Design of an Asynchronous Processor

Christos Panagiotis Sotiriou

Doctor of Philosophy
University of Edinburgh
2001

Abstract

Designing performance-scalable computer architectures is becoming an increasingly complex problem in small dimension integrated circuits, as the effects of physical laws increasingly constrain circuit design and computer architecture.

This thesis investigates the implementation of asynchronous circuits and asynchronous computer architectures. In the area of asynchronous circuits, it proposes the direct-mapped approach to control circuit design, originally devised by Hollaar, mapped to CMOS technology. In the area of asynchronous computer architecture, it investigates scalable, concurrent computer architectures, with the aim of solving the problems of scaling performance and utilising the increasing device count. The design and implementation of two hardware structures, Shared Register Files and μnet (micronet) architectures is detailed, together with their incorporation into the design of an asynchronous prototype processor, the A1 chip.

The Shared Register File approach provides a scalable and segmented datapath by partitioning the conventional monolithic register file into multiple register files which physically share registers. Communication and synchronisation between the shared register files takes place via the shared registers. This approach can be used to implement a clustered uniprocessor or a single-chip multiprocessor system. The shared register file approach allows for the exploitation of program level concurrency, where different parts of the same program or different programs can run on the different shared register file datapaths. The design and implementation of shared register files is presented.

The μnet approach is a methodology for asynchronous processor design, which allows fine-grain instruction level parallelism to be exploited. It implements a processor architecture as a non-linear pipeline with inputs at every pipeline stage. In this way, a μnet architecture exploits more fine-grain parallelism than a conventional pipelined architecture. The design and implementation of generic, scalable μnet architectures is described and evaluated.

Acknowledgements

First and foremost I would like to thank my supervisor Prof. Roland N. Ibbett for his invaluable help, support and guidance. Without him this work would not have been possible.

I also want to thank my fellow PhD students, with which we became close friends, and everyone that made my stay in Edinburgh a very special and memorable time.

Finally, I would like to thank my family for all their help and support.

Declaration

I declare that this thesis has been composed by myself and all the work contained in this thesis is my own, except where otherwise stated in the text.

Christos Panagiotis Sotiriou

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scalable Architectures

1.1.1 Scalability in Architectures

Scalability requires a compositional architecture, where components can be added or removed. These components should be designed with emphasis on both physical scalability and potentially scalable performance. Since the 1970s, fine-grain, scalable architectures have been identified as the best possible way to utilise the increased device count that the breakthrough in VLSI technology has provided [DL99]. The key aspects of these architectures relevant to performance scalability have been identified as regularity in structure, programmability in terms of the hardware-software interface and, most importantly, concurrency.

With the transition from Very Large Scale Integration (VLSI) to Giga Scale Integration (GSI), another increase in the device count is expected of the order of 1000. The number of available transistors on a single chip is rising exponentially and is expected to be in the order of 1 billion before 2010.

Additionally, as feature sizes scale, the metal wire bandwidth is becoming a potential limiting factor to circuit speed and inter-circuit communication. The propagation of signals in metal wires is now becoming comparable to the switching speed of transistors and the capacitance, resistance or even inductance of metal wires has to be considered in circuit design [CAD]. One way of tackling this problem is to change the metallisation material, currently Aluminium, to reduce RC delay and IR drop. The only three metals with better conductivity than Aluminium are silver, copper and gold. Unfortunately, even using silver, the improvement in resistance cannot exceed a factor of 2. Copper interconnect is now being used by some manufacturers [IBM97][MOT97a], despite its processing difficulties; it requires an extra barrier layer as it diffuses through SiO_2 and an extra passivation layer to prevent corrosion. The net effect is a 20 to 30%

reduction of the effective resistance [CS96].

So, these technology issues must be considered when new architectures are being designed. The important challenges faced by computer architects are to manage locality, to reduce the communication overhead and to provide efficient synchronisation mechanisms.

1.1.2 Scaling Performance

Increases in the performance of a computer system can be achieved in two different ways; the time it takes to perform a certain task can be reduced and/or more tasks can be performed in a unit of time. The former is usually bounded by physical and implementation constraints imposed by the circuit technology. The speed of transistors, the RC delays of metal tracks, the parasitic capacitances between layers and the power supply voltage are some of the factors which influence circuit speed. Such factors depend on the technology type (*e.g.* CMOS or BiCMOS) and the feature size (*e.g.* 0.7μm, 0.35μm). Therefore, as the technology sets an upper bound to speed, concurrency is necessary for scaling performance. All of today's high-performance architectures exploit concurrency in some way to achieve performance.

Compositionality and concurrency are very much related issues. Any system which is able to perform operations in parallel must be compositional in some way. An element in such an architecture which is not compositional may act as a constraint on performance. As the performance requirements increase, so does the size of compositional architectures. Non-compositional, centralised components are potential bottlenecks, especially those whose performance scales badly with the size of the architecture. Concurrency can exist both in time and in space and both between program instructions and data.

1.1.2.1 Instruction Level Concurrency

Since the third generation of computers (1965-74), techniques such as pipelining, multiple functional units and scoreboarding have been used to exploit instruction level concurrency (or instruction level parallelism, ILP) and to scale performance [HP90]. The first general-purpose machines to introduce pipelining were the IBM 7030 [Blo59], known as Stretch, and Atlas [KELS62]. The IBM 7030, for example, overlapped fetch, decode and execute using a 4-stage pipeline. Then, in 1963, the CDC 6600 introduced extensive use of multiple functional units (FUs) along with scoreboarding [Tho64].

The scoreboard is essentially a hardware data structure which analyses instruction dependencies and allows instructions to execute out of order when sufficient resources and no data dependencies exist. Three years later, the IBM 360/91 went a step further by using another data dependency analysis scheme, the Tomasulo Algorithm [Tom67]. This distributed the data dependency analysis logic among the architecture's hardware units and registers and eliminated false dependencies. True dependencies exist between two instructions when the result of one instruction is required by another, also known as Read-After-Write (RAW) hazards because of the order that they impose. True dependencies cannot be removed and have to be respected for correct program execution. Two types of false dependency exist. The first type of false dependency occurs when two instructions share the same destination register and is known as a Write-After-Write (WAW) hazard. The second type of false dependence occurs when the result of one instruction is the operand of another earlier in the instruction stream but the former has not yet read its operands, and is known as a Write-After-Read (WAR) hazard. False dependences can be removed by increasing the register usage and using different registers as destinations or operands for the instructions that cause them.

The next evolutionary step was to make the pipeline structure known to the compiler. That happened in the early 80's with Reduced Instruction Set Computer (RISC) machines, such as the Berkeley RISC [PS82]. The instruction set became simpler and better suited to conform to a pipelined structure. Hardware-software interaction was enhanced by allowing the compiler to select instructions for the pipeline structure. From then on, to exploit even more instruction level concurrency, machines which fetch and issue multiple instructions in a single step were proposed.

Two types of systems were devised, Very Long Instruction Word (VLIW) [Fis83] and superscalar [Joh91][SS95]. In VLIW architectures, multiple instructions are packed into a single fixed-format instruction word by the compiler and then each instruction in that word feeds into an appropriate functional unit of a multiple FU architecture. Superscalar architectures, in which, multiple independent instructions are fetched and issued, operate at the hardware level, placing fewer demands to the compiler. Superscalar architectures check dynamically for FU availability and can support out of order issue and execution.

1.1.2.2 Data Level Concurrency

Array processors followed a different approach to increasing concurrency [Hwa93]. Instead of increasing the complexity of the single processor, array processors replaced it with a number of simpler ones. The argument for array processors is that although each processor in an array processor system is not as powerful as the most powerful single processor, their combination is both much more powerful and cost-effective.

One of the first computers to adopt the idea of array processing was the Illiac IV [Hor82]. It was delivered in 1972 to NASA but difficulties with the project meant that only a quarter of the original design was implemented and this somewhat hindered the investigation of this style of architecture. After that, a number of such architectures emerged, the Burroughs BSP [KS82], the ICL DAP [Red73], the CM-2 [Thi90] and the MasPar MP1 [Mas91].

A typical array processor architecture comprises an array of processing elements and a central control unit. The control unit distributes array instructions and data among the processing elements. The processing elements can be elementary 1-bit processors as in the Illiac IV or the ICL DAP architectures or more complex as in the MasPar MP1. Typically, all processing elements execute the same instruction and masking logic is provided to enable or disable a processing element during the execution of an instruction. Communication between the processing elements is necessary and is implemented by a data-routing network, which is controlled by the executing program.

Vector architectures provide machine instructions that operate on data sets rather than scalar values, hence exploiting spatial parallelism. They use vector FUs which are pipelined and can operate on multiple data elements simultaneously, resulting in high data throughput and performance. The first vector machines were the CDC STAR-100 [HT72] and the TI ASC [Wat72] which were both announced in 1972. These were both memory-memory machines meaning that a vector operation had a high start-up overhead due to the amount of memory fetches. Also, the vector size ranged from several hundred to several thousand elements. The CRAY-1 [Rus78], introduced in 1976, was a vector-register architecture which reduced the start-up overhead of vector operations. The CRAY-1 was the first commercially successful vector machine due to its high vector and scalar performance. The evolution of vector machines continued and as the need for higher performance continued, so did the exploitation of concurrency. Deeper pipelining, exploitation of instruction level concurrency and the use of multiple processors followed. The CRAY X-MP was the first vector architecture to introduce multiprocessor configurations.

1.1.2.3 Program Level Concurrency

Architectures that exploit a different type of concurrency, *i.e.* explicit concurrency between program data and instructions were also devised. This class is referred to as parallel architectures. Such architectures include multiprocessors and multicomputers.

Multiprocessors and multicomputers are architectures with multiple processing elements, which are able to execute multiple program threads simultaneously. The nature of the processing elements can vary from a simple scalar processor to a complex deeply-pipelined vector processor. The difference between multiprocessors and multicomputers is the memory system and the communication medium between processing elements. In a multiprocessor system, processors communicate via shared variables in a common memory. In a multicomputer system, each computer node has a private, local memory and communication takes place between nodes through messages on communication links.

Multiprocessors can be classified according to their memory access model: the uniform memory access (UMA) model, the nonuniform memory access (NUMA) model, and the cache only memory architecture (COMA) model. The difference between these models is the structure of the memory hierarchy.

In the UMA model, all processors have equal access times to a uniformly shared memory. In the NUMA model, the memory is physically distributed across the processors and each processor has a local memory. The memory access time varies with the physical location of a memory word; the local memory is the fastest to access, remote memory access is longer because of the delay of the interconnect. The COMA model is a special case of the NUMA model, where the distributed memories are replaced by cache memories, which form a global address space.

Multicomputers are composed of multiple processors with local memories and a message-passing interconnection network which provides static connections between the nodes. The local memories in multicomputer systems can only be accessed by the processor they are attached to, this is why they are sometimes referred to as no-remote-memory-access (NORMA) machines.

1.1.3 Architectures of today and tomorrow

Contemporary processors like the Intel Pentium family [Sha98] and clones, the Alpha 21264 architecture [Com99] and the PowerPC architecture [Mot97b] are all ILP architectures. Their common characteristics is that they are all pipelined, have multiple FUs, operate on multiple instructions simultaneously and support out-of-order and speculative execution. ILP architectures use a mixture of static (compiler driven) and dynamic (hardware) techniques.

They typically fetch multiple instructions in a single operation, in the original program order. They then remove false instruction dependencies by renaming the logical registers assigned by the compiler to physical ones. After this stage, execution of instructions can occur out-of-order, *i.e.* depending on the availability of operands. Results of instructions can be forwarded to other instructions. The instruction's results are stored in the original program order in a queue called the reorder buffer. Their results are committed to the registers and memory in-order to ensure correct program behaviour. Branch instructions are handled in the fetch stage. Their outcome is typically predicted by some scheme and instruction execution continues speculatively. Depending on whether the prediction was correct or not, the results of speculatively executed instructions are commited.

The differences between these processors are their architectural parameters and their fabrication processes. Examples of such parameters are the size of the data and instruction caches, the number of registers and their size, the maximum number of instructions that can be issued, the size of the branch prediction tables, the number of FUs, etc.

The trend in ILP processor design is to keep increasing the processor resources and investigate new techniques for exploiting parallelism. The number of registers, the number of FUs and the number of instructions that a processor can handle in a given cycle are parameters which keep increasing.

There are limitations to this approach however [PJS97][ONH$^+$96]. The register requirements of ILP architectures are high [FJC95][MSAD92] both in terms of the number of registers and the number of ports because of the high number of instructions that are in flight in the architecture and the existence of multiple FUs. The performance scalability of the centralised, monolithic register file is a problem for such architectures. Forwarding results between instructions directly is a common approach used to bypass the register file and improve performance. However, a typical implementation to allow full connectivity between the inputs and outputs of all FUs also scales badly. This is because it is dominated by RC delays due the use of multiple result busses and tri-state circuits.

The continuous shrinking of transistor sizes, the interconnect problem and the ability to integrate more devices onto the same chip have triggerred a rethink about the architectures of tomorrow. The ability to integrate a few small-scale, embryonic processors on the same chip has given new potential to array processors and multiprocessor systems, giving rise to clustered and single-chip multiprocessor architectures. Both of these architectures are compositional and therefore scalable, whereas ILP processors are not. In general, they aim at a higher degree of parallelism than the ILP processor, but ILP processors have reached the limits of parallelism that they can exploit anyway.

Clustered and single-chip multiprocessor architectures are composed of a number of processing elements which are fed by a single or multiple instruction streams respectively. Each processing element is typically composed of a number of FUs and some form of local storage. The ability of the processing elements to communicate is paramount for the exploitation of parallelism and performance.

In this work, the problem of communication and synchronisation between processing elements in a clustered or single-chip multiprocessor system is considered.

1.1.4 The Rôle of the Register File

Historically, the first machines were accumulator based. Machine instructions always involved the accumulator, a special register for storing one of the operands of an operation, for reasons of hardware simplicity. This approach implied a high memory traffic as no means of temporary storage existed. The CDC 6600 introduced hardware registers because the multiple FUs of the machine could not be fed fast enough with operands at the speed of the main memory.

The register file (RF), a bank of general purpose registers has since become almost ubiquitous as a component in computer architectures. In contemporary architectures, the most common instruction types are load/store, *i.e.* instructions which transfer data between the RF and the lower levels of the memory hierarchy, and register-register, *i.e.* instructions which operate on data stored in registers and store their result back into the RF. The RF is the part of the memory hierarchy closest to the processor; it can be randomly accessed and results stored once can be read multiple times by multiple instructions. Although it is so commonly used, it is one of the most difficult components to scale in an otherwise scalable architecture, both in terms of performance and compositionality.

Assuming fixed length registers (say 32-bits), the speed of an RF depends primarily on two factors, the number of registers it contains and the number of access ports. The dependence is not linear but quadratic [Kum96]. This is because an increase in either of these parameters has an effect on the electrical parameters of the RF datapath, increasing both its capacitance and resistance, whose product yields the time constant which determines circuit speed.

The numbers of ports and registers are determined by the architecture. For maximum performance, if the number of FUs is increased, then the bandwidth between the FUs and the registers must be increased, implying an increase in the input and output ports of the RF. An architecture with n FUs requires 2n read and n write ports. Concurrent, look-ahead architectures, such as superscalar or VLIW, increase register usage even more, both through the use of register renaming to remove false dependencies and by operating on many instructions simultaneously [FJC95][MSAD92]. It is important therefore to find an alternative solution to the centralised or monolithic register file (MRF) approach.

1.1.5 Alternatives to the MRF

A number of scalable architectures have considered the MRF problem. Two approaches have been followed: multiple RFs and partitioned RFs.

The use of multiple RFs is the most common approach, where the MRF is segmented into a number of smaller RFs, *i.e.* with fewer registers and fewer ports. Each of these RFs is then allocated a number of functional units (FUs), yielding an architecture which is composed of a number of nodes or clusters of FUs. The most important characteristic which distinguishes this type of architecture is the communication method established between the nodes.

Capitanio's limited-connectivity VLIW machine [CDN92] (Figure 1.1) is a clustered VLIW architecture which has multiple RFs and uses extra busses and extra RF ports for inter-RF communication (busses at the top of Figure 1.1). Multiplexers are used to provide a fully connected network (a crossbar) between the multiple RFs allowing them to write to all RFs busses.

Fernandes' queue RF approach [Fer98] (Figure 1.2) is again a clustered VLIW architecture where all communication between RFs takes place via a number of queues. These queues establish a bi-directional communication ring between the clusters.

The multicluster architecture, Farkas et al, [FCJV97] (Figure 1.3) is a dynamically scheduled superscalar architecture that uses multiple nodes, each with its own RF, and that allows inter-node communication to take place at the register level by using multiple register transfer busses between the nodes.

The Wisconsin-Madison multiscalar processor [SBV95] (Figure 1.4) is a coarse-grain machine, again with multiple nodes each with its own RF, but each node runs a statically selected task and communication between the nodes takes place with a uni-directional ring mechanism which follows the order of execution.

The other approach is partitioned RFs [JC95]. This approach, instead of

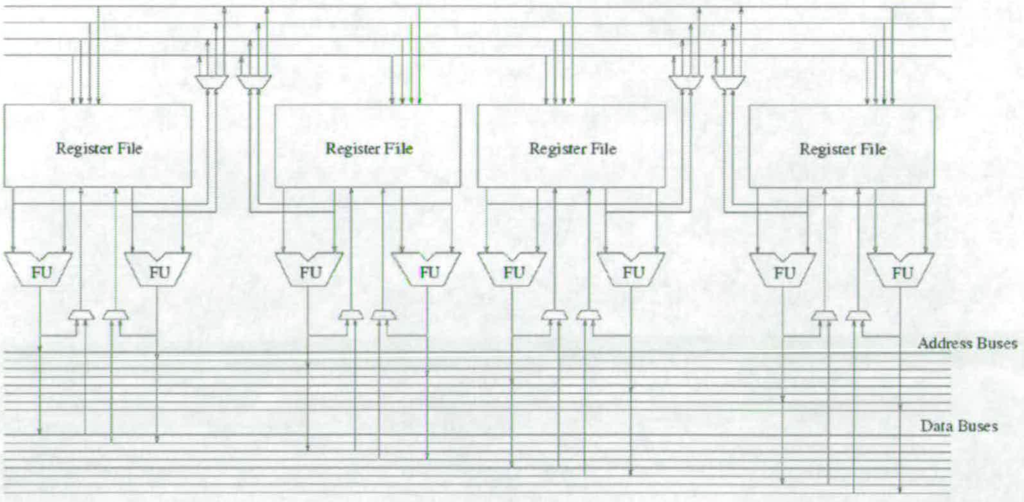


Figure 1.1: Example Limited Connectivity VLIW Architecture

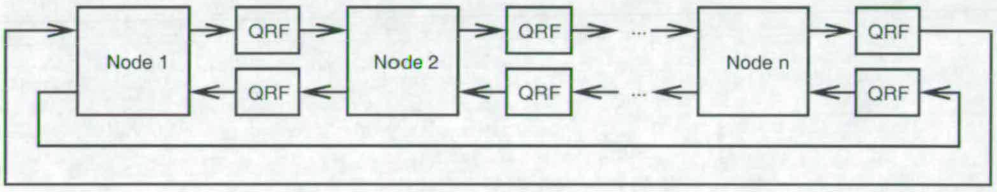


Figure 1.2: Clustered Queue Register File Architecture

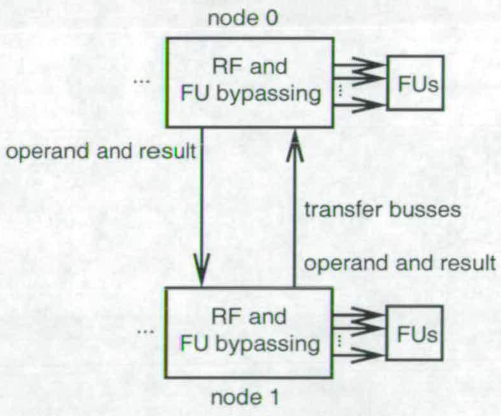


Figure 1.3: Part of a dual-node Multicluster Architecture

distributing the multiple RFs across the architecture, groups them together into a single partitioned RF and views each RF as a partition of the complete one (Figure 1.5). This is achieved by another level of register decoding in order to select the partition.

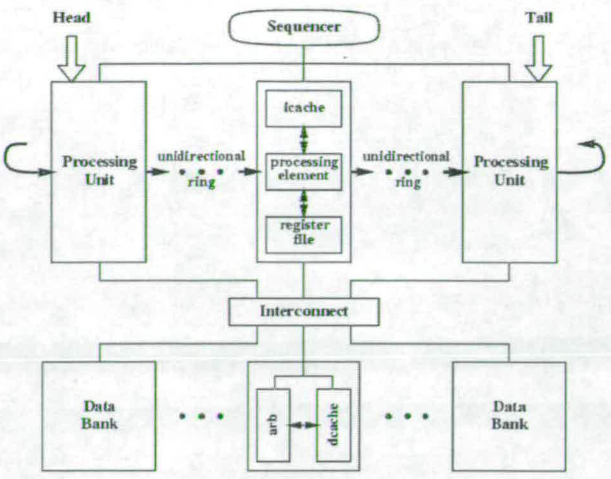


Figure 1.4: Example Multiscalar Architecture

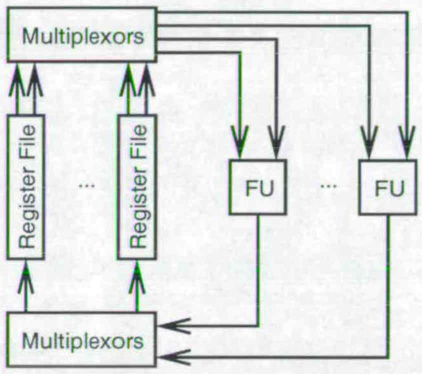


Figure 1.5: Partitioned Register File Architecture

1.2 The SRF Approach

1.2.1 Register Windows

The concept of register windows was introduced in the Berkeley RISC and SPARC architectures. It was conceived as a means to address the problem of efficient communication of register values between procedures in a program. Conventionally, procedure calls have to use the stack, *i.e.* the main memory hierarchy, to communicate register values and the time required for writing to the stack is quite considerable, firstly because all the registers of the machine have to be saved and secondly because main memory accesses are slow.

In an architecture with register windows, procedures can only use a subset of the RF at a time, the current register window, and this is composed of a set of input registers, for receiving data from the calling procedure, a set of local registers, for storing local variables, and a set of output registers for sending data to a called procedure.

Along with register windows came the idea of overlapping registers. The input and output registers of communicating procedures overlap. When a procedure calls another it writes to its output registers and after the call the output registers of the caller become the input registers of the callee. This is achieved through the use of the current window pointer (CWP), a register which points to the start of the current register window in the RF. A register windows RF is composed of a number of windows as shown in Figure 1.6. In addition, the last window overlaps with the first one.

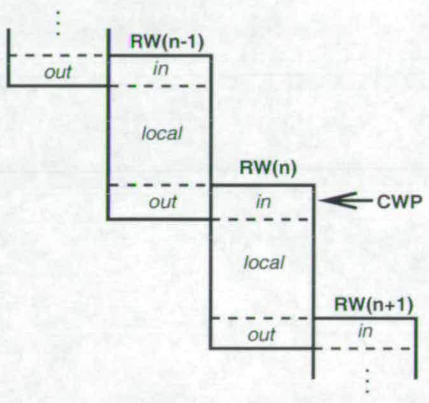


Figure 1.6: Register Windows

It is possible for overflow to occur, if the function call depth exceeds the number of windows. In that case, the main memory has to be used, so capacity is still a problem.

1.2.2 Overlapping or Shared Register Files

Register windows establish a communication pattern for communicating procedures of a single thread of code. Communication can be thought of as occurring in the time dimension, as when one procedure calls another, it stops, then the latter takes over until it is finished and then control is returned to the former.

In multiprocessor systems, the problem of communication through main memory between processors is similar to that of communication through main memory between procedures in a uniprocessor. The overlapping registers idea of the register windows scheme inspired the idea of *overlapping register files* or *shared register files*, where portions of RFs overlap and multiple RFs share a portion of their registers. A scheme of shared register files (SRFs) is shown in Figure 1.7.

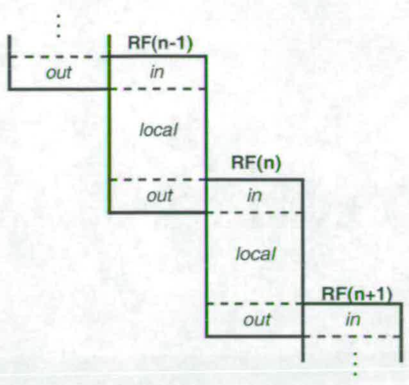


Figure 1.7: Shared Register Files

Shared register files establish a communication mechanism for multiple threads of code running on separate nodes. Each of these threads is executing using different RFs. Communication takes place in space, rather than time, as data between these threads is shared through shared registers.

In this scheme, multiple neighbouring RFs share registers for communication. If RF(n) wants to communicate with RF(n+1), it writes to its output section and then the data can be read from the latter's input section. The shared portions are physically shared. The scheme shown establishes a unidirectional connection between the SRFs. It is possible to expand this scheme for multi-way communication as will be shown later in this thesis.

1.2.3 SRFs in Architectures

SRFs provide a fine-grain communication mechanism for scalable architectures by providing a simple, efficient and scalable method for segmenting the centralised RF. The details of such an architecture will affect the SRF sharing scheme but the SRFs themselves do not impose a particular architecture. Different configurations of SRFs are discussed in Chapter 4.

SRFs manage locality by providing local sections for local processing and shared sections for inter-RF communication. The design and implementation process of SRFs will show that they have a natural mapping to implementation and not only provide conceptual but also physical locality for the local sections and a scalable connection method for communication.

SRFs reduce the communication overhead by providing flexibility in the degree of communication and by their ability to communicate multiple values at the same time. The degree of communication is reflected by the number of shared registers, the number of shared sections in an SRF and the register sharing scheme.

Although only one value can be written at any one time in any one SRF, (unless multiple write ports are used), shared sections can be accessed directly by all neighbours without the need for arbitration. This provides the potential for multi-way communication with low complexity.

SRFs provide implicit synchronisation at the register level through the use of a register locking mechanism, which is much more desirable for exploiting concurrency than coarse grain process level synchronisation.

1.3 Timing in Circuits and Architectures

The most common approach to designing digital control circuits is the synchronous one, *i.e.* the utilisation of a timing reference signal, called a clock, for separating system states. In high-level terms, the clock signal has two phases, an active phase and a wait phase. During the active phase, an operation is performed, whereas during the wait phase communication of results takes place. The clock period, *i.e.* the time between the clock signal changes is determined by the speed of an operation.

Because of the dominance of the synchronous approach in circuit design, most contemporary systems including processor architectures are synchronous. Historically, the synchronous approach has dominated due to its simplicity, however with today's complex systems it is no longer clear whether the synchronous approach is still the simplest one.

1.3.1 Synchronous Systems

Synchronous systems are time-driven. The most important parameter of a synchronous system is the clock period (or the clock frequency, its inverse), which is either specified before a system design, or estimated after a system has been designed. All clocked blocks in a synchronous system must have delays which are less than the clock period for correct circuit operation.

The most common way of increasing the performance of a synchronous system is to make the clock period smaller. This involves identifying the longest paths in the control circuit logic, the critical paths, as they are called, and attempting to make them faster. The global nature of the clock signal implies that all of the control circuits must have delays of the same order. Therefore, operations longer than a specified clock period must be broken down into suboperations which take less than or equal to the clock period. This implies than if the speed of an operation is not an exact fraction of the clock period, a performance penalty is incurred.

This is one of the difficulties of synchronous design, *i.e.* that inhomogenuities between circuit speeds cannot be efficiently accommodated.

The implementation of synchronous circuits is becoming more and more difficult as the density of integration increases and higher clock frequencies are required in order to achieve high-performance. Clock buffering and clock skew are becoming important problems in GSI. As the number of devices that can be implemented on a single chip is increased, so must the drive strength of the clock to drive them. Therefore, the silicon area which is occupied by the clock routing and buffers is increasing. In addition, minimising clock skew, *i.e.* the differences in the arrival time of the clock at different circuit parts, is a hard problem, because both the clock buffers and the wire delays must match among different circuit parts. In addition, as the wire delays are becoming increasingly significant, the amount of die area which is reachable in a single clock cycle is dropping [Mat97].

The power consumption of synchronous circuits is another problem. Not only the clock buffers and clock routing, but also circuits which are inactive consume a lot of power without performing any useful function. The increasing power dissipation of the clock, with increasing clock frequencies and integration densities, has presented the need for power management techniques. In a synchronous circuit, current is drawn globally when the clock switches. This maximises radio interference at frequencies which are harmonics of the clock frequency.

1.3.2 Asynchronous Systems

Asynchronous (also called self-timed) systems do not rely on an external timing reference. They are composed of asynchronous circuit blocks. Communication between these blocks is no longer based on timing, but on an asynchronous communication protocol. Each of these blocks is responsible for communication with other blocks and for detecting the completion of its operation.

In this way, each block operates autonomously, taking only as much time as is necessary to perform its function, rather than waiting for the next clock transition to occur. It is often the case that the time required for the completion of an operation is variable and depends on the operands, rather than being fixed. In such cases, an asynchronous implementation is more advantageous, as it can accommodate these variations and does not incur a performance penalty. In a synchronous implementation, the clock period must allow for the worst case, the critical path, yielding worst-case performance for the particular circuit being implemented, whereas an asynchronous implementation that is data-dependent will yield average-case performance.

The most important advantages of asynchronous systems, which are direct consequences of the autonomy of their constituent components, are compositionality and therefore scalability, expandability and ease of improvement.

Compositionality and scalability are key aspects for scalable performance, as they allow for the exploitation of parallelism. Expandability and ease of improvement relate to compositionality. Once a system has been specified and possibly implemented, it is generally possible to expand one of its parts, without having to make global changes. In addition, it is possible to replace an asynchronous component with an improved one, without having to make any changes to the system.

The asynchronous system model is well suited to the GSI era, as the communication mechanism does not rely on explicit timing assumptions. Its properties make it particularly attractive for implementing systems-on-a-chip, the latest trend in semiconductor manufacturing, where single-chip systems can be constructed from standard circuit blocks. The asynchronous model solves the important problem of interfacing between these independent blocks and allows for scalable and expandable systems to be implemented. The absence of a clock signal is also advantageous for mixed analogue and digital circuits, as the clock poses interference problems for analogue circuit parts, which are usually separated, for this reason, from the digital parts as much as possible.

1.4 Asynchronous Processors

Historically, asynchronous architectures first appeared in the 1950s. One of the earliest machines to exploit asynchronous operation was the Atlas machine [KELS62], designed at the University of Manchester in the late 1950's. Asynchronous operation was used because Atlas had a single accumulator for floating-point instructions and the floating-point unit had a much longer latency for multiply and divide operations than it did for add or subtract. The MU5 Computer System [IC78], the successor to Atlas at Manchester, inherited this asynchrony and exploited it more extensively. It employed asynchronous communication among processor units (although some of the units were internally synchronous) and between processor units and memory, firstly to allow different instructions to follow different paths through the various sections of the pipeline and secondly to address the problem of variable functional unit delays.

The complexity of asynchronous circuit design and the preconception that asynchronous circuits are wasteful in logic and area impeded asynchronous processor design. But the problems of the synchronous approach gave new potential to research into asynchronous architectures. In 1989, the first fully-asynchronous microprocessor [MBL+89] was designed and implemented at the University of California, based on Martin's Communicating Processes method [Mar90a]. In the same year, Sutherland's asynchronous micropipelines [Sut89] gave new potential to asynchronous research.

Today, a number of asynchronous architectures exist, some of which have been fabricated and found to operate correctly. The AMULET family of processors first appeared in 1994 with AMULET1 [FDG+94]. AMULET2 [FGR+97] and AMULET3 [GFC99] followed. The AMULET processors implement the ARM instruction set and follow the micropipeline approach. The first processor, AMULET1, employed a 2-phase, bundled data design style and a register locking mechanism for respecting dependencies between instructions. The AMULET2 used a 4-phase design style and was a more complex architecture including data forwarding and branch prediction. The AMULET3 processor implements the latest version of the ARM architecture which includes "thumb" instructions, a set of compressed instructions to improve code density. In the AMULET3 the register locking mechanism has been replaced by an reorder buffer.

The MiniMIPS Processor [MLM+97] developed at the University of California is an asynchronous MIPS R3000 architecture. It implements precise exceptions and allows for bypassing of the register file. The MiniMIPS allows for out-of-order execution of instructions and employs an instruction queue for writing back the results of instructions in the original program order.

The Counterflow Pipeline Processor [RFS94] developed at Sun is a novel type of architecture which mixes instruction and data flow. In a counterflow architecture instructions and data flow in opposite directions. Instructions look for their operands and then for a pipeline stage where they can be executed. The problem with counterflow architectures is the circuit complexity of the logic required at the pipeline stages.

The TITAC [NUK+94] and TITAC2 [TKI+97] processors were developed at the Tokyo Institute of technology. TITAC implements a simple accumulator-based instruction set. TITAC2 is an asynchronous pipelined processor with a five stage pipeline. Due to the fact that both processors use a dual-rail encoded datapath, they have a high gate count compared to a synchronous equivalent datapath.

The SCALP processor [End96] is an asynchronous architecture aiming at power efficiency. The SCALP processor allows for explicit forwarding of instruction operands by encoding functional unit identifiers into the processor's instruction set. It still uses a register file, as values cannot be forwarded beyond branches and often the results of one instruction are used by several others.

As industry is beginning to realise the benefits of asynchronous circuit design, commercial asynchronous chips are beginning to appear. Commercial asynchronous chips include the Cogency DSP [PDF$^+$98], the Philips 80C51 microcontroller [vGvBP$^+$98] and the Sharp DDMPs [TMI99]. Cogency's asynchronous DSP and Philips' 80C51 are both compatible with synchronous versions.

The Cogency DSP architecture employs the 4-phase, bundled data protocol and implements a three stage pipeline. One of the interesting features of this architecture is the implementation of communication between the datapath units. A central control unit sends control signals to the datapath units instructing them to stall or proceed, hence resolving data dependencies in the datapath. This feature saves on the chip area compared to the synchronous design. The asynchronous version shows a reduction in power consumption of up to 47% and a great difference in electromagnetic interference. The Philips 80C51 microcontroller demonstrated a power reduction of about 25%.

1.5 Aims of the thesis

This thesis investigates asynchronous circuit and asynchronous processor design.

In the area of asynchronous circuit design, it presents the CMOS direct-mapped asynchronous finite state machine approach. The simplicity of this approach, and its robustness are demonstrated.

In the area of asynchronous processor systems, it investigates the circuit implementation of hardware structures for supporting concurrency and scalability. Two approaches are presented; Shared Register Files (SRFs) and μnet architectures.

The SRF approach can be used to implement clustered uniprocessors or single-chip multiprocessor systems and aims at exploiting program level concurrency. The implementation of SRFs and the effect that register sharing has on the access times of register files are studied. An SRF system is contrasted, at the layout level, with the more conventional bus-based alternative.

The μnet approach [Reb96] is an architectural approach for exploiting fine-grain parallelism in an asynchronous datapath. It is capable of exploiting a higher amount of instruction level parallelism than a conventional pipelined architecture. An implementation methodology is presented for both scalar and superscalar

μnet-based architectures.

All of these ideas have been implemented in a prototype chip design, the A1 processor architecture, which has been laid out and simulated at the transistor level. The A1 is a dual-node multiprocessor, which employs the SRF approach for communication between the nodes. Each node is implemented with the μnet approach and is capable of executing a simple instruction set.

1.6 Thesis Structure

The structure of the remaining chapters of this thesis is as follows. Chapter 2 reviews some of the problems associated with implementing asynchronous control circuits and describes the CMOS direct-mapped approach for designing asynchronous finite state machines. Chapter 3 presents the basic principles of asynchronous processor systems, along with two methods for exploiting temporal and spatial parallelism, *i.e.* the μnet (micronet) approach and shared register files. Chapter 4 describes the implementation of different configurations of asynchronous shared register files, discusses issues of their implementation and contrasts the performance of a four shared register file system to that of a more conventional bus-based system. Chapter 5 describes the design and implementation of the A1 processor. Chapter 6 presents the conclusions and proposes possible future work.

Chapter 2

Asynchronous Direct-Mapped Finite-State Machines

In this chapter, the problems associated with implementing asynchronous control circuits are described along with the existing approaches for designing asynchronous finite-state machines. The CMOS direct-mapped approach for designing asynchronous finite-state machines approach in CMOS technology is presented. Based on the one-hot encoding method, it is a simple, elegant and intuitive approach that produces regular, fast asynchronous control circuits.

2.1 Digital Circuit Design

Digital circuits can be divided into control circuits and datapaths. A datapath is a set of interconnected elements through which data flows for an operation to be performed. The operation of the datapath elements is controlled by control circuits.

A digital control circuit can be combinational or sequential. In a combinational circuit the output signals are functions only of the input signals, whereas a sequential circuit has internal state, and its output signals, as well as its future internal state, depend both on its inputs and its current internal state. Sequential circuits are also referred to as finite state machines (FSMs).

Although certain circuits can be implemented combinationally, it is frequently the case that a circuit cannot be implemented as an input-output mapping. This occurs when it is necessary for a circuit to perform a sequence of operations, "remembering" the step in the sequence that it is currently in. In such a case the storage of an internal circuit state is necessary and the circuit must be implemented sequentially.

2.2 Circuit Specification Methods

In order to implement a circuit its behaviour must first be specified. There are various ways of specifying a circuit's behaviour [Hau93], some of which are closer than others to the details of the implementation. After the specification has been produced, it may need to be converted into an implementable form. It is possible, for example, to specify a circuit at a high level of abstraction, for example as a program or as a graph of transitions, and at that level even the nature of the circuit, *i.e.* whether it is combinational or sequential is not obvious. Such a specification will need to be analysed and refined into a lower level specification. The lowest level of a circuit specification can be directly mapped to a circuit implementation.

For combinational circuits, a directly implementable specification is a simple function that defines an input-output mapping. For sequential circuits, it is a sequential function, *i.e.* a function that maps inputs to outputs assuming a particular internal state. The input-output mapping for combinational circuits is specified by a truth table. A truth table is a one dimensional array, where rows represent the input signals and the table entries the output signals. The truth table for an AND gate is shown in Figure 2.1. Signals a and b are the inputs and o is the output.

a	b	o
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2.1: Truth Table for an AND gate

Sequential circuits can be specified in a tabular form as a flow table [Ung69] or in a graph form as a state diagram. A flow table is a two dimensional array, where columns correspond to input values, rows to the internal states and the table entries are ordered pairs representing the next internal state and the current output respectively. Figure 2.2 shows the flow table specification of a 2-bit counter with an input x and two outputs. The numbers in the left-hand column represent the circuit states and the table entries represent the next state and the outputs. When the next state is the same as the current state, the circuit is stable and the next state table entry is shown in brackets.

A state diagram represents the relationship between inputs, states and outputs graphically. States are represented by vertices and transitions between states by

	x	
	0	1
1	(1), 00	2, 01
2	3, 11	(2), 01
3	(3), 11	4, 10
4	1, 00	(4), 10

Figure 2.2: Flow Table for a 2-bit grey-code Counter

labelled edges that connect the states. The circuit outputs are specified for each vertex in the graph, *i.e.* for each circuit state. The state diagram for the 2-bit counter is shown in Figure 2.3. The labels inside the states indicate the state and the circuit outputs. Note that the stable transitions are not shown.

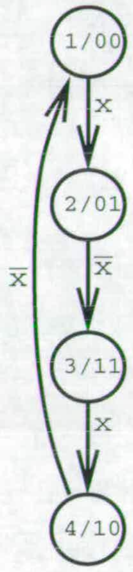


Figure 2.3: State Diagram for 2-bit Counter

2.3 Implementation of Digital Control Circuits

2.3.1 Transistors and Logic Gates

The fundamental element used for the implementation of modern electronic circuits is the transistor. A transistor (**trans**fer-re**sistor**) is an analogue electronic device with three ports, such that the voltage or current across or through one pair of these ports, controls the current through the other pair.

CMOS (Complementary Metal Oxide Semiconductor) technology [WE93] provides two transistor types, an n-type transistor and a p-type transistor, fabricated by using negatively doped and positively doped silicon respectively. The MOS structure provides a gate, a source and a drain port. The gate port controls, by a field-effect mechanism, the flow of current between the source and the drain ports. CMOS transistors can be used to implement digital logic gates, *i.e.* digital circuits that perform the fundamental logical operations, such as NOT, AND, OR, NAND, NOR, XOR, etc.

2.3.2 Combinational Logic

Combinational circuits can be implemented by connecting together transistors or logic gates to produce the output signals, *c.f.* Figure 2.4. In CMOS, so called complex gates can be used to produce an output signal depending on a number of inputs [WE93][Bla92]. These typically contain a pull-down network of n-types and a pull-up network of p-types, Figure 2.5. The function of the pull-down network is to short the output of the gate to logic 0 and of the pull-up network to short the output to logic 1. For correct digital operation these must be mutually exclusive.

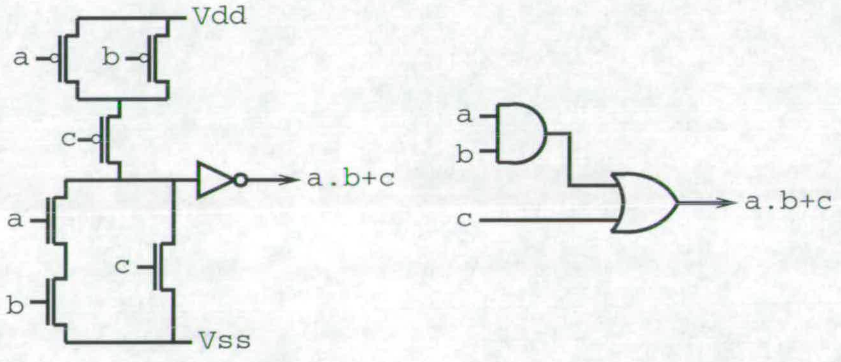


Figure 2.4: Different Realisations of a.b+c

2.3.3 Sequential Logic

Implementing sequential circuits is a more complex task - sequential circuits require feedback. Firstly, the internal state of the circuit must be stored in memory elements. Secondly, the future internal state must be produced and this must not interfere with the current state. The circuits that produce the future state and the outputs are combinational. The general form of a sequential circuit or finite-state-machine (FSM) is shown in Figure 2.6.

The internal state is represented by the internal state variables, which must be used as inputs to the circuit to produce the future internal state, or the excita-

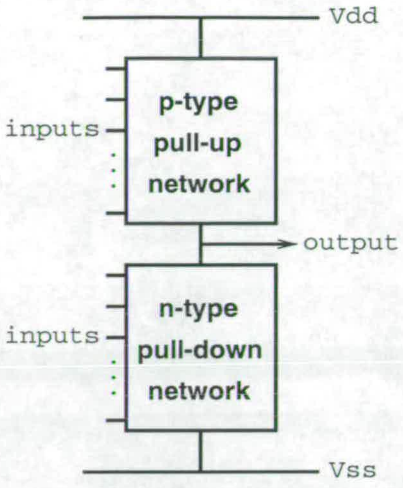


Figure 2.5: Complex CMOS Gate Structure

tion state variables. When the future internal state is different from the current internal state, the circuit is unstable and changes state. The number of state variables depends on the number of states of the circuit, for n states, a minimum of $log_2 n$ state variables are needed.

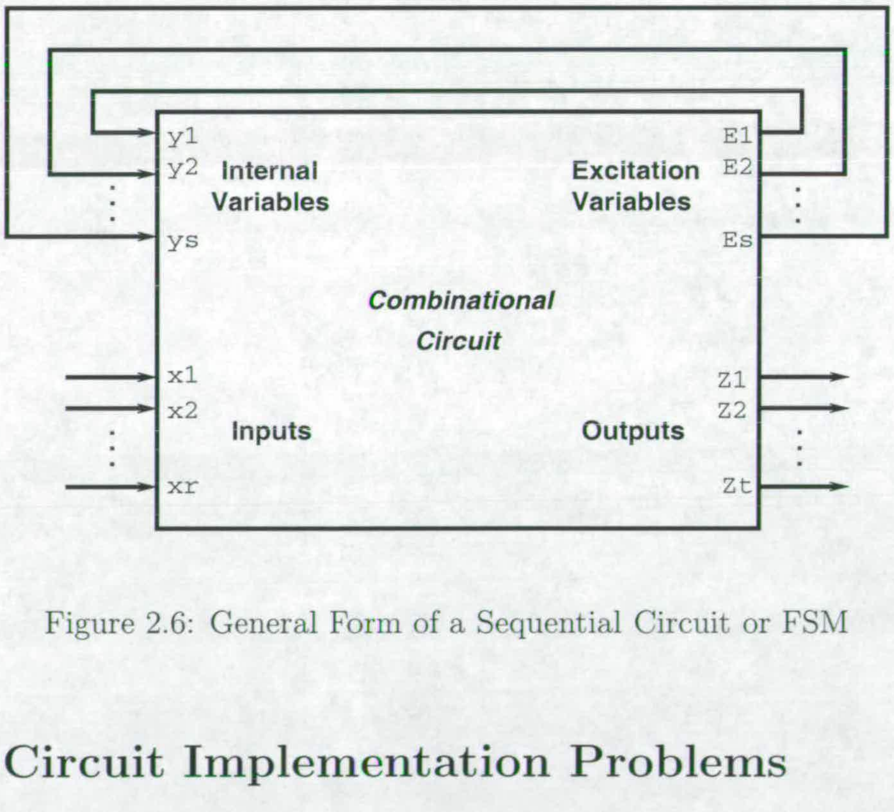


Figure 2.6: General Form of a Sequential Circuit or FSM

2.4 Circuit Implementation Problems

Although digital circuits use discrete values and can be modelled in a discrete manner, they are implemented using inherently analogue devices and interconnections which have a continuous rather than a discrete response and finite, varying delays depending on physical parameters such as device sizing and path length.

This mismatch between the logical and physical implementation, and in particular the finite, varying delays between circuit components, can lead to circuit failures called hazards. Depending on the nature of the circuit, combinational or sequential, different types of hazards are possible. Sequential circuits also present another problem, races. These circuit failures are described below.

2.4.1 Hazards

Delays in circuit elements and the interconnect can produce transient errors, called hazards [Mil65][Ung69], that may also cause incorrect circuit behaviour. Two types of hazard exist depending on whether the circuit is combinational or sequential.

2.4.1.1 Combinational Hazards

The existence of hazards in a combinational circuit depends on its implementation. A combinational circuit implemented by simple logic gates does not necessarily have the same hazards when implemented as a complex gate. The advantage of using complex gates is that they minimise the possibility of hazards because they localise the interconnect and element delays in a small area. In addition, the existence of hazards in a combinational circuit depends on its implementation, for example as a sum-of-products or as a product-of-sums. According to switching theory, in single-input-change (SIC) combinational circuits, *i.e.* circuits where only one input is assumed or allowed to change at a time, two types of combinational hazards are possible, static hazards and dynamic hazards, shown in Figure 2.7.

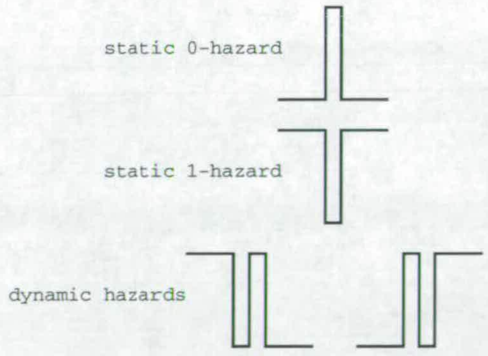


Figure 2.7: Combinational Hazards

A static hazard is a short, positive (0-hazard) or negative (1-hazard) glitch on a combinational logic output that should have been stable, *i.e.* not change value after an input change. In logic gate circuits, static hazards are caused by differences in propagation delays in the logic gates and the interconnect. Given any combinational function, a sum-of-products expression can be realised with no 1-hazards, provided that not only are all the 1-points covered by a single product term, but also all adjacent 1-points are too. Sum-of-products circuits implemented by logic gates or as complex gates do not contain 0-hazards, as for a 0-hazard to occur the minterms must include both a signal and its complement.

Dynamic hazards occur on input transitions that cause the output to change. They manifest themselves as a superfluous output changes, before the output settles to its final value. As with 0-hazards, they cannot occur in SIC sum-of-products circuits.

If multiple-input-changes (MIC) are allowed in a combinational circuit, a third type of hazard is possible, a function hazard. Function hazards are multiple output transitions in response to multiple input transitions, when the output should have remained stable. Function hazards can occur even in basic gates, but only for MIC. They cannot be removed through logic design and are a property of the implemented combinational function. They occur when, for an input transition, the minimum length path between the transition points in the function's Karnaugh map contains more than one function change. For MIC combinational logic, if more than one prime implicant is present in a function, the possibility of a function hazard exists. To stop function hazards from occurring, constraints must be set for the environment that provides the inputs, or the circuit must be changed to implement a different specification.

2.4.1.2 Sequential Hazards

In FSMs, Figure 2.6, where the internal state outputs are fed back to the internal state inputs, incorrect circuit operation is possible due to sequential hazards. Three types of sequential hazard are possible, essential hazards, d-trio hazards and transient hazards.

Essential and transient hazards occur when the internal state change is perceived by some part of the sequential circuit before the input change, due to propagation delays in the circuit. If that part of the circuit produces another state variable, then the hazard is an essential one; if it produces an output, then it is a transient one. Practically, the problem is that there is not enough delay between the outputs of the internal states and the inputs, *i.e.* the current and future states are not properly isolated.

Switching theory specifies the presence of an essential hazard in the following way: for an input x and a total state T^1, assuming that the system is initially at T and that x changes once, then if the total state T' reached is not the same as the one reached if x is changed two more times, an essential hazard exists. Transient hazards happen because the input change is perceived at the same time as the internal state change and that causes a combinational hazard at an output signal.

D-trio hazards, also called nonessential hazards, occur when, for an input change, the circuit does eventually settle to the correct final state, but does so by going through a third state, different from both the initial internal state and the final state. Therefore, the state transitions are correct but the path through this third state may cause hazards at the outputs. Formally, for an input x and a total state T, assuming the system is originally at T and x changes once, then if the total state T' reached is the same as the one reached if x is changed two more times, but on the second change, a different state T'' is reached, then a d-trio hazard exists.

Essential and transient hazards are a property of the sequential circuit implementation and can be detected by the flow table and the Karnaugh map of the output logic. They cannot be removed by logic design.

2.4.2 Races

A race condition occurs in a sequential circuit when more than one state variable must change in the course of a state transition. If the correct behaviour of the circuit depends upon the outcome of the race, *i.e.* the order of state variable changes, then the race is called critical.

Critical races in a sequential circuit can be removed by changing the assignment of state variables. For a sequential circuit to be free of critical races, the assignment of state variables must be such that every transient intermediate state between a starting state and a destination state also produces the same destination state. This may require more state variables to be added.

2.4.3 Example

As an example of a circuit with hazards, consider the implementation of the 2-bit grey-code counter that was specified in Figures 2.2 and 2.3 as an asynchronous FSM.

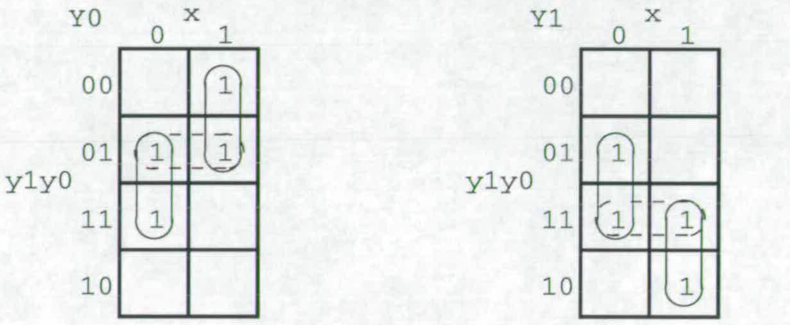
Figure 2.8 shows its flow table specification and the state encodings assigned to states 1 to 4. In this case, it is possible to encode the states to have the same

[1]total state is the set of inputs and internal state

	x		
	0	1	state enc.
1	(1), 00	2, 01	00
2	3, 11	(2), 01	01
3	(3), 11	4, 10	11
4	1, 00	(4), 10	10

Figure 2.8: Flow Table for a 2-bit grey-code Counter

value as the outputs to save on the circuit's size. State changes in this example are sequential, 1→2→3→4→1, so the encoding is valid and no races are possible, as for any state change, only one state signal changes at a time.



$Y0 = x.\overline{y1} + \overline{x}.y0$ $Y1 = x.y1 + \overline{x}.y0$

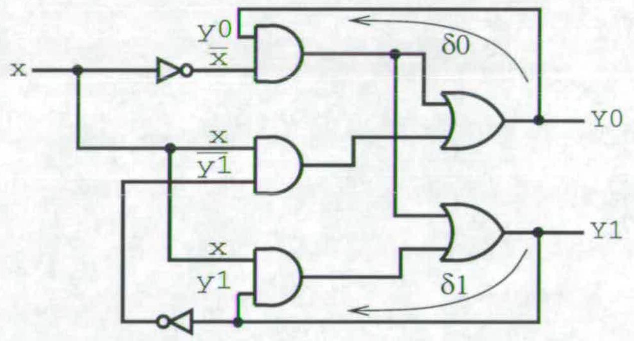


Figure 2.9: Realisation of the 2-bit grey-code Counter

Figure 2.9 shows the Karnaugh maps derived from the flow table specification for the output signals Y0 and Y1, their logic equations and the minimum gate circuit. The logic equations that produce the minimum gate circuit are formed by the prime implicant groupings indicated by the solid lines in the maps.

Signals y0 and y1 are the fedback versions of Y0 and Y1, the previous state, delayed by the feedback delays $\delta 0$ and $\delta 1$ respectively, and perceived as inputs. This minimal gate circuit contains 7 gates, and as, in CMOS, the 2-input AND and OR gates are composed of 6 transistors each and the inverters of 2, the total number of transistors required for this circuit is 34.

This minimum gate circuit has both combinational and sequential hazards, if arbitrary gate delays are assumed. In terms of combinational hazards, the outputs have static-1 hazards for `x` transitions when `y1y0` are 01 for `Y0` and 11 for `Y1` respectively, because, for these transitions, the output variables cross the prime implicant groupings. For these to be removed, the minterms indicated by the dotted lines must also be included in the equations. Dynamic hazards are not possible in this circuit as it only has one input. Sequential hazards can be detected from the flow table. Starting from any stable state, if `x` changes three times, the resultant state will be different from the case where `x` changes once. For example, in state 1, if `x` is originally 0 and changes three times 0→1→0, then the final state will be 3, but if it changes once 0→1, it will be 2. Hence, the circuit has essential hazards for all states. As the outputs are also the states, in this case the essential hazards are also transient ones. There are no D-trios.

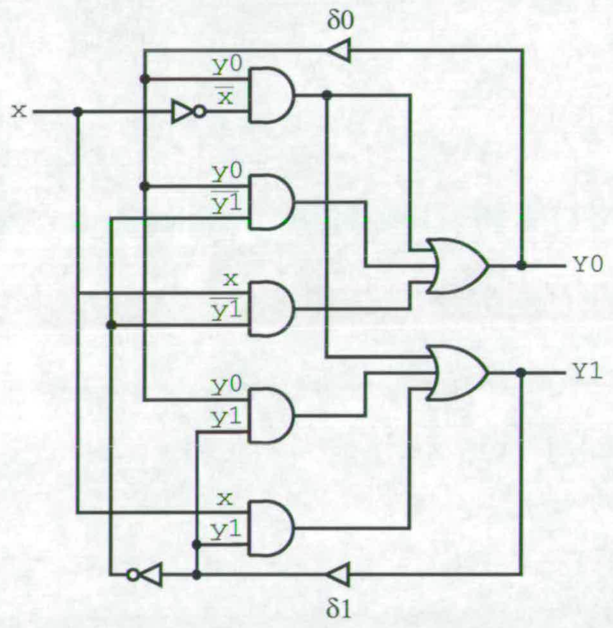


Figure 2.10: Hazard-Free Realisation of the 2-bit grey-code Counter

Figure 2.10 shows the hazard-free realisation of the circuit. The static hazards are removed by adding the extra minterms. The essential hazards are eliminated by controlling the feedback delay of the output variables. Delay elements with delays $\delta 0$ and $\delta 1$ are shown. The condition for correct circuit operation is $\delta 0$, $\delta 1$ $> \delta_{inverter-x}$, *i.e.* the feedback path delays should be greater that the delay of the inverters that produce the inverses of `x` and `y1`. The number of transistors in this circuit is 50, not counting the delay elements.

2.5 Delay Models

In order to detect the existence of hazards or to verify their absence in a combinational circuit or an FSM, delay models are used. These models assign delay values to devices and interconnect. The delay values are modelled as taking on any value between zero and some upper bound. Various different delay models have been described in the literature.

The most general delay model is the delay-insensitive one. This assigns delays to devices and interconnect, and all interconnect paths are considered to be independent. A circuit which works correctly under the delay-insensitive model is independent of the fabrication technology. The delay-insensitive model is the most robust and mathematically elegant, but true delay-insensitive implementations are typically unrealisable [Mar90b].

The speed-independent delay model assigns delays only to devices, assuming that interconnect delays are negligible, *i.e.* this model does not consider the effects of interconnect on the circuit design.

Two delay assumptions, the isochronous fork and the equipotential region model specific delay constraints in more detail. The isochronous fork assumption is that the difference in delays on a given set of electrically connected wires is insignificant. A signal driven on an isochronous wire propagates across the interconnection so that it reaches all its destination devices simultaneously. The equipotential region assumption is that a set of independent wires have indistinguishable delays. This assumption requires that the lengths of these independent wires are approximately equal.

The quasi delay-insensitive model is a delay-insensitive model where some of the forked connections must be isochronous for a circuit to be hazard-free. This is a realistic model that tries to bridge the gap between the two extremes of delay-insensitivity and speed-independence.

The choice of delay model depends on the assumptions that can be made about the circuit and engineering knowledge about the ranges of delay values.

2.6 Modes of Operation of Sequential Circuits

In order to work around hazards and races, sequential circuits specify their interaction with the environment, namely their mode of operation. The two most general modes of operation are synchronous and asynchronous. Synchronous circuits only require a single delay assumption, the clock period, whereas asynchronous circuits, depending on their design, may require restrictions on the environment and on the timing of their gates and interconnections.

2.6.1 Synchronous Circuits

Synchronous or clocked circuits utilise a periodic pulse signal called a clock. The clock signal is used to separate the internal system states. This class of circuits filters the future internal state with flip-flops that are activated by the clock signal. The flip-flops may be activated by the level of the clock, *i.e.* high or low, or by the change of the clock (a clock edge), *i.e.* from low to high or from high to low. The former type of circuits are called level-triggered, whereas the latter are called edge-triggered.

Level-triggered clocked circuits feed back the internal states through level-triggered flip-flops. If a single clock is used then for correct operation such a circuit must operate in pulse mode. Pulse mode operation selects the high phase of the clock, so it must be long enough for the future internal states to be generated and for the internal state flip-flops to change state, but shorter than the time required for the change in the flip-flops to propagate back into the circuit. So, pulse mode operation imposes lower and upper bounds on the width of the high phase of the clock. If multiple clocks are used, with multiple level-triggered flip-flops in the feedback path, then this delay assumption can be removed.

Edge-triggered synchronous circuits use edge-triggered flip-flops to feedback the internal states. This type of flip-flop (Master-Slave) absorbs the change in state on a clock edge and is implemented by two level-triggered flip-flops where the one is fed by the clock signal and the other by its inverse. This technique removes the need for multiple clocks.

For correct operation, the interaction between a synchronous sequential circuit and its environment must ensure that its inputs do not change when the circuit is in the active phase, *i.e.* when the clock enables a change of state. Multiple input changes are permitted as long as they happen during the inactive phase of the circuit.

Critical races and sequential hazards are not a problem with synchronous circuits, as the clock separates system states in a discrete manner. So, the assignment of state signals can be arbitrary. This is the motivation for the use of such a system.

The price paid for the use of synchronous circuits is that constraints in the clock must be set to allow for worst cases of circuit delays and manufacturing tolerances. Furthermore, a clock signal must be generated and distributed among all the control circuits in a digital design. Hence, the loading on the clock signal is very significant and a significant amount of buffering is necessary to allow for the required degree of fan-out. Usually, a tree of buffers is used because if the clock signals across different circuits are skewed, *i.e.* they differ in time by a small amount, that may cause the overall system to malfunction, as the skew makes the active phase shorter. In addition, clocked circuits always consume power, even when the inputs do not change and the internal state remains constant. This is because the transitions of the clock signal always switch transistors.

2.6.2 Asynchronous Circuits

Asynchronous circuits do not use a clock, but separate system states by internal signals. The mode of operation of an asynchronous circuit characterises the allowable circuit delays and the circuit's interaction with the environment.

Depending on the delay model that it works correctly under, an asynchronous circuit can be delay-insensitive, speed-independent or quasi delay-insensitive. The class of delay-insensitive circuits is very limited for practical purposes, so the most robust asynchronous circuits are quasi delay-insensitive.

Circuits that impose the constraint on the environment that it is not allowed to change their inputs until their internal state has stabilised are called fundamental mode circuits. Depending on the number of inputs that the environment is allowed to change simultaneously, the circuit is called single-input-change (SIC) or multiple-input-change (MIC).

This thesis will only consider circuit implementations of asynchronous finite state machine specifications. The advantage of implementing asynchronous circuits starting from an asynchronous finite state machine (AFSM) specification is that they can be directly implemented in the form of a sequential circuit (*c.f.* Figure 2.6). In addition, the circuit structure in terms of states, inputs and outputs is apparent and defined by the designer. Higher level approaches, such as petri-nets [Mur89] and signal transition graphs (STGs) [Chu87], micropipelines [Sut89] and programming and compilation approaches [Mar90a][vB93][BS89] have eventually to be converted into an FSM specification before implementation.

2.7 Asynchronous Finite State Machines

In an AFSM, the states of the machine are assigned binary codes, *i.e.* encodings, and the outputs of the FSM are typically a function of the current state and the inputs. In synchronous circuits, the assignment of encodings to states can be unrelated to the circuit operation, whereas in asynchronous circuits, due to the possibility of critical races between state variables, this assignment is of paramount importance.

2.7.0.1 Single-Input-Change AFSMs

In the general case, an AFSM will change state on any input change. The problems involved in implementing AFSMs, namely races and hazards, depend on the state encoding, the input and environment constraints. Unger's work [Ung69] demonstrated that sequential asynchronous FSM circuits are mostly limited to fundamental mode operation due to the common presence of essential hazards. It also demonstrated that only single-input-change circuits without essential hazards can be realised by a delay-free circuit. In practice, the single-input-change requirement may be unrealisable or too restrictive for performance reasons.

2.7.0.2 Burst-Mode AFSMs

Multiple-input-change AFSM circuits require input restrictions or the use of special design/implementation techniques. One technique that permits constrained multiple-input-changes is the burst-mode design style [Ste94]. A burst is defined as a set of signal transitions that can occur in arbitrary order. A burst-mode FSM is one whose input and output activity is separated into bursts. Typically, a set of input transitions takes place, but the machine will stay stable until the input burst has completed, then change state, and then produce an output burst. Three constraints are placed on the circuit by the burst-mode style. Firstly, all inputs and outputs must strictly alternate for all valid state paths, *i.e.* a transition on an input x in a state S must never be followed by another transition of the same type in one of the following states unless there is an opposite transition in between. Because of this property, burst-mode FSMs are transition based rather than level based, as opposed to a general AFSM. Secondly, no transition burst is allowed to be a subset of another from the same starting state, to avoid ambiguity. Thirdly, if a state of the AFSM has multiple destination states, then the input transitions to these states must be mutually exclusive to avoid interference. These restrictions eliminate static and function hazards. However, essential hazards are possible, so for correct operation burst-mode FSMs require fundamental mode. It is possible to implement burst-mode FSMs using complex CMOS gates or combinational logic [Ste94] [DCS93] or by using the 3D burst-mode AFSM approach [YDN92]. The disadvantages of this approach are that (a) it requires more logic to be implemented, due to the form of the flow table and therefore produces a larger circuit, and (b) it implies that fundamental mode operation must be ensured by the environment.

2.7.0.3 Direct-Mapped AFSMs

A different approach to building asynchronous FSMs is the direct implementation or direct-mapped approach [Hol82]. This method will be presented in more depth because a similar direct-mapped approach, but using CMOS complex gates, has been investigated and used to implement the control circuits in this research.

The direct-mapped approach uses the "one-hot" state assignment method to encode the machine states. The one-hot state assignment method [Ung69] assigns a state variable for each row in a flow table, *i.e.* one signal per state of the state machine; when the machine is stable only one of the state signals is asserted. This technique eliminates races between state variables as it does not encode states. It also simplifies the circuit implementation as the equation generating each state is of a regular form, $Y_i = T_i + y_i\overline{H_i}$ [Ung69], where T_i is the transition term, *i.e.* the sum of all state transitions that lead to state i, and $y_i\overline{H_i}$ is the hold term which keeps state i asserted until another state is entered. In a one-hot AFSM the state diagram or flow-table structure maps directly to the circuit level connectivity.

The one-hot encoding method implies that the number of state signals and circuitry required to implement a circuit is directly proportional to its number of states, *i.e.* for an arbitrary circuit with n states, n state variables are required and n circuit blocks to produce these variables.

The one-hot state encoding is a special case of state variable assignment. AFSM approaches that do not use the one-hot encoding do not assign a single state variable signal per state, but a state variable code. This implies that multiple state variables must be decoded, but that less state signals will be needed. Such approaches must analyse the flow-table and particularly the state transitions. Then, the code assignment must be such that no critical races exist between state transitions. Extra state variables may be required and also possibly extra rows, as "bridging" states, depending on the state assignment strategy.

Some of these assignments, like one-hot encoding, are single-transition-time (STT) assignments. This means that a state transition takes place in the time of a single signal transition. The advantage of STT assignments is circuit speed, although they may require more state variables and therefore an increase in circuit area.

Possible state encoding schemes [Mil65][Ung69][Man91] include the shared and multiple row assignments and the connected row sets assignment, all of which are not STT assignments, and STT assignments, such as the one-shot state assignment and unicode STT algorithms. The number of state variables required to implement a flow-table depends on its structure and the state assignment method. The connected row set assignment, for example, is a general type of state assignment that requires $2log_2n$ state variables but is not STT. Unicode STT assignments have been shown to require $\frac{log_2n^3+log_2n}{6}$ state variables.

The advantages of one-hot coding as a state assignment are that it produces fast circuits as it is STT, it eliminates state variable races, without requiring analysis of the flow-table for deriving a row-to-state code mapping, and it produces a regular circuit as the state variable equations are regular. It does, however, require more state variables than other state assignments. Overall, it is an elegant and simple way of implementing asynchronous control circuits.

2.7.0.4 Hollaar's Approach

Hollaar proposed a direct-mapped implementation of one-hot encoded asynchronous FSMs [Hol82] based on set-reset (SR) flip-flops (FFs). This provides an even more regular implementation than the AND-OR form of the logic equations. The set input of the SR-FFs is driven by the transition term combinational logic, whereas the reset input is driven by the inverse of the hold term logic.

An example portion of an AFSM constructed in this way and its state diagram are shown in Figure 2.11. This illustrates the implementation of a sequential section of a state diagram. Portions of states `s1` and `s2` are shown. Note that the SR-FF inputs are active low because NAND gates are used. State `s2` is entered on the transition of signal `x`. The circuit operates as follows: if the input `x` is asserted, then the output of the NAND gate to which `x` is input will drop as both `x` and `s1` are high. The consequence of this is the setting of state `s2` and the resetting of state `s1`.

Although direct-mapped AFSMs are not specifically implemented for non-fundamental mode operation and multiple input changes, they have those abilities. The constraints they impose on the inputs are the following. If an input does not cause a transition from the current state of the FSM, then no input restrictions are imposed on it. Inputs that do cause a transition from the current state must be (i) mutually exclusive if they cause transitions to multiple states and (ii) remain stable until the machine has entered a following state. These conditions must be fulfilled during the circuit design phase. So this approach allows MIC and non-fundamental mode operation as long as the inputs fulfill the above constraints.

During the process of changing state, a critical race is possible. For every state

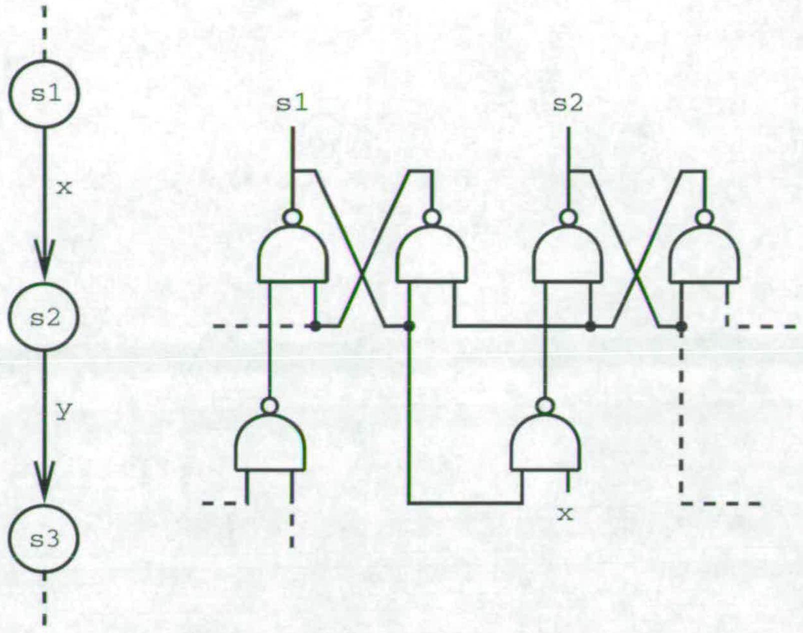


Figure 2.11: Hollaar's One-hot AFSM Example

transition, two state variables change value. The current state variable changes from 1 to 0 and the next state variable changes from 0 to 1. The correct order, which must be ensured by the implementation is 10→11→01, *i.e.* the next state's variable is set first and then the current state's is reset.

In non-fundamental mode operation, where inputs can change before the machine is allowed to settle, the same critical race arises for a sequence of three or more states. In this case, for correct operation, the state flip-flops should not reset before they reset their predecessors. For example, for three states, if their transition requirements are fulfilled, two correct orders are possible depending on the set/reset ordering, 100→110→011→001 or 100→110→111→011→001. In the former, the first state is left as the third is entered, whereas in the latter momentarily all states are active. They are both acceptable as long as the machine ends up in the third state with the first and second states inactive.

For both cases, the race can be eliminated by ensuring uniform gate delays [Hol82][2]. In that way, the next state will be set before the current one is reset for the first case and the resetting of a state will be faster than the setting of the next state for the second case.

A special case which requires particular attention for direct-mapped AFSMs is the implementation of scale-of-two loops. A scale-of-two loop is the case where one state has another state, which acts as both its predecessor and successor. The

[2]Hollaar did not consider wire delays.

implementation discussed before cannot be used for scale-of-two loops.

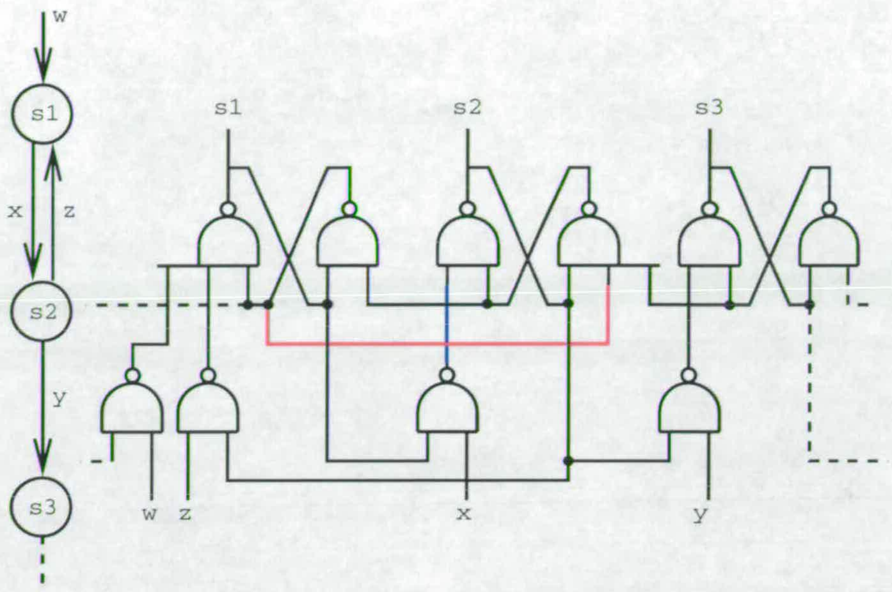


Figure 2.12: Scale-of-two loop implementation

Consider the example in Figure 2.12. Transitions `x` and `z` form a scale-of-two loop. Assume that the circuit is initially in state `s1` and `x` becomes true. Then the set input of the state `s2` flip-flip (coloured blue) will become active. But, as can be seen in the circuit of Figure 2.12, as state `s1` is `s2`'s successor, `s2` is reset (by the connection coloured red) when `s1` is active. Thus, both the set and reset inputs to `s2` will be active when `x` becomes true putting the flip-flop in an unstable state where both its outputs become 1.

This problem occurs because in a scale-of-two loop both transitions 10-11-01 and 01-11-10 are possible and the unstable state 11 may lead to both 01 and 10. The proposed solution is either to convert the scale-of-two loop to a scale-of-three loop and avoid the problem or to modify the reset logic incorporating the input causing the transition. In this example, the reset line of state `s2`, *i.e.* the complement of `s1`, must be replaced by the logical OR of the complement of `s1` and the input that causes the transition to `s2`, `x`. The reset line of s1 must also be modified appropriately. The updated circuit is shown in Figure 2.13.

2.8 Asynchronous CMOS Direct-Mapped FSMs

The asynchronous CMOS direct-mapped FSM is an alternative approach to designing asynchronous control circuits. It is similar to the domino CMOS structure [Bla92] and Hollaar's one-hot encoded state machine. The domino structure

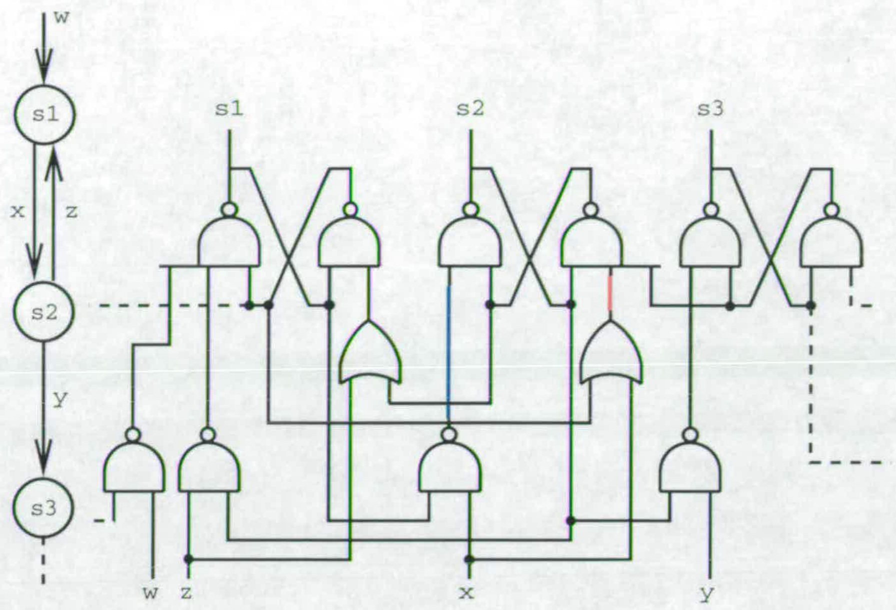


Figure 2.13: Properly functioning scale-of-two loop

is shown in Figure 2.14.

2.8.1 Domino CMOS Structure

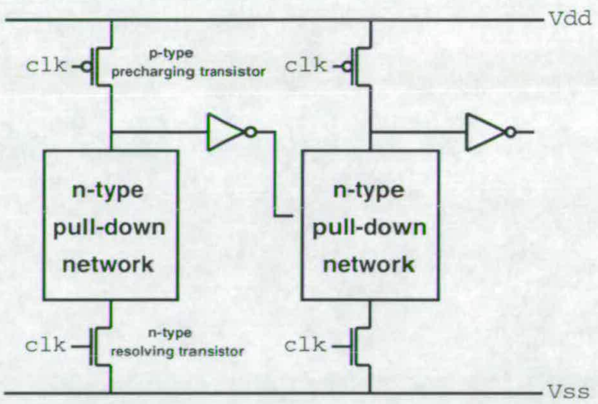


Figure 2.14: Synchronous Domino CMOS Structure

In a domino CMOS structure, a sequence of gates is put into a precharged state and then the gates resolve in series, one after the other like a stack of dominos falling over. Each gate consists of a precharging pull-up p-type transistor, an n-type pull-down network, *i.e.* a set of interconnected n-type transistors, an n-type resolving transistor and an inverter at the output. The purpose of such a structure is to implement a complex logic function dynamically as a function of smaller functions. Typically, the gates are all controlled by the same clock. The function implemented is realised by the n-type pull-down network and the result of the function's output is valid when the clock pulse is high.

2.8.2 CMOS Direct-Mapped AFSMs

In this section, an overview of the CMOS direct-mapped approach is presented. The next section presents AFSM examples and discusses the various characteristics of the approach in more detail.

The CMOS direct-mapped FSM implements a one-hot encoded AFSM using complex state-storing gates (Figure 2.15). This scheme is similar in circuit design to the synchronous domino stage and has the same functionality as Hollaar's approach, only in CMOS. Each state gate is equivalent to one SR-FF with its set and reset logic as used in Hollaar's method. So, each state gate implements one state of the state machine. A state gate consists of a single or multiple p-type pull-up networks, a single or multiple p-type resolving transistors, a single or multiple n-type pull-down networks, a single or multiple n-type resolving transistors and a pair of back to back inverters, acting as a storage element, at the output of the gate. The positions of the resolving transistors and the networks can be interchanged.

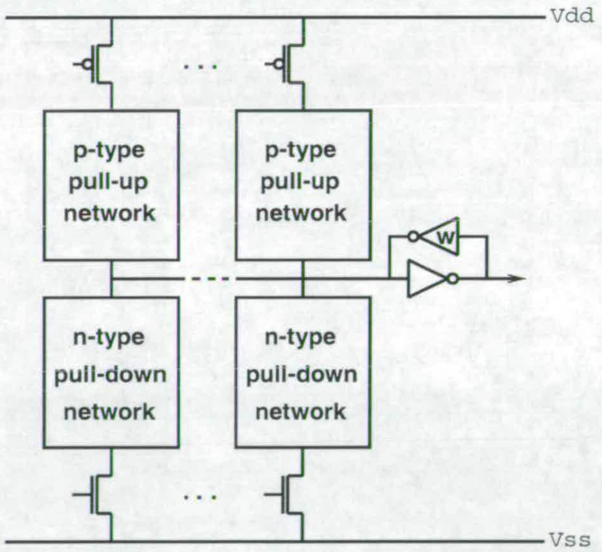


Figure 2.15: Asynchronous Direct-Mapped State Gate

The states of the FSM are the outputs of a group of such gates. When the output of a gate is high (active) then the machine is in that state. Normally, only one state is active at any time, although it is possible to have parallel FSM paths, where multiple states are active simultaneously.

In a state gate, the n-type pull-down networks detect the conditions appropriate to enter the state, like the set logic in Hollaar's approach, and the pull-up networks, like Hollaar's reset logic, detect the conditions to leave it. The pull-down and pull-up networks are activated by the resolving transistors. The n-type resolving transistors are fed with the outputs of the previous states, enabling the n-type pull-down networks to discharge the state gate and so enter the corresponding state. The p-type resolving transistors are fed with the outputs of the following states enabling the p-type pull-up networks to leave the state. The number of n-type pull-down networks depends on the number of states that a state can be entered from and the number of p-type pull-ups on the number of states that are followed from a state. Hence, the logic for entering and leaving a state scales easily.

In the simplest case, there is a single n-type pull-down of a single transistor which corresponds to the signal that triggers the transition into this state. So, a total of two n-types is the minimum. An n-type pull-down network may be more complex depending on the conditions for entering the state. Correspondingly, for p-types, in the simplest case there is a single p-type pull-up network, which in its simplest form is a short circuit, *i.e.* it simply connects the input of the inverter to the p-type resolving transistor. The input of the p-type resolving transistor is fed by the inverted output of the following state, as in Hollaar's approach.

The form of the p-type pull-up network depends on the FSM structure. Normally, only a single resolving p-type transistor is required. In the case where one or more of the previous states of one state are also its following states, *i.e.* there are one or more scale-of-two loops in the state diagram, then there is a need for a more complex p-type pull-up to ensure that the n and p-type parts of the gate are never simultaneously ON. The structure of the p-type network in that case should be the same as the n-type network of the following state, only with inverted inputs. This is discussed in more detail in section 2.8.3.5.

The two back-to-back inverters are used to store the state of the gate, as the gate is not statically driven. They form a 1-bit storage element. The advantage of a dynamic gate is that no hazards, static or dynamic, are possible because the n-types do not have complementary p-types. As in the domino approach, a state gate precharges high (machine state is low), and then the n-type pulls it low. The feedback inverter labelled w is weaker, *i.e.* has smaller width to length ratio ($\frac{W}{L}$) transistors and therefore smaller current driving capacity. Its $\frac{W}{L}$ ratio should be smaller than the equivalent size of the n and p-types so that the state gate can be forced to a different state. This is also discussed in more detail in section 2.8.3.7.

Because of the one-hot encoding, during the process of changing state, a critical race is possible, as in Hollaar's approach. The circuit implementation must ensure that the order of state changes is 10→11→01. This is also achieved by transistor sizing (discussed in more detail in section 2.8.3.6).

A complete FSM is thus implemented by interconnecting the state gates to match the FSM structure. The FSM outputs can be generated directly from the FSM states. On reset, such an FSM must be correctly initialised. This process requires extra pull-up and pull-down reset transistors of appropriate sizing. Normally, only one state should be reset high and all others should be reset low.

2.8.3 The CMOS Direct-Mapped Approach

Apart from the one-hot critical race, this approach does not have the problem of races and is less prone to hazards compared to conventional AFSMs because of the one-hot coding. Hazards in direct-mapped AFSMs are further discussed in section 2.8.3.10. It is potentially faster than a burst-mode FSM or a conventional AFSM as, due to its structure, there are less gate delays between an input and an output change. Also, it does not require fundamental mode for correct operation. As with Hollaar's approach, due to the implementation, the inputs of the FSM can arrive earlier than fundamental mode would allow. Also, multiple input changes are permitted.

2.8.3.1 Simple AFSM Example

Figure 2.16 shows an example portion of a domino AFSM constructed from a linear state diagram.

Signals `ns2` and `ns3` are the inverted versions of `s2` and `s3`. If inputs `x` and `y` are low to begin with and initially the machine is in state s1, then signals `s2` and `s3` will be low.

Now, consider the circuit operation. If input `x` is asserted, the (`s1`, `x`) pull-down chain will pull signal `ns2` low and raise `s2` putting the machine into state s2. Signal `s1` will be lowered at this stage. In state s2 the machine is now waiting for `y` to be asserted to change state. When this happens, the (`s2`, `y`) pull-down chain will pull `ns3` low. This has two consequences, firstly `s3` will be raised by the s3 stage gate inverter and secondly `ns2` will be raised by the s2 p-type resolving transistor. (This is the same process by which state s1 was lowered.) The machine has now entered state s3.

As was mentioned, in the process of entering the following state and leaving the previous one, a delay assumption is present that removes the possible critical race.

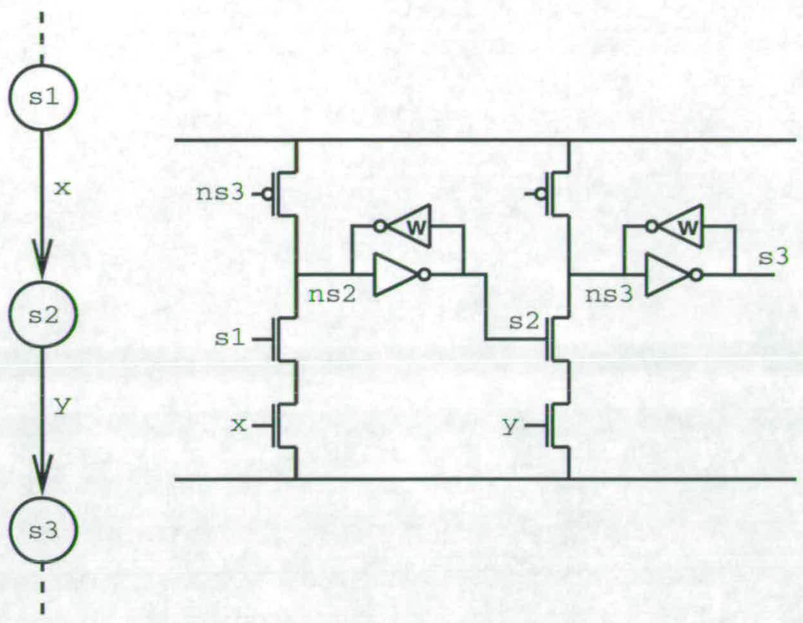


Figure 2.16: Asynchronous Direct-Mapped Example

For correct operation, the following state must be entered before the previous one is left. In this example, faulty operation can occur if `ns3` is lowered, then `ns2` is raised, but `ns3` has not been lowered long enough by the (`s2`, `y`) chain for the back to back inverters to settle and then returns high.

Now, consider the case when the inputs `x` and `y` are asserted simultaneously, *i.e.* non-fundamental mode operation. Then, state `s2` will only briefly be entered and the machine will stop at state `s3`, provided that the conditions for leaving state `s3` are not fulfilled. So, the machine will function correctly for multiple input changes and inputs can arrive faster than fundamental mode operation allows. The length of time that state `s2` is active in this example is dependent on the switching times of the gates and the propagation delays of the interconnect, so it is implementation dependent. This illustrates an important issue in asynchronous circuit design. If state s2 is to be used as an output and feeds to another circuit part, then the FSM must be modified as it is very hard to ensure that its assertion as a pulse rather than a level will be noticed. In that case, the circuit design should be changed so that the condition for leaving state s2 is its acknowledgement by the circuit's environment, and therefore transition `y` should be modified to `y` AND the acknowledgement for output s2. This will ensure delay independent operation.

2.8.3.2 Closed Loops

Now, consider the case where another transition on an input z is added to bring the FSM back to state s1. The modified state diagram and circuit are shown in Figure 2.17. The morphology of the circuit clearly resembles the state diagram.

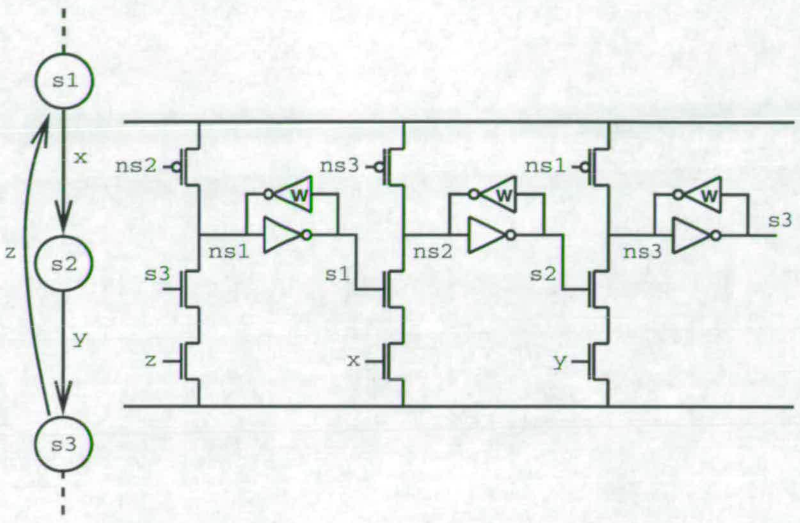


Figure 2.17: Closed Loop Example

In this case, if inputs x, y and z are high simultaneously, the machine will start oscillating between these states. This behaviour is usually unacceptable for the FSM interface, if the states are used as outputs, but the machine will still work correctly to specification. If two of the inputs are mutually exclusive, an assumption which must be guaranteed by the environment, then the oscillation will not occur.

Another interesting case is where `z = not x`. The machine will stop in state s3 and then wait for x to be deasserted. There is a potential hazard in this case. If the inputs x and its inverse are not generated simultaneously, then it is possible to move from state s3 to s2 rather than s1. This can also happen if `y = not x`.

2.8.3.3 Parallel Path Expansion

Another possibility is the existence of a point in the state diagram where a sequential part of the FSM expands into parallel paths. This is illustrated in Figure 2.18.

In this case states s2 and s3 are parallel and can be entered from state s1 on x and y respectively. In the circuit, the output from state s1 is therefore fed to both the state gates.

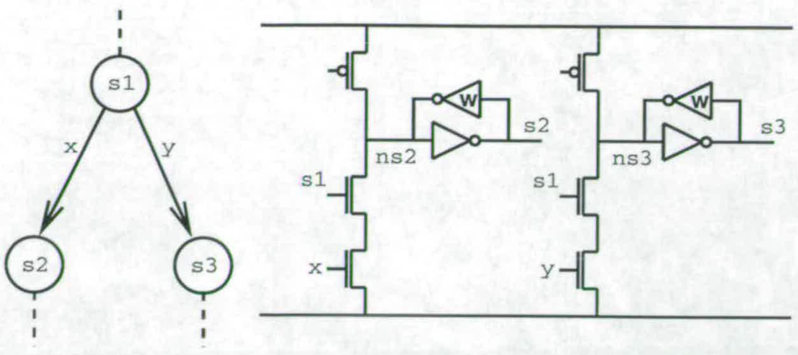


Figure 2.18: Parallel Path Expansion Example, states s2 and s3

Figure 2.19 shows the stage gate for state s1. As state s1 can be left by both s2 and s3 two pull-up resolving transistors are required.

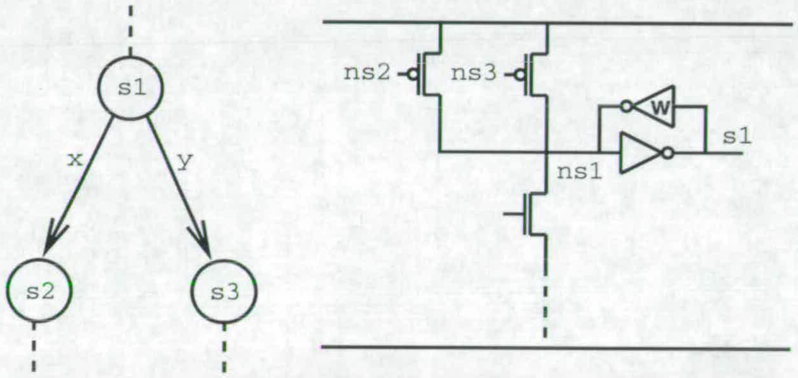


Figure 2.19: Parallel Path Expansion Example, state s1

Assume that the machine is in state s1 and the two inputs are asserted almost simultaneously. Then either both states s2 and s3 will be entered, or one of them will, depending on the arrival timing of the two input signals. This behaviour is not desirable because it is non-deterministic. Therefore signals `x` and `y` should be guaranteed mutually exclusive by the environment. Note that this mutual exclusion property only has to hold when the machine is in state s1, so it is possible to design the circuit in such a way that, before the transition to state s1, signals `x` and `y` are mutually exclusive. The case where `y = not x` is similar, it must be guaranteed before the machine has entered state s1 that both signals are stable.

2.8.3.4 Parallel Path Merging

After a parallel paths expansion, the paths may merge into a single one. This is shown in Figure 2.20.

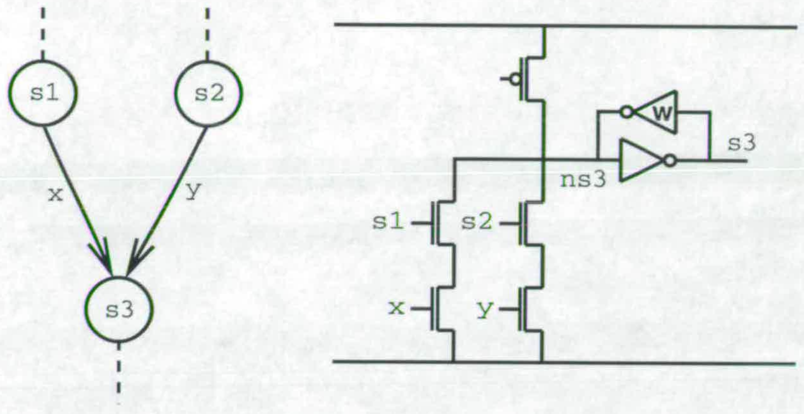


Figure 2.20: Parallel Path Merging Example, state s3

State s3 is the state where the two paths merge, so it requires two pull-down resolving transistors and two pull-down networks. This is because states s1 and s2 are both s3's predecessors. For the same reason, the p-type resolving transistors of s1 and s2, Fig 2.21 are both fed by signal `ns3`, which will go low when state s3 has been entered.

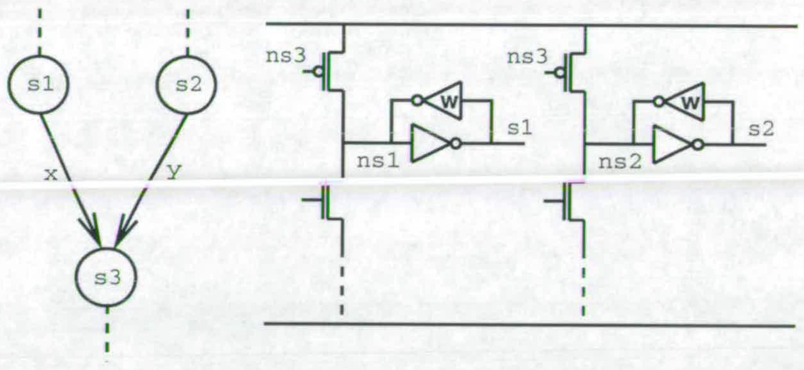


Figure 2.21: Parallel Path Merging Example, states s1 and s2

2.8.3.5 Scale-of-two Loops

As was mentioned in section 2.8.2, in the case of a scale-of-two loop, the p-type transistor circuitry required is more complex than the single resolving p-type transistor used in the previous examples. The reason for this, as in Hollaar's implementation, is that the actions of entering and leaving a state interfere. This problem was illustrated in section 2.7.0.4 and manifested itself as the simultaneous assertion of both the set and reset inputs of a state.

In CMOS direct-mapped AFSMs, the n-types perform the action corresponding to the set input of the flip-flop in Hollaar's approach and the p-types the action corresponding to the reset input. Figure 2.22 shows the implementation of the same scale-of-two loop (transitions `x` and `z`) whose implementation was considered by Hollaar's approach in section 2.7.0.4, Figure 2.13. In this case, the p-type networks are left in their simplest form.

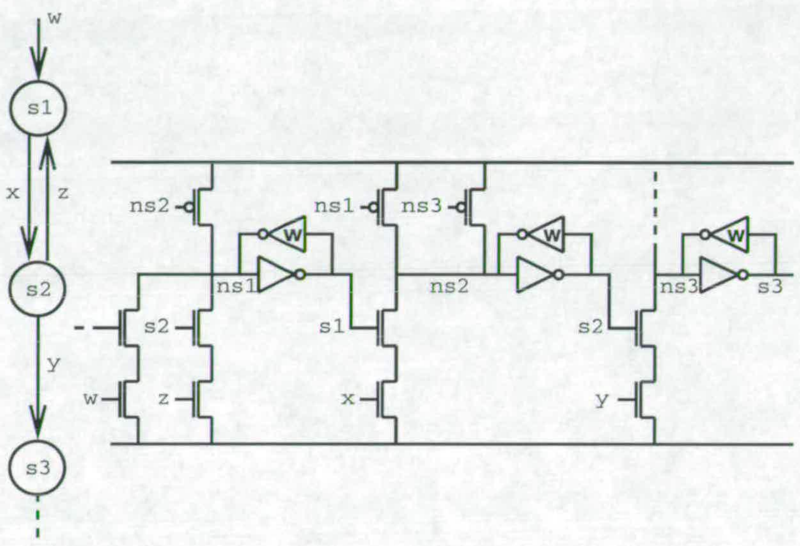


Figure 2.22: Scale-of-two loop CMOS implementation

Consider state s1. It has two predecessors, one state not shown in the diagram and state s2. Transitions to state s1 from these states happen on inputs `w` and `z` respectively. Therefore, two pull-down networks and two pull-down resolving transistors are required. Consider the circuit operation for the transition from state s1. The machine is originally in state s1 and `x` is asserted, so output node `ns2` will be pulled low by the (`s1`, `x`) chain. But s2 is also s1's successor and the p-type with `ns1` as its input will pull `ns2` up. So, the p-types and n-types of state s2 will be ON simultaneously, which is not desirable behaviour as this will not produce a correct logic value for `ns2`. This is similar to the behaviour of the circuit of Figure 2.12 where the set and reset inputs of state s2 are both asserted.

Figure 2.23 shows an updated version of the circuit, where departure from states s1 and s2 is made sensitive to the inputs `x` and `z` respectively, the signals that form the state-of-two loop. Now, when `x` is asserted in state s1, the p-type pull-up chain will be OFF as the input `z` will not be active and state s2 will be entered correctly. The transition from s2 to s1 works in a similar manner. If both

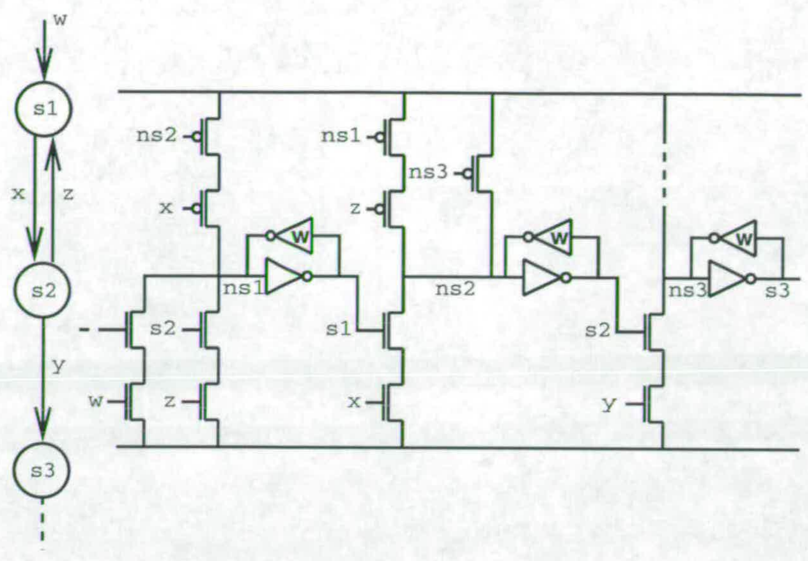


Figure 2.23: Properly Functioning Scale-of-two loop

inputs x and z, which form the state-of-two loop, are asserted simultaneously then both states s1 and s2 will be asserted simultaneously and stay asserted until one of those signals returns low. For example, if both x and z were originally asserted and then eventually x was deasserted, state s1 would go low and the machine would stay in s2. So, even if both inputs in a scale-of-two loop stay asserted, the machine will still function correctly.

In Hollaar's circuit, the reset inputs are active low. For this reason, the inputs to the OR gates that feed them are in the opposite order, z into the reset input of s1 and x into s2. In Hollaar's approach the reset input of a state is ORed with the input causing the transition to the state's predecessor. In the CMOS direct-mapped approach, the extra series p-type is fed with the input causing the transition to the state's successor.

In the general case, where two states *m* and *n* form a state-of-two loop and n is m's successor, the p-type network of the state *m* must be identical to the n-type pull-down of *n*'s. If more than one scale-of-two loop is present between state *m* and its successors, then *m* will have multiple complex p-type networks.

2.8.3.6 Transistor Sizing and the One-hot Critical Race

It was mentioned that the feedback inverter of state gates must be weak, *i.e.* have a smaller $\frac{W}{L}$ ratio than the n and p-types. As the feedback inverter holds the output of the n and p-type chains to its old value, the equivalent sizes of these chains[3] must be such as to force the node to change state.

For example, when a state is low, the output of the n and p-types was high, and is now held high by the feedback inverter. For a state change to take place, it should turn to low, so the n-types must be able to force that node low. Their equivalent $\frac{W}{L}$ should be sufficiently larger than that of the p-type of the feedback inverter that drives that node high. Similarly, the p-types of the stage gate should be sufficiently larger than the n-type of the feedback inverter. During a state change, the n-types or p-types will force their output and that change will be reflected at the output of the forward inverter, *i.e.* the input of the feedback inverter.

The relative sizing of the p and n-type networks is important to eliminate the one-hot critical race. It must be ensured that the current, previous state pair changes are in the order 10→11→01, *i.e.* the next state must be entered before the current one is left. In the CMOS implementation, the next state is entered by its n-types, whereas the current state is left by its p-types. For correct operation, the n-type resolving transistor of the following state must be ON long enough for the following state to be entered. This n-type resolving transistor will be turned from ON to OFF by the p-types of the previous state, as the next state's variable begins to rise. For the next state to be properly entered, it must stay ON long enough for the state gate to switch. It therefore follows that the p-type pull-up must be slower than the n-type pull-down. This can be ensured by the transistor sizing by making the p-types of smaller $\frac{W}{L}$ ratio compared to the n-types. In practice however, as p-types are inherently slower that n-types (about four times for the processes that were used in this research), there is no need for the pull-up p-types to have different sizes from the n-types. For the case of more than two states, with their transition requirements simultaneously fulfilled, correct operation will occur if the delays of state gates and their interconnections are relatively uniform.

In the following chapters, all of the control circuits are implemented using the direct-mapped approach. Typical values used in these implementations were $\frac{2}{1}$ for the strong inverter and $\frac{1}{1}$ for the weak one. For the n-type chains an equivalent $\frac{W}{L}$ of $\frac{2}{1}$ and for the p-types $\frac{8}{1}$ were used.

[3]The equivalent size is calculated in the same way as the equivalent resistance for a network of series and parallel resistors.

2.8.3.7 State Output Buffering

The state outputs can be fed directly to another circuit block or to additional output logic that produces the output signals. Care must be taken when loading the outputs of state gates without any additional buffering.

Excessive loading of a state output, by long routing or by feeding it to multiple inputs, may unbalance the weak/strong relationship of the back-to-back inverters, making it impossible for the output to change state. To solve this problem, two extra buffering inverters can be added at the state outputs.

2.8.3.8 AFSM Initialisation

Typically, the AFSM should be initialised by a reset signal into a single initial state, so one state resets high and all others low. This can be achieved by extra reset transistors connected in parallel with the n and p-chains and which drive the state gate during the circuit initialisation. A state that resets high requires a reset n-type, whereas states that reset 0, reset p-types. The size of these reset transistors should be large enough to force a state change like that of the n and p-type chains.

2.8.3.9 Persistent States

So far it has been assumed that an AFSM's state is left when its immediate sucessor is entered. It is possible to relax this condition and create persistent states, *i.e.* states which are left by a state other than their immediate sucessor. Persistent states are possible due to the one-hot encoding and in a machine with persistent states, multiple states may be active.

The use of persistent states often removes the need for extra circuitry. In this work, persistent states were found useful in two cases: in the design of an arbitration circuit for remembering the requestor (*c.f.* Section 4.4.1) and in the design of the A1 processor's μcontrollers where it was often necessary to hold handshake signals high while performing another handshake (*c.f.* Section 5.4.2).

2.8.3.10 Hazards in Direct-Mapped AFSMs

Direct-Mapped AFSMs will not have any combinational hazards at the state outputs because of their design. As long as the machine is stable in some state, no combinational hazards are possible, no matter what the input changes. The active state output is held high by the state-storage structure, so even if some of the inputs now return low, as a response to the assertion of that state, the output will remain stable. In addition, only if another state is entered can that output go low.

In this case, where there is a state change, there is the possibility of an essential hazard, depending on the flow-table structure that the machine implements. The essential hazard can lead the machine to the wrong final state, and can occur if arbitrary delays are assigned to the circuit components.

This can be illustrated by contrasting the conventional AFSM implementation of a 2-bit grey-code counter, which was discussed in section 2.4.3, with its direct-mapped CMOS implementation. The flow table specification is shown again in Figure 2.24.

	x	
	0	1
1	(1), 00	2, 01
2	3, 11	(2), 01
3	(3), 11	4, 10
4	1, 00	(4), 10

Figure 2.24: Flow Table for a 2-bit grey-code Counter

The design and realisation of the asynchronous FSM circuit that implements this specification was shown in Figure 2.9, and its hazard-free version in Figure 2.10. That circuit can be realised using direct-mapped AFSMs by the circuit shown in Figure 2.25.

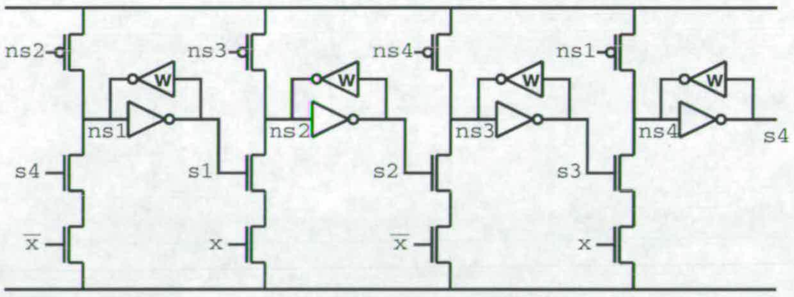


Figure 2.25: Four State Counter

The difference between the two circuits is that the direct-mapped circuit does not generate 2-bit grey-code but only 1-bit state signals. This current version of the circuit requires 30 transistors. If grey-code outputs are to be generated, the extra logic requires 2 OR gates. The grey-code output equations would in that case be `y0 = s1 + s2` and `y1 = s3 + s2`.

The flow-table specification contains essential hazards for all stable states. For example, with the machine originally in state 1,0, if x changes to 1, it is possible to end up in state 3, if the state change is perceived before the input change. So, in the conventional AFSM circuit, to avoid this hazard, it was necessary to include a feedback delay at the state outputs, to ensure that the state change is perceived after the input change. The direct-mapped AFSM exhibits the same essential hazard. With the machine in state s1, if `x` is asserted, then the machine will enter state s2, but if the delay in generating the inverse of `x` that feeds into the s3 state gate is very long, it can happen that the machine enters s3 because the inverse of `x` is still high, *i.e.* the state change is perceived before the input change. So, the condition for eliminating the possibility of the essential hazard is that the delay of the state gates, δ_{sg} must be greater that that of the inverter that produces $\overline{\mathrm{x}}$, $\delta_{inverter-x}$, *i.e.* $\delta_{sg} > \delta_{inverter-x}$. This can be ensured at the physical level. D-trio hazards are possible in the same way.

The logic that generates the outputs, such as `y0` and `y1` in this example, may indeed contain transient output hazards, depending on the delays between the state outputs and the output logic. It is therefore preferable that the use of output logic is minimised and that all the circuit outputs are state machine outputs which handshake with the FSM states.

For this example, the total number of transistors including the extra OR gates is 40 for the direct-mapped AFSM. Contrasting this to the conventional AFSM transistor count of 34 for the minimum gate circuit and 50 for the hazard free version, it can be concluded that the direct-mapped method is better for this example, even though it uses more state signals.

2.8.4 Comparison with other implementation techniques

In this section, the direct-mapped CMOS implementation of a few example circuits is contrasted with their implementation with different circuit design approaches.

2.8.4.1 Latch Control Circuits

The first example is the implementation of four-phase, level-triggered latch control circuits. A latch controller is the control part of an asynchronous pipeline stage. In its simplest form, a latch control circuit must store input data into a latch and pass them to the output, *i.e.* the next pipeline stage.

The control signal sequencing must be such that it is possible for all pipeline stages to be filled. This implies latching the input data before checking that the output stage is busy. In addition, if the completion of the input handshake depends on the initiation of the output handshake, then the latch controller is said to be semi-decoupled, whereas if the input and output handshakes are entirely independent, then it is said to be fully-decoupled.

Figure 2.26 shows the implementation of a semi-decoupled latch controller [FD96]. This implementation uses asymmetric C-gates and has been produced using the Signal Transition Graph (STG) approach [Chu87]. The structure of an asymmetric C-gate is shown in Figure 2.27.

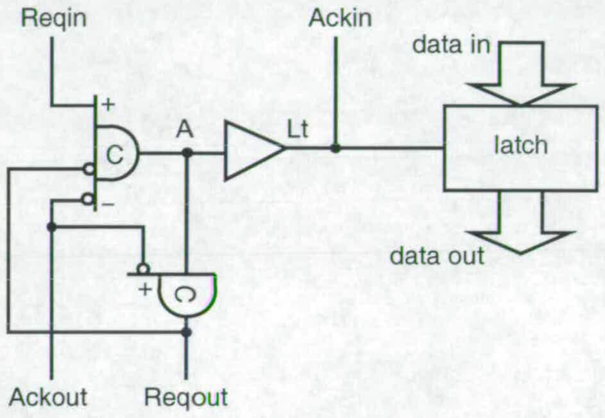


Figure 2.26: Semi-decoupled latch controller using asymmetric C-gates

The asymmetric C-gate is like a generalised version of a C-Muller gate, where an input may control the rising, the falling or both edges of the output depending on whether is it connected to the extension marked '+', '-' or to the main body of the gate.

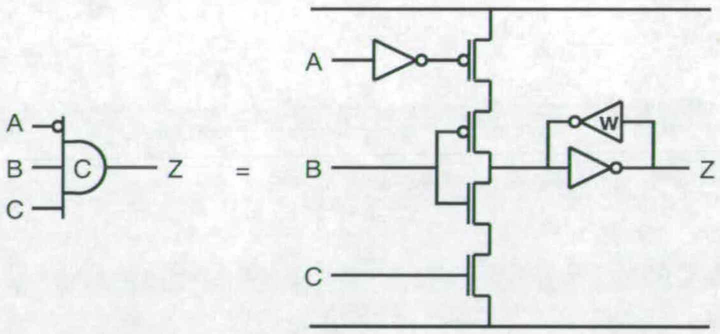


Figure 2.27: Structure of an asymmetric C-gate

This implementation of the semi-decoupled latch controller requires two asymmetric C-gates and a buffer, represented by the triangle in the diagram, for driving the latch enable bits. The total number of transistors required to implementing the control part, *i.e.* the two asymmetric C-gates is 20.

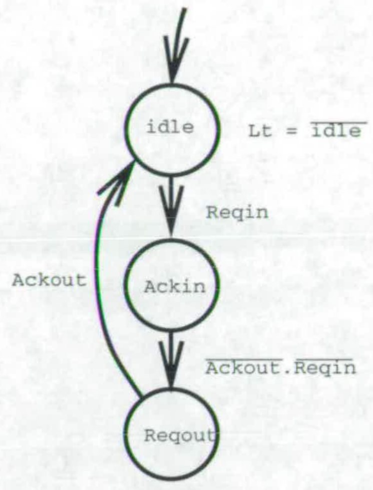

idle
Lt = $\overline{\text{idle}}$
Reqin
Ackout
Ackin
$\overline{\text{Ackout.Reqin}}$
Reqout

Figure 2.28: State graph for semi-decoupled latch controller

The state graph shown in Figure 2.28 can be used to implement the semi-decoupled latch controller as a CMOS direct-mapped AFSM. The AFSM circuit corresponding to this state graph is shown in Figure 2.29. The total number of transistors required to implement the latch control in this way is 26.

Hence, for this particular example, the direct-mapped approach requires more transistors than the STG approach in order to implement the same circuit.

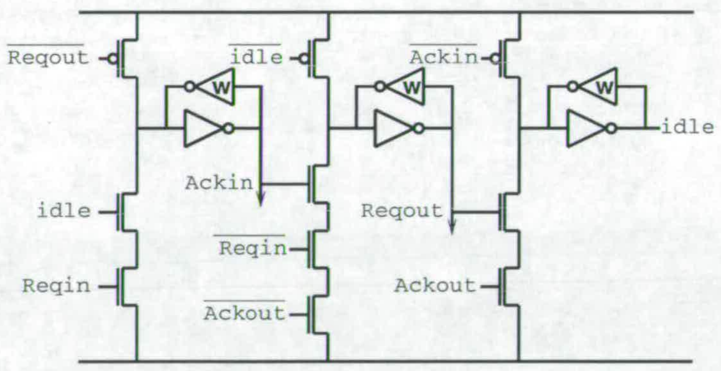

$\overline{\text{Reqout}}$
$\overline{\text{idle}}$
$\overline{\text{Ackin}}$
W
W
W
idle
Ackin
idle
Reqout
$\overline{\text{Reqin}}$
Reqin
Ackout
$\overline{\text{Ackout}}$

Figure 2.29: Semi-decoupled latch controller using DM-AFSMs

None of these two circuits fully-decouple the input/output handshakes. The problem with the asymmetric C-gate circuit is that the `Ackin` signal cannot return low until `Ackout` has been asserted. Hence, the previous pipeline stage in this case would wait longer than necessary incurring a potential performance loss.

In the direct-mapped version, the situation is similar. Here, signal `Reqout` cannot be asserted until `Reqin` has been acknowledged, *i.e.* `Ackin` has been asserted, and `Reqin` has returned low. Hence, in this case, the output handshake is delayed longer than necessary.

Figure 2.30 shows the asymmetric C-gate implementation of a fully-decoupled latch controller [FD96].

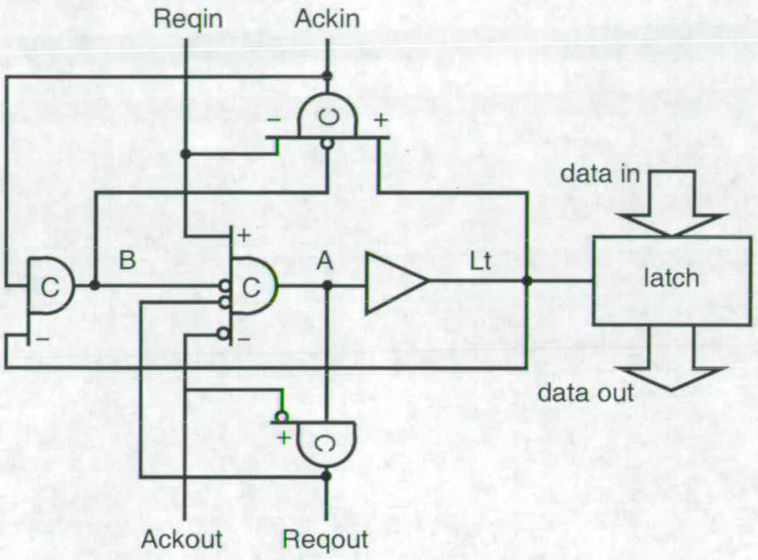


Figure 2.30: Fully-decoupled latch controller using asymmetric C-gates

This circuit fully-decouples the input and output handshakes. In this circuit, as soon as the data has been latched, *i.e.* `Lt` has been asserted, the handshake on the input side, `Ackin`, will return to zero. The transistor count for the control part of this circuit is 42.

The fully-decoupled latch controller cannot be implemented by a single AFSM. This is because implementing this circuit as a single AFSM implies imposing an order on the sequencing of the handshakes and that order implies that they are not decoupled. To implement the fully-decoupled latch controller as an AFSM, two AFSMs, or one with multiple parallel paths is required.

Figure 2.31 shows an AFSM state graph that contains two parallel paths, *i.e.* effectively two AFSMs, in order to fully decouple the input and output handshakes.

The circles in the AFSM represent the points in the state graph where the flow is parallelised or sequentialised. When `Reqin` is asserted, the flow is parallelised and both states `Ackin` and `Reqout` are entered. The circle at the bottom joins the two flows and brings them into state `idle`.

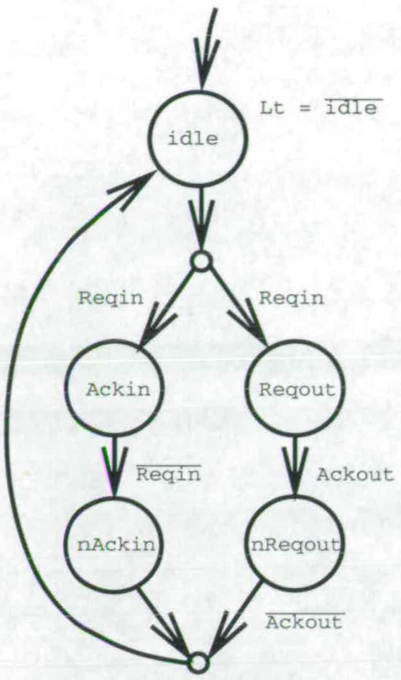

Lt = $\overline{\text{idle}}$
idle
Reqin
Reqin
Ackin
Reqout
$\overline{\text{Reqin}}$
Ackout
nAckin
nReqout
$\overline{\text{Ackout}}$

Figure 2.31: State graph for fully-decoupled latch controller

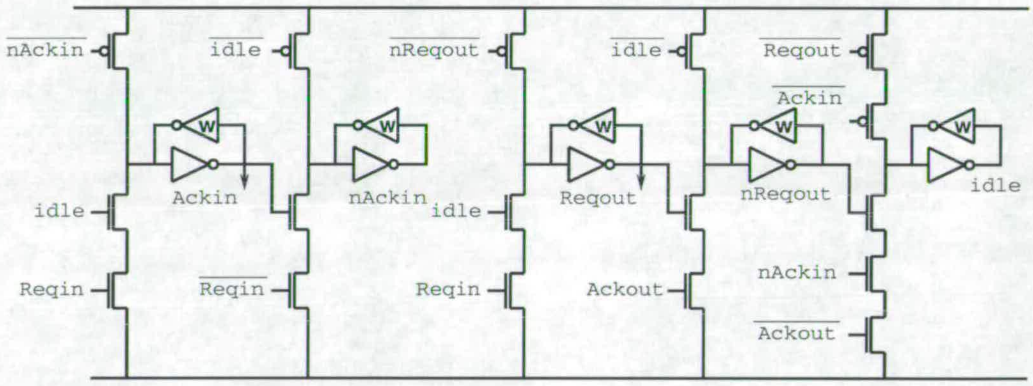

$\overline{\text{nAckin}}$
$\overline{\text{idle}}$
$\overline{\text{nReqout}}$
$\overline{\text{idle}}$
$\overline{\text{Reqout}}$
$\overline{\text{Ackin}}$
w
Ackin
nAckin
Reqout
nReqout
idle
idle
idle
Reqin
$\overline{\text{Reqin}}$
Reqin
Ackout
nAckin
$\overline{\text{Ackout}}$

Figure 2.32: Fully-decoupled latch controller using DM-AFSMs

Figure 2.32 shows the AFSM implementation of the state graph. The number of transistors required for this circuit is 41. Hence, for this example, the number of transistors required by both an asymmetric C-gate realisation and a direct-mapped AFSM realisation are almost identical.

The next section compares the implementation of an example burst-mode AFSM to a direct-mapped AFSM.

2.8.4.2 Comparison with a Burst-mode FSM example

The second example is the implementation of a control circuit using the burst-mode design style and the direct-mapped AFSM approach. Figure 2.33 shows a burst-mode state graph of an AFSM. It is taken from [DCS93]. This example is one of the control circuits of a multicomputer communication chip called the Post Office.

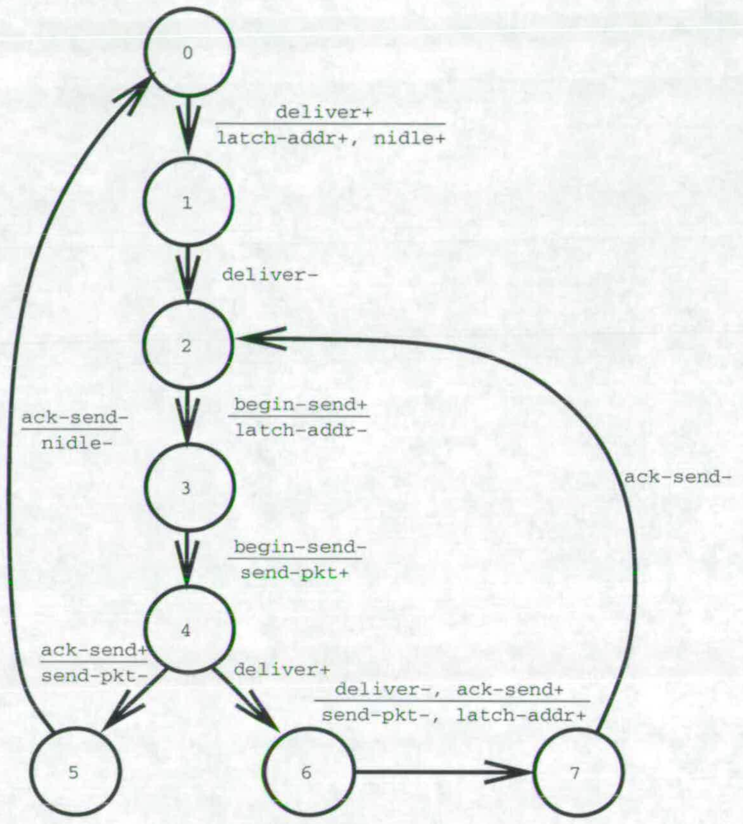


Figure 2.33: Sbuf-send-ctl Burst-mode AFSM (Davis et. al)

This circuit has three inputs, *i.e.* `deliver`, `begin-send` and `ack-send`, and three outputs, *i.e.* `latch-addr`, `idle` (which is active low) and `send-pkt`.

When minimised by the MEAT tool, this specification results in an sum-of-products implementation with two state variables. The complex CMOS gates that implement the two state variables, `Y0` and `Y1` are shown in Figures 2.34 and 2.35 respectively.

The total number of transistors required to implement these two gates is 22 transistors. To implement the complete circuit 22 additional transistors are required in order to generate the three outputs. This brings the total circuit size to 44 transistors.

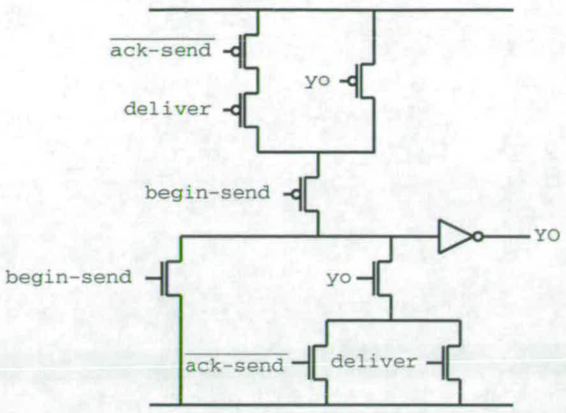
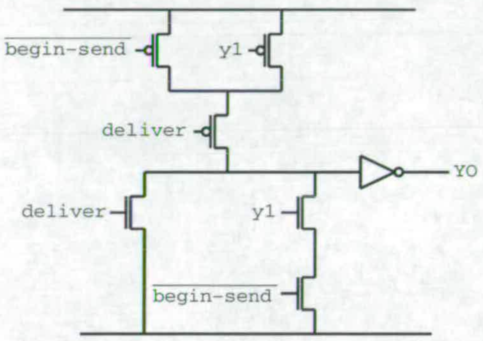


Figure 2.34: Sum-of-Products realisation for output Y0 of Sbuf-send-ctl AFSM



$\overline{\text{begin-send}}$
y1
deliver
YO
deliver
y1
$\overline{\text{begin-send}}$

Figure 2.35: Sum-of-Products realisation for output Y1 of Sbuf-send-ctl AFSM

The minimised state graph has fewer states than the burst-mode state graph. This is because some of the states of the latter can be merged. For example, states 1 and 2 have been merged into state `wait`. States 5 and 7 have been removed by adding signal `ack-send` in the input conditions of state `wait`. The total number of transistors required to implement the specification as a direct-mapped AFSM is 48.

In this example too the difference in transistor sizes between the burst-mode complex gate realisation and the direct-mapped approach is quite small.

From studying the implementation of these example circuits it can be concluded that it is not straightforward that the direct-mapped AFSM approach incurs a size penalty, it is rather circuit-dependent. Due to the one-hot encoding of direct-mapped AFSMs, it is likely that circuits with a large number of states will require more transistors if implemented using the direct-mapped approach than if implemented using a different encoding method and the conventional sum-of-products implementation. However, most practical asynchronous control circuits are relatively small in size, as asynchronous systems are highly modular.

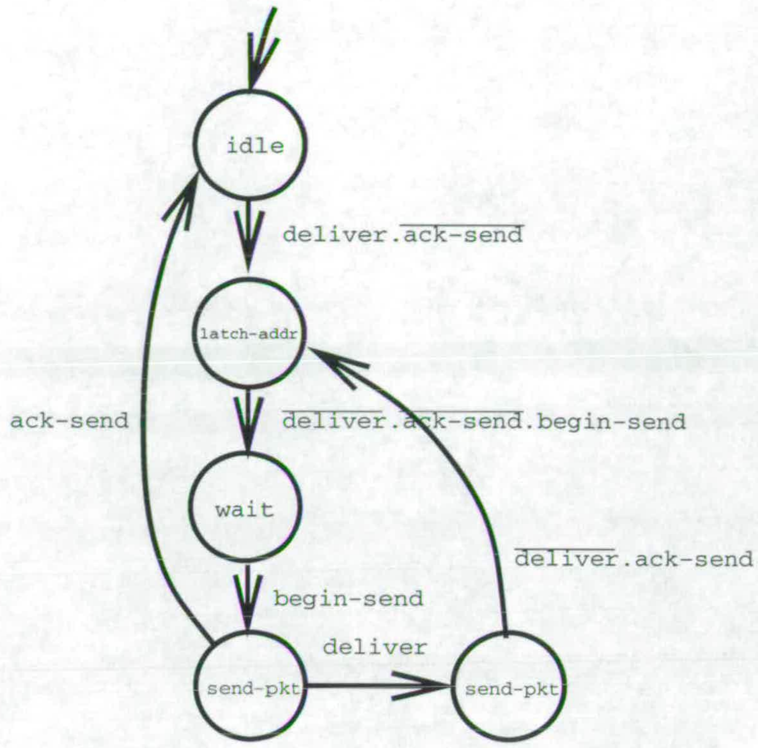

idle
deliver.$\overline{\text{ack-send}}$
latch-addr
ack-send
$\overline{\text{deliver}}$.$\overline{\text{ack-send}}$.begin-send
wait
$\overline{\text{deliver}}$.ack-send
begin-send
deliver
send-pkt
send-pkt

Figure 2.36: Sbuf-send-ctl minimised AFSM state graph

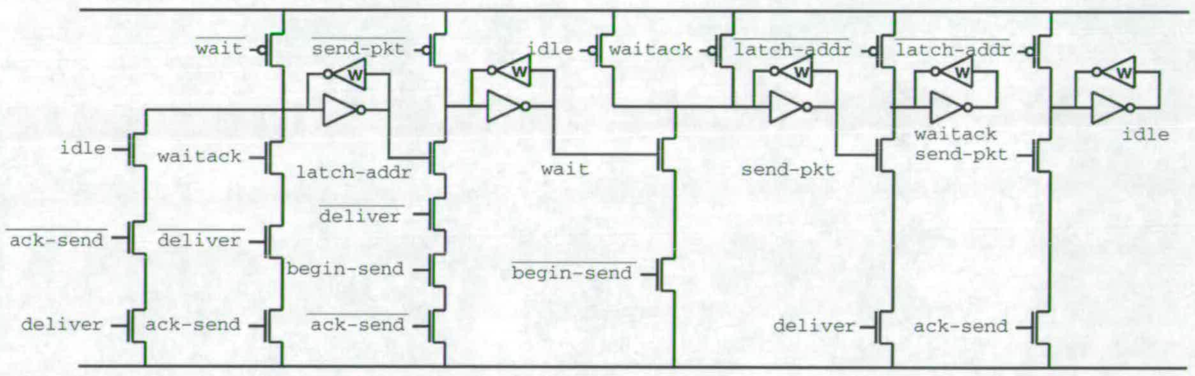

$\overline{\text{wait}}$
$\overline{\text{send-pkt}}$
idle
waitack
$\overline{\text{latch-addr}}$
$\overline{\text{latch-addr}}$
w
idle
waitack
latch-addr
wait
send-pkt
waitack
send-pkt
idle
$\overline{\text{deliver}}$
$\overline{\text{ack-send}}$
$\overline{\text{deliver}}$
begin-send
$\overline{\text{begin-send}}$
deliver
ack-send
$\overline{\text{ack-send}}$
deliver
ack-send

Figure 2.37: Sbuf-send-ctl minimised AFSM state graph

In addition, the direct-mapped approach is simpler to implement and produces faster circuits.

2.8.5 Automating the CMOS Direct-Mapped Approach

Due to the regular nature of the CMOS Direct-Mapped implementation, it is relatively straightforward to automate the design process and to produce, from an AFSM specification, a transistor-level circuit description.

A simple synthesis program has been written in the C programming language to demonstrate this. The specification of the circuit which is to be implemented is given in terms of the circuit states, the transitions between these states and the circuit inputs that cause these transitions. The program produces a circuit description in CDL/Spice file format. The CDL/Spice file can then be simulated using HSPICE or imported into the Cadence tool set. In this way, the circuit produced by the synthesis tool can be further developed, *i.e.* its layout can be automatically generated, or saved in a library and used in a circuit design.

As an example of the circuit synthesis process, consider the following AFSM specification; there are three states `s0`, `s1` and `s2`, where `s0` is the initial one, two primary inputs, l and r and the following transitions: from state `s0` to state `s1` (`s0`$\rightarrow$`s1`) on input `lnr` (where `lnr` $= l\overline{r}$), `s1`$\rightarrow$`s0` on `nlnr` (where `nlnr` $= \overline{l}\overline{r}$), `s1`$\rightarrow$`s2` on `lr` (where `lr` $= lr$) and `s2`$\rightarrow$`s0` on `nlnr`. This AFSM specification detects, whenever state `s2` is entered, that the inputs l and r performed a high transition in sequence, *i.e.* l is asserted and then r. The need to recognise the order of transitions is quite common in mechanical systems, for determining the direction of motion or measuring displacement. From the specification, the synthesis tool then produces the CDL/Spice circuit description shown in Figure 2.38.

At the top of the CDL/Spice file, the global supply signals, `vdd`, `gnd`, and the AFSM reset signal, `rst` are defined. Then, the strong and weak inverters are described as subcircuits, as they are used to form the state gates. The strong inverter is three times wider in this example.

Then, each state of the AFSM is defined as a subcircuit, the n-type transistors which form the n-type pull-down networks, then the p-types, which form the pull-up networks and finally the two back-to-back inverters. All transistors are of equal width.

At the bottom of the file, the state gate subcircuits are joined together to form the complete AFSM, which is defined as the `auto-fsm` subcircuit. The reset transistors are also added at this stage.

2.9 Conclusions

In this chapter, the CMOS direct-mapped AFSM approach to control circuit design was presented. It produces regular, fast, asynchronous control circuits without the need to analyse the flow-table specification to derive a state variable assignment. It allows for MIC, non-fundamental mode asynchronous operation. The complex state gates are free of combinational hazards. Sequential hazards can

```
*** This is an automatically generated CDL/Spice file ***

.global vdd:p gnd:g rst:p
.pins vdd gnd

.subckt inverter-strong out / in
Mp1 out in vdd vdd p w=2.4u l=0.8u
Mn1 out in gnd gnd n w=2.4u l=0.8u
.ends inverter

.subckt inverter-weak out / in
Mp3 out in vdd vdd p w=0.8u l=0.8u
Mn3 out in gnd gnd n w=0.8u l=0.8u
.ends inverter

.subckt state_storage out / in
Xi5 out in /inverter-strong
Xi4 in out /inverter-weak
.ends state_storage

.subckt state-0 ns0 s0 / nlnr ns1 s1 ns2 s2
Mn6 ns0 nlnr node0 gnd n l=0.8 w=2.4
Mn7 node0 s1 gnd gnd n l=0.8 w=2.4
Mn8 ns0 nlnr node1 gnd n l=0.8 w=2.4
Mn9 node1 s2 gnd gnd n l=0.8 w=2.4
Mp10 node2 ns1 vdd vdd p l=0.8 w=2.4
Mp11 ns0 nlnr node2 vdd p l=0.8 w=2.4
Xss12 s0 ns0 /state_storage
.ends state-0

.subckt state-1 ns1 s1 / lnr ns0 s0
Mn13 ns1 lnr node3 gnd n l=0.8 w=2.4
Mn14 node3 s0 gnd gnd n l=0.8 w=2.4
Mp15 node4 ns0 vdd vdd p l=0.8 w=2.4
Mp16 ns1 lnr node4 vdd p l=0.8 w=2.4
Mp17 ns1 ns2 vdd vdd p l=0.8 w=1.6
Xss18 s1 ns1 /state_storage
.ends state-1

.subckt state-2 ns2 s2 / lr ns1 s1
Mn19 ns2 lr node5 gnd n l=0.8 w=2.4
Mn20 node5 s1 gnd gnd n l=0.8 w=2.4
Mp21 ns2 ns0 vdd vdd p l=0.8 w=1.6
Xss22 s2 ns2 /state_storage
.ends state-2

.subckt auto-fsm ns0 s0 ns1 s1 ns2 s2 / nlnr lnr lr
Xsi23 ns0 s0 nlnr ns1 s1 ns2 s2 /state-0
Xi24 node6 rst /inverter-strong
Mn25 ns0 node6 gnd gnd n l=0.8 w=3.2
Xsi26 ns1 s1 lnr ns0 s0 /state-1
Mp27 ns1 rst vdd vdd p l=0.8 w=3.2
Xsi28 ns2 s2 lr ns1 s1 /state-2
Mp29 ns2 rst vdd vdd p l=0.8 w=3.2
.ends auto-fsm
```

Figure 2.38: CMOS Direct-Mapped AFSM Synthesis Tool CDL/Spice Output

be present, depending on the flow-table, and must be handled by ensuring that hazard-free delay assumptions are fulfilled at the physical level. Transient hazards are also possible if output logic is used. Direct-mapped AFSMs will generally require more state variable signals, although that does not necessarily imply that their circuit size is larger than an AFSM with a different state encoding. At the physical level, fewer feedback signals are required, as connectivity between the state gates is local. The CMOS direct-mapped approach can easily be automated. From a state machine diagram, a circuit netlist can be derived.

Chapter 3

Asynchronous Processor Design

In this chapter, the fundamentals of asynchronous systems and asynchronous processors are reviewed. Two hardware mechanisms for exploiting concurrency are presented; shared register files and μnet architectures. Shared register files have explicitly defined common regions and allow for communication and synchronisation to take place through the shared registers. μnet architectures break down instructions into μinstructions, which they attempt to issue and execute concurrently in the processor datapath.

3.1 Fundamentals of Asynchronous Systems

An asynchronous system is a composition of interconnected asynchronous units. These units can themselves be asynchronous systems, or monolithic asynchronous circuits. Due to the fact that these asynchronous units do not rely on specific timing assumptions, they are autonomous, *i.e.* the interface they provide to the system is independent of their functionality and implementation. This characteristic makes asynchronous systems compositional, scalable and flexible. An asynchronous unit in a system can easily be modified by swapping one part for another without worrying about timing or interfacing issues.

3.1.1 Communication and Synchronisation

Communication between units in an asynchronous system is typically achieved via a two-phase or four-phase communication protocol (handshake), Figure 3.1.

The two-phase version is transition based and works as follows. The sender initiates the communication operation by asserting a request signal. At that time any data relative to this communication should be available. The data travels as a bundle relative to the request signal. The receiver will then accept the data and, when it has done so, it will assert an acknowledgement signal. This signals

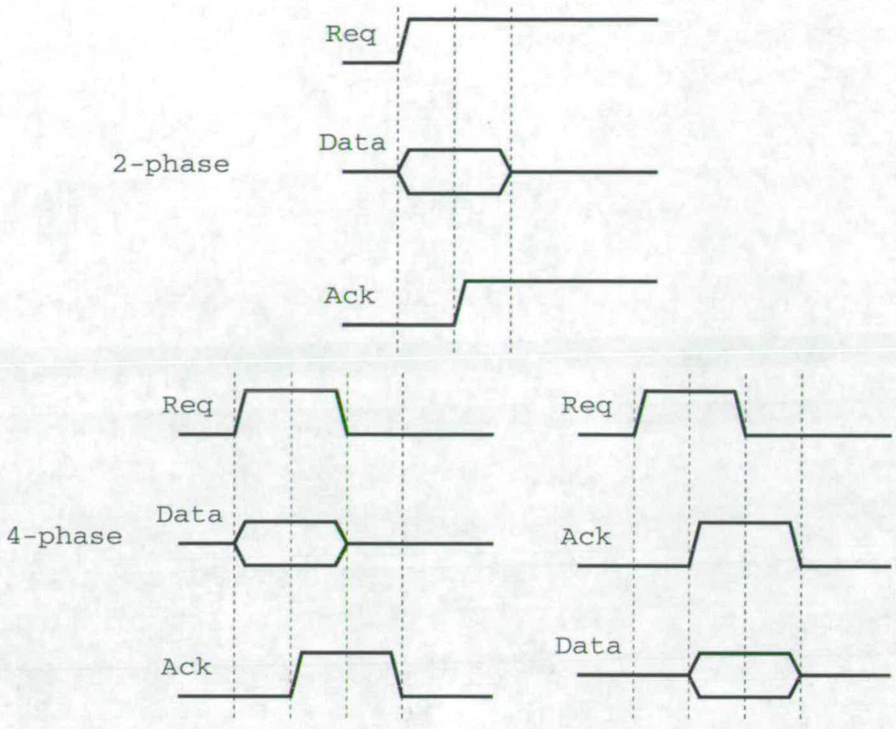


Figure 3.1: 2 and 4-phase Handshaking Protocols

the end of the communication. At the end of a two-phase handshake, both the request and the acknowledge signals are asserted. The next communication action will deassert the request signal and so on.

The four-phase handshake is level based. It works in the same way as the two-phase, only both signals must return to zero at the end of the communication. The bundled data may be attached to the request or acknowledge signals as the bottom part of Figure 3.1 shows. In the 4-phase handshake on the left, the data is attached to the request signal, whereas in the 4-phase handshake on the right, the data is attached to the acknowledge signal. After the acknowledgement signal has been raised, the request must return to zero and then the acknowledgement signal will also return to zero. It is not allowable to raise the request signal when the acknowledge is still high.

3.1.2 Completion Detection

As there is no constant timing reference, it is necessary in asynchronous circuits to detect the completion of an operation. The ability to do this is of great importance as it can greatly increase performance by making the latency of an operation sensitive to its input data rather than being fixed on the basis of its critical path. For many operations, the critical path or the worst-case delay path can be considerably greater than the average case delay. There are two ways of achieving this, bounded or relative delay and transition detection.

The bounded or relative delay approach relates the latency for the completion of an operation to a reference delay, which must be guaranteed to be slightly greater. The reference delay is usually the delay of another signal. The assertion or deassertion of the reference signal will signal the completion of the operation, for this reason its implementation should be clear of any hazards. It should be asserted and deasserted monotonically. The bounded delay approach is particularly suitable for implementing bundled datapath operations, where all bits have approximately the same latency. An extra bit can then be used, the delay of which is guaranteed at the physical level to be an upper bound to the delays of the others. This bit is usually initialised high or low at the initiation of the operation and when it changes state that signals completion. Implementing the reference signal logic requires implementing an extra delay path of similar delay. That can be provided by simply copying the logic that produces the output. This approach requires only a slight area increase. In practice, the circuit operation is not necessarily at the worst-case speed, because the delay of the reference signal need not be constant.

The transition detection approach detects transitions on the outputs of the circuit that performs the operation. For some operations, an output transition may not be the final one, so care must be taken when using this approach. Transition detection approaches must detect both 0→1 and 1→0 transitions and also remember the initial state of the output signal. The most common way of implementing transition detection is dual-rail encoding. This uses two signals rather than a single one to represent an output value. It represents a logical 0 by a 01, a logical 1 by a 10 and the remaining values 00 and 11 are invalid and used for initialising the circuit. When a circuit input is one of the invalid values then the output will also be invalid, so dual-rail coding also acts as a synchronisation mechanism, as the output is waiting for a valid input.

The detection of the completion of an operation is implemented by exclusive OR-ing the dual-rail coded output. If more than one output is present, the completion signals must be ANDed. When dual-rail coding is used, the logic that implements the operation has to be modified for dual-rail coded inputs and dual-rail coded outputs. It may seem that dual-rail coding is expensive as it requires doubling the number of signals involved and therefore potentially doubling the logic. But, dual-rail coding does not have to be used for all the datapath input and output signals.

In principle, dual-rail coding can be used as a synchronisation mechanism, replacing a handshake, although a handshaking protocol is more commonly used, even with dual-rail encoding, as it is in general simpler to implement and requires less signals.

Completion detection is associated with a special type of hazard, the delay hazard. The delay hazard manifests itself at the output of the combinational circuit that produces the completion signal. This occurs in the case when multiple gates of varying delays are used to produce the completion signal, as a sum-of-products circuit, and multiple gates turn on but some switch faster than others. A quick response from the environment may produce a static 1-hazard if the gate that was turned on is now turned off because of an input change and the other gate(s) have still not yet turned on.

Implementing the completion detection circuitry as a dynamic, precharged gate, rather than in a sum-of-products circuit eliminates the delay hazard. During the inactive circuit period, the completion detection gate output is precharged low, for example, by a pull-down transistor. Then, a number of pull-up transistor chains, which implement the completion detection mechanism and depend on the nature of the operation, can then assert it, while the circuit is active. In this way, the $0 \rightarrow 1$ transition is monotonic.

3.1.3 Arbitration

In certain cases it is desirable to establish a many-to-one relationship between units in an asynchronous system. This implies a one-to-many connectivity between a number of sources and a single sink. As any of these sources should be able to communicate with the sink, and multiple sources may be active at any one time, additional circuitry is required to select, in the case of multiple active sources, a single one. This circuit is called an arbiter and the process is called arbitration.

An n-way arbiter arbitrates between n sources, *i.e.* n request/acknowledge pairs into a single sink, a single request/acknowledge pair. The implementation of an arbiter is based on the mutual exclusion element, a 2-way version of which is shown in Figure 3.2.

The mutual exclusion element consists of two parts, the digital part that actually performs the mutual exclusion action by a set-reset flip-flop, and the analogue part, a metastability filter for the case when multiple requests occur simultaneously and the flip-flip becomes unstable.

The disadvantage of this type of arbiter is that it does not guarantee fairness,

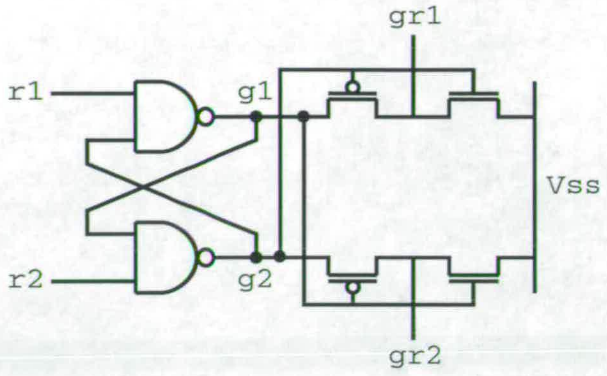


Figure 3.2: Mutual Exclusion Element

i.e. the result of the arbitration can always be the same on a clash. In addition, in the case where the flip-flop becomes unstable, there is no upper bound to the time it takes for it to become stabilised [KW76], as this depends on physical circuit delays and random processes such as noise.

3.2 Asynchronous Processor Design

This section reviews the most common processor design technique, *i.e.* pipelining and contrasts synchronous and asynchronous pipelines.

3.2.1 Pipelining

A pipeline, Figure 3.3, is a collection of processing stages that perform a function over a stream of data.

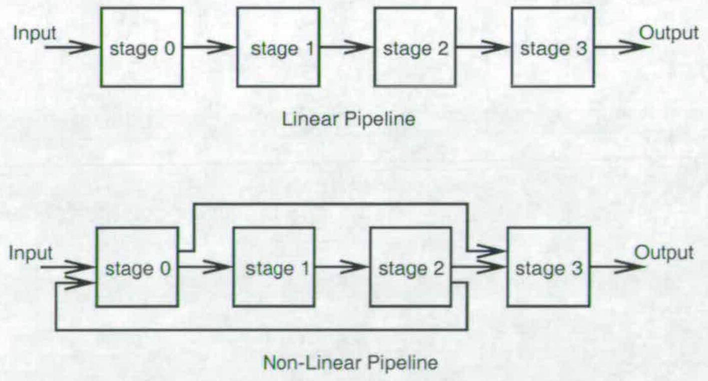


Figure 3.3: Linear and Non-Linear Pipelines

In a linear pipeline, the processing stages are linearly connected. A non-linear or dynamic pipeline is one which contains additional feedforward and feedback connections. A linear pipeline always performs a fixed function, whereas a dynamic one can be reconfigured, by making use of its feedforward and feedback connections, to perform variable functions. A pipeline has typically a single input, where data can be inserted, and potentially multiple outputs, where data may be removed. Depending on the implementation of the control flow, a pipeline may be synchronous or asynchronous.

3.2.2 Synchronous Pipelining

A synchronous pipeline can be implemented as shown in Figure 3.4.

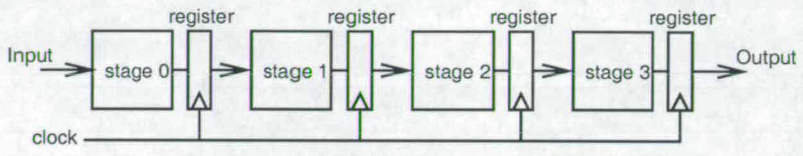


Figure 3.4: Synchronous Pipeline Implementation

Clocked registers are used for interfacing between the pipeline stages. Different clocking strategies may be used, the most common being edge-triggered. When a clock edge arrives, the edge-triggered latches transfer data to the next pipeline stage simultaneously. In a synchronous pipeline, it is desirable to have an approximately equal delay for all the pipeline stages, so that the clock period, and hence the speed of the pipeline, can be determined.

3.2.3 Asynchronous Pipelining

In an asynchronous pipeline, Fig. 3.5, the flow of data between pipeline stages is controlled by an asynchronous communication protocol, commonly a handshake.

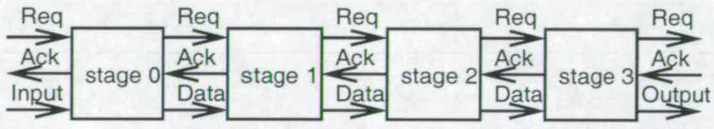


Figure 3.5: Asynchronous Pipeline Implementation

Communication between pipeline stages in an asynchronous pipeline takes place individually between pairs of pipeline stages that are ready to transmit and receive data respectively, rather than being globally controlled. In an asynchronous pipeline, the delay of the pipeline stages may vary. In such a case, the speed of the pipeline is limited by the speed of the slowest stage, but even so, it still performs better than a synchronous one, where all the stages would operate at that slowest speed. In addition, it is often the case that the delays of asynchronous pipeline stages are data-dependent and therefore variable.

A micropipeline [Sut89] is a simple implementation form of an asynchronous pipeline. Fig. 3.6 shows a two-phase micropipeline with logic between the pipeline stages. The delay inserted between the pipeline stages must match the processing delay through the logic.

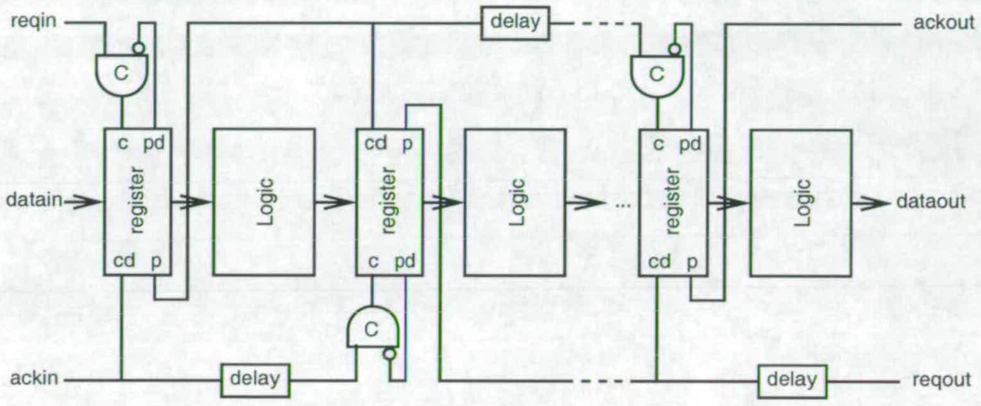


Figure 3.6: Two-Phase Micropipeline Implementation

3.2.4 Instruction Pipelines

An instruction pipeline breaks down the instruction's execution, and in consequence the datapath of an architecture, into distinct stages. In this way, the pipeline stages can be occupied by different instructions, and as the different pipeline stages are operating concurrently, temporal parallelism is exploited. By breaking down the execution of instructions into N stages, the throughput of instructions is, in general, increased by N.

Figure 3.7 shows the structure of a typical instruction pipeline.

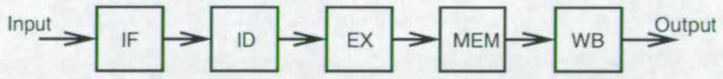


Figure 3.7: A typical Instruction Pipeline

There are five pipeline stages, `IF`, Instruction Fetch, `ID`, Instruction Decode, `EX`, Execute, `MEM`, Memory Access, and `WB`, Write-Back. In this pipeline, when instruction n is in the `WB` stage, instruction $(n-1)$ is in the `MEM` stage, instruction $(n-2)$ is in the `EX` and so on. If multiple similar pipeline stages are provided, for example multiple EX stages, then spatial parallelism can also be exploited. In this case, an instruction in its ID stage does not necessarily have to wait for the previous one to finish its EX stage if another EX stage is available.

3.2.5 Instruction Pipeline Hazards

Certain conditions, called hazards, prevent the normal, continuous pipeline operation, and cause pipeline stages to stall.

There are three classes of hazards; structural hazards, control hazards and data hazards. Structural hazards arise when the architecture cannot accommodate some instruction combinations due to resource conflicts, for example there may be one RF write port but two write requests, and one of them will have to be stalled. Control hazards arise from the pipelining of control transfer instructions. When a conditional control transfer instruction enters the pipeline, its outcome may not be known until it propagates all the way through the pipeline. Therefore, it is not known at the front of the pipeline, from which of the two possible paths instructions should be fetched. The pipeline must stall until the branch outcome is known. Data hazards arise from dependencies between instructions which must be respected for correct program execution.

Hazards reduce the amount of temporal parallelism that can be exploited and therefore the performance gain of pipelining. A pipeline stall is the situation where one or more pipeline stages are not allowed to communicate data but must wait.

3.3 The μnet (micronet) architectural approach

The μnet (or micronet), introduced by Rebello [Reb96], is a mechanism for organising an asynchronous datapath in order to exploit fine-grain temporal and spatial parallelism. It is effectively an extension of instruction pipelining and of asynchronous micropipelines.

Rebello's work concentrated on architectural level simulations of a scalar μnet architecture without being concerned about the specific circuitry required to implement it.

In the following sections, the implementation of the μnet is investigated with the aim of implementing a transistor-level μnet processor.

3.4 μnet Structure

The fundamental characteristic of the μnet approach is that program instructions are broken-down into μoperations, *i.e.* their corresponding basic datapath operations. These can be parallelised, when possible, for one or more instructions being executed in the datapath, hence exploiting fine-grain parallelism. The μoperations are distributed by the μnet control unit, by assigning a pair of handshaking signals to each μoperation, which handshake with the appropriate μblock, implementing that μoperation.

Figures 3.8 and 3.9 show an example μnet architecture. Figure 3.8 shows the control unit connectivity to the datapath components. In this diagram, the solid arrows represent the four-phase μoperation handshakes. Figure 3.9 shows possible connections between datapath components. The solid arrows here represent additional datapath handshakes that carry data between the datapath components.

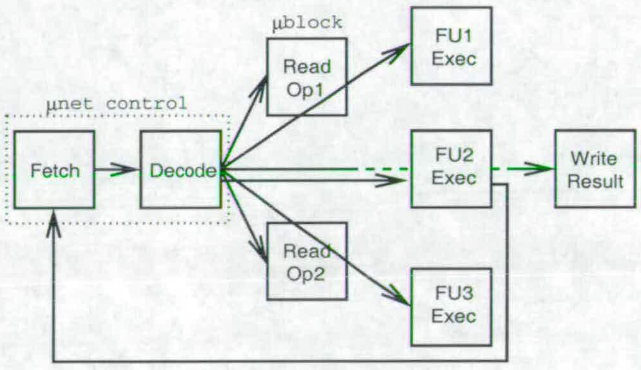


Figure 3.8: Example of a μnet Architecture - μoperation Issue

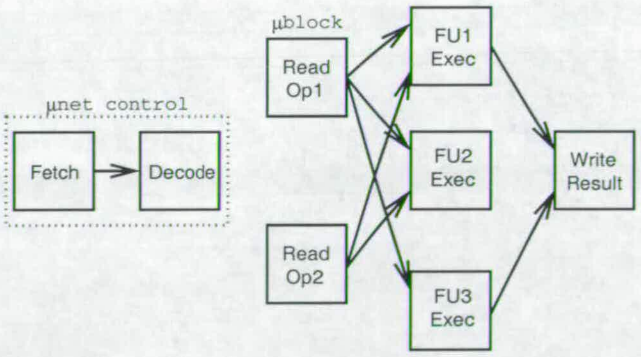


Figure 3.9: Example of a μnet Architecture - Datapath Handshakes

As can be seen in Figure 3.8, the control unit can issue μoperations to all of the μblocks in the architecture. In this way, instructions only use the μblocks that are necessary for their execution, in contrast to a pipeline, where an instruction must always pass through all of the stages. For example, an instruction which uses only one RF port will only use that μblock, an immediate instruction which does not need to read register values or use the FUs will only use the Write Result μblock.

The μnet is effectively an asynchronous, fine-grain, non-linear pipelined structure, where stages are datapath operations, and pipeline inputs are provided at all pipeline stages.

3.5 μoperations

The μoperation signals are equivalent to the datapath control signals in a synchronous processor architecture, which may be generated by a central control unit or by different pipeline stages. The μoperations, much like the datapath control signals, depend on each other. In a synchronous processor, such dependencies are respected by the order of the pipeline stages, and by the order that control signals are asserted at different clock cycles. In a μnet architecture, all μoperations are issued into the datapath simultaneously, and therefore dependencies must be respected by additional control circuitry implemented in the datapath structure.

3.6 μoperation Dependencies

μoperation dependencies may be data or control dependencies. Data dependencies exist between μoperations and their data; for example, a μoperation that adds two numbers cannot execute before its data are available. Control dependencies exist between μoperations that must be executed in a certain order.

The dependencies between μoperations can be represented in the form of a graph. Figure 3.10 shows the μoperation dependencies of the A1 architecture. The longest dependency chain in this graph involves three levels. `Rx` and `Ry` do not depend on any other μoperations, whereas `AOp` and `COp` depend on both `Rx` and `Ry`. `MOp` depends only on `Ry`. `Wz` depends on `AOp` and `MOp` and has a hidden data dependence, drawn with a dashed line, with the immediate value, labelled `Imm`. The hidden dependence exists because, although there is no explicit immediate value handshake, there is a data dependence between the immediate value of the current instruction and its write-back μoperation, `Wz`.

$$\begin{tikzcd}
 & & & \mathtt{Wz} & & & & & \\
 & \mathtt{AOp} \arrow[urr] & & \mathtt{MOp} \arrow[u] & & \mathtt{Imm} \arrow[ull, dashed] & & \mathtt{COp} & \\
\mathtt{Rx} \arrow[ur] & & \mathtt{Ry} \arrow[ul] & \mathtt{Ry} \arrow[u] & & & \mathtt{Rx} \arrow[ur] & & \mathtt{Ry} \arrow[ul]
\end{tikzcd}$$

Figure 3.10: μoperation Dependencies in the A1 Processor

A pipeline can also be drawn in the form of a dependence graph, Figure 3.11. In this graph, the operations are the coarser-grain instruction execution stages; `IF`, Instruction Fetch, `ID`, Instruction Decode, `EX`, Execute, `MEM`, Memory and `WB`, Write-Back, which are identical for all instructions.

$$\begin{tikzcd}
\mathtt{IF} \arrow[r] & \mathtt{ID} \arrow[r] & \mathtt{EX} \arrow[r] & \mathtt{MEM} \arrow[r] & \mathtt{WB}
\end{tikzcd}$$

Figure 3.11: Operation Dependencies in a pipeline

The circuit implementation of a generic μnet, which respects the μoperation data and control dependencies is discussed in the next section.

3.7 Generic μnet Implementation

In order to exploit the maximum possible parallelism for a specified set of μoperations and dependencies between them, each μoperation must be mapped to a μnet stage (or μblock) in the circuit implementation. Each such stage implements a fine-grain datapath operation and requires access to an FU port. A μnet stage resembles a pipeline stage. The difference between them is that a μnet provides inputs at each one of its stages, whereas a pipeline usually provides a single input where instructions enter. Communication between these stages, and the isolation of their data, requires additional control circuitry and data registers.

The number of μoperations, and the dependencies between them, determine the morphology of the μnet datapath, *i.e.* the connectivity between μblocks, and the amount of additional control circuitry and data registers necessary to implement the architecture.

3.7.1 μnet Control Implementation

The purpose of the additional control circuitry is to implement communication between μnet stages, and to store data into the datapath data registers. Often, the control circuits must synchronise handshakes and sequentialise data operations, for example, wait until two request signals have been asserted, or assert a request signal when the data has been stored into a data register. In this way, the μoperation dependencies are respected. The nature of the control can also be extracted from the dependencies graph.

The μoperation handshakes (generated by the control unit) must connect to the μblocks, which perform as much of the operation as possible locally. Then, when data is available or required, they synchronise and communicate with other units. The data dependencies between μoperations require synchronisation and communication to be implemented between them. In general, the initiation of a datapath operation requires synchronisation with the data, and the completion of a datapath operation must be followed by a communication action. Hence, the μblocks that correspond to dependent μoperations will require additional control handshakes to be implemented between them.

For the A1 μoperation dependencies of Figure 3.10, for example, it is necessary to implement a handshake between the `Rx` and `AOp` μoperations. In the A1 processor, this was implemented by using the bus handshake signals. One-to-many dependencies are also present, for example `Rx` may potentially handshake with `AOp` or `COp` and `Ry` with `AOp`, `MOp` or `COp`.

In the case of a multiple dependence, the state of the relevant μoperation handshakes, *i.e.* the global datapath state, is used to determine which pair of μblocks must communicate. For example, if the μoperations `Rx` and `AOp` have their request signals asserted, this implies that they must communicate. The write-back μoperation, `Wz` in the A1 implementation, also uses the state of the relevant μoperation handshakes to resolve the multiple dependencies and determine the data source for the write back. When multiple dependences exist, the μoperation acknowledgement signals should only be asserted after the multiple dependencies have been resolved, *i.e.* their state has been read.

Only in the case when multiple μoperations depend on the same data source, should the multiple μblocks that correspond to them be allowed to request data from the single μblock unit that corresponds to the dependent μoperation. This multi-way forwarding of data may be exploited in the case when register contents need to be fed to multiple FUs of the architecture. In the A1 design this is not possible, as it is not allowed by the instruction decoding and is not supported by the circuit design. Multi-way data forwarding requires multi-way data synchronisation to be implemented, and also requires a more complex instruction decoding process.

Figure 3.12 shows the necessary handshakes that must be implemented for the dependencies that were shown in Figure 3.10.

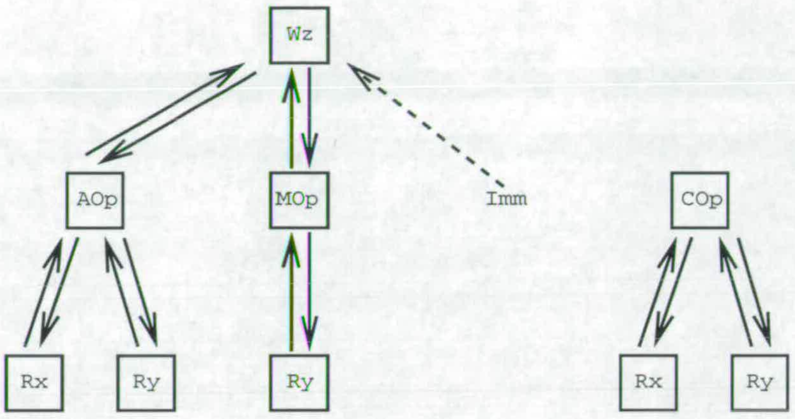


Figure 3.12: μoperation Control Implementation

Each μoperation in Figure 3.10 corresponds to a μblock, *i.e.* a datapath unit in Figure 3.12. Dependent μoperations must handshake with each other. In this diagram multiple instances of the same μblock are shown. In the circuit design though, each μoperation must correspond to a unique μblock.

Figure 3.13 shows a more detailed diagram of the required control, which is closer to the implementation, as each μblock is unique. The horizontal handshakes in the diagram represent the handshakes between the μblock and the control unit (not shown).

In this diagram, it is evident that the dependencies map to necessary communication paths. The two μoperations at the lowest level of the hierarchy, `Rx` and `Ry` that have multiple dependencies must handshake with two and three units and with the control unit respectively.

3.7.2 μnet Data Registers

As handshakes follow multiple paths, so do the data. Each handshake that spawns from a μoperation dependency potentially carries data. In order to isolate the data of different μblocks, data registers are necessary.

The number of registers required depends on the amount of data required per μblock. Figure 3.13 requires a total of 7 data registers, 3 for `AOp` (2 for the operands and one for the result), 2 for `MOp` (one for the operand and one for the

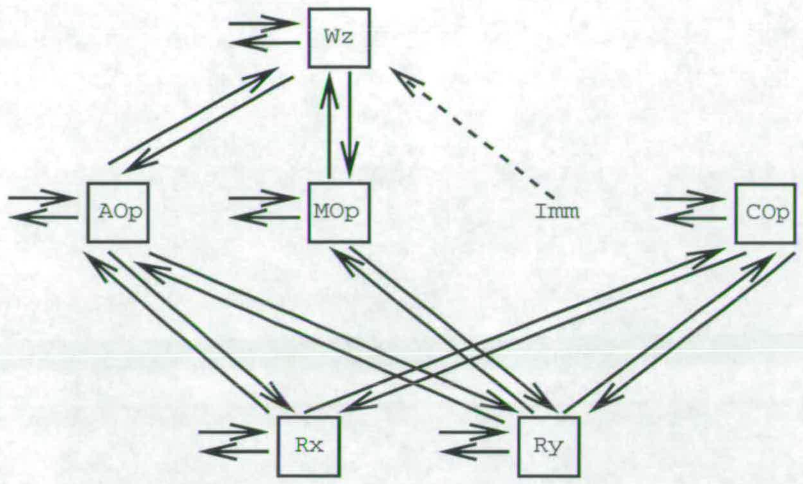


Figure 3.13: μoperation Detailed Control Implementation

result) and 2 for `COp`. This number does not count the extra instruction data registers. There are 3 instruction data registers required, one for the immediate value, and two for the register indices of `Rx` and `Ry`.

In the A1, two of these data registers were dropped to reduce the circuit size, at the cost of parallelism between μblocks. The result of the adder is not stored in a data register and thus the `AOp` μoperation must wait, to hold the data valid, until the data has been written back. Hence, there is no parallelism between the `AOp` and `Wz` μoperations. The memory data register has also been dropped by assuming that the memory holds the data valid.

3.8 Scaling a μnet Datapath

So far, only a scalar μnet datapath has been considered. Except for the register read operations, `Rx` and `Ry`, only single instances of one type occur.

In a scalable μnet architecture, the datapath must have the ability to handle multiple units of the same functionality, for example, multiple FUs, multiple RF read ports, etc. Hence, μoperations and μblocks do not necessarily have a simple one-to-one mapping. The assignment of μoperations to available μblocks must be implemented either in hardware, by the control unit, or in software, by a compiler.

3.8.1 Implementing a Scalable μnet Datapath

Implementing the assignment of μoperations to multiple available μblocks in hardware involves implementing a mechanism for selecting an available unit for a μoperation. For example, in an architecture with two adders, an `AOp` μoperation can be sent to one of the two adders, depending on availability. In addition, if both are available, one must be selected. The control unit must map the `AOp` μoperation to `AOp1` or `AOp2`.

An asynchronous hardware implementation of a circuit that will select units based on their availability is challenging and prone to metastability. The cases where multiple units are available and one of them must be selected and where none are available must be handled. A circuit which will sequentially check the units for availability when an input request arrives is prone to metastability, *i.e.* it may mulfunction under certain timing conditions. Figure 3.14 shows the AFSM of such a circuit.

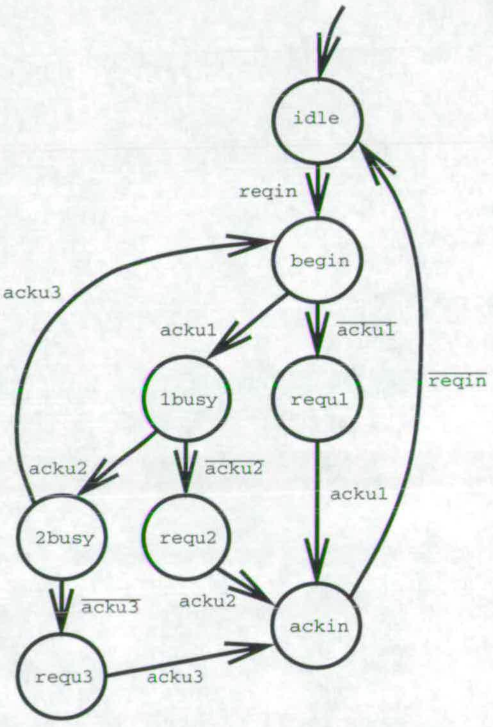

idle
reqin
begin
acku3
acku1
$\overline{\text{acku1}}$
$\overline{\text{reqin}}$
1busy
requ1
acku2
$\overline{\text{acku2}}$
acku1
2busy
requ2
$\overline{\text{acku3}}$
acku2
ackin
requ3
acku3

Figure 3.14: Example FSM of a unit selection circuit

The circuit has an input handshake pair, `reqin` and `ackin`, which represents the incoming request. Three units are available, where the incoming request can be serviced with handshake signals, `requ1`, `acku1`, `requ2`, `acku2` and `requ3` and `acku3`. The AFSM upon receiving an incoming request enters the `begin` state. At this point, the status of the first unit is checked by inspecting the status of its acknowledgement signal.

If `acku1` is deasserted, then this signals that unit 1 is available, thus the machine enters state `requ1`. When unit 1 has received the data and `acku1` is asserted, the incoming request is acknowledged by asserting `ackin`. When `reqin` returns low, another request can be serviced, as the machine enters returns to state `idle`.

If at state `begin` signal `acku1` was asserted, then that would signal that unit 1 is busy. The machine would enter state `1busy` and check unit 2. This process continues until the last unit is checked and then if this is also busy, the machine returns to state `begin` and tries again to find an available unit.

The problem with this approach is the timing of the unit acknowledgement signals, *i.e.* `acku1`, `acku2` and `acku3`. States `begin`, `1busy` and `2busy`, where the status of the acknowledgement signals is checked, have two successors which are entered upon the acknowledgment signal or its inverse. It is possible for the acknowledgement signal to change at such a time as to cause both of these states to be entered and thus the circuit would malfunction.

To implement the assignment of μoperations in software, the compiler must be made aware of the μnet's architectural characteristics, *i.e.* the number and type of μblocks and the dependencies between them. It also implies the use of a different instruction format. As the selection of μblocks is performed by the compiler, the instruction width will have to be increased to include a μblock identification field. A positive consequence of this is that the instruction decoding in the control unit will be greatly simplified. The advantage of a software implementation is that the compiler has a broader-view of the dependencies between instructions and can potentially utilise the datapath more efficiently.

Once μoperations have been mapped to their μblocks, the dependencies between them become fixed. In the scalable μnet, the units which are replicated also replicate the dependencies. For example, if another adder is added to the architecture of Figure 3.13, this will create two dependencies with the RF ports and one with the write-back unit. The extra dependencies imply extra interconnections.

Although it is possible to restrict the connectivity, exploiting as much parallelism as possible requires a scalable μnet to provide full connectivity between all the possible combinations of dependencies. It is also necessary that μblocks of independent chains of operations can communicate simultaneously with no interference. To implement the full-connectivity requirement, the control handshake that synchronises and communicates data between a pair of μblocks, along with their attached data, must be connected from each μblock to all other μblocks with which communication can occur. This is shown in Figure 3.15.

As more dependencies have now been added, by adding multiple units of the

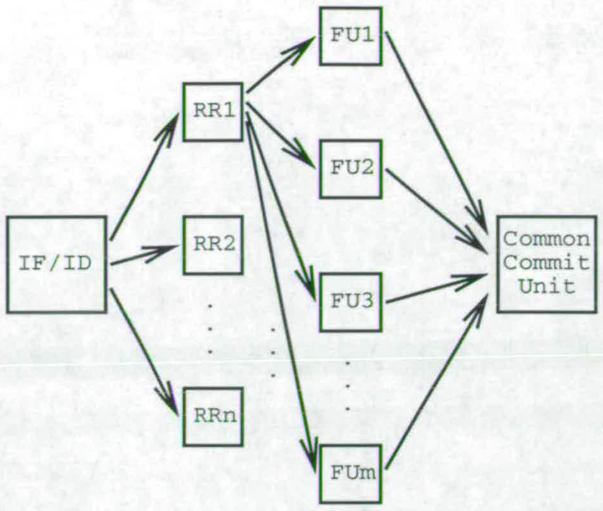


Figure 3.15: Example of a scaled μnet implementation

same type, more dependencies must be resolved. In the scalar μnet, allowing μblocks to inspect the global state, *i.e.* the state of the dependent μoperation handshakes is sufficient to determine which μblocks must communicate. In a superscalar architecture with multiple dependent μblocks, such as multiple read ports and multiple FUs, allowing a μblock to inspect the global state, *i.e.* the state of all the dependent μoperation handshakes, is not sufficient, as multiple μblocks of the same type may be made active by multiple instructions. Hence, it will not be known which of these active handshakes are relevant to the current μoperation that is being executed in a μblock.

A solution to this problem is for the control unit to supply, to each μblock, the next μoperation that is to be executed. This resolves the dependencies and indicates to each μblock where any results generated will be sent.

Figure 3.16 shows part of a scalable μnet datapath. In this datapath, there are multiple register read ports, multiple adders and multiple write-back units. The two-way arrows represent the handshaking signal pairs of μoperations. A particular instruction is to use μoperations `Rx1`, `Ry3`, `AOp4` and `Wz2`, which correspond to ports 1 and 3 of the register file, adder 4 and write-back unit 2. Each datapath μblock also receives the μoperation handshake of the next μoperation, *i.e.* `Rx1` and `Ry1` receive the `AOp4` handshake and `AOp4` receives the `Wz2` handshake.

Interconnecting a μblock to the other μblocks that it can communicate with can be implemented using busses to route the control handshakes between μblocks and their data, instead of fixed connections. Hence, the handshake that specifies the next μoperation which is to be executed, determines the control and data busses to which the output handshake and data will be connected. Figure

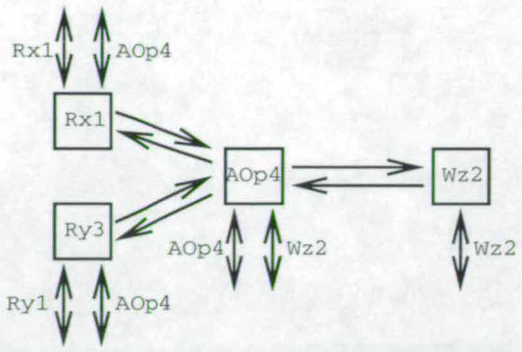


Figure 3.16: Part of a scalable μnet implementation

3.17 shows the control and data bus interconnections between two sets of fully-connected μblocks. There are m busses, which connect the n μblocks on the left side of the figure to the m μblocks to the right. Each μblock must steer its outputs to m busses in this case.

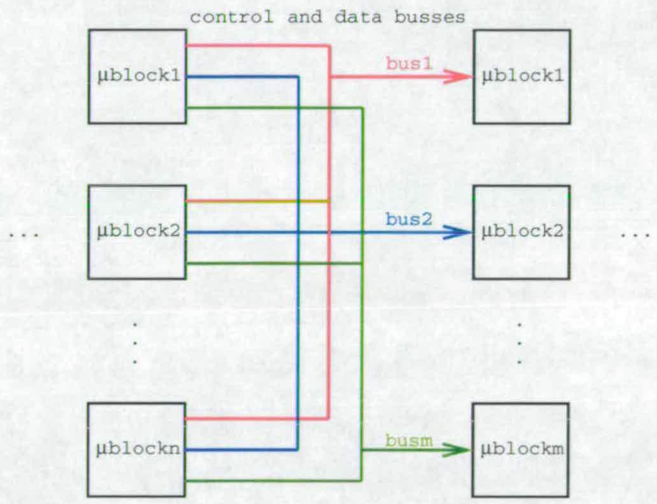


Figure 3.17: Bus Interconnections in a scaled μnet implementation

This approach cannot deal with collisions, *i.e.* two or more μblocks attempting to communicate with the same μblock cannot take place, as the bus control and data signals would, in such a case, assume undefined values and the circuit would malfunction. As long as no feedback connections exist in the μnet structure, appropriate μblock selection will prevent collisions from occurring.

3.8.2 Multiple μnet Datapaths

A different way of scaling a μnet architecture is to replicate the entire μnet datapath, instead of adding more datapath units. Figure 3.18 shows how a scalar

μnet datapath can be replicated.

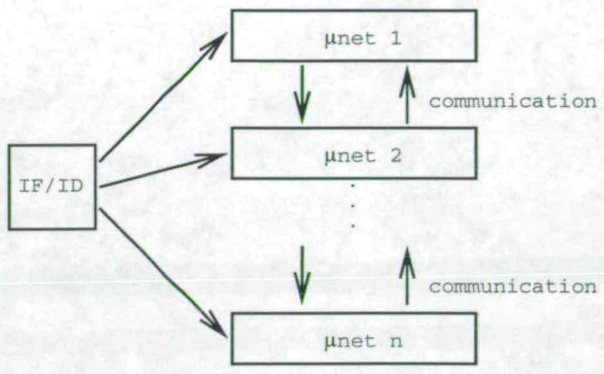


Figure 3.18: Multiple μnet Architecture

In this scheme, instructions from the same instruction stream are distributed to different μnets, effectively forming a clustered uniprocessor architecture. It is also possible to assign different instruction streams to the different μnets, forming a single-chip multiprocessor architecture.

For both of these architectures, communication between the nodes is necessary to exploit a sufficient amount of program level concurrency. The next section presents an approach for achieving this, Shared Register Files.

3.9 Shared Register Files

The Shared Register File (SRF) approach partitions the conventional monolithic register file of a processor into multiple register files that share registers. The SRF approach does not imply a particular architecture, it is only a technique for segmenting the MRF and communicating register values. The SRFs can be incorporated into a processor datapath in different ways: (a) the datapath may be partitioned too, *i.e.* the processor FUs are local to the SRFs, a clustered uniprocessor architecture, (b) the datapath and the flow of control are partitioned, *i.e.* not only the FUs, but also the instruction flow is local to the SRFs, *i.e.* a multiprocessor architecture and (c) the datapath is not partitioned, *i.e.* the SRFs connect to a single set of FUs. Cases (a) and (b) implement architectures that exploit program level concurrency, by running different code fragments on the different SRFs from a single or multiple flows of control.

3.9.1 Concept of register sharing

A shared register is one that is common to two or more RFs of an architecture. Hence, it can be read or written to by all the RFs to which it is common. A shared register file (SRF) is one that contains a multiplicity of shared registers. In addition, an SRF may also contain registers not accesible by other RFs, referred to as local registers. In an SRF organisation, the shared register identifiers overlap across RFs. This overlap provides the communication and synchronisation mechanism. For example, shared register 0 of RF(0) may be the same physical register as register 28 of RF(3).

3.9.2 Possible Sharing Schemes

Shared RF organisations depend on four parameters: the size of the shared regions, *i.e.* the number of shared registers, the number of ports of a shared region, *i.e.* the degree of sharing, the number of shared regions per RF and the register mapping which establishes the register connections. These parameters affect the logical and physical topology of the RFs and dictate the paths and degree of communication.

The simplest register sharing scheme was first mentioned in Chapter 1, Section 1.2.2 and is shown again in Figure 3.19. This establishes a one-way or unidirectional communication path between RFs and is therefore referred to as unidirectional or 1-way register sharing.

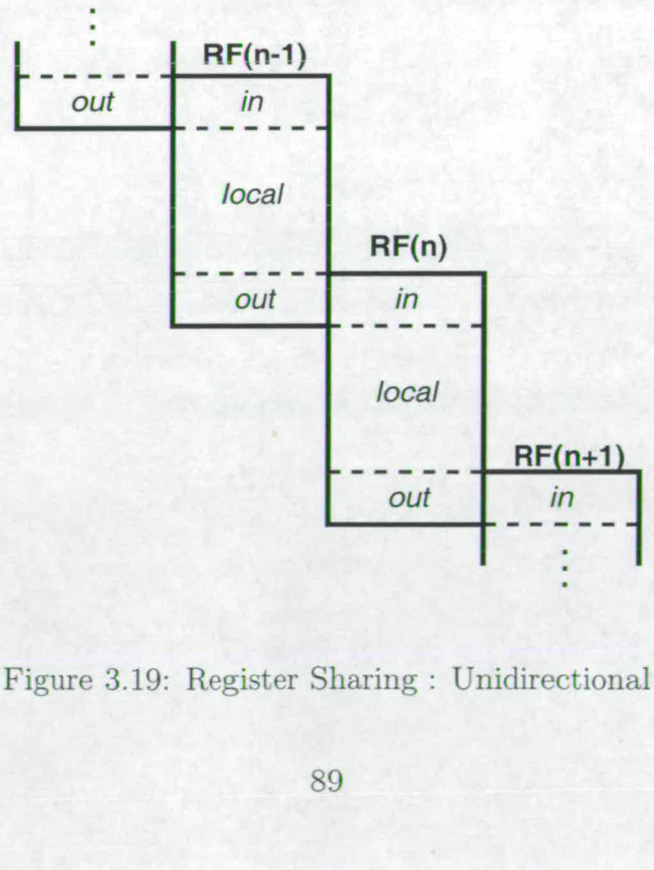


Figure 3.19: Register Sharing : Unidirectional

This scheme, inspired by the register windows concept (Section 1.2.1) divides each RF into three sections: a local section and two shared sections, labelled *in* and *out*. For each RF, its bottom output section, which is used to send data to its successor, overlaps with the latter's input section, which is used to receive data from its predecessor. Therefore, data written to the *out* section of RF(n) can be read by RF(n+1) as it maps to one of its *in* registers. The RFs are organised in a circular manner so that the last one overlaps with the first.

As each RF can only write to its output section and only read from its input section, and these are respectively read from and written to by its neighbours, no extra read or write ports are required. Because of this, there is virtually no access time penalty for accessing registers of another RF.

A more versatile scheme is one that will allow one RF to read and write to more that one neighbour, establishing two communication paths. This 2-way or bidirectional scheme is shown in Figure 3.20.

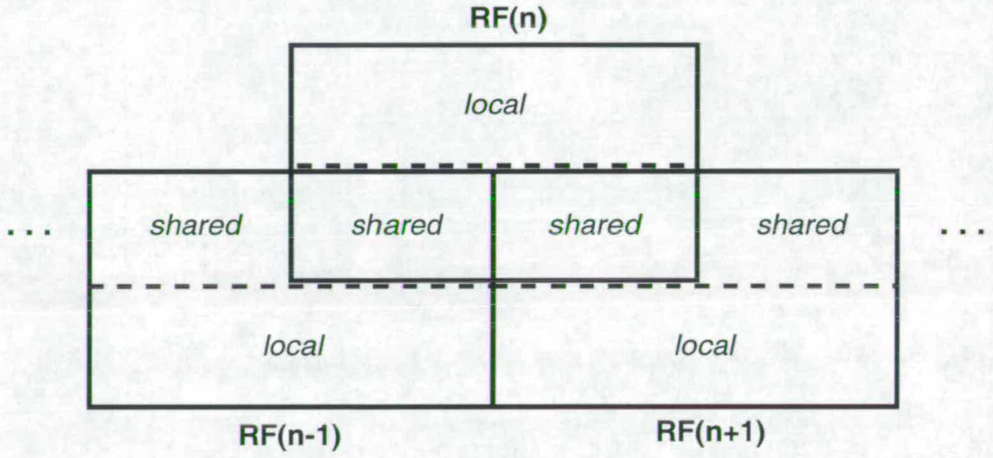


Figure 3.20: Register Sharing : Bidirectional

In the 2-way scheme, each RF has two shared sections, one common with its predecessor and one common with its sucessor. The difference here is that the shared sections can both be written to and read from both processing nodes that share them. As can be seen from Figure 3.20, each shared section requires read and write access from two RFs, and therefore requires one additional read and one additional write port.

A scheme with greater communication ability, but one which would imply a greater number of processing nodes, is the 4-way or quad scheme. As its name implies, each RF in this scheme has four shared sections which it shares with three of its neighbours, as shown in Figure 3.21.

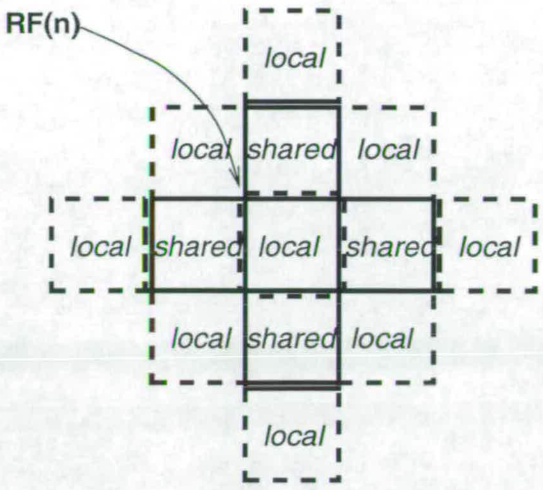


Figure 3.21: Register Sharing : 4-way

This organisation is very similar to a 2D processor mesh, where each RF can communicate in four directions: north, south, east and west. Each shared section in the 4-way scheme can be accessed by four RFs, hence three additional read/write ports are necessary.

These three schemes are not the only ones possible. Thus, the degree of sharing can be further increased, to 6-way, 8-way, etc. and the number of shared sections per processing node can also be increased. In the bidirectional scheme, for example, each RF could have four rather that two shared sections communicating with each other in two paths, horizontally and vertically.

3.9.3 SRF design

When all the parameters for an SRF scheme have been defined, the design of the SRF can take place. This involves specifying the logical structure of each SRF, that is the register mapping, which determines the relationship between register identifiers and register location, and the sharing architecture which determines how the shared sections of this and its neighbouring RFs overlap.

To provide for both logical and physical scalability, an SRF should be designed as a repeatable circuit block. This allows for an arbitrary number of SRFs to be connected.

3.9.4 SRF and MRF issues

SRFs differ from their monolithic counterparts in two respects. Firstly, not all their registers have the same number of ports. Registers of shared sections will have more ports than the local ones. Secondly, register accesses of one SRF may need to be routed to another. A register access in one SRF's shared section could mean that the particular register may not physically be in that SRF. This depends on the physical design of the sharing scheme, as will be illustrated later.

Because of this, local and shared registers will have different access times. The difference between local and shared accesses can be thought of as the communication latency. Comparing an SRF with an MRF with the same number of physical registers, we can expect a slowdown of all register accesses. The reason for this is the existence of registers with different numbers of ports which all connect to the same bus. A slowdown of the local registers is to be expected due to second order capacitive effects; a register with more ports introduces more capacitance on the input and output busses. This slowdown of all register accesses compared to an MRF with the same number of physical registers can be thought of as the sharing overhead, the price paid for having the ability to communicate.

The nature of RFs is that they are datapath components, essentially arrays of static RAM cells with multiple input and output ports, where all cells connect to the same input and output busses [WE93]. This implies that their access time is significantly affected by low level layout details and process parameters. Factors like the circuit topology, *i.e.* the organisation of the datapath and the control logic, parasitic delays (such as the capacitance of metal tracks) and process characteristics (such as the number of metal levels), all directly affect the access time. At the schematic level it is very hard to model the capacitive loading on an RF's busses or the effect of data and control routing, as these are usually implemented using different metal layers. In order to be able to assess the access times of SRFs and to study the impact of register sharing, a set of both SRFs and MRFs were laid out and simulated.

3.9.5 Asynchronous vs. Synchronous SRFs.

SRFs with a higher communication ability than 1-way require a larger number of ports for their shared sections. This implies that the access times of shared registers will be greater than that of local register. SRFs are effectively a register-level equivalent of Non-Uniform-Memory-Access memories, where the speed of access varies depending on the type of access (local or shared). This characteristic makes SRFs hard to implement using the synchronous approach, as a multicycle implementation is necessary for the shared accesses. Asynchronous design, on the other hand, can accommodate the non-uniformity of accesses and for this reason SRFs are easier to implement as asynchronous components.

3.10 Conclusions

In this chapter, the fundamentals of asynchronous systems and asynchronous processors have been reviewed. Two hardware mechanisms for exploiting concurrency have been presented; shared register files and μnet architectures. Shared register files have explicitly defined common regions and allow for communication and synchronisation to take place through the shared registers. They allow for coarse-grain parallelism to be exploited by providing a scalable and segmented datapath organisation which can be used in clustered uniprocessors or single-chip multiprocessor architectures. μnet architectures allow for fine-grain parallelism to be exploited, by allowing μinstructions to be issued and executed concurrently in a μnet datapath. An implementation methodology for scalar and superscalar μnets has been presented. By combining together these two approaches, a scalable asynchronous processor which exploits both fine-grain and coarse-grain parallelism can be implented. The next chapter discusses the implementation of Shared Register Files.

Chapter 4

Implementation of Shared Register Files

In this chapter the shared register approach is presented as a means of partitioning the monolithic RF and of enabling inter-RF communication. This technique aims at a scalable organisation of RFs with explicitly identified common regions where all necessary communication and synchronisation takes place through the shared registers. Further on, the shared register file approach attempts to avoid using complex interconnections and long wires by localising all communication at the "heart" of the datapath, *i.e.* the RFs.

4.1 Asynchronous SRF Circuit Design

The SRFs were designed and laid out (but not fabricated) using ES2's 0.7μm, 5V, 2 level metal, digital CMOS process [EUR]. This technology was supplied by EUROPRACTICE-IMEC. The custom layout package was used with the Cadence Opus layout tools. This package contains a technology file for the layout editor containing all the technology details, a design rule checker for Cadence's DIVA DRC tool and a full flat post-layout circuit extractor. The circuits were laid out hierarchically using the layout editor, then verified for DRC errors and then a post-layout netlist with all the parasitic devices was created and fed into the HSPICE simulator [Met90].

The circuits were designed with minimum size transistors and minimum width wires, unless otherwise required, for reasons of speed. The widths of the power tracks were calculated for each circuit block depending on the current drawn by each circuit. Wider transistors were used when implementing transistor chains (to keep the equivalent load and current drive similar to that of a minimum size transistor) and for buffering. As the purpose of these experiments was to study relative delays, the use of buffering was kept to a minimum to reduce circuit complexity and save on design and implementation time.

To implement the logical sharing structure physically, a physically repeatable pattern must be created, as mentioned above, where neighbouring RFs logically overlap. This is implemented by allocating part of the shared section of one SRF to its neighbours. For example, as shown later in Section 4.1.3, in the 2-way or bidirectional scheme, the shared section on the left, in Figure 4.1, between RF(n) and RF(n-1) resides in RF(n), but the shared section on the right, between RF(n) and RF(n+1) resides in RF(n+1). Thus, an arbitrary number of SRFs can be connected in a bidirectional ring.

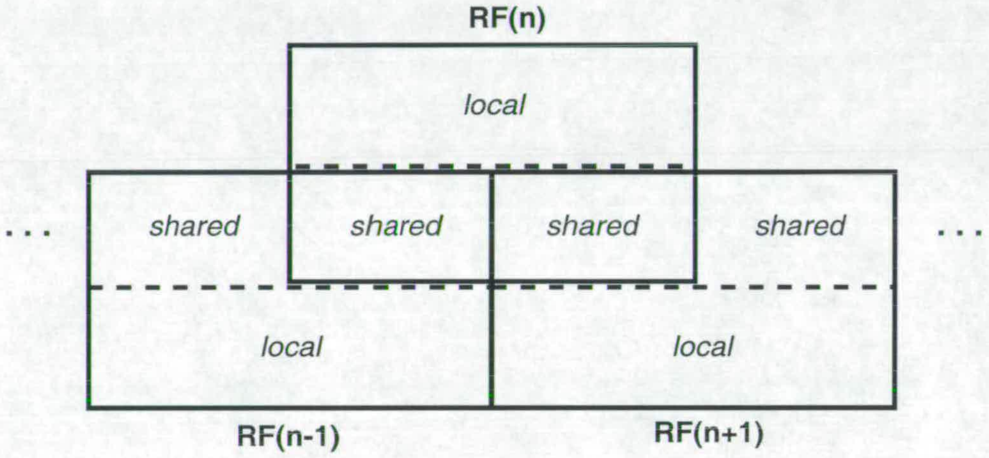


Figure 4.1: Register Sharing : Bidirectional

Three types of register operation were implemented; read, write and clear. All SRFs in the experiments have one port for each operation. All ports implement a four-phase asynchronous handshaking protocol with two signal wires each, a request and an acknowledge.

Figure 4.2 shows the layout of a 1-bit register cell. This is a 1-bit dynamic register cell with a weak p-type feedback transistor. It has an extra buffer for driving the output. This ensures that no matter what the output loading is, the state of the cell can still be changed. The two n-type pass transistors at the input and output are used for writes and reads respectively. Increasing the number of ports implies increasing the number of pass transistors.

Each register is composed of 32 register cells (Figure 4.3) whose write and read lines are driven by x32 buffers. Registers are organised in a 2D array and selected by row and column. This reduces the complexity of the decoding logic and yields a better layout aspect ratio [WE93]. The shared registers have extra ports for remote accesses. The SRF main input and output data busses are formed by

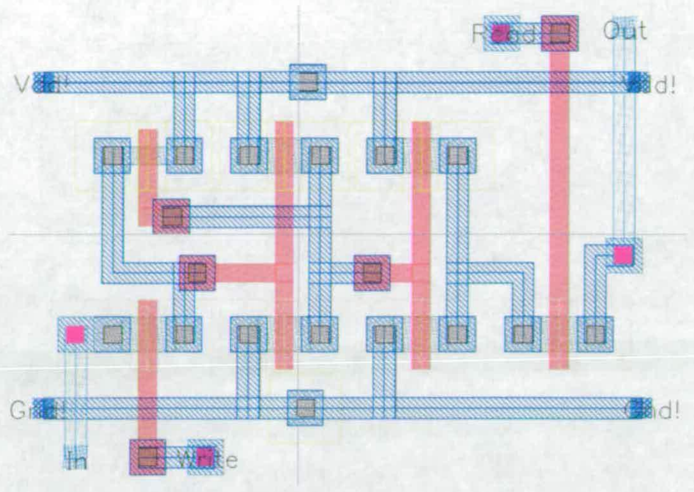


Figure 4.2: 1-bit register cell layout

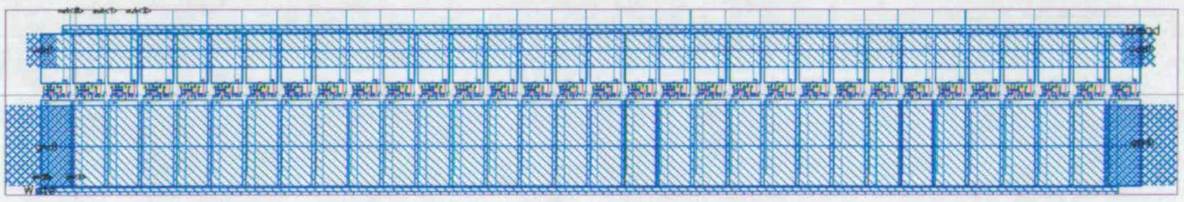


Figure 4.3: 32-bit register cell

joining together all the input and output data busses of the registers that can be accessed. These include the input and output busses of the local registers and the input and output busses of the local port of the shared registers. The remote ports of the shared registers are also connected together to form the remote input and output data busses. This is where other SRF busses can be connected to.

An SRF operation takes place as follows:

- a register access is initiated.
- the register identifier is decoded and register select signals are asserted.
- for a read access, the register select signals enable the outputs of the selected register to drive the SRF's output bus.
- for a write access, the register select signals enable the inputs of the selected register to be driven from the SRF's input bus.
- for a local register access, the register select signals enable a local register, for a shared register access they enable one of the shared register ports.

- for shared registers which reside in another SRF, the register select signals, which are routed to the other SRF, drive one of these shared register ports onto this SRF's input or output bus.
- when the data have been read, written or cleared, then the operation has completed.

A register has different access times depending on the access distance to it, with local registers being the fastest to access. Access times to shared registers depend on the distance of the shared register - a shared register may be physically contained in the same SRF where the operation was initiated, or in another SRF to which the initiating SRF has access. The former can be called a shared-near access and the latter a shared-far access. Thus, each SRF has three possible access times when connected into a system of SRFs: local (the fastest), shared-near and shared-far (the slowest).

When an SRF is simulated as an isolated circuit block, the links to its neighbours are unconnected, so the shared-far access time cannot be measured. In addition, the access times of the local and shared accesses may change depending on the placement and distance of its neighbours, as these links load the SRF's input and output busses.

Connectivity between SRFs is achieved by connecting the remote data busses of one SRF to another's main SRF busses and connecting some of the latter's register select signal to the former's shared registers.

4.1.1 Completion Detection

Completion detection is necessary to detect that data have been read or written to and that the operation has completed. Register locking is a useful and common approach for synchronising out-of-order read and write operations. Both are implemented using a 33rd bit, the valid bit. The valid bit is set when data are written to a register to indicate that the register contents are valid. The valid bit has multiple read ports, one associated with each of the read, write and clear operations. These connect together to form the valid bit busses. In the simplest case, *i.e.* 3 ports, one write, one read and one clear, there are three valid bit busses.

One of the ways of detecting completion in an asynchronous circuit is to use a constant delay path reference, which is longer by circuit design than the datapath (*c.f.* Section 3.1.2). The valid bit is used in this way. During an operation, the corresponding valid bit bus is firstly precharged before the register to be accessed is selected. The precharging is of the opposite value to that which will signal completion of the operation. For reads and writes the valid bit bus must be read as a logic 1. For a write, this means that the SRAM cell of the valid bit has been written to and therefore so have the other 32. For a read, it means that the SRAM cell of the valid bit has been read and therefore so have the other 32, so the output bus data are valid. So for reads and writes, the valid bit bus is precharged to a logic 0. Clear operations clear the valid bit and their completion is signalled by a logic 0, so the clear valid bit bus is precharged to a logic 1.

Due to the use of the valid bit as a locking mechanism, a read of a register which is waiting to be written, *i.e.* its valid bit is zero, will thus not complete until the write takes place. For correct operation, valid bits must be cleared before a register with valid data can be written to. This is usually performed in a processor before instructions are issued. Although reads and writes of shared sections can be performed by multiple SRFs, clears have been physically restricted to the SRF in which an operation takes place. This restricts the number of clear valid bit ports and busses to one.

4.1.2 Control Circuitry

The SRF's control circuits were implemented from asynchronous state machine specifications using the direct-mapped AFSM method described in Chapter 2.

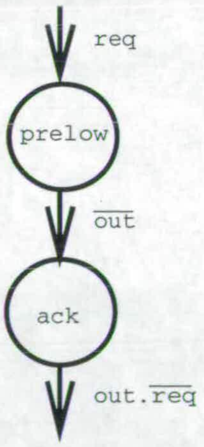


Figure 4.4: FSM of read/write port logic

Figures 4.4 and 4.5 show the FSM and the layout of the read and write port logic. The handshake signals are `req` and `ack`, signal `out` is the bus of the completion detection signals and signal `prelow` precharges this bus low by being connected to a pull-down transistor not shown in the FSM. This FSM does not

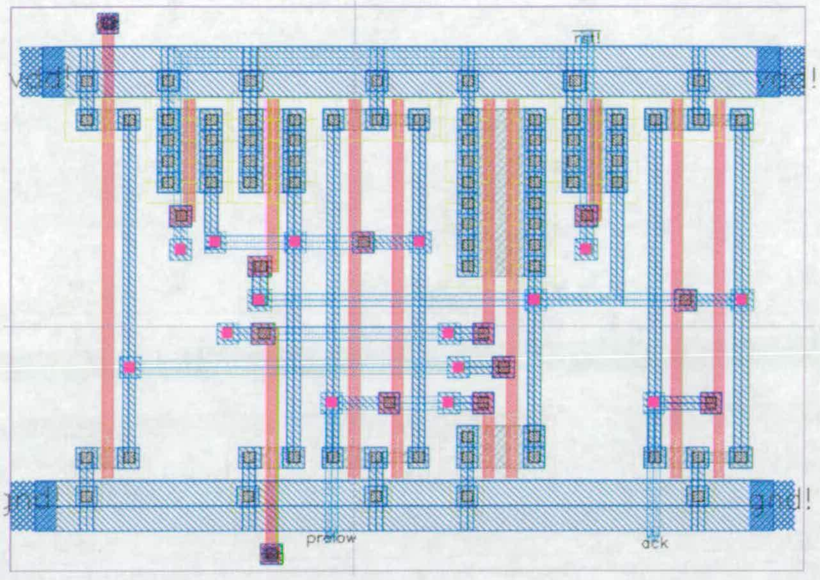


Figure 4.5: Layout of the read/write port logic

reset to either of its states; this is why the top and bottom transistions are shown to come and go from and to the outside. When `req` is asserted, this triggers the start of the read or write operation and puts the FSM in the `prelow` state. This precharges the completion detection bus. When the bus of the completion detection signal (`out`) has been pulled to a low enough value, the machine asserts the `ack` signal, by entering that state. The acknowledge signal is fed to the column decoders enabling them to select the register being accessed. This initiates the access. When the register is selected for a read or a write, its valid bit will be enabled onto the `out` bus. When the `out` bus is asserted and the `req` signal has gone low, the `ack` signal goes low, signifying the end of the operation.

In the following section the implementation of the three different organisations studied here is described. The 2-way scheme is described first as it illustrates more clearly the SRF issues and then the other two, the unidirectional and the 4-way scheme. The access times of the unconnected SRFs are measured and assesed.

4.1.3 The 2-way sharing organisation

The 2-way sharing organisation (Figure 3.20) establishes a bidirectional ring between SRFs by having each share a section with two of its neighbours. The implementation is illustrated in Figure 4.6. When a register access occurs, the register select signals, represented by the dashed lines, may access a local or a shared register residing in the SRF to which the access was made, or a shared register in the next SRF.

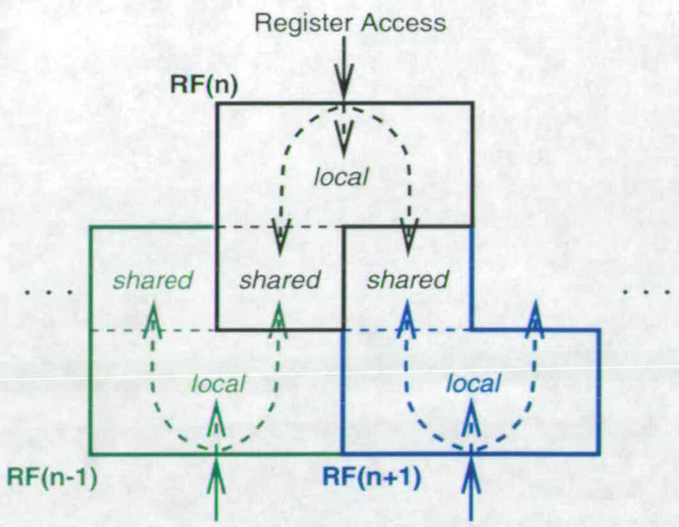


Figure 4.6: Bidirectional Connectivity Diagram

The floorplan for a 2-way SRF implemented with 16 local and 8 shared registers per SRF is shown in Figure 4.7. The shared registers are those which physically exist in the SRF. This shows 16 local registers at the top of the picture labelled with register indices from 8 to 20 and 8 shared registers at the bottom labelled 0 to 7. The total number of registers accessible from this SRF when connected to its neighbours is 32. Shared registers 24 to 31 exist physically in the next SRF and can only be accessed when the SRF is appropriately connected. The shared registers are shown wider (*c.f.* Figure 4.8 which shows the physical layout) as they have one extra read and one extra write port. The remaining blocks illustrate the position of the SRF's control logic and its data busses.

As registers are organised in a 2D array, their outputs and inputs have to be connected by busses running both vertically and horizontally. The busses in the middle contain the two main busses of the SRF. These connect all the inputs and all the outputs of the local registers along with the inputs and the outputs of the first ports of the shared registers. The third bus in the middle connects the outputs of the 2nd port of the shared registers while the bus at the bottom of the diagram connects the inputs of the 2nd port of the shared registers. The last two are to be connected to the data busses of the previous SRF.

Contrasting the layout with the floorplan, the local and shared registers and the control circuits can be distinguished. Their difference in width is apparent. The `METAL1` layer, the darker blue, is used for horizontal routing and power and ground lines, whereas `METAL2`, the lighter blue, is used for vertical routing. The data bits and completion signals of the registers are routed vertically using `METAL2`

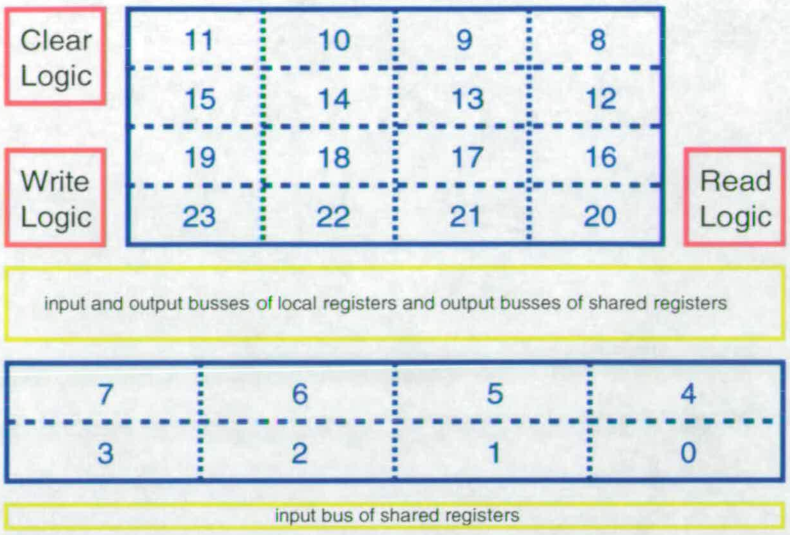

Clear Logic
11
10
9
8
15
14
13
12
19
18
17
16
23
22
21
20
Write Logic
Read Logic
input and output busses of local registers and output busses of shared registers
7
6
5
4
3
2
1
0
input bus of shared registers

Figure 4.7: Bidirectional Register File Floorplan - 16 local, 8 shared registers.

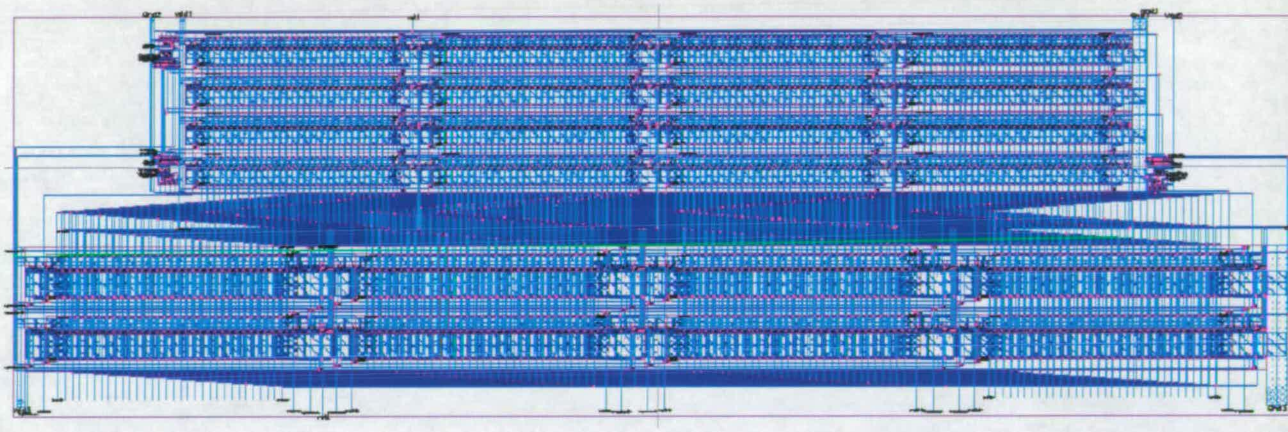


Figure 4.8: Bidirectional Register File Layout - 16 local, 8 shared registers.

and horizontally using `METAL1` to form the SRF's data and completion signal busses.

Connecting SRFs into a 2-way system involves connecting the following signals for each pair of SRFs:

- data busses, input and output, of the shared registers of SRF(n+1) to SRF(n).
- completion signals of the shared registers of SRF(n+1) to the completion signals of SRF(n).
- register select signals for columns of SRF(n) to the columns of the shared registers of SRF(n+1).

- register select signals for rows of SRF(n) which access the shared registers of SRF(n+1) to these registers.

The layout strategy followed has been to place the shared registers, which will be physically wider than the local ones as they have a greater number of ports, at the bottom of the SRF layout, and the local registers at the top. The SRF port circuits, that enable the SRF to be connected into a system, and include the handshaking circuit and the register decoding have been fitted into the empty vertical space created by the difference in width between the shared and local registers.

The register sharing programming model defines linear ranges of the register indices of an SRF which map to local and shared registers respectively. These ranges map to groups of local and shared registers in the layout. It is possible to mix local and shared registers in the physical level, i.e. in the layout, while maintaining the logical grouping of local and shared register by routing the register select signals appropriately, but this would imply an irregular layout. If this approach was followed, it would be harder to find space to place the SRF port circuits. In addition, such a layout would be a lot harder to route, debug and test because it is irregular.

One of the most important factors that determines register access time is the distance of a register from the port circuit. The further away the register is, the longer the register select signals have to travel to select it. When considering the physical organisation of the registers it is desirable to place them uniformely around the ports to keep the effect of the different register positions negligible.

In the implementation presented, registers are organised in rows and columns. This avoids using a monolithic register index decoder, as decoding circuitry does not scale well in terms of speed or number of transistors, so it is best if it is kept small. The number of columns in the SRF has been fixed to four, for both local and shared sections. The height of the local or shared sections can vary depending on the number of each. It is limited by the number of bits that the row decoder can handle. For example, in the bidirectional SRF, a 3 to 8 decoder has been used allowing for a total of 8 rows, but some of these row select signals route to the neighbour, for example in the 2-way SRF with 16 local and 8 shared registers, 4 row select signals route to the local registers, 2 to the shared registers and the remaining 2 must route to the second port of the neighbouring SRF.

The row and column organisation could have been different. For example, two rather than four columns could have been used. That would increase the row decoding complexity, a 4 to 16 row decoder would be required, and would yield a different layout aspect ratio. The choice of a four column layout organisation was made because, firstly it yields an overall layout which is closer to a square shape than other column organisations, *i.e.* three or five, and secondly as it keeps the decoding logic simple.

Due to the asynchronous circuit operation, the access time of each register can be observed to vary depending on its physical location and on the previous access. The former is due to signals travelling different distances and the latter because the column select signals are always enabled and therefore another access in the same column will be faster.

To visualise this access time variation over an RF, an access time map can be drawn. Such a map shows the access times of registers relative to their position and for a fixed access order. The latter is important as it also affects the access time. The access time map of the 2-way SRF with 16 local and 8 shared registers is shown in Figure 4.9.

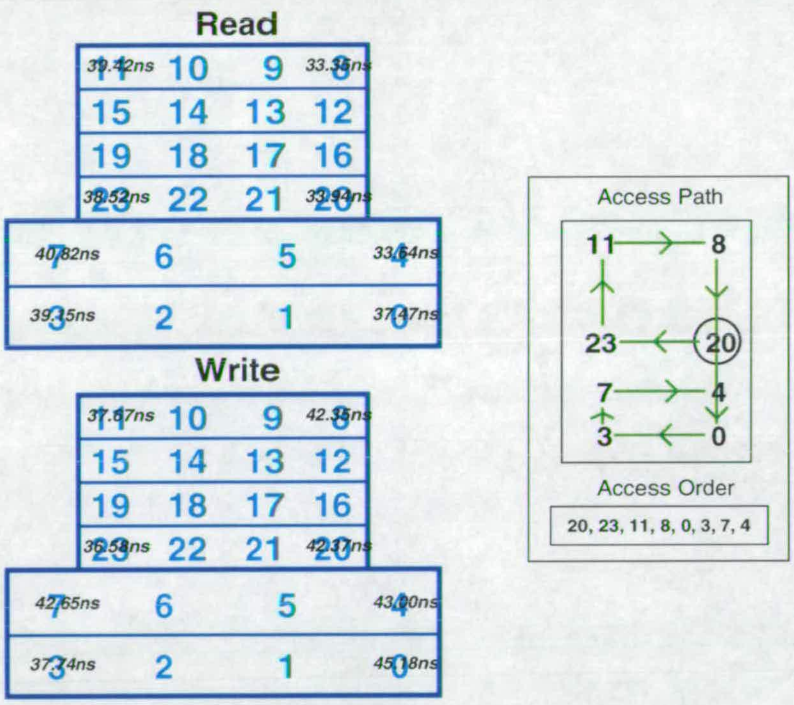


Figure 4.9: Access Time Map for the 16/8, 2-way Shared RF

The access order is shown and illustrated on the right hand side. On the left the measured access times are overlayed on top of the register locations for both reads and writes. Register 20 is accessed first, on the bottom right corner of the local area and then the other three corners in a clockwise manner. Then, the four corners of the shared area are accessed, again clockwise. Recalling that the write logic is physically on the left of the local area and the read logic on the right we can see from the diagram that write accesses to the left of the SRF are faster than those on the right and the opposite for reads. In addition, we can see that consecutive accesses on the same row are faster, for example the read of register 8 is faster than the read of register 20 although register 20 is closer to the read logic; that is because the access preceding that to register 8 was to register 14 and the top row was already selected when register 8 was accessed. Another example is the write to register 3; it is faster than the next write, to register 7 although register 7 is closer to the write logic, this is again because the access preceding that to register 3 was to register 0.

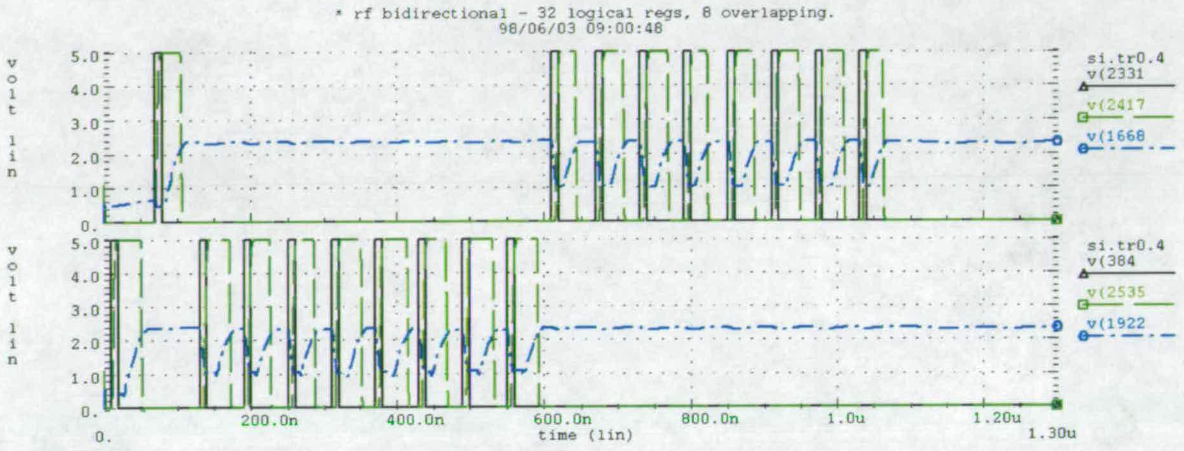


Figure 4.10: Access Time Measurement Waveforms

Figure 4.10 shows the testbench simulation waveforms from HSPICE. The top panel shows the read handshakes and the read valid bit bus completion signal and the bottom panel the write handshakes and the write valid bit completion signal. The request signals are drawn in black, the acknowledge in green and the completion signals in blue. The completion signals are precharged low at the beginning of an operation, then left to float and when detected high this signals completion of the operation.

From the access times measured with this fixed ordering we can estimate the average read and write access times for an asynchronous SRF. This is done by averaging the four local and the four shared access times for both reads and writes to produce the average local and average shared access times. These averages are not meaningful in themselves, but are useful metrics for comparing different SRF configurations. For this SRF these values are 36.31ns and 39.74ns for reads and 37.77ns and 42.14ns for writes.

To study the effect of increasing the number of shared registers two more 2-way

SRFs were implemented, one with 20 local and 4 shared registers and the other with 8 local and 16 shared registers, *i.e.* the total number of physical registers remained constant.

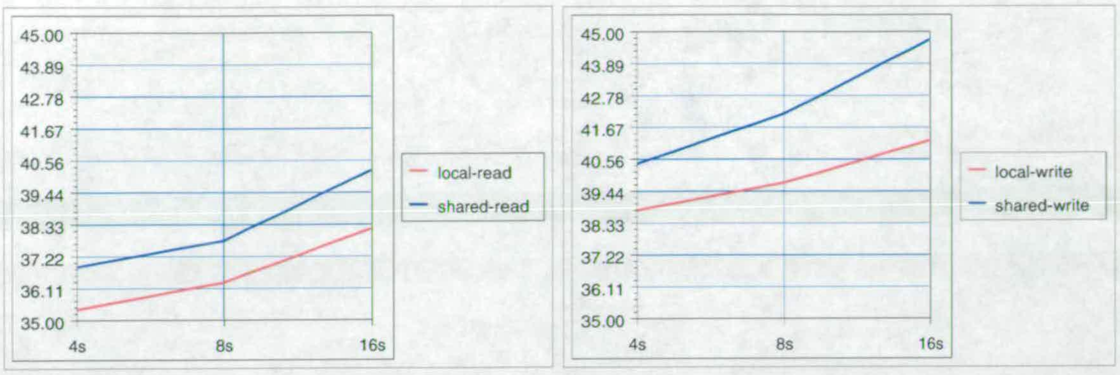


Figure 4.11: Access times(ns) for 2-way SRF with 4, 8 and 16 shared registers

No. of shared registers	local-read	local-write	shared-read	shared-write
4	35.37	38.78	36.86	40.40
8	36.31	39.74	37.77	42.14
16	38.19	41.21	40.22	44.73

Table 4.1: Access times(ns) for 2-way SRF with 4, 8 and 16 shared registers

The average access times for the three 2-way SRFs are shown in Figure 4.11 and Table 4.1. The graph on the left, in Figure 4.11, is the average access time in ns for local and shared reads and the one on the right for local and shared writes. The effect of swapping local for shared registers is a quadratic increase in access times of both local and shared registers.

This quadratic increase is due to the increase of both the resistance and the capacitance of the RF datapath when swapping local registers for shared ones, as the latter have a larger number of ports. This effect was first mentioned in Section 1.1.4 in relation to the access time for an MRF and is an expected result. So, the larger the shared section of a 2-way SRF, the slower the access time, with a quadratic dependance on the latter.

4.1.4 The 1-way sharing organisation

The 1-way organisation was first described in Section 1.2.2, Figure 3.19. This establishes a unidirectional ring between SRFs using a connectivity similar to the register windows mechanism.

This scheme is different from the 2-way scheme as the shared registers here require no extra read or write ports. This is because some of the registers in each SRF are read-only and some write-only. The fact that all registers, local or shared, have the same number of ports implies that an SRF in this scheme will show little difference in access time relative to a monolithic scheme with the same total number of registers.

The differences between the two are the routing of the register select signals and that of the data and valid bit busses. For the shared registers of this scheme, their register select signals will come from the previous SRF for writes and will go to the next SRF for reads. The data and valid bit busses must also be connected appropriately, just as in the 2-way. The input bus of the previous SRF is connected to the shared registers' input bus and the output bus is connected to the next SRF's output bus. The valid bit outputs of the shared registers go to the previous SRF and the valid bit busses of the local registers have to be connected to the valid bit busses of the next SRF's shared registers.

The fact that more ports are not required for the shared registers of an SRF in this scheme implies that the number of shared registers does not have a first order effect on the SRF's access time.

Figure 4.12 shows the layout of a unidirectional RF.

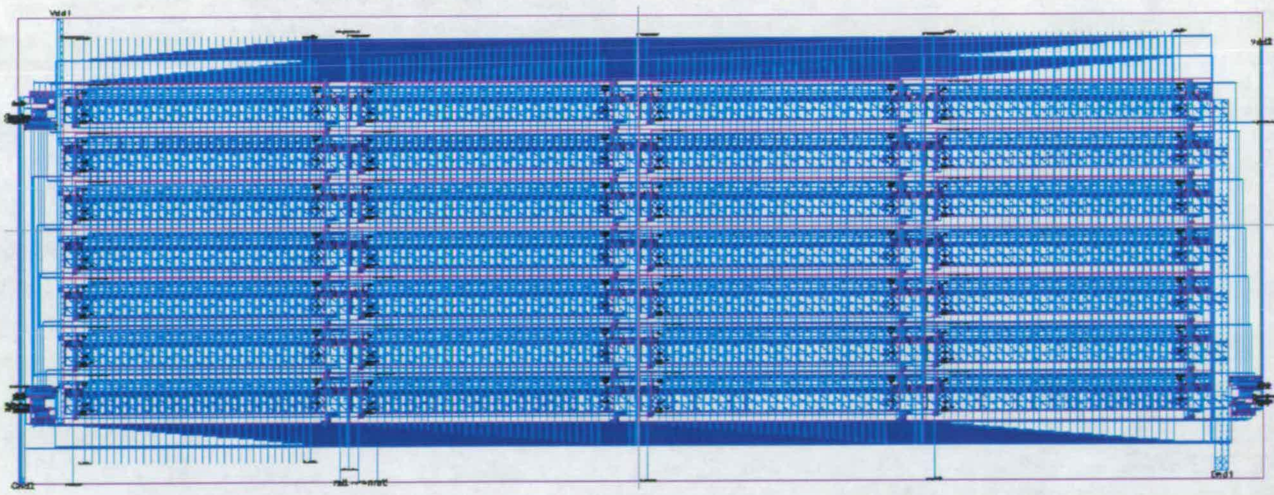


Figure 4.12: Unidirectional Register File Layout - 28 local, 4 shared registers.

The access time was estimated in the same way as for the 2-way, *i.e.* by averaging the measured access times of the four corners. The values obtained were 33.57ns for local and shared reads and 36.01ns for local and shared writes.

4.1.5 The 4-way sharing organisation

The 4-way scheme was first described in Section 3.9.2, Figure 3.21. This establishes a 4-way communication between every SRF and its neighbours. Each SRF has access to four shared sections that it shares with its neighbours to the east, west, north and south.

Physically, each SRF is composed of one local section, one shared section and three external interfaces, through which it can be connected to the shared sections of its neighbours. Figure 4.13 shows part of such a configuration.

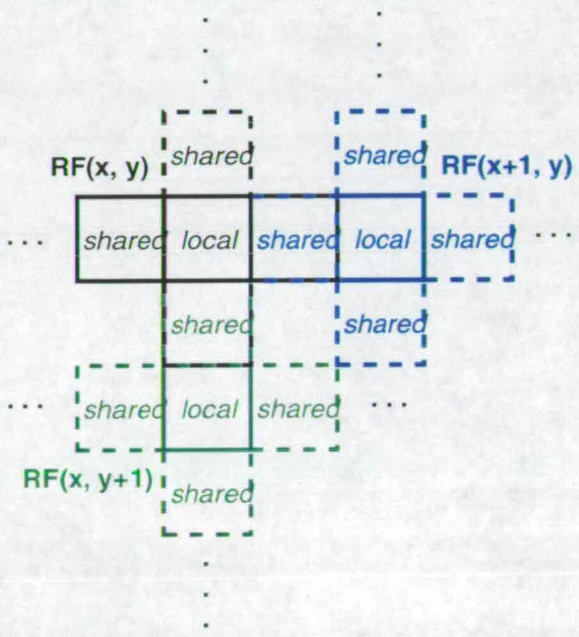


Figure 4.13: 4-way Connectivity Diagram

In the 4-way sharing organisation the register indices are organised differently, as the number of shared sections is more than two. Local registers start from register index 0 up to n, for n local registers. The rest of the indices are allocated to the east, west, north and south shared sections respectively.

The layout of the 4-way SRF with 16 local and 8 shared registers is shown in Figure 4.14. The total number of registers accessible from this SRF when connected to its neighbours is 48. The differences between this and the 2-way scheme are the register indexing, the width of the shared registers (wider as they require four rather than two ports) and also the number of busses required, four read and four write.

Three 4-way SRFs were implemented with 4 shared and 20 local registers, 8 shared and 16 local registers and 12 shared and 12 local registers. The average

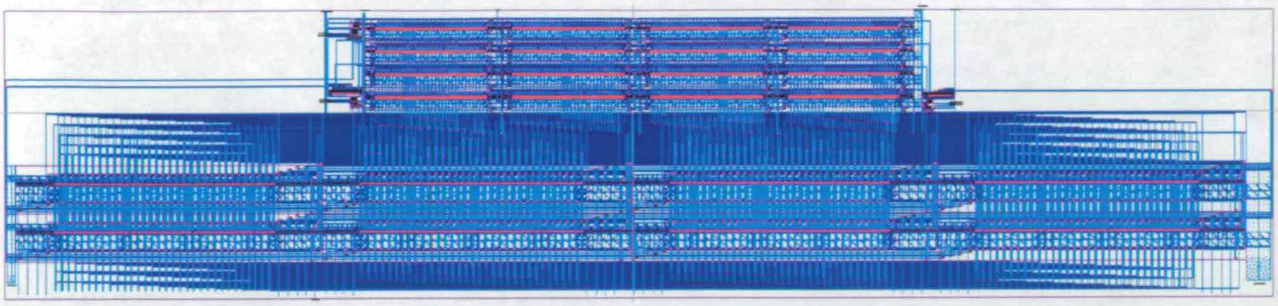


Figure 4.14: 4-way Register File Layout - 16 local, 8 shared registers.

access times for the three 4-way SRFs are shown in Figure 4.15 and Table 4.2.

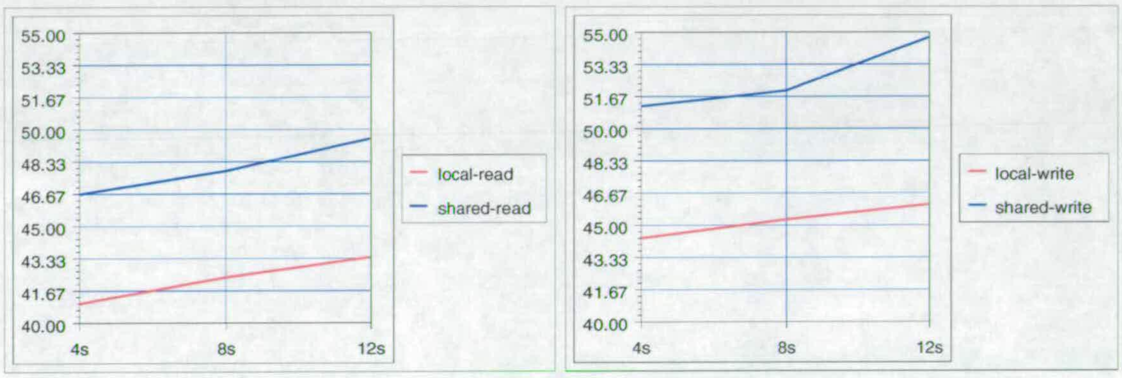


Figure 4.15: Access times(ns) for 4-way SRF with 4, 8 and 12 shared registers

No. of shared registers	local-read	local-write	shared-read	shared-write
4	41.09	44.04	46.77	51.76
8	40.96	46.21	47.91	51.41
12	43.25	46.45	49.55	54.81

Table 4.2: Access times(ns) for 4-way SRF with 4, 8 and 16 shared registers

4.1.6 SRF Access Times

The access times for reads and writes for all the SRFs studied here are shown in Figure 4.16.

Moving along the x-axis changes the configuration. The gap between the two lines of the upper graph shows the difference in access time between a local and a shared register read and the gap between the two lines of the lower graph the difference between a local and a shared register write. For the 1-way SRF there

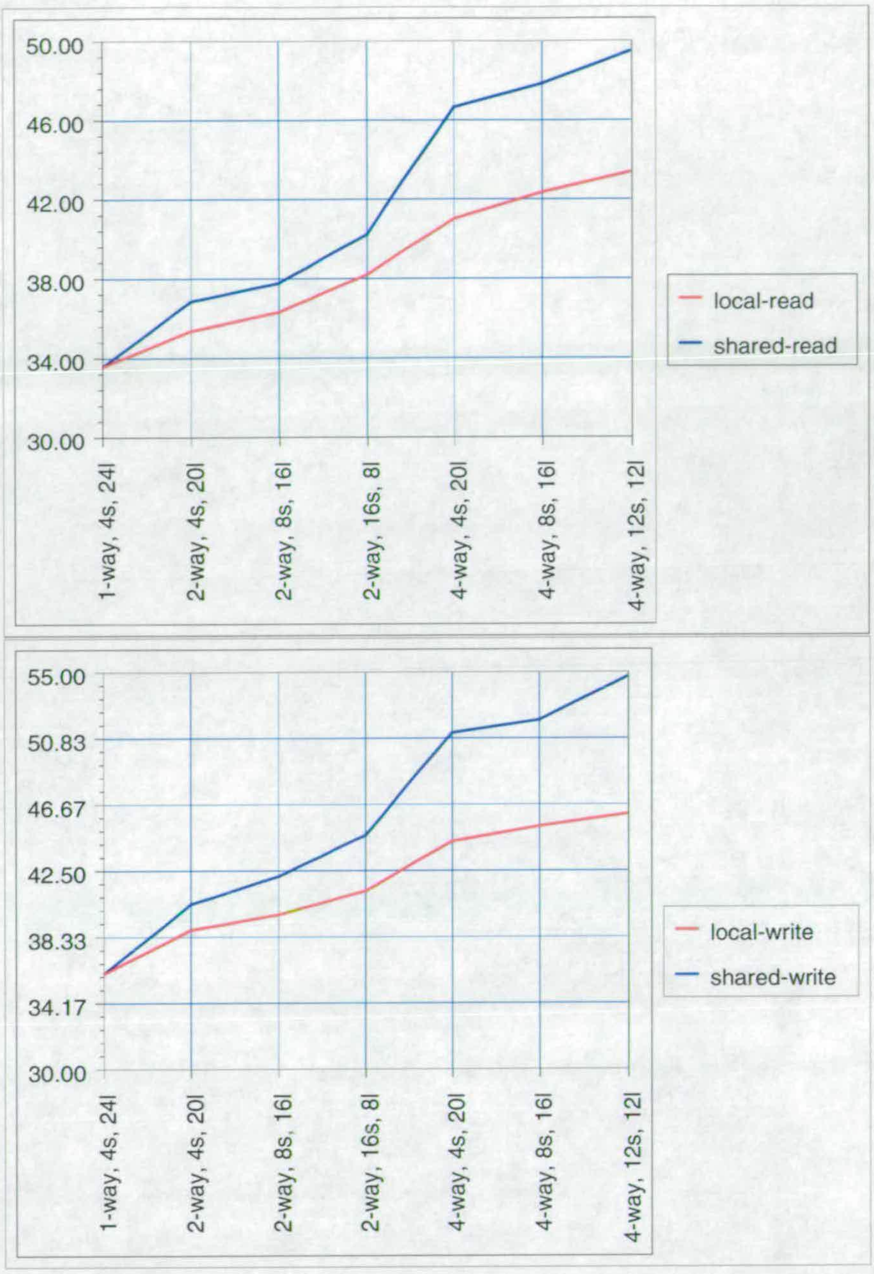


Figure 4.16: Average Access Times in ns for Local and Shared Registers

is no difference between a local and a shared access. For the 2-way and 4-way configurations, the graph shows the effect of swapping local registers for shared registers while keeping the total number of physical registers in a cluster constant.

The difference between a local register access and a shared register access increases with increasing distance along the x-axis, *i.e.* the read and write graphs move further apart. A quadratic increase in the access time can be observed among the 2-way and 4-way configurations and there are clear jumps when moving between different configurations.

The ratio between these unconnected SRF access times and between the access times of local and shared registers varies depending on the connectivity between an RF and its neighbours, as will be shown in Section 4.3. It has been found that the placement of a neighbouring RF has a significant effect on the local and shared register access times, as the control and data signals travel different distances depending on the neighbours' positions.

4.2 Effectiveness of SRFs in a system

In the previous section the implementation of three SRF organisations was detailed and the measured access times were presented. In this section the effectiveness of the SRF approach is quantified. Unconnected SRFs and, in the later sections a fully connected system of SRFs, are evaluated using quantitative performance metrics.

In order to consider the impact of using SRFs in an architecture, the SRFs presented are compared with a reference MRF. This has the same number of total physical registers as the SRFs and the same number of ports as the local section of the SRFs. It is used as a reference because the SRFs are effectively derived from it.

4.2.1 SRF Performance Metrics

The use of SRFs has two effects on performance. It slows down all register accesses by a certain amount, and implies an extra delay for shared register accesses. These two effects can be abstracted to two metrics, *cost*, *i.e.* a universal penalty for using this approach, and *communication latency*, *i.e.* an extra delay for communication. The metrics can be defined as follows. The *cost* metric is the ratio of extra time required for register accesses (local) compared to the reference MRF, *i.e.* with the same number of physical registers. The *communication latency* is the ratio of extra time required for a shared register access compared to a local one. The *normalised communication latency* is the ratio of extra time required for a shared register access compared to the access time of the reference MRF. The latter is used as it makes communication delay more apparent.

4.2.2 SRF Performance

The values calculated for the performance metrics described for the 2-way and 4-way unconnected SRFs are shown in Table 4.3.

2-way		*reads*	*writes*
4 shared			
	Cost(%)	9.34	9.18
	Latency(%)	4.22	4.18
	Norm-Lat(%)	13.95	13.74
8 shared			
	Cost(%)	12.25	11.89
	Latency(%)	4.03	6.04
	Norm-Lat(%)	16.76	18.64
16 shared			
	Cost(%)	18.06	16.02
	Latency(%)	5.32	8.55
	Norm-Lat(%)	24.33	25.93
4-way		*reads*	*writes*
4 shared			
	Cost(%)	27.02	23.99
	Latency(%)	13.83	17.53
	Norm-Lat(%)	44.58	45.73
8 shared			
	Cost(%)	26.62	30.15
	Latency(%)	16.97	11.26
	Norm-Lat(%)	48.1	58.92
12 shared			
	Cost(%)	33.7	30.78
	Latency(%)	14.57	18
	Norm-Lat(%)	53.17	54.31

Table 4.3: SRF Performance Metrics

To illustrate the use of SRFs and the effect of the cost and latency metrics on performance, the execution of a simple program is considered at the instruction level.

Consider a program (Figure 4.17) that is to add eight numbers, n1, n2, ... n8 and accumulate the result in a register. It first adds the first two numbers together and then adds the next number to the result until all the numbers have been added. Two operations are shown in the program, `OF`, operand fetch, which reads a register, and `ADD`, which adds two numbers together and writes the result back to the RF. The brackets show the value being fetched or the result of an add. The time steps show the time taken to execute a program operation; the steps are not necessarily of equal length.

These two operations are assumed to be executed on an imaginary processor in which everything takes zero time except the register accesses. The time taken

time	*operation*
t_0	OF1(n1)
t_1	OF2(n2)
t_2	ADD(n1+n2)
t_3	OF1(n1+n2)
t_4	OF2(n3)
t_5	ADD(n1+n2+n3)
...	...
t_{14}	OF2(n8)
t_{15}	ADD(n1+n2+n3+n4+n5+n6+n7+n8)

Figure 4.17: Execution of a program that adds 8 numbers

to execute an OF operation is the time for a register read and the time for an ADD is that for a register write. The reason for doing this is to show the effect of different RF organisations on the execution time of this program. The processor can have an MRF or multiple SRFs, *i.e.* multiple nodes as its datapath. These operations can then be executed on a single node or multiple nodes of an imaginary processor.

The OF and ADD operations take 1 time unit for a processor with the reference MRF as its datapath. For other configurations, an ADD still takes 1 time unit whereas an OF takes (1 + cost), where the cost depends on the configuration. For shared reads and writes the time taken is (1 + cost + latency) for the corresponding values of cost and latency. It will also be assumed that the cost and latency values calculated for the unconnected SRFs do not change when SRFs are connected into a system. In reality they could change but here it will be assumed that the change is relatively small. All the operations of the program will be assumed to execute on the processor in lockstep. If multiple operations execute in parallel on a processor configuration then, before the next operation can be executed, all the preceding ones must finish, *i.e.* the time of the longest operation dominates. This simplifies the timing as operations do not overlap. Then, if the program of Figure 4.17 is executed on the processor with the reference MRF (24 physical registers) this will take $16 * 1 = 16$ time units.

Consider now the execution of the program on a 4 node, 2-way SRF system where it is allocated to two execution nodes of the system with the aim of utilising these two SRFs as much as possible. Then, the program must be allocated in such a way that both nodes of the system execute instructions for as long as possible. The execution of the program with such an allocation is shown in Figure 4.18.

The bold text in brackets shows the shared register operations. SW stands for shared write and SR for shared read. The program is broken down into two threads with four adds taking place in the first SRF and four in the second one. At time step t_8, the result of the four additions in SRF0 is communicated to SRF1

time	SRF0	SRF1
t_0	OF1(n1)	OF1(n3)
t_1	OF2(n2)	OF2(n4)
t_2	ADD(n1+n2)	ADD(n3+n4)
t_3	OF1(n5)	OF1(n7)
t_4	OF2(n6)	OF2(n8)
t_5	ADD(n5+n6)	ADD(n7+n8)
t_6	OF1(n1+n2)	OF1(n3+n4)
t_7	OF2(n5+n6)	OF2(n7+n8)
t_8	**[SW]** ADD(n1+n2+n5+n6)	ADD(n3+n4+n7+n8)
t_9		OF1(n3+n4+n7+n8)
t_{10}		**[SR]** OF2(n1+n2+n5+n6)
t_{11}		ADD(n1+n2+n3+n4+n5+n6+n7+n8)

Figure 4.18: Program execution on 2 nodes of a 4 node, 2-way organisation

by being written to a shared register. It is then read at time step t_{10} by SRF1 and the final result is calculated. If the 2-way organisation is to have 8 shared and 16 local registers, then by using the cost and latency data from Table 4.3, the OF delay can be calculated to be 1.1225 and the ADD delay to be 1.1189 for local accesses. For the shared write at t_8, its delay will be 1.1793. Thus the execution time can be calculated to be 13.5613 time units.

time	SRF0	SRF1	SRF2	SRF3
t_0	OF1(n1)	OF1(n3)	OF1(n5)	OF1(n7)
t_1	OF2(n2)	OF2(n4)	OF2(n6)	OF2(n8)
t_2	ADD(n1+n2)	**[SW]** ADD(n3+n4)	**[SW]** ADD(n5+n6)	ADD(n7+n8)
t_3	OF1(n1+n2)			**[SR]** OF1(n5+n6)
t_4	**[SR]** OF2(n3+n4)			OF2(n7+n8)
t_5	ADD(n1+n2+n3+n4)			**[SW]** ADD(n5+n6+n7+n8)
t_6	OF1(n1+n2+n3+n4)			
t_7	**[SR]** OF2(n5+n6+n7+n8)			
t_8	ADD(n1+n2+...+n8)			

Figure 4.19: Program execution on 4 nodes of a 4 node, 2-way organisation

Next, the execution of the program is reorganised to use all 4 execution nodes in a 4 node, 2-way organisation. Here, all the nodes are to be utilised as much as possible. This is shown in Figure 4.19. To begin with, an addition takes place in each node. Then the results of SRF1 and SRF2 are written to shared registers. Then, two more additions are performed in parallel in nodes 0 and 3 and then the final addition takes place in node 0 after a shared write from SRF3 at time step t_6. This time the execution time can be calculated to be 10.3337 time units.

Figure 4.20 shows the execution of this program using 4 nodes of a 4-way system. It begins similarly to the previous example, but the different communication pattern chosen makes the distribution of instructions different. Nodes 0 to 3 form a square with node 0 in the top right and other nodes numbered clockwise. If

time	SRF0	SRF1	SRF2	SRF3
t_0	OF1(n1)	OF1(n3)	OF1(n5)	OF1(n7)
t_1	OF2(n2)	OF2(n4)	OF2(n6)	OF2(n8)
t_2	ADD(n1+n2)	ADD(n3+n4)	**[SW]** ADD(n5+n6)	**[SW]** ADD(n7+n8)
t_3	OF1(n1+n2)	OF1(n3+n4)		
t_4	**[SR]** OF2(n7+n8)	**[SR]** OF2(n5+n6)		
t_5	ADD(n1+n2+n7+n8)	**[SW]** ADD(n3+n4+n5+n6)		
t_6	OF1(n1+n2+n7+n8)			
t_7	**[SR]** OF2(n3+n4+n5+n6)			
t_8	ADD(n1+n2+...+n8)			

Figure 4.20: Program execution on 4 nodes of a 4-way organisation

this 4-way organisation has 4 shared and 20 local registers, then the execution time can be calculated to be 11.9781 time units.

The execution times (in time units) for executing this program on all these different organisations are shown in Table 4.4.

organisation	*exec. time*	*speedup*
MRF 24 regs.	16	1
4xSRFs 2-way, 8 sh., 16 loc. (2 used)	13.5613	1.18
4xSRFs 2-way, 8 sh., 16 loc. (4 used)	10.3337	1.55
nxSRFs 4-way, 4 sh., 20 loc. (4 used)	11.9781	1.34

Table 4.4: Execution times for different organisations

By increasing the parallelism of the datapath and using an SRF, the execution time of the program has decreased, under the assumptions considered. Hence, as can be seen in the table, when using 2 nodes of a 4xSRF processor rather than a single node with an MRF with the same number of registers as each of these nodes, a speedup of 1.18 can be calculated. When all 4 nodes of this system are used, a greater speedup of 1.55 can be calculated. When moving to a 4-way organisation though, although the available degree of register sharing increases, this cannot be exploited by this program and the higher cost of the 4-way drops the speedup to 1.34.

These execution times are a bit too optimistic, however, because they assume that connecting SRFs together does not significantly increase their cost and latency parameters. This example was presented simply to illustrate the tradeoff between using an SRF organisation which makes a certain degree of sharing and effective parallelism available and the cost that such an organisation puts on the execution time of program operations.

4.3 A Four SRF 2-way System

So far, only unconnected SRFs have been considered. In this section the effects of connecting SRFs together in a complete organisation will be studied.

As mentioned in the implementation section, connecting SRFs into a 2-way system involves connecting the following signals for each pair of SRFs:

- data busses, input and output, of the shared registers of SRF(n+1) to SRF(n).
- completion signals of the shared registers of SRF(n+1) to the completion signals of SRF(n).
- register select signals for columns of SRF(n) to the columns of the shared registers of SRF(n+1).
- register select signals for rows of SRF(n) which access the shared registers of SRF(n+1) to these registers.

The distances that these lines have to travel will affect the access times of each SRF in the system differently.

The SRF system implemented (but not fabricated) using ES2's 0.7μm process, is a 4 SRF, 2-way with 16 local and 2 pairs of 8 shared registers per SRF. This system was chosen because the number of shared registers is large enough to be realistic for an architecture and the number of SRFs is large enough to illustrate the SRF interconnection issues. Each SRF has 24 physical registers and the system has a total of 96 registers.

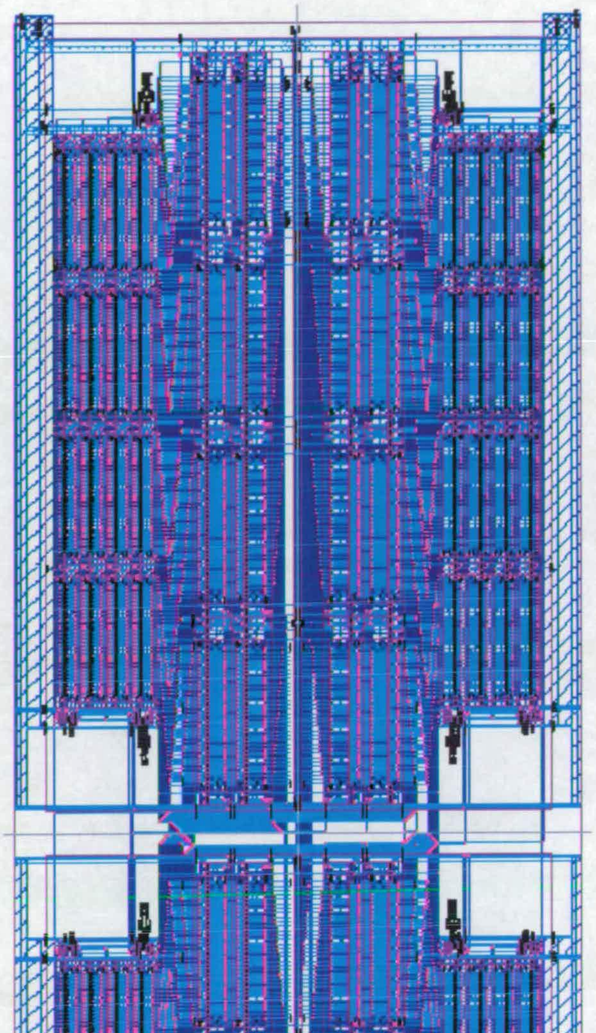


Figure 4.21: Layout of the 4 SRF 2-way System

The layout of the 4 SRF 2-way system is shown in Figure 4.21. The SRFs are numbered 0 to 3, SRF0 is the top right SRF and the SRF numbers increase clockwise. SRF1 is the reflected image of SRF0 about the x-axis, SRF2 is the reflected image of SRF0 about the x and y-axis and SRF3 is the reflection of SRF0 about the y-axis.

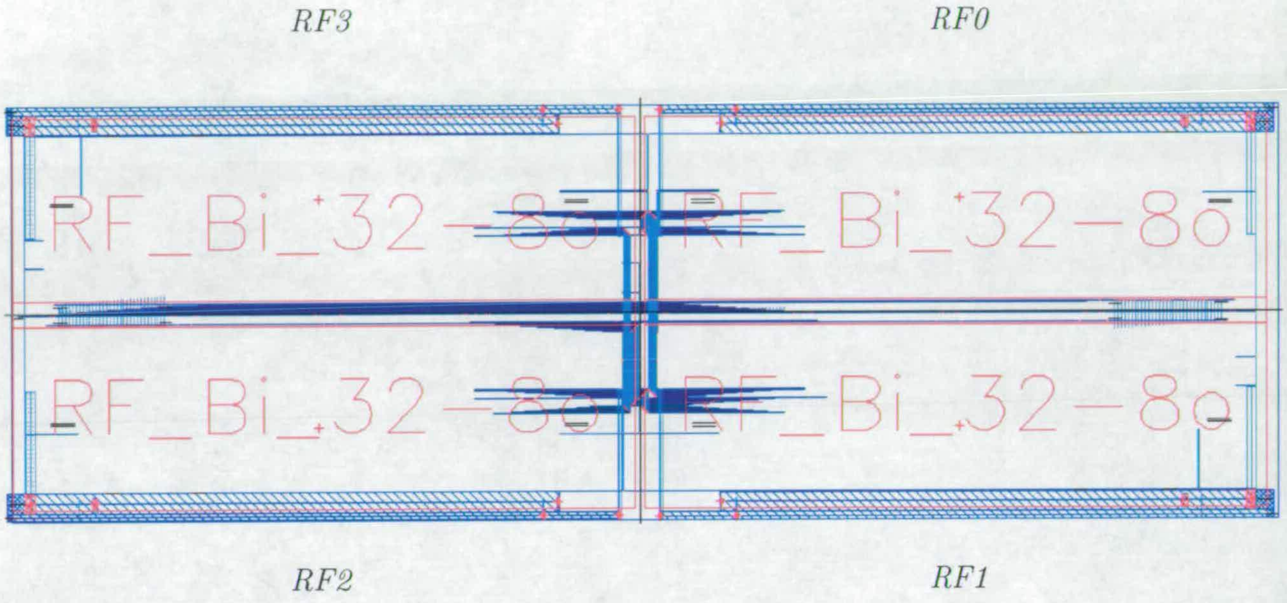


Figure 4.22: Top level Connectivity of the 4 SRF 2-way system

Figure 4.22 shows the top-level connectivity of the layout. The metal lines in light and dark blue (`METAL2` and `METAL1` respectively) show the numbers of signals and how they connect between SRFs. The thick `METAL1` lines on the top and bottom of the picture are the `VDD` and `VSS` tracks. The vertical `METAL2` lines on the right between SRF0 and SRF1 connect the inputs of the shared registers of the latter to the bus of the former. Similarly, the `METAL2` lines on the left of the picture connect the inputs of the shared registers of SRF3 to SRF2. The horizontal `METAL1` tracks at the centre of the picture connect the input bus of the shared registers of SRF0 to the bus of SRF3. The `METAL1` and `METAL2` lines at the periphery of the SRFs connect the row and column register select signals. The rightmost set of vertical `METAL2` lines at the centre of the picture connect the output bus of the shared registers of SRF1 to SRF0 along with their read and write completion signals to SRF0. On their left and at the bottom are the connections between the outputs of the shared registers of SRF2 and SRF1's bus, along with their write and read completion signals, whereas at the top the connections between the outputs of the SRF0's shared registers and SRF3's bus, along with their write and read completion signals. There are two more sets of `METAL2` lines to the left. The first set connects the inputs of the shared registers of SRF2 to the bus of SRF1. The second one connects the outputs of the shared registers of SRF3 to the bus of SRF2. Thus, all the busses of the shared registers, their completion signals and the register select signals are connected.

The access times, from HSPICE, for each SRF in the system are shown in Table 4.5. In a system of SRFs, shared register accesses are slower when the shared register resides in another SRF rather than where the access was initiated. The access distance of a shared register in this case is greater than when the shared register resides in the same SRF. In the table local register access times are labelled l-read and l-write respectively. Shared register access times are labelled s-read-n, s-write-n for near accesses and s-read-f and s-write-f for far accesses.

The access times are calculated in the same way as for the unconnected SRFs, *i.e.* by averaging the access times of the four corners of a portion of the SRF (local, shared-near or shared-far) accessed in a fixed order.

SRF	l-read	l-write	s-read-n	s-write-n	s-read-f	s-write-f
0	44.72	47.81	46.84	50.90	48.57	51.22
1	55.22	54.52	55.47	56.94	60	56.55
2	45.86	47.99	47.11	50.67	49.02	51.46
3	55.45	54.11	55.88	57.05	59.55	57.06

Table 4.5: Access Times(ns) for 4 SRF 2-way System

Table 4.6 shows the cost and latency performance metrics for each SRF in the system relative to the reference MRF, *i.e.* with 24 registers. Contrast these metrics with those of an unconnected SRF. These were shown in table 4.3 and are repeated in table 4.7.

From these two tables it can be observed that the cost metric has changed significantly. For SRFs 0 and 2, the cost has increased by 3 and 3.2 times respectively. For SRFs 1 and 3 it has increased by 5.15 and 5.12 times respectively. The latencies for near accesses which are comparable to the latency of the unconnected SRF have remained close to their unconnected value for SRFs 0 and 2, with a difference of 0.57 and -0.88 respectively. For SRFs 1 and 3, they have decreased with differences of -2.82 and -1.93 respectively. The latencies for far accesses are, for SRFs 0 and 2, 1.41 and 1.7 times greater than the near latencies respectively. For SRFs 1 and 3, they are 2.8 and 2.07 times greater respectively.

The increase in the cost was expected. As the shared register data busses and completion detection signals of one SRF are connected to the bus of another, the loading of the bus and the completion signals increases the latter's access time. The distances of the register select signals, for both rows and columns, from

SRF0		*reads*	*writes*
	Cost(%)	38.24	34.61
	Latency-near(%)	4.75	6.47
	Latency-far(%)	8.61	7.14
	Norm-Lat-near(%)	44.8	43.3
	Norm-Lat-far(%)	50.14	44.21
SRF1		*reads*	*writes*
	Cost(%)	70.7	53.5
	Latency-near(%)	0	4.44
	Latency-far(%)	8.66	3.73
	Norm-Lat-near(%)	71.47	60.31
	Norm-Lat-far(%)	85.48	59.21
SRF2		*reads*	*writes*
	Cost(%)	41.77	35.11
	Latency-near(%)	2.73	5.59
	Latency-far(%)	6.9	7.24
	Norm-Lat-near(%)	45.63	42.66
	Norm-Lat-far(%)	51.54	44.88
SRF3		*reads*	*writes*
	Cost(%)	71.41	52.34
	Latency-near(%)	0.78	5.44
	Latency-far(%)	7.4	5.46
	Norm-Lat-near(%)	72.74	60.62
	Norm-Lat-far(%)	84.09	60.65

Table 4.6: Performance Metrics for 4 SRF 2-way System

2 way, 8 shared, 16 local	*reads*	*writes*	*average*
Cost(%)	12.25	11.89	12.07
Latency(%)	4.03	6.04	5.04
Norm-Lat(%)	16.76	18.64	17.7

Table 4.7: SRF Metrics for an unconnected SRF of this system

registers also has an impact on the access times of far registers. The distance that these have to travel depends on the distance between two SRFs and their relative position. Due to their relative position, *i.e.* facing each other, the SRF0, SRF1 and SRF2, SRF3 pairs can be connected efficiently. The shared registers' data busses, their completion signals and the column register select signals of SRF1 and SRF3 connect vertically to SRF0 and SRF2 respectively. The row select signals also travel mostly vertically but at the periphery of the SRF pairs. Hence, the connections between these pairs are relatively short and efficient. For the other two pairs though, *i.e.* SRF1, SRF2 and SRF3, SRF0 the busses and completion signals cannot be routed so easily and longer routes between the horizontal pairs are required. These can be seen in Figure 4.21 between the SRFs at the centre of the figure. The column register select signals can be routed horizontally. The row select signals have to travel significant distances for these pairs. The write port is on the left and the read port on the right of an SRF. As the system is symmetrical about the x and y axes, the ports near the centre are write ports and the ports at the outside are read ports. Hence, the read register select signals travel quite significant distances between the horizontal pairs.

SRF pair	*read*	*write*
0, 1	1557.6	1525.0
1, 2	3258.6	1291.4
2, 3	1557.6	1550.0
3, 0	3220.0	1215.2

Table 4.8: Average Interconnect Distances for control and data signals (in μm) between SRFs

Table 4.8 shows the approximate average interconnect distances that control and data signals travel between the clusters. It can be seen that for reads, the SRF1 to SRF2 and SRF3 to SRF0 connections are more than twice as long as the SRF0 to SRF1 and SRF2 to SRF3 connections. For writes, the differences in ratios in the order of 1.2 to 1.3.

The SRFs have inhomogeneous access times due to the interconnect delays depending on their physical position. The layout organisation of the SRFs implies that two SRFs placed vertically opposite to each other, like SRF0 and SRF1 in this system, can be routed together with short connections, whereas SRFs placed horizontally opposite, like SRF1 and SRF2, require long connections. These connections are directional. For example, SRF0 can read and write to SRF1's shared section but not vice versa. This stems from the physical organisation of the 2-way sharing scheme. So, an SRF with long connections to its neighbour will have a slow access time. For example, SRF1 requires long connections to SRF2, hence its slow access time. On the other hand, SRF2 has short connections to SRF3, hence it will be faster. Second order capacitive effects imply that the large capacitance of the SRF1 bus will affect the access time of SRF2, due to sidewall capacitances between them, but the first order factor is the length and the loading of each SRF's bus.

The assumption about the cost of an SRF increasing only slightly when con-

organisation	*exec. time*	*speedup*
MRF 24 regs.	16	1
4xSRFs 2-way, 8 sh., 16 loc. (2 used)	19.7960	0.81
4xSRFs 2-way, 8 sh., 16 loc. (4 used)	12.7416	1.26

Table 4.9: Updated Execution times for SRF system

nected to an SRF system does not consider the placement and routing effects. In Section 4.2.2, the effect of the SRF organisation on the execution of a program was investigated. Now that accurate cost and latency values for the SRF system are available, the execution time of the program can be more realisticaly estimated. In that section, to calculate the execution time it was assumed that the nodes were executing instructions in lockstep and in the case of a slow operation taking place in parallel with a faster one, both must have finished, before the next program instruction could be executed, *i.e.* the longest operation dominated. Because of the fact that SRFs in a system are found to have inhomogeneous access times, this assumption must be relaxed if the execution time is not to be overestimated.

The new execution times for the program are shown in Figure 4.9. Due to the increase in the cost of register accesses, it is now slower to execute the program on 2 SRFs of this system compared to the MRF with 24 registers. It is still faster though to execute the program on 4 SRFs. So, if enough parallelism is available, the penalty of using shared registers is redeemed.

4.4 Bus-Based Systems

An alternative, and more conventional approach to segmenting the MRF is a bus-based, multiple RF system. Such a system is composed of a number of MRFs which can communicate through a set of busses. The number of busses and the bus interconnections can vary depending on the design of such a system. In some designs, the busses are shared only between a certain number of RFs in a system, whereas in others, like the limited-connectivity VLIW approach, the RFs are connected to busses through a crossbar, allowing any RF to write to any other RF's bus. In a system with multiple devices such as RFs sharing a common bus, arbitration is required to stop the multiple devices using the single bus simultaneously. To compare an SRF system with a bus-based system, the two systems have to be equivalent.

To reason about this equivalence, an architecture in which these systems can be connected must be considered. RFs in an SRF system or in a bus-based system accommodate a certain number of FUs in a particular architecture. For example, it could be that each node of such a system was connected to a set of multiple FUs to form a clustered architecture. On the other hand, it could be that the nodes shared a single set of FUs, where control was implemented using a scoreboard. The important aspects about equivalence of the two systems are the total number of physical registers that they contain, the number of ports that these registers have, the degree of inter-RF communication, and finally that they have to be interchangable in an architecture. The first two are obvious. The degree of inter-RF communication is the connectivity that such a bus-based or SRF system provides between RFs. The interchangeability property means that for two systems to be equivalent it should be possible to exchange one for the other in an architecture without modifying any other aspects of that architecture. For example, an SRF system with 4 SRFs is not interchangable with a bus-based system with 2 SRF because they cannot be exchanged in an architecture without modifying other aspects of it.

It can be concluded that the 4 SRF 2-way organisation with 16 local and 8 shared registers presented in the previous section is identical with a 4 MRF bus-based system where each MRF has 24 registers, the same number of ports as an SRF's local section in the SRF system and a bus connectivity to match the achievable degree of communication of the SRF system. Such a bus system in shown in Figure 4.23.

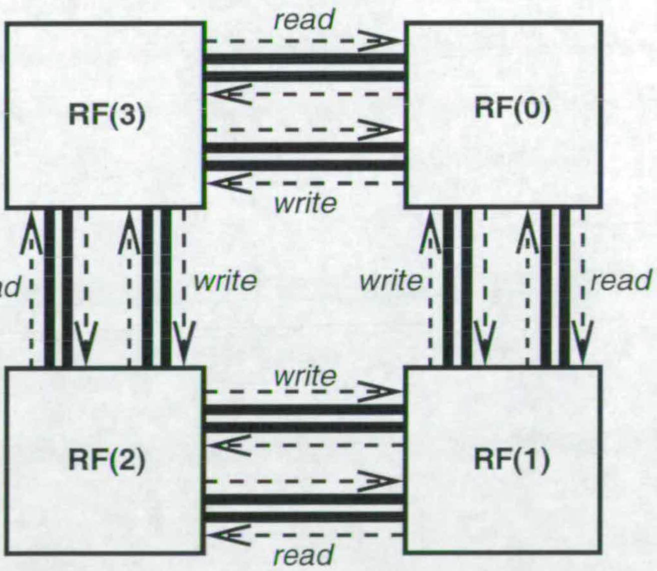


Figure 4.23: Four MRF 2-way Bus System

In this system each RF can communicate with its two neighbours, enabling data in such a system to flow in both directions, as in the 2-way SRF system.

Because of this, two busses per operation are required, making a total of four busses between each pair of RFs necessary. An access to an RF can come from three possible sources, the two internal ones from its neighbours and the external port. This implies that three-way arbitration is required to sink these three sources into a single RF port. The complexity of this system, particularly the large number of busses and the three-way arbitration makes it hard to implement. A unidirectional bus system, a simpler alternative allowing unidirectional flow of data, is shown in Figure 4.24. This simpler system was designed and laid out in order to compare its performance with the 4 node, 2-way SRF system, Section 4.3.

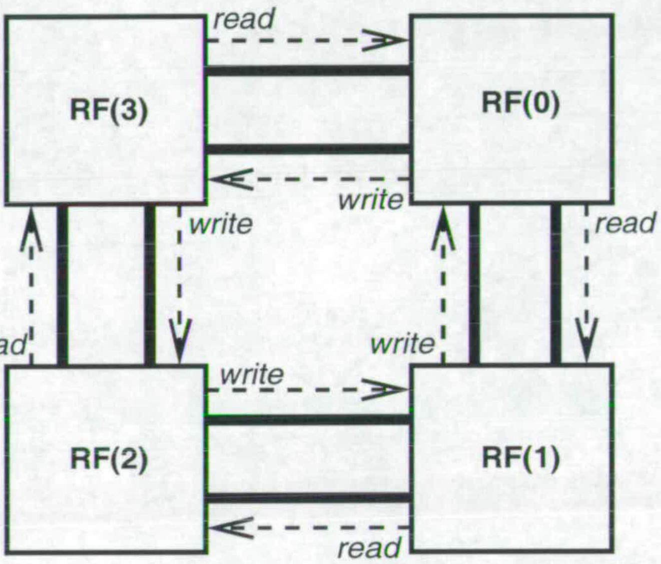


Figure 4.24: Four MRF Unidirectional Bus System

In this system, a maximum of two accesses per port are possible, requiring two-way rather than three-way arbitration and two rather than four busses are required per RF pair. The data busses and the register index lines of these two data sources must be multiplexed so that the selected ones will pass onto the RF's busses and the RF's register index lines. The multiplexing of the data busses and of the register index lines is implemented using tri-states. In addition, each of these data busses has a register index bus for identifying which register is to be to read. The two data busses and the two register index busses are connected to tri-state elements which enable the ones which are allowed as a result of the arbitration. 5 tri-state elements are required for the register index lines and 32 for the data busses, all fed by the result of the arbitration. The large number of tri-states for the data busses means that buffering of the select signal is required otherwise it will change state very slowly.

There are two main differences between a bus-based system that allows bidirectional flow of data and a 2-way system of SRFs. Firstly, in a bus based system, the value of any of the registers of an RF can be communicated, whereas in an SRF system only the shared registers can be communicated between SRFs. Secondly, in a bus based system, an access from one RF to another blocks both. Neither the RF which initiated the access nor the RF that actually performs the access can be used for another access until the first access has completed in both.

4.4.1 Implementation

The unidirectional bus system was implemented to compare its performance with the SRF system. Connecting the MRFs into a bus system involves connecting:

- input data bus of RF(n+1) to RF(n) through tri-states.
- output data bus of RF(n) to RF(n+1) through tri-states.
- read register index bus of RF(n+1) to RF(n) through tri-states.
- write register index bus of RF(n) to RF(n+1) through tri-states.
- local handshake signals and external handshake signals from neighbouring RFs to arbitration circuits.

Each RF in this system has two request inputs and a single acknowledgement output. One request is for local accesses and the other for remote ones. Only one request should take part in the handshaking protocol.

The additional circuitry required to implement the bus system includes the bus control circuits, one per RF port, which "glue" onto the RF ports, and the tri-stating of the busses. Each bus control circuit is composed of a 2-way mutual exclusion element (*c.f.* Figure 3.2), the arbitration control circuit and a complex CMOS gate for producing the RF acknowledgement signal.

The 2-way mutual exclusion element arbitrates between an access from the local port and one coming from the neighbouring RF. The request signals of the local and remote handshakes are connected to the two inputs of the mutual exclusion element. The result of the arbitration, *i.e.* the outputs of the mutual exclusion element selects which of the two sources will drive the input or output busses, depending on the type of access, and the register index busses. The mutual exclusion element outputs are also fed into the arbitration control circuit, which registers whether the access is local or remote, performs the handshaking with the RF port and generates an acknowledgement signal when the access has completed. This acknowledgement signal must assert the output acknowledge of this RF, in the case of a local access, or the acknowledgement signal of the neighbour in the case of a remote access. This is achieved by the acknowledgement gate.

The FSM and layout of the arbitration control circuit are shown in Figures 4.25 and 4.26 respectively.

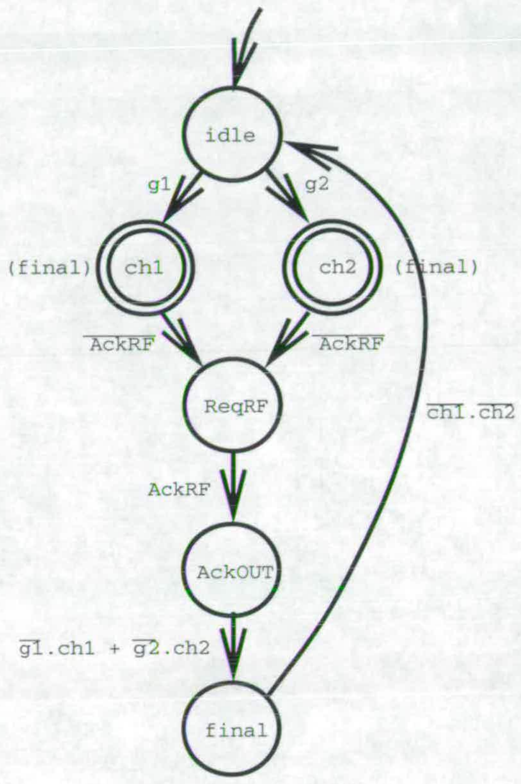

idle
g1
g2
(final)
ch1
ch2
(final)
$\overline{\text{AckRF}}$
$\overline{\text{AckRF}}$
ReqRF
$\overline{\text{ch1}}.\overline{\text{ch2}}$
AckRF
AckOUT
$\overline{\text{g1}}$.ch1 + $\overline{\text{g2}}$.ch2
final

Figure 4.25: Arbitration control circuit FSM for Bus System Ports

The gate that produces the RF acknowledgement signal is shown in Figure 4.27.

The machine resets into state `idle`. This is indicated by the incoming arrow with no label. When an access is initiated, then depending on the outcome of the arbitration, one of the two mutual exclusion element outputs, `g1` or `g2` will be asserted. This will put the FSM into either state `ch1` or `ch2`. As the two inputs `g1` and `g2` are mutually exclusive, there is no race between states `ch1` and `ch2` and only one will be entered.

States `ch1` and `ch2` are persistant states and are only left when state `final` is finished. This is the mechanism by which the arbitration control circuit registers whether the access is a local or a remote one; `ch1` indicates a local access, whereas `ch2` indicates a remote access. Then, the handshake with the RF port takes place,

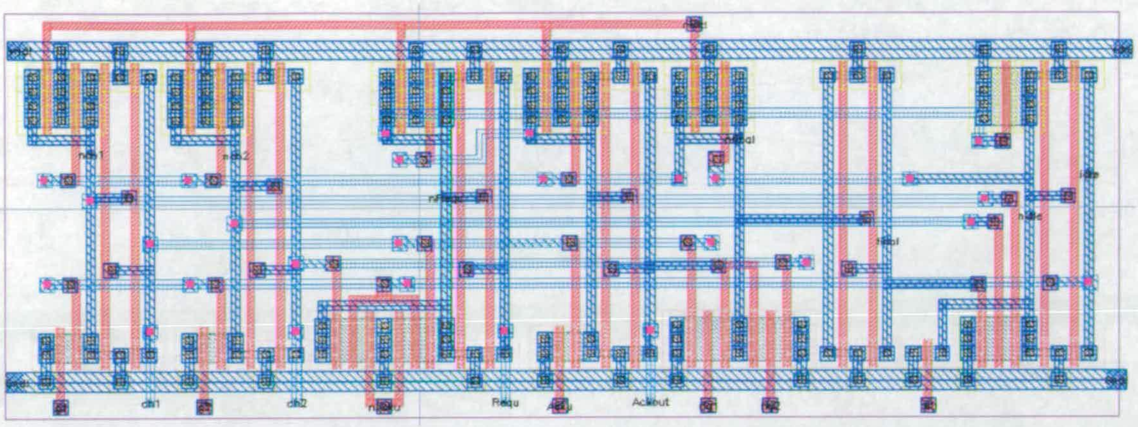


Figure 4.26: Arbitration control circuit layout for Bus System Ports

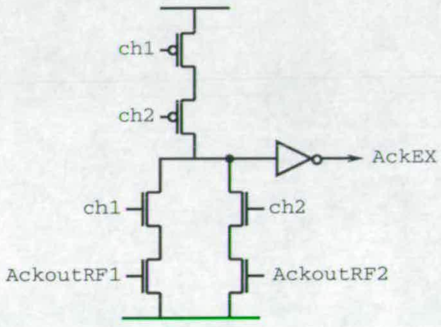


Figure 4.27: RF acknowledgement gate

i.e. signals Requ, Acku. When signal Acku has been asserted, then the local or remote access can be acknowledged. This is achieved by signal AckOUT and the complex gate. Depending on whether ch1 or ch2 are asserted, that identifies the acknowledgement signal, *i.e.* AckoutRF1 or AckoutRF2, that will assert the acknowledgement signal of the RF. When the acknowledgement has been received, the request signal will become deasserted, dropping signal g1 or g2. The transition to state final can then take place. The expression $\overline{g1}.ch1 + \overline{g2}.ch2$ ensures that the appropriate request signal is deasserted, as it is possible that the other one is now asserted. State final is necessary to deassert the two persistant states ch1 and ch2, which also deasserts the acknowledgement signal. When both states ch1 and ch2 are left, the FSM returns to state idle.

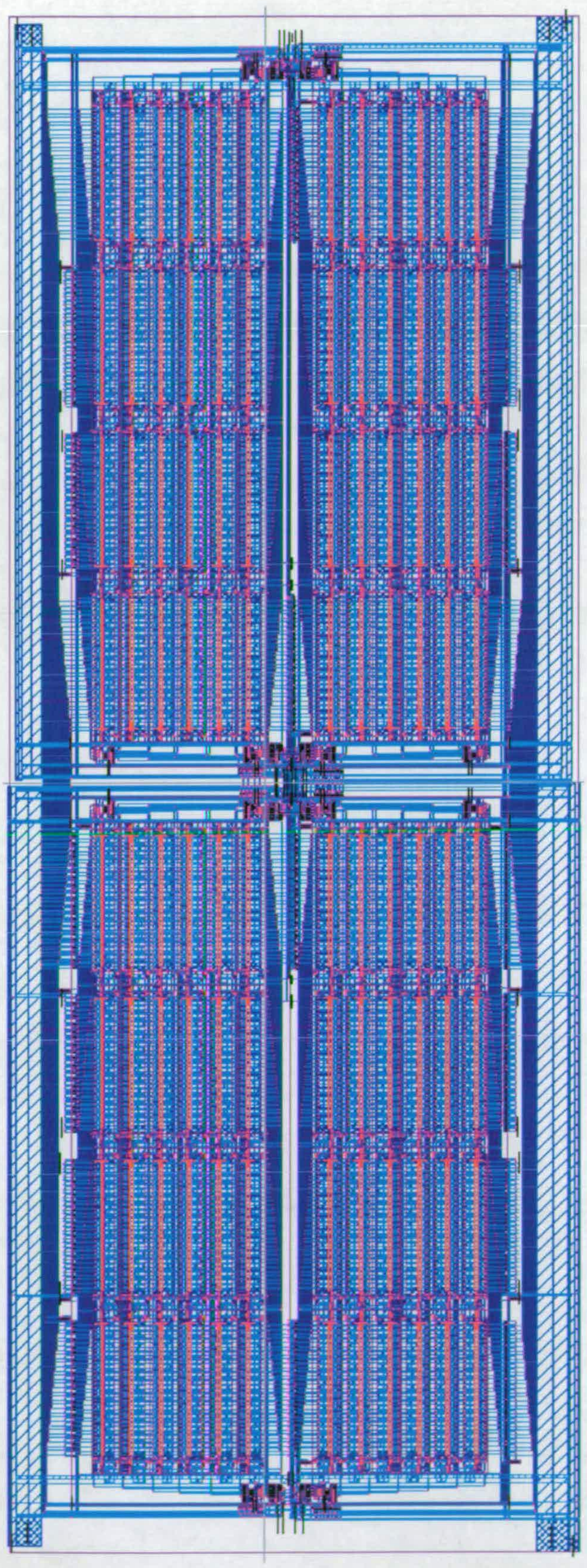


Figure 4.28: Layout of the 4 MRF Arbitrated Bus System

The layout of a four RF shared bus system is shown in Figure 4.28. This has the same number of total registers per RF as the SRF system presented in Section 4.3. As in the SRF system, the RFs are numbered 0 to 3, RF0 is the top right RF and the RF numbers increase clockwise. Each RF in the bus system has 24 registers and the bus connectivity allows each RF to read from its successor and write to its predecessor. The RFs are symmetrical to each other as in the SRF system, RF1 is the reflected image of RF0 about the x-axis, RF2 is the reflected image of RF0 about the x and y-axis and RF3 is the reflection of RF0 about the y-axis.

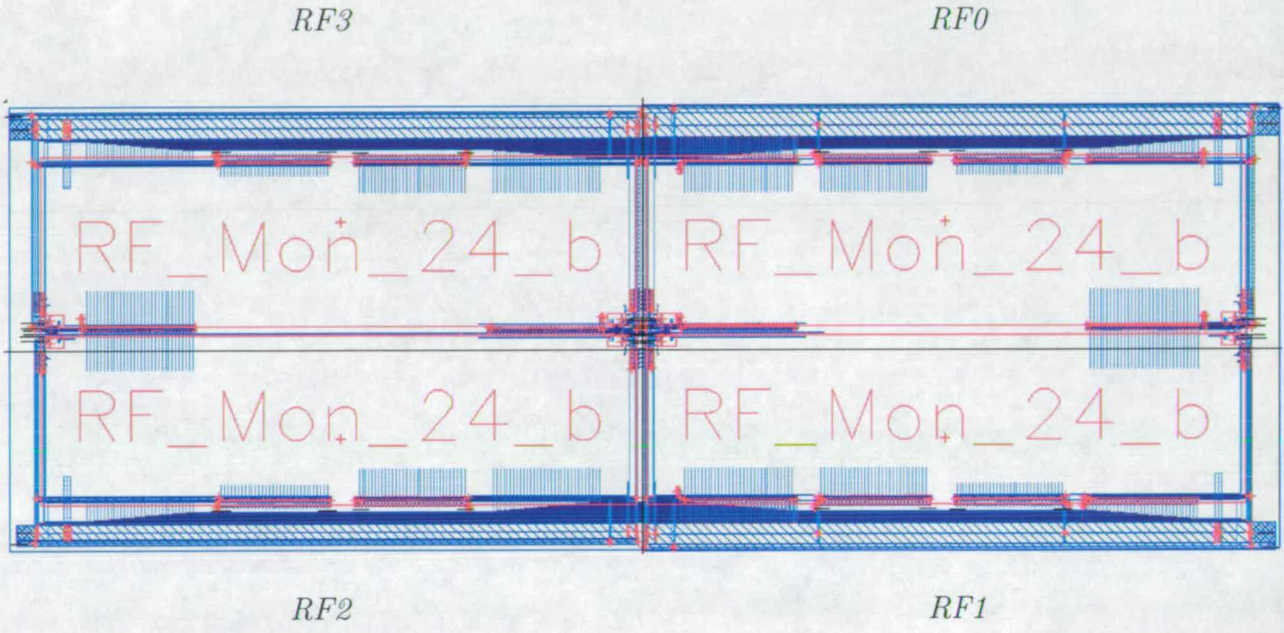


Figure 4.29: Top level Connectivity of the 4 MRF Arbitrated Bus system

Figure 4.29 shows the top-level connectivity of the layout. This highlights the tri-state buffers and the bus routing. Comparing this with Figure 4.22, which shows the top-level connectivity of the 4 SRF 2-way system, it can be seen that a lot more routing is required for the bus system and this hints that interconnect delays may affect the performance of the bus system to a greater extent.

The loading of the tri-states onto an RF's bus is another potential problem. The inputs and outputs of the registers already use pass transistors. Connecting tri-states to the input and output busses connects another set of pass transistors in series. Pass transistors are slow in the first place, because they route a signal through the transistor channel.

A total of 16 sets of tri-states is necessary for the data and register index busses, *i.e.* 2 sets of 8, one for reads and the other for writes. So, the design contains 8 read and 8 write busses. The tri-states for the data busses contain a buffer of strength 32 and an extra pass transistor circuit for detecting that the busses have been driven. The buffer amplifies the bus enable signal to drive all the data bus bits. The extra pass transistor circuit is identical to the tri-states, except that its output is held low with an extra pull-down transistor, rather than floating when its enable signal is low. Its input and enable signals are shorted to the bus enable signal, the strength 32 buffer's output, and it is placed next to the bit furthest from the control logic to ensure that its delay is an upper bound to the delays of the individual bits. When its output turns high, this signifies that the bus has been driven and its signal is fed back to the control logic.

Each RF has an external and an internal port for each type of access. The former is for accesses from outside the system, the latter for accesses between neighbours. The external port is to connect to other architectural components, FUs for example. For RF0, the data busses of the external ports are at the top of the RF, 2nd and 3rd from the right, the former being the output bus for external reads and the latter the input bus for external writes. These connect to sets of tri-states and onto the RF's bus. The relative position for these is the same for all RFs. The register index busses and the request and acknowledge handshake signals are near the accessed port, at the bottom left of RF0 for the write port and at the bottom right for the read port. The other two sets of tri-states connecting to RF0 are for reads and writes from neighbouring RFs. The bus at the bottom left of RF0 is the write input bus of RF1 connecting to RF0 through tri-states enabling RF1 to write to it. The bus at the bottom right of RF0 is the read input bus of RF1 connecting to RF0 through tri-states enabling RF0 to read from it. The other pairs of RFs in the system are connected in a similar way.

The access times for each RF in the bus system are shown in Table 4.10. These were calculated by averaging the HSPICE access times of the four corners of each RF accessed in a fixed order. In this table the local access times are labelled l-read and l-write. The access times for accessing the registers of a neighbour RF are labelled n-read, for reading from the next RF in the system, and p-write, for writing to the previous RF in the system.

RF	l-read	l-write	n-read	p-write
0	65.59	55.68	60.31	54.94
1	64.99	55.78	143.48	50.25
2	65.50	55.16	60.61	54.13
3	65.51	55.79	128.58	49.99

Table 4.10: Access Times(ns) for 4 MRF Bus System

Table 4.11 shows the cost and latency performance metrics for each RF in the bus system relative to the reference MRF, *i.e.* an individual, unconnected MRF.

RF0		*reads*	*writes*
	Cost(%)	102.76	56.76
	Latency(%)	-8.06	-1.33
	Norm-Lat(%)	86.43	54.68
RF1		*reads*	*writes*
	Cost(%)	100.9	57.04
	Latency(%)	120.78	-9.92
	Norm-Lat(%)	343.53	41.47
RF2		*reads*	*writes*
	Cost(%)	102.48	55.3
	Latency(%)	-7.47	-1.87
	Norm-Lat(%)	87.36	52.4
RF3		*reads*	*writes*
	Cost(%)	102.51	57.07
	Latency(%)	96.28	-10.4
	Norm-Lat(%)	297.47	40.74

Table 4.11: Performance Metrics for a 4 MRF Bus System

Just as in the SRF system, RFs 1 and 3 have different access time characteristics due to their relative position. The first striking observation about these timings is the very long access times for reading from the next RF for RFs 1 and 3. For RF1 to read from RF2, it is 2.2 times slower than a local read, and for RF3 to read from RF0, it is 2 times slower. In addition, reads to the next RF are faster for RFs 0 and 2 to local accesses, not something expected. The reasons for this are the distance of the tri-states from the control logic and the length that the bus data signals must travel between RFs. For RF0, for example, the tri-states for local reads are on its top side, 2nd bus from the right. On the other hand, the tri-states for reading from the next RF are at the bottom right, next to the control logic. In this example, the inter-RF bus is short, connected vertically in `METAL2`. A read access to the next RF will enable the tri-states faster, data on the bus will be valid earlier and will therefore finish sooner. For RF1, the read bus of RF2 has to be routed horizontally from RF2 to RF1 and then connected to tri-states. The bus enable signal and the data lines must travel a significant distance - the data lines have to travel a distance of approximately 10,000μm. This is similar for reads of RF0 from RF3. This is why these accesses are significantly slower. Access times for writes are relatively uniform but all writes to the previous RF are slightly faster than local writes for the same reason as the reads.

The cost for reads is higher than for writes and on average almost double that of the SRF system. The cost for writes is slightly less than 30% slower compared to the SRF system. Using a bus system increases the cost of register accesses more than using shared registers. The latencies for remote accesses are mostly negative except for reads from RF1 and RF3.

The cost and latency metrics can now be used to calculate the execution time for the program considered in Section 4.2.2 using this bus system. The differences between executing the program on an SRF and a bus system are that in a bus system, one access alone is enough to transmit data and a remote access on the bus system blocks both the source and destination RFs. In the SRF system communication takes place by a shared write followed by a shared read, whereas in the bus system a remote read can read any register. A remote access though implies that the destination RF is blocked and cannot access any register, for as long as the remote access takes. The different communication ability means that the allocation of the program to the MRFs of the bus system has to be different.

time	MRF0	MRF1	MRF2	MRF3
t_0	OF1(n1)	OF1(n3)	OF1(n5)	OF1(n7)
t_1	OF2(n2)	OF2(n4)	OF2(n6)	OF2(n8)
t_2	ADD(n1+n2)	ADD(n3+n4)	ADD(n5+n6)	ADD(n7+n8)
t_3	OF1(n1+n2)		OF1(n5+n6)	
t_4	**[RN]** OF2(n3+n4)	XR	**[RN]** OF2(n7+n8)	XR
t_5	ADD(n1+n2+n3+n4)	XW	**[WP]** ADD(n5+n6+n7+n8)	
t_6	OF1(n1+n2+n3+n4)			
t_7	**[RN]** OF2(n5+n6+n7+n8)	XR		
t_8	ADD(n1+n2+...+n8)			

Figure 4.30: Program execution on 4 nodes of a 4 node, MRF bus organisation

Figure 4.30 shows one possible way of executing the example program on a 4 node processor with an arbitrated bus organisation. The bold text in brackets shows remote register accesses through the shared busses, RN stands for read from the next MRF, WP for write to the previous MRF. The X's show the RFs which lock because of a remote access and the operation for which they lock, XR stands for read locked, XW for write locked. Assuming, just as with the SRF system, that the nodes execute instructions without synchronising at every time unit, the execution time for the program can be calculated to be 16.7072 time units. The execution times for the different organisations are summarised in Figure 4.12.

So, the high value of the cost parameter that the bus-based MRF organisation imposes, makes this system slower than the single MRF even though it allows for more parallelism to be exploited.

organisation	*exec. time*	*speedup*
MRF 24 regs.	16	1
4xSRFs 2-way, 8 sh., 16 loc. (2 used)	19.7960	0.81
4xSRFs 2-way, 8 sh., 16 loc. (4 used)	12.7416	1.26
4xMRFs unidirectional, 24 regs. each	16.7072	0.96

Table 4.12: Updated Execution times for SRF system

4.5 Conclusions

In this chapter, the shared register approach was presented, targetted to scalable architectures. SRFs have explicitly defined common register regions used for communication and synchronisation. The degree of sharing can be varied to match a particular architecture. The asynchronous implementations of unidirectional, 2-way and 4-way SRFs were presented, along with the implementation of a 4 SRF, 2-way system. Cost and latency performance metrics were used to quantify the effect of this approach to the architecture and were used to calculate the execution time of a simple program running on an imaginary processor with different RF organisations. It was found that interconnect delays have a significant effect on circuit performance, even for relatively large dimension processes like the one used (0.7μm) and they should not be overlooked. It was shown that the amount of interconnect required for a bus system to establish unidirectional communication between RFs was greater than that required for an SRF system with 2-way communication. The SRF system with 16 shared registers per pair of SRFs performed better than the unidirectional bus system with the same number of physical registers. Through the exploitation of parallelism the SRF system can perform better than a smaller MRF with as many registers as one RF of the SRF system. This demonstrates the performance potential of the SRF approach.

Chapter 5

The A1 Processor Design

In this chapter, the structure and implementation of the A1 prototype processor are described. The A1 prototype processor is a fully asynchronous, micronet-based, shared register file, dual-node architecture. The two processor nodes are decoupled, as they execute different instruction threads. Communication between the two nodes takes place at the register level, using shared register files (SRFs). The A1 processor exploits fine-grain parallelism, at the thread-level by sharing registers and at the instruction-level by allowing independent μoperations to execute concurrently.

5.1 The A1 Prototype Chip

The top-level chip layout is shown in Figure 5.1 and the top-level hierarchy showing the pad connections is shown in Figure 5.2. The chip width and length dimensions are approximately 11.5mm and 10.8mm respectively.

The A1-chip has been designed and laid out (but not fabricated) using ALCATEL's 0.7μm, 5V, 2-level metal, digital CMOS process supplied by EUROPRACTICE-IMEC. The move from the ES2 process, used for the design and layout of the SRFs in Chapter 4, to the ALCATEL process was due to the fact that the former has been discontinued. As with the SRF design and layout, the Cadence Opus tools were used for layout design, design-rule checking and circuit extraction and HSPICE was used for circuit simulation.

As can be seen in Figure 5.1, the A1-chip is symmetrical across the x-axis, the top part being node 0 and the bottom being node 1. The size of the chip is dominated by the SRFs, which are easy to distinguish. The chip width is dominated by their shared section. The other processor units have been placed in the rectangular space between the local and shared registers and above the local registers.

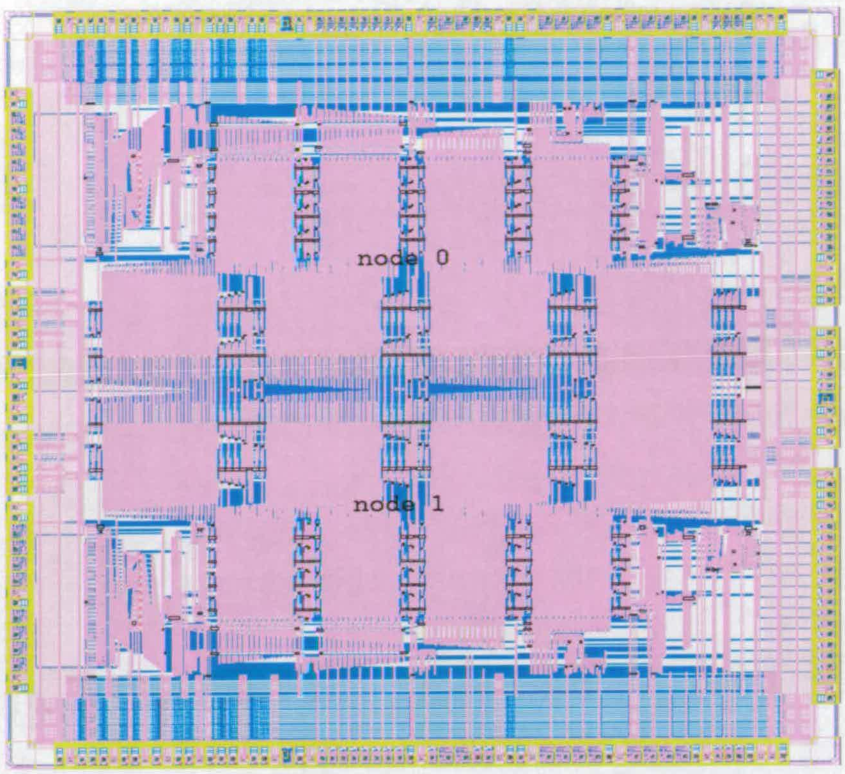


Figure 5.1: The A1 Prototype top-level Chip Layout

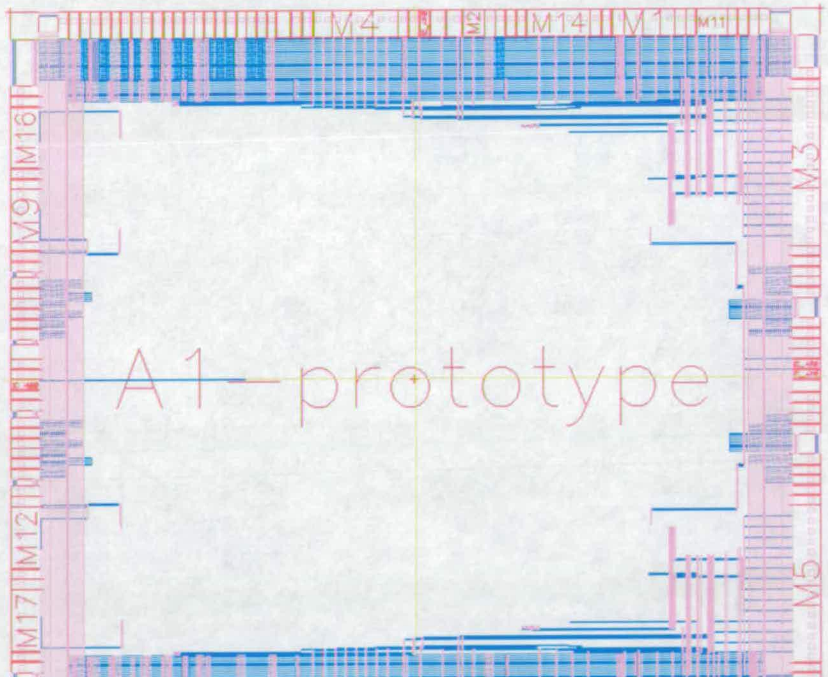


Figure 5.2: The A1 Prototype top-level Hierarchy

The type of SRF used for the A1 is a 2-way scheme with 8 shared and 16 local registers. The shared section of SRF0 is connected to the shared section of SRF1 and vice versa. This enables the 2-way communication of register values. Each node contains three functional units (FUs).

Figure 5.3 shows the layout of a processor node. Its width and length dimensions are approximately 9.5mm and 4.1mm respectively.

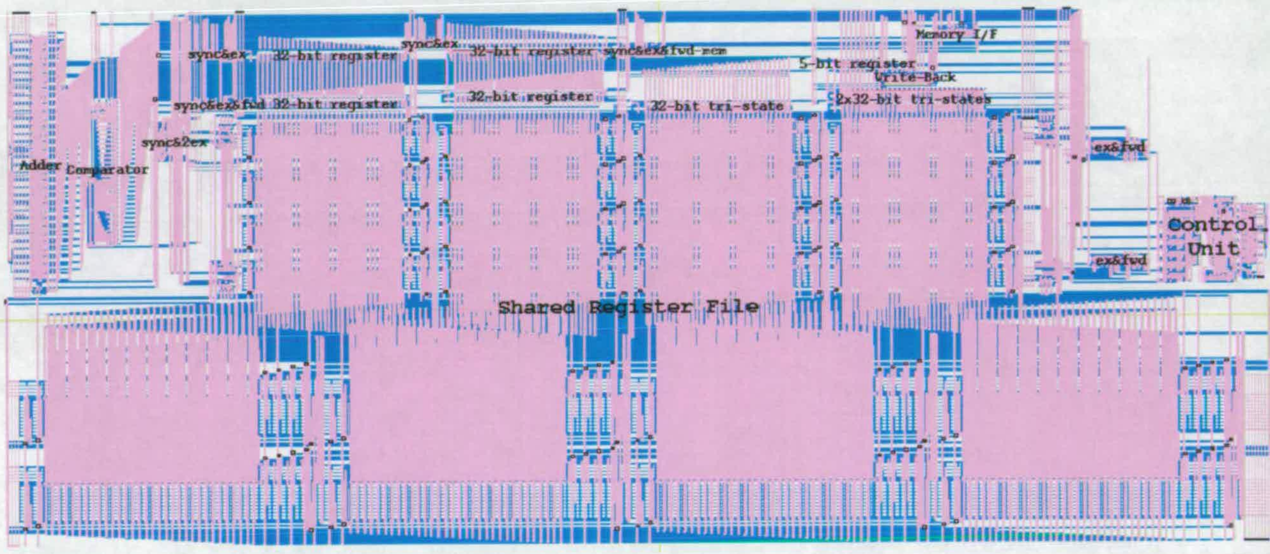


Figure 5.3: Processor node layout

In the next sections, the architectural design and circuit implementation of the A1 processor are detailed.

5.2 The A1 Architecture

Implementing the A1 architecture required mapping the abstract μnet concept into a CMOS implementation. A high-level diagram showing the μnet structure and μinstruction or μoperation stages of each processor node is shown in Figure 5.4.

Each processor node is a scalar processing unit, *i.e.* it fetches and issues a single instruction at a time. Each node is composed of a control unit (μnet control in the diagram) that fetches, decodes and issues instructions into the μnet datapath, and the datapath itself, which contains μblocks, *i.e.* datapath units that implement the μnet μoperations, a 2-way shared RF with 16 local and 8 shared registers per node and 2 read ports and 1 write port, the functional units

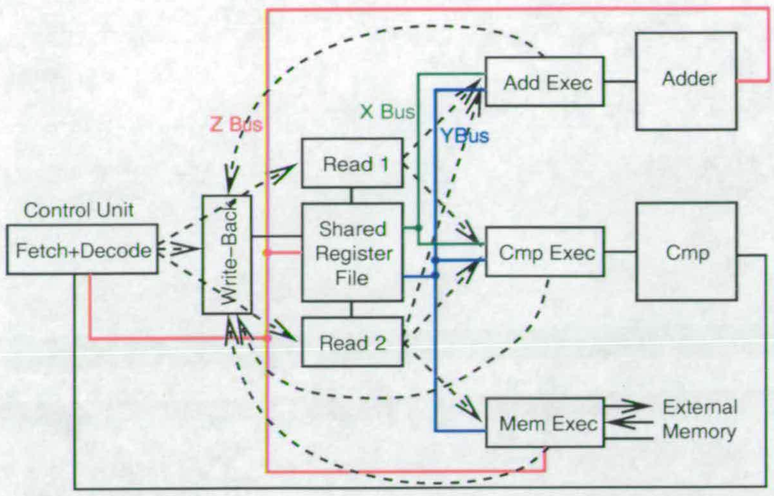


Figure 5.4: High-level Diagram of an A1 node

(FUs) of the node, an external memory interface and a write-back unit. The FUs include a 32-bit adder and a 32-bit comparator.

Instruction execution takes place as follows. An instruction is broken down into μinstructions (μoperations), and these are sent, in parallel, to their corresponding μblocks. A μoperation consists either of control or of both control and data signals and may feed more than one μblock. The control part of a μoperation consists of a handshake, whereas the data part consists of any associated data (a register index for example, or a result). Each μblock consists of control circuitry, possible buffering and access to a datapath operation.

The handshaking protocol between the control unit and the μnet datapath is interpreted in the following way. When the `Ack` signal of a μoperation is asserted, then that μoperation has been received, and is considered issued. When all of the μoperations that make up an instruction have been acknowledged, then the current instruction is considered issued. The next instruction can then be fetched and the control unit will attempt to issue the next instruction's μoperations. For as long as a μoperation's `Ack` signal stays asserted, that signifies that the μblock is busy and a new μoperation that must use that same μblock must wait. When the `Ack` signal is deasserted, a new μoperation can be issued. The μoperations must synchronise with data and with other μoperations.

Figure 5.5 shows the instruction formats.

The A1's instruction set is small and simple, with an instruction width of 17+1[1] bits. Five instructions have been implemented; `LI`, load immediate, `ADD`,

[1]bit 17 is used for memory instructions to distinguish loads from stores.

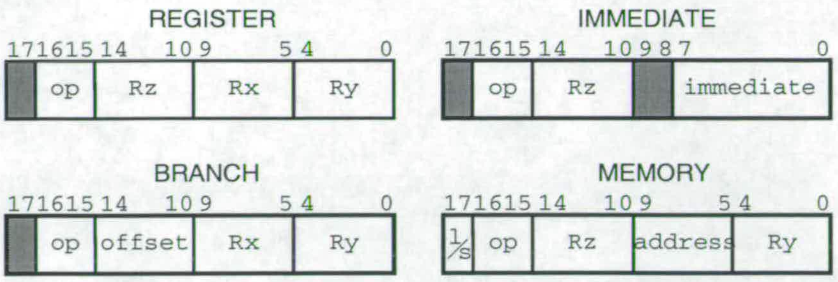


Figure 5.5: A1 Instruction Formats

addition, BEQ, branch if equal and MEM-LD and MEM-ST, memory loads and stores. The instruction opcode is decoded by bits 16 and 15 of the instruction and additionally by bit 17 if the instruction is a memory operation. Five bits are allocated for register accesses, 8 bits are available for immediate values, 5 bits for branches relative to the PC, and 5 bits for the memory address. For relative branches, one of the problems encountered with this instruction format is that backward branching requires a negative value, and the branching field is shorter than the program counter (PC) which is 8-bits wide. To overcome this problem, while keeping the design simple, branches are assumed to be negative and bits 5 to 7 of the offset are set in the case of a branch instruction.

Table 5.1 shows how different instructions are broken down into μinstructions.

		μoperations					
Instruction	opcode (op_{16} op_{15})	Rx	Ry	Wz	AOp	MOp	COp
LI	00	0	0	1	0	0	0
ADD	01	1	1	1	1	0	0
BEQ	10	1	1	0	0	0	1
MEM-LD (op_{17}=0)	11	0	0	1	0	1	0
MEM-ST (op_{17}=1)	11	0	1	0	0	1	0

Table 5.1: A1 μinstruction Decoding

The number of μoperations in each instruction varies depending on the instruction itself. Six different μoperations are defined. Rx and Ry read a register from the first port of the SRF into the XBus and from the second one into the YBus respectively. Wz writes results back into the SRF. AOp performs an addition. MOp uses the memory interface to read or write data from/to an external memory. COp performs a bitwise compare and is used as part of the BEQ instruction.

A detailed architectural diagram of a node's datapath is shown in Figure 5.6 (the control unit that issues the μoperations to the datapath is not shown).

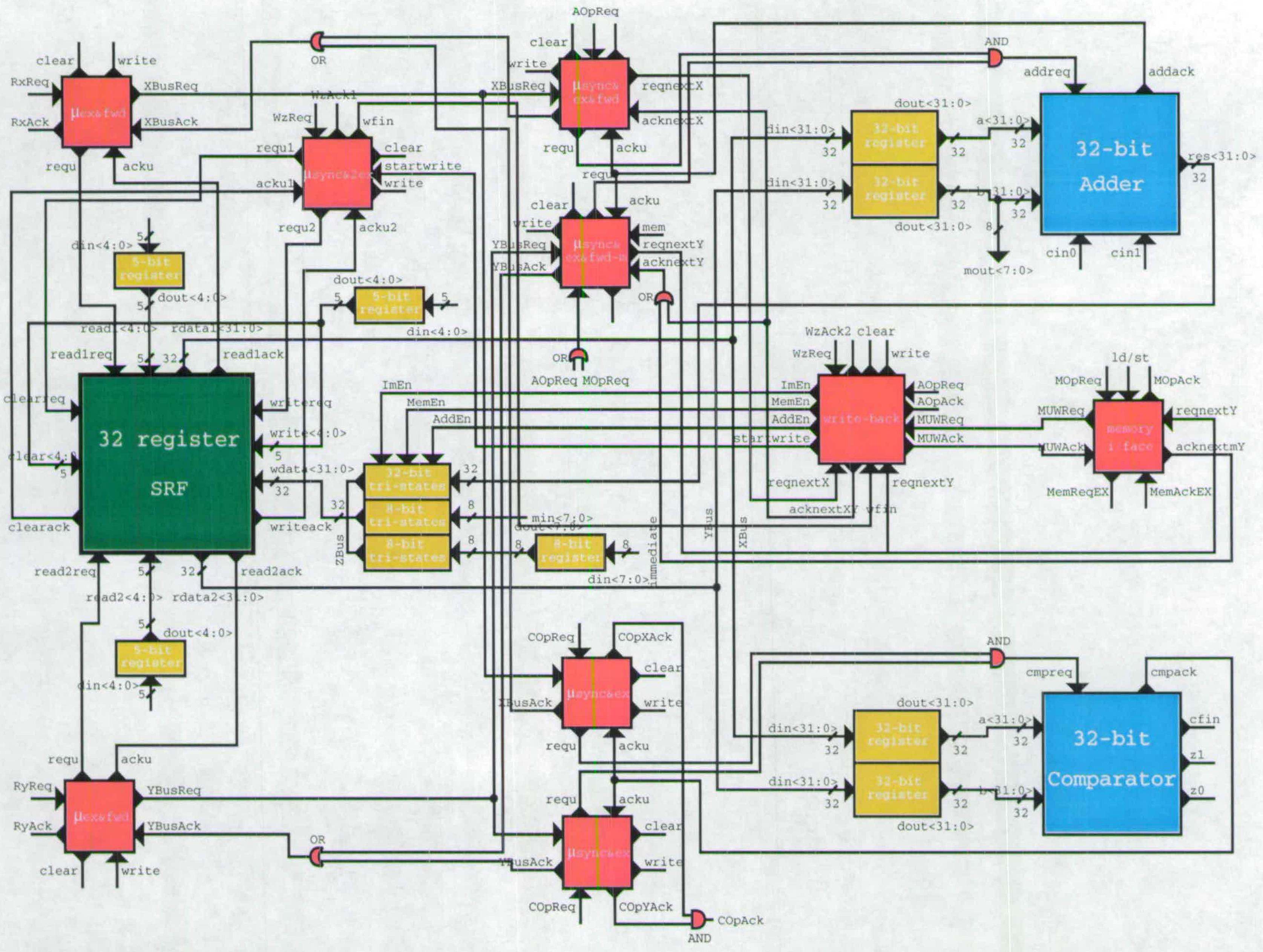


Figure 5.6: Datapath Architecture of one Node of the A1 processor

The diagram shows the two functional units (FUs) of the architecture, the 32-bit adder and the 32-bit comparator in blue. Both of these FUs implement completion detection, and therefore, the time required to perform these operations is dependent on the input data. Both units have 32-bit inputs; the adder has two dual-rail coded carry inputs and a 32-bit result output, whereas the comparator has a completion signal output, `Cfin`, that is asserted when the comparison has completed, and two dual rail coded outputs `z1` and `z0`, which determine whether the branch is to be taken or not. These three outputs connect to the control unit in order to modify the value of the PC.

The SRF is shown in green. The two read ports are labelled `read1` and `read2`, the write port `write` and the clear port `clear`. Each port consists of two handshake signals, a 5-bit long register index, and except for the clear port, a 32-bit input or output.

As μoperations carry data with them and parallelism between independent μoperations is to be exploited, decoupling registers are necessary to separate the data that μoperations carry, between the different μblocks and between μblocks and the control unit. The decoupling registers store a μoperation's attached data. They are similar in functionality to pipeline registers. For the write-back, tri-state elements are necessary to implement the multiplexing and to select the source of the write: the adder result, the memory input or the immediate value. In the diagram the decoupling registers and the tri-states are shown in yellow.

Decoupling registers are used in three sections of the datapath, in the operand fetch part, for storing the register indices, in the execute part, for storing the input data of the FUs, and finally in the write-back part for storing the value of the immediate. In this way, when all the μoperations of the current instruction are issued and the next one is fetched, the new instruction's fields will not interfere with the register indices. In addition, an instruction using another FU does not interfere with an issued instruction because the FU inputs are buffered. Separate buffers are used, in order to allow the forwarding of the two operands to the FUs to take place in parallel. When both registers have been loaded, the operation can begin. In practice, the amount of parallelism that can be exploited depends on the speed of the μoperations, the instruction order and the type of μoperations required by each instruction. As the instruction set implemented is very simple, and there are only two FUs and only two read ports, the amount of parallelism exploited is limited.

The control circuits are shown in the diagram in red. These include a total of seven μcontrol circuits, for sequencing the actions of μoperations, and the write-back and memory interface units. There are three control circuits for each port of the SRF, one μ`ex&fwd` (execute and forward) unit per read port and a μ`sync&2ex` (synchronise and execute two operations) for the write port. There are two control circuits per FU, one per operand, one μ`sync&exec&fwd` (synchronise and execute and forward) and one μ`sync&ex&fwd-m` (synchronise and execute and forward to memory) unit for the adder and two μ`sync&ex` (synchronise and execute) units for the comparator.

The basic functionality of a μcontrol circuit is to receive data, to perform an operation in the datapath to which it is attached, and to forward the operation's result data to another datapath unit. The different types of μcontrol circuits are slight variations of this basic functionality. The μcontrol circuits are equivalent to latch control circuits of micropipelines [DW95]. The simplest type of μcontrol circuit is the μ`ex&fwd`. Its purpose is to execute a local action with the data that is already available and then forward the result. Buffering of the input data is necessary, so it is attached to a local register. Hence, before the local action can be performed, the data must be buffered. The second type, μ`sync&ex`, must synchronise two operations, then buffer the incoming data and perform a datapath operation without forwarding a result. The data travels with one of the two handshakes. The third type, μ`sync&2ex` synchronises two operations, buffers the data and then performs two datapath operations. Finally, the μ`sync&ex&fwd` combines the actions of synchronisation, local buffering, local execution and data forwarding.

5.3 Instruction Flow and Execution

Depending on the type of instruction that is to be executed, different μoperations are issued to the datapath and executed. In order to understand how μoperations and their data flow into the datapath and are used to execute instructions, the execution of an instruction can be broken down into the following stages; μoperation issue, operand fetch, operand data arrival, functional unit execution, result data arrival and result write-back.

5.3.1 μoperation Issue

The first stage of the execution of an instruction is the issue of its μoperations from the control unit into the datapath. Depending on its type, a μoperation may feed to only a single datapath μblock, thus initiating only one datapath operation, or it may feed to multiple ones and initiate multiple operations. In the latter case, the acknowledgement signal for such a μoperation must be generated either by using all of the acknowledgement signals of the μblocks to which that μoperation is connected, or by using one of them, that of the last operation to complete. The μblocks which are initiated may themselves initiate datapath operations, in the same way. The control handshakes that take place between the datapath μblocks are similar in nature to the μoperation handshakes between the datapath and the control unit, as they are necessary to execute the instructions.

As can be seen from the detailed architectural diagram, Figure 5.6, in the case of addition, the `AOpReq` signal is fed to three units, the two μblocks that send data to the adder, *i.e.* μ`sync&ex&fwd` and μ`sync&ex&fwd-m` and to the `write-back` unit. The `AOpAck` signal is generated by the `write-back` unit. For branches, the `COpReq` signal is fed, much like the `AOpReq`, to the two μblocks that send data to the comparator. In this case, the completion of the comparator signals the end of the operation, so the `COpAck` signal is generated by ORing the two `COpAckX` and `COpAckY` signals of the two units. These two signals show that the μ`sync&ex` blocks have detected the completion of the operation.

Another interesting case is the `Wz` μoperation. The `WzReq` signal is fed to both the `write-back` unit and to the μ`sync&2ex` block that writes the results back into the SRF. The `WzAck` signal is generated by ORing the busy signals of the `write-back` and the μ`sync&2ex` units, so as long as one of them is still active, it will stay asserted.

5.3.2 Operand Fetch

Instructions that need to read one or two operands from the SRF will have one or both `Rx` and `Ry` μoperations activated. The data parts of these μoperations are the register indices, and these are buffered by 5-bit registers, one per port. This is necessary to avoid possible interference, in the case when another instruction is fetched.

The two μ`ex&fwd` units begin their operation when triggered by `RxReq` and `RyReq` respectively. They are assigned to the two SRF ports. They perform the register read operation and when the SRF has asserted its acknowledgement signal, *i.e.* the value of the register being read is on the output bus of the port, they then initiate a bus handshake by asserting the `XBusReq` and `YBusReq` signals respectively. If only one operand is to be read, then only one read and one bus handshake takes place. The request signal of the SRF must remain asserted until the data travels down the bus lines and is successfully received.

5.3.3 Operand Data Arrival

An FU operation cannot begin unless its data is available. The data must synchronise with the FU μoperation which has already been initiated by the control unit. The XBusReq and YBusReq signals, generated at the operand fetch stage, handshake with the μcontrol circuits of the FU operands, one per operand. The bus data have to be buffered by 32-bit registers, to release the bus and the operand fetch circuits, and allow another instruction to use them. As soon as the bus data has been written to an operand register, the bus acknowledgement signal, XBusAck or YBusAck is asserted.

The operands may not arrive simultaneously, depending on the SRF delay. A shared access, for example, may take longer than a local one, or there may be a RAW dependence, where one of the registers has not been written back yet, *i.e.* its contents are not marked as valid. When an operand has been written to its register, the request signal of the corresponding operand μcontrol circuit, requ, is asserted. This signal shows that this operand is ready to be used for execution at the FU. The adder and the comparator require two operands to arrive before execution can begin, whereas the external memory interface requires only one, the memory data to be written into the memory (in the case of a store). The memory interface and the second operand of the adder share the Y Bus and the same operand μcontrol circuit, μsync&ex&fwd-m. The difference between this and the μsync&ex&fwd circuit, used for the first operand, is that the former does not perform the requ, acku handshake if its mem input is asserted, *i.e.* in the case of a memory instruction.

5.3.4 Functional Unit Execution

Only FU instructions enter this stage. As soon as the number of necessary operands is available, the functional unit operation can proceed. For FU operations that require more than one operand, the requ signals of the operand μcontrol circuits are ANDed together to produce FU's request signal. As can be seen in the architectural diagram, Figure 5.6, this is how the request signals of the adder, addreq, and the comparator, cmpreq, are generated.

The FU operation then takes place and, when it has completed, the acknowledgement signal of the FU is asserted. Both the adder and the comparator implement completion detection, so the time required for the execution stage of ADD and BEQ instructions depends on the values of the instruction's operands. The acknowledgement signal of the FU is fed to the μcontrol circuits of both operands. For the μsync&ex operand circuits of the comparator, the assertion of the FU acknowledgement signal completes their operation and, in fact, the execution of a BEQ instruction in the datapath. They then return to their idle state.

5.3.5 Result Data Arrival

Three types of instructions write data back into the SRF: ADD, LD and LI. Memory stores send data to memory and do not use this stage. The write-back process can only be initiated when the data to be written back is available and placed on the SRF's input bus. The three possible sources that can write-back data are multiplexed by tri-state elements, the outputs of which are connected together to form the Z Bus, and connected to the input bus of the SRF. The three enable signals, ImEn, MemEn and AddEn select the write-back data source.

Instructions that require data to be written back have their Wz μoperation asserted. The WzReq signal feeds to the write-back unit and to the μsync&2ex control circuit.

The arrival and availability of data for the write-back is checked by the write-back unit. For immediate instructions the write-back data is always available, as it is part of the instruction, and is stored in an 8-bit register, when the Wz μoperation is active.

Memory loads enter the datapath at this stage. When the ld/st input of the memory unit is deasserted, that identifies the memory instruction as a load and handshaking with the external memory unit is performed by signals MemReqEX and MemAckEX. The external memory interface must obey the bundled data protocol, so when the MemAckEX signal is asserted, the memory data is considered valid. When the MemAckEX signal is asserted, the MUWReq signal, that feeds to the write-back unit and initiates the write-back for loads, is asserted.

Stores do not use the MUWReq, MUWAck handshake. When the data of a store instruction has been written by the μsync&ex&fwd-m circuit into the 32-bit, Y Bus register that store instructions share with adds, the μsync&ex&fwd-m circuit will assert its reqnextY signal, which feeds to both the write-back and to the memory i/face circuits. That shows that the memory data is available, and will initiate the MemReqEX, MemAckEX handshake. In the case of a store, when the MemAckEX signal is asserted, that completes both the operation of the memory i/f, which enters its idle state, and the store instruction.

ADD instructions must wait until the result of the addition is available. When the addack signal of the adder is asserted, then the two μsync&ex&fwd and μsync&ex&fwd-mem circuits will assert their reqnextX and reqnextY signals respectively. At this point, the result of the addition is on the adder's output and the write-back can be initiated. The addreq signal must stay asserted to keep the result data valid, until the write-back has completed.

5.3.6 Result Write-back

The writing-back of results is performed by the μsync&2ex control circuit, that controls the clear and write ports of the SRF and is initiated by the WzReq signal. As writing-back must synchronise with the data, the write-back process does not actually start until the startwrite signal is asserted, which is controlled by the write-back unit. The two handshaking signal pairs of the μsync&2ex control circuit, requ1, acku1 and requ2, acku2 are connected to the clearreq, clearack and writereq, writeack SRF signals respectively, handshaking with the SRF's clear and write ports.

Except for the data to be written back, the write-back μoperation also requires the index of the destination register into which the data is to be stored. That index must be buffered, as are the operand fetch register indices, to allow another instruction to be fetched, without its data interfering with the current instruction's write-back μoperation. The outputs of the 5-bit write-back register are fed to the clear and write register index inputs of the SRF.

As soon as the startwrite signal is asserted, a clear operation is performed, followed by a write. A clear operation will deassert the register valid bit of the register that will be written to, and then the write will store the result into that register and set its valid bit. The valid bit implements a register locking mechanism, so that any hazards between instructions that are "in-flight" simultaneously in the datapath are respected.

The assertion of the writeack signal of the SRF signifies that the write-back has completed. This signal is shorted to the w-fin output of the μsync&2ex, and it serves as the acknowledgement of the startwrite signal. When w-fin is asserted, both the μsync&2ex control circuit and the write-back unit will enter their idle state, and the instruction's execution will have completed.

5.3.7 Register Locking Mechanism

In the A1, the register locking mechanism is implemented in the following way; a register clear preceds a register write. The reason for performing the register clear before the write is because at this point it is known that any register operands of the instruction will have been read. Alternatively, if the clear took place when the instruction was issued, then instructions whose result is the same as one of their operands would have to be handled in a special way, increasing the design complexity.

By performing the clear so late in an instructions execution, a timing assumption is introduced and it is in theory possible that a RAW hazard might not be handled appropriately. This may happen in the case where the register contents are valid before a register clear, *i.e.* the register valid bit is set, and the following instruction attempts to read this register. If the read takes place before the clear, it is possible that the previous register value is read. However, due to the delays of the architecture, the clear of one instruction will take place before the read of the next one, so this problem does not occur in practice.

5.4 Processor Components

In this section, the design and implementation of the processor components, along with their performance, where relevant, is described. The processor's control circuits have been implemented from FSM diagrams using the Direct-Mapped CMOS AFSM approach described in Chapter 2.

5.4.1 Data Registers

The registers used for data buffering are composed of the appropriate number of SRAM cells, and an extra valid bit for performing bounded-delay completion detection. They have three inputs signals, `prech`, `clear` and `write` and two precharged output signals `out-c` and `out-w`. The input signals precharge the two output lines, clear the valid bit and write to the register and set the valid bit respectively. The two output signals are the outputs of two read ports of the register's valid bit, and are enabled by the `clear` and `write` inputs respectively. They are used as acknowledgement signals to the inputs.

Precharging of the output port lines is necessary to detect the output transition correctly, independent of the initial value of the valid-bit and of circuit delays. Precharging sets the floating node to the opposite of the value that the transition that is to be detected ends, for example to detect a $0 \rightarrow 1$ transition, the node is precharged to 0, for a $1 \rightarrow 0$, to 1. This forces and guarantees a single output transition. If precharging were not used, the possibility of either no transitions or two output transitions occurring would exist. These two cases cannot be distinguished without timing assumptions that are unrealistic in an asynchronous implementation.

5.4.2 μcontrollers

The five μcontrol circuits, *i.e.* μ`sync&ex`, μ`sync&ex&fwd`, μ`ex&fwd`, μ`sync&2ex` and μ`sync&ex&fwd-m`, and their functionalities were described in Section 5.2.

The operations that these circuits perform and the sequencing of these operations depends on the μcontrol circuit type. These include synchronisation of two handshakes, data buffering in a local register that is attached to the μcontrol circuit, generating control handshakes and signals and interfacing with datapath elements to perform datapath operations.

5.4.2.1 μsync&ex

Figs 5.7 and 5.8 show the FSM diagram and layout of the μ`sync&ex` circuit respectively.

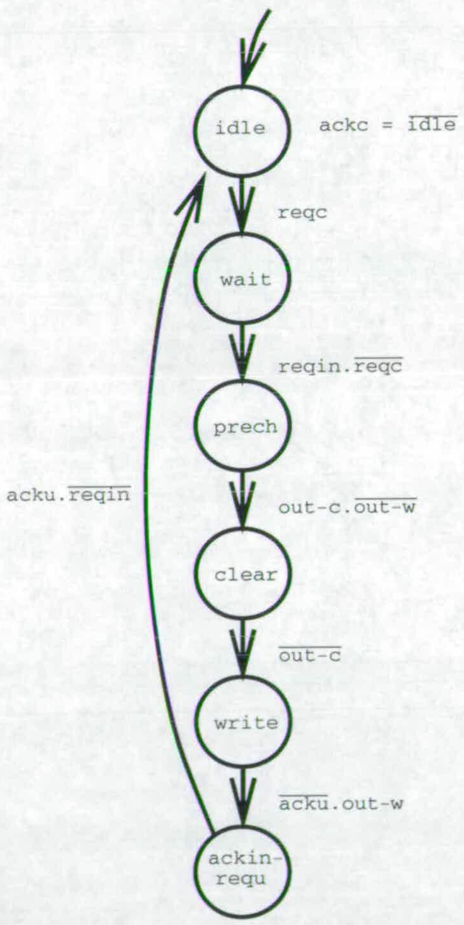

idle
ackc = $\overline{\text{idle}}$
reqc
wait
reqin.$\overline{\text{reqc}}$
prech
acku.$\overline{\text{reqin}}$
out-c.$\overline{\text{out-w}}$
clear
$\overline{\text{out-c}}$
write
$\overline{\text{acku}}$.out-w
ackin-
requ

Figure 5.7: FSM of μsync&ex control circuit

The μ`sync&ex` circuit synchronises the μoperation that is issued by the control unit, `reqc`, `ackc`, with its data, which is attached to another μoperation, `reqin`,

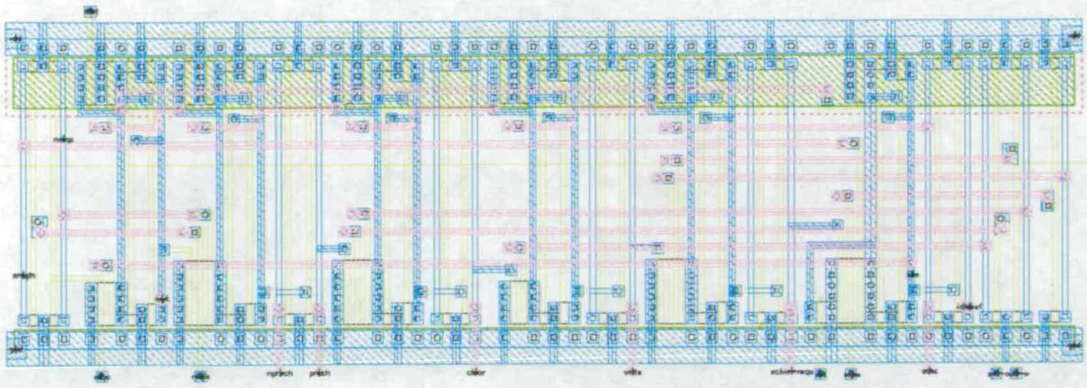


Figure 5.8: Layout of μsync&ex control circuit

`ackin`. It then stores the data into a local register and performs a datapath operation, `requ`, `acku`. The `reqin`, `ackin` data are connected to the inputs of the local register, the outputs of which are inputs to the datapath operation.

In most cases, the state names represent output signals, for example state write asserts the write output, in others, there is no corresponding output signal, as with state wait.

The assertion of the `reqc` signal initiates the circuit operation. The FSM leaves the `idle` state and enters the `wait` state. The `ackc` output signal is asserted at this stage. The acknowledgement signals of μcontrol circuits are asserted as early as possible, to enable the μinstructions to be issued as soon as possible into the datapath, so as to disengage the control unit and allow it to fetch and consider the next instruction for issue. When the `reqin` handshake is asserted, and therefore the input data is available, the FSM will leave the `wait` state. This implements the synchronisation between the two handshakes.

The next three states, `prech`, `clear` and `write` perform the local buffering. In state `prech`, the `out-c` and `out-w` lines are precharged high and low respectively, as their low$\rightarrow$high and high$\rightarrow$low transitions are to be detected. These two lines float when the `clear` and `write` register inputs are deasserted. After precharging, a clear is performed, followed by a write. When the `clear` input is asserted, the register is cleared and the `out-c` output is driven with the valid-bit value. The `clear` state is left when the `out-c` signal returns low. The write works in a similar way; its next state `ackin-requ`, which produces both the `ackin` and `requ` signals, will only be entered when the `acku` signal is deasserted, thus obeying the four-phase protocol.

As the data have now been written to the register, the `ackin` and `requ` signals can be asserted, to signal that the `reqin`, `ackin` μoperation has completed and to initiate the FU operation. This is the final stage in the control circuit's operation. When the FU operation has been acknowledged, and the `reqin`, `ackin` μoperation has dropped its request signal, the FSM will return to its `idle` state. The `ackc` signal, which is not shown in the FSM diagram, is produced by negating the `idle` state signal. In this way, the control unit will not re-assert its request signal until the FSM has finished its operation.

The other four circuits, μ`ex&fwd`, μ`sync&2ex`, μ`sync&ex&fwd` and μ`sync&ex&fwd-m`, are variations of the μ`sync&ex` circuit.

5.4.2.2 μsync&ex&fwd

The μ`sync&ex&fwd` circuit forwards the result of the FU operation to another unit. It performs an extra `reqnext`, `acknext` handshake (Figure 5.9), with which the FU result data travel.

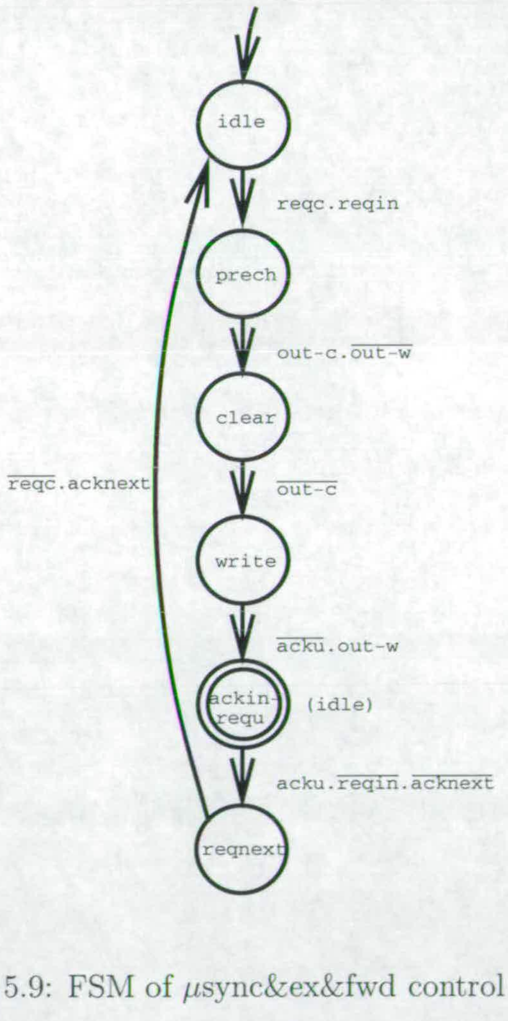

idle
reqc.reqin
prech
out-c.$\overline{\text{out-w}}$
clear
$\overline{\text{reqc}}$.acknext
$\overline{\text{out-c}}$
write
$\overline{\text{acku}}$.out-w
ackin-requ
(idle)
acku.$\overline{\text{reqin}}$.$\overline{\text{acknext}}$
reqnext

Figure 5.9: FSM of μsync&ex&fwd control circuit

The μ`sync&ex&fwd` circuit can be used to form an asynchronous pipeline, where the input and output handshakes of each pipeline stage are fully-decoupled.

It is similar to a fully-decoupled micropipeline latch control circuit [FD96], as the timing of the output acknowledgement does not influence the completion of the input handshake. Its silicon layout is shown in Figure 5.10.

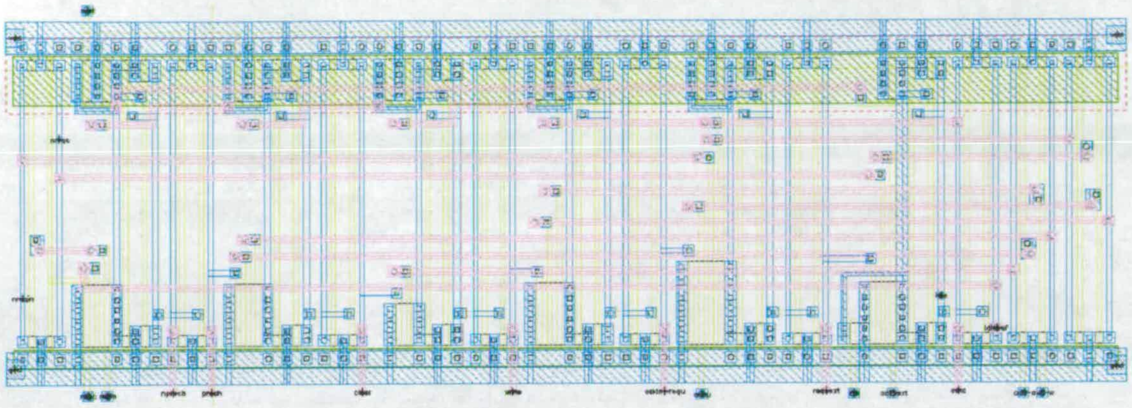


Figure 5.10: Layout of μsync&ex&fwd control circuit

This FSM is almost identical to the FSM of the μ`sync&ex` (Figure 5.7) up to state `ackin-requ`. As this circuit does not interface directly to the control unit, the `wait` state has been dropped, to save on one state, and the `reqc` signal is acknowledged late. After the FU operation has completed and `acku` has been asserted, the `reqnext`, `acknext` handshake will forward the data. State `ackin-requ`, must stay asserted, for as long as the data forwarding handshake is in progress to ensure that the FU data are valid. State `ackin-requ` is double-circled because it is a persistent state (*c.f.* Section 2.8.3.9). It is left when state `idle` (shown next to it in brackets in Figure 5.9) is entered.

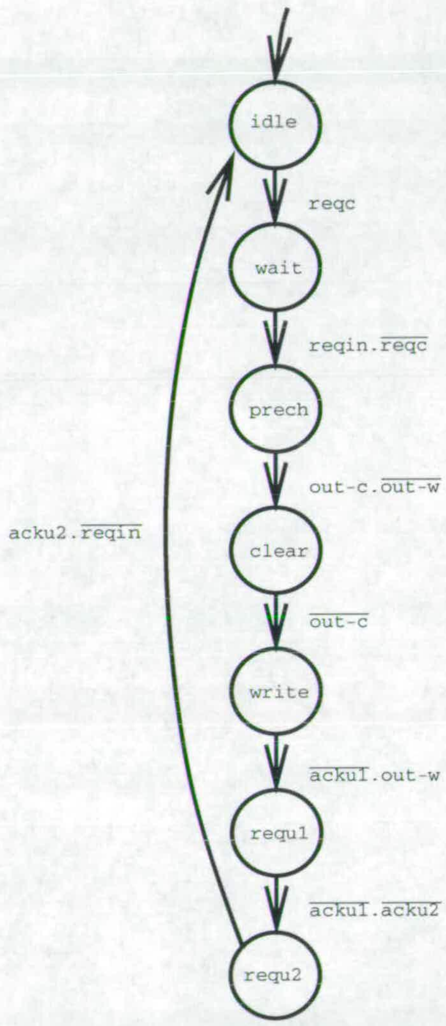
The fact that the `idle` state resets both its immediate predecessor and state `ackin-requ` is reflected in the p-type transistor routing in Figure 5.10. For all state blocks, apart from the last two, the purple horizontal metal routing at the top of the figure connects the p-type pull-up with the inverse of the next state (three purple lines near the top of the layout), whereas the p-type pull-ups of the last two are shorted and connected to the inverse of the `idle` state.

5.4.2.3 μex&fwd

The μ`ex&fwd` circuit, which is used for performing the operand fetch, has not been implemented as a new circuit. As its only difference from the μ`sync&ex&fwd` is that synchronisation between two operations is not necessary, it is implemented by connecting together the `reqc` and `reqin` signals to the input request and assigning `ackc` as its acknowledgement.

5.4.2.4 μsync&2ex

The purpose of the μ`sync&2ex` circuit is to interface with the write and clear ports of the SRF. It performs two operations, a clear followed by a write with the same operand data. The FSM and layout of this circuit are shown in Figures 5.11 and 5.12.



idle
reqc
wait
reqin.$\overline{\text{reqc}}$
prech
out-c.$\overline{\text{out-w}}$
acku2.$\overline{\text{reqin}}$
clear
$\overline{\text{out-c}}$
write
$\overline{\text{acku1}}$.out-w
requ1
$\overline{\text{acku1}}$.$\overline{\text{acku2}}$
requ2

Figure 5.11: FSM of μsync&2ex control circuit

This FSM is identical to that of μ`sync&ex` (Figure 5.7) up to the state `write`. After this the data have been stored into the register, and the two handshakes take place, the one after the other.

5.4.2.5 μsync&ex&fwd-m

The μ`sync&ex&fwd-m` is a generalisation of the μ`sync&ex&fwd` circuit that can behave either as the the former, or only synchronise and forward data, with-

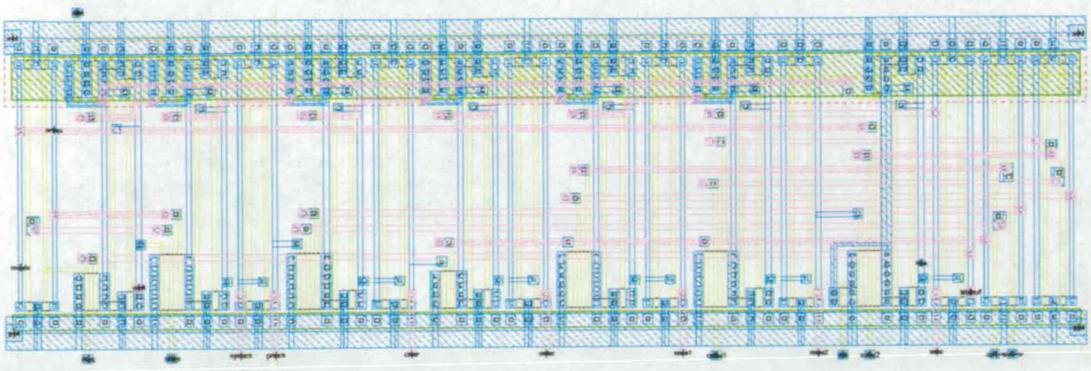


Figure 5.12: Layout of μsync&2ex control circuit

out performing the execution handshake. It is used to synchronise the operand bus data and depending on the type of instruction, perform an FU operation and forward the result, or forward the bus data. The FSM and layout of the μ`sync&ex&fwd-m` circuit are shown in Figures 5.13 and 5.14.

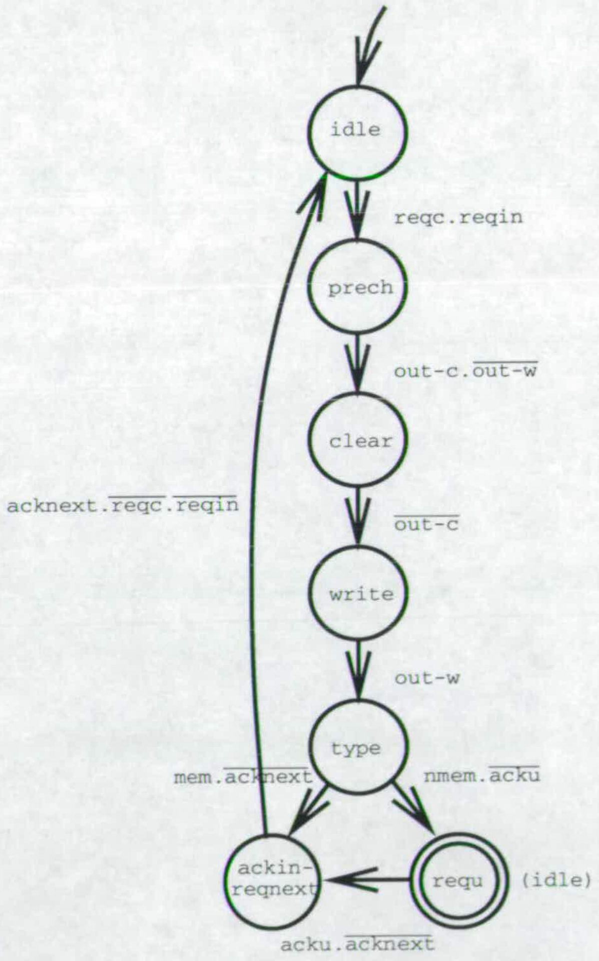

idle
reqc.reqin
prech
out-c.$\overline{\text{out-w}}$
clear
acknext.$\overline{\text{reqc}}$.$\overline{\text{reqin}}$
$\overline{\text{out-c}}$
write
out-w
type
mem.$\overline{\text{acknext}}$
nmem.$\overline{\text{acku}}$
ackin-reqnext
requ
(idle)
acku.$\overline{\text{acknext}}$

Figure 5.13: FSM of μsync&ex&fwd-m control circuit

This FSM is similar to the FSM of the μ`sync&ex&fwd` (Figure 5.9) up to state `write`. When the writing of the register has completed, the `type` state will be entered, and the next operation, FU or forwarding, depends on the `mem`, `nmem` inputs. The `nmem` input is the negation of `mem`, and must be settled in value before state `type` is entered to avoid both its successors being entered, and a one-hot FSM sequential hazard occuring.

If the instruction does not require a memory access, then state `requ` is entered, and the FU operation is performed. If it is a memory instruction, then state `ackin-reqnext` is entered, and the forwarding of the bus data is performed. Different data busses are associated with the `reqnext`, `acknext` signals depending on the operation performed. Due to the fact that the bus data must remain valid until the `ackin-reqnext` state, as the instruction may be a memory one, the `ackin` signal is asserted after the `reqnext`, `acknext` handshake has completed.

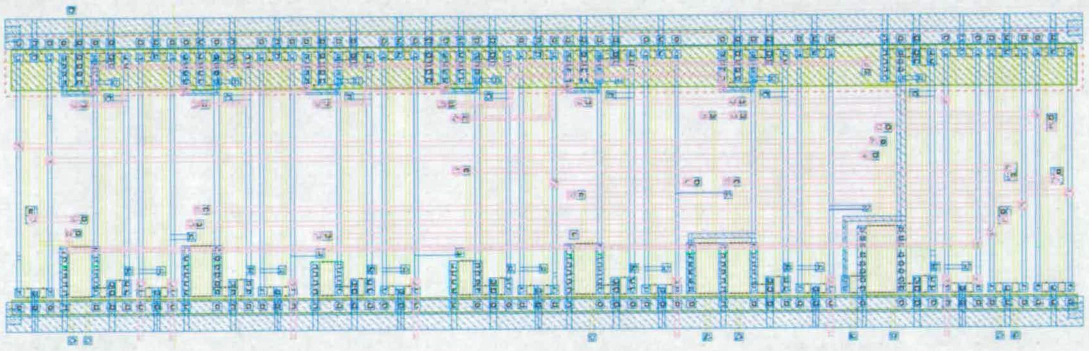


Figure 5.14: Layout of μsync&ex&fwd-m control circuit

5.4.2.6 Simulation of the μsync&ex&fwd circuit

As the operation of the μsync&ex&fwd circuit is characteristic of the μcontrol circuits and it is also similar to the fully-decoupled latch control circuit of a micropipeline, its circuit simulation output is studied in this section.

Figure 5.15 shows the graphical simulation output of the typical circuit operation. The top diagram shows the handshake signals and the bottom one the register control signals.

After the circuit is initialised, the two handshakes that are to be synchronised, `reqc` and `reqin`, are asserted at the simulation time of 2ns and 3ns respectively. As the circuit leaves its `idle` state (signal `idlebuf` in the bottom diagram), the `ackc` output is asserted. At this point, the process of writing the incoming data into the register is initiated. As shown in the bottom diagram, the `prech` state is entered, and the `out-c` input, which comes from the valid bit of the

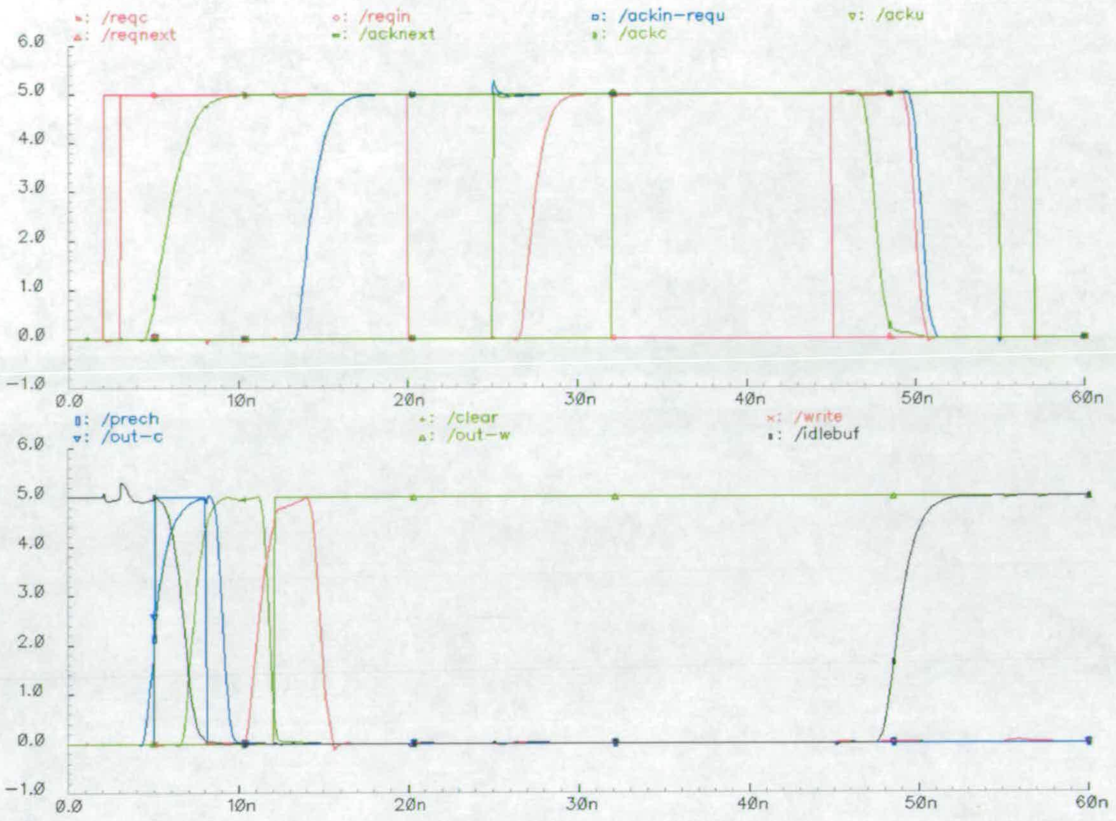


Figure 5.15: Simulation of the μsync&ex&fwd control circuit

register, is asserted. Then, the `clear` output clears the `out-c` input, and hence the register valid bit is cleared. Then, state `write` is entered and the corresponding output is asserted. Input `out-w` is asserted at the simulation time of 12ns. At this point, the data have been written, so the FU operation can be initiated and the `reqin` input can be acknowledged. Hence, the `ackin-requ` output is asserted, and `reqin` gets deasserted at the simulation time of 20ns. At 25ns, the FU acknowledgement, `acku`, is asserted signalling the completion of the FU operation and the availability of output data, and `reqnext` is asserted at 28ns. The corresponding acknowledgement, `acknext`, is asserted at 32ns, signaling the data transfer. Finally, the `reqc` input drops at 45ns. As an instruction will take quite a long time to issue, this signal will take a long time to return to zero. At this point, the `ackc`, `reqnext` and `ackin-requ` signals are deasserted, and in response to this, signals `acku` and `acknext` are also deasserted.

5.4.3 Processor Control Unit

The purpose of the control unit is to fetch and issue instructions into the datapath. The control unit is responsible for the control flow of the program, and contains the processor's program counter (PC). It must be connected to an external instruction memory, from which instructions are to be fetched.

The control unit first performs an instruction fetch, by sending the value of the PC to the instruction memory. When it receives an instruction from memory, it breaks down the instruction into μinstructions, and issues them in parallel to the datapath. When the issue of all of the μinstructions of the instruction has been acknowledged, the PC is incremented and the next instruction can be fetched. The PC is 8-bits wide. The layout of the control unit is shown in Figure 5.16.

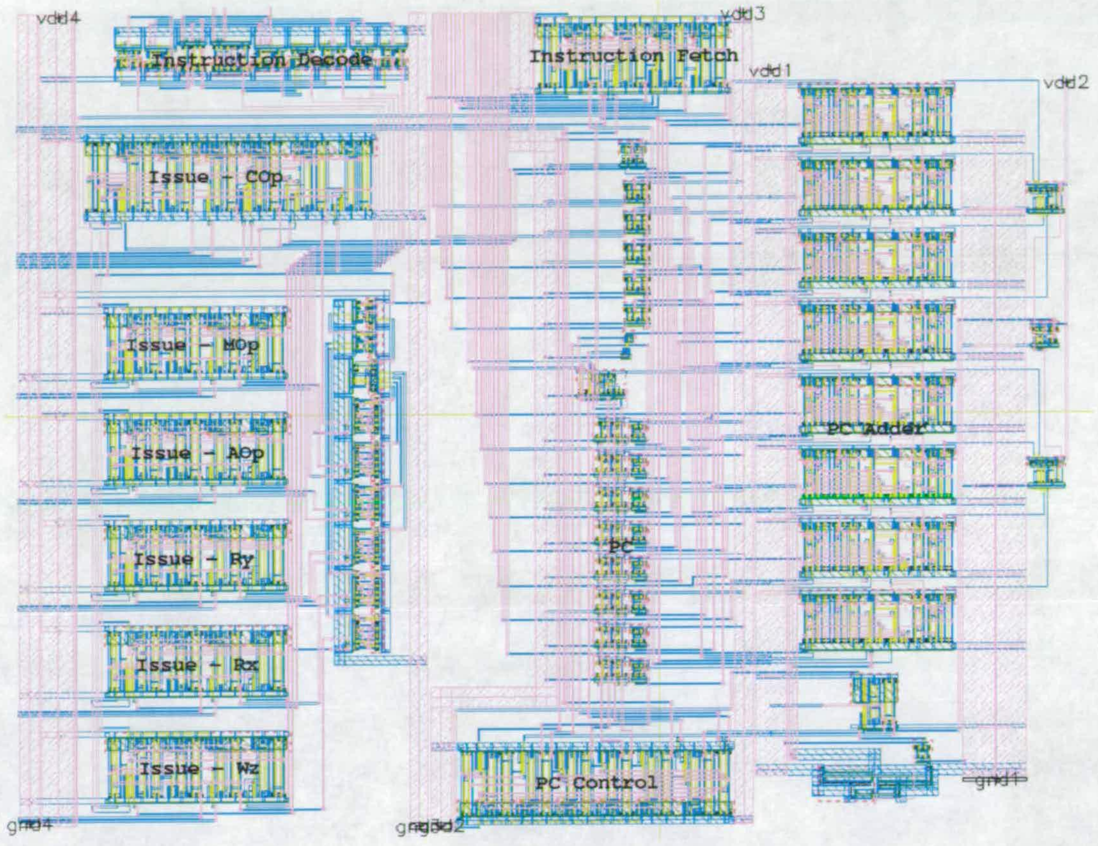


Figure 5.16: Control Unit Layout

The control unit contains an 8-bit adder for incrementing the PC and relative branches and decode and issue detection circuitry. It also contains a total of eight control circuits. Two of them are used for the instruction fetch and the PC control respectively, and the remaining six are the μoperation issue circuits, one per μoperation.

5.4.3.1 Instruction Fetch

Instruction fetching is performed by the `CTRL_fetch` circuit. Its FSM and layout are shown in Figures 5.17 and 5.18 respectively.

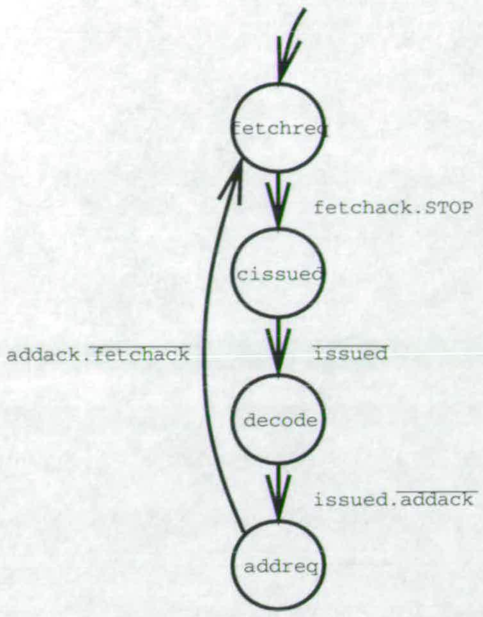

fetchreq
fetchack.STOP
cissued
addack.$\overline{\text{fetchack}}$
$\overline{\text{issued}}$
decode
issued.$\overline{\text{addack}}$
addreq

Figure 5.17: FSM of the instruction fetch and issue control circuit

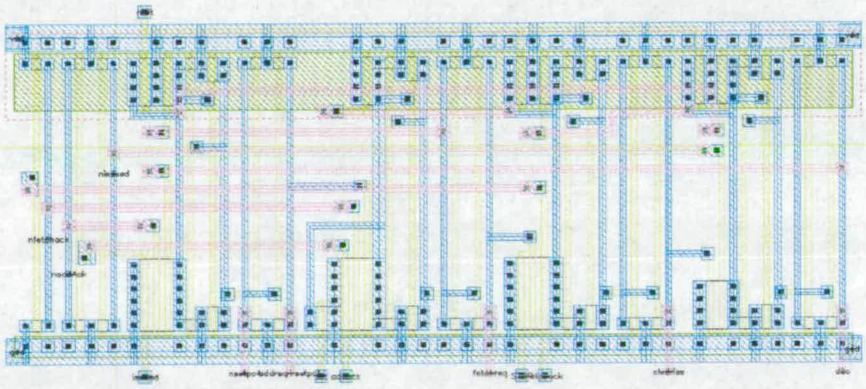


Figure 5.18: Layout of the instruction fetch and issue control circuit

When the processor is reset, the `CTRL_fetch` circuit will be initialised in the `fetchreq` state and will initiate the fetching of the first instruction. One of the circuit inputs, `STOP`, is introduced for debugging purposes. The `STOP` input is to be connected to an external switch and external logic, so that when activated, it allows a program to be single stepped, by preventing the next instruction from being fetched. In this way, it would be possible, once the circuit was fabricated, to observe the datapath state at the end of an instruction, using, for example, a logic analyser.

The `decode` signal indicates that the fetching of the instruction has completed, and passes control to the μoperation issue circuits. At this stage, the instruction is assumed to have already been decoded into μoperations, as there is no completion signal associated with instruction decode.

5.4.3.2 Instruction Decode

When the instruction has been fetched, it is decoded by a combinational logic circuit, which implements the instruction to μoperations mapping that was shown in Table 5.1.

$$\boxed{\begin{array}{l} Wz = \overline{op_1} + (op_1 op_0 \overline{op_m}) \\ Rx = op_1 \text{ xor } op_0 \\ Ry = (op_1 \text{ xor } op_0) + (op_1 op_0 op_m) \\ AOp = \overline{op_1} op_0 \\ MOp = op_1 op_0 \\ COp = op_1 \overline{op_0} \end{array}}$$

Figure 5.19: Instruction decode combinational logic equations

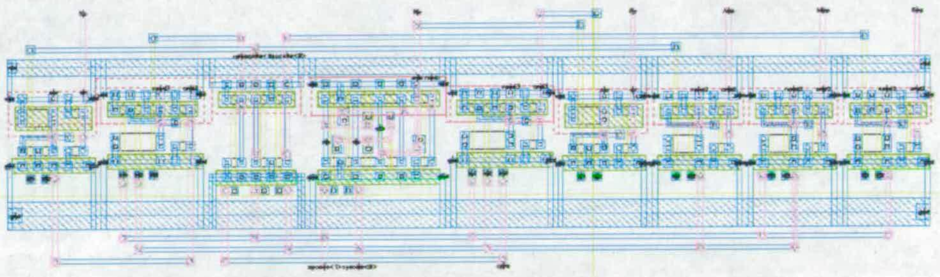


Figure 5.20: Layout of the instruction decode circuit

The instruction decode combinational logic equations are shown in Figure 5.19 and the corresponding layout in Figure 5.20. Inputs op_0, op_1 and op_m are bits 15, 16 and 17 of the instruction respectively. The μoperation outputs are each fed to their corresponding issue circuit, in order to activate it.

5.4.3.3 Instruction Issue

Each μoperation which is active must be issued into the datapath. There are six issue circuits, one per μoperation. They are all identical, except for the one that issues the `COp` μoperation, used for branches, due to its special functionality, *i.e.* potentially modifying the value of the PC. The FSM and layouts for the other five μoperation issue circuits are shown in Figures 5.21 and 5.22 respectively.

The `decode` input is fed by the fetch control circuit and enables the issuing of the instruction to begin. The `active` input is connected to the output of the

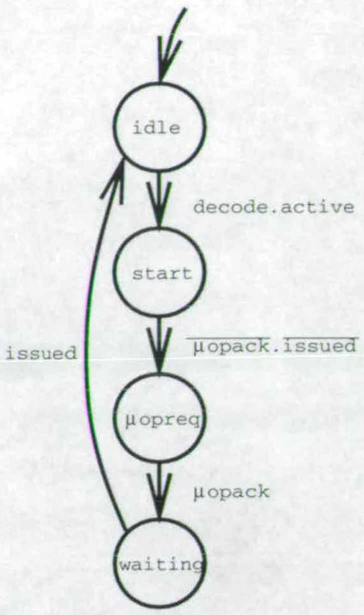


Figure 5.21: FSM of the μoperation issue circuit

decode circuit to which this issue circuit corresponds. It is set if this μoperation is active for the current instruction. Hence, if this μoperation is to be issued, the μ`opreq` state will be entered. When the μoperation has been issued, the μ`opack` input will be asserted and the FSM will enter state `waiting`. The issue circuits must wait in this state, until all the μoperations have been issued and the `issued` signal is asserted.

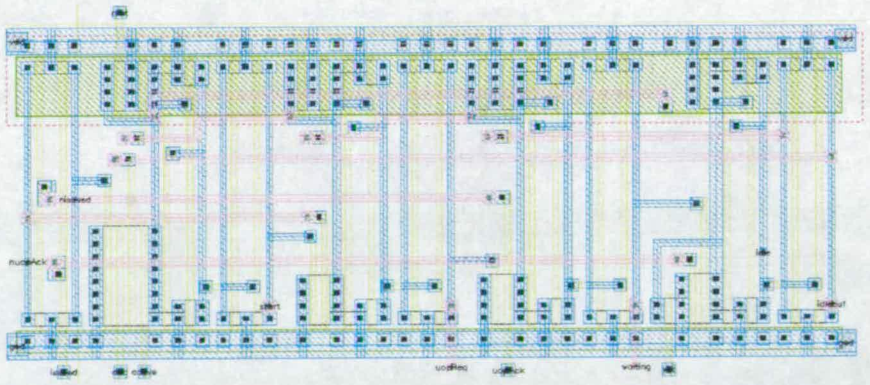
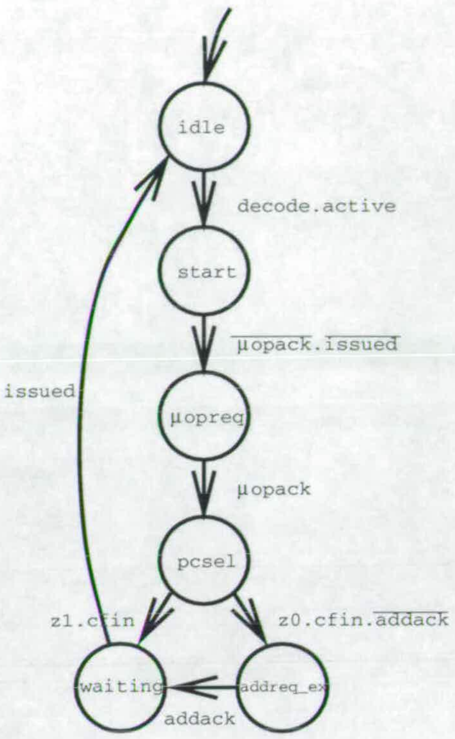


Figure 5.22: Layout of the μoperation issue circuit

Figure 5.23 shows the FSM of the `COp` μoperation issue circuit and Figure 5.24 shows its layout. This FSM has a different behaviour from the FSMs of the other issue circuits, after the μoperation has been issued, *i.e.* after state μ`opreq`. It will then enter state `pcsel` and wait until the comparator has completed its operation,



idle
decode.active
start
$\overline{\mu\text{opack}}.\overline{\text{issued}}$
issued
μopreq
μopack
pcsel
z1.cfin
z0.cfin.$\overline{\text{addack}}$
waiting
addreq_ex
addack

Figure 5.23: FSM of the COp μoperation issue circuit

i.e. signal `cfin` is asserted and, depending on the result of the comparison, will add the branch offset to the PC, or directly enter its `waiting` state. Signal `z0` will be asserted if the comparison outcome was true, *i.e.* the processor must branch, `z1` otherwise. Signal `addreqx_ex` enables the offset as the 2nd operand of the PC adder (the 1st operand is the PC itself) and initiates the PC addition. When the PC addition is completed, the `waiting` state is entered.

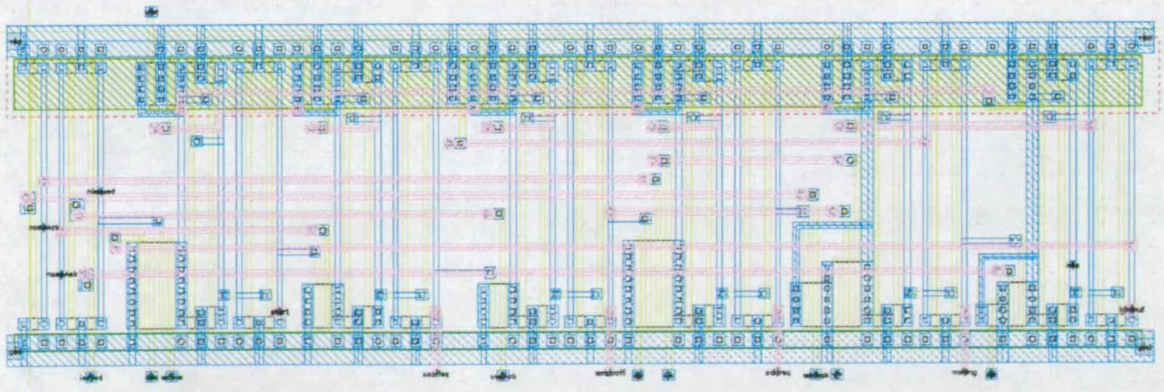


Figure 5.24: Layout of the COp μoperation issue circuit

5.4.3.4 Issue Detection

The issue of an instruction's μoperations is detected by performing a bitwise compare of the outputs of the instruction decode circuit with the `waiting` output of their corresponding issue circuit. It is implemented as an XOR/AND tree, the output of which is fed to the `issued` input of the fetch control circuit. The layout for the issue detection circuit is shown in Figure 5.25.

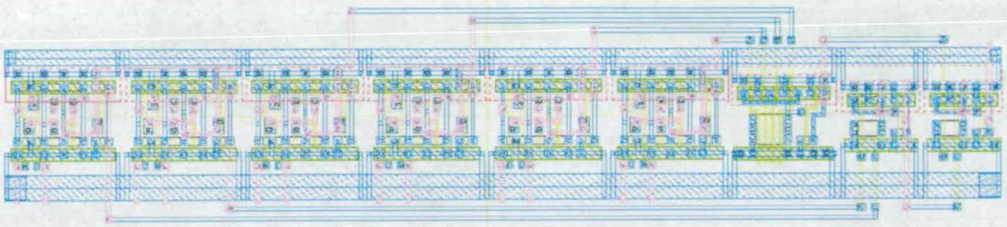


Figure 5.25: Layout of the μoperation issue circuit

5.4.3.5 Program Counter Control

The PC control circuit is responsible for incrementing or adding the branch offset value to the PC. It interfaces to a register that holds the value of the PC. This is necessary to isolate the outputs of the adder from their inputs. The FSM and layouts of the PC control circuit are shown in Figures 5.26 and 5.27 respectively.

Signal `addreq` is the input that initiates the PC addition. Signals `requ` and `acku` handshake with the 8-bit PC adder. Signal `requ` is generated by negating the `idle` state signal, so as soon as the FSM leaves the `idle` state it will be asserted. The assertion of `acku` signals the completion of the addition, and at this point the result must be stored into the PC register. After the result has been written and the `addreq` signal has been deasserted, the FSM will return to its `idle` state. Signal `write` enables the inputs of the PC register to write the result of the addition. An early inverse of this signal enables the PC register outputs, so that no feedback occurs.

5.4.4 Processor Control Unit Simulation

Figure 5.28 shows the graphical simulation output of the control unit when fetching and issuing an immediate, `LI`, instruction. The top diagram shows the external signals, *i.e.* the fetch handshake, signals `fetchreq` and `fetchack` in red and green respectively, and the 8-bit PC outputs.

After the circuit is reset, the `fetchreq` output is asserted. At this point, the PC value is zero. Then, an immediate instruction is presented and the `fetchack`

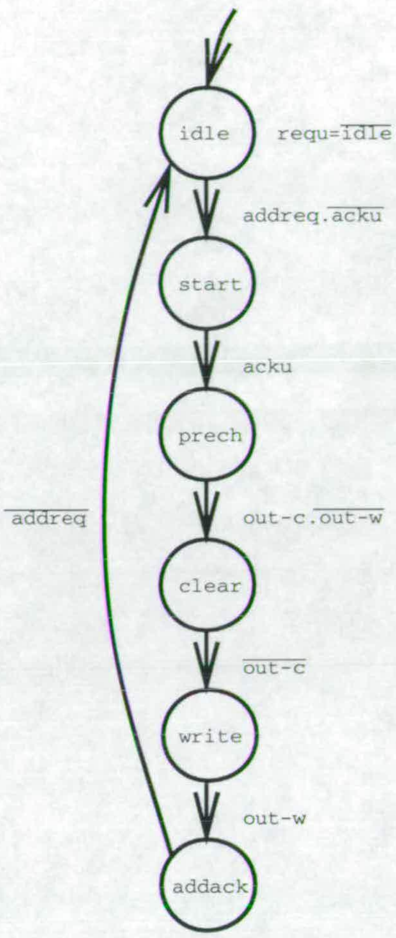

idle
requ=$\overline{\text{idle}}$
addreq.$\overline{\text{acku}}$
start
acku
prech
$\overline{\text{addreq}}$
out-c.$\overline{\text{out-w}}$
clear
$\overline{\text{out-c}}$
write
out-w
addack

Figure 5.26: FSM of the program counter control circuit

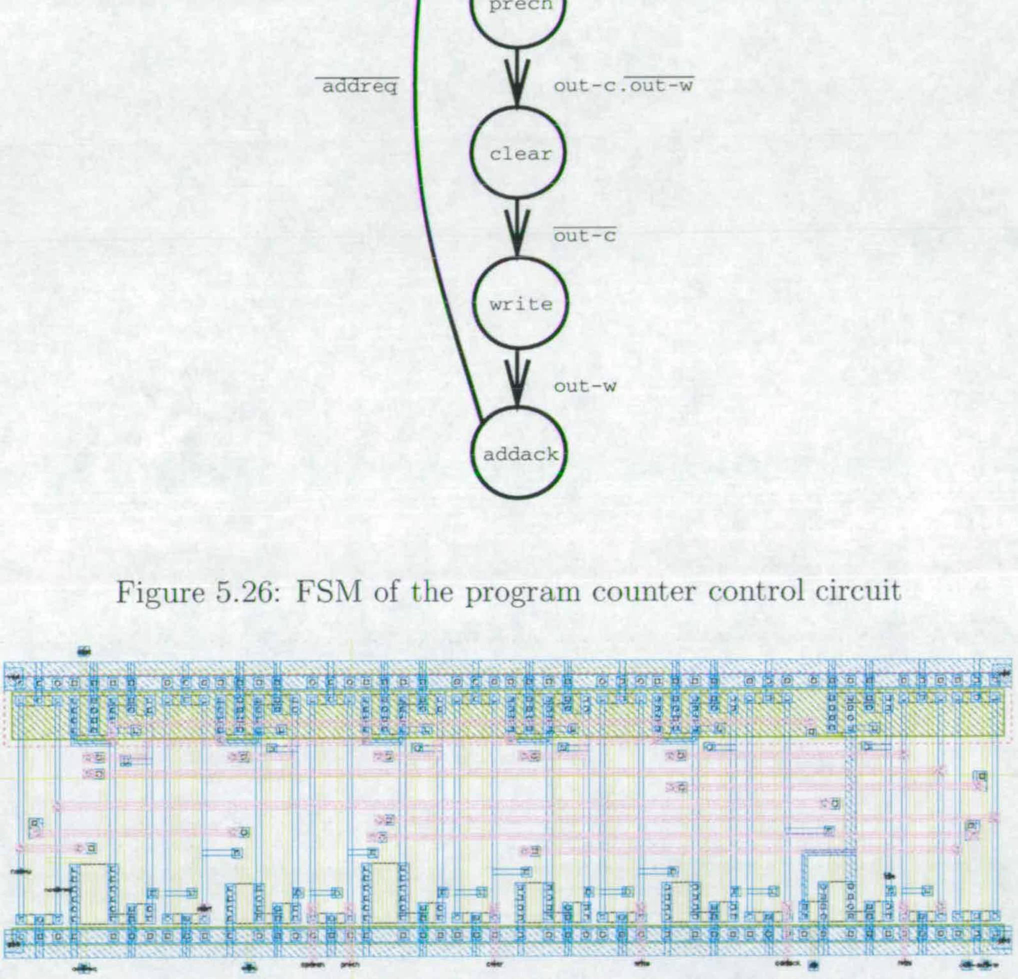


Figure 5.27: Layout of the program counter control circuit

signal is asserted at the simulation time of 6ns. After the instruction is decoded and issued, the `fetchreq` signal is again asserted, at the simulation time of approximately 84ns, and at this time, the least-significant-bit of the PC has been asserted, requesting an instruction from address 1. The PC output rises almost simultaneously with the request signal, and as the output of the PC register does

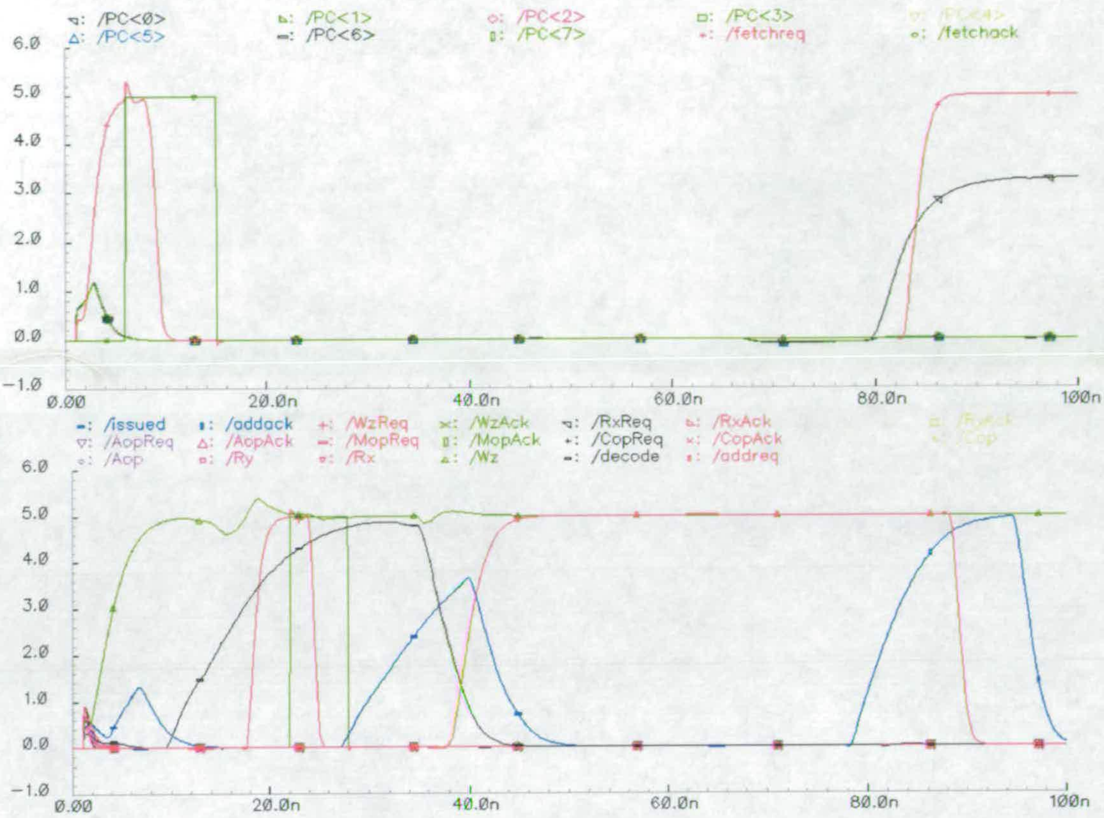


Figure 5.28: Simulation of the fetch and issue of an immediate instruction

not use full pass-transistor logic, it rises to a level lower than the supply line[2].

The bottom diagram shows the internal control signals. As soon as the instruction has arrived, it is decoded by the instruction decode circuitry, and its output `Wz` is asserted. The instruction fetch and issue circuit will then enter its `decode` state and assert this signal. This enables the instruction issue circuits to initiate the issue, and the request signal of the only μoperation that is to be issued, `WzReq`, is asserted at simulation time of approximately 20ns. After the `WzAck` signal is asserted, the issue detection circuit asserts the `issued` signal. The instruction has now been issued and the fetch and issue circuit asserts the `addreq` signal to increment the PC value. When the PC has been incremented, *i.e.* signal `addack` has been asserted, another instruction can be fetched, and `fetchreq` is again asserted.

Figure 5.29 shows the simulation output for a branch-if-equal, `BEQ` instruction.

In this simulation, the outcome of the branch will evaluate to true, so the value of the PC must be appropriately updated. The branch offset is set to -4, *i.e.* bits 14 to 10 of the instruction are set to 00100 (*c.f.* Section 5.2). In the top

[2]due to the transistor threshold voltage drop.

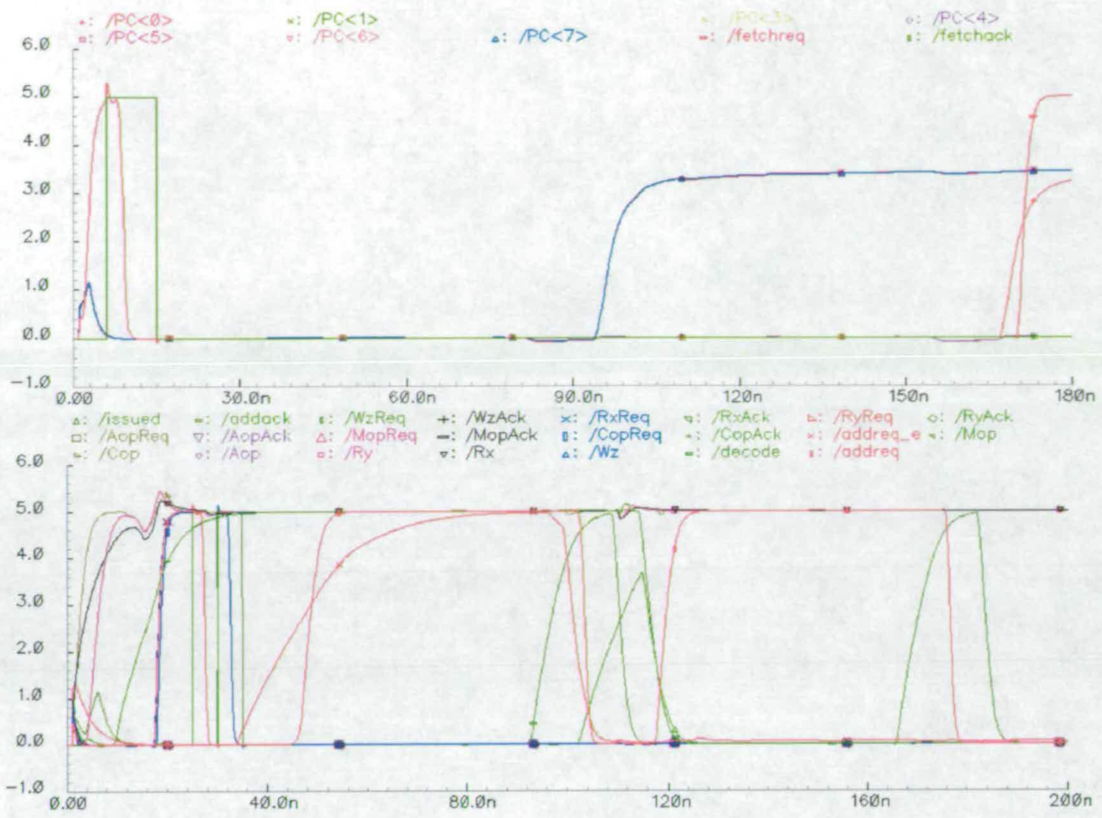


Figure 5.29: Simulation of the fetch and issue of a branch instruction

diagram, the value of the PC of the second instruction fetch is 11101001.

The bottom diagram which shows the internal control signals is more complex in this case. Three μoperations are decoded, when the instruction data arrive, `Rx`, `Ry` and `COp`. When signal `decode` is asserted, the corresponding request signals are asserted. The acknowledgement signals `RxAck` and `RyAck` are simultanesouly asserted at the simulation time of 25ns, and `COpAck` is asserted at the simulation time of 30ns. As the comparison evaluates to true, state `addreq_ex` of the `COp` instruction issue circuit is entered and its corresponding output is asserted. This asserts the `addreq` signal of the PC control adder and sets bits 5 to 8 of the PC. After the addition has been performed at the simulation time of approximately 100ns, the `issued` signal is asserted. Then, the PC is incremented and the next instruction can be fetched.

5.4.5 Memory Interface

The memory interface connects a processor node to an external data memory via an 8-bit data bus and two handshake signals, `MemReqEX` and `MemAckEX`. The type of the operation, *i.e.* a read or a write is determined by bit 17 of the instruction.

The FSM and layout of the memory interface are shown in Figures 5.30 and 5.31.

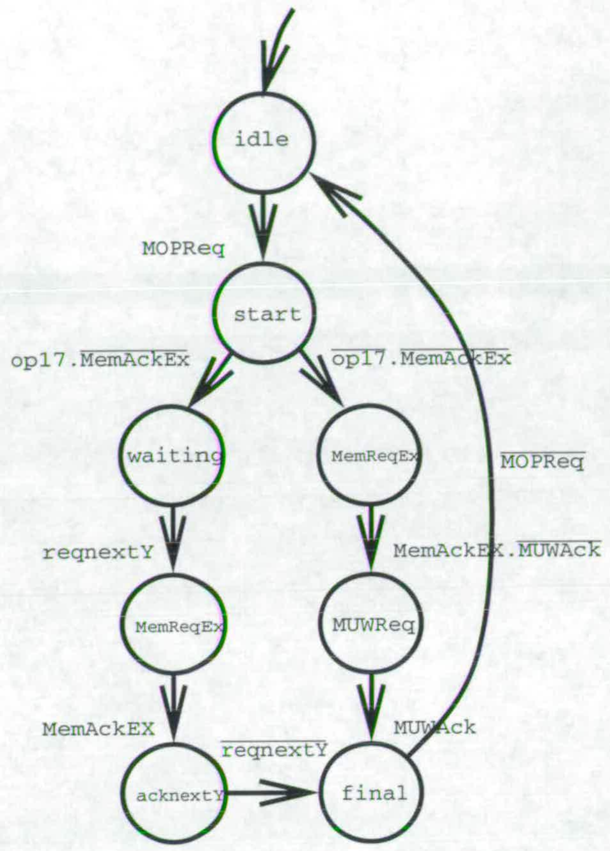

idle
MOPReq
start
op17.$\overline{\text{MemAckEx}}$
$\overline{\text{op17}}$.MemAckEx
waiting
MemReqEx
$\overline{\text{MOPReq}}$
reqnextY
MemAckEX.$\overline{\text{MUWAck}}$
MemReqEx
MUWReq
MemAckEX
MUWAck
$\overline{\text{reqnextY}}$
acknextY
final

Figure 5.30: FSM of Memory Unit

The FSM leaves the `idle` state when the `MOpReq` μoperation is asserted. Signal `MOpAck` is generated early by inverting the `idle` signal. From state `start`, the FSM is divided into two paths, the LHS is for stores, the RHS for loads, and the selection of the path depends on bit 17 of the instruction (`op17`).

Stores must wait until the data has been put onto the bus. Memory instructions use the `Ry` μoperation and data is placed on the Y bus. When `reqnextY` has been asserted, the external handshake may take place. Then the Y bus data is acknowledged (`acknextY`) and the FSM will return via `final` to state `idle`.

Loads initiate the memory operation, and then perform a request to the write-back unit, by entering state `MUWReq` and asserting this signal. Signal `MemReqEX` does not stay asserted until the write-back completes; the data on the memory bus is assumed to stay valid until the next request. The assertion of signal `MUWAck` signals the completion of the write-back, and then, as long as the memory μoperation request has been deasserted, the FSM will return to state `idle`.

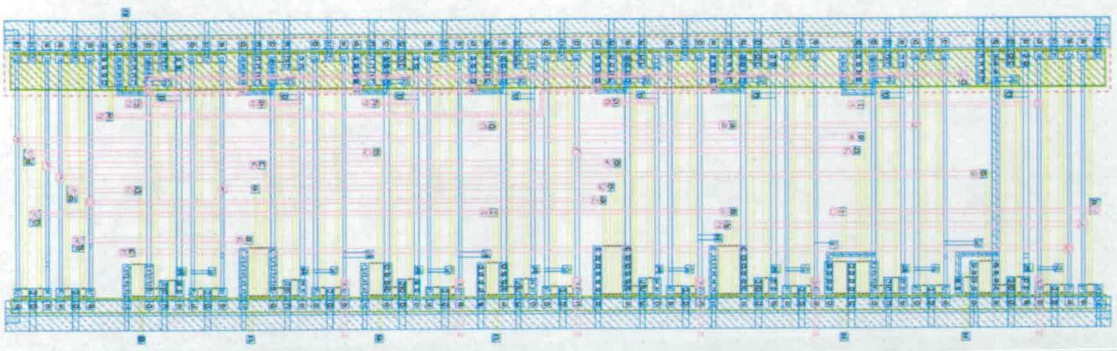


Figure 5.31: Layout of the Memory Unit

5.4.6 Write-Back Unit

The purpose of the write-back unit is to write data back into the SRF. The data comes from one of three possible sources, an ADD instruction, an immediate, or a load (Figure 5.6). The write-back unit is connected to the μ`sync&2ex` which controls the clear and write ports of the SRF, to an 8-bit immediate register and to a three-way multiplexer which feeds to the Z bus, the input data bus of the SRF. The FSM and layout of the write-back unit are shown in Figures 5.32 and 5.33.

As with the memory interface, the write-back FSM leaves its `idle` state, when its corresponding μoperation is asserted, *i.e.* `WzReq`. Its acknowledgement is also generated in the same way, *i.e.* by negating the `idle` state signal. When the FSM has entered state `waiting`, it must effectively decode the current instruction, to decide which path in the FSM is to be followed.

If signals `reqnextX` and `reqnextY` are asserted, then this implies that the result of an ADD instruction is to be writen back and the leftmost path is followed, and state `AEn, startwrite, AOpAck` is entered. Signal `AEn` enables the data output of the adder onto the Z bus, signal `startwrite` initiates the clear/write operation (Section 5.3.6) and signal `AOpAck` is the acknowledgement of the `AOp` μoperation to the control unit. Signal `finished` indicates the end of the write-back operation and after handshaking with the μ`sync&ex&fwd` circuits of the adder (signals `acknextX` and `acknextY`), and as long as the `WzReq` has been lowered, the circuit will return to its `idle` state.

If, on the other hand, the instruction is an immediate one, *i.e.* the two least significant bits of the instruction are zero, then the value contained in the instruction can be written back. As was explained in Section 5.3.6, it is necessary to hold the immediate value in a register. The way this is implemented is the same as in the μcontrol circuits presented in the previous sections. To reduce the

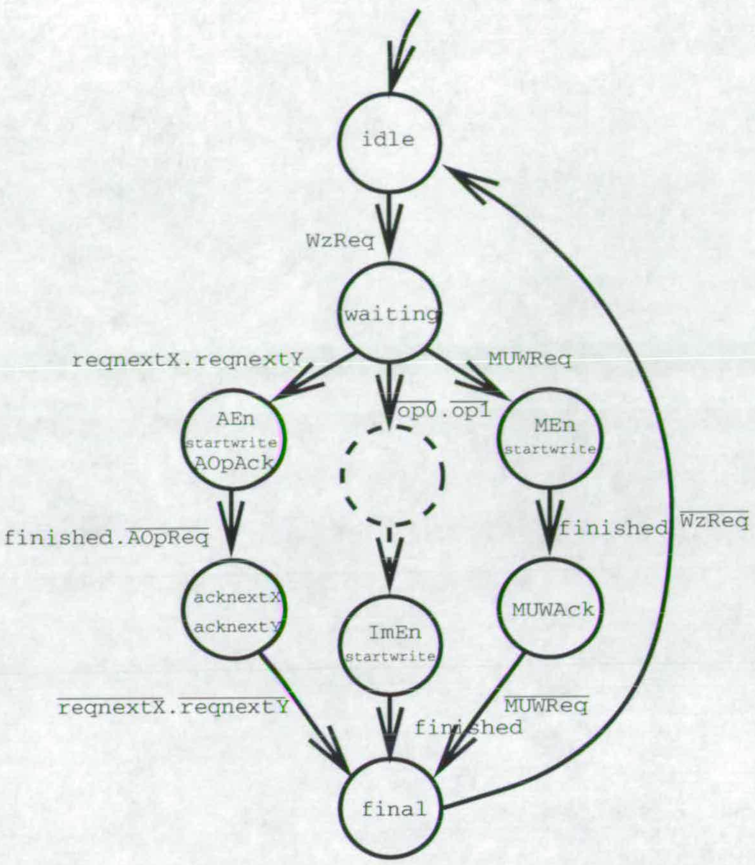

idle
WzReq
waiting
reqnextX.reqnextY
MUWReq
$\overline{\text{op0}}.\overline{\text{op1}}$
AEn
startwrite
AOpAck
MEn
startwrite
finished.$\overline{\text{AOpReq}}$
finished
$\overline{\text{WzReq}}$
acknextX
acknextY
ImEn
startwrite
MUWAck
$\overline{\text{reqnextX}}.\overline{\text{reqnextY}}$
$\overline{\text{MUWReq}}$
finished
final

Figure 5.32: FSM of Write-Back Unit

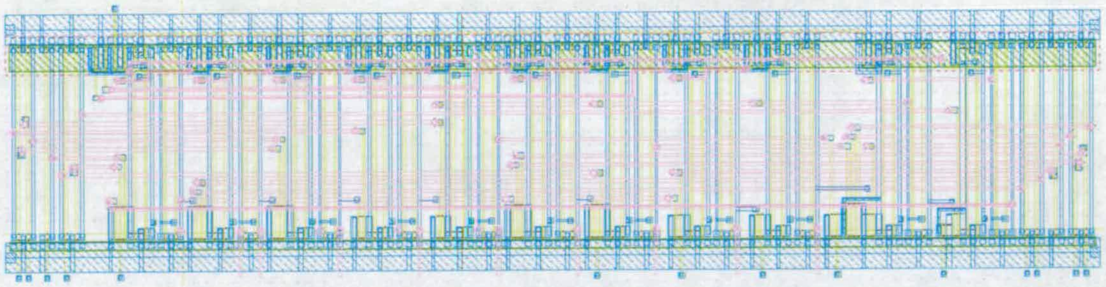


Figure 5.33: Layout of the Write-Back Unit

diagram complexity, the immediate value write to the local data register is drawn as a single dotted state. After the immediate value has been buffered, the `ImEn` signal enables the contents of the immediate register to be multiplexed onto the Z bus and performs the write-back in the same way.

Finally, the assertion of `MUWReq` implies that a load instruction is to write back. State `MEn, startwrite` is entered, and when the write-back has completed, `MUWAck` is asserted. On the completion of the handshake with the memory unit, *i.e.* MUWReq, MUWAck, the write-back circuit completes its operation.

5.4.7 32-bit and 8-bit Adders with Completion Detection

There are two adders in the A1 design, an 8-bit adder in the control unit and a 32-bit adder in the datapath. Their design is identical, except for their bit length, and is based on the use of a ripple carry [WE93]. An n-bit adder is implemented as a chain of n asynchronous full adders. The full adder design implements completion detection by dual-rail coding the input and output carries [Gar93].

$$\boxed{\begin{aligned} sum &= a \text{ xor } b \text{ xor } cin \\ cout &= (a \text{ xor } b)cin + ab \end{aligned}}$$

Figure 5.34: Combinational logic equations for addition

The combinational logic equations for addition can be expressed in the form shown in Figure 5.34, where a and b are the operands of the addition, sum is the result and cin and $cout$ are the values of the input and output carries respectively. As can be seen from these equations, if the two inputs, a and b, are both 0 or both 1, then the value of the output carry does not depend on the input carry bit, so the time required to produce a result is data dependent.

To implement the dual-rail encoding for the carry input and the carry output, they are represented by two signals each, `cin0` and `cin1` and `cout0` and `cout1` respectively. The dual-rail encoding assigns the represented value to the LSB of the dual-rail code and its inverse to the MSB, *i.e.* a 0 is represented by 10 (MSB=1, LSB=0) and a 1 is represented by 01 (MSB=0, LSB=1). Values 00 and 11 do not represent valid data and are used for initialising the circuit. The dual-rail combinational logic is shown in Figure 5.35. The completion of the full-adders can be detected by exclusive-ORing the carry output bits, `cout0` and `cout1`.

The full adder can be divided into two sections, the result generation section (Figure 5.36) and the carry generation and completion detection section (Figure 5.37). Pass-transistor logic [WE93] exclusive-OR (XOR) gates are used for most of the circuit.

The `sum` output is generated by chaining two of these gates. The advantage

$$\boxed{\begin{aligned} cout0 &= (a \text{ xor } b)cin0 + ab \\ cout1 &= (a \text{ xor } b)cin1 + \overline{a}\overline{b} \end{aligned}}$$

Figure 5.35: Dual-rail encoded carry equations

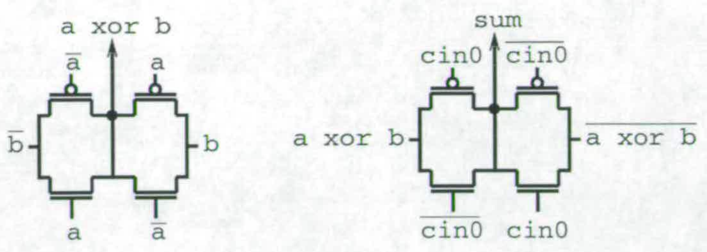


Figure 5.36: Full-Adder, result generation

of this implementation is that it saves on the number of transistors required. The pass-transistor logic implementation uses 16 transistors to implement the result logic, whereas CMOS gates would require 42 transistors. The disadvantage of pass-transistor logic is its low output drive strength, making buffering of such outputs essential.

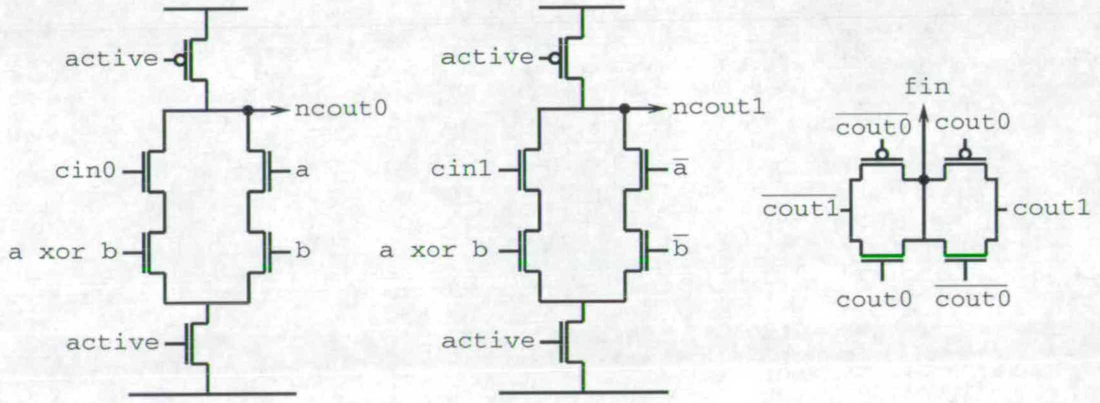


Figure 5.37: Full-Adder, carry output generation and completion detection

The dual-rail carry outputs are implemented using dynamic logic. They are both precharged high by signal `active`, which is deasserted while the full-adder is idle. They then resolve, when `active` is asserted. Outputs `cout0` and `cout1` are generated by inverting the signals `ncout0` and `ncout1`. The `fin` signal is the completion detection output which, when asserted, shows that the full adder is finished.

The layout of the full-adder is shown in Figure 5.38, and the layout of the 32-bit ripple-carry adder implemented by chaining together 32 full-adders is shown in Figure 5.39.

Apart from the 32 full-adder cells, the 32-bit ripple-carry adder contains a completion detection tree of AND gates which implements the global completion detection signal, buffering for the global `active` control signal and control logic for implementing the handshaking protocol. The 8-bit adder is implemented in

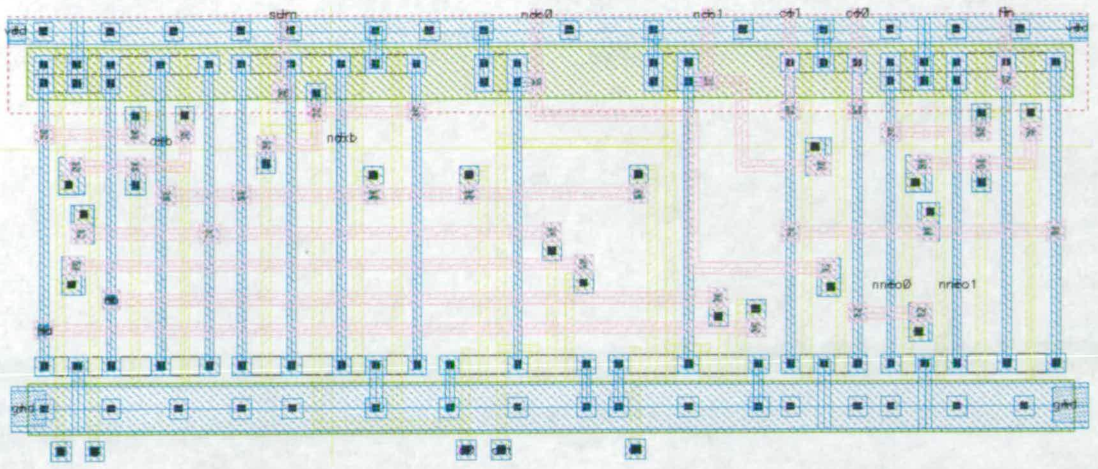


Figure 5.38: Layout of the asynchronous full-adder

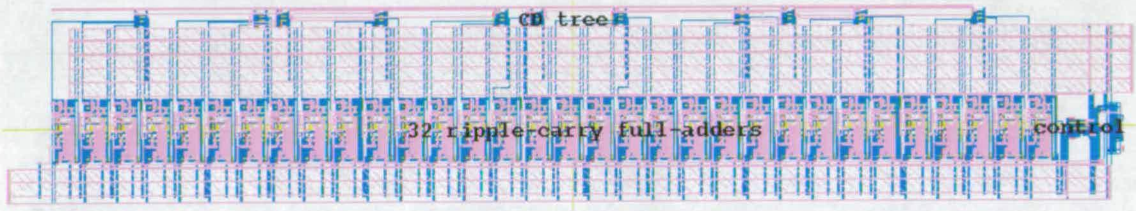


Figure 5.39: Layout of the asynchronous 32-bit ripple-carry Adder

the same way.

The same control logic can be used for implementing an adder of arbitrary bit length. Its FSM and layout are shown in Figures 5.40 and 5.41.

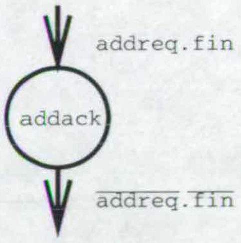


Figure 5.40: FSM of the adder control logic

To save on the number of states, this FSM is implemented as a single CMOS gate with one state. It differs from the AFSM examples seen so far, as there is no active state on reset. The `addack` state is entered by the n-types, and left by the p-types. Both share the same inputs (Figure 5.41). In fact, this FSM is effectively the two-input C-Muller gate [Mil65].

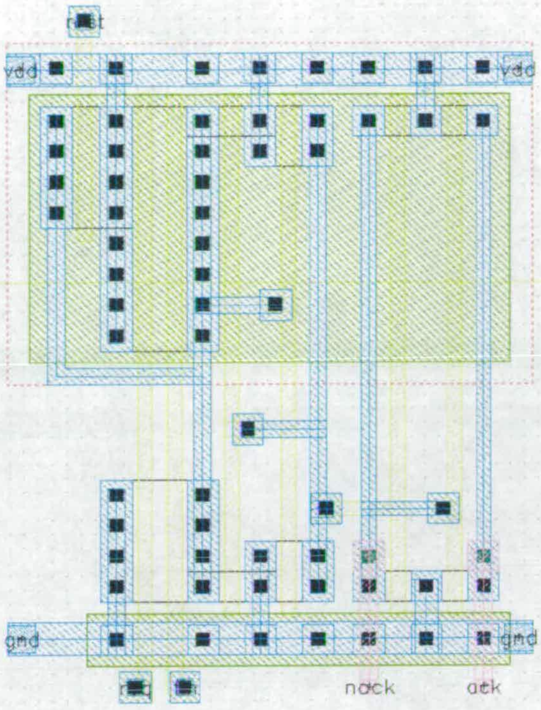


Figure 5.41: Layout of the adder control logic

With this implementation, the speed of addition depends on the input data. The worst-case performance occurs when a carry generated at the LSB propagates sequentially all the way to the last bit of the adder. This occurs when one of the two operands has all its bits asserted (111...1), and the other has only its LSB asserted, *i.e.* is the number 1. Carries that are generated in parallel speed up the addition. It can be seen from the carry out equations (Figure 5.34) that if both inputs to the addition have the value 1, then the value of the carry output is a 1, no matter what the carry input value may be. So, adding zero to zero, *i.e.* all the bits are zero and no carries are generated, takes approximately the same time as adding the two largest values that can be represented by n bits, *i.e.* all the bits are set and n carries are generated in parallel.

The performance of the 32-bit adder was measured in this way, *i.e.* by varying the number of sequentially propagated carries of the addition. This was done by adding the number 1 to a number with a varying number of 1's, from the LSB up.

Figure 5.42 shows the graphical HSPICE simulation output for an addition with no carries and all 32 inputs being zero. This graph shows the handshake signals of the adder, `addreq` and `addack`, labelled `req` and `ack` in the graph and coloured red and green respectively. It also shows the global completion signal, labelled `fin32` in the graph, in blue. The rest of the signals include the `active` control signal, the result outputs and completion signals of the four least-significant bits, `res` and `fin`.

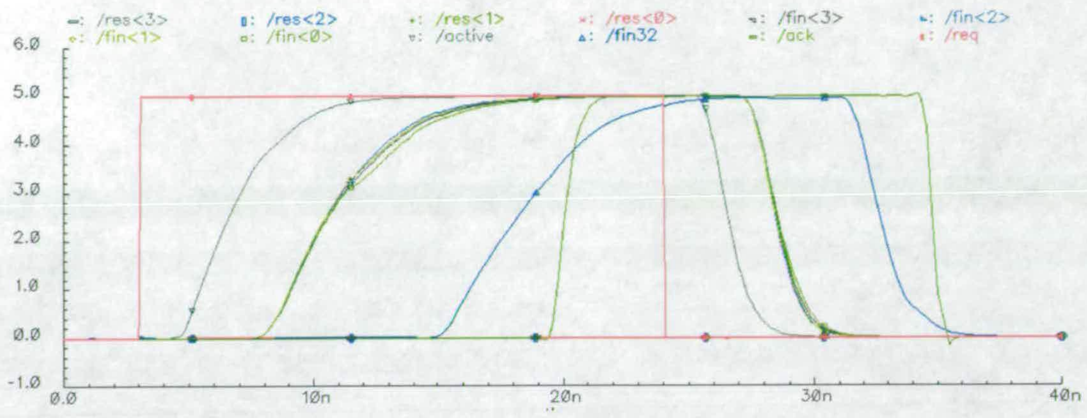


Figure 5.42: Simulation of an addition with no carries

After the assertion of the `req` signal, the `active` signal is asserted, which feeds to the 32 full-adders. Then, all the `fin` signals will be asserted almost simultaneously, as there are no carries. This can be seen in the graph as the four `fin` signals are all asserted simultaneously. Then, the AND-tree will assert the global completion signal, and `ack` will be asserted, indicating that the result is available. When the `req` signal is deasserted, `active` will be deasserted and so will the partial and then the global completion signal, deasserting the `ack` signal.

As can be seen from Figure 5.42, the fastest addition time is 17ns. This is the time it takes to produce the result, *i.e.* the time it takes for the acknowledgement signal to be asserted, after the request signal is asserted. The time to complete the handshake and reset the completion signals is an extra 10.73ns.

Figure 5.43 shows the simulation output when adding the number 1 to itself, *i.e.* the two least-significant bits are asserted. This will produce a single carry at bit 1 of the adder. The carry will not propagate sequentially, but will be absorbed by bit 1.

This graph is similar to the previous one, only the result bit `res<1>` gets asserted. This graph does not contain the negative part of the handshake. The glitch on `res<0>` is caused because the carry inputs to the least-significant bit is only considered after `active` is asserted, therefore the result bit momentarily rises when the inputs arrive. The time taken for the addition is approximately the same, 17ns. As the data will have arrived when the request signal has been asserted, *i.e.* to obey the 4-phase protocol, the carry output of bit 0 is generated before its carry in has been received, so bits 1 and 0 receive their carries

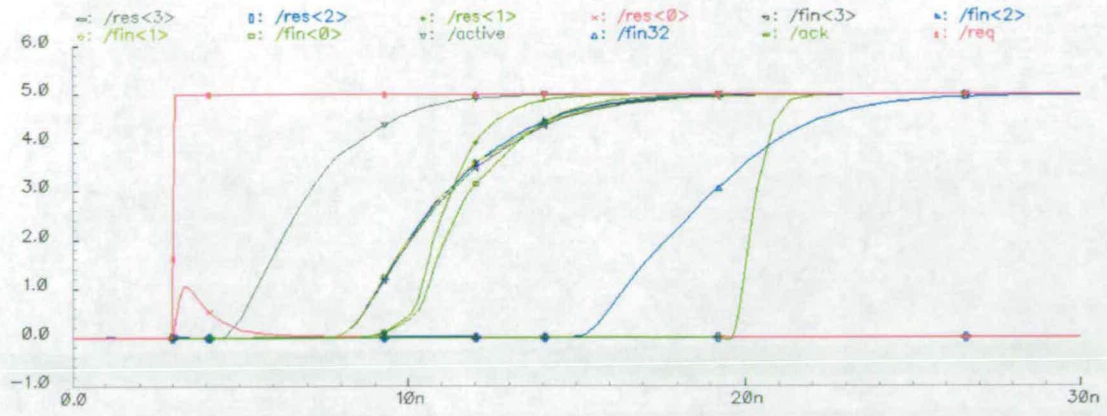


Figure 5.43: Simulation of the 01+01 addition

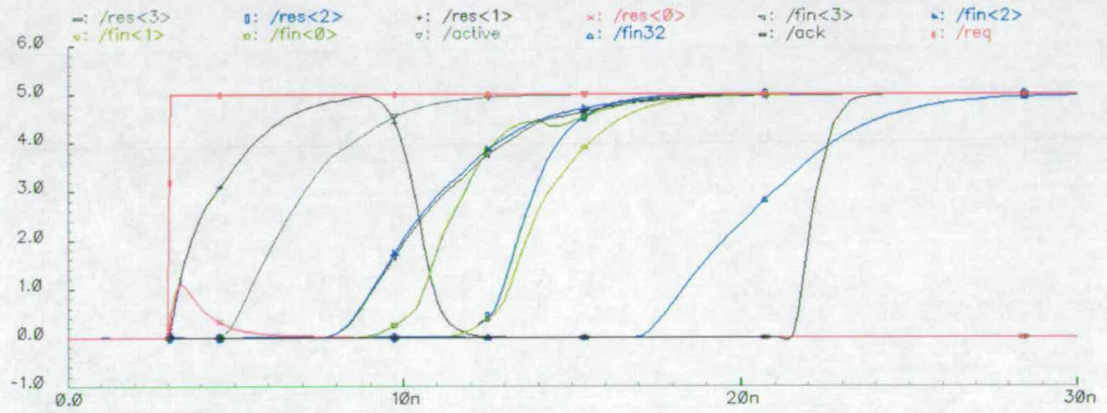


Figure 5.44: Simulation of an addition with one sequential carry, 01+11

simultaneously.

Figure 5.44 shows the simulation output when a sequential carry must be propagated. The addition performed is 01+11. In the graph, it can be seen that now bit 1 of the result gets asserted before its carry inputs arrive, and then returns to zero, for the same reason that the glitch occurs on bit 0. The completion signals of bits 2 and 3, `fin<3>` and `fin<2>` get asserted almost simultaneously. This is because they do not depend on the previous carries. The completion signal of bit 0 is asserted next, `fin<0>`. Signal `fin<1>` depends on the two dual-rail coded carry outputs of bit 0 and is asserted last. This addition takes 19.05ns.

To increase the number of places that the carry generated at bit 0 must be propagated, 1s are appended to the most-significant bit of the second operand. Figures 5.45 and 5.46 show the simulation output for 2 and 3 sequential carries.

The same pattern can be observed here. In Figure 5.45, `fin<3>` is asserted first, and then `fin<0>`, `fin<1>` and `fin<2>`, as the carry propagates. As the carries propagate, the result bits change value in both figures.

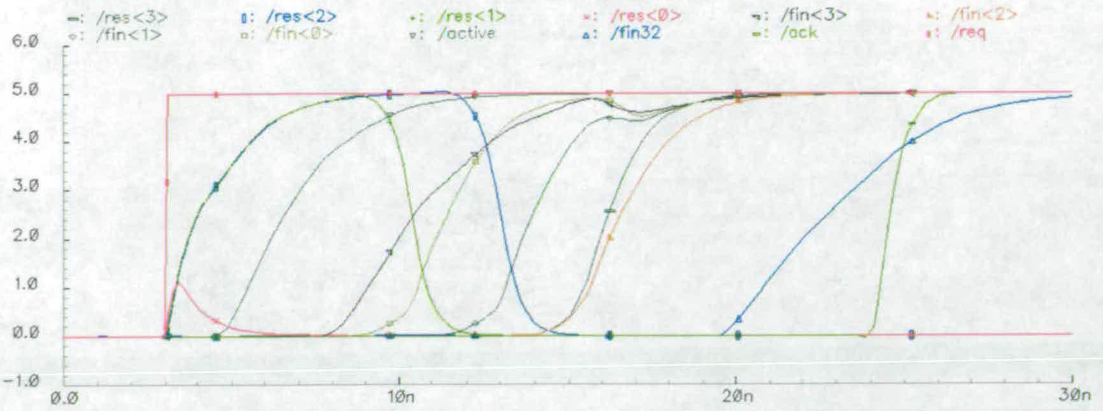


Figure 5.45: Simulation of an addition with two sequential carries, 001+111

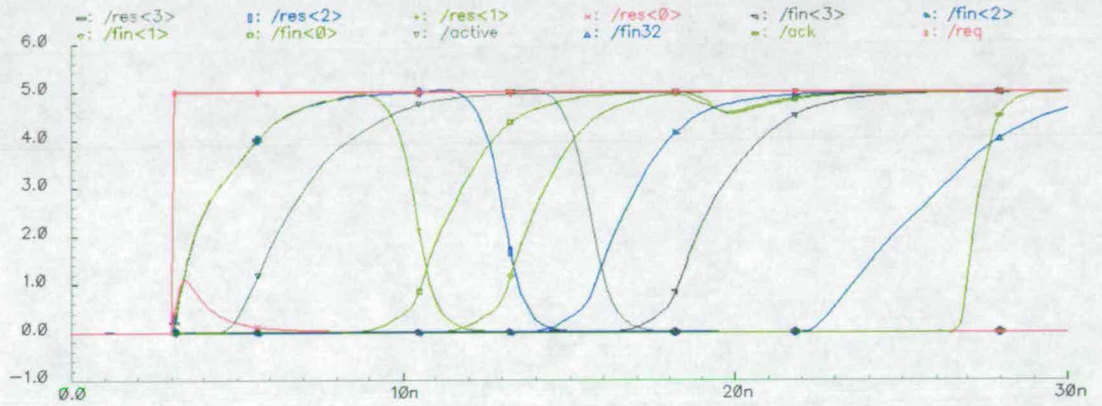


Figure 5.46: Simulation of an addition with three sequential carries, 0001+1111

The addition speed can then be plotted as a function of the number of sequentially propagated carries, Figure 5.47.

The relationship between the addition speed and the number of sequential carries is linear, with the difference between the best and worst case performance being in the order of ten, 17ns for no carries and 95.42ns for 31 carries. In the latter case, n circuit stages must evaluate before the addition completes. This is what limits the performance of a synchronous implementation of a ripple-carry adder; its clock period must be set to the worst-case addition time.

A study at Manchester University [Gar93], that considered both data processing operations and address calculations, reports that the highest percentage of operations contained between 2 and 4 sequential carries. Other operations which occurred frequently contained 16, 12, 32 and 20 sequential carries, in this order. Therefore, as the worst-case addition time is not close to the average case addition time, asynchronous adder design is advantageous and more economical in this case, as it requires no carry look-ahead logic.

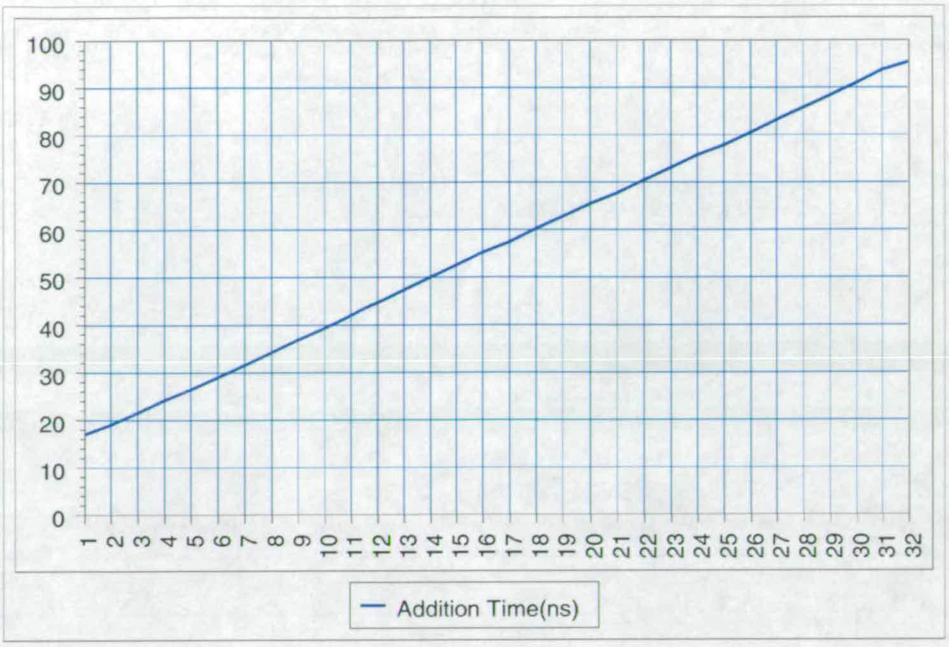


Figure 5.47: Addition Time in ns as a function of the number of sequential carries

5.4.8 32-bit Comparator with Completion Detection

The 32-bit comparator is used by the BEQ branch instruction. It compares two 32-bit numbers and provides two outputs to the control unit, a completion signal, `fin`, and a dual-rail encoded result signal represented by two wires, `z1` and `z0`. If the two numbers are equal, `z0` will be asserted and `z1` deasserted and the opposite otherwise.

The comparator is implemented by exclusive-ORing the bits of the two operands together and then feeding the result bits into a 32-bit cascaded dynamic NAND gate and their inverse to a 32-bit cascaded dynamic OR gate. The outputs of these gates are then exclusive-ORed to provide the completion detection signal. The layout of the 32-bit comparator is shown in Figure 5.48.

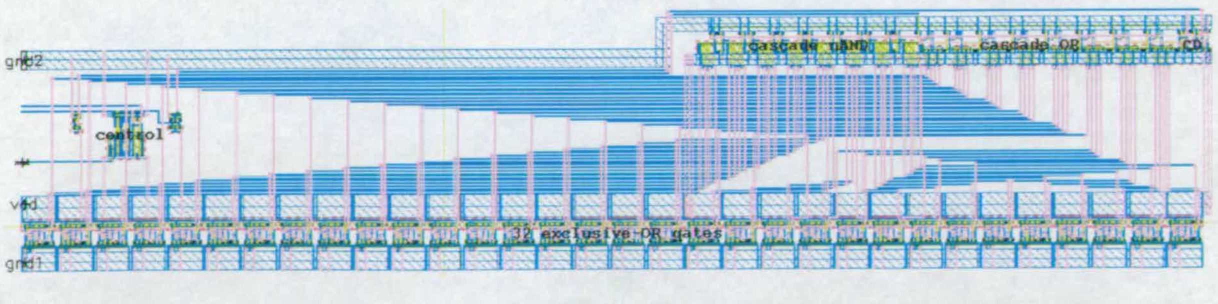


Figure 5.48: Layout of the asynchronous 32-bit comparator

The design of the cascaded dynamic NAND gate is shown in Figure 5.49.

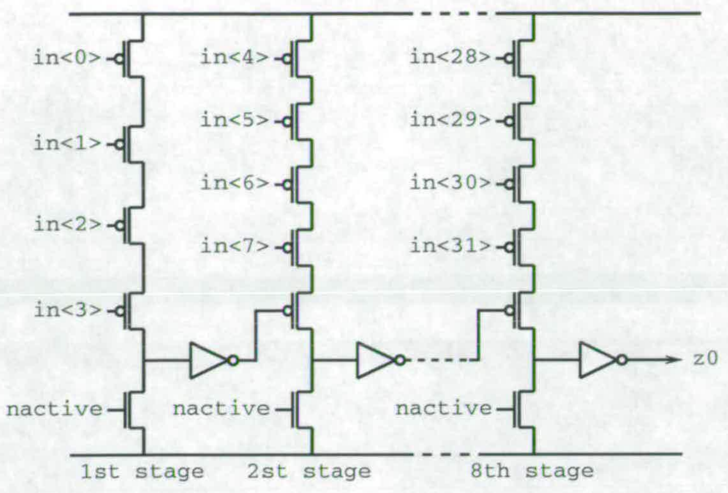


Figure 5.49: cascade 32-input dynamic NAND gate

The inputs to this gate come from the exclusive-OR outputs, the inputs of which are the two comparator operands. The output `z0` is asserted if all the 32 inputs all low, *i.e.* the two operands are identical, and therefore, the result of the comparison is true. The circuit is broken down into eight stages of four inputs each. The output of each stage is inverted and fed onto the next. The cascaded dynamic OR gate is implemented in a similar way. It is shown in Figure 5.50.

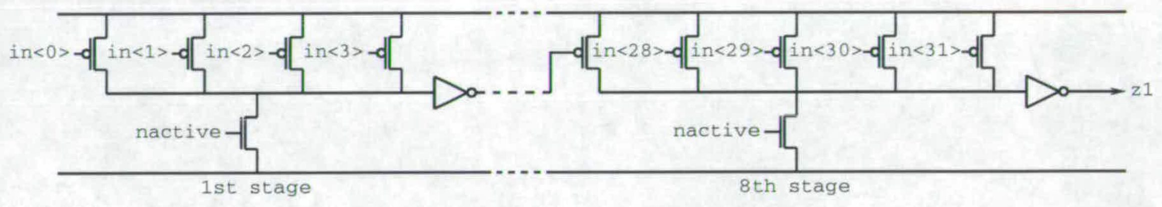


Figure 5.50: cascade 32-input dynamic OR gate

The inputs to this gate are the inverted exclusive-OR (nXOR) of the operand bits. Thus, the output `z1` is asserted if any of the inputs are high, *i.e.* the two operands differ, and therefore, the result of the comparison is false. It is implemented in the same way, *i.e.* with eight stages of four inputs each.

The two cascaded gate outputs `z0` and `z1` are XOR-ed to produce the completion detection signal, `fin`. The control logic of the comparator is identical to that of the adder.

Figure 5.51 shows the graphical HSPICE simulation output for a true and a false comparison. The request signal, `req` is drawn in req, and the acknowledgement signal, ack in green. The completion signal fin is in blue. The two result signals z0 and z1 are drawn in purple and grey.

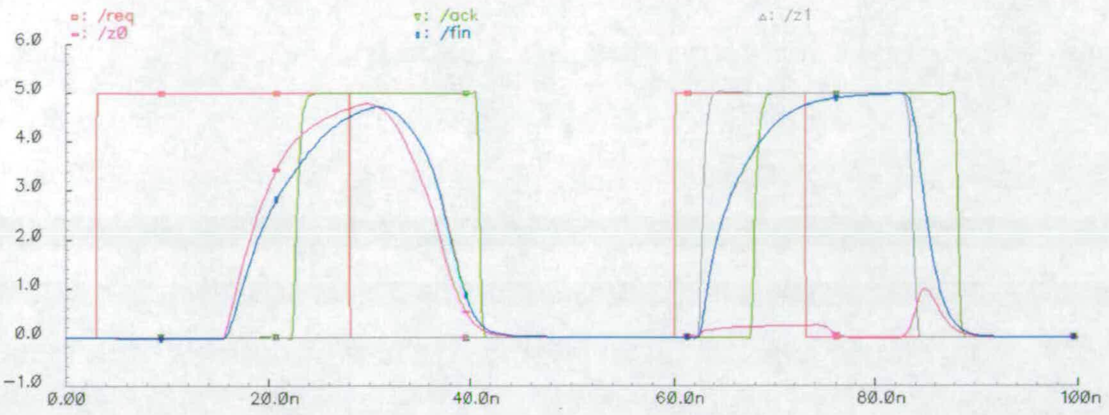


Figure 5.51: Simulation of a true and a false comparison

The first comparison is one which should evaluate to true. After the inputs are applied, z0 and the completion signal fin are asserted. The acknowledgement signal is asserted 19.72ns after the request signal. After the request signal drops, the comparators' signals also drop. The ack signal takes 12.9ns to drop in this case.

The second comparison should evaluate to false. The z1 signal rises a lot faster in this case, as the dynamic OR gate produces the result. It takes 7.95ns for the acknowledgement signal to get asserted in this case, and 15.28ns to return to zero. Signal z1 will take longer to return to zero, as it is the output of the dynamic OR gate.

Hence, a comparison which will evaluate to true will take longer than one which will evaluate to false. The results are summarised in Table 5.2.

	True Comparison	False Comparison
Comparison Time	19.72	7.95
Return Time	12.9	15.28
Total Time	32.62	23.23

Table 5.2: Summary of the comparison speeds

5.4.9 Shared Register Files

Each node of the A1 contains a 2-way Shared Register File, which contains 16 local registers and 8 shared registers. The total number of physical registers of the A1 is 48, whereas the total number of logical registers is 64, as 16 registers are shared.

The layout of the 2-way SRF contained in each node is shown in Figure 5.52.

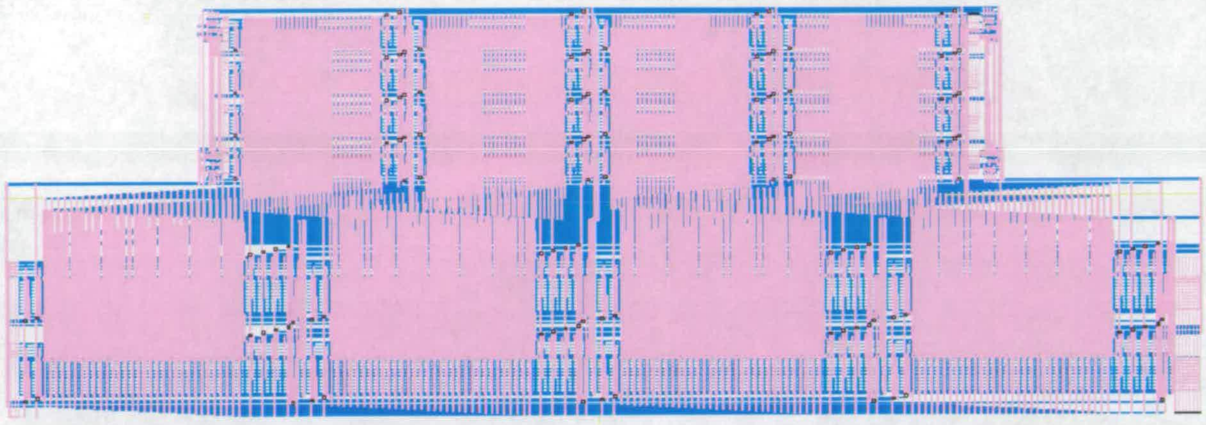


Figure 5.52: Layout of the Shared Register File with 8 shared and 16 local registers

The SRF design was presented in Chapter 4. The A1 SRFs are different in three aspects however. Firstly, they are implemented for a different fabrication process, ALCATEL 0.7μm rather than ES2 0.7μm. Secondly, they have two read ports and one write port, rather than one read port and one write port, to allow instructions with two operands (`ADD` and `BEQ`) to perform two reads in parallel. This implies that shared registers require four read ports and two write ports rather than two read ports and two write ports. So, the A1 SRFs have twice as many read busses. Finally, the control circuit design has been modified, so that the SRF provides the register data of a read access on the positive acknowledgement edge, rather than the negative. In this way, the SRF data can be held active on the SRF bus by holding the request signal high. The SRF connections allow node 0 to access shared registers in node 1 and vice versa.

As in Chapter 3, an access time map can be drawn to measure the access time variation of different registers in the SRF. The access time map of one unconnected A1 SRF is shown in Figure 5.53. This is similar to the 2-way access time map that was shown in Chapter 3, Figure 4.9. The average access times for the different types of accesses are summarised in Table 5.3.

local-read	local-write	shared-read	shared-write
45.04	47.38	49.58	52.02

Table 5.3: Access times(ns) for 2-way SRF with 4, 8 and 16 shared registers

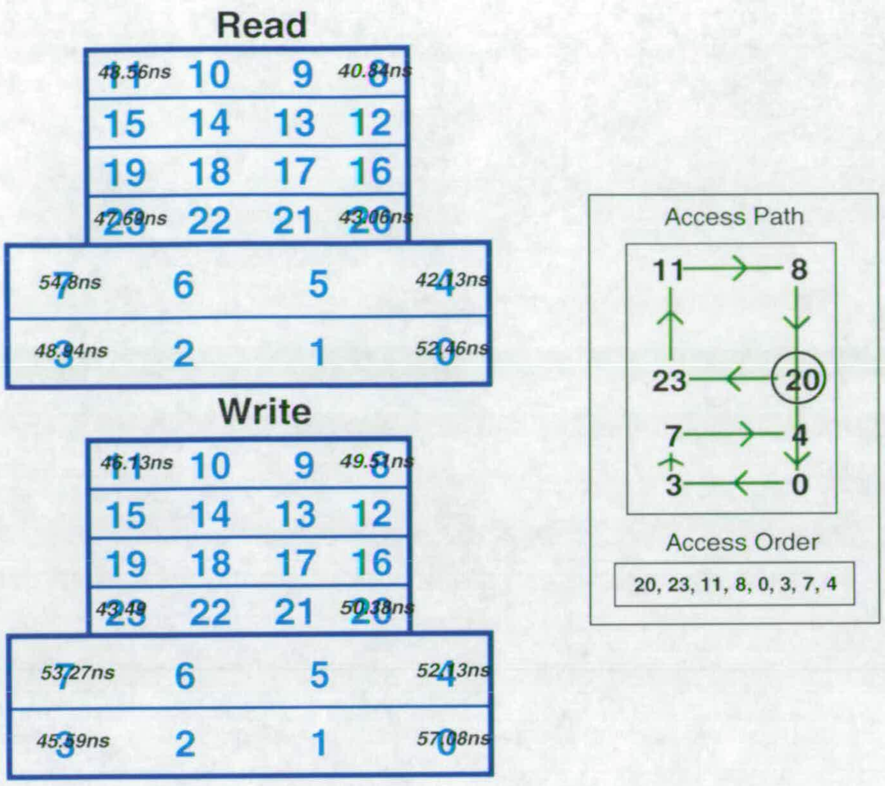


Figure 5.53: Access time map for one unconnected A1 SRF

local-read	local-write	shared-read	shared-write
1.14	1.1	1.19	1.13

Table 5.4: Access time Ratios between A1 ALCATEL SRF and 2-way ES2 1-read port SRF

Table 5.4 contrasts these access times with those of the 2-way SRF with the same number of shared registers but one read port, implemented using the ES2 0.7μm process[3]. Hence, adding a read port and migrating to a new process has increased the access time on average by a factor of approximately 1.14.

As mentioned, the control logic for the read, write and clear ports has been slightly modified. In Chapter 3, the access would complete when the four-phase acknowledgement signal returned low, but this presents problems when the SRF is to be connected to a system, as the data lines cannot be held valid by the request signal. Hence, the control logic had to be changed so that for read accesses the acknowledgement is asserted when the data is available, whereas for write accesses, it is asserted when the data have been written. As in Chapter 3, the read and write port logic is identical.

[3]The simulation temperature is 70°C for both.

The FSM and layout of the read and write port logic are shown in Figures 5.54 and 5.55.

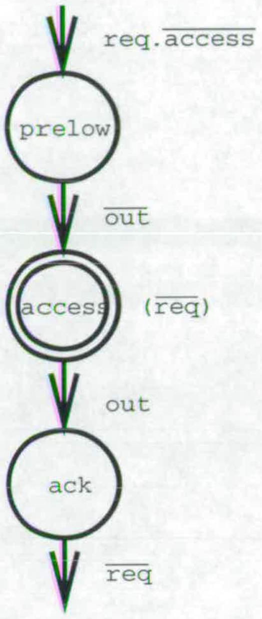

req.$\overline{\texttt{access}}$
prelow
$\overline{\texttt{out}}$
access
($\overline{\texttt{req}}$)
out
ack
$\overline{\texttt{req}}$

Figure 5.54: FSM of the SRF read/write port logic

This FSM does not initialise to any state. The assertion of the request signal, `req` and the fact that the SRF is not already processing an access, *i.e.* signal `access` being low, will take the machine to state `prelow`. This state must precharge the SRF bus completion signal `out` to low, so that a low→high transition can be detected. When `out` is low, the FSM will enter state `access` which enables the column decoders to select the appropriate register (*c.f.* Chapter 3). When the input `out` has been asserted, *i.e.* the valid bit of the selected register has been asserted, then the FSM will enter state `ack`, without leaving state `access`. This holds the data valid until the `req` input is deasserted. Then, both states `ack` and `access` will be left, and another access may be serviced.

The clear port control logic is similar. The only two differences are that the precharge signal precharges high, in order to detect a high→low transition, and the order of the input `out` transitions is reversed in the FSM, *i.e.* state `prehigh` of the clear port FSM is left when `out` is high and state `ack` is entered when `out` is low.

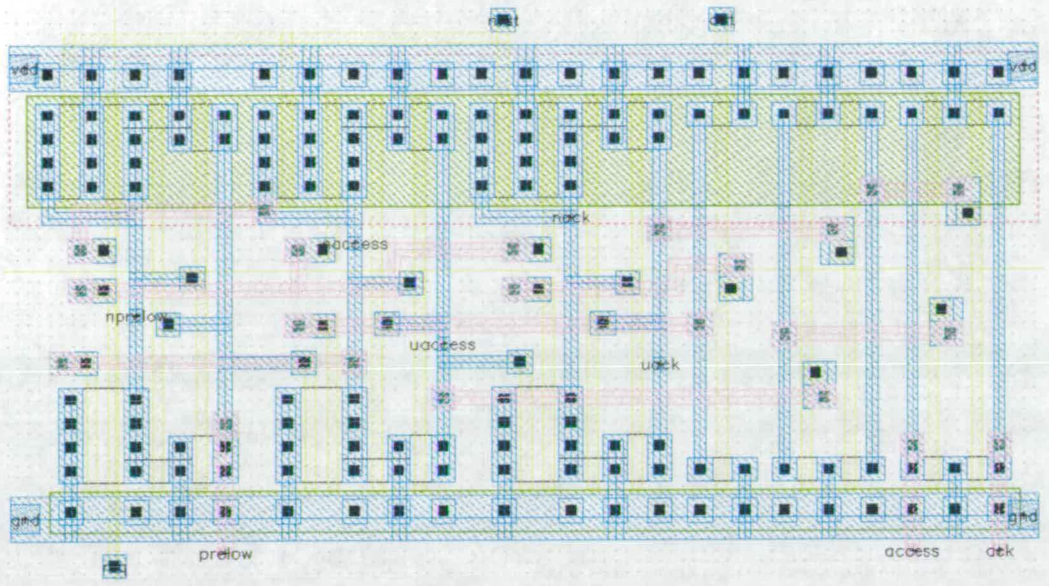


Figure 5.55: Layout of the SRF read/write port logic

5.5 A1 Testing and Simulations

Two tests been used to verify the correct operation of the A1 processor, a simple program test and an SRF addition test. The simple program test, which executes on a single node (the `UP_node` circuit, Figure 5.3), verifies that all of the processor instructions execute correctly. The SRF addition test executes a program that performs a short series of additions on one and two nodes of the entire chip, to investigate the effect of communicating data through a shared register.

5.5.1 Simple Program Test

Figure 5.56 shows the simple program test and the μoperations corresponding to each program instruction. It consists of two immediate instructions, one add, two memory instructions and a branch. The result of register r4 (the number 68), which is stored in memory, verifies that the first three program instructions have been executed correctly. The value of PC after the branch instruction is executed, verifies that the branch instruction is correctly executed.

Both the result of register r4 and the PC value were found to be correct therefore demonstrating that the program was executed correctly. Figures 5.57-5.61 show the HSPICE graphical simulation outputs of various datapath signals.

Figure 5.57 shows the instruction fetch part of the control unit, *i.e.* the fetch handshake and the PC. The first instruction is fetched at 6ns simulation time, and the last, *i.e.* the branch instruction, at 1050ns. As can be seen from

Instruction	Rx	Ry	Wz	AOp	MOp	COp
LI r2, 65			X			
LI r0, 3			X			
ADD r4, r2, r0	X	X	X	X		
LD r5, 0			X		X	
ST r4, 0					X	
BEQ r4, r5, -4						X

Figure 5.56: Simple test program for testing all the A1 processor instructions

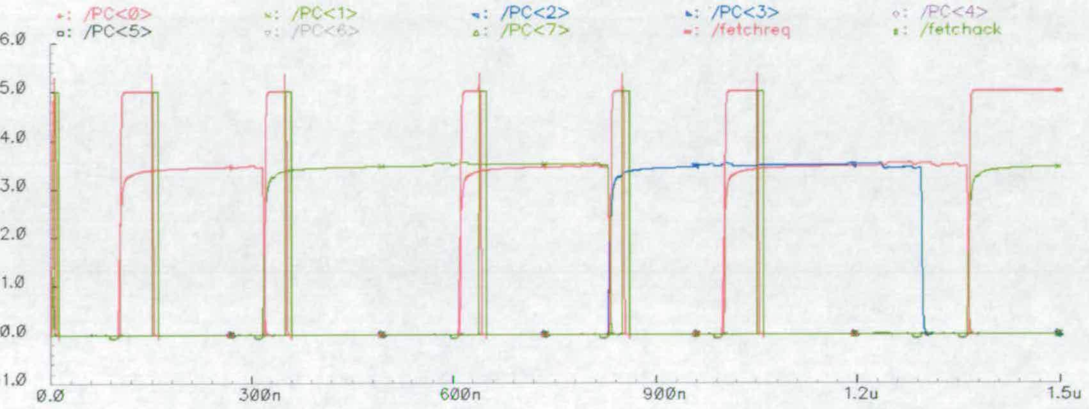


Figure 5.57: Simulation of the simple program - Instruction Fetch

the diagram, the PC value increases by one at every instruction fetch until the branch instruction is executed.

Figure 5.58 shows the register μoperation handshakes of the processor control unit, *i.e.* the handshakes of μoperations `Rx`, `Ry` and `Wz`. In the top diagram, the two `Rx` handshakes correspond to the `ADD` and `BEQ` instructions. In the middle diagram, the `Ry` handshakes correspond to the `ADD`, `ST` and `BEQ` instructions. Finally, the four `Wz` handshakes of the bottom diagram correspond to the first four instructions.

The exploitation of parallelism between different μoperations can be observed in these three diagrams. The write-back μoperation of the second instruction (`LI r0, 3`), `Wz`, is executed in parallel in the datapath with the `Rx` and `Ry` μoperations of the third one (`ADD r4, r2, r0`). Also, the write-back μoperation of the fourth instruction (`LD r5, 0`) is executed in parallel with the `Ry` μoperation of the fifth (`ST r4, 0`).

Figure 5.59 shows the FU μoperation handshakes, *i.e.* `AOp`, `COp` and `MOp`. As can be seen by the top diagram, the assertion of the `AOpAck` acknowledgement signal is delayed, compared to the acknowledgement signals of the other μoperations. The `AOpAck` does not get asserted until the addition has been performed and the

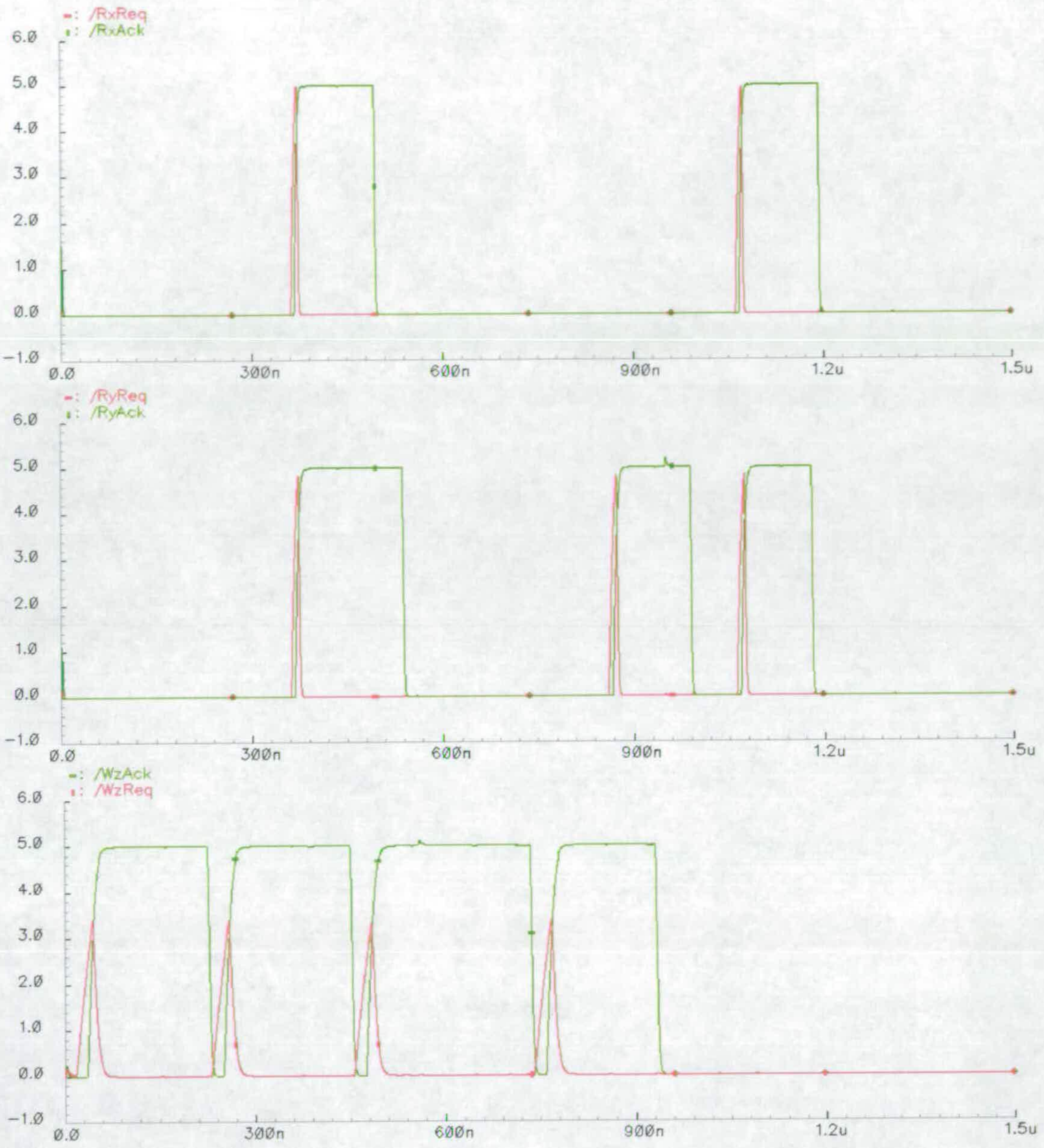


Figure 5.58: Simulation of the simple program - Register μoperation Handshakes

result has arrived at the write-back unit (*c.f.* Section 5.4.6, Figure 5.32). The reason for this is the lack of a datapath register for storing the adder's result. The fact that the result is not stored in a datapath register, implies that the handshakes of the adder and of the add μoperation must not complete until the data has been written back. The former is achieved by the two μcontrol circuits of the adder (μ`sync&ex&fwd` and μ`sync&ex&fwd-m`). The latter is achieved by generating the `AOpAck` signal from the signal that enables the adder's result onto the Z bus (signal `AEn, startwrite, AOpAck`, Section 5.4.6, Figure 5.32). Although this implementation fulfills the necessary requirement, it would have been

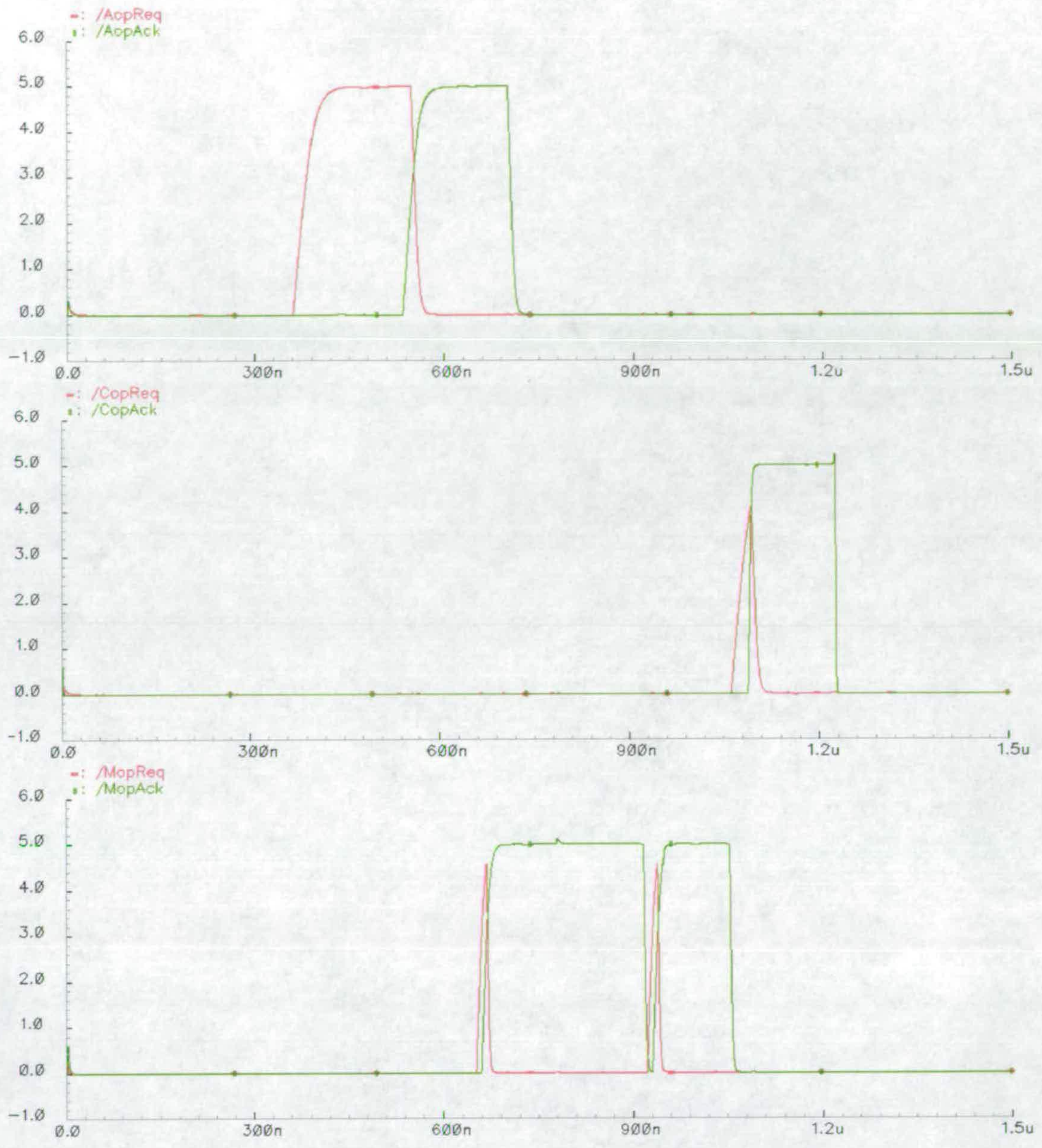


Figure 5.59: Simulation of the simple program - FU μoperation Handshakes

beneficial to assert the `AOpAck` signal sooner, as this would issue the instruction faster.

Figure 5.60 shows the RF port handshakes. The two top panels show the handshakes of the first and second RF read ports respectively, *i.e.* `Read1` and `Read2`. The first read operations on both ports are performed by the `ADD` instruction. The second read on port 2 is performed by the `ST` instruction and the last two reads of both ports are performed by the `BEQ` instruction. The third and fourth panel show the handshakes of the clear and write RF ports respectively. A clear always precedes a write to eliminate possible instruction hazards. As can be

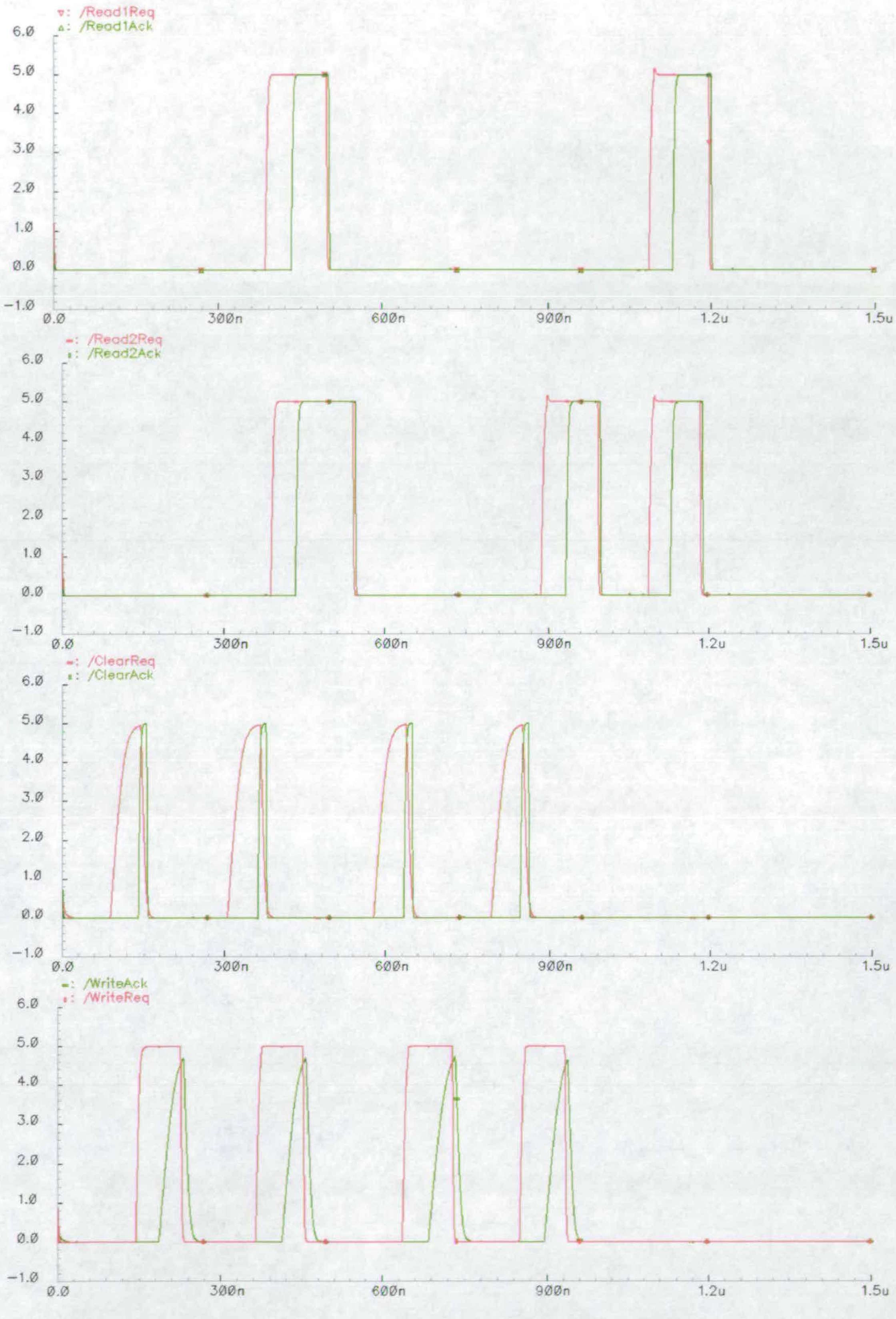


Figure 5.60: Simulation of the simple program - Register File Ports

seen by the diagrams, the register write of the second instruction (`LI r0, 3`) and the register read of the third (`ADD r4, r2, r0`) coincide. Similarly, the register write of the fourth instruction (`LD r5, 0`) coincides with the register read of the fifth (`ST r4, 0`). By clearing the register valid bit possible register dependencies between these instructions are respected. In the case of a RAW hazard, a register read will not complete until the register valid bit is set, *i.e.* a pending register write has written its result. Hence, this simple register locking mechanism synchronises the instructions' data without the need for a centralised complex data structure such as a scoreboard.

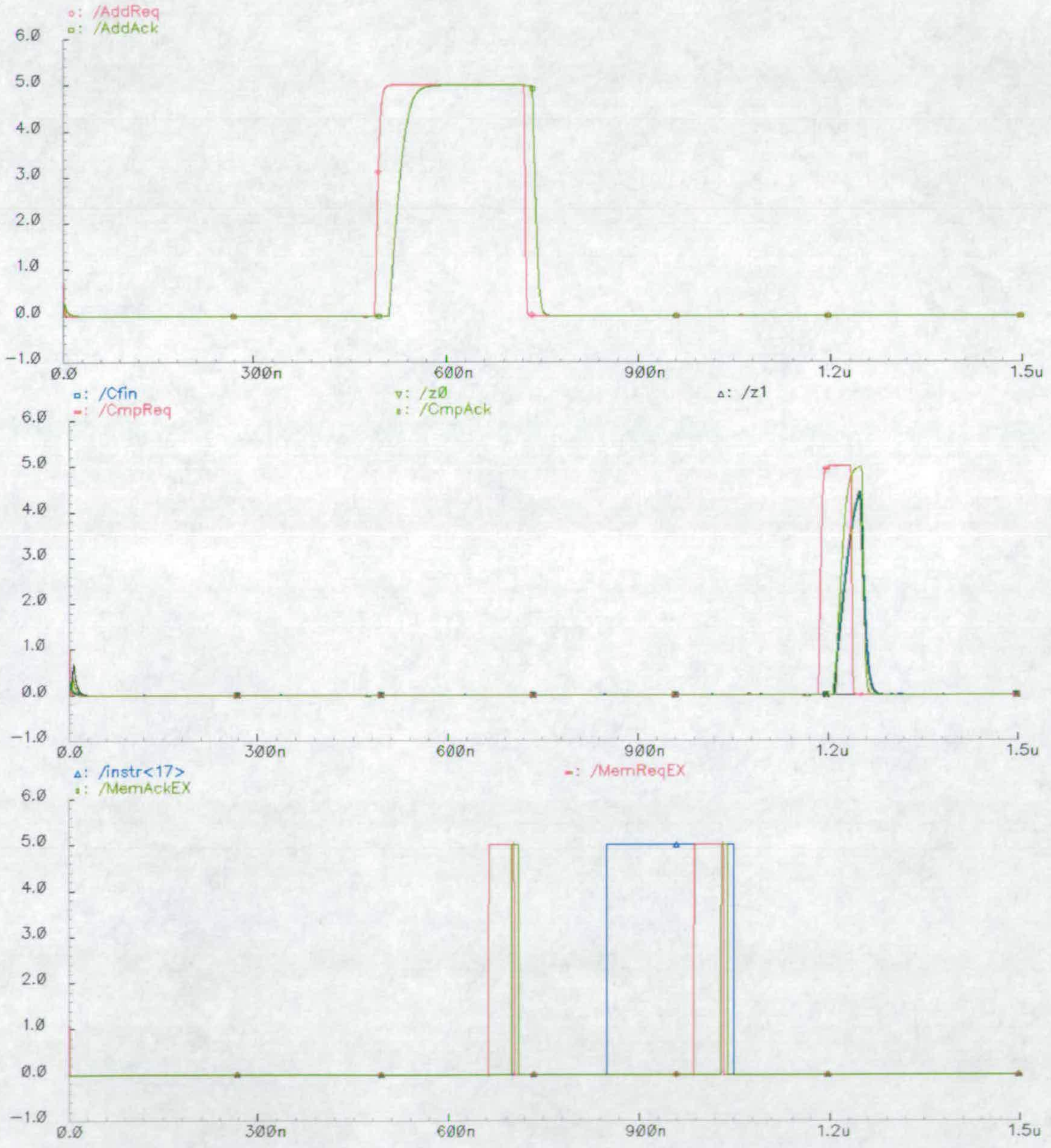


Figure 5.61: Simulation of the simple program - FU Handshakes

Figure 5.61 shows the FU handshakes. The top panel shows the handshake of the adder, the middle one the handshake of the comparator and the bottom one the handshake of the memory unit. The parallelism in the datapath is also evident here. Part of the addition handshake coincides with the load from memory. As can be measured by the top panel, the addition (65 + 3) takes approximately 40ns and the comparison takes approximately 36ns.

5.5.2 SRF Addition Test

Figures 5.62 and 5.63 show the one and two-node addition programs respectively.

node 0
LI r8, 1
LI r9, 2
ADD r10, r9, r8
LI r11, 3
LI r12, 4
ADD r13, r11, r12
ADD r14, r10, r13
ST r14, 0

Figure 5.62: SRF Program - one node addition

The one-node addition program performs three sequential additions in order to add four numbers together. The two-node program performs the first two of these additions in parallel and then using a shared register adds their results, in a similar manner to the program presented in Section 4.2.2.

node 0	*node 1*
LI r8, 1	LI r8, 3
LI r9, 2	LI r9, 4
[**SW**] ADD sr0, r9, r8	ADD r10, r9, r8
	[**SR**] ADD r11, r10, sr24
	ST r11, 0

Figure 5.63: SRF Program - two node addition

As with the simple program, the value stored into memory, *i.e.* register r14 in the one-node program and register r11 in the two-node program, verify that the program executes correctly. Both of these programs were simulated using HSPICE and were found to produce the correct result (*i.e.* the number 10).

Figures 5.64 and 5.65 show the HSPICE graphical simulation outputs of the instruction fetch handshakes for the one-node and two-node programs respectively.

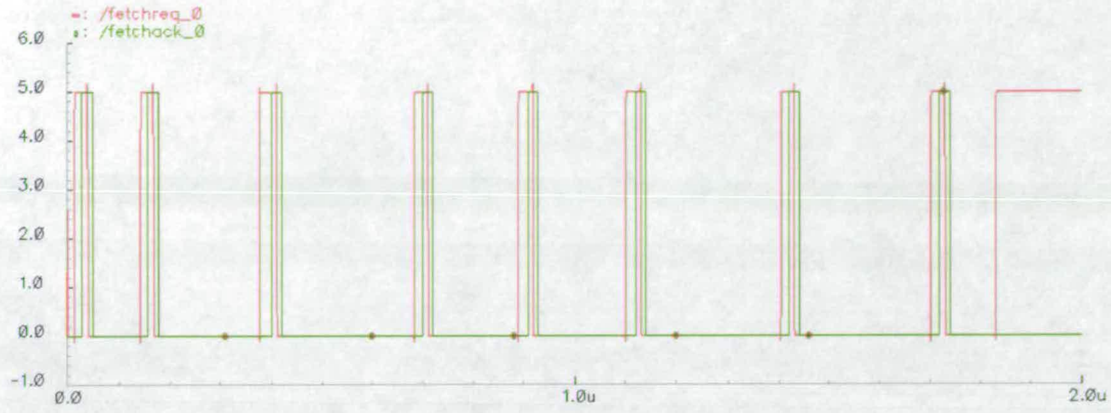


Figure 5.64: One-node SRF addition program simulation - Instruction Fetch

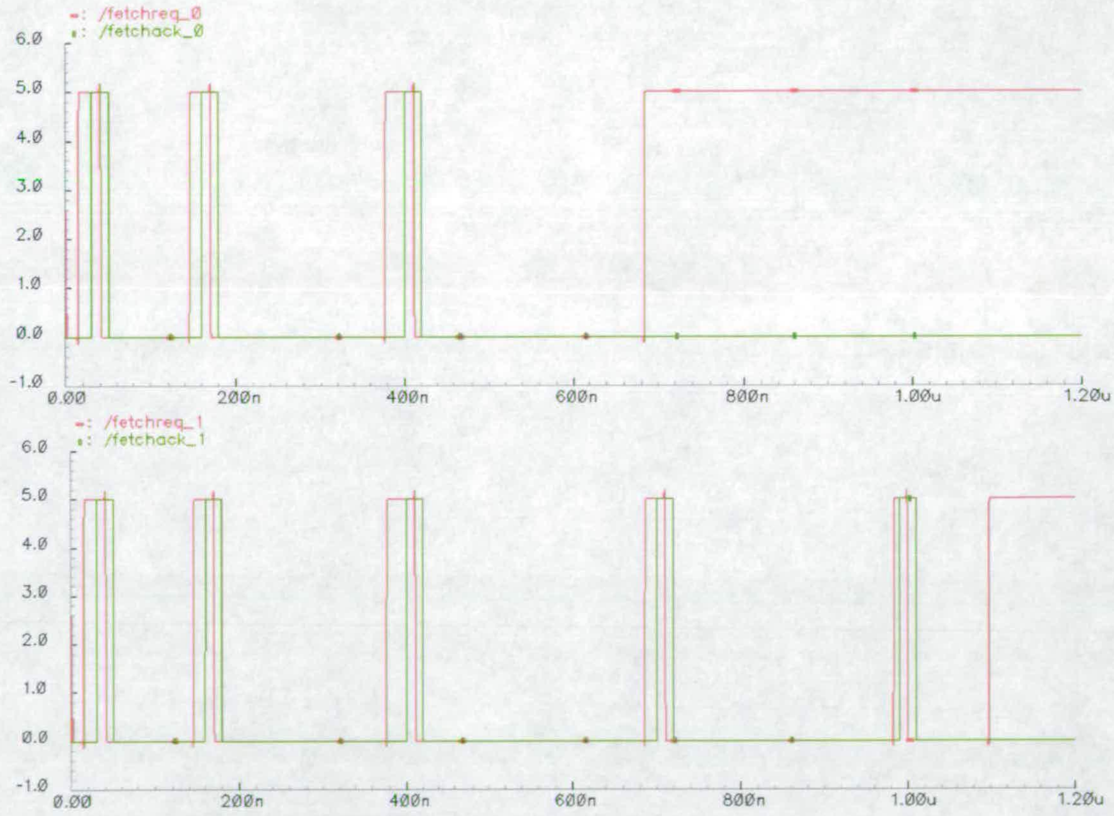


Figure 5.65: Two-node SRF addition program simulation - Instruction Fetch

In general, the timing of instruction fetching is not directly proportional to the execution time of a program, as the fetching of an instruction depends on the issue of the previous instruction into the datapath, not on the completion of its execution. In this case however, the two programs that are being compared are identical instruction per instruction. In addition, there is a chain of dependencies between the instructions and all of the program instructions except the last one, *i.e.* the store, use the write-back unit, and hence cannot be issued before their predecessor has completed its execution. Therefore, in this case the instruction fetch timings are proportional to the instruction execution times.

For the one-node program after all the program instructions have been fetched and issued, the fetch unit asserts the `fetchreq` signal at approximately 1.84μsecs. For the two-node program, after the final instruction of node 1 has been fetched and issued, the fetch unit asserts its `fetchreq` signal at approximately 1.1μsecs. These timings indicate that the two-node program is approximately 40.2% faster than the one-node program. The speedup value is approximately 1.7.

5.6 Conclusions

In this chapter, the structure and implementation of the A1 prototype processor have been described. The A1 prototype processor is a fully asynchronous, μnet-based, shared register file, dual-node multiprocessor architecture. The two processor nodes are identical, and execute instructions independently of each other. Communication between them can only take place through the shared register mechanism. The two processor nodes use the μnet approach for exploiting fine-grain parallelism at the instruction level. The architecture of the nodes has been described, along with the implementation of the various processor components and their performance. Two test programs were shown to demonstrate and verify the processors' operation, a simple single-node program and an addition program which can run on one or both processor nodes. This demonstrated the ability to exploit parallelism in the μnet datapath.

Chapter 6

Conclusions and Future Work

The main outcome of the work described in this thesis has been the design and implementation of the A1 prototype processor. The A1 prototype was implemented by combining together the various techniques described in the earlier chapters. The control circuit design was implemented using the asynchronous CMOS direct-mapped FSM approach, the μnet approach was used to implement the architecture of the processor nodes, and communication between the nodes was implemented using Shared Register Files.

The A1 processor simulations have demonstrated that the processor functions correctly as it correctly executes all of its instructions and it also correctly executed the two test programs. The execution time of instructions is variable, depending on the instruction type and its operands. The A1 can exploit fine-grain parallelism in the datapath between μoperations. It can also exploit program level parallelism by executing parts of the same program on different nodes and communicating the results through shared registers. These results demonstrate the implementation feasibility and viability of the approaches which were followed, both at the circuit and at the architectural level.

6.1 A1 Evaluation

6.1.1 Parallelism

The simulations demonstrated the potential of the A1 to exploit fine-grain parallelism. The μoperations of different instructions and their datapath operations overlap. In particular, the μoperation handshakes of the FUs, and the FU handshakes themselves overlap. This fine-grain parallelism would not have been exploited by a pipeline, as all of these FU operations would have been merged into a single pipeline stage. In addition, by exploiting asynchrony, the execution time of instructions depends on the instruction type and on the operands of the instruction. These are all benefits of the μnet approach; it can not only exploit temporal parallelism, but both also spatial parallelism. By introducing SRFs and implementing a dual-node architecture, program level concurrency can also be exploited. With the dual-node architecture it is possible for the two processors to run either independent code, or "threads" of the same program. These threads can synchronise and communicate through the shared registers. Hence, explicit communication instructions or locks for synchronisation are not necessary. As the SRF approach is scalable, it is possible to increase the number of shared registers, if more data need to be shared between threads, or the number of nodes, if more threads executing concurrently are needed.

The amount of fine-grain parallelism which can be exploited depends on numerous factors. The decoupling of μoperations is important. In the A1 processor, a latch register was not included at the output of the adder. This register could be included in a revised design. The impact of the lack of this register is that the addition μoperation is delayed as the adder itself must wait until the data has been written back. This reduces the amount of parallelism that can be exploited, as another addition μoperation, if available, cannot be issued.

It can be concluded that to decouple operations which produce results it is necessary to insert latch registers for both the inputs and for the output of the operation.

6.1.2 Performance

It is common to use the clock frequency of a synchronous processor as a measure of performance. A more accurate performance measure is the average number of instructions executed per second, *i.e.* MIPS (Million Instructions Per Second). But even this metric is not accurate enough, as different instructions have different execution times even on a synchronous machine, so the instruction rate is not constant.

In an asynchronous processor, it is even harder to give a general measure of performance, as the execution rate and thus the measurable performance of the A1 processor is even more variable depending on the program executed than that of a synchronous processor. The ordering of instructions and the data that these instructions operate on determine their execution rate, firstly because of the execution of instructions depends on the state of the datapath and the dependencies between them and secondly because the latencies of the FUs are data-dependent.

To get an idea of the A1 processor performance, the rate of instruction execution can be approximated by the test programs executed in Section 5.5. Unfortunately, it is impossible to accurately measure the instruction execution rate because of the nature of the architecture. The instruction fetch rate can be measured though, giving an idea of performance. In the simple program, Section 5.5.1, each instruction takes approximatelly 228ns to be issued. In the SRF addition test program, Section 5.5.2, the average issue time for the one-node program is 230ns and for the two-node program 224ns. From these timings an equivalent average clock frequency can be calculated as 4.39MHz.

This clock frequency implies an approximately 228ns period for a synchronous system. For the 0.7μm process used, the input to output, low-to-high gate delay is approximately 0.6ns, whereas the high-to-low gate delay is approximately 1ns (for an unloaded, minimum transistor-size gate).

Therefore, the performance of the A1 processor is evidently not as good as would be expected for a processor implemented using this technology. The reason for this is not due to the processor architecture itself, but because little effort was made to optimise the transistor sizing in the circuit implementation.

Most of the transistors in the A1 design are minimum size, except for circuit parts where sizing transistors was necessary in order to achieve correct circuit operation. These parts include the register cells and the AFSM control circuits. In addition, most control signals were not buffered to increase their drive strength, except for the enable signals of the 32-bit registers, where CMOS chain buffers were used. It is expected that the A1 processor performance would be greatly improved if the transistor sizes were optimal, but unfortunately, one of the early decisions in this research was that it would be too time-consuming for transistor sizing analysis to be performed.

6.1.3 Design Problems

The A1 processor simulations also identified a minor design problem, which will have to be rectified if the A1 prototype is to be fabricated. It is a timing violation which occurs in the memory output port for `ST` instructions.

The request signal for the memory write is not synchronised with the data coming from the RF, hence violating the handshaking protocol. When the circuit was designed it was assumed that the data would have arrived by the time the request was asserted. Unfortunately, this is not the case as the simulation showed, which shows both the importance of simulation and the fact that such global assumptions about relative delays should generally not be made. Solving this problem requires additional synchronisation to be implemented between the memory interface and the register read port. The external memory request should only be asserted after the RF port has asserted its acknowledgement signal, *i.e.* when the data is on the output bus. The logic required to implement this is straightforward, one extra pull-down transistor and one metal route.

6.1.4 Silicon Areas

A practical aspect that the A1 implementation demonstrated is the silicon area required to implement the various processor components. Table 6.1 shows the areas of some of the processor components.

Component	mm^2
Shared Register File	26.3
Adder	1.1
Control Unit	0.5
Comparator	0.3
32-bit Latch Register	0.1

Table 6.1: Areas of various A1 components

The largest, by far, is the size of the SRF, with the 32-bit adder being the second largest component. This shows the importance of knowing the optimal sizes of local and shared registers.

6.1.5 Testing

If the A1 processor, or an evolution of it, is to be fabricated, its functionality will have to verified. Testing the processor requires connecting it to a small system. This must include asynchronous instruction and data memories, so that the processor can fetch the instructions and read and write data. One quick and cost-effective way of implementing them is using programmable logic. Then, the processor and the two memories can be connected on an experimental board and the system can be tested.

The testing mechanism that the A1 processor contains is the `STOP` input pad. The `STOP` input controls the instruction fetch, as was mentioned in Section 5.4.3.1. By connecting this input to external logic and a switch it is possible to single step the program being executed and the status of the datapath to be frozen at the end of each instruction. To inspect the status of the datapath additional test pads will have to added. For more thorough testing before the fabrication of this or future versions of the prototype, Built-in Self-Test (BIST) [KBJND96] techniques can be added.

One of the established BIST techniques is scanning. Scanning involves serially loading, via an external test interface, the inputs of a circuit part, letting the values propagate through the logic and then serially reading its outputs. This technique permits loading of multiple inputs and reading of multiple outputs by a single input and a single output pin. In this way the different circuit parts can be tested. In synchronous circuits, scanning is implemented by serial shift registers, also called scan chains.

Although the current implementation of the A1 does not include BIST, the design of the CMOS direct-mapped AFSMs can be extended so that a future implementation will. Figures 6.1 and 6.2 contrast a conventional AFSM structure with one with BIST additions.

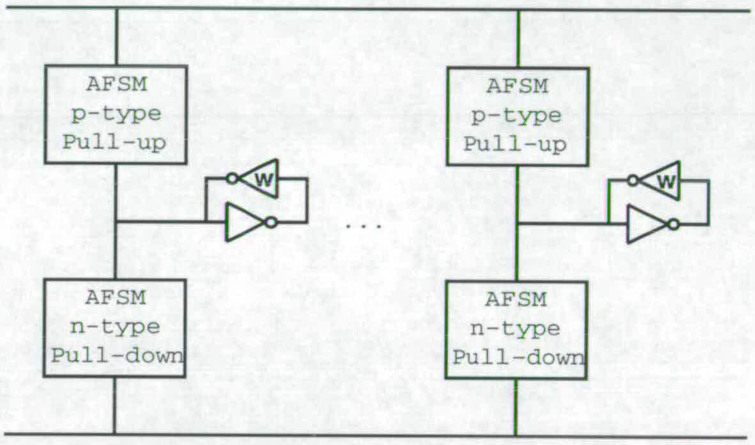


Figure 6.1: Conventional CMOS direct-mapped AFSM structure

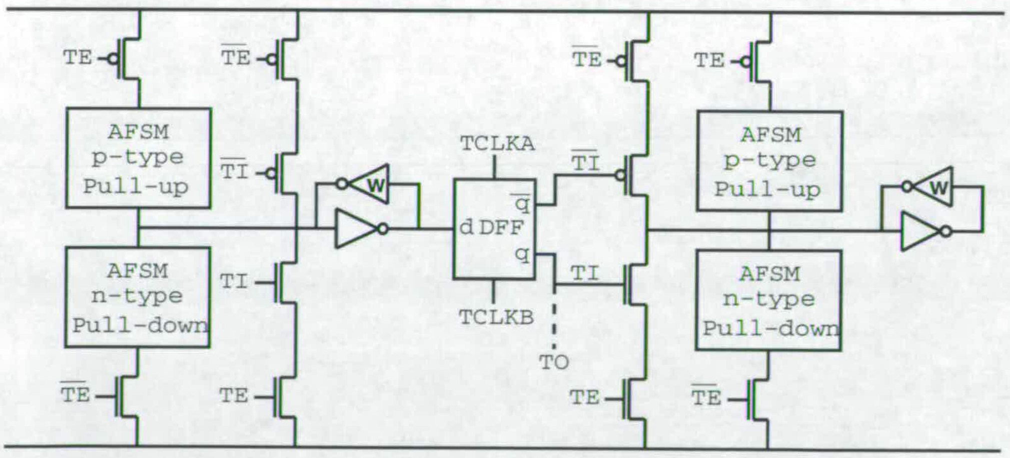


Figure 6.2: CMOS direct-mapped AFSM with BIST additions

These BIST additions allow for the loading and unloading of the AFSMs states via a synchronous test interface. Five signals are added, TE (Test Enable), TI (Test Input), TO (Test Output) and two clock signals (TCLKA and TCLKB). Six transistors are added to each state gate of the AFSM structure. Two transistors, one n-type and one p-type are connected to the n and p-type networks of the AFSM in order to disable them during the scanning. The other four load the state gate with the value of the TI signal, when TE is enabled. Under normal operation, TE is deasserted and this circuit part is disabled. The output of the state gate is fed into a synchronous 2-phase D-type flip-flop, DFF, a possible circuit design of which is shown in Figure 6.3.

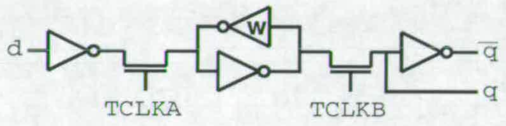


Figure 6.3: 2-phase synchronous DFF circuit

The outputs of the flip-flop, q and $\overline{q}$ are fed into another state gate of the AFSM. This structure forms, during the scanning, a synchronous shift register between the AFSM states. The two clock signals, TCLKA and TCLKB, should be non-overlapping. The D-type captures during the TCLKA active phase and passes the value during the TCLKB active phase. By clocking these pins and supplying input data through the TI input pin the states of an AFSM can be loaded.

Once the states of this AFSM have been loaded, the TE signal can be deasserted, and the AFSM will be allowed to operate. To verify that the circuit operates correctly the AFSM states of the circuit part being tested and those of its neighbours must be inspected. This is achieved via the TO test pin, which is connected to one of the AFSM states. The state of the system is frozen by asserting the TE signal and then, by clocking the synchronous latches, the AFSM states are unloaded onto the TO pin at every clock cycle.

6.2 Future Work

The work carried out has demonstrated the implementation feasibility of both Shared Register Files and μnet architectures. The next step of this research is architectural exploration of both techniques.

6.2.1 Shared Register File Architectures

The aim of an architectural level exploration of Shared Register File architectures would be to investigate the performance impact that the register sharing mechanism has on the execution times of real programs running on various SRF architectures. The architectural exploration space for these SRF architectures would include different SRF organisations, *i.e.* unidirectional, 2-way, 4-way, etc., and the architectural parameters of these organisations, *i.e.* the number of shared sections per RF and the numbers of local and shared registers of each RF.

To perform architectural level simulations, access times for these various SRF organisations are required. These can either be obtained by performing HSPICE simulations of a particular organisation, which requires a transistor-level implementation, or by mathematical modelling, *i.e.* extrapolation of access time graphs. Another possibility is to derive an analytical SRF model, *i.e.* attempt to estimate the access time of an SRF as a function of the individual circuit components. These three different approaches vary in flexibility and accuracy. HSPICE simulations are the most accurate. The accuracy of the other two models is debatable.

The next requirement is the specification of an SRF architecture, and of course the choice of a architectural simulation tool. Various architectural characteristics must be specified. These include the number of SRF nodes, the structure of each node, the number and distribution of FUs, the memory hierarchy and the number of flows of control. The memory hierarchy organisation is of paramount importance because it has a direct effect on the register sharing mechanism. For example, if a unified cache is used between all the nodes, then this provides a second level of variable sharing. The same holds for a unified memory. For shared register files to be proven effective it has to be shown that communicating register values through the shared registers is indeed better than using a unified cache.

The problem of distributing instructions to the architectural nodes can be solved more effectively by the compiler than by a hardware mechanism. This is because the compiler will always have a larger window of instructions in view than any hardware mechanism and it also has a global view of the program. Hence, a parallelising compiler must be implemented. The task of the parallelising compiler would be to convert a sequential program (or multiple sequential programs in the case of multiple control flows) to multiple threads, one per architectural node, which use the shared register mechanism for communication.

In order to realistically model the execution of real programs on these architectures, appropriate benchmark programs must be selected. The choice of benchmarks depends on their characteristics. Desired benchmark characteristics include parallelisable programs, in order to exploit the shared registers, and sequential programs, to measure performance in the case where parallelisation is not possible. In addition, the benchmarks must solve a real problem, so that they model effectively the behaviour of a real program.

6.2.2 μnet Architectures

A scalar μnet architecture has already been simulated at the architectural level by Rebello [Reb96]. Although that work demonstrated the potential of the scalar μnet, it did not study the performance scalability of the architecture or the effect that architectural parameters have on the architecture. Hence, as with the SRF approach, additional architectural level exploration is necessary for both scalar and superscalar architectures. They have to be contrasted to conventional scalar and superscalar architectures in order to verify that the benefits of fine-grain parallelism are greater that the cost of a more complex datapath structure. In addition, superscalar μnets will have to account for the cost of arbitration between multiple μinstructions requesting the same μblocks.

The μnet approach can benefit from compiler support. Even in a scalar μnet the ordering of instructions has an effect on performance due to the dependencies between instructions and the use that they make of the datapath μblocks. One complicating factor is that the latency of the FU operations is variable and data dependent. There is ongoing work in this area [SS00].

6.3 Conclusions

Overall, this thesis presented the CMOS direct-mapped AFSM approach as a solution to the problems involved with asynchronous control circuit design, and used this approach to demonstrate the implementation feasibility of Shared Register Files and μnet-based architectures. Further architectural exploration of both Shared Register Files and μnet-based architectures is necessary to demonstrate their performance on real programs.

Bibliography

[Bla92] G. M. Blair. *MOS Circuit Design: an explanation.* Chartwell-Bratt, 1992.

[Blo59] E. Bloch. The Engineering Design of the Stretch Computer. In *Fall Joint Computer Conference*, pages 48–59, 1959.

[BS89] E. Brunvand and R. F. Sproull. Translating Concurrent Programs into Delay-Insensitive Circuits. In *Proceedings of ICCAD, pp. 262-265*, 1989.

[CAD] CADENCE. Deep Submicron Design Problems. Cadence Web Site. `http://www.cadence.com/software/DeepSubmicron/dsm_paper.html`.

[CDN92] A. Capitanio, N. Dutt, and A. Nicolau. Design Considerations for Limited Connectivity VLIW Architectures. Technical Report TR59-92, University of California, Irvine, 1992.

[Chu87] T. A. Chu. Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications. Technical Report MIT/LCS/TR-393, Massachusetts Institute of Technology, June 1987.

[Com99] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[CS96] C. Y. Chang and S. M. Sze, editors. *ULSI Technology.* McGraw Hill, 1996.

[DCS93] A. Davis, B. Coates, and K. Stevens. Automatic Synthesis of Fast Compact Asynchronous Circuits. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions.* Elsevier Science Publishers, 1993.

[DL99] W. J. Dally and S. Lacy. VLSI Architecture: Past, Present and Future. In *Advanced Research in VLSI, Atlanta*, 1999.

[DW95] P. Day and J. V. Woods. Investigation into Micropipeline Latch Design Styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[End96] P. B. Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor.* PhD thesis, Department of Computer Science, University of Manchester, 1996.

[EUR] EUROPRACTICE. ES2 0.7μm CMOS technology documentation.

[FCJV97] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vrazenic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proceedings of the 30th International Symposium on MicroArchitecture (MICRO-30)*, 1997.

[FD96] S. B. Furber and P. Day. Four-Phase Micropipeline Latch Control Circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[FDG$^+$94] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A Micropipelined ARM. In *Proceedings of the IEEE Computer Conference (COMPCON)*, pages 476–485, March 1994.

[Fer98] M. M. Fernandes. *A Clustered VLIW Architecture Based on Queue Register Files.* PhD thesis, Department of Computer Science, University of Edinburgh, 1998.

[FGR$^+$97] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver. AMULET2e: An Asynchronous Embedded Controller. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, April 1997.

[Fis83] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *10th Symposium on Computer Architecture*, pages 140–150, New York, 1983. ACM Press.

[FJC95] K. R. Farkas, N. P. Jouppi, and P. Chow. Register File Considerations in Dynamically Scheduled Processors. Technical Report

95/10, Digital WRL, 1995. `http://www.research.digital.com:80/wrl/techreports/abstracts/95.10.html`.

[Gar93] J. D. Garside. A CMOS VLSI Implementation of an Asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181–207. Elsevier Science Publishers, 1993.

[GFC99] J. D. Garside, S. B. Furber, and S-H Chung. AMULET3 Revealed. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–59, April 1999.

[Hau93] S. Hauck. Asynchronous Design Methodologies: An Overview. Technical Report TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.

[Hol82] L. A. Hollaar. Direct Implementation of Asynchronous Control Units. *IEEE Transactions on Computers, Vol. C-31, No. 12*, December 1982.

[Hor82] R. M. Hord. *The Illiac-IV, The First Supercomputer.* Computer Science Press, Rockville, Md., 1982.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[HT72] R. G. Hintz and D. P. Tate. Control Data STAR-100 Processor Design. In *COMPCON Digest*, volume 1-4, 1972.

[Hwa93] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[IBM97] IBM. Back to the Future: Copper Comes of Age ? IBM Research Magazine No. 4, 1997. `http://www.research.ibm.com/resources/magazine/1997/issue_4/copper497.html`.

[IC78] R. N. Ibbett and P. C. Capon. The Development of the MU-5 Computer System. *Communications of the ACM*, 21:13–24, January 1978.

[JC95] J. Janssen and H. Corporaal. Partitioned Register File for TTAs. In *Proceedings on the 28th Annual Sympsium on MicroArchitecture (MICRO-28)*, 1995.

[Joh91] M. Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[KBJND96] B. Konemann, B. Bennetts, N. Jarwala, and B. Nadeau-Dostie. Built-In Self-Test: Assuring System Integrity. *IEEE Computer*, November 1996.

[KELS62] T. Kilburn, D. G. B. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers EC-11*, pages 223–235, April 1962.

[KS82] D. J. Kuck and R. A. Stokes. The Burroughs Scientific Processor (BSP). *IEEE Transactions on Computers*, pages 363–376, May 1982.

[Kum96] R. Kumar. Scalable Register File Organisations for a Multiple Issue Microprocessor. *IEE Electronic Letters*, 32(7), 28th March 1996.

[KW76] D. J. Kinniment and J. V. Woods. Synchronisation and Arbitration Circuits in Digital Systems. *IEE Proceedings*, 123:961–966, 1976.

[Man91] M. Morris Mano. *Digital Design.* Prentice-Hall, 1991.

[Mar90a] A. J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, California Institute of Technology, 1990. Addison Wesley.

[Mar90b] A. J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In *Advanced Research in VLSI.* MIT Press, 1990.

[Mas91] MasPar. The MasPar Family Data-Parallel Computer. Technical report, MasPar Computer Corporation, Sunnyvale, CA, 1991.

[Mat97] D. Matzke. Will Physical Scalability Sabotage Performance Gains ? *IEEE Computer*, September 1997.

[MBL$^+$89] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The Design of an Asynchronous Microprocessor. In *Proceeding of Advanced Research in VLSI*, pages 351–373, 1989.

[Met90] Meta-Software. *HSPICE User's Manual H9001*, 1990.

[Mil65] R. E. Miller. *Switching Theory, Volume II: Sequential Circuits and Machines.* John Wiley and Sons, 1965.

[MLM$^+$97] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee. The Design of an Asynchronous MIPS R3000 Microprocessor. In *Proceedings of Advanced Research in VLSI*, pages 164–181, September 1997.

[MOT97a] MOTOROLA. New Dual Inlaid Copper Interconnect. Motorola Press Release, September 30 1997. `http://mot2.indirect.com/press/html/PR970930A.html`.

[Mot97b] Motorola, IBM. *PowerPC 750 RISC Microprocessor Technical Summary*, 1997.

[MSAD92] W. Mangione-Smith, S. G. Abraham, and E.S. Davidson. Register Requirements of Pipelined Processors. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 260–271, 1992.

[Mur89] T. Murata. Petri-Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77-4, pages 541–580, 1989.

[NUK$^+$94] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Tokamura. TITAC: Design of a quasi-delay-insensitive Microprocessor. *IEEE Design and Test of Computers*, 11(2):60–63, 1994.

[ONH$^+$96] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for the Single-Chip Multiprocessor. In *ASPLOS VII*, October 1996.

[PDF$^+$98] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A Low-Power, Low Noise, Configurable Self-Timed DSP. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, 1998.

[PJS97] S. Palacharla, N. P. Jouppi, and J.E. Smith. Complexity-Efficient Superscalar Processors. In *Proceedings of the 24th Annual Symposium on Computer Architecture*, 1997.

[PS82] D. Patterson and C. Sequin. A VLSI RISC. *IEEE Computer, 15(9)*, 1982.

[Reb96] V. E. F. Rebello. *On the Distribution of Control in Asynchronous Processor Architectures.* PhD thesis, Department of Computer Science, The University of Edinburgh, 1996.

[Red73] S. F. Reddaway. DAP - A Distributed Array Processor. In *Proceedings of the 1st Annual Symposium on Computer Architecture*, December 1973.

[RFS94] C. E. Molnar R. F. Sproull, I. E. Sutherland. The Counterflow Pipeline Processor Architecture. *IEEE Design and Test of Computers*, 11(3):48–59, 1994.

[Rus78] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21:63–72, 1978.

[SBV95] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[Sha98] T. Shanley. *Pentium Pro and Pentium II System Architecture.* Addison-Wesley, 1998.

[SS95] J. E. Smith and G. S. Sohi. The Microarchitecture of Superscalar Processors. Technical report, University of Wisconsin, 1995.

[SS00] S. Sotelo-Salazar. *Global Optimisation and Scheduling for Asynchronous Processor Architectures.* PhD thesis, Institute for Computing Systems Architecture, Division of Informatics, University of Edinburgh, 2000.

[Ste94] K. S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems.* PhD thesis, Department of Computer Science, The University of Calgary, September 1994.

[Sut89] I. E. Sutherland. Micropipelines. *Communications of the ACM, Volume 32, Number 6*, 1989.

[Thi90] Thinking Machines Corporation, Cambridge, MA. *The CM-2 Technical Summary*, 1990.

[Tho64] J. E. Thornton. Parallel Operation in Control Data 6600. In *AFIPS Fall Joint Computer Conference 26*, volume 2, pages 33–40, 1964.

[TKI+97] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya. TITAC-2: An Asynchronous 32-bit Microprocessor based on Scalable-Delay-Insensitive Model. In *Proceedings of International Conference on Computer Design*, pages 288–294, October 1997.

[TMI99] H. Terada, S. Miyata, and M. Iwata. DDMPs: Self-Timed Super-Pipelined Data-Driven Multimedia Processor. In *Proceedings of the IEEE*, volume 87 of *2*, February 1999.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal for Research and Development 11:1*, pages 25–33, January 1967.

[Ung69] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley Interscience, Department of Electrical Engineering, Columbia University, 1969.

[vB93] K. van Berkel. *Handshake Circuits: An asynchronous architecture for VLSI programming.* Cambridge University Press, 1993.

[vGvBP+98] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann. An Asynchronous low-power 80C51 Microcontroller. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1998.

[Wat72] W. J. Watson. The TI ASC - A Highly Modular and Flexible Super Computer Architecture. In *AFIPSFJCC*, volume 41, pages 221–228, 1972.

[WE93] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design 2nd Edition.* Addison Wesley, 1993.

[YDN92] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D Asynchronous State Machines. In *Proc. International Conf. Computer Design (ICCD)*, Cambridge, Massachusets, 1992.