the slowness of the asynchronous control circuits used in parts of the SCALP implementation.

Consider for example the branch unit control circuit described in section 6.6 and shown in figure 6.20. This circuit is typical of the control circuits used in SCALP's functional units. Two points can be observed about this circuit:

- The circuit's structure is very much like a flow chart describing the required functionality.

- The circuit is very sequential in nature.

In contrast consider how a synchronous implementation of this block would be carried out. Initially a specification could be drawn up describing how each of the control inputs would be considered in turn to decide on the correct action for this cycle. This specification could then be optimised, possibly to as little as three levels of logic, and used as the basis for a finite state machine.

The difference between the synchronous and asynchronous implementations is that there is no optimisation of the asynchronous circuit, resulting in sequential and hence slow behaviour. This is a direct consequence of the macromodule design approach.

An important question at this point is whether optimisation techniques for this type of asynchronous circuit exist, and if so whether they perform as well as well known synchronous (and combinational) optimisation techniques. The answer is that they probably do exist but they are more complex than the synchronous techniques and are less mature. This question is considered further in section 8.3.2.

7.7 Power

The combination of a number of factors was intended to increase SCALP’s power efficiency. As previous sections have shown in several cases the expected benefits have not been achieved. The following subsections considers the effects of these factors on the processor’s overall power efficiency.

7.7.1 High code density

Section 7.2 evaluated SCALP’s code density in comparison with the ARM and ARMLESS processors and found that it does not achieve any increase. Consequently there will be no increase in power efficiency resulting from decreased processor to memory bandwidth and related activity.

7.7.2 Parallelism

Sections 7.3 and 7.4 considered the parallelism due to pipelining and to the multiple functional units respectively. Section 7.4 concluded that for the example programs the level of parallelism was rarely greater than one. This low level of parallelism was attributed to the effect of

- the branch latency;
- the presence of data dependencies between instructions;
- contention for functional units.

It was also observed that conventional superscalar processors are able to obtain a greater degree of parallelism. Consequently SCALP’s power efficiency cannot be better than these conventional architectures in this respect.

One important area in which the SCALP architecture could be improved is the provision of branch prediction which would increase the level of pipeline parallelism. The is considered in section 8.3.1.

7.7.3 Asynchronous Implementation

The use of asynchronous logic has had two effects on SCALP’s power efficiency. Firstly in an asynchronous circuit inactive blocks consume no power; in a conventional synchronous circuit the clock is applied to all blocks at all times. This effect leads to a substantial power saving compared to a synchronous implementation of the SCALP architecure as shown in table 7.18. On the other hand this statistic can be seen as simply reflecting the low utilisation of the available parallelism within the processor.

The second effect of the use of asynchronous logic is its influence on the overall speed of the processor. As was observed in section 7.6 the sequential nature of the asynchronous control circuits makes them slow in comparison with synchronous circuits. The pipeline stage cycle times shown in table 7.7 are generally greater than the assumed cycles times for ARM and ARMLESS used in section 7.5. To obtain equivalent performance from the

Program	Average % activity of functional units	Power Saving
qsort	14.9	6.7
shading	28.7	3.5
strtree	14.2	7.0
sumlist	20.9	4.8
wc	16.8	6.0

Table 7.18: Power Saving due to Inactive Functional Units

synchronous and asynchronous circuits the supply voltages would have to be adjusted, substantially favouring the synchronous circuit.

7.7.4 Variable width datapath operations

SCALP’s datapath will consume less power when it operates on byte rather than word quantities. The occurance of byte operations in the example programs is not in any sense representative of any real programs and data on the occurance of these operations in real programs is not available, making estimation of the power saving attributable to this factor difficult.

7.7.5 Summary

In conclusion SCALP’s overall power efficiency is poor in comparison with conventional processors.

7.8 Evaluating the SCALP Programming Model

Section 7.1.1 described five example programs that have been written for SCALP in order to evaluate its performance. Writing these programs identified problems with the processor’s execution model that result in longer code sequences than were expected and hence poor code density - evidenced by the results in section 7.2. This section discusses these problems.

In writing the example programs it became clear that the explicit forwarding mechanism was useful in fewer cases than had been expected. For very many instructions it was necessary to send the result to the register bank, and very many operands were read from registers. Only in relatively few cases could results be sent directly to the functional unit of their next use. Consider the following code which sums the values in an array and saves the sum in another variable in memory:

```
t = 0
for i in 1 to 1000 loop
  t = t + x[i]
end
a = t
```

Ignoring the loop control and array indexing and concentrating on just the summation, this may be compiled to the following code on a conventional processor:

```
     MOV R2,#0       ; R2 stores variable t
     ....
L1:  LDR R1,[...]    ; load x[i] into R1
     ADD R2,R2,R1    ; t = t + x[i]
     ....
     BRNE L1         ; do next loop iteration if terminal
                     ; count not reached
     STR R2,[...]    ; a = t
```

Consider what happens to the result of the addition in this code. For 999 of the 1000 iterations of the loop the result is written to R2 by the add and read from it in the next iteration by the following add. However in the 1000th iteration it is written to R2 by the add and read by the store corresponding to the assignment to variable a.

If the result of that add was always used by the same destination functional unit (i.e. the ALU) then the add could be encoded as follows:

```
add -> alu_a
```

but in this case there are two possible destinations. The only general solution is to use the register bank:

```
add -> regd
write 2
```

and to read from the register bank when the destination is known. In this case there are ways of avoiding the problem, such as the following:

```
for i in 1 to 1000 loop
  t = t + x[i]
end
a = t + 0

for i in 1 to 999 loop
  t = t + x[i]
end
t = t + x[1000]
a = t
```

In the first of these examples an additional “dummy” addition is carried out so that the result of the add within the loop is always sent to the same destination, the ALU. In the second example the last iteration of the loop is unrolled so that the result of the last addition can be sent to a different destination. However these two “tricks” cannot be applied in other more general cases, and in the example programs it was often found necessary to write results to temporary registers.

Of all the example programs the most inefficient in terms of explicit forwarding was the quicksort program. Because of the nature of the program virtually all branches are highly data-dependent. Its typical behaviour is to load a value from memory and compare it with some other value. Whether the value will be used again and if so where it will be used is not known until the result of the comparison has been computed.

An important observation about these examples is that they are consistent with the data in table 5.1 but they indicate that that data is slightly misleading. It is true to say that in a dynamic instruction trace 64 % of instruction results are used exactly once, but it is not true to say that for 64 % of static instructions the result is always used by exactly one subsequent instruction. As far as code generation for SCALP is concerned it is this static property that is important.

Section 8.2 returns to this problem to consider ways in which the architecture can be improved.

Chapter 8: Conclusions

The design and implementation of SCALP has been very different from the design and implementation of other processors. SCALP's objective has not been purely performance as is the case with most other designs; rather it has been largely motivated by power efficiency. SCALP's architecture is novel: it replaces one of the fundamental entities - the register bank - with an alternative model for result-operand communication. Finally the implementation is unusual, using asynchronous logic.

With all of these areas of change it is not surprising to find that SCALP does not perform perfectly throughout. Despite this, much can be learnt from this implementation about how one should, and how one should not, design a microprocessor for power efficiency. This final chapter summarises the features of the SCALP architecture and implementation and considers ways in which it could be improved upon. These and other possible future areas of work are described, including ways in which SCALP's good points could be incorporated into more conventional processors.

8.1 Summary

Motivation

Previous low power processors have applied mainly low level power efficiency techniques as they need to maintain code compatibility with previous implementations of the architecture. In contrast SCALP concentrates on higher level techniques, specifically code density, parallelism and asynchronous implementation.

High code density is beneficial to power efficiency because the energy consumed in the memory system is in proportion to the amount of code that must be fetched. Areas where code density increases can be made are identified as the encoding of register specifiers, which can take up half of the bits in a conventional instruction set, and the use of variable rather than fixed length instructions.

By increasing the amount of parallelism in a circuit and reducing the supply voltage it is possible to operate with the same throughput but with reduced power consumption. Many conventional processors employ parallelism in the form of pipelining and superscalar execution in order to increase performance. As the potential maximum parallelism in the processor increases it becomes increasingly difficult to exploit; techniques such as forwarding, branch prediction, and out of order issue must be used. These all lead to increases in the complexity of the control logic which will potentially increase power consumption and reduce performance. It seems desirable to move some of the responsibility to the compiler in order to make simpler control logic possible.

Implementation using asynchronous logic has advantages in terms of performance and power efficiency compared with synchronous implementation, especially in systems with a variable workload where dynamic supply voltage adjustment is possible. Asynchronous logic lends itself well to the implementation of pipelines and parallel functional units until non-local communication such as forwarding is considered. For reasonable performance without locally synchronous behaviour a technique called conditional forwarding is required. Conditional forwarding is only possible if the instruction set indicates how the result of an instruction will be used in the instruction producing the result itself. This is referred to as explicit forwarding.

Architecture

The SCALP architecture incorporates many of the features suggested by the preceding paragraphs. Most importantly SCALP does not have a global register bank; rather instructions specify destinations for their results in terms of the functional unit input queue to which the result should be sent. Source operands are implicitly taken from these queues. In this way SCALP requires only three bits per instruction to indicate the flow of data through the program, compared with up to fifteen bits for a conventional processor.

One problem with the explicit forwarding mechanism is that when the result of an instruction is used by an instruction beyond a branch the destination cannot be statically encoded into the instruction. In this case the result must be sent to the register unit and written into a register, and subsequently read from the register when its use is known. A second problem is that SCALP provides a fairly inefficient mechanism for sending results to more than one destination. These effects mean that SCALP requires more instructions than a conventional processor. The result is an overall code density that is no better than a conventional processor such as the ARM.

SCALP has five functional units (ALU, Register, Load/Store, Move, Branch) and uses variable length instructions. To simplify the instruction issuer the instruction length and functional unit information is made as simple as possible to decode. In particular the instructions have only two possible lengths and for the longer instructions all control information is contained within the first part. Instructions are decoded and issued in groups and a control field for each group indicates which parts of the group contain short and which long instructions.

Implementation

SCALP is the first known example of a superscalar processor to be implemented using asynchronous logic. The implementation to a gate level VHDL model has been sucessful and is supported by a set of software tools. The design has been carried out almost entirely by hand as the techniques used are relatively new and any automatic tools are still experimental.

The implementation uses a four phase asynchronous signalling protocol with bundled data. The datapath is generally similar to synchronous datapaths but adders make use of completion detection. The control logic uses a macromodular style with largely speed independent interconnections of modules. The modules use a bounded delay timing model internally.

Despite the architectural efforts to reduce the complexity of the instruction issue problem the issuer is a large part of the processor taking up 39 % of the gate count. Despite its large size neither it nor the result forwarding network limit performance. Performance is generally limited by three factors: branch latency, dependencies between instructions and contention for functional units. The branch latency is around 8 times as long as a typical instruction execution period without branches, so 8 potential instruction executions are lost for each branch.

The asynchronous logic style used is relatively slow compared with synchronous implementations; typical cycle times for functional units are around 50 gate delays. This seems to be a consequence of the macromodular design style which leads to very sequential “flowchart-like” circuits.

8.2 Improving the SCALP Architecture

Evaluation of the SCALP architecture has shown it to have some weaknesses, leading to worse code density than had been expected. However it is certainly not clear that the fundamental idea of replacing register-based communication with explicit forwarding is hopeless. This section considers how the weaknesses in the architecture can be put right.

Section 7.8 revealed that SCALP’s explicit forwarding mechanism is flawed in the way that it deals with results being passed to destination instructions beyond branches.

Furthermore the mechanism also does not deal very well with results that are used more than once as explicit duplicate operations are required.

The solution to these problems must be to separate in the instruction stream the routing of results from the instructions that produce those results. In this way the results can be routed after the results of any branch instructions are known. Furthermore this approach lends itself to multiple use of results as more than one piece of routing information can be associated with each instruction. One encoding scheme is shown in figure 8.1.

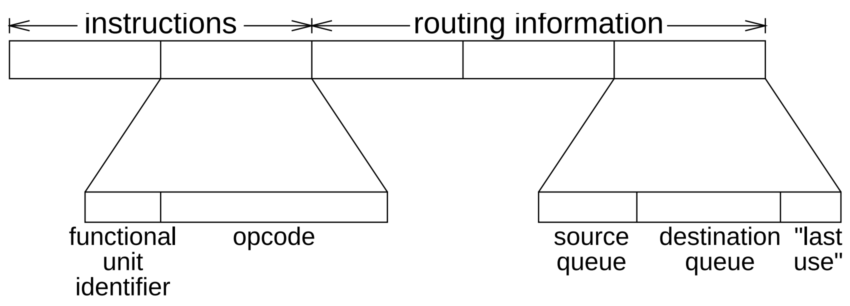


Figure 8.1: Instruction Encoding with Separate Instructions and Result Routing

In the scheme shown in figure 8.1 the instruction stream is divided into instructions and routing information. The format of the instructions is similar to the existing SCALP instructions except that there are no destination bits. Each item of routing information indicates a source queue (i.e. a functional unit output) and a destination queue (i.e. a functional unit input). To allow for multiple uses of results it incorporates an additional bit. This may mean either “this is the last use of this result” or “this is the first use of a new result from this queue”.

This scheme has similarities to the Transport Triggered Architecture described in section 5.7.3 which is encouraging as it is known that this architectures makes a good compiler target.

To evaluate this encoding the inner loops of the example programs described in section 7.1.1 have been rewritten in the new format. The static code sizes for these inner loops are given in table 8.1. It can be seen that the new encoding increases code density by around 20 %.

Program	SCALP			SCALP with separate result routing			
	Instrs	Immeds	Bits	Instrs	Immeds	Moves	Bits
shading	10	0	130	7	0	8	118
wc	13	2	195	10	2	9	180
strtree	16	0	208	10	0	10	160
qsort	31	0	403	19	0	16	286
sumlist	8	0	104	5	0	6	86
Geometric mean			186				153

Table 8.1: Example Program Code Sizes for SCALP with Separate Result Routing

Disappointingly even with this modification SCALP's code density remains significantly lower than can be achieved with some variants of conventional instruction sets such as ARM Thumb (section 2.1.3).

8.3 Improving the SCALP Implementation

Some areas in which the SCALP implementation performs poorly and could be improved have already been identified; specifically the pipeline throughput would be substantially increased if branch prediction were used and the speed of the asynchronous control circuits could be increased by employing more sophisticated implementation techniques.

8.3.1 Branch Prediction

Section 7.3.2 reports that the performance of the processor is significantly affected by the lack of branch prediction. A conventional branch prediction scheme could be adapted for use with SCALP's asynchronous implementation, and indeed one scheme has already been applied to an asynchronous processor: AMULET2 makes use of an associative memory to find new PC values.

For SCALP a technique used by conventional superscalar processors and described in [JOHN91] might be more appropriate. This scheme stores branch prediction information in the instruction cache. Each cache line would store one FIG plus additional information to indicate which cache line contains the next FIG. By giving the cache line number rather than an address the need to perform an associative lookup in the cache is removed, making the cache potentially faster and more power efficient. The cache lines must also store information indicating which instructions within the FIG will not be executed because they lie after a taken branch or before the branch target. This scheme allows for only one prediction per FIG.

The implementation of prediction checking and incorrect prediction recovery would be influenced by the asynchronous nature of the design. Incorrect prediction recovery is similar to exception recovery and as was mentioned in section 5.6 SCALP's unusual transient state makes this process different from conventional processors. There are various ways in which SCALP could implement prediction checking and recovery:

(i) Wait for confirmation at the issuer

In this scheme the instruction issuer does not issue instructions until the branch unit has confirmed that all preceding instructions have been correctly predicted. This is very similar to the current scheme where other functional units' tokens are not allowed to overtake the branch unit's token.

(ii) Wait for confirmation at the functional units

In this scheme the instruction issuer issues instructions from beyond branches but the functional units do not start to execute them until the branch prediction has been confirmed. When the prediction has failed the functional units are responsible for deleting the wrong instructions from their instruction queues. This scheme takes less time between the branch outcome being confirmed and other functional units starting execution.

(iii) Speculative execution in the functional units

In this scheme functional units execute instructions before the branch outcome is known and subsequently the result network or operand queues discard results that come from incorrectly executed instructions. This scheme permits the highest performance but in order to recover from incorrect predictions it is necessary for functional units to keep copies of their old operands until the instructions have been confirmed.

[JOHN91] claims that the speedup attributable to branch prediction is around 30 %. This agrees with estimates made on the basis of the results in section 7.4; if the periods during which no functional unit is active were eliminated the speed of the SCALP example programs would be increased by 34 %.

8.3.2 Faster Asynchronous Logic

As was noted in section 7.6 the speed of SCALP's asynchronous functional units is poor in comparison with the speed of synchronous processors. This can be attributed at least in part to the macromodular style of the asynchronous control logic in these blocks.

There are two techniques that can be applied to help improve this performance. One option is to move from the current style of largely speed independent interconnections of macromodules to a less constrained more hand-crafted style. The AMULET1 and AMULET2 designs for example use this approach and obtain better performance than SCALP. The disadvantage of this technique is that the design takes more effort, is more susceptible to errors and requires more careful verification.

The other possibility is to make greater use of automatic optimised synthesis. Tools exist that are able to synthesise asynchronous circuits from formal specifications such as state graphs or signal transition graphs. Examples include Forcage [KISH94], Assassin [YKMA95], Meat [DAVIA93], and ATACS [MYER95]. At the present time these tools and our understanding of their behaviour are not suficiently developed to make them of significant use. In future implementations provided by these tools have the potential to avoid the highly sequential nature of the macromodular circuits used by SCALP.

8.3.3 Variable Length Instructions

Although the instruction issuer does not in general limit the performance of the processor, if the functional units could be accelerated the issuer could become a bottleneck. The speed and complexity of the instruction issuer are made worse by two aspects of its functionality: variable length instructions and branches. In light of this it is worth considering removing some features to simplify and speed up the design.

Branch instructions are indispensable, and indeed the need for branch prediction makes this part of the instruction issuer more complex. While it is attractive from this point of view to consider eliminating variable length instructions it is important to note that they contribute greatly to increasing code density.

One possibility is to simplify the way in which variable length instructions are dealt with. The idea is to make all instructions the same length, but to have for example “ALU instruction instructions” and “ALU immediate instructions”. From the point of view of the instruction issuer these instructions are for different functional units and are dealt with as such, making a much simpler and hence faster (though larger) instruction issuer possible. At the functional unit inputs the immediate fields are matched with the instructions in the same way that operands from the operand queues are matched.

8.3.4 Functional Unit Pipelining

Although the processor as a whole is pipelined the individual functional units are not. In most cases there is no point in adding internal pipelining as the units are internally very simple; the register bank for example would be hard to pipeline. One exception is the load/ store unit. Adding internal pipelining to this unit so that address calculation is carried out first and then the main memory access is carried out would be beneficial to performance and would not add greatly to complexity.

8.4 The Future of SCALP

Section 8.2 shows SCALP's explicit forwarding mechanism can be improved upon; yet the changes described give only a relatively small improvement that still fails to make SCALP competitive with conventional instruction sets in terms of code density. This suggests that any future design may do better by moving towards a more conventional architecture.

On the other hand the Transport Triggered Architecture indicates that a similar system can operate efficiently. The key to understanding this area must be the development of a compiler and the analysis of substantial benchmark programs. A SCALP compiler and other software tools must be a priority.

In terms of implementation it would be very worthwhile to investigate more highly optimised forms of asynchronous circuit generation.

The power efficiency benefit of SCALP's asynchronous implementation has not been proved. This is because the power efficiency benefits of asynchronous implementation are largely dependent on the application; specifically asynchronous logic saves power when the workload is variable and more so if the supply voltage can be adjusted. In a fixed benchmark situation such as measuring the power consumed in executing a loop of a benchmark program asynchronous logic should not be expected to outstrip a conventional synchronous design. To demonstrate and measure the benefit it is necessary not only to build a microprocessor but also to build the hardware and software of a real system executing an application which makes variable demands on the processor.

SCALP attempts to increase power efficiency by means of parallelism. It does so while executing a single instruction stream in similar way to conventional superscalar processors. The parallelism possible with this approach is known to be limited, and it is probable that for much greater parallelism and hence much greater power efficiency a different form of parallelism would be preferable. These other forms, such as message passing or shared memory multiprocessors, have the disadvantage that algorithms must be recoded to take advantage of them. This may be the cost of increased power efficiency.

8.5 SCALP and Conventional Architectures

It can be argued that the greatest challenge facing computer architects is not how to build good new architectures but rather how to apply new implementation techniques to existing instruction sets, maintaining code compatibility. This is most clearly seen in the world of 8086 compatible processors where the huge effort put in to new implementations makes up for the technical inferiority of the underlying instruction set architecture. It is interesting to consider whether the SCALP ideas could in some way be applied to more conventional instruction sets.

A hybrid processor could be constructed as follows: the programmer-visible instruction set is a conventional register-based one. This code is read into the instruction cache as normal. During its first execution the processor detects dependencies between instructions that need forwarding and stores this explicit forwarding information back into the cache with the code. On subsequent executions this information activates explicit forwarding mechanisms to increase the performance of the code.

For an asynchronous processor this approach would provide the benefits of conditional synchronisation and bring out the potential performance of the asynchronous pipelines.

On the other hand because of the conventional instruction set any code density benefit would be lost.

8.6 Conclusions

This work ends on a mixed note. There are several areas of related work that suggest that the ideas underlying SCALP are sound ones:

- The Philips work [BERK95] [KESS95] indicates that asynchronous logic can have substantial power efficiency benefits.
- The Berkeley InfoPad work [CHAN94] indicates that parallelism can have substantial power efficiency benefits.
- The D16 design [BUND94] indicates that high code density can have power efficiency benefits.
- The transport triggered architectures work [CORP93] indicates that a programming model with no global register bank and the concept of explicit forwarding can be successful.

On the other hand the SCALP implementation has not performed as well as had been hoped. It can be seen that SCALP’s explicit forwarding mechanism is flawed but improvements proposed in this chapter do not make a dramatic improvement. It can also be seen that it is easier to implement a highly parallel system for a special purpose computation, as in the InfoPad, than for a general purpose processor.

In conclusion asynchronous parallel processors offer the possibility of increased power efficiency, but they cannot achieve this using conventional instruction sets and architectures. One alternative architecture has been proposed and evaluated. Although not hugely successful the SCALP architecture points to possibilities for future asynchronous superscalar low power processors.

Appendix A: Benchmarks

Various statistics presented in the thesis are based on analysis of benchmark programs undertaken by the author or reported by others. The purpose of this section is to summarise the nature of these benchmarks and to justify their relavence to this work.

The sources of data are sumarised in table A.1.

Reference	Processor	Applications
[ENDE93] (Appendix C)	SPARC (Load/Store architecture with register windows)	ccreg (C compiler) compress (data compression) dhry10 (synthetic benchmark) espresso (logic minimisation) fgrep (text search) gcc (C compiler) (same as [HENN90]) rleflip (image manipulation)
[HENN90] (figure 2.17)	DLX (MIPS-like Load/Store architec-ture)	gcc (C compiler) Tex (text formatter) Spice (analogue circuit simulator)

Table A.1: Benchmark Programs (Part 1 of 2)

Reference	Processor	Applications
[JOHN91] (table 3-2)	MIPS R2000 (Load/ Store architecture)	5diff (text file comparison) awk (pattern scanning and processing) ccom (optimizing C compiler) compress (file compressing using Lempel-Ziv encoding) doduc (Monte-Carlo simulation, double precision floating point) espresso (logic minimization) gnuchess (computer chess program) grep (reports occurences of a string in one or more text files) irsim (delay simulator for VLIS layouts) latex (document preparation system) linpack (linear equation solver, double precision floating point) nroff (text formatter for a typewritter-like device) simple (hyrodynamics code) spice2g6 (circuit simulator) troff (text formatter for typesetting device) wolf (standard-cell placement using simulated annealing) whetstone (standard floating point benchmark, double precision floating point) yacc (compiles a context-free grammar into LR(1) parser tables)

Table A.1: Benchmark Programs (Part 2 of 2)

In all cases the programs have been compiled with optimising compilers.

The applications used are intended to be typical of those commonly used in workstation environments. Some applications of low power processors have workstation-like characteristics (e.g. portable computers) and in these cases these benchmarks are relevant. In other applications with a different type of workload (e.g. signal processing) these results are less relevant and should be used with more caution.

References

[ALID94] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, M. Papaefthymiou, “Precomputation-Based Sequential Logic Optimization for Low Power”, Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, San Jose, California, 1994, pages 74-81.

[ARM95] Advanced RISC Machines Ltd, “An Introduction to Thumb”, 1995, Cambridge, U.K.

[BENI95] L. Benini, G. De Micheli, “Transformation and synthesis of FSMs for low-power gated-clock implementation”, Proceedings of the 1995 International Symposium on Low Power Design, Dana Point, California, pages 21-26.

[BERK91] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, F. Schalij, “The VLSI programming language Tangram and its translation into handshake circuits”, Proceedings of the European Conference on Design Automation, Amsterdam, 1991, pages 384-389.

[BERK95] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Ronken, F. Schalij, R. van de Weil, “A Single-Rail Re-implementation of a DCC Error Detector Using a Generic Standard-Cell Library”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pages 72-79.

[BRUN93] E. Brunvand, “The NSR Processor”, Proceeding of the 26th Annual Hawaii International Conference on System Sciences, pages 428-435, Maui, Hawaii, 1993.

[BUND94] J. Bunda, W. C. Athas, D. Fussell, “Evaluating Power Implications of CMOS Microprocessor Design Decisions”, Proceedings of the 1994 International Workshop on Low Power Design, Napa, California, pages 147-152.

[CHAN94] A. P. Chandrakasan, A. Burnstein, R. W. Brodersen, “A Low-Power Chipset for a Portable Multimedia I/O Terminal”, IEEE Journal of Solid State Circuits, Volume 29, Number 12, December 1994, pages 1415-1428.

[CHIE95] C.-H. Chien, M. A. Franklin, T. Pan, P. Prabhu, “ARAS: Asynchronous RISC Architectue Simulator”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pages 210-219.

[CHO92] K.-R. Cho, K. Okura, K. Asada, “Design of a 32-bit Fully Asynchronous Microprocessor (FAM)”, Proceedings of the 35th Midwest Symposium on Circuits and Systems, 1992.

[CORP93] H. Corporaal, “Evaluating Transport Triggered Architectures for scalar applications”, Microprocessing and Microprogramming, No 38, pages 45-52, 1993. http://einstein.et.tudelft.nl/~heco/documents/documents.html

[CORR95] A. Correale, “Overview of the Power Minimization Techniques Employed in the IBM PowerPC 4xx Embedded Controllers”, Proceedings of the 1995 International Symposium on Low Power Design, Dana Point, California, pages 75-80.

[DAVIA93] A. Davis, B. Coates, K. Stevens, “Automatic Synthesis of Fast Compact Asynchronous Control Circuits”, Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, U.K., 1993.

[DAVII93] I. David, R. Ginosar, M. Yoeli, “Self-Timed Architecture of a Reduced Instruction Set Computer”, Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, U.K., 1993.

[DAY95] P. Day, J. V. Woods, “Investigation into Micropipeline Latch Design Styles”, IEEE Transactions on VLSI, Volume 3, Number 2, June 1995, pages 264-272.

[DEAN92] M. E. Dean, “Strip: A Self-Timed RISC Processor”, Technical Report CSL-TR-92-543, Stanford University, 1992.

[ELST95] C. J. Elston, D. B. Christianson, P. A. Findlay, G. B. Steven, “Hades - Towards the Design of an Asynchronous Superscalar Processor”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pages 200-209.

[ENDE93] P. B. Endecott, “Processor Architectures for Power Efficiency and Asynchronous Implementation”, M.Sc. thesis, University of Manchester, U.K., 1993. `http://www.cs.man.ac.uk/amulet/publications/thesis/endecott93_msc.html`

[FARN95] C. Farnsworth, “The AMULET Low Power Cell Library”, Technical Report, University of Manchester, In Preparation, 1995?.

[FURB89] S. B. Furber, “VLSI RISC Architecture and Organization”, Marcel Dekker, New York, 1989, ISBN 0-8247-8151-1.

[GARS93] J. D. Garside, “A CMOS VLSI Implementation of an Asynchronous ALU”, Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies, Manchester, U.K., 1993.

[GLEB95] A. L. Glebov, D. Blaauw, L. G. Jones, “Transistor reordering for low power CMOS gates using an SP-BDD representation”, Proceedings of the 1995 International Symposium on Low Power Design, Dana Point, California, pages 161-166.

[GOOD85] J. R. Goodman, J.-T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, H. C. Young, “PIPE: A VLSI Decoupled Architecture”, Proceedings of The 12th Annual International Symposium on Computer Architecture, Boston, Massachusetts, 1985, pages 20-27.

[GWEN95] L. Gwennap, “P6 Underperforms on 16-Bit Software”, Microprocessor Report, Volume 9, Number 10, July 31 1995, page 1.

[HENN90] J. L. Hennessy, D. A. Patterson, “Computer Architecture A Quantitative Approach”, Morgan Kaufmann, Palo Alto, 1990, ISBN 1-55860-069-8.

[HOOG94] J. Hoogerbrugge, H. Corporaal, “Transport-Triggering vs. Operation-Triggering”, International Conference on Compiler Construction, Edinburgh, April 1994. `http://einstein.et.tudelft.nl/~heco/documents/documents.html`

[JOHN91] M. Johnson, “Superscalar Microprocessor Design”, Prentice Hall, Englewood Cliffs, 1991, ISBN 0-13-875634-1.

[KEAR95] D. Kearney, N. W. Bergmann, “Performance Evaluation of Asynchronous Logic Pipelines with Data Dependent Processing Delays”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pages 4-13.

[KESS95] J. Kessels, “VLSI Programming of a Low-Power Asynchronous Reed-Solomon Decoder for the DCC Player”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pages 44-52.

[KHOO95] K.-Y. Khoo, A. N. Willson, “Charge Recovery on a Databus”, Proceedings of the 1995 International Symposium on Low Power Design, Dana Point, California, pages 185-189.

[KISH94] M. Kishinevsky, A. Kondratyev, A. Taubin, V. Varshavsky, “Concurrent Hardware - The Theory and Practice of Self-Timed Circuits”, Wiley Series in Parallel Computing, 1994.

[MART89] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, P. J. Hazewindus, “The Design of an Asynchronous Microprocessor”, Advance research in VLSI; Proceedings of the Decennial Caltech Conference on VLSI, MIT Press, pages 351-373, 1989; also as technical report Caltech-CS-TR-89-02, Computer Science Depatment, California Institute of Technology, 1989.

[MITC90] D. A. P. Mitchell, J. A. Thompson, G. A. Manson, G. R. Brookes, “Inside the Transputer”, Blackwell Scientific Publications, 1990, ISBN 0-632-01689-2.

[MORT95] S. V. Morton, S. S. Appleton, M. J. Liebelt, “ECSTAC: A Fast Asynchronous Microprocessor”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pages 180-189.

[MYER95] C. J. Myers, P. A. Beerel, T. H.-Y. Meng, “Technology Mapping of Timed Circuits”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995, pages 138-147.

[NANY94] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, A. Takamura, “TITAC: Design of a Quasi-Delay-Insensitive Microprocessor”, IEEE Design and Test of Computers, Volume 11, No 2, 1994, pages 50-63.

[NIEL94] L. S. Nielsen, J. Spars[O/], “Low-power operation using self-timed circuits and adaptive scaling of the supply voltage”, Proceedings of the 1994 International Workshop on Low Power Design, Napa, California, pages 99-104.

[OLSO94] E. Olson, S. M. Kang, “Low-Power State Assignment for Finite State Machines”, Proceedings of the 1994 International Workshop on Low Power Design, Napa, California, pages 63-68.

[PAVE94] N. C. Paver, “The Design and Implementation of an Asynchronous Microprocessor”, Ph.D. Thesis, Department of Computer Science, University of Manchester, U.K., 1994. `http://www.cs.man.ac.uk/amulet/publications/thesis/paver94_phd.html`

[POUN95] D. Pountain, “Transport-Triggered Architectures”, Byte, February 1995, pages 151-152.

[RABA94] J. M. Rabaey, “Design Solutions and Challenges for Low Power Systems”, Tutorial #2, International Conference on Computer Aided Design, 1994.

[RICH92] W. F. Richardson, E. Brunvand, “The NSR Processor Prototype”, Technical Report UUCS-92-029, University of Utah, 1992. ftp://ftp.cs.utah.edu/techreports/1995/UUCS-92-029.ps.Z

[RICH95] W. F. Richardson, E. Brunvand, "Fred: An Architecture for a Self-Timed Decoupled Computer", Technical Report UUCS-95-008, University of Utah, 1995. `ftp://ftp.cs.utah.edu/techreports/1995/UUCS-95-008.ps.Z`

[SEIT80] C. Seitz, "System Timing", Chapter 7 of "Introduction to VLSI Systems" by C. Mead and L. Conway, Addison Wesley, ISBN 0-201-04358-0, 1980.

[SOLO94] P. M. Solomon, D. J. Frank, "The Case for Reversible Computation", Proceedings of the 1994 International Workshop on Low Power Design, Napa, California, pages 93-98.

[SPRO94] R. F. Sproull, I. E. Sutherland, C. E. Molnar, "Counterflow Pipeline Processor Architecture", IEEE Design and Test of Computers, Volume 11, No 3, 1994. Also as Sun Microsystems Laboratories Inc. Technical Report SMLI TR-94-25.

[SUN92] Sun Microsystems Inc., "The SuperSPARC Microprocessor", Technical white paper, 1992, Mountain View, California.

[SUTH89] I. E. Sutherland, "Micropipelines", Communications of the ACM, Volume 32, No. 6, June 1989, pages 720-738.

[SWAI95] O. Swai, "A Micropileine Visualisation Tool", M.Sc. Dissertation, University of Manchester, U.K., to be submitted, 1995.

[TURL95a] J. L. Turley, "Hitachi SH3 Hits 100 MIPS", Microprocessor Report, Volume 9, Number 3, March 6 1995, pages 12-14.

[TURL95b] J. L. Turley, "NEC Plunges into PDA Processor Market", Microprocessor Report, Volume 9, Number 4, March 27 1995, pages 12-13.

[USAM95] K. Usami, M. Horowitz, "Clustered Voltage Scaling Technique for Low-Power Design", Proceedings of the 1995 International Symposium on Low Power Design, Dana Point, California, pages 3-8.

[WEST93] N. H. E. Weste, K. Eshragian, “Principles of CMOS VLSI Design”, Second Edition, Addison-Wesley, 1993, ISBN 0-201-53376-6.

[WULF88] W. A. Wulf, “The WM Computer Architecture”, Computer Architecture News, Volume 16, No. 1, 1988, pages 70-84.

[YKMA95] C. Ykman-Couvreur, B. Lin, “Optimised state assignment for asynchronous circuit synthesis”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, U.K., 1995.