A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors

By

Edward Ashford Lee

B.S. (Yale University) 1979
S.M. (Massachusetts Institute of Technology) 1981

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Engineering

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: David G. Messerschmitt 5/23/86
Chairman .... Date
D. G. Morrison 5/29/86
Robert W. Brodersen 6/10/86

....

ACKNOWLEDGEMENTS

Financial support for my work came from a variety of sources. Special appreciation goes to General Electric, whose generous fellowship supported me for the first year, and to IBM, whose fellowship supported me for all subsequent years. During the summers, Grant ECS-8211071 from the National Science Foundation and a grant from the Shell Development Corporation provided support.

On the personal side, I am particularly indebted to my advisor, Professor David Messerschmitt, who treated me like a colleague well before it was deserved, and to my committee members Prof. Robert Brodersen and Prof. Frank Morrison. Frequent early consultation with Amine Haoui, Don Chin, and Graham Brand, senior graduate students when I started, helped keep me on the right path. Prof. Eugene Lawler and Prof. Alvin Despain both directed me to valuable prior art. Prof. David Schwartz, of Georgia Tech, and Silesh Rao, formerly of Stanford, now at Bell Labs, wrote dissertations that were an inspiration, and gave me their valuable time to explain their ideas. Conversations with Teresa Meng, Ilovich Mordechay, Keshab Parhi, and Vijay Madisetti were enlightening. Rodion Rathbone carefully read and commented on an early draft. Finally, my best critic was Rhonda Righter, who suffered through many dinners during which I made my first attempts to express vague new ideas.

Table of Contents

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: SYNCHRONOUS DATA FLOW 7

2.1. THE DATA FLOW PARADIGM 11

2.2. PRIOR ART AND RELATED MODELS 15

2.3. SYNCHRONOUS DATA FLOW GRAPHS 25

2.4 A LARGE GRAIN COMPILER FOR A SEQUENTIAL MACHINE 30

2.5 THE PARALLEL SCHEDULING PROBLEM 51

2.6 LIMITATIONS OF THE MODEL 60

2.7 CONCLUSIONS 63

CHAPTER 3: IMPROVING PERFORMANCE OF A PARALLEL SCHEDULE 65

3.1 CUTSET TRANSFORMATIONS 68

3.2 THE BOUND ON THE COMPUTATION RATE 78

3.3 TRANSFORMING GENERAL SDF GRAPHS INTO HOMOGENEOUS SDF GRAPHS 84

3.4 THE OPTIMAL BLOCKING FACTOR 93

3.5 MINIMIZING MEMORY USAGE 101

CHAPTER 4: THE π ARCHITECTURE 107

4.1. MONOLITHIC PROGRAMMABLE DSPs 108

4.2. HAZARDS 117

4.3 THE PROGRAMMING/PERFORMANCE TRADEOFF 119

4.4 PIPELINING AND INTERLEAVING 130

4.5 A SAMPLE ARCHITECTURE 147

4.6 PROGRAMMING EXAMPLES 164

4.7 THE COLUMBIA ARCHITECTURE 173

4.8 CONCLUSIONS 177

CHAPTER 5: PROGRAMMING A π PROCESSOR USING SDF GRAPHS 178

5.1 BUFFERS AND DELAYS 180

5.2 LANGUAGE DESIGN 188

5.3 A VOICEBAND DATA MODEM EXAMPLE 193

5.4 CONCLUSIONS 203

CHAPTER 6: FURTHER WORK 205

6.1 FUNDAMENTALS 206

6.3 HARDWARE 210

6.2 SOFTWARE 212

6.2 APPLICATIONS 214

6.2 CONCLUSION 214

GLOSSARY 216

REFERENCES 220

1

INTRODUCTION

The main objective of this thesis is to propose techniques that will help harness the potential of VLSI for high performance, real-time digital signal processing. Ideally, state of the art implementation techniques will be within the reach of the algorithm developers, communications specialists, numerical analysts, and signal processing specialists. Currently, a major technological chasm separates these individuals from the hardware and software tools required to implement and test their ideas in real-time. In spite of slow steady progress with silicon compilation, chip design still requires a circuit designer to do the layout. The language of a circuit designer is usually foreign to the communications specialist with an interest in estimation, adaptive filters, or distributed control. Software implementation techniques are not much better. Although significant progress has been made with software interfaces on general purpose computers, such machines often cannot approach real-time, and interfacing such machines in real environments is often difficult even for the architec-

ture specialist. High performance architectures, such as programmable monolithic DSP chips, usually require specialized programming in assembly language, complicated by extensive pipelining and parallelism.

With VLSI technology improving, the problem may be getting worse, not better. Increased parallelism in architectures means that algorithm designers need to worry more about "vectorizing" their algorithms, or finding pipelinable or systolic versions. Adaptation of algorithms to parallel implementations is traditionally done on a case-by-case basis, with each application laboriously analyzed for available concurrency, and each parallel algorithm studied for quantization effects and convergence (see for example[Thom77a, John84a, Ahme82a, Kung83a, Kung80a]). Techniques for automatically finding parallelism in programs have met with limited success[Padu80a, Fish84a], so although parallel and pipelined architectures offer greatly improved performance, they usually require more implementation effort. (There are experimental architectures specialized to particular languages which also try to avoid this phenomenon.) This thesis represents one small effort to close the algorithm-implementation chasm without compromising the performance of the implementations.

The long term objective is to rapidly and easily prototype real-time signal processing systems. The aim is to bring state-of-the-art implementation tools closer to the algorithm designer, without requiring valuable time and energy to learn a language that is not relevant in any fundamental way to his or her expertise. This is an ambitious goal, and maybe the best we can hope to do in the near term is to keep the chasm from widening as more parallelism intrudes

into implementation architectures.

This thesis has two main themes. The first is a proposal for a new programming methodology, called *synchronous data flow* (SDF). The second is an architecture for real-time programmable digital signal processors that takes advantage of SDF programming to significantly improve performance. We attempt to enhance performance while *narrowing* the algorithm-implementation chasm. Attributes of the hardware and software methodologies are designed specifically for a complementary fit.

The basic idea behind the architecture is old[Shar74a], and can be used to overcome difficulties associated with deeply pipelined programmable processors. The technique, previously applied to general purpose supercomputers[Smit78a, Jord84a, Cohn83a], is to interleave multiple programs through the pipeline in such a way that each program is unaffected by the pipelining. An instruction from a given program is fetched only after execution of the previous instruction from the same program has been completed. Meanwhile, instructions from the other programs are fetched, so that the hardware resources of the pipelined machine can be fully used. In each clock cycle, an instruction from a new program is fetched, suggesting that the technique can be described as a context switch on every clock cycle. The main advantage is that extensive pipelining can be introduced into the architecture without affecting the instruction set. This advantage is relatively technology independent, because it can probably be used to extensively pipeline any architecture implemented in any technology. For this reason, although we refer to some relatively old programmable digital signal processors (dating back to

1979), the architecture proposed in this thesis should be relevant for some time to come, barring a major upheaval in the technology. The cost in hardware of this technique is not trivial, because the entire processor state must be stored (and be readily accessible) for each interleaved process. But the cost is much less than the cost of brute-force parallelism, in which a non-pipelined programmable processor would be replicated several times[Shar74a]. The interleaving technique has not gained wide acceptance in the design of general purpose computers, but we believe that because of the narrow application domain and the existence of an efficient and practical programming technique, pipelined interleaving is ideally suited to the design of programmable digital signal processors.

From the point of view of the user, the most visible feature is parallelism, which replaces pipelining. The programmer has to construct parallel programs that cooperate on a signal processing task, itself not a trivial task. Unless an appropriate method for constructing such programs is devised, the value of such an architecture is questionable. We propose programming signal processing applications using *synchronous data flow* graphs. SDF graphs are related to data flow graphs[Denn80a], with the difference that nodes in the graph consume and produce a *fixed* amount of data on each input or output path. In signal processing terminology, systems with fixed sample rates related by rational factors can be specified as SDF graphs. Taken alone, such a programming methodology helps to narrow the algorithm-implementation chasm because data flow (especially *large grain* data flow[Acke82a]) is similar to block diagrams and signal flow graphs that signal processing specialists like to use to describe algorithms[Gold69a, Croc75a, Babb84a]. Specifying signal processing algorithms

as block diagrams is certainly more natural than specifying them as Fortran programs, although comfortable familiarity with Fortran may lead some readers to deny this.

Although programmer convenience is of overriding importance, the technique would not be useful if efficient implementations were not possible. Fortunately, they are. SDF graphs naturally exhibit considerable concurrency which can be automatically and efficiently mapped onto parallel processors. Unlike more general data flow, SDF graphs can be *statically* scheduled (at compile time), so that at run time, most processing resources do signal processing rather than support for the programming methodology. Demonstrating that this is possible (and practical) is a major objective of this thesis. SDF can also be used on conventional architectures with single or parallel processors, but the π processor is a particularly good match.

This first chapter is only a brief overview of the material to come. Most of the substantial body of literature contributing to the background of this work is described and referenced in the body of the thesis. Chapter 2 describes the SDF paradigm, proves that static scheduling is possible for all correctly constructed SDF graphs, and details how such scheduling can be done. Chapter 3 describes the lower bound on the throughput of a given SDF graph and systematic methods for approaching this bound. Chapter 4 describes the pipelining and interleaving technique and outlines a specific π architecture. Chapter 5 shows how SDF can be practically implemented on a π processor and details an example of a practical application, a voiceband data modem implementation. The modem application illustrates the attractiveness of the programming methodol-

ogy both for programming ease and efficient implementation on modest numbers of parallel processors. Chapter 6 wraps it up with some conclusions and an extensive catalog of spinoff problems derived from this work.

Chapters 2 and 4 are intended to each stand alone, in case particular readers are interested strictly in the π or SDF topics. Chapter 3 depends on chapter 2. Chapter 5 depends on a rudimentary understanding of chapters 2 through 4.

2

SYNCHRONOUS DATA FLOW

To achieve high performance in real-time signal processing and related numeric-intensive computations, the need to depart from the simplicity of von Neumann computer architectures is axiomatic. Pipelining and parallelism are two forms of concurrency that can take advantage of increasing VLSI complexity, but both approaches can considerably complicate the programming. It would be unacceptable to design a digital signal processor (DSP) architecture with spectacular throughput that required an army of specialists to program. In chapter 4 we will describe an architecture for programmable DSPs that replaces deeply pipelined architectures with what appears to be cooperating parallel processors. The modified architecture is called a π processor. The technique is to interleave concurrent processes through the pipeline in such a way that each instruction in each process is completed before the next instruction in the same process is begun. This technique has been used in the peripheral processors of the CDC 6600, the HEP-1[Smit78a, Jord84a], and an experimental architecture

at Columbia[Cohn83a], all large general purpose machines. Applied to signal processors, the technique is much simpler, and promises to mitigate the difficulties associated with deep pipelining, allowing a potentially large gain in performance through deeper pipelining than is generally considered practical for programmable processors. But since pipeline difficulties are replaced with difficulties associated with parallelism, a reasonably effective solution to the parallel programming problem for modest numbers of processors is a mandatory prerequisite.

For the purposes of this chapter, the π processor described in chapter 4 can be viewed simply as a set of synchronized parallel processors sharing memory without contention. The programming of a π processor is, on the surface, no different from programming any such multiprocessor system. However, the fact that the application is real-time digital signal processing greatly simplifies the programming problem, opening up the possibility of conceptually simple and elegant programming techniques.

The π processor promises considerably enhanced performance at the expense of a non-Von Neumann structure. In the software realm, however, deviations from von Neumann programming are often viewed with suspicion. The hardware of most successful commercial signal processors today deviates significantly from Von Nuemann architecture, but compromises are made to preserve sequential programming. Two notable exceptions are the Bell Labs DSP family[Chap81a, Kers85a], the NEC data flow chip[Chas84a], and the Motorola DSP56000[Moto86a], all of which are programmed with concurrency in mind. For the majority, however, preserving von Neumann programming style is given

priority. Interestingly, Motorala touts the *sequential* nature of the programming of the 56000, a Von Neumann attribute, but the programming includes significant parallelism.

Sequential programming has a long and distinguished history. Often, a new non-von Neumann architecture has elaborate hardware and software techniques enabling a programmer to write sequential code irrespective of the parallel nature of the underlying hardware. For example, in machines with multiple function units, such as the CDC6600 and Cray family, so called "scoreboarding" hardware resolves conflicts to ensure the integrity of sequential code. In deeply pipelined machines such as the IBM 360 model 91, interlocking mechanisms[Kogg81a] resolve pipeline conflicts. In the MIT Lincoln Labs signal processor[Paul80a] specialized associative memories are used to ensure the integrity of data precedences.

The affinity for von Neumann programming is not at all surprising, stemming from familiarity and a proven track record, but the cost is high in the design of specialized digital signal processors. In chapter 4, the TI TMS32010 and the Bell Labs DSP20, two pipelined chips that differ radically only in programming methodology, will be compared. They achieve exactly the same performance on the most basic benchmark, the FIR (finite impulse response) filter, but the Bell Labs chip outperforms the TI chip on the next most basic benchmark, the IIR (infinite impulse response) filter. Surprisingly, close examination reveals that the arithmetic hardware (multiplier and ALU) of the Bell Labs chip is half as fast as in the TI chip. The performance gain appears to follow from the departure from conventional sequential programming.

However, programming the Bell Labs chip is not easy. The code more closely resembles horizontal microcode than assembly languages. Programmers invariably adhere to the quaint custom of programming these processors in assembler-level languages, for maximum use of hardware resources. Satisfactory compilers have failed to appear.

In this chapter, we propose programming signal processors using a technique based on large grain data flow (LGDF) languages[Acke82a], which should ease the programming task by enhancing the modularity of code and permitting algorithms to be described more naturally. In addition, concurrency is immediately evident in the program description, so parallel hardware resources such as those in the π architecture can be used more effectively. The technique is particularly well suited to programming the π processor, so it is described in that context, but it is also potentially useful for programming existing single and multiple processor systems, and may be extensible to semi-custom hardware definition for a parametrized macrocell silicon compiler. This latter application is proposed in chapter 6.

We begin by reviewing the data flow paradigm and its relationship with previous methods applied to signal processing. *Synchronous data flow* (SDF) is introduced, with its suitability for describing signal processing systems explained. The advantage of SDF over conventional data flow is that more efficient run-time code can be generated because the data flow nodes can be scheduled at compile time, rather than at run-time. A class of algorithms for constructing sequential (single processor) schedules is proven valid, and a simple heuristic for constructing parallel (multi-processor) schedules is described.

Finally, the limitations of the model are considered.

2.1. THE DATA FLOW PARADIGM

For concurrency, a program is broken into subtasks which are then automatically, semiautomatically, or manually scheduled onto parallel processors or π slices, either at compile time (*statically*) or at run time (*dynamically*). Automatic breakdown of an ordinary sequential program is an appealing concept[Padu80a], but the success of existing techniques is limited. If the programmer provides the breakdown as a natural consequence of the programming methodology, we should expect more efficient use of concurrent resources.

Dividing a program into subtasks is not new to programmers; structured programming has insisted on it throughout much of the history of computers. The usual technique for breaking up a program is to divide it into subroutines, functions, or procedures. But these are all ill suited to parallel execution, because they are written with sequential execution in mind. Furthermore, procedures are not usually a natural way of describing DSP algorithms; functional blocks interconnected with signal flow paths are more suitable.

DSP systems are usually described using *block diagrams* consisting of functional blocks connected by data flow paths. An example that we will discuss extensively in chapter 5, a voiceband data modem, is illustrated in figure 2-1. The figure illustrates an implementation of a 2400 BPS, 600 baud frequency-division-multiplexed full-duplex data modem with bandsplitting filters, and a fractionally-spaced passband adaptive equalizer[Unge76a, Gitl81a, Falc76a]. Such diagrams are often viewed as descriptions of hardware realizations, which are usually inherently highly concurrent. Block diagram descriptions are

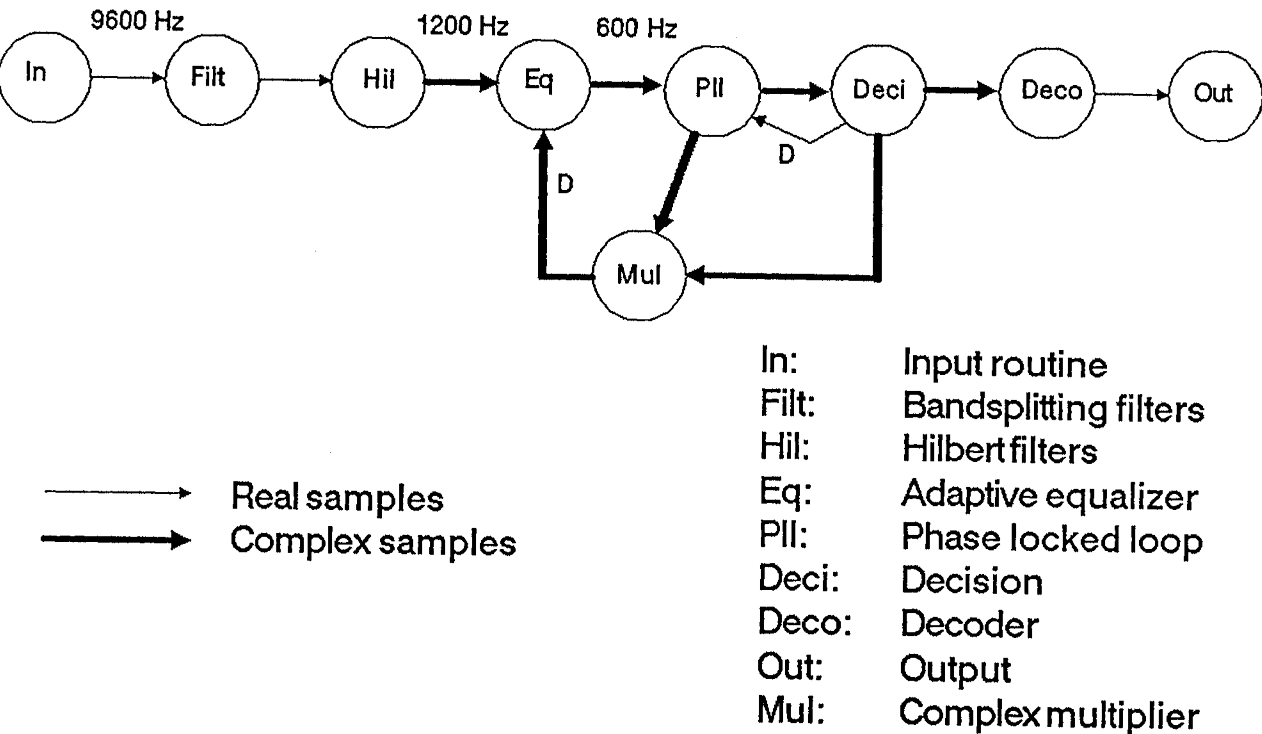


Figure 2-1: A block diagram of a 2400 bit per second, 600 baud modem. Three sample rates are evident. The first, 9600 Hz, is the Nyquist rate samples of the data bearing signal, the second is twice the baud rate (at the input to a fractionally spaced equalizer), and the third is the baud rate, 600Hz.

modular, meaning that once a block is defined it is easily re-used. They can also be hierarchical, where a block may itself represent another block-diagram, yielding programs with much of the elegance of structured programming. *Mixed mode* programming is possible, where frequently used blocks are programmed in assembly language and less common blocks in a high level language. Hardware descriptions can also be mixed with assembly language or higher level functional descriptions, in principle. Concurrency is explicit, without requiring undue programming effort. Furthermore, we will show that additional concurrency can often be easily found, automatically, by a compiler.

The block diagram of figure 2-1 is a *data flow graph*. The fundamental premise behind data flow graphs is that each node represents a function that can be invoked whenever input data is available to it. Functions may be elemental (addition, multiplication, etc.) or non-elemental (digital filters, FFT units, modulators, phase locked loops, etc.), and the directed arcs represent paths taken by successive data samples. The complexity of the functions (or the "granularity") will determine the amount of parallelism available. If the granularity is at the level of signal processing subsystems (second order sections, butterfly units, etc.), the paradigm is called *large grain data flow* (LGDF)[Davi78a, Rumb77a, Babb84a, Acke82a]. Otherwise, it is called *atomic* data flow, from the Greek word *atomos*, meaning indivisible. Atomic data flow is obviously a special case of LGDF. Data flow specifications are related to the experimental computer language techniques known as *applicative languages*, *single assignment languages*, and *functional languages*.

The original *data flow*[Denn80a, Wats82a] is a computer architecture technique, not a programming technique for multiprocessors. Specialized hardware blocks perform their function (*fire*) when operands appear at their inputs. At the lowest level, therefore, the machine is *data driven*[Trel81a]. Control of the hardware is achieved by supplying operands to hardware blocks. This carries over to the software realm. Software functions fire when inputs are available.

LGDF is ideally suited for signal processing, and has been adopted in simulators in the past[Mess84a]. Other signal processing systems use a data-driven paradigm to partition a task among cooperating processors[Snyd84a], and many so called "block diagram languages" have been developed to permit programmers to describe signal processing systems more naturally. These and other related models are discussed in the following subsections.

In addition to being natural for DSP, large grain data flow has another significant advantage for signal processing. As long as the integrity of the flow of data is preserved, any implementation of a data flow description will produce the same results. This means that the same software description of a signal processing system can be simulated on a single processor or multiple processors, implemented in specialized hardware, or even, ultimately, compiled into a VLSI chip[Jhon85a].

Common objections leveled against data flow center around the overhead required to schedule and synchronize the hardware or software blocks. However, it will be shown in this chapter very low overhead is required for most types of synchronization required in digital signal processing. In particular, for *synchronous* DSP systems, in which sample rates are known and are rational

multiples of one another, it is possible to statically (at compile time) schedule blocks onto parallel processors. In figure 2-1, sample rates are shown explicitly. A data flow graph from which *relative* sample rates can be determined (the absolute numbers are not important, only their relation to one another) is called a *synchronous data flow graph* (SDF graph). The SDF paradigm is rigorously defined in section 2.3, and its properties explored.

2.2. PRIOR ART AND RELATED MODELS

Before proceeding, it is worth reviewing the rich collection of related methods for defining DSP systems.

2.2.1. Block Diagram Languages

Many so called *block diagram languages* have been developed to permit users to more easily implement signal processing algorithms, at least in simulation, rather than real-time. Block diagrams are an appropriate description of DSP systems. Some examples are BLODI[Kell61a], PATSI[Gold69a], BLODIB[Kara65a], LOTUS[Dert69a], DARE[Korn77a], MITSYN[Henk75a], Circus[Crys74a], and TOPSIM[Dipaa]. But these simulators are based on the principle of *next state simulation*[Gold69a, Kope80a] and thus have difficulty with multiple sample rates, not to mention asynchronous systems. (We use the term *asynchronous* here in the DSP sense to refer to systems with sample rates that are not related by a rational multiplicative factor.) Although true asynchrony is relatively rare in digital signal processing, multiple sample rates are common, stemming from the frequent use of decimation and interpolation. The technique we propose in this paper shares the appropriateness of block diagram

languages, and it handles multiple sample rates easily. Furthermore, a π processor architecture combined with a data flow programming paradigm easily handles systems with limited asynchrony.

2.2.2. Large Grain Data Flow

One way to avoid the limitations of next-state simulators but retain the convenience of expressing algorithms as block diagrams is to use LGDF, as done in BLOSIM[Mess84b, Mess84a], a single-processor digital signal processing simulator that naturally accommodates multiple sample rates and asynchronous systems. Dynamically allocated linked-lists are used to buffer data between data flow blocks. The scheduling is straightforward; a simple heuristic is used to construct a list of all blocks approximately ordered according to their precedences. Blocks are invoked in the order specified by this list. When a block is invoked, the software within the block checks the input buffers to see if there is adequate data, and runs until such data is exhausted. If a block has no inputs, it runs until it has generated a predetermined amount of new data on its output buffers. The system continues forever, or until a deadlock occurs, in which no block in the list has data on its input buffers. This type of control is effectively *dynamic* because the execution of block functions is determined at run time, in response to changing conditions. The ordering of the list of blocks approximately according to their precedences helps ensure that most blocks are ready to run when their input buffers are checked, but it is not required for correct execution. Another way of ensuring this is to maintain a list of runnable blocks which is expanded each time a block generates enough data to run another block.

Such dynamic control mechanisms are traditional in data flow implementations, but they involve considerable overhead. First of all, in the latest version of BLOSIM, buffers are unbounded in size so that a block does not need to check its output buffers to see if there is room. The amount of memory required for the buffers is difficult to determine, and even for fully synchronous systems may depend on the order in which the blocks are invoked. In programmable DSPs, where memory is a scarce commodity, such inefficiencies must be avoided. Secondly, the supervisory overhead can be substantial. Checking input buffers, and possibly trying many blocks before one is finally run, consumes program execution time, also a scarce commodity in real-time signal processing applications. On the other hand, this mechanism supports asynchronous communication between blocks, and no elaborate scheduling is required for correct execution of the program. This gives it a generality that is extremely useful for simulation, but probably too costly for real-time implementations on DSPs.

2.2.3. Signal Flow Graphs

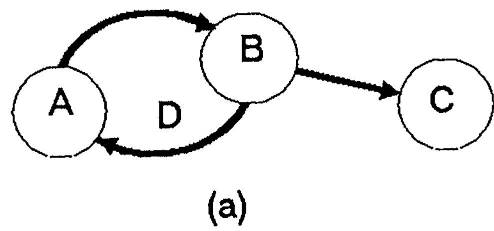
Some work with synchronous data flow descriptions, centered around *signal flow graphs*, was originally used to describe linear, single-sample-rate systems[Croc75a]. Crochiere and Oppenheim[Croc75a] systematically translate signal flow graphs into *acyclic precedence graphs*. The method is quite simple, based on the observation that arcs with unit delays are not precedence relationships, but arcs without delays are. Equivalently, arcs with delays can be broken and replaced with I/O operations. But the method does not consider the repetitive nature of a desired schedule, and therefore does not always properly indicate long-term precedences when more than one delay is present in a loop.

It also does not support multiple sample rates. In spite of these deficiencies, Brafman[Braf78a] and Zeman[Zema83a] both recommend this method to obtain acyclic precedence graphs, and then both apply critical path methods to the scheduling problem.

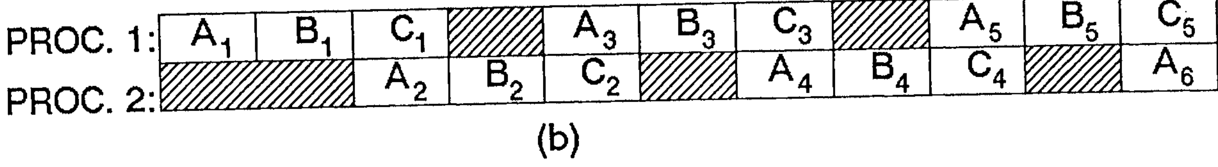
2.2.4. Homogeneous Synchronous Data Flow

The term "signal flow graph" is often used to described single-sample-rate data flow graphs, regardless of whether the the system is linear. *Homogeneous* SDF graphs are a special case of single-sample-rate SDF graphs where each time a node is invoked it consumes or produces exactly one sample on each input or output path. Multiprocessor implementations of algorithms specified this way have been explored at Georgia Tech[Schw85a]. Algorithms are assumed to repeatedly operate on an infinite stream of data, and optimal periodic schedules can be systematically generated. One of the implementations proposed is called *skewed single instruction multiple data* (SSIMD)[Barn82a, Barn83a], in which a set of processing elements perform the same functions, but skewed in time with respect to one another. This allows much more flexibility than traditional SIMD, and is particularly well suited to simple signal processing tasks. SSIMD is described for atomic data flow graphs but it applies as well to LGDF. It is worth reviewing in more detail. Such a review serves as a good introduction to the scheduling problem for SDF graphs.

Figure 2-2(a) shows a simple data flow graph, where A, B, and C designate operations in the program, such as multiplications or additions, the arcs illustrate the flow of data, and the directed loop represents a recursion. The D associated with an arc indicates a unit delay, required for computability in any



(a)



(b)

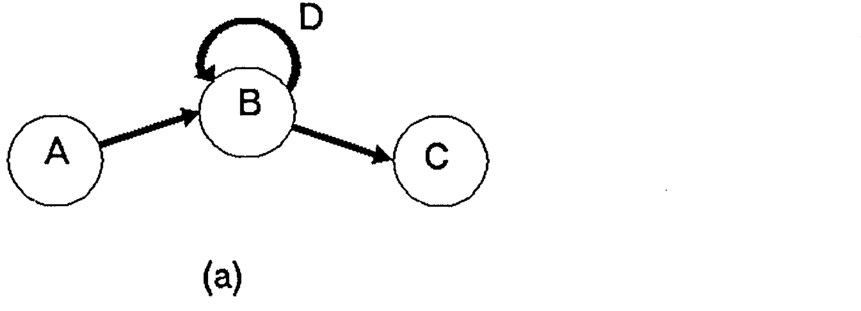
Figure 2-2.
(a) A single-sample-rate data flow graph with a directed loop and a delay, indicating recursion.
(b) A two processor SSIMD schedule, assuming equal execution times.

directed loop. Figure 2-2(b) shows how the functions A, B, and C can be repetitively executed by two cooperating processors, assuming that the execution time of each block is identical. The notation A_1 (or A_2 ...) refers to the first (or second ...) execution of the function A, and time proceeds to the right. Observe that each of the two processors performs exactly the same sequence of operations, but skewed in time. Hence the name SSIMD. The amount of skew is determined by the precedence constraint that the nth run of A must be preceded by the $(n-1)^{th}$ run of B. For SSIMD, the problem of scheduling tasks onto parallel processors simply reduces to determining a single processor schedule and the optimum starting time for each processor.

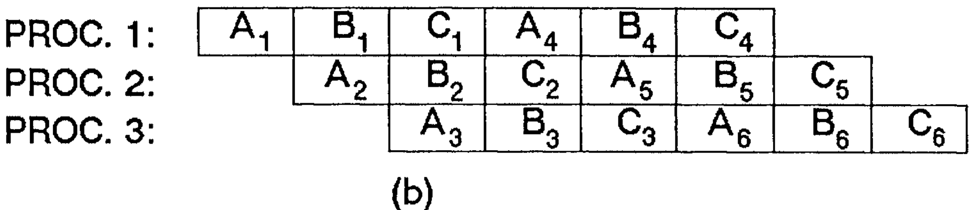
If a state variable is involved, for example a parameter in memory that is updated in a data dependent way, then the data flow graph will have self-loops, as shown in figure 2-3(a). These are handled just like the recursion. A three processor schedule is shown in figure 2-3(b). Without state variables or other directed loops, as shown in figure 2-4(a), there is no distinction between the first and second execution of an operation, and the schedule can have arbitrary start times. If the start times are identical for all processors, as shown in figure 2-4(b) for three processors, then SSIMD reduces to classical SIMD, a much more restrictive paradigm sometimes applied to the construction of concurrent programs.

Signal processing algorithms repetitively operating on an infinite stream of data can be mapped onto parallel processors using the SSIMD paradigm. Surprisingly, in many cases, the mapping is pretty good. However, exceptions are easily constructed. Figure 2-5(a) shows a data flow graph that results in the suboptimal SSIMD schedule of figure 2-5(b). Clearly the *multiple instruction, multiple data* (MIMD) schedule of figure 2-5(c) will run faster. Since the architecture of a π processor readily accommodates MIMD programs, a scheduling methodology yielding the schedule of figure 2-5(c) is preferable. SSIMD schedules remain a special case that can be used when they are optimal, or nearly so.

The technique has been generalized to what are called *cyclo-static systems*[Schw85a], in which the function performed by each processor is periodic, but each processor can perform different functions. Optimal scheduling can be done for such systems. The scheduling algorithm given by Schwartz is of

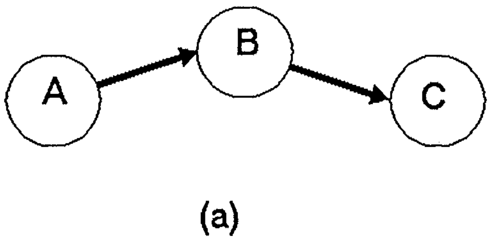


(a)



(b)

Figure 2-3: A single-sample-rate data flow graph with a self loop (a), and an SSIMD schedule (b), assuming the execution times are all the same.

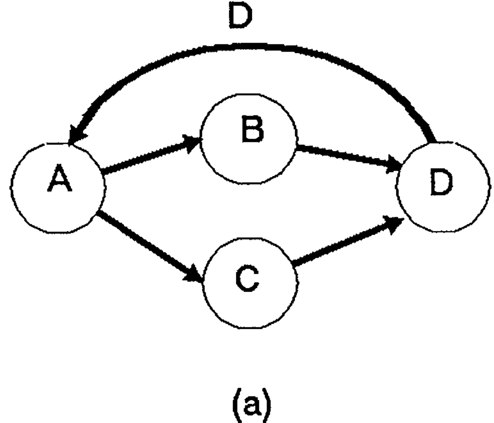


(a)

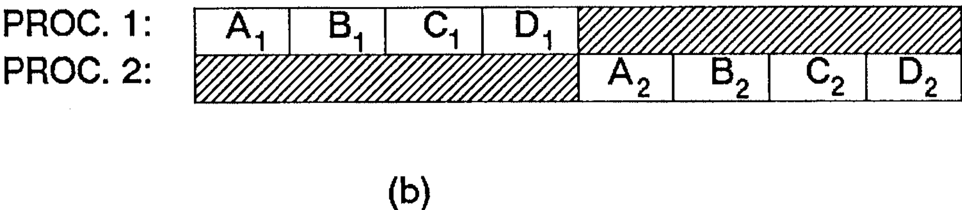
PROC. 1:	A_1	B_1	C_1	A_4	B_4	C_4
PROC. 2:	A_2	B_2	C_2	A_5	B_5	C_5
PROC. 3:	A_3	B_3	C_3	A_6	B_6	C_6

(b)

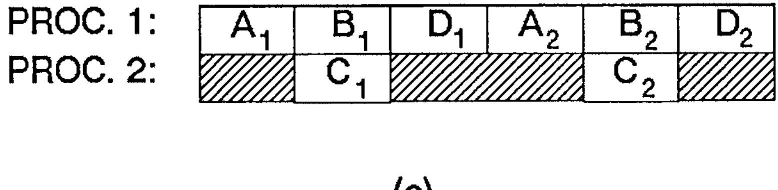
Figure 2-4: A single-sample-rate data flow graph with no recursion (a) and an SSIMD schedule (b). Note that there is no precedence between successive invocations of the same node, because there are no self-loops.



(a)



(b)



(c)

Figure 2-5:

(a) A single-sample-rate SDF graph.

(b) A SSIMD schedule with iteration period 4.

(c) A MIMD schedule, with iteration period 3.

exponential complexity, but Schwartz argues that for special problems the complexity is manageable. Particularly, constructing maximum throughput schedules using the minimum number of processors from a homogeneous SDF graph is apparently easy. The admirable work on cyclo-static systems has some deficiencies in our application, however. Primarily, it has no provision for multiple sample rates, thus restricting the range of applications. Also, when the problem is to find a maximum throughput schedule for a fixed number of processors, the complexity may become unmanageable.

2.2.5. Reduced Dependence Graphs.

Explored recently at Stanford, *reduced dependence graphs* are specifications of systems in terms of periodic acyclic precedence graphs, where only one period is illustrated, and its dependence on previous periods is done by indexing[Karp67a, Rao85a]. The resulting description is close to a data flow graph with the sample rate again restricted to be uniform throughout the system. A major difference is that delays on arcs are generalized to vectors in an index space so iterations in time can be combined with iterations in any other variable. Reduced dependence graphs are used to describe *regular iterative algorithms*, which can then be mapped onto processor arrays. This approach seems particularly well suited to descriptions of well structured algorithms to be implemented in systolic arrays. The range of applications is again excessively limited for our objectives.

2.2.6. Computation Graphs.

Computation graphs are large grain data flow graphs which, like signal flow graphs, and reduced dependence graphs, are restricted to modeling *synchronous* systems. Unlike these previous models, however, computation graphs naturally accommodate multiple sample rates. The differences between SDF graphs and computation graphs are so minor as to be insignificant, but our use of the model differs significantly.

Computation graphs were introduced in 1966 by Karp and Miller[Karp66a] and were further explored by Reiter[Reit67a]. The analysis concentrated on fundamental theoretical considerations, with, for example, a proof that computation graphs are *determinate*, meaning that any admissible execution yields the same result. Such a theorem, of course, also underlies the validity of data flow. Other early analysis using the general computation graph model concentrated on graphs that *terminate*, or deadlock after some time. Most DSP applications, however, do not terminate, so these results are not useful in this application. Simplified versions of the model have been explored by Commoner and Holt[Comm71a] and Reiter[Reit68a], but the restrictions imposed on the model are excessive. Specifically, the restricted models have nodes that can only consume or produce one sample on each input or output path. Reiter[Reit68a] tackled a scheduling problem, but assumed that each node in the graph corresponded to a processor, so only the firing times of the nodes needed to be determined.

2.2.7. Petri Nets.

Computation graphs have been shown to be a special case of *Petri nets*[Pete77a, Pete81a, Ager79a] or *vector addition systems*[Karp69a]. These more general models can be used to describe asynchronous systems, but implementations generally require expensive dynamic control flow.

2.2.8. Signal Representation Language

A final technique worth mentioning, although it has no apparent connection with data flow, is the *signal representation language*[Kope84a, Kope85a]. This representation deals with finite segments of signals as objects, operating on them as one would operate on numbers in a stack oriented calculator. Although undoubtedly a useful tool for experimenting with algorithms, the treatment of signals as objects does not seem at all appropriate for real-time applications.

2.3. SYNCHRONOUS DATA FLOW GRAPHS

In this chapter we concentrate on *synchronous* systems. At the risk of being pedantic, we define this precisely. A node in a data flow graph is a function that is invoked when there is enough input available to perform a computation (nodes lacking inputs can be invoked at any time). When a node is invoked, it will *consume* a fixed number of new input samples on each input path. These samples may remain in the system for some time to be used as old samples[Mess84a], but they will never again be considered new samples. A node is said to be *synchronous* if we can specify a-priori the number of input samples consumed on each input and the number of output samples produced on each output each time the node is invoked. Thus a synchronous node is

shown in figure 2-6(a) with a number associated with each input or output specifying the number of inputs consumed or the number of outputs produced. These numbers are part of the node definition. For example, a digital filter node would have one input and one output, and the number of input samples consumed or output samples produced would be one. A 2:1 decimator node would also have one input and one output, but would consume two samples for every sample produced. A *synchronous data flow* (SDF) graph is a network of synchronous nodes, as in figure 2-6(b). The modem example of figure 2-1 is shown in figure 2-7 as a SDF graph.

An immediate potential objection to the SDF paradigm is that general SDF graphs can be expressed as single sample rate SDF graphs, as shown in figure 2-8(a). This means that the prior art applicable to single sample rate SDF graphs can be applied to multiple sample rate SDF graphs[Comm71a, Reit68a, Croc75a, Braf78a, Zema83a, Schw85a]. However, the transformation is not always simple, as illustrated in figure 2-8(b), and may require replicating some of the nodes. A systematic method for performing this transformation is described in chapter 3, where the transformation is used to find a tight bound on the throughput achievable for a given SDF graph.

As mentioned before, SDF graphs are closely related to computation graphs, introduced in 1966 by Karp and Miller[Karp66a] and further explored by Reiter[Reit67a]. Computation graphs are slightly more elaborate than SDF graphs, in that each input to a node has two numbers associated with it, a *threshold* and the number of samples consumed. The threshold specifies the number of samples required to invoke the node, and may be different from the number

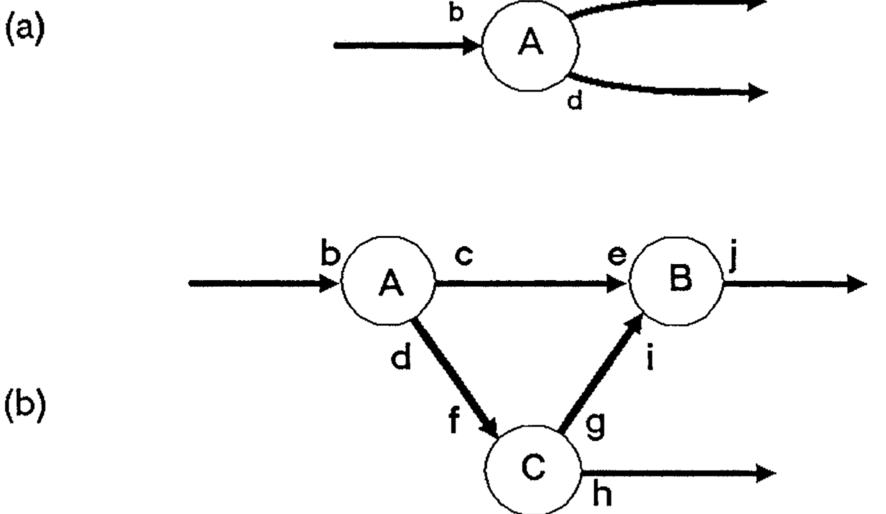


Figure 2-6: A synchronous data flow node (a) and graph (b).

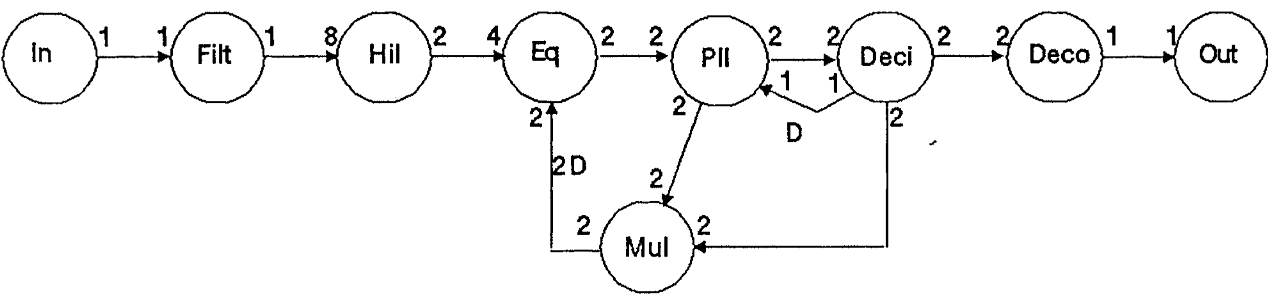


Figure 2-7: A voiceband data modem described as a synchronous data flow graph.

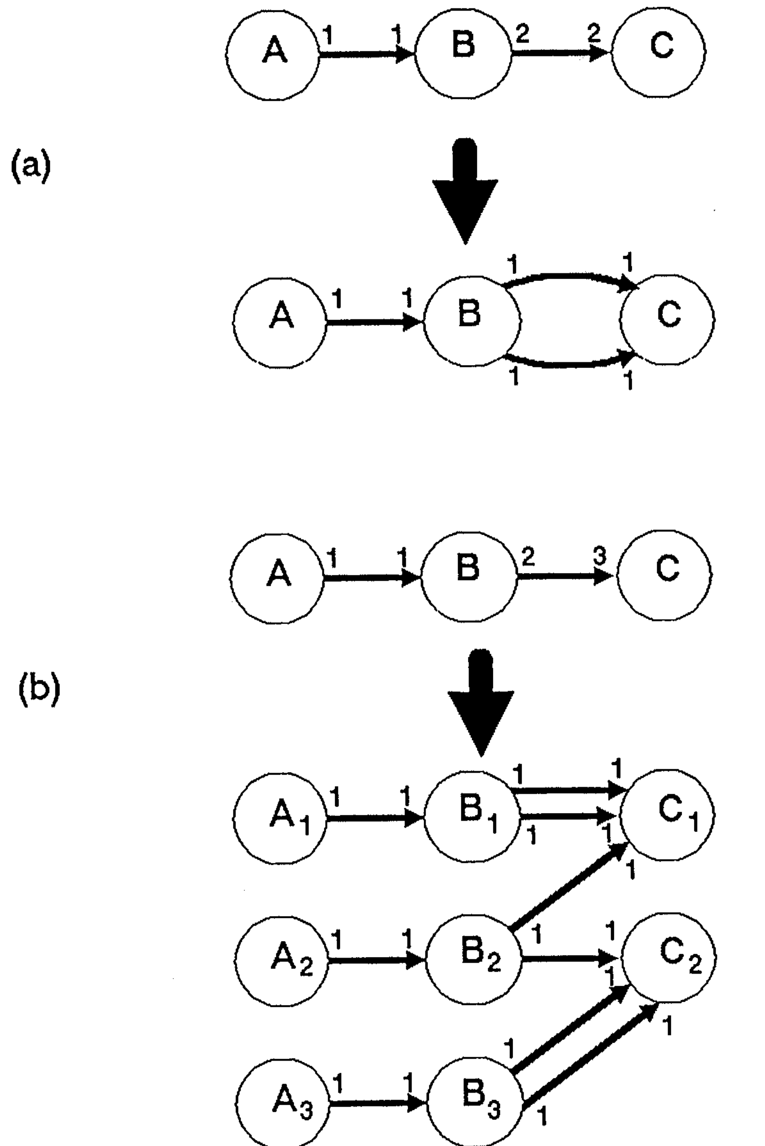


Figure 2-8: Two transformations of multiple-sample-rate SDF graphs into single-sample-rate SDF graphs using parallel data flow paths. The transformation in (a) is clearly simpler than the transformation in (b).

of samples consumed by the node. It cannot, of course, be smaller than the number of samples consumed. The use of a distinct threshold in the model, however, does not significantly change the results presented in this thesis, so for simplicity, we assume these two numbers are the same. To make it easier to describe such applications, we expand the model slightly to allow nodes with no inputs. These can fire at any time. Other results presented in[Karp66a] are only applicable to computations that terminate.

Implementing the signal processing system described by a SDF graph requires *buffering* the data samples passed between nodes and *scheduling* nodes so that they are executed when data is available. This could be done *dynamically*, in which case a run-time supervisor determines when nodes are ready for execution and schedules them onto processors as they become free. This run-time supervisor may be a software routine or specialized hardware, and is the same as the control mechanisms generally associated with data flow. It is a costly approach, however, in that the supervisory overhead can become severe, particularly if relatively little computation is done each time a node is invoked.

Our first task is to show that SDF graphs can be scheduled statically (at compile time), regardless of the number of processors, so that the overhead associated with dynamic control evaporates. Specifically, a *large grain compiler* determines the order in which nodes can be executed and constructs sequential code for each processor. Communication between nodes and between processors is set up by the compiler, so no run-time control is required beyond the traditional sequential control in the processors. The SDF paradigm gives the programmer a natural interface for easily constructing well structured signal pro-

cessing programs, with evident concurrency, and the large grain compiler maps this concurrency onto parallel processors. This chapter is dedicated mainly to demonstrating the feasibility of such a large grain compiler.

2.4. A LARGE GRAIN COMPILER FOR A SEQUENTIAL MACHINE

We need a methodology for translating from a SDF graph to a set of sequential programs running on a number of processors. Such a compiler has two basic tasks:

- Allocation of shared memory for the passing of data between nodes, if shared memory exists, or setting up communication paths if not.
- Scheduling nodes onto processors in such a way that data is available for a node when that node is invoked.

The first task is not an unfamiliar one. A single processor solution (which also handles asynchronous systems) is given by the buffer management techniques in BLOSIM[Mess84b]. The buffering problem is discussed in chapter 5, so this chapter will concentrate on the second task, that of scheduling nodes onto processors so that data is available when a node is invoked.

Some assumptions are necessary.

- The SDF graph has more nodes than processors. This is not a necessary assumption for feasibility, but it is certainly necessary for efficient use of available computing resources.
- The SDF graph is connected. If not, the separate graphs can be scheduled separately using subsets of the processors.

- The SDF graph is non terminating (cf.[Karp66a, Reit67a]) meaning that it can run forever without deadlock. As mentioned earlier, this assumption is natural for signal processing. We give below necessary and sufficient conditions for this to be true.

Specifically, our ultimate goal is a *periodic admissible parallel schedule*, designated PAPS. The schedule should be *periodic* because of the assumption that we are repetitively applying the same program on an infinite stream of data. The desired schedule is *admissible*, meaning that nodes will be scheduled to run only when data is available, and that a finite amount of memory is required for buffering. It is *parallel* in that more than one processing resource can be used. A special case is a periodic admissible *sequential* schedule, or PASS, which implements a SDF graph on a single processor. The method for constructing a PASS leads to a simple solution to the problem of constructing a PAPS, so we begin with the sequential schedule.

2.4.1. Construction of a PASS

A simple SDF graph is shown in figure 2-9, with each node and each arc labeled with a number. (The connections to the outside world are not considered, and for the remainder of this chapter, will not be shown. Thus, a node with one input from the outside will be considered a node with no inputs, which can therefore be scheduled at any time. The limitations of this approximation are discussed later in the chapter.) A SDF graph can be characterized by a matrix similar to the incidence matrix associated with directed graphs in graph theory. It is constructed by first numbering each node and arc, as in figure 2-9, and assigning a column to each node and a row to each arc. The $(i,j)^{th}$ entry in

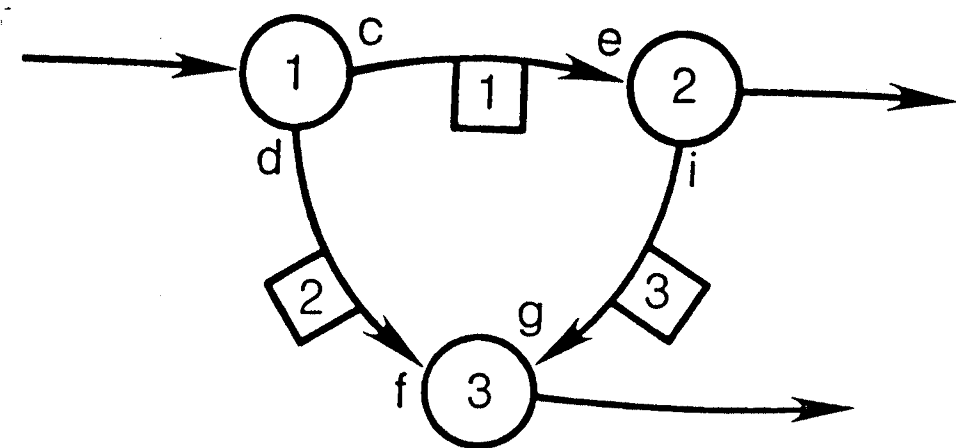


Figure 2-9. A synchronous data flow (SDF) graph showing the numbering of the nodes and arcs. The input and output arcs are ignored for now.

the matrix is the amount of data produced by node j on arc i each time it is invoked. If node j consumes data from arc i, the number is negative, and if it is not connected to arc i, then the number is zero. For the graph in figure 2-9 we get

$$\Gamma = \begin{bmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{bmatrix} \qquad (2.1)$$

This matrix can be called a *topology matrix*, and need not be square, in general.

If a node has a connection to itself (a *self loop*), then only one entry in Γ describes this link. This entry gives the net difference between the amount of data produced on this link and the amount consumed each time the node is invoked. This difference should clearly be zero for a correctly constructed graph, so the Γ entry describing a self loop should be zero.

We can replace each arc with a FIFO queue (buffer) to pass data from one node to another. The size of the queue will vary at different times in the execution. Define the vector $\mathbf{b}(n)$ to contain the queue sizes of all the buffers at time n. In BLOSIM[Mess84b] buffers are also used to store old samples (samples that have already been "consumed"), making implementations of delay lines particularly easy. These past samples are not considered part of the buffer size here.

For the sequential schedule, only one node can be invoked at a time, and for the purposes of scheduling it does not matter how long it runs. Thus, the index n can simply be incremented each time a node finishes and a new node is begun. We specify the node invoked at time n with a vector $\mathbf{v}(n)$, which has a one in the position corresponding to the number of the node that is invoked at time n and zeros for each node that is not invoked. For the system in figure 2-9, in a sequential schedule, $\mathbf{v}(n)$ can take one of three values,

$$\mathbf{v}(n) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \; OR \; \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \; OR \; \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.2}$$

depending on which of the three nodes is invoked. Each time a node is invoked, it will consume data from zero or more input arcs and produce data on zero or more output arcs. The change in the size of the buffer queues caused by invoking a node is given by

$$\mathbf{b}(n+1) = \mathbf{b}(n) + \Gamma \mathbf{v}(n) \tag{2.3}$$

The topology matrix Γ characterizes the effect on the buffers of running a node program.

This simple computation model is powerful. First we note that the computation model handles delays. The term *delay* is used in the signal processing sense, corresponding to a sample offset between the input and the output. We define a *unit delay* on an arc from node A to node B to mean that the nth sample consumed by B will be the $(n-1)^{th}$ sample produced by A. This implies that the first sample the destination node consumes is not produced by the source node at all, but is part of the initial state of the arc buffer. Indeed, a delay of d samples on an arc is implemented in our model simply by setting an initial condition for equation (2.3). Specifically, the initial buffer state, **b**(0), should have a d in the position corresponding to the arc with the delay of d units.

To make this idea firm, consider the example system in figure 2-10. The symbol "D" on an arc means a single sample delay, while "2D" means a two

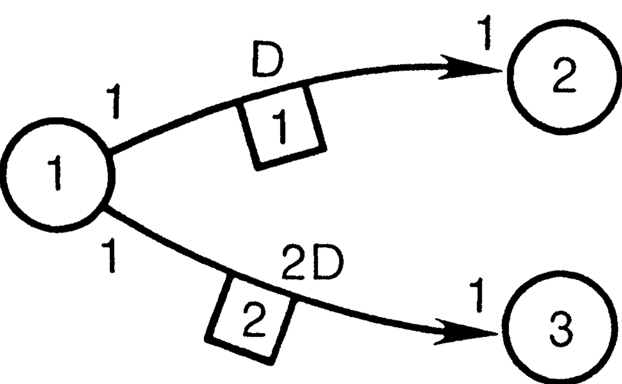


Figure 2-10. An example of a SDF graph with delays on the arcs. A delay simply means that the nth sample produced by the source is the $(n+1)^{th}$ sample consumed by the destination.

sample delay. The initial condition for the buffers is thus

$$b(0) = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{2.4}$$

Because of these initial conditions, node 2 can be invoked once and node 3 twice before node 1 is invoked at all. Delays, therefore, affect the way the system starts up.

Given this computation model we can

- find necessary and sufficient conditions for the existence of a PASS, and hence a PAPS;
- find practical algorithms that provably find a PASS if one exists;
- find practical algorithms that construct a reasonable (but not necessarily optimal) PAPS, if a PASS exists.

We begin by showing that a necessary condition for the existence of a PASS is

$$rank(\Gamma) = s-1 \tag{2.5}$$

where s is the number of nodes in the graph. We need a series of lemmas before we can prove this.

LEMMA 1: All topology matrices for a given SDF graph have the same rank.

PROOF: Topology matrices are related by renumbering of nodes and arcs, which translates into row and column permutations in the topology matrix. Such operations preserve the rank. QED.

LEMMA 2: A topology matrix for a *tree* graph has rank $s-1$ where s is the number of nodes (a tree is a connected graph without cycles, where we ignore the directions of the arcs).

PROOF: Proof is by induction. The lemma is clearly true for a two node tree. Assume that for an N node tree $rank(\Gamma_N) = N-1$. Adding one node and one link connecting that node to our graph will yield an $N+1$ node tree. A topology matrix for the new graph can be written

$$\Gamma_{N+1} = \left[\begin{array}{c|c} \Gamma_N & \mathbf{O} \\ \hline \multicolumn{2}{c}{\rho^T} \end{array} \right]$$

where $\mathbf{O}$ is a column vector full of zeros, and ρ^T is a row vector corresponding to the arc we just added. The last entry in the vector ρ^T is non-zero, because the node we just added corresponds to the last column, and it must be connected to the graph. Hence the last row is linearly independent from the other rows, so $rank(\Gamma_{N+1}) = rank(\Gamma_N) + 1$. QED.

LEMMA 3: For a connected SDF graph with topology matrix Γ,

$$rank(\Gamma) \geq s-1$$

where s is the number of nodes in the graph.

PROOF: Consider any spanning tree τ of the connected SDF graph (a spanning tree is a tree that includes every node in the graph). Now define Γ_τ to be the topology matrix for this subgraph. By lemma 2 $rank(\Gamma_\tau) = s-1$. Adding arcs

to the subgraph simply adds rows to the topology matrix. Adding rows to a matrix can increase the rank, if the rows are linearly independent of existing rows, but cannot decrease it. QED.

COROLLARY: $rank(\Gamma)$ is $s-1$ or s.

PROOF: Γ has only s columns, so its rank cannot exceed s. Therefore, by lemma 3, s and $s-1$ are the only possibilities. QED.

DEFINITION 1: An *admissible sequential schedule* ϕ is a non-empty ordered list of nodes such that if the nodes are executed in the sequence given by ϕ, the amount of data in the buffers ("buffer sizes") will remain non-negative and bounded. Each node must appear in ϕ at least once.

A *periodic admissible sequential schedule* (PASS) is a periodic and infinite admissible sequential schedule. It is specified by a list ϕ that is the list of nodes in one period.

For the example in figure 2-12 (ahead), $\phi = \{1,2,3,3\}$ is a PASS, but $\phi = \{2,1,3,3\}$ is not because node 2 cannot be run before node 1. The list $\phi = \{1,2,3\}$ is not a PASS because the infinite schedule resulting from repetitions of this list will result in an infinite accumulation of data samples on the arcs leading into node 3.

THEOREM 1: For a connected SDF graph with s nodes and topology matrix Γ, $rank(\Gamma) = s-1$ is a necessary condition for a PASS to exist.

PROOF: We must prove that the existence of a PASS of period p implies $rank(\Gamma) = s-1$. Observe from equation (2.3) that we can write

$$\mathbf{b}(p) = \mathbf{b}(0) + \Gamma \mathbf{q}$$

where

$$\mathbf{q} = \sum_{n=0}^{p-1} \mathbf{v}(n).$$

Since the PASS is periodic, we can write

$$\mathbf{b}(np) = \mathbf{b}(0) + n\,\Gamma \mathbf{q}.$$

Since the PASS is admissible, the buffers must remain bounded, by definition 1. The buffers remain bounded if and only if

$$\Gamma \mathbf{q} = \mathbf{O}$$

where $\mathbf{O}$ is a vector full of zeros. For $\mathbf{q} \neq \mathbf{O}$, this implies that $rank(\Gamma) < s$, where s is the dimension of $\mathbf{q}$. From the corollary of lemma 3, $rank(\Gamma)$ is either s or $s-1$, and so it must be $s-1$. QED.

This theorem tells us that if we have a SDF graph with a topology matrix of rank s, then the graph is somehow defective, because no PASS can be found for it. Figure 2-11 illustrates such a graph and its topology matrix. Any schedule for this graph will result either in deadlock or unbounded buffer sizes, as the reader can easily verify. The rank of the topology matrix indicates a *sample rate inconsistency* in the graph. In figure 2-12, by contrast, a graph without this defect is shown. The topology matrix has rank $s-1 = 2$, so we can find a vector $\mathbf{q}$ such that $\Gamma \mathbf{q} = \mathbf{O}$. Furthermore, the following theorem shows that we can find a positive integer vector $\mathbf{q}$ in the nullspace of Γ. This vector tells us how many times we should invoke each node in one period of a PASS. Referring

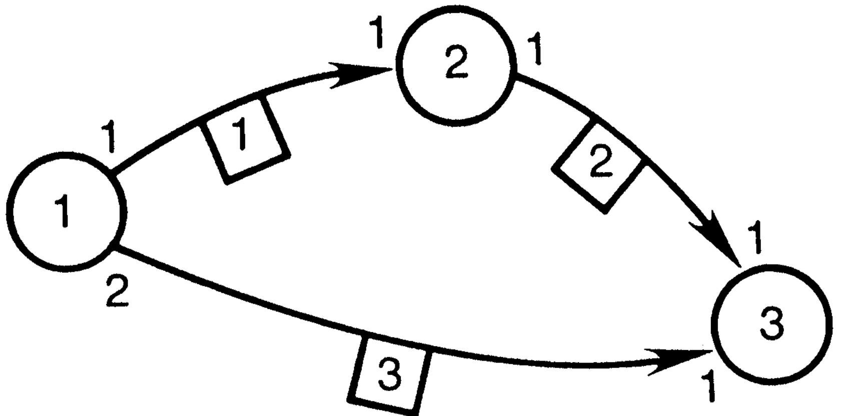


Figure 2-11. Example of a defective SDF graph with sample rate inconsistencies. The topology matrix is

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \end{bmatrix} \qquad \text{rank}(\Gamma) = s = 3$$

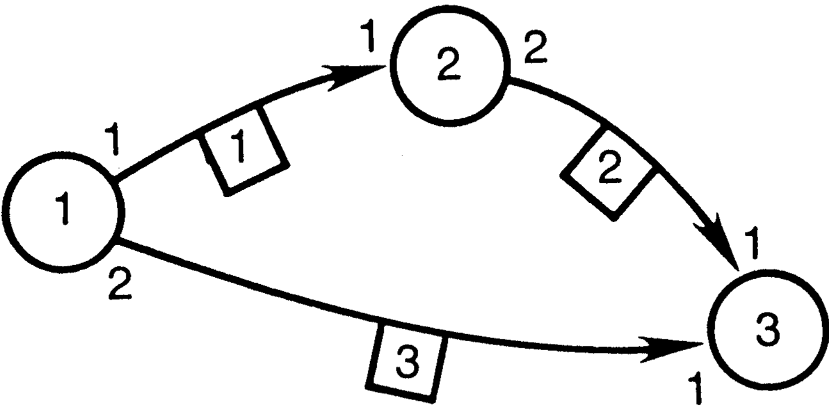


Figure 2-12. A SDF graph with consistent sample rates, and a positive integer vector $\mathbf{q}$ in the nullspace of the topology matrix Γ.

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 2 & -1 \\ 2 & 0 & -1 \end{bmatrix} \qquad \mathbf{q} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} \in \eta(\Gamma)$$

again to figure 2-12, the reader can easily verify that if we invoke node 1 once, node 2 once, followed by node 3 twice, that the buffers will end up once again in their initial state. As before, we prove some lemmas before getting to the theorem.

LEMMA 4: Assume a connected SDF graph with topology matrix Γ. Let $\mathbf{q}$ be any vector such that $\Gamma\mathbf{q} = \mathbf{O}$. Denote a connected path through the graph by the set $B = \{b_1, \cdots, b_L\}$, where each entry designates a node, and node b_1 is connected to node b_2, node b_2 to node b_3, up to b_L. Then all q_i, $i \in B$ are zero, or all are strictly positive, or all are strictly negative. Furthermore, if any q_i is rational then all q_i are rational.

PROOF: By induction. First consider a connected path of two nodes, $B_2 = \{b_1, b_2\}$. If the arc connecting these two nodes is the jth arc, then

$$q_{b_1}\Gamma_{jb_1} + q_{b_2}\Gamma_{jb_2} = 0$$

(by definition of the topology matrix, the jth row has only two entries). Also by definition, Γ_{jb_1} and Γ_{jb_2} are nonzero integers of opposite sign. The lemma thus follows immediately for B_2.

Now assuming the lemma is true for B_n, proving it true for B_{n+1} is trivial, using the same reasoning as in the proof for B_2, and considering the connection between nodes b_n and b_{n+1}.

COROLLARY: Given a SDF graph as in lemma 4, either all q_i are zero, or all are strictly positive, or all are strictly negative. Furthermore, if any one q_i is rational, then all are.

PROOF: In a connected SDF graph, a path exists from any node to any other. Thus, the corollary follows immediately from the lemma.

THEOREM 2: For a connected SDF graph with s nodes and topology matrix Γ, and with $rank(\Gamma) = s-1$, we can find a positive integer vector $\mathbf{q} \neq \mathbf{O}$ such that $\Gamma\mathbf{q} = \mathbf{O}$, where $\mathbf{O}$ is the zero vector.

PROOF: Since $rank(\Gamma) = s-1$, a vector $\mathbf{v} \neq \mathbf{O}$ can be found such that $\Gamma\mathbf{v} = \mathbf{O}$. Furthermore, for any scalar α, $\Gamma(\alpha\mathbf{v}) = \mathbf{O}$. Let $\alpha = 1/\ v_1$ and $\mathbf{v}' = \alpha\mathbf{v}$. Then $v'_1 = 1$, and by the corollary to lemma 4, all other elements in $\mathbf{v}'$ are positive

rational numbers. Let η be a common multiple of all the denominators of the elements of v' and let $\mathbf{q} = \eta \mathbf{v}'$. Then $\mathbf{q}$ is a positive integer vector such that $\Gamma \mathbf{q} = \mathbf{O}$. QED.

It may be desirable to solve for the *smallest* positive integer vector in the nullspace, in the sense of the sum of the elements. To do this, reduce each rational entry in v' so that its numerator and denominator are relatively prime. Euclid's algorithm (see for example[Blah85a]) will work for this. Now find the *least* common multiple η of all the denominators, again using Euclid's algorithm. Now $\eta \mathbf{v}'$ is the smallest positive integer vector in the nullspace of Γ.

We now have a necessary condition for the existence of a PASS, that the rank of Γ be $s-1$. A sufficient condition and an algorithm for finding a PASS would be useful. We now characterize a class of algorithms that will find a PASS if such exists, and will fail clearly if not. Thus, successful completion of such an algorithm is a sufficient condition for the existence of the PASS.

DEFINITION 2: A *predecessor* to a node η is a node feeding data to η.

LEMMA 5: To determine whether a node η in a SDF graph can be scheduled at time i, it is sufficient to know how many times η and its predecessors have been scheduled, and to know $\mathbf{b}(0)$, the initial state of the buffers. That is, we need not know in what order the predecessors were scheduled nor what other nodes have been scheduled in between.

PROOF: To schedule node η, each input buffer must have sufficient data. The

size of each input buffer j at time i is given by $[\mathbf{b}(i)]_j$, the jth entry in the vector $\mathbf{b}(i)$. From equation (2.3) we can write

$$\mathbf{b}(i) = \mathbf{b}(0) + \Gamma\mathbf{q}(i)$$

where

$$\mathbf{q}(i) = \sum_{n=0}^{i-1} \mathbf{v}(n). \tag{2.6}$$

The vector $\mathbf{q}(i)$ only contains information about how many times each node has been invoked before iteration i. The buffer sizes $[\mathbf{b}(i)]_j$ clearly depend only on $[\mathbf{b}(0)]_j$ and $[\Gamma\mathbf{q}(i)]_j$. The jth rows of Γ each have only two entries, corresponding to the two nodes connected to the jth buffer, so only the two corresponding entries of the $\mathbf{q}(i)$ vector can affect the buffer size. These entries specify the number of times η and its predecessors have been invoked, so this information and the initial buffer sizes $[\mathbf{b}(0)]_j$ is all that is needed. QED.

DEFINITION 3 *(CLASS S ALGORITHMS):* Given a positive integer vector $\mathbf{q}$ s.t. $\Gamma\mathbf{q} = \mathbf{O}$ and an initial state for the buffers $\mathbf{b}(0)$, the ith node is *runnable* at a given time if it has not been run q_i times and running it will not cause a buffer size to go negative. A *class S algorithm* is any algorithm that schedules a node if it is runnable, updates $\mathbf{b}(n)$ and stops (*terminates*) only when no more nodes are runnable. If a class S algorithms terminates before it has scheduled each node the number of times specified in the $\mathbf{q}$ vector, then it is said to be *deadlocked*.

Class S algorithms ("S" for Sequential) construct static schedules by simulating the effects on the buffers of an actual run. That is, the node programs are

not actually run. But they could be run, and the algorithm would not change in any significant way. Therefore, any dynamic (run time) scheduling algorithm becomes a class S algorithm simply by specifying a stopping condition, which depends on the vector **q**. It is necessary to prove that the stopping condition is sufficient to construct a PASS for any valid graph.

THEOREM 3: Given a SDF graph with topology matrix Γ and given a positive integer vector **q** s.t. $\Gamma \mathbf{q} = \mathbf{O}$, if a PASS of period $p = \mathbf{1}^T \mathbf{q}$ exists, where $\mathbf{1}^T$ is a row vector full of ones, any class S algorithm will find such a PASS.

PROOF: It is sufficient to prove that if a PASS ϕ of any period p exists, a class S algorithm will not deadlock before the termination condition is satisfied.

Assume that a PASS ϕ exists, and define $\phi(n)$ to its first n entries, for any n such that $1 \leqslant n \leqslant p$. Assume a given class S algorithm iteratively constructs a schedule, and define $\chi(n)$ to be the list of the first n nodes scheduled by iteration n.

We need to show that as n increases, the algorithm will build $\chi(n)$ and not deadlock before $n = p$, when the termination condition is satisfied. That is, we need to show that for all $n \in (1,...,p)$, there is a node that is runnable for any $\chi(n)$ that the algorithm may have constructed.

If $\chi(n)$ is any permutation of $\phi(n)$, then the $(n+1)^{th}$ entry in ϕ is runnable, by lemma 5, because all necessary predecessors must be in $\phi(n)$ and thus in $\chi(n)$. Otherwise, the first node α in $\phi(n)$ and not in $\chi(n)$ is runnable, also by

lemma 5. This is true for all $n \in (1,...,p)$, so the algorithm will not deadlock before $n = p$.

At $n = p$, each node i has been scheduled q_i times because no node can be scheduled more that q_i times (by definition 3), and $p = 1^T \mathbf{q}$. Therefore, the termination condition is satisfied, and $\chi(p)$ is a PASS. QED.

Theorem 3 tells us that if we are given a positive integer vector **q** in the nullspace of the topology matrix, that class S algorithms will find a PASS with its period equal to the sum of the elements in the vector, if such a PASS exists. It is possible, even if $rank(\Gamma) = s-1$, for no PASS to exist. Two such graphs are shown in figure 2-13. Networks with insufficient delays in directed loops are not computable.

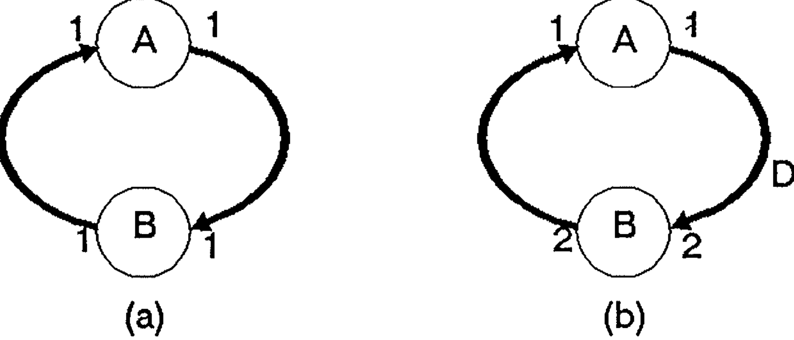


Figure 2-13. Two SDF graphs with consistent sample rates but no admissible schedule.

One problem remains. There are an infinite number of vectors in the nullspace of the topology matrix. How do we select one to use in the class S algorithm? We now set out to prove that given any positive integer vector in the nullspace of the topology matrix, if a class S algorithm fails to find a PASS then no PASS of any period exists.

LEMMA 6: Connecting one more node to a graph increases the rank of the topology matrix by at least one.

The proof of this lemma follows the same kinds of arguments as the proof of lemma 2. Rows are added to the topology matrix to describe the added connections to the new node, and these rows must be linearly independent of rows already in the topology matrix.

LEMMA 7: For any connected SDF graph with s nodes and topology matrix $\mathbf{\Gamma}$ a connected subgraph L with m nodes has a topology matrix $\mathbf{\Gamma}_L$ for which

$$rank(\mathbf{\Gamma}) = s-1 \quad \Rightarrow \quad rank(\mathbf{\Gamma}_L) = m-1.$$

I. e. all subgraphs have the right rank.

PROOF: By contraposition. We prove that

$$rank(\mathbf{\Gamma}_L) \neq m-1 \quad \Rightarrow \quad rank(\mathbf{\Gamma}) \neq s-1.$$

From the corollary to lemma 3, if $rank(\mathbf{\Gamma}_L) \neq m-1$ then $rank(\mathbf{\Gamma}_L) = m$. Then $rank(\mathbf{\Gamma}) \geq m + (s-m) = s$, by repeated applications of lemma 6, so $rank(\mathbf{\Gamma}) = s$. QED.

The next lemma shows that given a nullspace vector **q**, in order to run any node the number of times specified by this vector. it is never necessary to run any other node more than the number of times specified by the vector.

LEMMA 8: Consider the subgraph of a SDF graph formed by any node α and all its immediate predecessors (nodes that feed it data, which may include α itself). Construct a topology matrix Γ for this subgraph. If the original graph has a PASS, then by theorem 1 and lemma 7, $rank(\Gamma) = m - 1$, where m is the number of nodes in the subgraph. Find any positive integer vector **q** s.t. $\Gamma \mathbf{q} = \mathbf{O}$. Such a vector exists because of theorem 2. Then it is never necessary to run any predecessor β more than q_β times in order to run α x times, for any $x \leq q_\alpha$.

PROOF: The node α will not consume any data produced by the yth run of β for any $y > q_\beta$. From the definition of Γ and **q** we know that $aq_\alpha = bq_\beta$ where a and b are the amount of data consumed and produced on the link from β to α. Therefore, running β only q_β times generates enough data on the link to run α q_α times. More runs will not help. QED.

THEOREM 4: Given a SDF graph with topology matrix Γ and a positive integer vector **q** s.t. $\Gamma \mathbf{q} = \mathbf{O}$, a PASS of period $p = 1^T \mathbf{q}$ exists if and only if a PASS of period Np exists for any integer N.

PROOF:

PART 1:

It is trivial to prove that the existence of a PASS of period p implies the existence of a PASS of period Np because the first PASS can be composed N times to produce the second PASS.

PART 2:

We now prove that the existence of a PASS ϕ of period Np implies the existence of a PASS of period p.

Consider the subset θ of ϕ containing the first q_α runs of each node α. If θ is the first p elements of ϕ then it is a schedule of period p and we are done. If it is not, then there must be some node β that is executed more than q_β times before all nodes have been executed $\mathbf{q}$ times. But by lemma 8, these "more than $\mathbf{q}$" executions of β cannot be necessary for the later "less than or equal to $\mathbf{q}$" executions of other nodes. Therefore, the "less than or equal to $\mathbf{q}$" executions can be moved up in the list ϕ so that they precede all "more than $\mathbf{q}$" executions of β, yielding a new PASS ϕ' of period Np. If this process is repeated until all "less than $\mathbf{q}$" executions precede all "more than $\mathbf{q}$" executions, then the first p elements of the resulting schedule will constitute a schedule of period p. QED.

COROLLARY: Given any positive integer vector $\mathbf{q} \in \eta(\Gamma)$, the null space of Γ, a PASS of period $p = \mathbf{1}^T\mathbf{q}$ exists if and only if a PASS exists of period $r = \mathbf{1}^T\mathrm{v}$ for any other positive integer vector $\mathrm{v} \in \eta(\Gamma)$.

PROOF: For any PASS at all to exist, it is necessary that $rank(\Gamma) = s-1$, by

theorem 1. So the nullspace of Γ has dimension one, and we can find a scalar c such that

$$\mathbf{q} = c\,\mathbf{v}.$$

Furthermore, if both of these vectors are integer vectors, then c is rational and we can write

$$c = \frac{n}{d}$$

where n and d are both integers. Therefore,

$$d\,\mathbf{q} = n\,\mathbf{v}.$$

By theorem 4, a PASS of period $p = 1^T \mathbf{q}$ exists if and only if a PASS of period $dp = 1^T (d\,\mathbf{q})$ exists. By theorem 4 again, a PASS of period dp exists if and only if a PASS of period $r = 1^T \mathbf{v}$ exists. QED.

DISCUSSION: The four theorems and their corollaries have great practical importance. We have specified a very broad class of algorithms, deisgnated *class S algorithms*, which, given a positive integer vector $\mathbf{q}$ in the nullspace of the topology matrix, find a PASS with period equal to the sum of the elements in $\mathbf{q}$. Theorem 3 guarantees that these algorithms will find a PASS if one exists. Theorems 1 and 2 guarantee that such a vector $\mathbf{q}$ exists if a PASS exists, and the proof to theorem 2 suggests a simple algorithm for solving for the smallest such vector. The corollary to theorem 4 tells us that is does not matter what positive integer vector we use from the nullspace of the topology matrix, so we can simplify our system by using the smallest such vector, thus obtaining a PASS with minimum period.

Given these theorems, we now give a simple sequential scheduling algorithm that is of class S, and therefore will find a PASS if one exists.

1. Solve for the smallest positive integer vector $\mathbf{q} \in \eta(\Gamma)$.
2. Form an arbitrarily ordered list L of all nodes in the system.
3. For each $\alpha \in L$, schedule α if it is runnable, trying each node once.
4. If each node α has been scheduled q_α times, STOP.
5. If no node in L can be scheduled, indicate a deadlock (an error in the graph)
6. Else, go to 3 and repeat.

Theorem 3 tells us that this algorithms will not deadlock if a PASS exists. Two SDF graphs which cause deadlock and have no PASS are shown in figure 2-13.

Since the run time is the same for any PASS (the one machine available is always busy), no algorithm will produce a better run time than this one. However, class S algorithms exist which construct schedules minimizing the memory required to buffer data between nodes. Using dynamic programming or integer programming, such algorithms are easily constructed. Such algorithms are discussed in chapter 3.

A large grain data flow programming methodology offers concrete advantages for single processor implementations. The ability to interconnect modular blocks of code (nodes) in a natural way could considerably ease the task of programming high performance signal processors, even if the blocks of code themselves are programmed in assembly language. The gain is somewhat analogous to that experienced in VLSI design through the use of standard cells. For synchronous systems, the penalty in run time overhead is minimal. But a single

processor implementation cannot take advantage of the explicit concurrency in a SDF description. The next section is dedicated to explaining how the concurrency in the description can be used to improve the throughput of a multiprocessor implementation.

2.5. THE PARALLEL SCHEDULING PROBLEM

Clearly, if a workable schedule for a single processor can be generated, then a workable schedule for a multiprocessor system can also be generated. Trivially, all the computation could be scheduled onto only one of the processors. However, in general, the run time can be reduced substantially by distributing the load more evenly. We show in this section how the multiprocessor scheduling problem can be reduced to a familiar problem in operations research for which good heuristic methods are available.

We assume a tightly coupled parallel architecture, so that communication costs are not the overriding concern. Furthermore, we assume homogeneity; all processors are the same, so they process a node in a SDF graph in the same amount of time. The π processor of chapter 4 meets these assumptions. It is not necessary that the processors be synchronous, although the implementation will be simpler if they are.

A periodic admissible *parallel* schedule (PAPS) is a set of lists $\{\psi_i \; ; i = 1, \cdots ,M\}$ where M is the number of processors, and ψ_i specifies a periodic schedule for processor i. If ϕ is the corresponding PASS with the smallest possible period P_s, then it follows that the total number P_p of node invocations in the PAPS should be some integer multiple J of P_s. The multiple J is

called the *blocking factor*. We could, of course, choose $J = 1$, but as we will show below, schedules that run faster might result if a larger J is used. If the "best" blocking factor is known, then construction of a good PAPS is not too hard. Unfortunately, as we will see, a method for determining the optimal blocking factor has not been found, so heuristic solutions are required.

For a sequential schedule, precedences are enforced by the schedule. For a multiprocessor schedule, the situation is not so simple. We will assume that some method enforces the integrity of the parallel schedules. That is, if a schedule on a given processor dictates that a node should be invoked, but there is no input data for that node, then the processor halts until this input data is available. The task of the scheduler is thus to construct a PAPS that minimizes the *iteration period*, the run time for one period of the PAPS divided by J, and avoids deadlocks. The mechanism to enforce the integrity of the communication between nodes on different processors could use semaphores in shared memory or simple "instruction-count" synchronization, where no-ops are executed as necessary to maintain synchrony among processors, depending on the multiprocessor architecture. Instruction count synchronization will work with the π architecture of chapter 4 as long as the execution time of each node in the graph is data independent.

The first step is to construct an acyclic precedence graph for J periods of the PASS ϕ. A precise class S algorithm will be given for this procedure below, but we start by illustrating it with the example in figure 2-14(a). The SDF graph in figure 2-14(a) is neither acyclic nor a precedence graph. Examination of the number of inputs consumed and outputs produced for each node reveals that

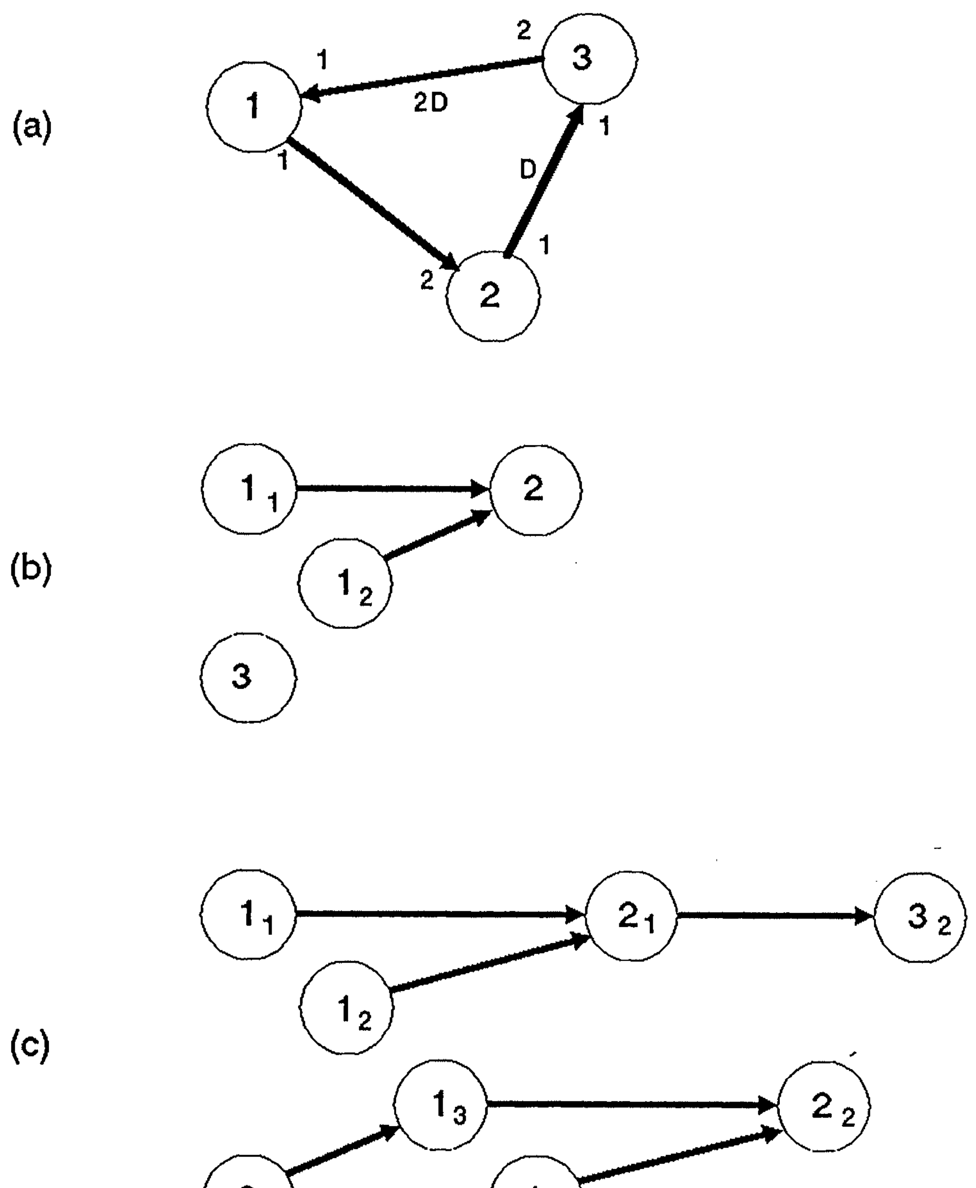


Figure 2-14:

(a) A synchronous data flow graph without self-loops.

(b) An acyclic precedence graph for $J = 1$.

(c) An acyclic precedence graph for $J = 2$.

node 1 should be invoked twice as often as the other two nodes. Further, given the delays on two of the arcs, we note that there are three possible minimum period PASS's, $\phi_1 = \{1,3,1,2\}$, $\phi_2 = \{3,1,1,2\}$, or $\phi_1 = \{1,1,3,2\}$, each with period $P_s = 4$. A schedule that is not a PASS is $\phi_1 = \{2,1,3,1\}$, because node 2 is not immediately runnable. Figure 2-14(b) shows the precedences involved in all three schedules. Figure 2-14(c) shows the precedences using a blocking factor of two ($J=2$). In these figures, there is no precedence shown between successive invocations of the same node. A practical implementation, however, is likely to have such precedences in order to preserve the integrity of the buffers. In other words, two processors accessing the same buffer at the same time may not be tolerable, depending on how the buffers are implemented. Precedences between successive invocations of the same node are the result of implicit *self-loops* in the SDF graph, shown explicitly in figure 2-15(a). The $J=1$ and $J=2$ precedence graphs are also shown. The self-loops are also required, of course, if the node has a *state* that is updated when it is invoked. Self loops have no effect on the topology matrix, because the number of samples consumed and produced are the same, and the node is the same, so they cancel. We will henceforth assume that all nodes have self loops, thus avoiding the potential implementation difficulties.

If we have two processors available, a PAPS for $J=1$ is

$$\psi_1 = \{3\}$$

$$\psi_2 = \{1,1,2\}.$$

When this system starts up, nodes 3 and 1 will run concurrently. The precise timing of the run depends on the run time of the nodes. If we assume that the run time of node 1 is a single time unit, the run time of node 2 is 2 time units, and the run time of node 3 is 3 time units, then the timing is shown in figure

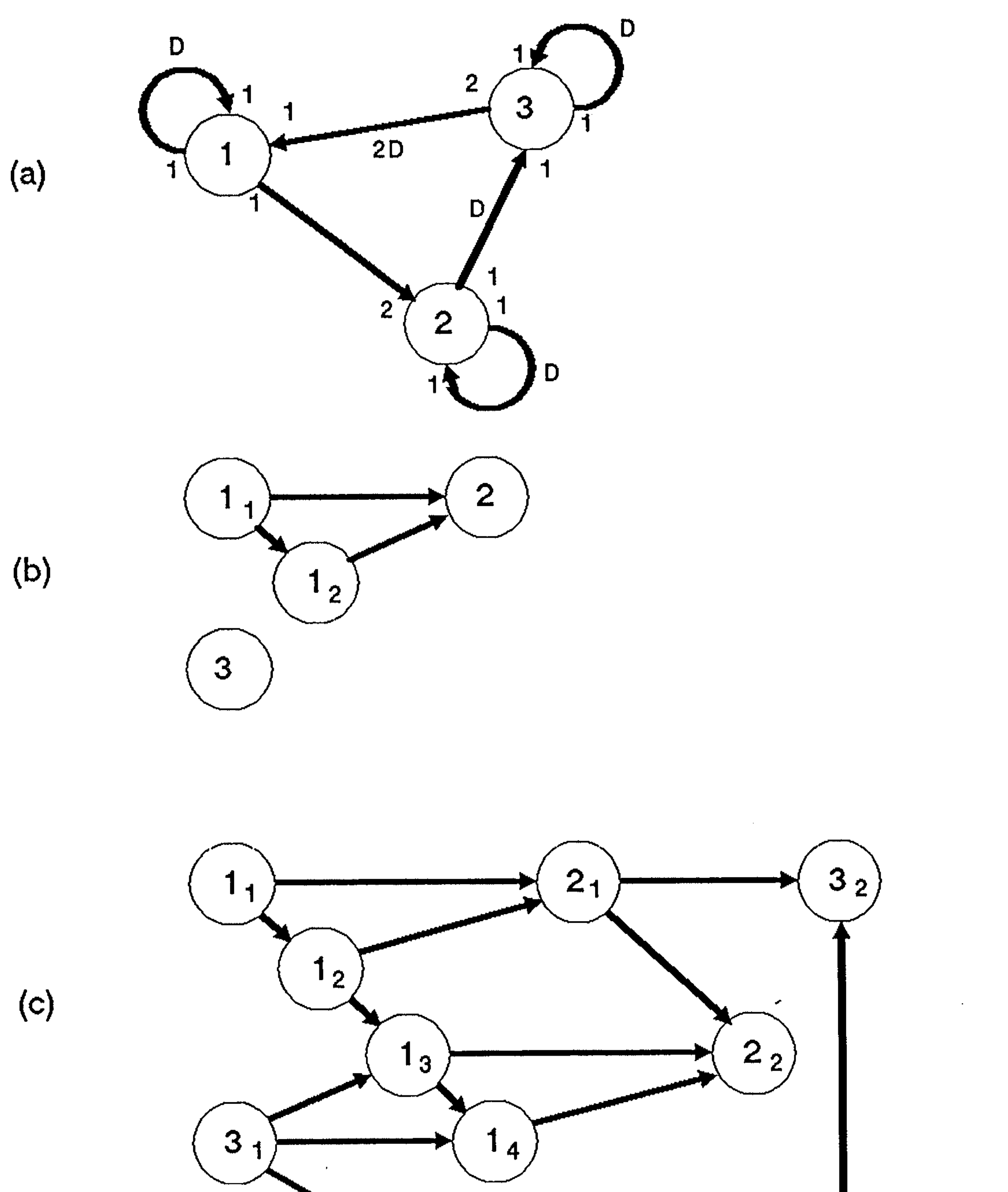


Figure 2-15:

(a) A synchronous data flow graph with self-loops.

(b) An acyclic precedence graph for $J = 1$.

(c) An acyclic precedence graph for $J = 2$.

2-16(a). We assume for now that the entire system is resynchronized after each execution of one period of the PAPS. Such a schedule is called a *blocked schedule*.

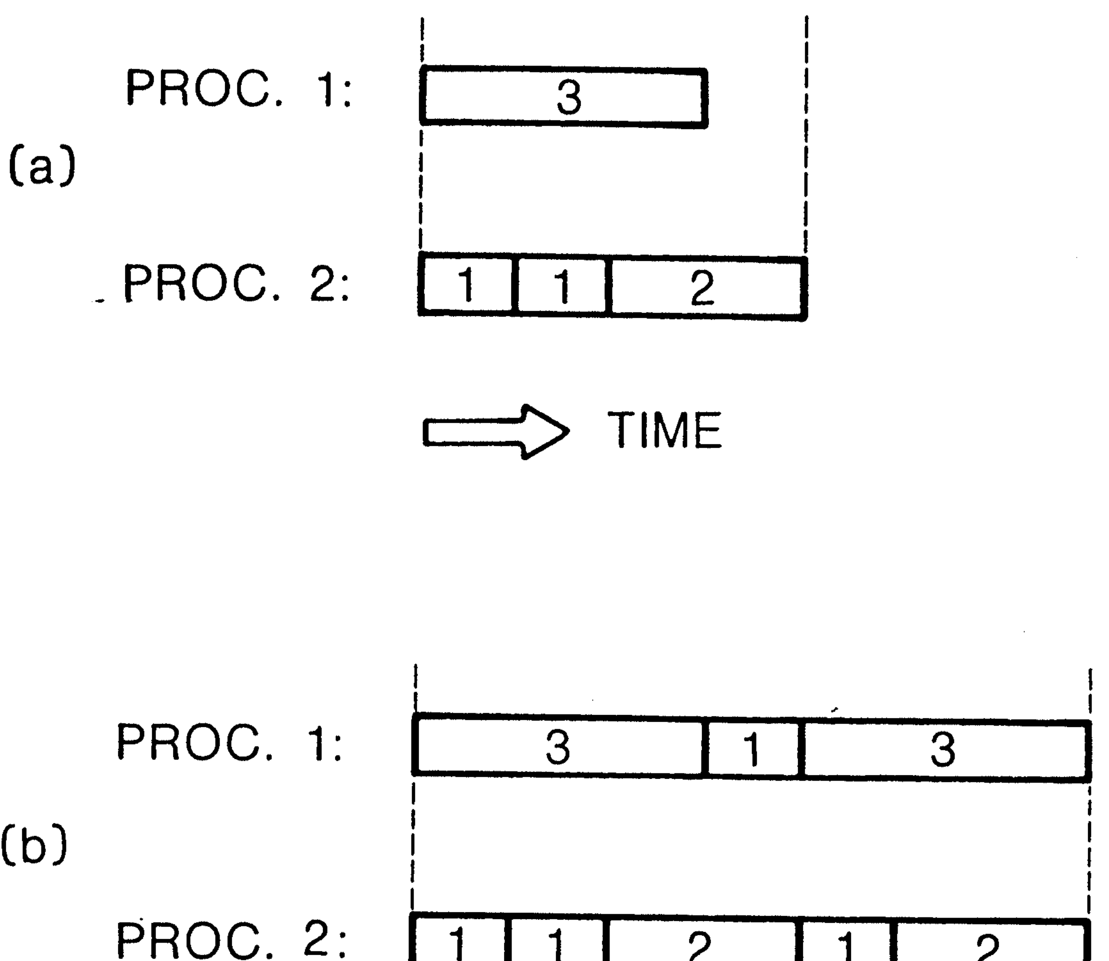


Figure 2-16. One period of each of two periodic blocked schedules for the SDF graph in figure 2-15(a) for $J=1$ (a) and $J=2$ (b).

A PAPS constructed for $J=2$, using the precedence graph of figure 2-15(c) will, however, perform better. Such a PAPS is given by

$$\psi_1 = \{3,1,3\}$$

$$\psi_2 = \{1,1,2,1,2\}$$

and its timing is shown in figure 2-16(b). Since both processors are kept always busy, this schedule is better than the $J=1$ schedule, and no better schedule exists. The problem of choosing the blocking factor J will be considered in the following chapter.

The problem of constructing a parallel schedule given an acyclic precedence graph is a familiar one. It is identical with assembly line problems in operations research, and can be solved for the optimal schedule, but the solution is combinatorial in complexity. This may not be a problem for small SDF graphs, and for large ones we can use well studied heuristic methods, the best being members of a family of "critical path" methods[Adam74a]. An early example, known as the Hu-level-scheduling algorithm[Hu61a], closely approximates an optimal solution for most graphs[Kohl75a, Adam74a], and is simple. To implement this method, a *level* is determined for each node in the acyclic precedence graph, where the level of a given node is the worst case of the total of the run times of nodes on a path from the given node to the terminal node of the graph. The terminal node is a node with no successors. If there is no unique terminal node, one can be created with zero run time. This node is then considered a successor to all nodes that otherwise have no successors. Figure 2-17(a) shows the levels for the $J=1$ precedence graph and figure 2-17(b) shows them for the $J=2$ precedence graph, for the example of figure 2-15. Finally, the Hu level scheduling algorithm simply schedules available nodes with the highest level first.

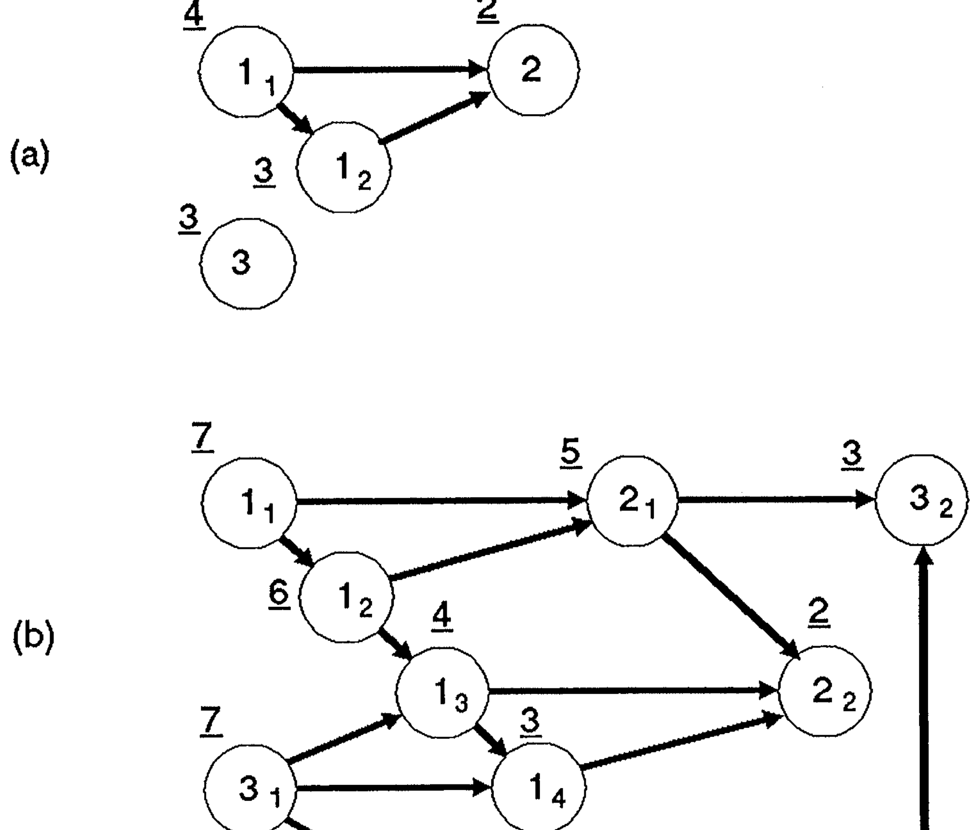


Figure 2-17:
(a) An acyclic precedence graph for $J = 1$ with levels indicated.
(b) An acyclic precedence graph for $J = 2$ with levels indicated.

When there are more available nodes with the same highest level than there are processors, a reasonable heuristic is to schedule the ones with the longest run time first. Such an algorithm produces the schedules shown in figure 2-16, the optimal schedules for the given precedence graphs. More elaborate examples are given in chapter 5.

We now give a class S algorithm that systematically constructs an acyclic precedence graph. First we need to understand how we can determine when the execution of a particular node is necessary for the invocation of another node.

Consider a SDF graph with a single arc a connecting node η to node α. Assume this arc is part of a SDF graph with topology matrix Γ. The number of samples required to run α j times is $-j\,\Gamma_{a\alpha}$, where $\Gamma_{a\alpha}$ is the entry in the topology matrix corresponding to the connection between arc a and node α. Of these samples, b_a are provided as initial conditions. If $b_a \geqslant -j\,\Gamma_{a\alpha}$ then there is no dependence of the jth run of α on η. Otherwise, the number of samples required of η is $-j\,\Gamma_{a\alpha} - b_a$. Each run of η produces $\Gamma_{a\eta}$ samples. Therefore, the jth run of α depends on the first d runs of η, where

$$d = \left\lceil \frac{-j\,\Gamma_{a\alpha} - b_a}{\Gamma_{a\eta}} \right\rceil, \tag{2.7}$$

where the notation $\lceil \cdots \rceil$ indicates the ceiling function.

Now we give a precise algorithm. We assume that we are given the smallest integer vector $\mathbf{q}$ in the nullspace of Γ and the "best" multiple J, so that we wish to construct an acyclic precedence graph with the number of repetitions of each node given by $J\mathbf{q}$. We will discuss later how we get J. Each time we add a node to the graph we will increment a counter i, update the buffer state $\mathbf{b}(i)$, and update the vector $\mathbf{q}(i)$ defined in equation 2.6. This latter vector indicates

how many instances of each node have been put into the precedence graph. We let L designate an arbitrarily ordered list of all nodes in the graph.

```
INITIALIZATION:
i=0;
q(0) = o;

THE MAIN BODY:
while nodes are runnable {
    for each α ∈ L {
        if α is runnable then {
            create the (q_α(i)+1)^th instance of the node α;
            for each input arc a on α {
                let η be the predecessor node for arc a;
                compute d using equation (2.7);
                if d<0 then let d=0;
                establish precedence links with the first d instances of η;
            }
            let v(i) be a vector with zeros except a 1 in position α;
            let b(i+1) = b(i) + Γv(i);
            let i=i+1;
        }
    }
}
```

This algorithm has been programmed in LISP as part of the *Gabriel* system, an experimental synchronous data flow programming environment. The examples in chapter 5 were all scheduled using this program.

2.6. LIMITATIONS OF THE MODEL

We rely on experience to claim that most signal processing systems are adequately described by SDF graphs. However, the model does not describe all systems of interest. In this section, we explore some specific limitations.

2.6.1. Conditionals

The SDF model permits conditional control flow within a node, but not on a greater scale. While large scale conditional control flow is a mainstay in general purpose computing, it is rare in signal processing. Occasionally, however, it is required, and therefore must be supported by any practical programming system. Two types on conditional control may be required, *data dependent* or *state dependent*. An example of a system with data dependent control flow contains a node that passes its input sample to its first output if the sample is less than some threshold, and to its second output otherwise. Such a node is an *asynchronous* node, because it is not possible to specify *a priori* how many samples will be produced on each output when the node is invoked. Systems with asynchronous nodes are dealt with in the next subsection.

State dependent control flow refers to such control structures as iteration, where the number of iterations does not depend on data coming into the system from outside. Such iteration is easily handled by the SDF model. On a small scale, of course, it may he handled entirely within a node. On a larger scale, it may be handled by replicating a node as many times as required. The iteration is then managed by the scheduler.

2.6.2. Asynchronous Graphs

Although rare in signal processing, asynchronous graphs do exist. That is, we can conceive of nodes where the amount of data consumed or produced on the input or output paths is data dependent, so no fixed number can be specified statically. The simplest solution is to divide a graph into synchronous subgraphs connected only by asynchronous links. Then these subgraphs can be

scheduled on different processors with an asynchronous communication protocol enforced in interprocessor communication. Such a protocol is generally readily available in multiprocessor systems. The asynchronous links are then handled by the scheduler as if they were connections to the outside world (discussed in the next subsection).

Another solution that is not so simple but may sometimes yield better performance in exceptional circumstances, is to implement a run time supervisor, as done in[Mess84b, Mess84a]. The run time supervisor would only handle the scheduling of entire synchronous subsystems, a much smaller task than scheduling all the nodes.

2.6.3. Connections to the Outside World

The SDF model does not adequately address the possible real time nature of connections to the outside world. Arcs into a SDF graph from the outside world are ignored by the scheduler. It may be desirable to schedule a node that collects inputs as regularly as possible, to minimize the amount of buffering required on the inputs. As it stands now, the model cannot reflect this, so buffering of input data is likely to be required.

2.6.4. Data Dependent Run Times

In the construction of a PAPS, we assume the run time of each node is known *a priori*. The run time, however, may be data dependent. However, in hard real time applications, it must also be bounded, independent of the data. The schedule must perform even with worst case data that causes maximum run times for all nodes. In this situation, there is no disadvantage to scheduling

using the worst case run times.

2.7. CONCLUSION

This chapter describes the theory necessary to develop a signal processing programming methodology that offers programmer convenience without squandering computation resources. Programs are described as synchronous data flow graphs, where connections between nodes indicate the flow of data samples, and the function of each node can be specified using a conventional programming language. Blocks are executed whenever input data samples are available. The advantages of such a description are numerous. First, it is a natural way to describe signal processing systems, where the nodes are second order recursive digital filters, FFT butterfly operators, adaptive filters, and so on. Second, such a description exhibits much of the available concurrency in a signal processing algorithm, making multiple processor implementations easier to achieve. Third, nodes are modular, and may be re-used in new system designs. Nodes are viewed as black boxes with input and output data streams, so re-using a node simply means reconnecting it in a new system. Fourth, multiple sample rates are easily described under the programming paradigm.

We describe high efficiency techniques for converting a large grain data flow description of a signal processing system into a set of ordinary sequential programs that run on parallel machines (or, as a special case, a single machine). This conversion is accomplished by a *large grain compiler* so called because it does not translate a high level language into a low level language, but rather assembles pieces of code (written in any language) for sequential or parallel execution. Given the number of samples produced or consumed by any node on

each invocation, techniques are given (and proven valid) for constructing sequential or parallel schedules that will execute deterministically, without the run time overhead generally associated with data flow. For the multiprocessor case, the problem of constructing a blocked schedule that executes with maximum throughput is shown to be equivalent to a standard operations research problem with well studied heuristic solutions that closely approximate the optimum. Given these techniques, the benefits of large grain data flow programming can be extended to those signal processing applications where performance demands are so severe that little inefficiency for the sake of programmer convenience can be tolerated. The next chapter discusses bounds on the performance of a parallel schedule, optimal blocking factors, cutset transformations for improving a schedule for a given blocking factor, and techniques for minimizing memory dedicated to buffering.

3

IMPROVING PERFORMANCE OF A PARALLEL SCHEDULE

In the last chapter we derived necessary and sufficient conditions for the existence of periodic admissible sequential schedules for a SDF graph and gave algorithms for both sequential and parallel scheduling. The periodic schedules generated are *blocked* schedules, meaning that one cycle is completed before the next is begun. In this chapter we assess the limitations of blocked schedules and describe techniques for improving performance.

Recall if $\mathbf{q}$ is the smallest positive integer vector in the nullspace of the topology matrix, and J is the blocking factor, then $J\mathbf{q}$ tells us how many times each node will be scheduled in one cycle of a periodic schedule. For some SDF graphs, it is possible to improve the performance of a blocked schedule for a given blocking factor by rearranging and adding delays to a graph. The basic technique is the *cutset transformation*, explained in section 3.1. However, even with such techniques, it is not always possible to construct a blocked schedule that performs as well as any unblocked schedule. To study the absolute

performance of blocked schedules, we need to understand the bounds on performance imposed by the structure of the algorithm.

For DSP applications in which algorithms are applied to an infinite stream of data, it is sometimes possible to take advantage of any number of parallel processors to arbitrarily increase the throughput. Most algorithms, however, involve some form of *recursion*, or feedback, in which an output is a function of previous outputs. Such recursions appear in SDF graphs as directed loops. Any algorithm involving recursive computations has an upper bound on its *throughput*, defined loosely as the number of outputs that can be computed per unit time. (More precise terminology is given in section 3.2.) Often it is possible to modify the realization of the algorithm to get unbounded throughput (See for example[Lu85a]), but given a realization of an algorithm, the throughput is bounded if there is recursion. If the throughput is bounded, then there is a limit on the number of parallel processors that can be used effectively.

For a special case of SDF graphs with a single sample rate, a bound on the throughput is the worst case computation time in a directed loop divided by the number of delays in the loop[Fett76a, Renf81a]. This result is reviewed in section 3.2. In section 3.3 we give a systematic method for transforming a general SDF graph into such a single-sample-rate SDF graph so that we can compute the throughput bound for general SDF graphs using this bound.

Once the bound on the throughput for general SDF graphs is known, we would hope that with an optimal blocking factor and enough processors, the bound can be met using a blocked schedule. Unfortunately, as we show in section 3.4, the optimal blocking factor is not always finite. In such circumstances,

restricting ourselves to blocked schedules implies an engineering tradeoff between approximating the throughput bound with a large blocking factor and minimizing the memory required by the realization. Alternatively, given the systematic method for translating a general SDF graph into a single sample rate SDF graph, the techniques given by Schwartz[Schw85a] can be used to construct *cyclo static* schedules. Cyclo static schedules, unlike blocked schedules, overlap successive cycles such that the throughput bound is always achievable. Unfortunately, the method Schwartz gives for constructing cyclo static schedules is exponential in complexity. Schwartz argues that for practical applications the complexity is manageable if we construct a schedule with maximum throughput using the minimum number of processors. The search for the solution is sufficiently constrained that most alternatives can be immediately rejected, so the algorithm runs quickly enough, at least for simple examples. In the case of the π processor, the number of processors is fixed *a priori*, and we wish to find the schedule that maximizes the throughput. The search is not adequately constrained, and all but the most trivial examples may not be solvable in a reasonable time. Therefore, for cyclo static scheduling to be applied to the π processor, a polynomial time algorithm (or heuristic) must be found.

Finally, in section 3.5, we describe techniques for minimizing of the amount of memory dedicated to buffering data. This is done through scheduling to minimize the maximum length of buffers and/or memory allocation so that multiple buffers share the same memory locations.

3.1. CUTSET TRANSFORMATIONS

Given a SDF graph, it is often possible to systematically manipulate the delays in the graph to enhance the parallelism and improve the throughput of a schedule with a particular blocking factor. This is done through *cutset transformations*. A simple example is illustrated in figure 3-1. The toplology matrix is

$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

and the smallest positive integer vector in its nullspace is $\mathbf{q} = [1,1,1]^T$. The SDF graph in figure 3-1(a) has the acyclic precedence graph in figure 3-1(b) for $J=1$, where J is the blocking factor. There is no evident concurrency in the precedence graph, and only one processor can be effectively used. However, putting delays on the feed-forward arcs as shown in figure 3-1(c) does not usually alter the computation. This is analogous to pipelining, and greatly increases the concurrency evident in the acyclic precedence graph for the $J=1$, shown in figure 3-1(d). If the three nodes have equal execution time, then three processors can be block-scheduled on three processors with 100% utilization.

The above simple example illustrates a principle that is quite useful. Consider any *feed-forward cutset* of a SDF graph. A *cutset* is a set of arcs that cross a closed surface in the graph (or curve, for a planar graph). The closed surface (or curve) cannot touch any node in the graph. Figure 3-2 shows the graph of figure 3-1 with the closed curves corresponding to the two cutsets on which we put the delays. A *feed-forward* cutset is one for which the directed arcs all go in the same direction across the surface.

For multiple-sample rate SDF graphs we cannot simply put an equal number of delays on all arcs in the cutset. Consider the example shown in

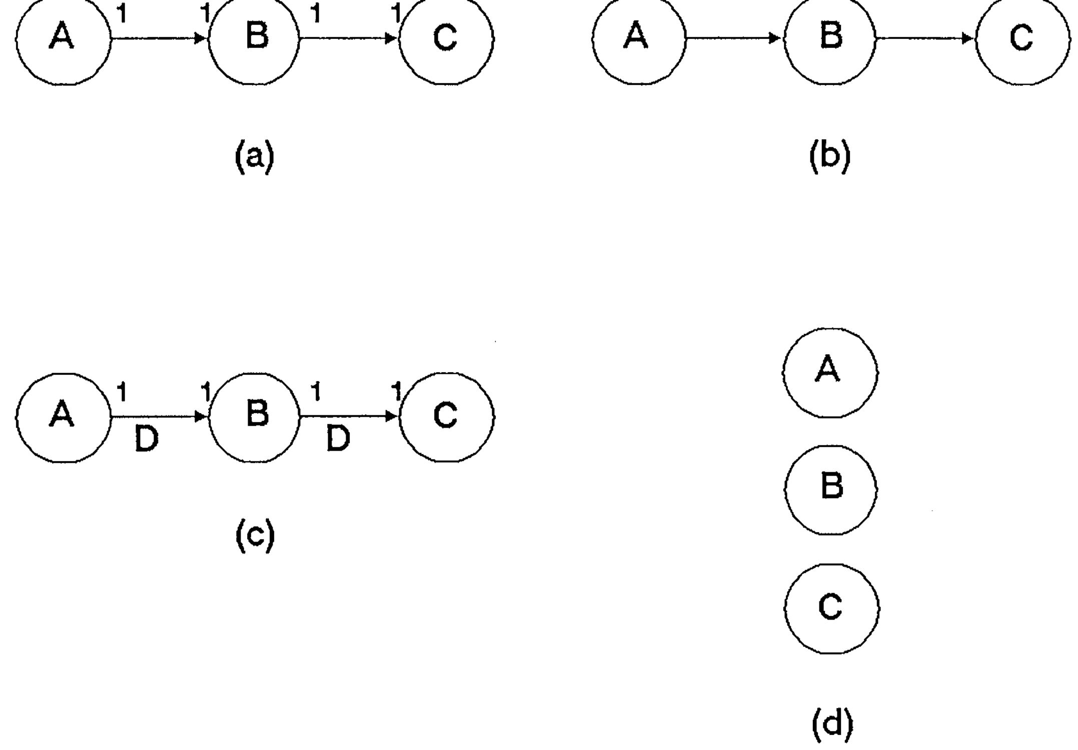


Figure 3-1:

(a) A synchronous data flow graph without directed loops.

(b) It's precedence graph for unity blocking factor.

(c) A pipelined synchronous data flow graph.

(d) It's precedence graph for unity blocking factor, showing much more concurrency.

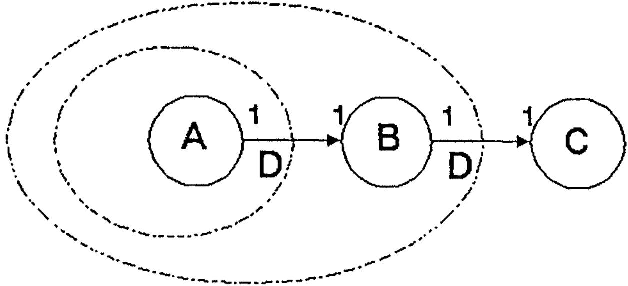


Figure 3-2: The SDF graph of figure 3-1 with the curves corresponding to the cutsets on which we put the delays.

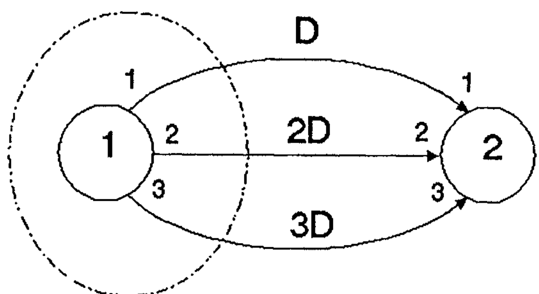


Figure 3-3: A multiple-sample-rate SDF graph with the proper delays on the feedforward cutset.

figure 3-3. If we put a unit delay on each arc in the cutset, we would not achieve pipelining (because node 2 cannot be invoked until node 1 is invoked once) and we would probably alter the computation. The proper number of delays is illustrated in the figure. To systematically determine the number of delays appropriate to pipeline a graph across a feed-forward cutset we need the notion of *consolidating* a subgraph.

Given a cutset, we wish to reduce the sub-graphs on either side of a cutset to single nodes, as shown in figure 3-4. The graph in figure 3-4(b) is said to be consolidated. We can determine the numbers a_1 through a_4 and b_1 through b_4 as follows. Define Γ_A to be the topology matrix for the subgraph inside the cutset curve and Γ_B to be the topology matrix for the subgraph outside the curve. These subgraphs do not include the arcs in the cutset. Compute $\mathbf{q}_A$ and $\mathbf{q}_B$, the smallest positive integer vectors in the nullspaces of the respective topology matrices. For the example shown in figure 3-4(a), we can determine $\mathbf{q}_A$ and $\mathbf{q}_B$ by inspection,

$$\mathbf{q}_A = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 2 \end{bmatrix} \qquad \mathbf{q}_B = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}.$$

Invoking the *subgraph* once means invoking each node in the subgraph the number of times specified by $\mathbf{q}_A$ or $\mathbf{q}_B$. To determine the number of samples produced on each arc out of the consolidated subgraph on each invocation, simply multiply the number of samples produced on that arc in the unconsolidated graph by the entry in $\mathbf{q}_A$ corresponding to the node connected to that arc. The figure shows the values of a_1 through a_4 and b_1 through b_4 computed in this

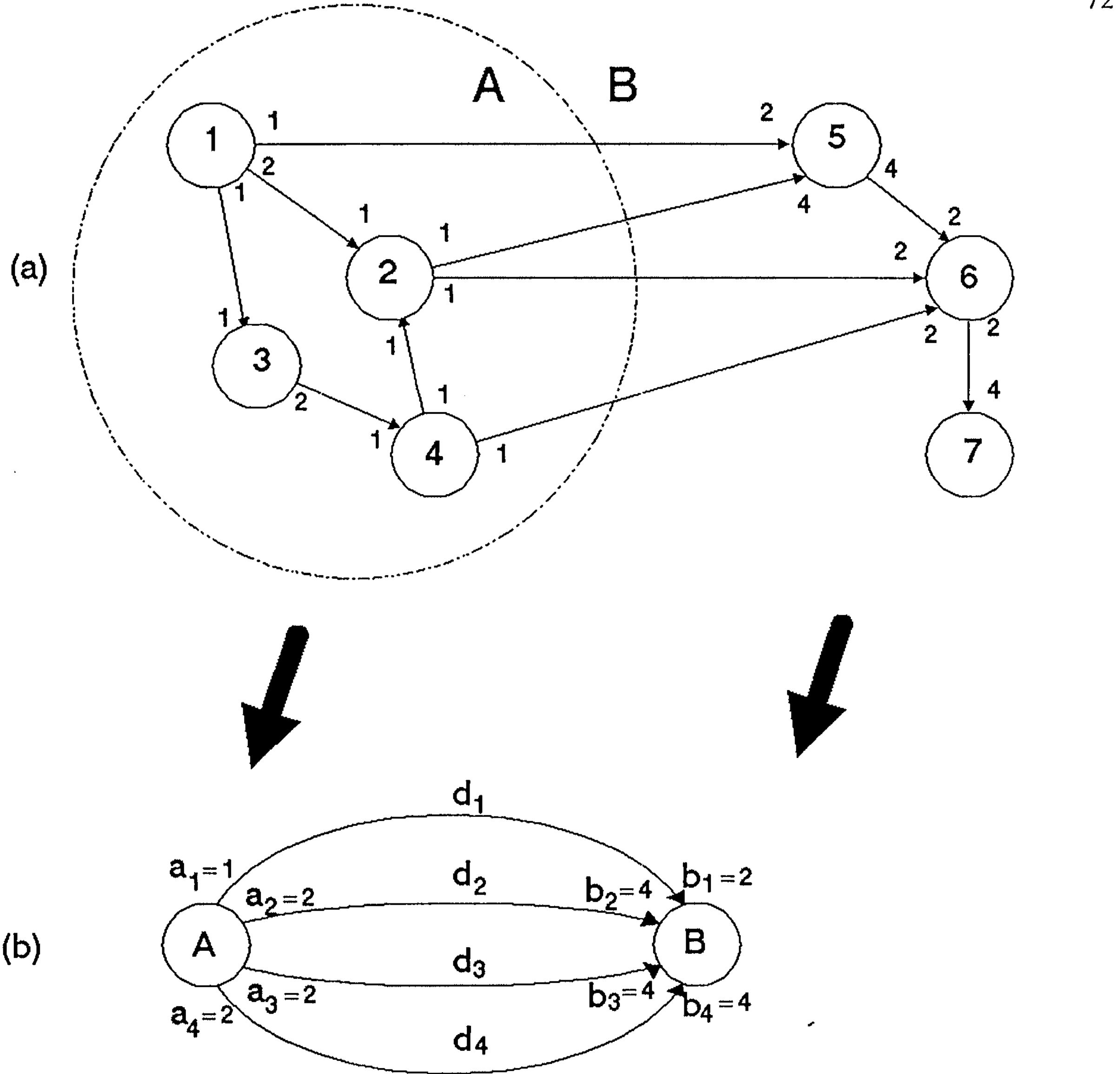


Figure 3-4: The nodes on either side of the cutset in (a) are consolidated in (b).

way.

Pipelining across a feed-forward cutset will be defined as adding delays to the cutset such that the consolidated destination node can be immediately invoked an integer number of times, and after such invocation, the buffers corresponding to the cutset arcs will have the same number of samples that they had before pipelining. Define the vectors $\mathbf{a}$ and $\mathbf{b}$ to contain the number of samples consumed and produced on the cutset, so that for the example in the figure,

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 2 \\ 4 \\ 4 \\ 4 \end{bmatrix}.$$

Also, define the vector $\mathbf{d}$ to contain the delays on the arcs in the cutset. Now to pipeline across the cutset, we can let

$$\mathbf{d} = n\,\mathbf{b}$$

for any positive integer n. In chapter 5, pipelining the voiceband data modem example results in a considerable improvement in the achievable throughput with a blocked schedule.

The notion of altering the delays on cutsets in a SDF graph can be generalized. To show how this might be useful, consider the example in figure 3-5(a), which has a precedence graph for $J=1$ shown in figure 3-5(b). Assuming all the run times are unity, for unit blocking factor, the critical path of the precedence graph is three, so the schedule period cannot be less than three. However, if the delays are rearranged as shown in figure 3-5(c), getting the precedence graph in 3-6(d), the critical path length is only two, so a schedule period of two is achievable with unity blocking factor. This technique is called *retiming* and has

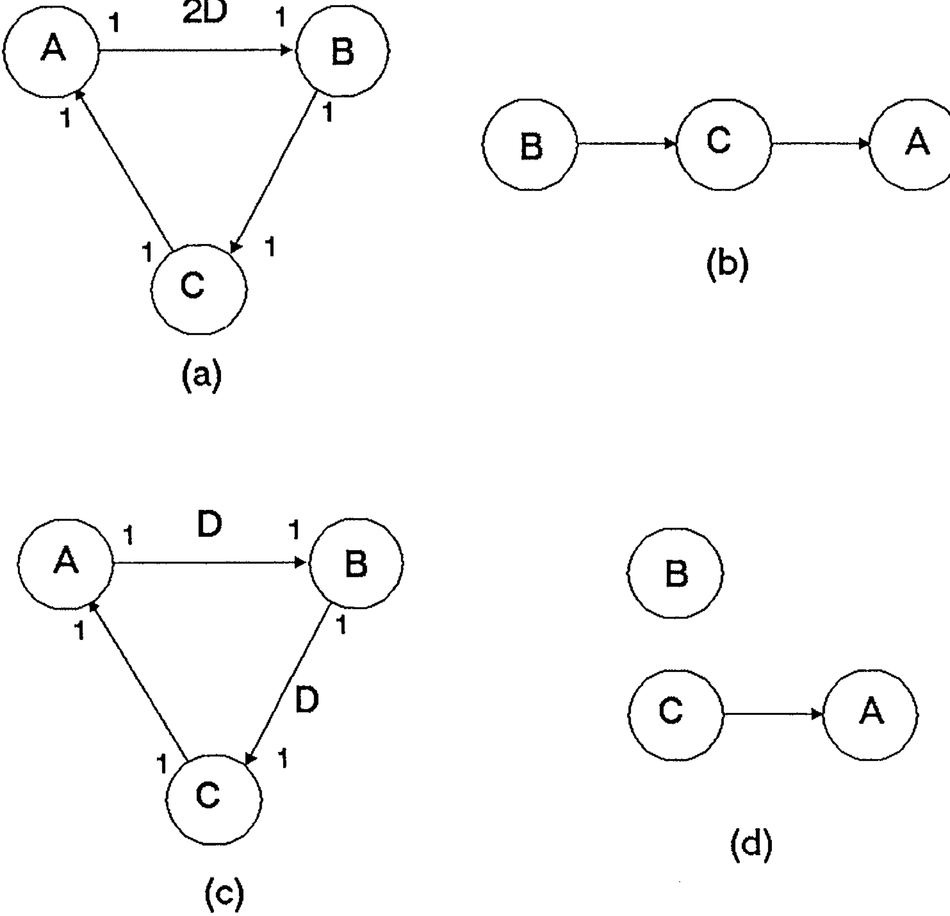


Figure 3-5: The SDF graph of (a) has precedence graph (b) for $J = 1$, indicating that its critical path includes all three nodes. If one of the delays is migrated across B, as shown in (c), then the precedence graph (d) shows a shorter critical path.

been successfully applied to maximizing the throughput of clocked digital circuits[Leis83a].

The transformation from figure 3-5(a) to 3-5(c) can be viewed as migrating one of the delays across node B. This is equivalent to running node B once before we begin the periodic schedule, clearly an admissible operation. Alternatively, the delay migration can be viewed as a cutset transformation, as done by Kung for single-sample-rate SDF graphs[Kung84a]. The cutset is shown in figure 3-6. The cutset transformation given by Kung is to put a fixed delay on any arc entering the cutset region and put an identical *negative* delay to any arc leaving the cutset region. In other words, positive delays can be put on arcs going one direction in the cutset if negative delays are put on arcs going the other way. Negative delays are like z operators in signal processing, as opposed to z^{-1}. Negative delays don't usually make much physical sense for infinite periodic computations, but can be used to offset other positive delays on the same arc.

Kung's transformation is easily extended to multiple sample rate SDF graphs. Given any cutset in a general SDF graph, the graph may be consolidated as shown in figure 3-4. Define the vectors

$$\mathbf{a} = \begin{bmatrix} a_1 \\ \cdots \\ a_N \\ -a_{N+1} \\ \cdots \\ -a_{N+M} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} -b_1 \\ \cdots \\ -b_N \\ b_{N+1} \\ \cdots \\ b_{N+M} \end{bmatrix}.$$

These vectors contain the information about the number of samples consumed and produced on the cutset arcs by each invocation of the consolidated nodes A

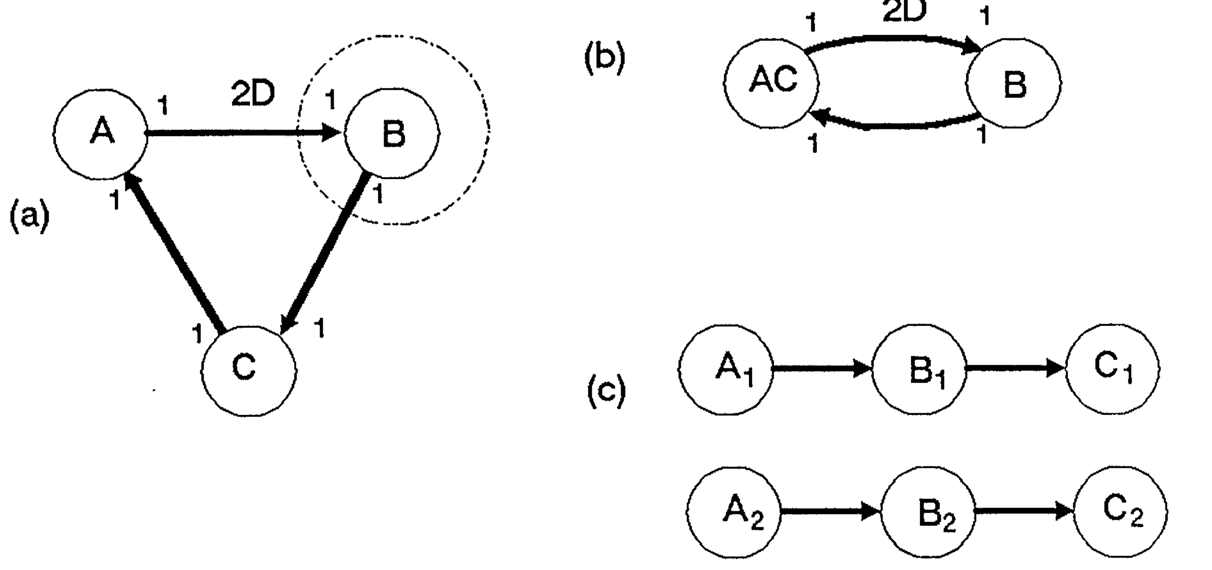


Figure 3-6:

(a) Transformations on the illustrated cutset can be used to improve the schedule for fixed blocking factor *J*. This is called *retiming*.

(b) The consolidated graph.

(c) Acyclic precedence graph with a blocking factor of two. Note that retiming will be less useful as the blocking factor is increased.

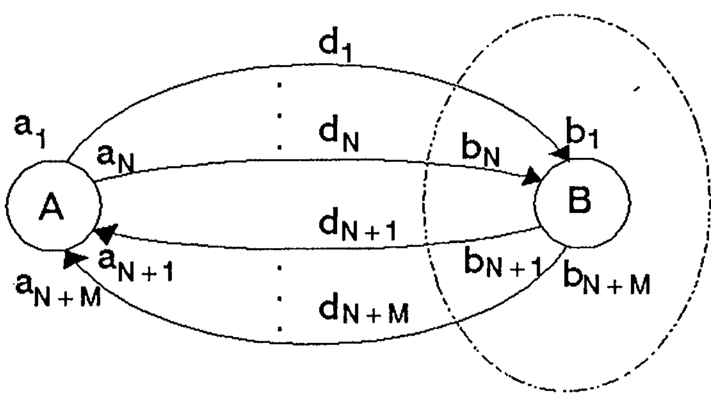


Figure 3-7: Cutset transformations are sometimes possible on cutsets with arcs going both ways. The nodes on either side of this cutset are shown consolidated.

and B. Positive numbers indicate samples produced while negative samples indicate samples consumed. Now define the cutset delay vector to be

$$\mathbf{d} = \begin{bmatrix} d_1 \\ \cdots \\ d_N \\ d_{N+1} \\ \cdots \\ d_{N+M} \end{bmatrix}.$$

As before we can put delays on the cutset arcs that are any integer multiple n (positive or negative) of the **b** vector

$$\mathbf{d} = n\,\mathbf{b}.$$

If n is positive, the delays going from right to left are negative. These delays should be added to any delays already on the arcs. The graph is essentially symmetric, so we could also put delays on the cutset arc that are any integer multiple of the **a** vector.

Returning now to the example in figures 3-5 and 3-6, we see that

$$\mathbf{a} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -1 \\ 1 \end{bmatrix},$$

and the pre-existing delay can be written

$$\mathbf{d}_0 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

To get the graph of figure 3-5(c), let $n=1$ to get the new delay vector

$$\mathbf{d}_1 = \mathbf{d}_0 + n\,\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

In general, cutset transformations are less useful as the blocking factor increases. The acyclic precedence with a blocking factor of two for the example in figure 3-5(a) is shown in figure 3-6(c). Notice that the critical path is the

same as with unity blocking factor. Suppose that the run time of each node is unity. Then with a blocking factor of two, the critical path has length three, so a schedule can be constructed that executes each node twice in every cycle of length three. This outperforms any blocked schedule using the retimed graph of figure 3-5(c) with unity blocking factor, which requires a period of two to execute each node once. We will see in the next section that the throughput that we can get with a blocking factor of two cannot be exceeded by any schedule. Furthermore, in the limit, with infinite blocking factor, retiming is useless.

Retiming is a useful technique when we are constrained to use a fixed blocking factor, or we wish to minimize the cost of implementing a schedule by minimizing the blocking factor. To determine how well we can expect to do, we need to understand the limits of the performance we should expect from a schedule for a given SDF graph.

3.2. THE BOUND ON THE COMPUTATION RATE

Recall that $J\mathbf{q}$, the blocking factor multiplied by the smallest positive integer vector in the nullspace of the topology matrix, tells us how many times we are to schedule each node in one cycle of a periodic schedule. Define the *schedule period* $S_J(\phi)$ to be the amount of time a particular schedule ϕ (with blocking factor J) requires for one cycle. Assume as in the previous chapter that the schedule is strictly *blocked*, meaning that each cycle must be completed before the next cycle is begun. The schedule period is not a good measure of the performance of a schedule because it increases with the blocking factor J but does not reflect the fact that more work may be done per cycle.

The vector q contains the number of times each node gets invoked in a schedule with unity blocking factor, and is therefore a measure of the minimum amount of work done in one cycle. With a larger blocking factor J, the amount of work done is always an integer multiple J of this. We define the *iteration period* $T_J(\phi)$ for the schedule ϕ to be the normalized schedule period

$$T_J(\phi) = \frac{S_J(\phi)}{J}.$$

This is a better measure of the performance of the schedule because two schedules with different blocking factors can now be compared; the one with a smaller iteration period has a greater computation rate. The *computation rate* of a schedule ϕ is defined to be $1/\ T_J(\phi)$. The iteration period is the average amount of time taken to invoke each node the number of times given in the vector q. Define the minimum iteration period over all periodic admissible parallel blocked schedules ϕ with blocking factor J to be

$$T_J = \min_{\phi} T_J(\phi).$$

The minimum iteration period is the reciprocal of the maximum achievable computation rate for a fixed blocking factor. It is equal to the computation time of the critical path in the acyclic precedence graph divided by the blocking factor (if a processor is dedicated to each node in the precedence graph, then then the computation time for one iteration will equal the computation time in the critical path, and the computation time cannot be reduced further). The *iteration bound* T_∞ is the minimum T_J over the extended positive integers J (i.e. $1 \leqslant J \leqslant \infty$),

$$T_\infty = \min_J T_J = \min_J \min_\phi \frac{S_J(\phi)}{J}.$$

Although the iteration bound is defined here in terms of blocked schedules, it is

clear that no unblocked schedule can undercut the iteration bound, because with $J = \infty$, any unblocked schedule is also a blocked schedule. Hence the iteration bound is the reciprocal of an absolute upper bound on the computation rate. For practical scheduling, in section 3.4 we discuss trying to find a finite value for J such that $T_J = T_\infty$. It turns out that such a J does not always exist.

One might conjecture that the minimum iteration period T_J is non-increasing with J, but this is not so. Consider the example in figures 2-15 through 2-17. The methods of the previous chapter can be used to construct an acyclic precedence graph for each J, and the critical path length of the acyclic precedence graph can be identified. The results are given in the following table.

J	Critical path	T_J
1	4	4.0
2	7	3.5
3	11	3.667
4	14	3.5
5	18	3.6
6	21	3.5

We will see that the iteration bound is 3.5, which can be met with blocked schedules with any even blocking factor, but cannot be met with any finite odd blocking factor. Clearly, the non-increasing conjecture is not true.

The iteration bound is easily determined for SDF graphs where the number of samples produced or consumed is unity for all arcs. We will call such a graph a *homogeneous* SDF graph. Homogeneous SDF graphs obviously have the same sample rate throughout (but not all single-sample-rate SDF graphs are homogeneous, see figure 3-8). Also, the vector $\mathbf{q} = [1, \cdots, 1]^T$ is in the nullspace of the topology matrix, which is populated only with ± 1, so a

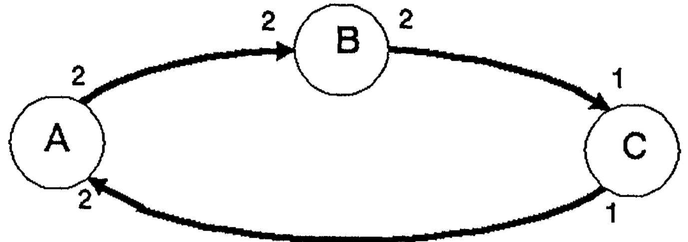


Figure 3-8: A single-sample-rate SDF graph that is not homogeneous.

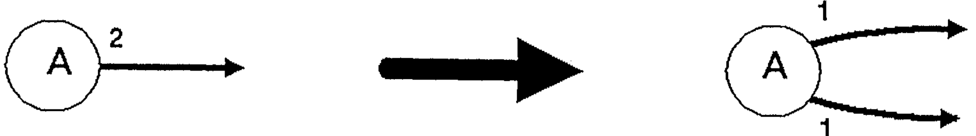


Figure 3-9: A two-sample output can be replaced with two one-sample outputs to transform a general SDF graph into a homogeneous SDF graph.

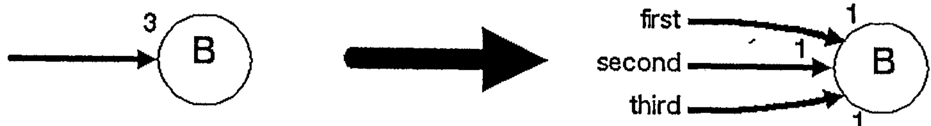


Figure 3-10: A three-sample input can be replaced by three one-sample inputs to transform a general SDF graph into a homogeneous SDF graph.

schedule with unity blocking factor will run each node once per period. For homogeneous SDF graphs, Renfors and Neuvo give a tight lower bound on the iteration period, which equals the iteration bound because it is tight[Renf81a, Fett76a, Schw85a]. We review this bound in this section. The iteration bound for multiple sample rate SDF graphs will be discussed in section 3.3.

Given any feedforward cutset in the graph, the iteration bound is determined by the maximum of the iteration bounds of the right or left subgraphs. This follows from our ability to decouple the right or left subgraphs using pipelining, as discussed in the previous section. As a consequence, we need only consider subgraphs with no feedforward cutsets, and determine the maximum iteration bound of all such subgraphs. Every node in a subgraph with no feedforward cutsets is in a directed loop, so we need only to consider directed loops.

Any implementation of a homogeneous SDF graph has a *sample period* T, which can be defined as the average amount of time between samples on any signal path. The sample period T equals the iteration period because each node is run once per iteration period and consumes or produces one sample on each signal path.

Call the set of directed loops in a graph Λ and consider a loop $\lambda \in \Lambda$. If all the nodes in the loop have zero execution time, then each logical delay in the loop corresponds to a time delay of T. Since the nodes in the loop generally have non-zero execution times, the actual time delay corresponding to each logical delay is less than T. Renfors and Neuvo call this actual time delay a *shimming delay*. It can be understood as the time a sample is delayed when it

encounters a logical delay, for a given implementation. It is obvious that all shimming delays must be non-negative. Call the total computation time in a loop C_λ, the number of logical delays $N_\lambda > 0$, and the total shimming delay S_λ. Then

$$C_\lambda + S_\lambda = N_\lambda T.$$

Since $S_\lambda \geqslant 0$,

$$T \geqslant \frac{C_\lambda}{N_\lambda}.$$

This implies that the sample period must be greater than or equal to the computation time of each loop divided by the number of logical delays in the loop. The Renfors and Neuvo bound on the sample period is therefore

$$T \geqslant T_0 = \max_{\lambda \in \Lambda} \frac{C_\lambda}{N_\lambda}. \tag{3.1}$$

Any loop $\lambda \in \Lambda$ such that

$$T_0 = \frac{C_\lambda}{N_\lambda}$$

is called a *critical loop*.

The iteration bound T_∞ is the best *achievable* iteration period over all blocking factors, so to keep the shimming delays positive, a realization has a sample period satisfying

$$T \geqslant T_\infty \geqslant T_0 = \max_{\lambda \in \Lambda} \frac{C_\lambda}{N_\lambda}.$$

To show that $T_\infty = T_0$ we simply need to show that

$$T = \max_{\lambda \in \Lambda} \frac{C_\lambda}{N_\lambda}$$

is achievable. Schwartz proves this for *reachable* graphs by elaborating and elucidating the proof by Renfors and Neuvo[Renf81a, Schw85a]. A reachable graph

is defined to be a SDF graph with a directed path from a unique "input" node to all other nodes in the graph. Any connected homogeneous SDF graph is easily converted into a reachable graph by defining a new node with zero execution time and establishing an arc from that node to any node that is otherwise not reachable. Schwartz's proof is by construction of an implementation that achieves the iteration bound, but since at least one processor is required for each node in the graph, there is no intention that the constructed implementation be efficient.

In conclusion, the Renfors and Neuvo lower bound T_0 on the sample period T for a realization of a homogeneous SDF graph is the maximum (over all directed loops) of the loop computation time divided by number of delays in the loop. This bound is equal to the iteration bound T_∞ , the best *achievable* iteration period. Determining this bound for general SDF graphs is not so easy. In the next section we show that general SDF graphs can be systematically transformed into homogeneous SDF graphs.

3.3. TRANSFORMING GENERAL SDF GRAPHS INTO HOMOGENEOUS SDF GRAPHS

In the previous section we reproduced a bound on the iteration period for homogeneous SDF graphs, which are single-sample-rate SDF graphs where the number of samples produced and consumed by each node is unity. The iteration bound is the reciprocal of the best (maximum) achievable computation rate. To find the bound for general SDF graphs, we can convert the graphs into equivalent homogeneous SDF graphs. By "equivalent" we mean that any schedule that is admissible for the general SDF graph will also be admissible for

the homogeneous graph, and vice-versa. Every node producing more than one sample on an output arc for each invocation can be replaced with a node that produces one sample each on several parallel output arcs, as shown in figure 3-9. We do the same with inputs, as shown in figure 3-10. These transformations suggest that multiple samples are passed from one node to another in parallel, but we generally need to preserve the ordering of samples. For example, complex numbers might be passed as a sequence of two numbers, the real part followed by the imaginary part, so the order is important. We adopt the convention that for any group of input arcs such as those in figure 3-10, the samples are ordered top to bottom, as shown.

Using these transformations, the nodes in figure 3-11(a) may be connected as shown in figure 3-11(b). If there is a unit delay on the arc, the connection is a little more complicated, as shown in figure 3-12. The first sample consumed by B is the sample in the buffer initially put there to implement the delay. The second sample consumed by B is the first sample produced by A. In general, the rule for handling delays is as follows.

(1) Each delay on an arc that is being replaced by parallel arcs causes a *circular permutation* of the parallel arcs. A single circular permutation of the set of parallel arcs in figure 3-13(a) means that the bottom arc crosses over the other arcs to become the top arc, as shown in figure 3-13(b). A double circular permutation is shown in figure 3-13(c).

(2) Each delay on an arc that is being replaced with parallel arcs becomes a single delay on the permuted parallel arcs, starting at the top. The single delay in figure 3-14(a) becomes the single delay in figure 3-14(b), and the

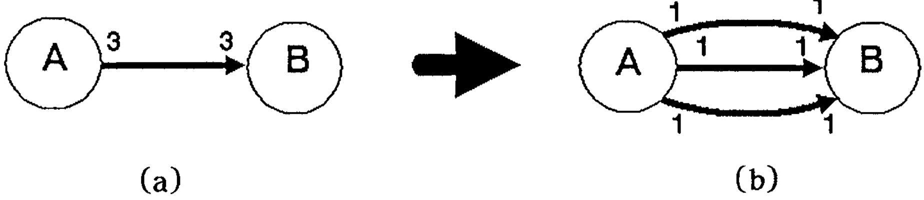


Figure 3-11: An arc with the same number of samples consumed and produced is easily transformed into parallel homogeneous arcs.

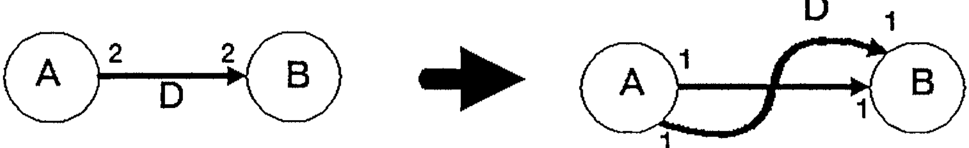
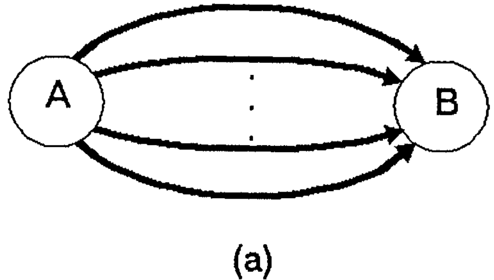
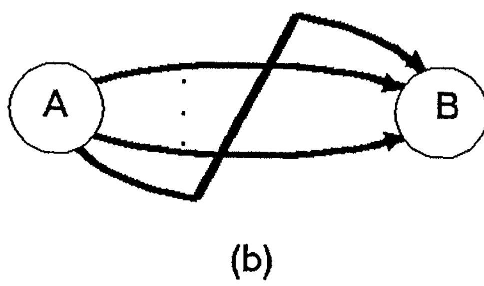


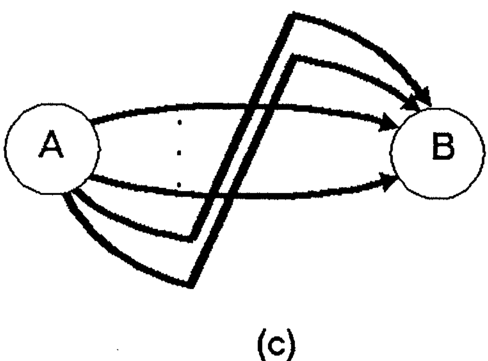
Figure 3-12: A delay on an arc causes a circular permutation of the parallel homogeneous arcs, as shown.



(a)



(b)



(c)

Figure 3-13:
(a) Unpermuted parallel homogeneous arcs.
(b) A single circular permutation.
(c) A double circular permutation.

double delay in 3-14(c) becomes the two single delays in figure 3-14(d). Obviously, if the number of delays equals the number of parallel arcs, as shown in figure 3-14(e), then the arcs are not permuted (equivalent to permuting N times) and each arc gets a single delay. If there are $M > N$ delays, then put a delay on each arc and apply this procedure assuming $M \ mod \ N$ delays.

This general procedure will preserve the ordering of samples for the special case when the nodes on either side of an arc produce and consume the same number of samples. But what do we do when this is not so? Consider the example in figure 3-15(a). This can be converted to the homogeneous SDF graph shown in figure 3-15(b). Here, A_i means the ith invocation of A. To do this systematically, observe that for the graph in figure 3-15(a),

$$q = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

is the smallest positive integer vector in the nullspace of the topology matrix. Thus, for unit blocking factor, node A must be invoked three times and node B twice. These invocations become separate nodes in the homogeneous SDF graph. The "equivalence" of the two graphs is guaranteed, however, in that any schedule that works for the general SDF graph will also work for the homogeneous SDF graph, and vice-versa.

If there are self-loops, as shown in figure 3-15(c), then the multiple invocations of each node are chained together and put into one big loop as shown in figure 3-15(d).

Each set of replicated nodes can now be consolidated, as shown in figure 3-4, and the delay operations noted above applied to the parallel arcs connecting

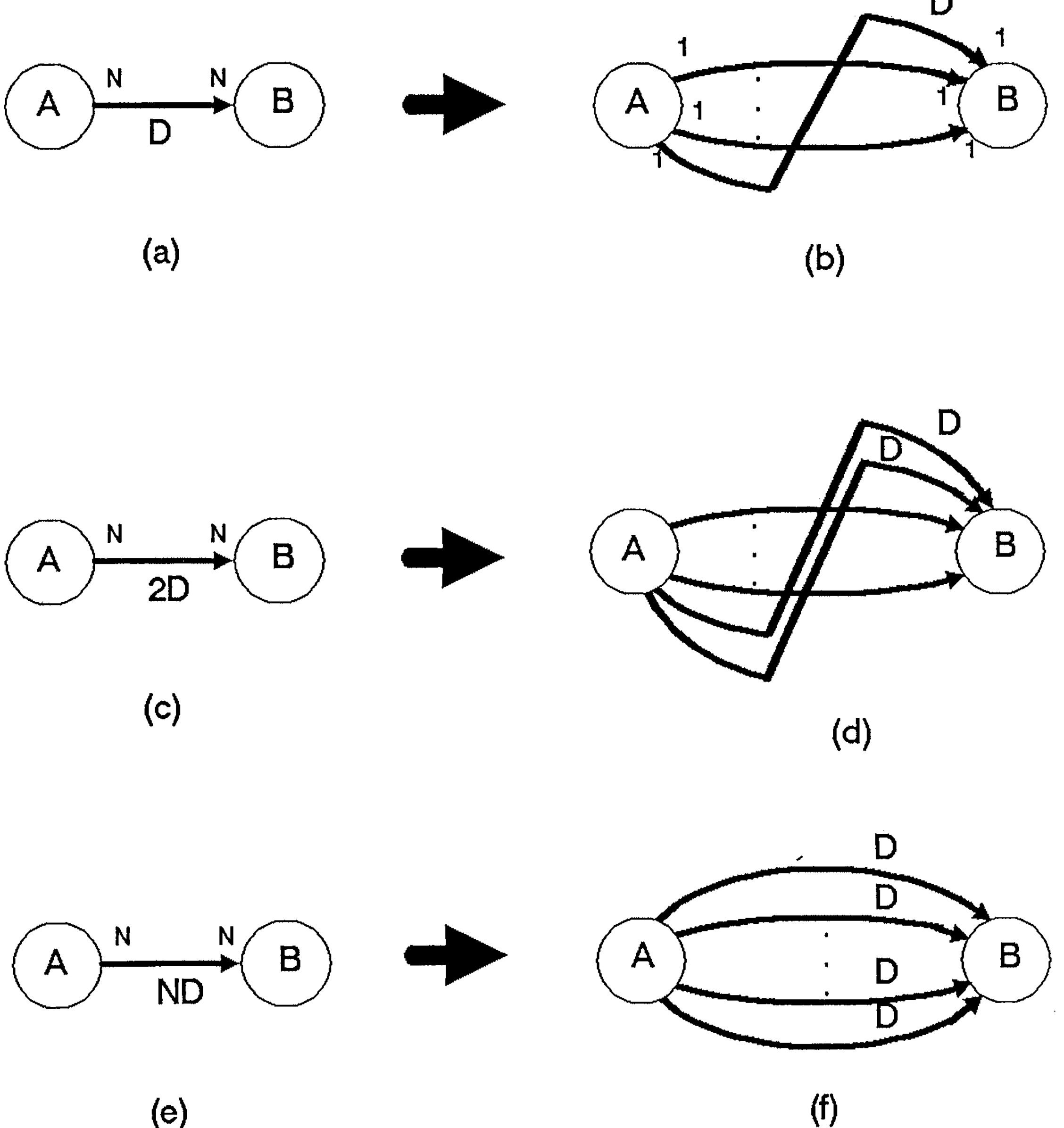


Figure 3-14: SDF graphs with varying numbers of delays (a,c,e) and homogeneous SDF graphs with the delays properly placed (b,d,f). The ones are omitted from (d) and (f) for clarity.

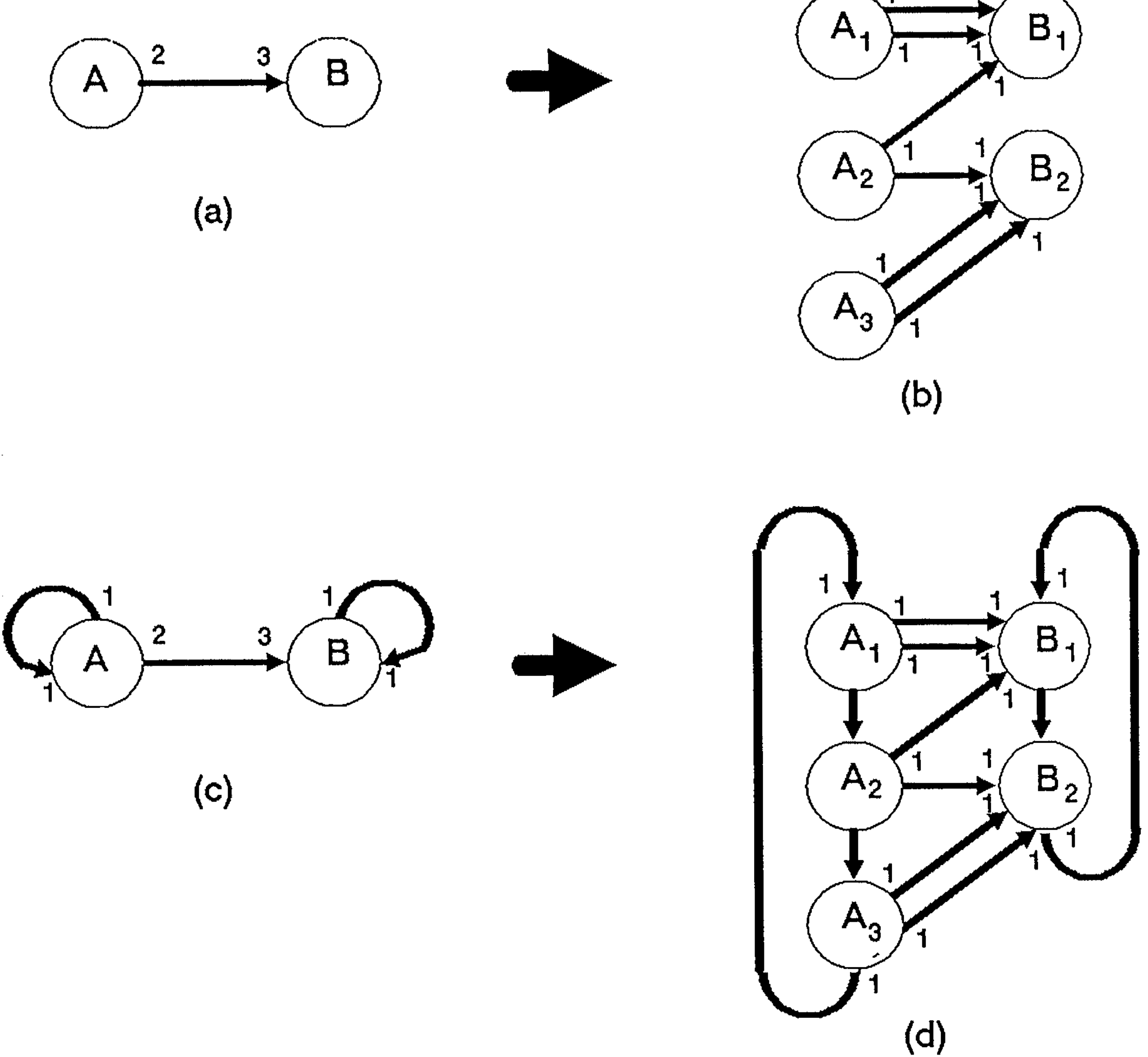


Figure 3-15: Two transformations of multiple-sample-rate SDF graphs into homogeneous SDF graphs using parallel data flow paths. The transformation from (c) to (d) shows how self-loops are handled.

the two consolidated nodes.

Let us illustrate the procedure by computing the iteration bound for the multiple sample rate SDF graph of figure 3-16(a). Notice that the logical delay on the path $A \rightarrow B$ does not correspond to the same time delay as the logical delay on the path $B \rightarrow C$, because the sample rates on these two paths are different. To transform this graph into a homogeneous SDF graph, solve for the smallest integer vector in the nullspace of the topology matrix,

$$\mathbf{q} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix},$$

where we have assumed that the nodes are numbered in alphabetical order. Replicate node A twice and chain the replicated nodes as shown in figure 3-16(c). The other two nodes are not replicated. The delay on the path $A \rightarrow B$ causes the circular permutation as shown. The two arcs on $C \rightarrow A$ are not permuted because there is no delay. Knowing the execution times of each of the nodes, we can now compute the iteration bound for the homogeneous SDF graph of figure 3-16(c). If the computation times of all the nodes are unity, then the bold path in the figure is the critical loop. It has a computation time of $C_\lambda = 3$ and only one delay, so the iteration bound is three. One cycle of a blocked schedule with unit blocking factor that achieves this bound with two processors is shown in figure 3-16(d).

Now that we have a method for computing the iteration bound of a general SDF graph, we wish to determine what blocking factor, if it is bounded, we should use to construct a blocked schedule that matches the bound.

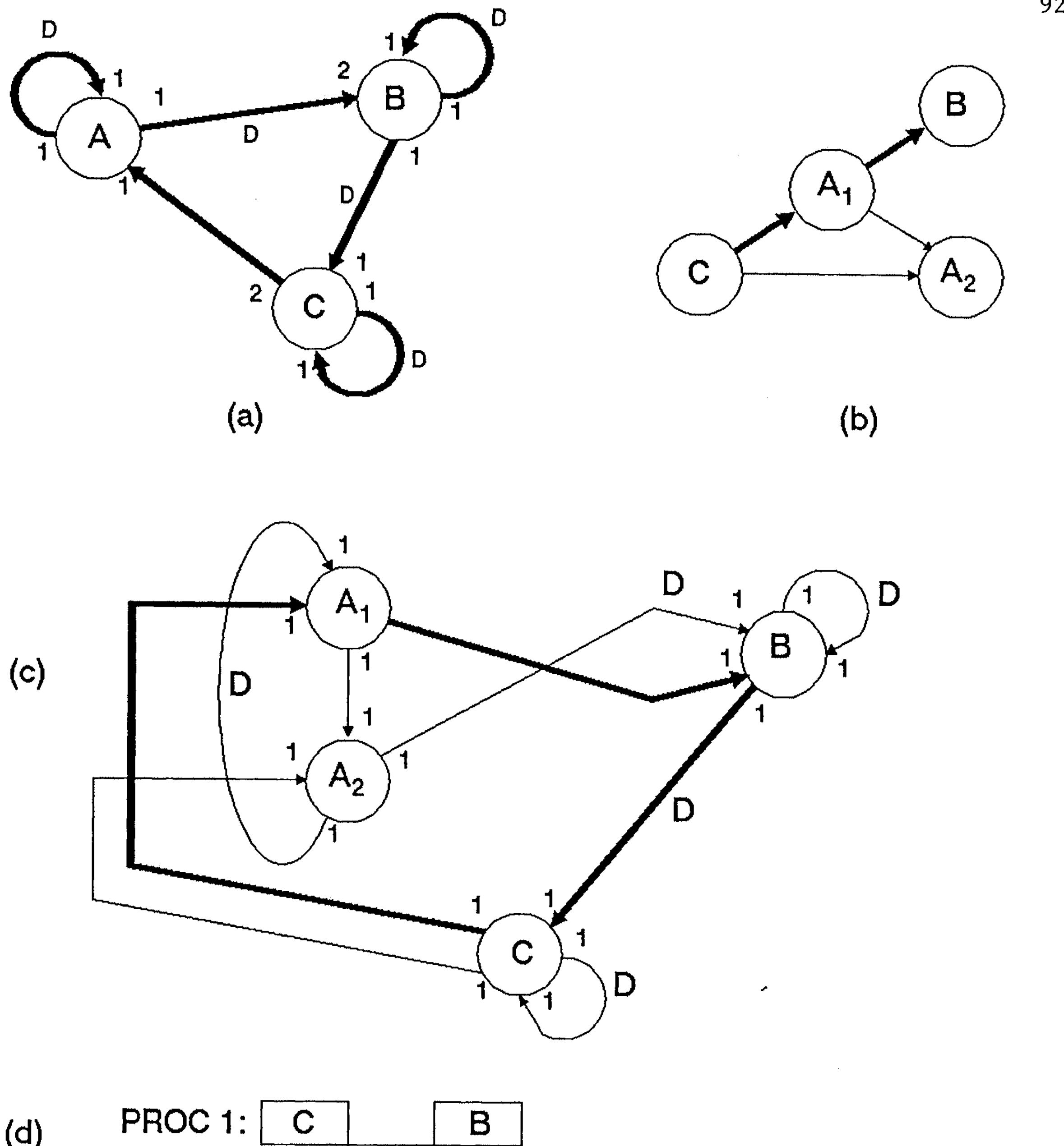


Figure 3-16:

(a) A multiple-sample-rate SDF graph.

(b) An acyclic precedence graph for unity blocking factor.

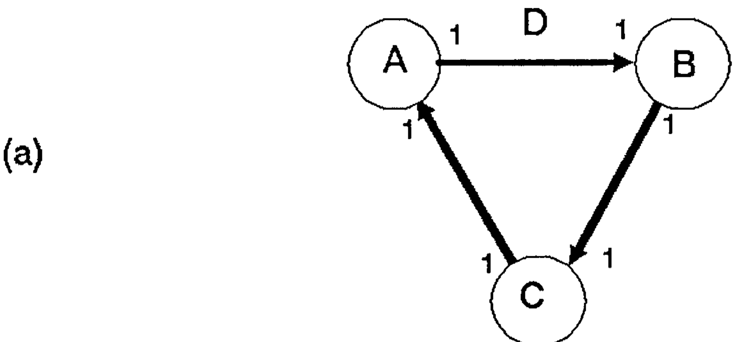
(c) An equivalent homogeneous SDF graph.

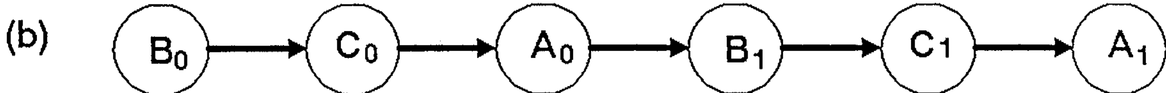
(d) One period of a blocked schedule with unity blocking factor that achieves the iteration bound of three.

3.4. THE OPTIMAL BLOCKING FACTOR

Unfortunately, the problem of systematically finding the optimal blocking factor is still open. An optimality condition was conjectured by Schwartz[Schw85a] (page 46), but unfortunately it is only a necessary and not sufficient condition for being able to construct a blocked schedule that achieves the iteration bound. We will show instead that the blocking factor that achieves the iteration bound does not always exist, and suggest heuristic solutions. When the optimal blocking factor does not exist, no blocked schedule can meet the iteration bound. Interestingly, the modem example described in chapter 5 does not benefit at all from a blocking factor greater than unity.

To understand Schwartz's conjecture on the optimality condition for the blocking factor, it is necessary to understand the mechanics of Schwartz's blocking. Consider the homogeneous SDF graph in figure 3-17(a). Assume that the run time of each node is unity. Then $T_0 = 3$ (with or without implicit self-loops), where T_0 is the Renfors and Neuvo bound, equal to the iteration bound. If we wish to construct a schedule for the blocking factor $J=2$, then using the technique of the previous section we would construct the acyclic precedence graph shown in figure 3-17(b). Schwartz, instead, constructs an analogous SDF graph, called a *blocked graph*, with twice as many nodes, as shown in figure 3-17(c). Notice that if we break the arc with the delay, then we have exactly the precedence graph of 3-17(b). Not surprisingly, the systematic technique given by Schwartz for constructing the blocked graph is somewhat similar to the algorithm given above for constructing the acyclic precedence graph. The iteration bound for the graph in 3-17(c) is $T_0 = 6$, so Schwartz defines the *iteration*





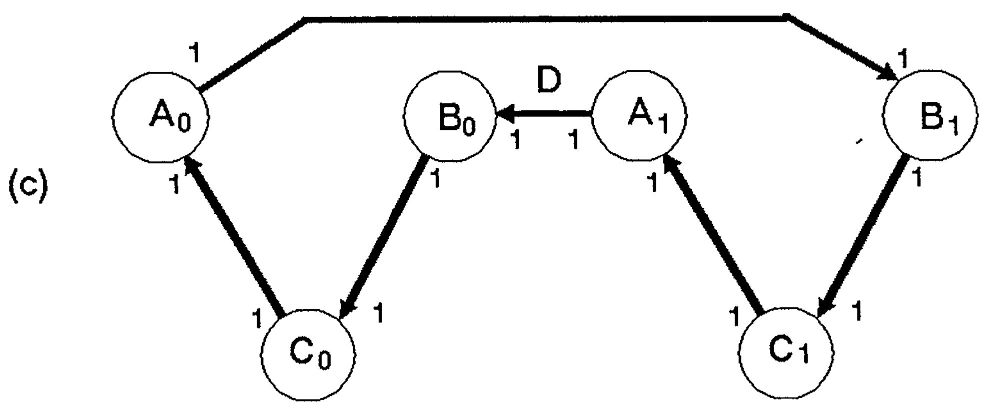


Figure 3-17:

(a) A single-sample-rate SDF graph.

(b) A precedence graph for $J = 2$.

(c) A homogeneous SDF graph representing the graph in (a) with a blocking factor $J = 2$ [Schw85].

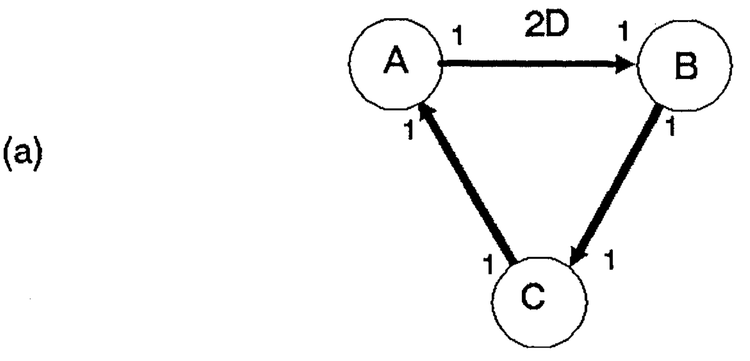
bound per output $T_{o/o}$ to be the iteration bound divided by the blocking factor. This terminology becomes awkward when multiple sample rates are considered, but works acceptably well for homogeneous graphs.

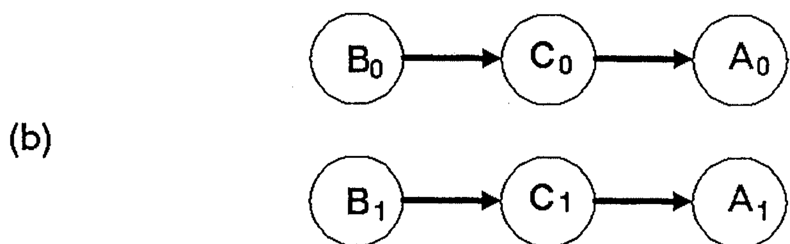
For the example in figure 3-17, there is no scheduling advantage to any blocking factor greater than unity, so although the example is useful to demonstrate Schwartz's blocked SDF graphs, it is not an interesting example. Consider instead the example in figure 3-18. With unity run times, the Renfors and Neuvo bound is $T_0 = 3/2$. With a blocking factor of unity, the minimal schedule period is $S_1 = 3$, so the minimal iteration period is $T_1 = 3$, which is short of the optimum by a factor of two. However, with a blocking factor of J=2, the acyclic precedence graph is shown in figure 3-18(b), the minimal schedule period is $S_2 = 3$, achievable for two or more processors, so the minimal iteration period equals the iteration bound

$$T_2 = T_0 = 3/2.$$

Recall that the length of the *critical path* in the acyclic precedence graph determines the minimum achievable schedule period S_J. Schwartz's blocked graph with a blocking factor of two is shown in figure 3-18(c). Just like the precedence graph 3-18(b), the blocked SDF graph shows two independent systems with three nodes each. Again, if the arcs with delays are broken, the blocked graph is identical to the precedence graph. The iteration bound is three and the iteration bound per output is 3/2.

According to Schwartz, the iteration bound can be met if the blocking factor J is large enough that the iteration bound for the blocked graph is an integer. This requires that the numerator in equation (3.1) be an integer multi-





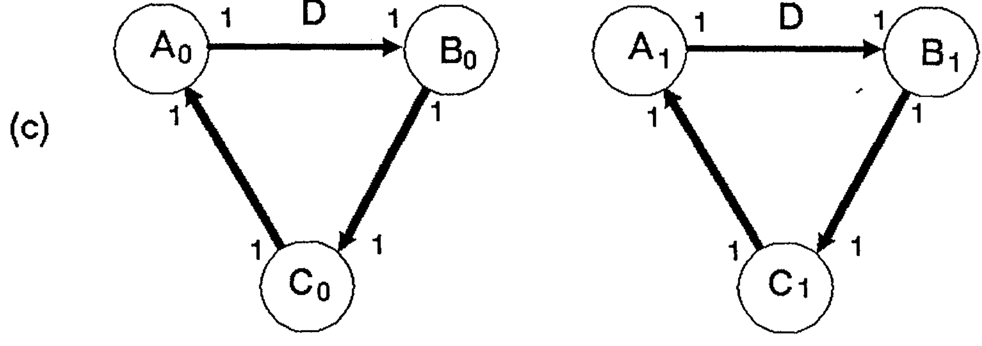


Figure 3-18:

(a) A homogeneous SDF graph.

(b) A precedence graph for $J = 2$.

(c) A homogeneous SDF graph representing the graph in (a) with a blocking factor $J = 2$ [Schw85].

ple of the denominator for the critical loop in the blocked graph. This blocking factor can always be found, and is always finite. However, for blocked schedules, in which we require that each period of the periodic schedule complete before the next period begin, this condition on J is only *necessary* and not sufficient. The counter-example to Schwartz's conjecture in figure 3-19(a) requires an infinite blocking factor to achieve the iteration bound. Assume unit run times. There are two loops in this graph, but the critical loop is the one including nodes 2 and 4. This loop sets the iteration bound at $T_0 = 2$. For the blocking factor of unity, the iteration bound of the graph is an integer, so Schwartz's conjecture indicates that unity is the optimal blocking factor. However, the iteration bound cannot be met by any blocked schedule for any finite blocking factor. The acyclic precedence graphs for $J=1$, $J=2$, and $J=3$ are shown in figure 3-20, with the critical path indicated with heavier lines. The length of the critical path increases by two each time J is increased, so the best achievable schedule period for a blocked schedule is

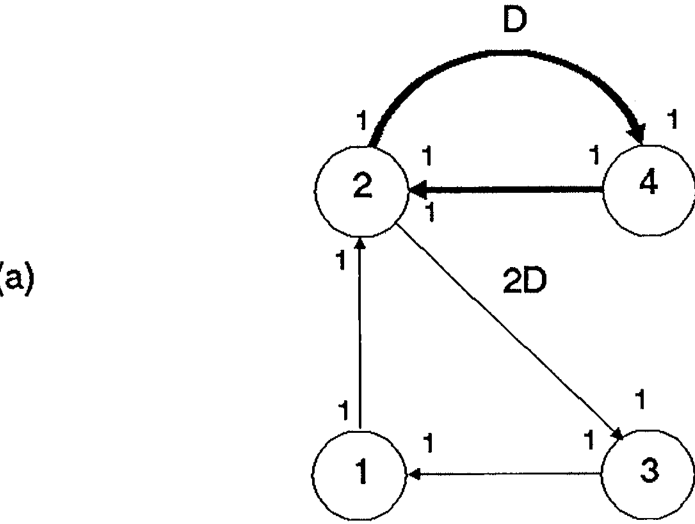
$$S_J = 1 + 2J.$$

The best iteration period is therefore

$$T_J = 2 + \frac{1}{J}.$$

This iteration period is always greater than the iteration bound $T_0 = 2$ for any finite J.

Schwartz proposes *cyclo static* schedules, rather than blocked schedules, to avoid the cost of blocking. For the example of 3-19(a), the cyclo-static schedule of figure 3-19(b) achieves the iteration bound. This suggests that using Schwartz's cyclo-static scheduling on the transformed homogeneous SDF graphs



(b)

		3	1	4	2	3	1	4	2	...
3	1	4	2	3	1	4	2			

Figure 3-19:

(a) A homogeneous SDF graph for which no blocked schedule achieves the iteration bound with finite blocking factor.

(b) A cyclo static schedule for this graph.

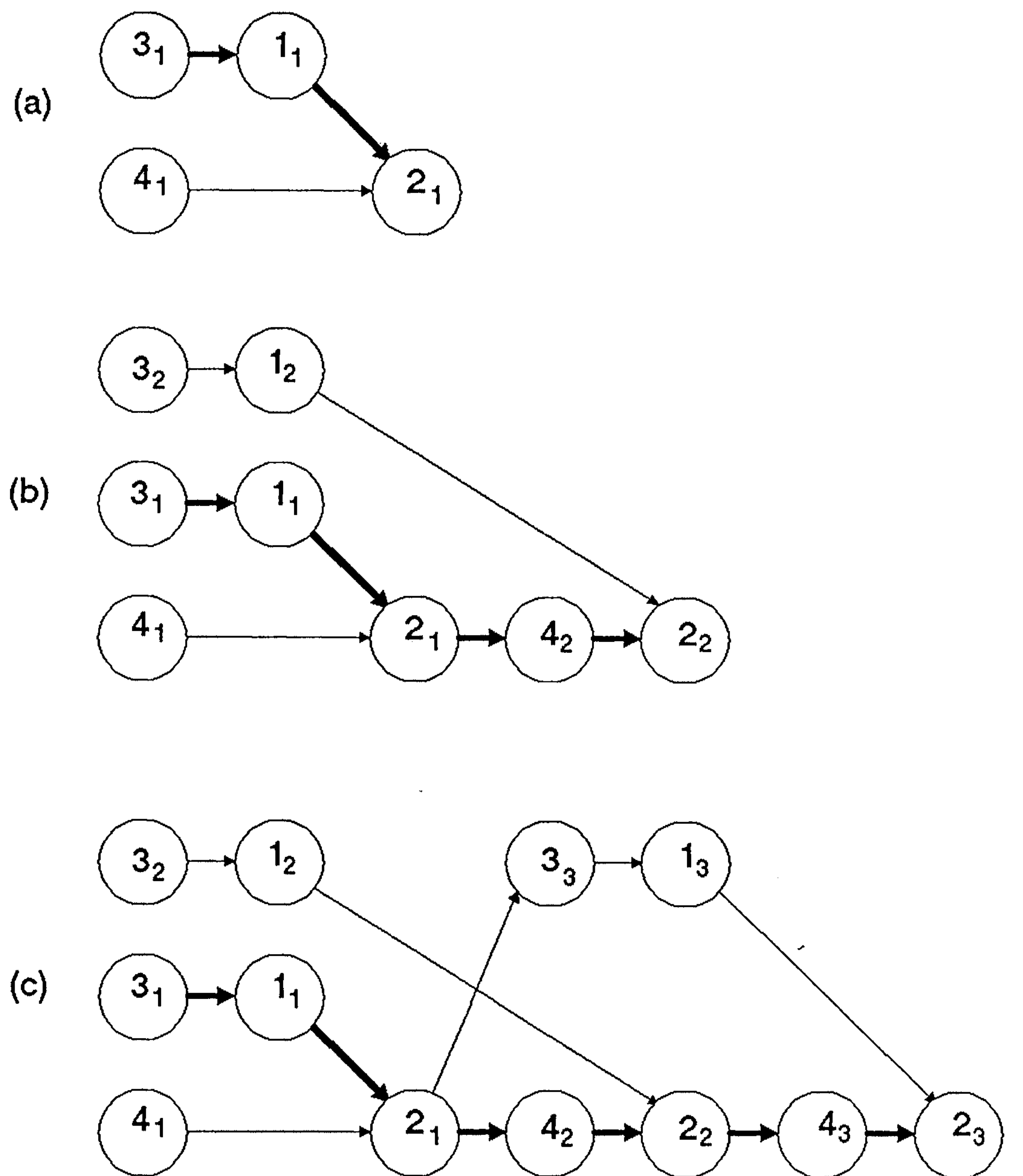


Figure 3-20: Acyclic precedence graphs for the SDF graph of figure 3-19 for (a) $J = 1$, (b) $J = 2$, and (c) $J = 3$. The critical paths are indicated with the bold branches.

may yield better schedules than the Hu level scheduling algorithm proposed in the previous chapter. However, the method proposed by Schwartz for finding the cyclo static schedule has complexity that is exponential in the number of nodes. For the problem of constructing a schedule that matches the iteration bound with the minimum number of processors the complexity is manageable, at least for simple examples, because of the optimality constraints. But our problem is to construct schedules that maximize the throughput subject to a constraint on the number of processors, and in this case, Schwartz's search method becomes intractable for all but the most trivial examples.

The conclusion, therefore, is that the technique conjectured by Schwartz for determining the optimal blocking factor does not work for blocked schedules, and for some graphs the optimal blocking factor is infinite. So no systematic technique is known for determining the optimal blocking factor. The theory has not even begun to tackle the problem of determining the optimal blocking factor when the number of processors is smaller than the number required to achieve the iteration bound.

Until a better solution is proposed, we rely on heuristics for determining J. A useful observation is that as J increases, the cost of implementing the periodic schedule increases because of the memory cost of storing the schedule. Furthermore, the time required to construct the schedule increases. One possible technique is to manually increase J until each increase results in negligible improvement in the schedule, or the scheduling requires excessive computer resources. This is clearly an issue deserving further study.

3.5. MINIMIZING MEMORY USAGE

To accommodate cost sensitive real-time signal processing applications, most monolithic programmable DSP chips are microcomputers, with on-board memory, rather than microprocessors. This makes single-chip implementations possible if the memory requirements of an application are sufficiently small. Conserving memory in an implementation is therefore desirable. This section describes two techniques for minimizing the memory used for buffering data between blocks. The first method is to construct the schedule such that the maximum memory usage is minimized. The second method is to share memory locations among buffers when said buffers do not simultaneously have valid data.

3.5.1. Scheduling to Minimize Memory Usage

So far, parallel schedules have been constructed to maximize throughput. Minimizing memory usage might imply a throughput penalty, because the set of maximum throughput schedules may be disjoint from the set of minimum memory schedules. For sequential schedules, however, the throughput is constant, independent of the schedule, so minimizing memory does not imply a throughput penalty. For this reason, the problem of scheduling to minimize memory usage is described here in the context of sequential schedules.

There are two reasonable memory usage criteria we may wish to optimize. The most straightforward is

Criterion A:

$$\min_{\phi} \; 1^T \left\{ \max_{0 \leqslant n \leqslant p} \; \mathbf{b}(n) \right\},$$

where ϕ is the schedule, $1^T = [1, \cdots, 1]$, p is the schedule period, and $\mathbf{b}(n)$ is the vector of buffer sizes at time n, and the max is computed for each element of the vector. That is,

$$\max \left\{ \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}.$$

Criterion A minimizes the sum of memory that must be allocated to each buffer if memory is not shared among buffers. Alternatively, we could minimize the total amount of memory dedicated to buffering at any given time;

Criterion B:

$$\min_{\phi} \; \max_{0 \leqslant n \leqslant p} \; 1^T \mathbf{b}(n).$$

To take advantage of a schedule constructed according to criterion B, we need a systematic method for sharing memory locations among buffers. Such a method is described in the next subsection.

The problem of constructing a schedule that minimizes memory under either criterion can be formulated as an integer programming problem, or as a dynamic programming problem. We will do the latter because it gives more insight. In general, the optimization is combinatorial in complexity, but for many SDF graphs, the complexity is often manageable because of the constraints on the problem. A heuristic with polynomial execution time would be desirable, but is not given in this thesis.

Define the vector $\mathbf{r}(n)$ to be of length s, the number of nodes in the SDF graph, and to contain the number of times each node has been invoked by time n. Thus, if $\mathbf{v}(n)$ is the execution vector, indicating which node is invoked at time n, then

$$\mathbf{r}(n) = \sum_{j=0}^{n-1} \mathbf{v}(j),$$

$$\mathbf{r}(0) = [0, \cdots, 0]^T = \mathbf{0},$$

and

$$\mathbf{r}(p) = \mathbf{q},$$

where p is the period of the schedule, and $\mathbf{q}$ is a positive integer vector in the nullspace of the topology matrix. We can now construct a trellis indicating all possible sequences of vectors $\mathbf{r}(n)$, as shown in figure 3-21. In the figure, there are three possible nodes that we can invoke at time 0, so $\mathbf{r}(1)$ has three possible values. The nodes that can be invoked at time 1 are dependent on the choice made at time 0, but all admissible paths converge after p stages on $\mathbf{r}(p) = \mathbf{q}$. Each path through the graph has a cost, measured as the amount of memory required under either criterion. The problem is to select the path with the least cost. This is easily recognized as a dynamic programming problem, because whenever two paths converge at the same point, the more costly path can be immediately rejected.

Consider constructing a sequential schedule with unity blocking factor for the example in figure 3-22. It is easy to see that

$$\mathbf{q} = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$$

is in the nullspace of the topology matrix. The full trellis is shown in figure 3-

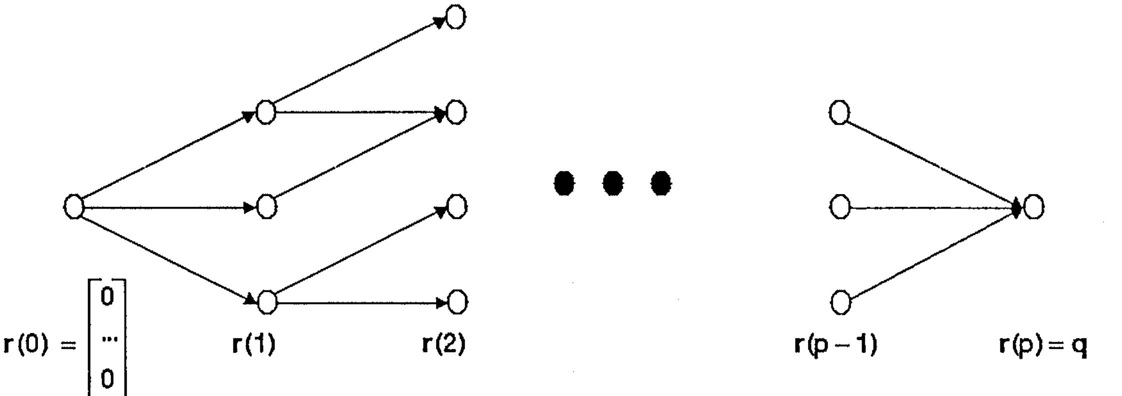


Figure 3-21: A trellis of admissible sequential schedules.

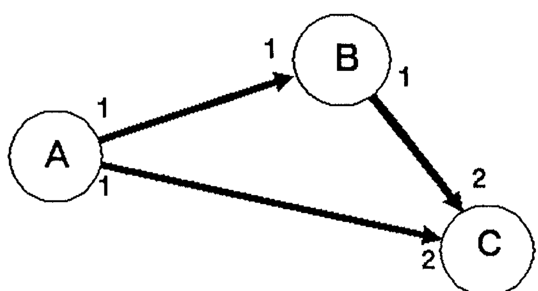


Figure 3-22: A SDF graph example.

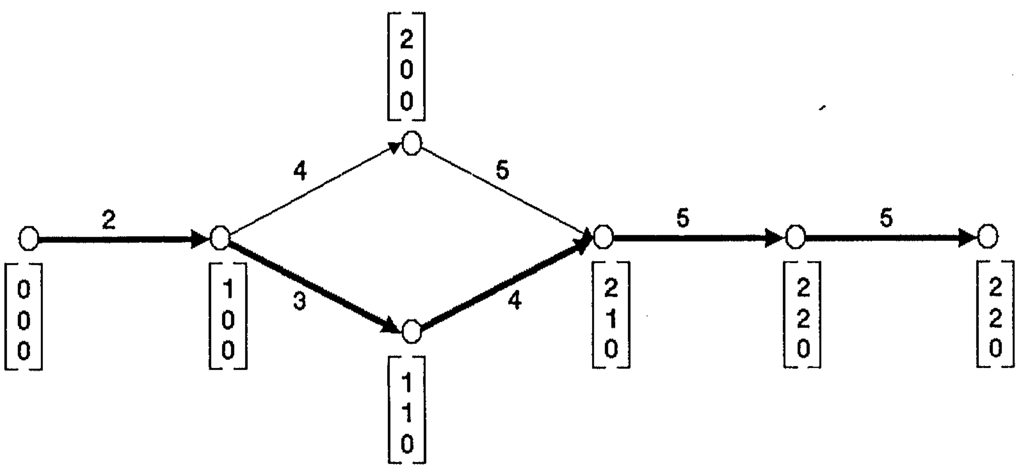


Figure 3-23: The trellis of unit blocking factor schedules for the graph in figure 3-22.

23. The nodes are labeled with $\mathbf{r}(n)$ and the arcs with the memory required by the schedule so far under criterion A. The only decision point is at $n=3$, where $\mathbf{r}(3)=[2,1,0]$, and the optimal path through the trellis is shown with bold lines. It is better to run node B before running node A a second time because the buffer on the arc $A \rightarrow B$ will only require one memory location.

3.5.2. Sharing Buffer Memory

The method illustrated in the previous subsection can be used with either memory minimization criterion to construct a minimum memory schedule. Given a schedule, it is sometimes possible to share the same memory locations in different buffers. The method is applicable for any schedule, regardless of whether memory use is minimized by the schedule, and therefore is still useful for multiprocessor realizations where throughput is more important than minimizing memory usage.

Any given location in any buffer has valid data for only part of each period of a periodic schedule. Determining when data is valid is an easy byproduct of constructing the schedule. Define a graph that contains one node for each buffer location. Connect all nodes that contain valid data at the same time. The problem is to allocate memory locations to nodes such that no two nodes with arcs connecting them share the same memory location and the total number of memory locations is minimized. This is equivalent to coloring the graph with the minimum number of colors such that no neighboring nodes have the same color. Graph coloring problems in general are NP-complete, but this particular graph is an *interval graph*, and interval graphs can be colored in linear time[Golu80a].

We have not yet implemented the algorithm, so the amount of memory it will save on practical applications is not clear. A successful technique for minimizing memory use through both scheduling and memory sharing should exhibit no increase in the amount of memory required when the granularity of the SDF graph is changed. That is, the memory required is ideally a function only of the size of the state required by the algorithm, and not the granularity of the specification of the algorithm.

Problems that might be encountered in implementing the memory sharing technique include difficulty reconciling memory sharing with circular buffer techniques. A processor with programming support for circular buffers usually requires that the memory in the buffer be contiguous, which imposes an additional constraint on the memory allocation algorithm. A second potential problem is the management of buffers that are used to store past data for future reference. Such buffers, however, contain part of the inherent state of the system, so it is probably unreasonable to expect to reduce the memory they require. In spite of these problems, memory sharing is likely to work well for buffers with a maximum length of unity, and such buffers are likely to dominate if scheduling is done to minimize buffer lengths.

4

THE π ARCHITECTURE

The impressive performance of monolithic digital signal processors (DSPs) is moderated by the difficulty of programming them. Hardware techniques such as extensive pipelining typically increase raw computational power, but if taken too far, they lead to unwieldy programming. Standard methods for overcoming this difficulty are effective only for modest amounts of pipelining[Sher84a, Kogg81a].

The tradeoff between programmability and performance is explored in section 4.3 by comparing DSPs from different manufacturers, with the conclusion that maximizing the performance of a moderately pipelined programmable architecture requires abandoning traditional sequential programming. In addition, extensive pipelining leads to lost resources due to *hazards*[Kogg81a]. To avoid these difficulties, architects tend to limit their use of pipelining, which means significant performance sacrifices. Here, architecture is considered *together* with programmability, yielding a *system solution* that permits efficiently

combining extensive pipelining with simple programming.

In this chapter, we resurrect an old but rarely used architectural technique to drastically alter the character of a deeply pipelined processor. Instead of a single program, multiple programs are interleaved in a pipeline. Each program experiences none of the difficulties of pipelining, and will be consequently easier to write and optimize, regardless of the amount of pipelining used. However, multiple processes must now cooperate on a signal processing task, or multiple signal processing tasks must share the same hardware. In the later case, the latency of each process will be greatly increased, compared to running on a non-interleaved pipelined processor. This may be important in some applications, such as control.

In the following chapter we illustrate the use of SDF programming for a π processor. Very low overhead implementation of such programs on such a processor is possible. The combination of transparent pipelining in the architecture and a natural paradigm for expressing concurrency in signal processing systems makes possible a powerful high level language for high performance signal processing.

4.1. MONOLITHIC PROGRAMMABLE DSPs

Monolithic programmable DSPs are usually applied to problems that differ significantly from general purpose computations. They are consistently numerically intensive, and many are *real-time*, meaning that the full set of input data is not available before output data must be computed[Acke82a]. For most such applications, algorithms are repetitively applied to an essentially *infinite* stream of input data. Most of these applications are *hard real-time*, meaning that the

time constraints are firm. In other words, the probability of failing to meet a time constraint must be identically zero. Consequently, probabilistic speedup techniques that are so effective in general purpose computing, such as cache memories, are more difficult to use effectively. Hard real-time numerically intensive applications demand much more performance than general purpose applications, so I will consider only applications of this type.

When extreme performance is demanded of DSPs, custom VLSI, in principle, has considerable advantages over off-the-shelf programmable devices. The hardware can be fine-tuned to provide only the requisite arithmetic precision and speed for each part of an algorithm. In addition, silicon compilers[Pope84a, Jhon85a, Kahr84a] have begun to demonstrate their efficacy by considerably reducing the design effort while maintaining some of the benefits of custom. However, the performance of programmable devices is not far behind custom for many applications, and there are considerable disadvantages to custom. The *design-time* for most applications of silicon compilers is still substantial, greater than the design time for software for programmable processors. Although silicon compilers are closing the gap, software improvements in programmable processors may widen it again. There is substantial room for improvement in real-time programming for signal processing. The *evaluation-time* for custom devices is considerably greater than for programmable devices. Evaluation of algorithm implementations can either be done in simulation, or in real-time, using fabricated hardware. Usually, both are required, especially if the application requires some experimentation with untried algorithms. Functional simulation clearly offers considerable advantages in turnaround time, but it generally runs a factor of 100-1000 slower than

real-time. This is sometimes not sufficient for proper evaluation of an implementation. For example, a superbly qualified design team might produce an adaptive equalizer chip for a modem. If the equalizer is fractionally spaced[Unge76a, Gitl81a], and the implementation does not use saturation arithmetic, tap drift[Gitl82a] will cause the equalizer to lose convergence after possibly hours of real-time operation. Tap drift is a slow phenomenon unlikely to show up in any but the longest simulations. A design change may be required quite late in a custom development. The tap drift phenomenon was not recognized until relatively late in the history of adaptive filters, suggesting that even algorithms that are considered understood can produce unexpected results. Real time evaluation, using fabricated chips, is slower because fabrication is in the loop. Even rapid turnaround techniques will be only able to reduce this to weeks, as compared to the seconds or minutes required to compile a program. A closely related problem is the *turnaround-time* for engineering changes. Designs need to be frozen earlier when custom chips are developed.

To take advantage of concurrency in algorithms, systolic arrays are sometimes promising for implementation[Kung80a]. But systolic arrays have so far failed to materialize as a competitive alternative except in very few isolated circumstances. They apply only to problems with regular structure, and therefore are unlikely to be useful for more than a part of a large application. Our aim here is to design a system that easily implements algorithms lacking regular structure.

Specialized architectures for signal processing have been around for quite some time[Alle75a], but monolithic DSPs are a relatively new phenomenon.

Before the emergence of the first generation chips, such as the DSP1 from Bell Labs[Chap81a], the S28211 from AMI[AMI,a], the TMS32010 from Texas Instruments[Texa83a], the MB8764 from Fujitsu[Tsud83a], the uPD7720 from NEC[NEC], and the IBM DSP[Unge85a], real-time programmable signal processing was mostly beyond the capacity of microcomputers and was therefore confined to the laboratory. Monolithic DSP devices achieve throughputs equivalent to one multiply and accumulate instruction every 100 to 800 nanoseconds, and use a restricted fixed point data precision of 16 to 20 bits. It is mainly the multiply and accumulate speed that makes these devices useful. They have been applied to digital filtering, modem design, transmultiplexers, and speech processing. Nonetheless, the first generation devices have barely the processing power required to be able to do useful work with voiceband applications, which involve sample rates ranging from 600 samples per second (for low speed modems) to 8000 samples per second for speech. (DSPs without hardware multipliers, such as the IBM RSP[Mint83a] and the Intel 2920, have not been as successful.) They are rarely useful for applications with higher sample rates. Singer[Sing85a] gives a detailed analysis of the limitations of some of these devices for certain speech and modem applications.

The first generation DSPs are all difficult to program compared to more traditional microprocessor architectures; the "assembly" language of an early device (the Bell Labs DSP1) closely resembles horizontal microcode; the programmer must be cognizant of each stage of the pipeline and the operations simultaneously performed in these stages. Other manufacturers make some attempt to hide the pipelining from the user, but the result is lost efficiency or sometimes cryptic programming techniques. There is a paucity of software

support, such as compilers, partly because of the difficulty of designing efficient compilers for such machines, and partly because these devices have capabilities so close to the bare minimum to be useful, that little inefficiency can be tolerated for the benefit of programmer convenience. "High level" programming for these devices typically consists of code generators for specialized applications, (FIR filters, for example[Mint83a]) and macro or subroutine packages.

One of the reasons that high level languages have failed to materialize is that traditional high level languages easily support features that are not appropriate in a signal processor. A compiler that attempts to manage capabilities such as strings, flexible I/O, and relative addressing will be difficult to write for a machine that was not intended for this type of use. For this reason, a more specialized, more *minimal* language is required.

Key features common to first generation DSPs are

- fixed point arithmetic, with hardware multiplication,
- Harvard architecture, or some variant,
- hard wired instruction set (not microcoded),
- parallelism, and
- pipelining.

An example of such a machine is shown in figure 4-1. The term *Harvard architecture* describes machines which use separate memories to store instructions and data. The advantage is that these memories can be accessed in parallel, so effective memory bandwidth increases. This technique can be modified by simply using more than one memory without restricting the contents of the memories to instructions or data. Besides parallel memories, other forms of

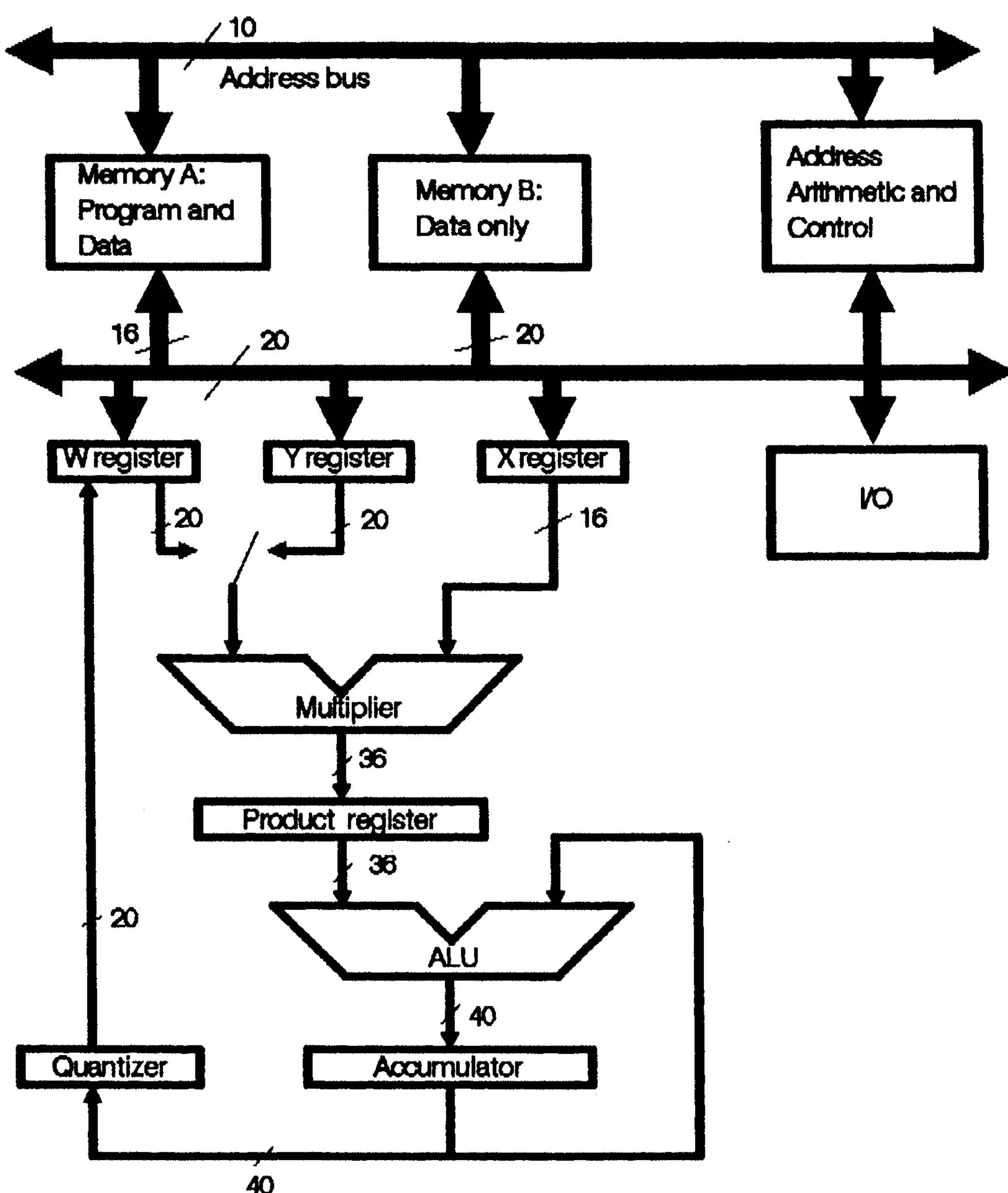


Figure 4 – 1: Simplified block diagram of a DSP

parallelism are used in these devices. Figure 4-1 shows separate hardware for I/O, and hardware for address arithmetic and control. The "quantizer" hardware is used to round the full precision (40 bit) accumulator contents to get a 20 bit result to store in memory.

The key feature we will be most concerned with is pipelining. The first generation DSPs generally have pipelined data paths with simple, single-path control. Figure 4-2 schematically illustrates the pipelined execution of a multiply and accumulate operation on the Bell Labs DSP1[Chap81a], the first generation device with the most pipelining. It is clear that there are certain arithmetic tasks that will be performed more efficiently than others. Inner products, critical in signal processing, are particularly efficient. Divisions, on the other hand, are considerably more difficult. Since every arithmetic instruction uses this data path, if an instruction appears that does not require a multiplication, *no efficiency is lost* by passing it through the multiplier (and multiplying by one) because the multiplier cycle would be lost anyway if the throughput of the multiplier equals that of the control circuitry and the memory.

The simple control flow in the data path is a principal reason that we can expect to dramatically increase the throughput of these devices by using deeper pipelining. Multipliers are particularly easy to divide into pipeline stages[Capp84a]. The cost of the additional pipelining (in VLSI real estate) is modest. For example, a 16×16 array multiplier with five pipeline stages has three times the throughput of a flow-through version, with only 13% more VLSI area[Capp84a]. But as the amount of pipelining increases, the programming problem dominates.

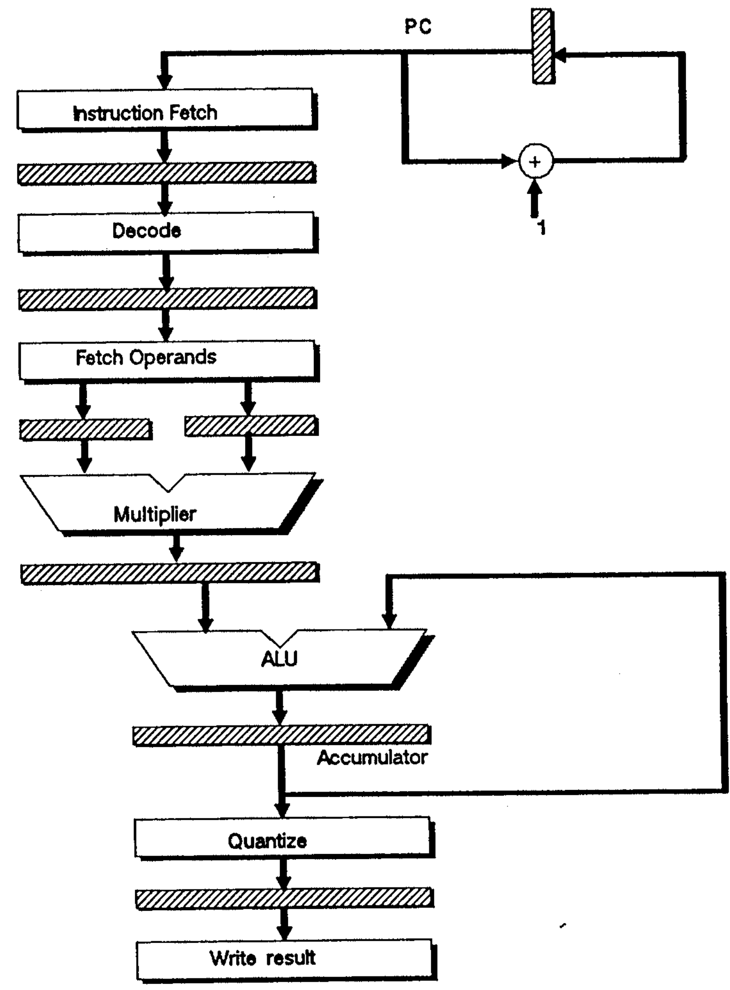


Figure 4-2: An abstraction of the arithmetic pipeline of the DSP in figure 4-1.

Inevitably, second generation DSPs have more capability. The Texas Instruments TMS32020 has a larger address space and an enhanced instruction set[Maga85a]; the Bell Labs DSP32 has 32 bit floating point arithmetic hardware[Kers85a]; the NTT DSP has 18 bit floating point arithmetic and elaborate addressing modes[Yama85a]. The Motorola DSP56000 has support for Circular buffers[Moto86a]. There are many ways to enhance a basic architecture for greater flexibility. The designers of semiconductor processes continue to supply systems engineers with greater circuit capacity on VLSI chips. However, circuit *speed* increases only modestly with each reduction in circuit geometries, especially for the processes that yield the greatest density. As a consequence, system designers tend to upgrade VLSI products by increasing their functionality rather than increasing the speed of the simple architecture. After a point, more enhanced architectures are not likely to perform the basic signal processing functions significantly faster than their predecessors. In order to expand the applications of these devices beyond the voiceband range, it will be necessary to use improving VLSI technology to improve the speed at which basic signal processing functions are performed and to increase the memory capacity, rather than to embellish the architecture.

Successful experiments with reduced instruction set computers (RISCs)[Patt81a] support the notion that simply increasing functionality is not necessarily the best use of the improving hardware technology. A RISC is a processor with a small instruction set that executes very fast. The chip resources saved by restricting the functionality are then applied to increasing the size of the register set or making other changes that improve the speed of the processor. Thus, costly hardware that supports esoteric and rarely used instructions is

avoided. The first generation DSPs share some of the RISC philosophy, but for a different reason. The technology has been unable to support more than a minimal instruction set combined with fast arithmetic hardware and memory on a single device. There is potential benefit in deliberately embracing the RISC philosophy and concentrating on enhancing the speed of these devices.

4.2. HAZARDS

Many existing high performance systems depend heavily on pipelining to achieve high throughput. Although many researchers discuss the virtues of pipelining, citing increased use of circuit resources for better performance[Shar74a, Jump78a, Davi73a, Capp83a, Capp84a], pipelining has considerable disadvantages in programmed machines. Pipeline *hazards*[Kogg81a] cause idle resources that degrade performance. The most common types of hazards are due to *data conflicts* and *branching conflicts*. Branching conflicts occur when instructions that are prefetched prior to the execution of a (possibly conditional) branch are not executed; machine cycles are wasted. Data conflicts occur when computed results are not ready when they are needed. For example, an instruction may require the result of a previous instruction, and the result may not be ready when execution of the second instruction begins. A delay must be inserted between the execution of these instructions in order to keep the second instruction from executing before its data is ready. Such delays cause idle machine cycles called *pipeline bubbles* and degrade the performance of the machine. The following table shows an example with three successive multiply and accumulate operations on an architecture with the pipelining of figure 4-1.

TIME SLOT	MULTIPLY AND ACCUMULATE OPERATIONS		
	1	2	3
1	I-fetch		
2	decode & address	I-fetch	
3	op. fetch	decode & address	
4	multiply	op. fetch	
5	accumulate	multiply	
6	quantize	accumulate	
7	write	quantize	I-fetch
8		write	decode & address
9			op. fetch
10			multiply
11			...

The first two operations have no operand dependencies, so the second can begin immediately after the first; but the third instruction is delayed. The third operation requires the result of the second, in this example. Its *operand fetch* needs to follow the *write* operation of the second instruction. The penalty is a pipeline bubble of four time slots. Kogge cites a 66% loss on the IBM 360/91 due to pipeline bubbles[Kogg81a]. In that machine, the extensive pipelining is transparent to the programmer; bubbles are automatically handled by the hardware using an *interlocking* mechanism. Conflicts are detected automatically and instructions are delayed.

There are, of course, some simple techniques available for reducing the number of cycles lost to pipeline bubbles. For example, if the programmer, compiler, assembler, or hardware can detect the conflict illustrated in the above table, a *short circuit*[Kogg81a] can be implemented, in which the offending operand in instruction 3 would be taken directly from the quantizer hardware, bypassing the memory. (The quantizer rounds or truncates the 40 bit accumulator so the result can be written in memory or used as a multiplier operand.)

Instruction 3 can be started one cycle earlier. A general form of this is implemented in a Lincoln Labs signal processor[Paul80a], using a *write queue*, which is a tagged memory with the write address serving as the tag. As a further defense against pipeline bubbles, compilers can optimize code by reordering instructions and predicting branching.

These techniques are not completely effective, however. Many researchers involved in practical design of more-or-less general purpose computers cite the difficulty of preventing pipeline bubbles, and conclude that after a point, nothing is gained by deeper pipelines[Sher84a, Patt81a].

4.3. THE PROGRAMMING/PERFORMANCE TRADEOFF

Architects of high performance DSPs determine how much of the peculiarities of the architecture are hidden from the programmer. Two first generation DSPs, the Bell Labs DSP20, a faster version of the DSP1[Chap81a], and the Texas Instruments TMS32010[Maga82a], have very similar capabilities, but radically different architectures. Comparing them is instructive.

The widely used Texas Instruments TMS32010 uses a limited amount of pipelining and variable instruction execution times so that the user need not be aware of the pipelining. The pipeline stages are shown in the following table (simplified).

stage	function
1	Instruction fetch
2	Decode and operand fetch
3	Execute

To avoid pipeline hazards, some instructions, such as branches, take more than one cycle to execute. Each instruction therefore appears to complete before the next instruction begins.

By contrast, the Bell Labs DSP20, with the same hardware capabilities, uses much more pipelining. To program the machine, the user is required to be fully aware of the pipelining. The second generation Texas Instruments device, the TMS32020[Maga85a], and the second generation Bell Labs device, the DSP32[Kers85a], will also be briefly compared, but the features of these devices differ more significantly, so the comparison is less meaningful.

4.3.1. Computing Inner Products

Perhaps the most basic benchmark for digitial signal processors is the finite impulse response (FIR) filter. An FIR filter consists of a delay line and a set of tap values, and the output is the inner product of samples in the delay line and the taps. The Bell Labs DSP20 and TI TMS32010 achieve precisely the same performance on this benchmark, but the TMS32010 requires considerably faster hardware to do this.

The TMS32010 has single operand instructions that execute in a pipeline schematically illustrated in the above table. This illustration of the pipelining neglects the finer division of each instruction cycle into finer machine cycles but

adequately explains the behavior of all instructions. The instruction cycle time is 200ns. Multiply and accumulate operations require three successive instructions:

load one operand
multiply by the other operand
accumulate

For successive multiply and accumulate operations, commonly used to compute inner products and FIR filters, the accumulate operation is combined with fetching the next operand.

load one operand
multiply by the other operand
accumulate, and load another operand
multiply by yet another operand
accumulate, and load a fourth operand
etc.

For long inner products the time required is 400ns per multiply.

This approach to computing inner products has been improved in two different ways in the TMS32020[Maga85a] and the IBM DSP[Unge85a]. In the TMS32020 a *repeat* (RPT) instruction precedes a *two operand* multiply and accumulate instruction. The second instruction is repeated the number of times specified by the operand of the RPT instruction without having to be refetched. The saved memory cycle is used to fetch the second operand in parallel with the first. In the IBM device, two accumulators are used to simultaneously compute

two outputs of an FIR filter. The computation of the two outputs is interleaved, so the technique is called *zipping*, presumably from the word *zipper*. Most operands (filter coefficients or data) are used twice each time they are fetched, once for each output. Both the repeat instruction and zipping save memory accesses, permitting the throughput to be roughly doubled, *but only for inner products*.

In the TMS32010 most instructions specify only one operand, so it takes twice as many instructions to perform the same function, compared to the Bell Labs device. Its instruction cycle time is twice as fast as the Bell Labs DSP20, however, so for inner products, the performance of the two devices is identical.

Achieving the same performance with fewer pipeline stages suggests that the pipelining in the Bell Labs chip is excessive. However, the TI device pays a price for its reduced pipelining. Twice as many instructions are required for the same operations, so twice the program memory bandwidth is devoted to instruction fetches (both devices have 16 bit instructions). Consequently, for the same speed memory (200ns per access), the TI device cannot access data in the program memory in a single cycle. Arithmetic instructions in the TMS32010, therefore, access data only from the data memory. This puts pressure on the data memory bandwidth. In particular, for efficient FIR filter implementation without circular buffers, three memory accesses are performed for every multiplication, two operand reads and a write (to shift data in the delay line). The Bell Labs DSP20 requires only one instruction fetch per two operand multiply, for a total of four memory accesses per FIR filter tap. The TMS32010 requires five. The data memory of the TMS32010 is designed to

include both a read and a write cycle in one 200ns instruction cycle, whereas the the DSP20 requires only a read *or* write cycle every 200ns.

More dramatically, the arithmetic hardware in the TMS32010 is twice as fast as that in the DSP20 to achieve the same performance. The multiplier and ALU of the DSP20 each have a full 400ns to complete their operations, whereas only 200ns is available in the TMS32010. This suggests that with more pipelining, the TMS32010 hardware could have been used to build a significantly faster processor.

4.3.2. Explicit Pipelines

There is a price for the additional pipelining of the DSP20. In addition to more difficult programming, we would expect increased incidence of pipeline bubbles, causing lost resources. However, the programmer of the DSP20 has *explicit* control over the pipeline, meaning that bubbles can be avoided by careful code construction.

There are many ways to make sure the programmer is aware of the pipelining of each instruction. A simple way is used in the Bell Labs DSP32[Kers85a]. For that device, a multiply and accumulate instruction looks like

```
a3 = a3 + (*r9++)*(*r8++);
```

The registers r8 and r9 are addressing registers, the asterisk is used to indicate indirection, and (*rN++) indicates a post auto-increment by one. The register a3 is one of four accumulators. The accumulator a3 is not updated until three instructions later. That is, if instructions I1, I2, and I3 all update a3, and instruction I4 uses a3, the value of a3 it uses is established by I1. A three

operand instruction is also possible, in which a memory write occurs.

```
*r10 = a3 = a 3 + (*r9++)*(*r8++);
```

The memory location specified by r10, however, is not updated until four instructions later. The programmer therefore must be aware of the scatter in time of the instruction execution due to pipelining. In addition, certain instruction sequences are prohibited, with no-ops required in between, because of conflicting demands on processor resources. Once the small set of programming rules is learned, the essentials of the pipelining is evident, and the programmer can often use this knowledge to optimize the code.

The instructions of the DSP32 are *data stationary*, meaning that each instruction fully specifies the operations on a set of operands, and the instruction execution is scattered in time. An alternative approach, *time stationary* programming, used in the earlier DSP20, makes pipelining even more explicit.

An example of an FIR filter implemented with time stationary code is given in the following table.

	i=-1	
	rx=coef_location_end	
	ry=data_location_end	
	rd=data_location_end	
		p=*rx++i * *ry++i
*rd++i=y	a=p	p=*rx++i * *ry++i
*rd++i=y	a=p+a	p=*rx++i * *ry++i
	...	
*rd++i=y	a=p+a	p=*rx++i * *ry++i
	a=p+a	
	w=a	
*rd=w		

The first four instructions simply set up some registers. Indirect address registers rx, ry, and rd are used, and the notation *rx++i means "get data from the memory address in rx, and then increment rx by the amount in register i." The accumulator is designated "a" and the product register "p". Most arithmetic instructions have four columns. The action specified in each row is executed simultaneously, but the programmer can understand the instruction as if it were executed from left to right. The rightmost column specifies the multiplication to be performed, and the operands on which it will be performed. The operand specification is actually moved up two instruction (*skewed*) by the assembler, and the programmer must be aware of this, particularly when implementing branches. The second column from the right specifies the ALU action, accumulation in the case of an FIR filter. The third column from the right seems to specify only a register transfer, but actually specifies a quantization operation. It is only used once in table 4 because only the result of the inner product is quantized. Finally, the leftmost column specifies a memory write operation.

Although programming the DSP32 is less cumbersome than the DSP20, both are more difficult than programming the TI TMS320 family. From the point of view of the programmer, programming the TMS32010 is both data and time stationary because each instruction appears to complete before the next instruction is initiated. The pipelining is not evident to the programmer. This is clearly desirable, but it has its price. As mentioned earlier, the TMS32010 requires a multiplier and ALU twice as fast as that of the DSP20 to achieve the same performance on inner product benchmarks. However, on the second most basic DSP benchmark, an infinite impulse response (IIR) filter, the DSP20

outperforms the TMS32010 in spite of its slower arithmetic hardware. This improvement is a consequence of the more cumbersome programming, which enables the programmer to work around data conflicts.

4.3.3. Recursive Filters

It is particularly easy to get good performance with a pipelined processor computing an inner product because there is no recursion. Surprisingly, the more pipelined processor outperforms the less pipelined processor on the second most basic signal processing benchmark, the second order recursive digital filter, or *biquad*.

For the five coefficient biquad of figure 4-3, the TMS32010 requires 20% more time than the DSP20. For a four coefficient biquad, the figure is 25%.

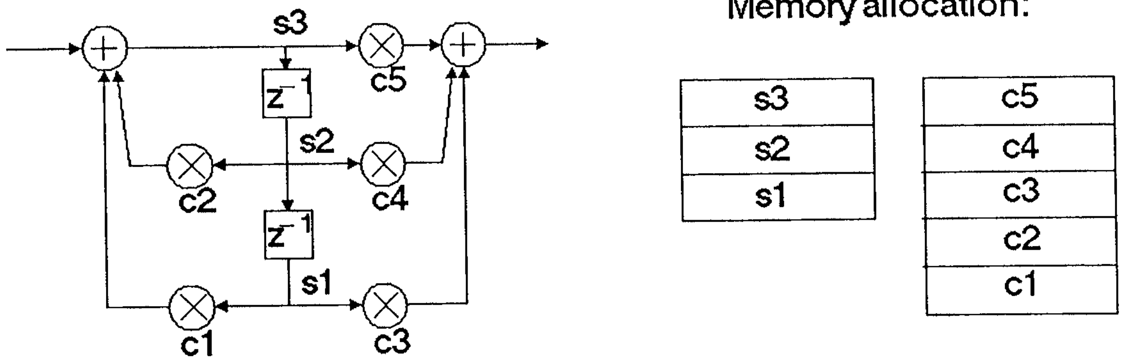


Figure 4-3. A five coefficient biquad.

Code segments for five coefficient biquads are given in the following tables. These code segments are written to cascade with similar code segments, getting cascaded second order sections. Recall that the instruction cycle time of the TMS32010 is half that of the DSP20. First, here is the code for the five coefficient biquad of figure 4-3 on the TI TMS320. The address register AR0 points to the data, s1 through s3, and AR1 points to the coefficients, c1 through c5. Memory is allocated as shown in the figure.

LTA	*-,AR1	; load s1	AR0 → s2
MPY	*-,AR0	; mult s1*c1	AR1 → c2
LTA	*+,AR1	; load s2	AR0 → s1
MPY	*-,AR0	; mult s2*c2	AR1 → c3
LTA	*-,AR1	; load s1	AR0 → s2
SACH	s3,0	; store to s3	
ZAC		; zero accumulator	
MPY	*-,AR0	; mult s1*c3	AR1 → c4
LTD	*-,AR1	; load s2, move s2 to s1	AR0 → s3
MPY	*-,AR0	; mult s2*c4	AR1 → c5
LTD	*-,AR1	; load s3, move s3 to s2	AR0 → s1 (next biquad)
MPY	*-,AR0	; mult s3*c5	AR1 → c1 (next biquad)

We have assumed that multiple biquads are cascaded using in-line code, and the data and coefficient memory spaces for each biquad are adjacent. The first instruction performs the last addition of the previous biquad, after which the input to the current biquad is in the accumulator. The last column indicates where the auxiliary registers point after execution of the instruction. The current auxiliary register is AR0 on entry, and the auxiliary registers are assumed to point to data as follows:

AR0 → s1

AR1 → c1.

The memory is allocated so that after executing the above instructions, the two auxiliary registers point to the appropriate data of the following biquad section.

Next we have the code for the same filter implemented on the Bell Labs DSP20. The address register ry points to the data, and the coefficients are immediate (stored in program memory along with the program). Only two memory locations (s1 and s2) are required for the data in this code.

		a=p+a	p=c1**ry++j;	ry → s2
		a=p+a	p=c2**ry++i;	ry → s1
		a=p+a	p=c3**ry++j;	ry → s2
*rd++j=y	w=a	a=p	p=c4**ry++j;	ry → s1 (next biquad), rd → s2
*rd++j=w		a=p+a	p=c5*w;	rd → s1 (next biquad)

The first instruction again performs the last addition of the previous biquad, after which the input to the current biquad is in the accumulator. The comments indicate where the auxiliary registers point after execution of the instruction. The auxiliary registers are assumed to point to data on entry as follows:

$$ry \rightarrow s1$$

$$rd \rightarrow s1.$$

The index registers have the values j=-1 and i=+1. The memory is allocated so that after executing the above instructions, the auxiliary registers point to the appropriate data of the following biquad.

The DSP20 implements cascaded biquads with five instructions per section, while the TMS32010 requires 12. The TMS32010 has twice the instruction rate, so it requires 20% more time per section.

4.3.4. Avoiding Pipeline Bubbles with Hand-Optimized Code

In contrast to the 66% percent wasted cycles cited by Kogge[Kogg81a], in one production program for the the Bell Labs DSP-1 examined by the author (a proprietary program implementing a V22.bis voiceband data modem), only 10% of the instructions did not make use of the arithmetic unit because of pipeline bubbles. This program was painstakingly hand optimized to minimize the pipelining bubbles; for example, algorithms were chosen that did not require branching. Dividing this program into functional blocks yields a low of 2.3% idle cycles for an adaptive equalizer and a high of 29% for a phase locked loop with a polynomial approximation to a sine and cosine. Not surprisingly, the pipelining penalty is lowest for the application (an adaptive equalizer) in the class for which the architecture is optimized (FIR filters).

Although pipeline bubbles can be minimized, in principle, if the programmer is given the control implied by explicit knowledge of the pipelining, this is not a desirable solution. First, the optimization of the code is extremely tedious. Second, the amount of practical pipelining is still bounded at a rather low number. If the architecture is modified so that the number of pipeline stages is doubled, then the percentage of idle cycles due to bubbles will double. An architecture with four times as much pipelining, therefore, will have 40% of its capability wasted due to pipeline bubbles, even for the best, hand optimized code, rather than 10%.

4.4. PIPELINING AND INTERLEAVING

In this section, a modification to standard pipelined architectures is proposed. The modified processors, designated π processors, have the key feature that increasing the amount of pipelining does not aggravate the programming.

To mitigate the problems associated with deep pipelining, we use interleaving multiple programs in a single pipeline. The basic idea of *pipelined interleaved execution* (shortened to PI or π) is quite simple. In the pipeline of figure 4-2, an instruction and its associated data pass through $N=7$ pipeline stages before completion. A π processor differs only in that the instructions going through the pipeline correspond to N different processes (N independent programs) rather than one. In figure 4-4, seven programs are interleaved so that two successive instructions from the same program are separated by $N-1$ instructions from other programs. The consequence is that when the second instruction of a program is issued, the first instruction has been completed. The figure shows seven program counters (PCs) stored in a delay line, or FIFO queue. After PC_n addresses the instruction memory to fetch the next instruction of the nth program, it is usually incremented by one and put back at the end of the queue. When PC_n appears again to fetch another instruction, the previous instruction of the same program has completely cleared the pipeline. Therefore, no pipelining is visible to the programmer writing program n. This π processor appears to the user as seven parallel processors (each with no pipelining) that share memory without contention. These virtual parallel processors are called *π slices*, since they are really just slices of the original processor.

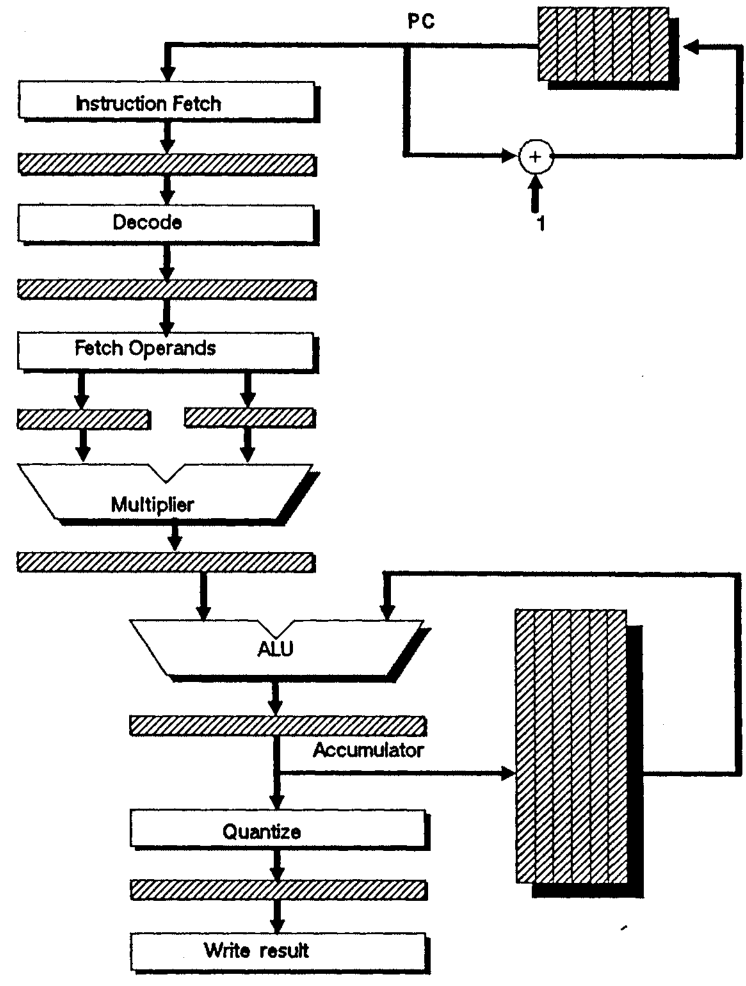


Figure 4-4: A π processor based on the DSP in figure 4-2.

4.4.1. History

Interleaving multiple processes on a single pipelined processor dates back at least to the peripheral processors in the CDC 6600[Thor64a, Thor70a] designed in the 1960s. The peripheral processor manages I/O devices that are much slower than the processor itself, and interleaving provides a convenient way to better use the resources of the processor. More recently, it was used in the HEP-1, a multiprocessor supercomputer[Smit78a, Dene81a] to overcome difficulties due to pipelined function units and random delays in memory accesses. Interleaved processes have also received considerable attention at Columbia University[Cohn83a] where the technique is being applied to general purpose computing. The architecture at Columbia, like the HEP, compensates for delays in memory accesses. It is discussed in considerable detail in section 4.7 because the programming methodology is related to synchronous data flow, and the contrast is interesting. What we wish to investigate is a much simpler device than the HEP or the Columbia architectures, a single chip microcomputer specialized to signal processing.

Pipelined interleaving has been used in telecommunications applications where a function needs to be applied to a large set of separate, independent signals. For example, eight ADPCM encoders can share the same pipelined hardware and independently encode or decode eight separate speech channels. Similarly, in the No. 4 ESS made by AT&T, 120 echo suppressors share the same hardware. In both of these cases, it is easier to share fast hardware than to try to design slow hardware that is proportionally smaller. However, unlike our proposal, these applications require all slices of the machine to perform the

same operation, which greatly restricts the applicability of the technique.

π processors are "shared resource" machines as defined by Flynn[Flyn72a, Flyn70a] in which processors compete for shared resources in either space or time. However, the architecture proposed here is different from many shared resource processors in an important and fundamental way; the entire processor is a shared resource, with the only private resource being the registers required to store the processor state. Jordan[Jord84a] calls the technique "Pipelined Multiple Instruction Streams" or PMIS, but this name does not suggest that the instruction streams are interleaved, so we prefer the designation "π". Shar and Davidson[Shar74a] are early advocates of the technique.

Shar and Davidson[Shar74a] critically examine the use of interleaved processes comparing the performance to an unpipelined microprogrammed machine taking several microorders to complete an instruction. They conclude that interleaving processes provides great advantages. However, many of the advantages they cite are a result of pipelining, not of interleaving. For example, they conclude that less control is required for an interleaved processor. They assume that each micro-order in the unpipelined machine is big enough to do anything in the machine, that there is no restriction on the order in which operations must be performed, as there is in a pipeline. With pipelining, each micro-order specifies real work for each pipeline stage, whereas without pipelining many fields of the micro-order specify no operation. This is an argument for pipelining, not for interleaving. Shar and Davidson make the technology dependent assumption that the memory cycle time is the same as an assembly instruction time, which corresponds to several microorders. They predict that

the interleaving technique would be less useful if faster memories are developed, but they neglect that more functionality per memory fetch may be demanded (for example, hardware multiplication/accumulation). In fact, our conclusion is that for DSPs the π technique becomes *more* useful if memories get faster relative to other circuits, because memory cycle times are a principle obstacle to extensive pipelining.

4.4.2. The Number of π Slices

It is not clear that a DSP requires as many pipeline stages as shown in figure 4-2. No other manufacturer, for example, has implemented separate hardware in the data pipeline to perform rounding and other quantization functions. Other manufacturers also have not added an extra pipeline stage at the end to write the result of a computation, relying instead on an additional instruction. Depending on the memory technology used, it may also be possible to decode the instruction in the same cycle with the instruction fetch or the operand fetch. The pipeline stages of figure 4-2, however, do make programs run faster, for a given memory technology. Since our interest is extensive pipelining for high performance, and figure 4-2 illustrates the most extensive pipelining in today's devices, we use it as our reference.

In practice, fewer than seven π slices are needed to overcome all pipelining difficulties in the architecture of figure 4-2. To determine the minimum number required to ensure the sequential nature of instructions, we observe that every loop in figure 4-4 must have the same number of latches. The two loops shown, one for the PC and one for the accumulator, each have seven latches. An important *implicit* loop is made explicit in figure 4-5. A memory location where a

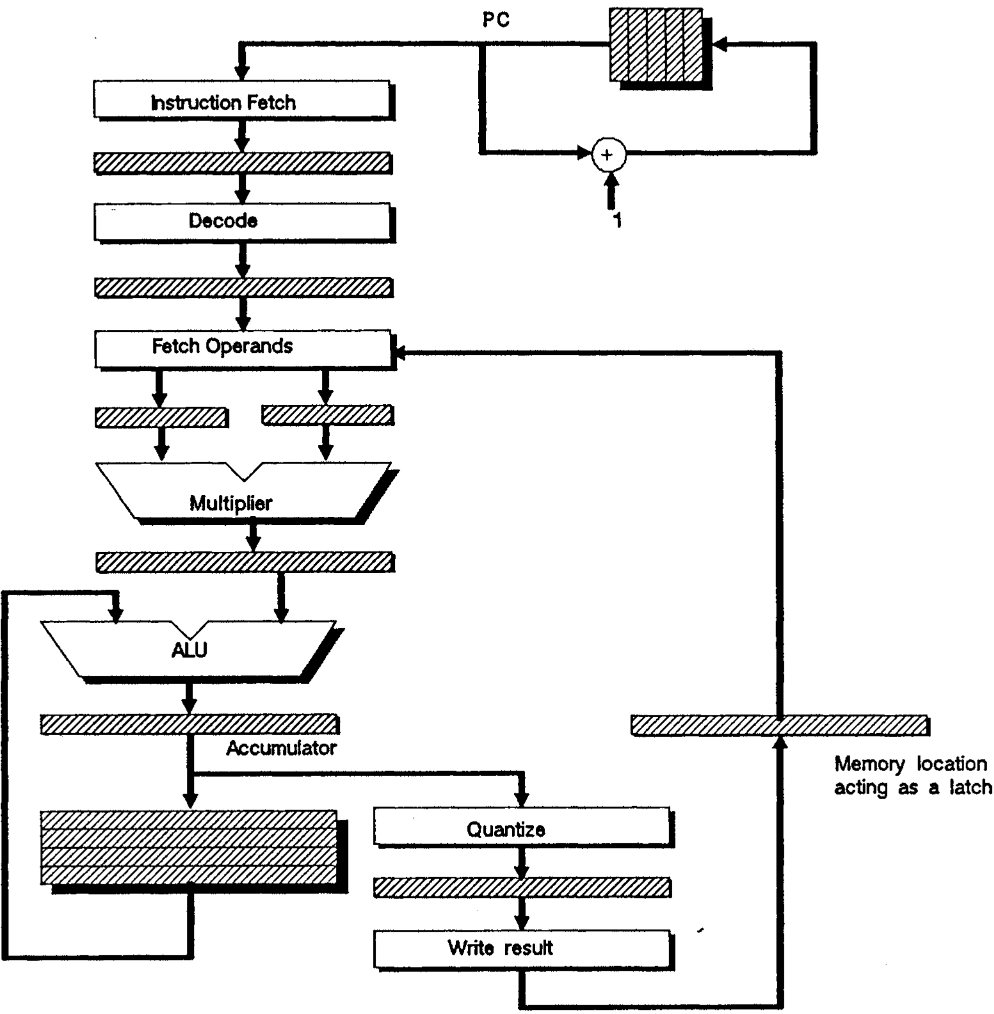


Figure 4-5: An illustration that data conflicts do not occur with a five slice, seven stage π processor. All loops have five latches.

result is written can be viewed as a latch, and the *fetch operand* block may take one of its operands from this latch. This is the case whenever an instruction is using the result of the previous instruction. The added loop has only five latches, so we can reduce the number of π slices to five, as shown in the figure, and the resulting interleaved architecture is still free of data conflicts. The number of π slices can be reduced one more by observing that if an instruction uses the result of the previous instruction, the result can be taken directly from the quantization hardware prior to the memory write. This is an example of a *short circuit*[Kogg81a]. However, in order to avoid *branching conflicts* we require at least five π slices with this architecture, as we will see, so there is no need for the short circuit.

Consider an unconditional branch first. The branch address is assumed to be part of the instruction, although this is not a critical constraint. Under this assumption, the branch address is available after the second stage of the pipeline, and therefore should be inserted into the PC delay line as shown in figure 4-6, in order to maintain a constant number of latches in every loop. The control path is shown with a dashed line from the decode hardware to the multiplexer.

Conditional branches based on condition codes generated by the ALU are also illustrated in figure 4-6. A conditional branch is decoded in the second stage, which sends a control signal to the *branch control* hardware. The control signal is delayed so that it arrives in the same cycle as the appropriate condition codes resulting from the previous instruction of the same π slice. A branch decision then determines which input is selected by the multiplexer. It is

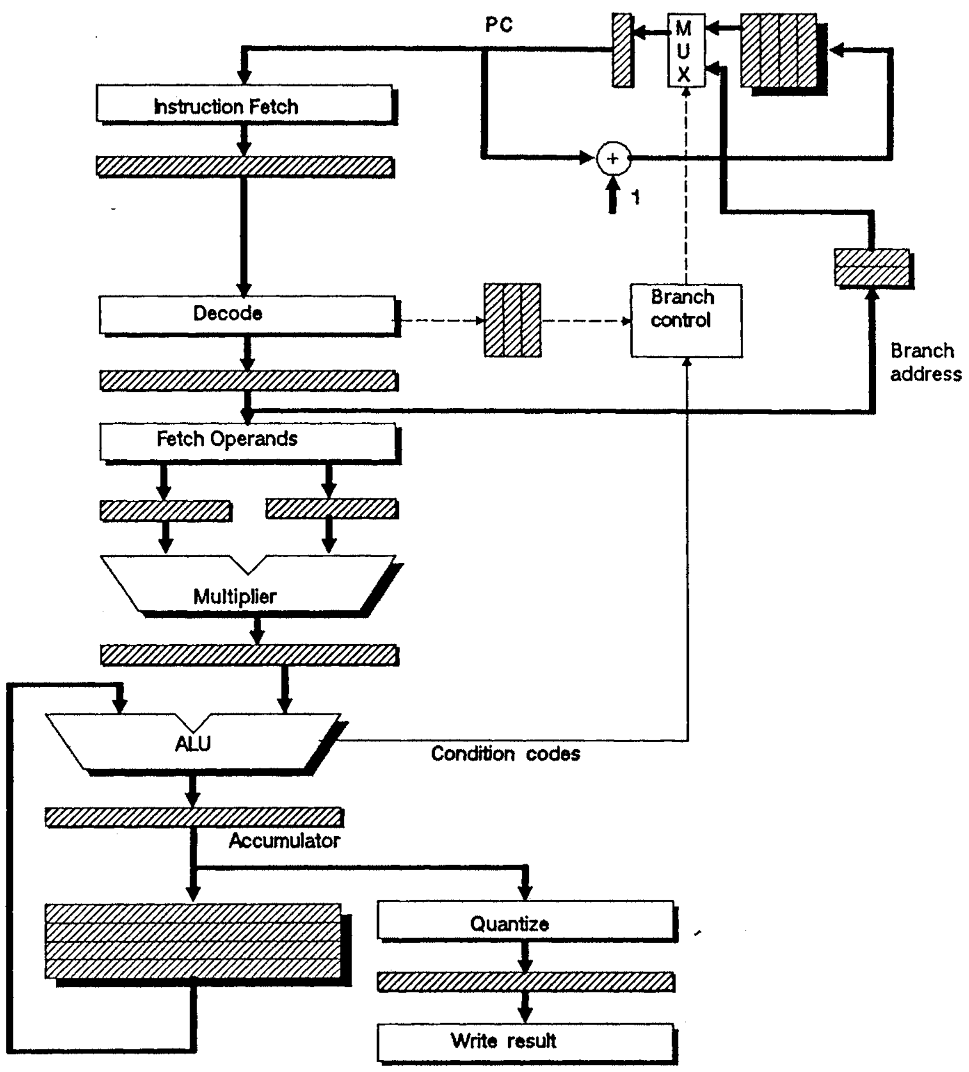


Figure 4-6: An illustration that branching conflicts do not occur with a five slice, seven stage π processor. All loops have five latches.

assumed that one instruction cycle is sufficient for generation of the condition codes, branch decision, and multiplexing. If not, one or more of these operations should share a cycle with the instruction fetch. If this is not possible either, then a sixth π slice is required, and all the loops have to be augmented to contain six latches.

4.4.3. Increasing the Pipelining

The principal hardware advantage of a π processor is the ability to do much more pipelining without suffering the detrimental effects usually associated with such pipelining. For example, if we were to add one pipeline stage within each stage of figure 4-6, we would double the number of π slices and nearly double the clock rate, thus nearly doubling the total computation power. It is particularly profitable, for example, to use more pipelining in the arithmetic hardware, as mentioned earlier[Capp84a]. Some of the pipeline stages, however, are difficult to divide, as discussed in the next section.

4.4.4. Memory

Memory accesses, particularly if the memory is dynamic, may be quite difficult, if not impossible to pipeline. However, if we wish to double the clock rate, the real objective is not to double the amount of pipelining, but to double the throughput of each pipeline stage. Figure 4-7 illustrates two ways of accomplishing this; the stage may be pipelined, or it may be interleaved. Such interleaving would normally require double the hardware, but a memory can simply be split into two smaller memories. If interleaved memory is used, the π slices no longer all share memory. With two-way interleaving, half of them

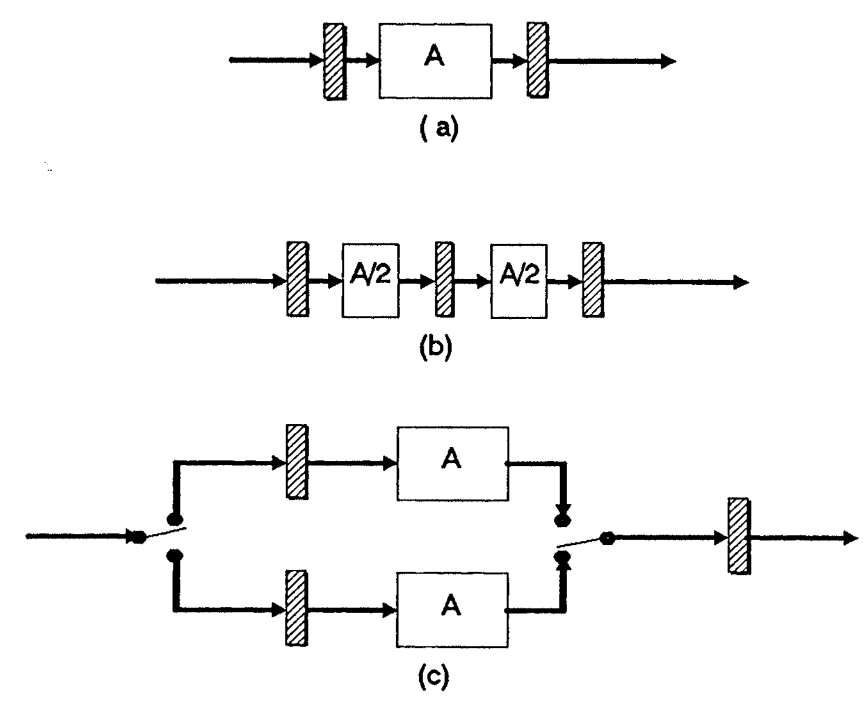


Figure 4-7: Techniques for increasing the throughput of a pipeline stage.

(a) A pipeline stage.

(b) Finer pipelining.

(c) Interleaving.

(called a π *cluster*) share one memory, and the other half share another. If communication among π slices is done through memory, then a special mechanism must be provided to communicate between π clusters. The two methods of figure 4-7 may be combined so that, in principle, any troublesome piece of hardware that resists further pipelining can be interleaved.

It is worth examining the current state of memory technology. Although rapid evolution is likely to make the numbers given here obsolete very quickly, the tradeoffs are best discussed in terms of existing technology. The number of π clusters can be written

$$\text{no. clusters} = \left\lceil \frac{\text{memory cycle time}}{\text{pipeline clock time}} \times BW \right\rceil$$

where BW is the memory bandwidth per instruction, or the number of accesses cycles per instruction per memory. The objective is to minimize the pipeline clock time while keeping the number of clusters acceptably low.

The most obvious way to do this is to minimize BW. The Bell Labs DSPs all have $BW = 2$. The TI DSPs have $BW = 1$, but they do half as much work per instruction. We can normalize BW by defining it to be the number of memory access cycles per *multiply and accumulate* per memory. Then $BW = 2$ for TI and Bell Labs. The Motorola DSP56000[Moto86a] uses three independent memories, one for programs and two for data, getting therefore $BW = 1$. This is the solution adopted in the architecture described in the following section.

Obviously, faster memories will minimize the number of π clusters. However, faster memories tend to be larger (in VLSI area), implying that less memory can be put on-chip. Static RAMs (SRAMs) reported at the 1985 ISSCC conference, mainly CMOS devices, had memory cell sizes below 100 square

microns and layout efficiencies of about 50% to 60%. The layout efficiency is the ratio of the area consumed by memory cells to the total die area of the chip. Address access times were around 45ns, but got as low as 17ns for a less dense RAM (cell size of about 270 square microns). Using three static RAM memories, therefore, the pipeline clock could drop as low as 17ns without any clustering at all. The cell sizes for Dynamic RAMs (DRAMs) reported at the same conference were about 36 square microns, and got as low as 20 square microns. The layout efficiency is about the same as for SRAMs. This implies that using static RAMs may imply about 5 to 14 times less memory will fit on the chip. This may not be a problem if all the memory is off chip, but putting all the memory off-chip may imply excessive bandwidth in off-chip communication. Furthermore, on-chip memory is effectively faster. The Bell Labs DSP32 is made using the same technology used for the 256K RAM, and the on-chip memory is made of the same one transistor memory cells as in the 256K RAM. This device has 32K bits of memory on board, and the access cycle time is 100ns. Using this dense memory technology and three memories, a pipeline clock time of 50ns requires two clusters, for example.

An attractive alternative is pipelined memories. Unfortunately, there does not appear to be much precedent. References to pipelined memories usually refer to systems where a complex memory system, supporting virtual memory or shared memory, for example, is pipelined. The memory itself is rarely, if ever, pipelined. For dynamic RAMs, the critical path is the precharging of the sense latches followed by the settling of these same latches. Such operations appear to be difficult to pipeline. The worst propagation and access delays occur concurrently with the precharging and sensing. It seems, therefore, that to

pipeline memory accesses, the sense latches would have to be duplicated for each new pipeline stage added. This technique is effectively equivalent to building a dual-port memory, and the area penalty for doing this is quite severe[Sher84a]. The approach greatly complicates the packing of circuitry. The NTT DSP[Yama85a], for example, uses dual ports to get two accesses from each memory in 140ns. Pipelining is easier in static memories, but since these memories are faster to begin with, the need is not as critical.

The safest conclusion is that interleaved memories are required for a fast pipeline clock. This implies that π slices in different clusters will not share memory, and a separate mechanism is required for them to communicate. This requires further investigation.

4.4.5. The Cost of State Replication

The principal hardware cost of interleaving is the need to keep around as many versions of each register as there are π slices, just as five program counters and accumulators are kept in figure 4-6. That is, the processor state must be replicated. However, these versions can be stored in FIFO queues implemented as delay lines with fixed length. These take little area in most VLSI semiconductor processes, and the benefit of being able to increase the amount of pipelining will offset this cost.

To put this issue in perspective, we note that the size of the processor state of the TI TMS32010 is 181 bits. The number of practical π slices is small (on the order of four through twenty, for architectures we have considered) so the total amount of memory dedicated to replicating the processor state is modest. The second generation TMS32020 uses 395 bits for the processor state, 104 of

which are I/O registers, timer registers, interrupt control, and system configuration registers, none of which would require replicating. So the processor state that would have to replicated to convert this architecture to a π processor is 291 bits, resulting in a still modest amount of memory. Most other DSPs exhibit similarly small processor states. The principal reason for this is that DSPs do not usually have general purpose registers, relying instead on the small and relatively simple memory.

A notable exception is the Bell Labs DSP32, which has four 40 bit accumulators and 21 16 bit registers. However, we must examine the reason for such a large processor state. Because of extensive pipelining in the DSP32, the potential for resources lost to pipeline bubbles is severe. The four accumulators and large register set, however, permit the programmer to interleave independent tasks so that otherwise wasted cycles can be put to productive use. That is, the programmer can often find unrelated tasks that need to be done, and can be done during the otherwise idle cycles. But so as not to interfere with the main task, a disjoint set of registers must be used. The large register set, and particularly the multiple accumulators, make this easy. The NTT DSP[Yama85a] also has four accumulators that can be used for interleaving. The IBM DSP[Unge85a] has two. This form of interleaving, although clearly a trend in DSP programming, is not as flexible as what we propose, because a single program counter directs all tasks and the interleaving must be done manually by the programmer.

4.4.6. Variations on the π

There are several alternative organizations of a π processor. One example is a processor with a variable number of π slices. Some algorithms may partition better into 10 slices, say than 7, making more effective use of the machine. Each π processor has the minimum number of slices, however, that ensures hazard-free operation. The number depends on the particular hardware resources put into the processor, and on the amount of pipelining. The only hardware cost associated with more slices than this minimum is the additional memory (longer dynamic delay lines) required to store the state of the additional slices. Further research is required to determine whether this flexibility is required often enough in real applications to be worth the cost.

Sometimes applications cannot be effectively divided for even modest parallel execution. Recursive algorithms, for example, are troublesome, because without altering the algorithm the recursion implies a fundamental limit on the amount of parallelism. This limit was discussed in detail in chapter 3. In the face of such applications, it may be advantageous to reduce the number of π slices below the minimum required for conflict-free operation. In this case, a mechanism is required to ensure that the integrity of the program is independent of the number of slices, since the program is written assuming no conflicts. One way to ensure this is to use hardware hazard detection and interlocking, as done in the IBM 360/91[Kogg81a]. This approach may be too expensive for implementation in a monolithic DSP. An alternative is to rely on the assembler to detect conflicts and insert **nops**. Instruction permutation could also be used to optimize the code. This approach is likely to work better if the number of

slices is only slightly below the minimum required for hazard-free execution. If, for example, only a single slice is being used, then the problem is not different from that of constructing an efficient assembler for an extensively pipelined machine programmed as if there were no pipelining. This problem is known to be difficult for deep pipelines[Kogg81a, Sher84a].

Another interesting variation on the π architecture runs the program from a given slice until a conflict is detected. We call such a processor a *self adaptive* π, illustrated in figure 4-8. This idea is due to Mintzer[Mint83a] who proposed switching contexts in a pipelined processor whenever a hazard is detected. The target machine obviously requires rapid context switching, which can be accomplished by clocking the π dynamic delay lines only when a conflict is detected. Mintzer proposed using PROCEEDs, an attribute appended to an instruction if the following instruction cannot be executed right away or is a **nop**. In other words, a single bit in each instruction, set by the assembler, indicates whether the following instruction can be executed right away. If not, the process (program counter and all registers) is put at the end of the process queue (the dynamic delay lines) and the next process is injected into the pipeline. This can be done without any overhead instructions, assuming that the assembler can always identify the instructions that can proceed. The main difficulty with this approach is that the *a-priori* performance of a given program will be difficult to determine. Essentially, each process becomes dependent on other processes. It is possible, for example, for a single process to hog the CPU indefinitely, if precautions are not taken to prevent this. Nonetheless, it is an interesting idea deserving further consideration.

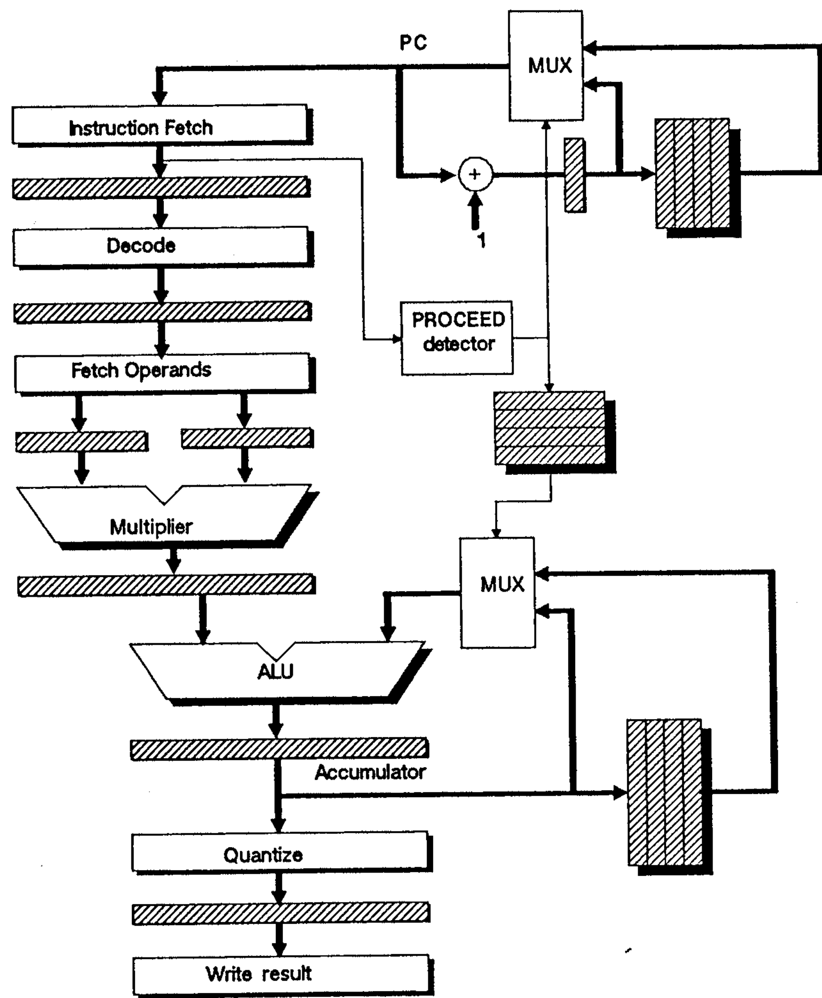


Figure 4-8: A self-adaptive π processor based on the DSP in figure 4-2. Any instruction with a PROCEED causes a context switch on the next instruction.

4.5. A SAMPLE ARCHITECTURE

So far, the π architecture has been discussed in the abstract. To begin testing the ideas we designed a specific architecture, designated the LM-π. The design philosophy is to deviate as little as possible from features generally available on existing DSPs, adding only features particularly useful for synchronous data flow programming. Major hardware differences are the pipelined interleaving, addressing modes supporting circular buffers, semaphores in memory supporting asynchronous data transfers between π slices, and an I/O mechanism that interfaces especially well with synchronous data flow programming. These are not trivial additions to the hardware, but they afford a considerably more efficient implementation of SDF programming, which itself affords considerably easier programming.

The LM-π has been implemented only as a functional software simulator. Committing the design to hardware may require design changes as the real cost of design choices becomes evident. Our philosophy in this architecture definition is to minimize the complexity consistent with our objectives. The 20 bit word size, therefore, is selected because it is the minimum to efficiently support circular buffers in this architecture. Similarly, the number of address registers is the minimum deemed necessary for SDF programming. A floating point architecture would probably enjoy wider application. With these caveats, and making no claim of completeness, we outline the architecture.

The overall block diagram is illustrated in figure 4-9. Its specific features are described in the following subsections. In the description below, we assume for clarity only rudimentary pipelining, but the architecture is designed

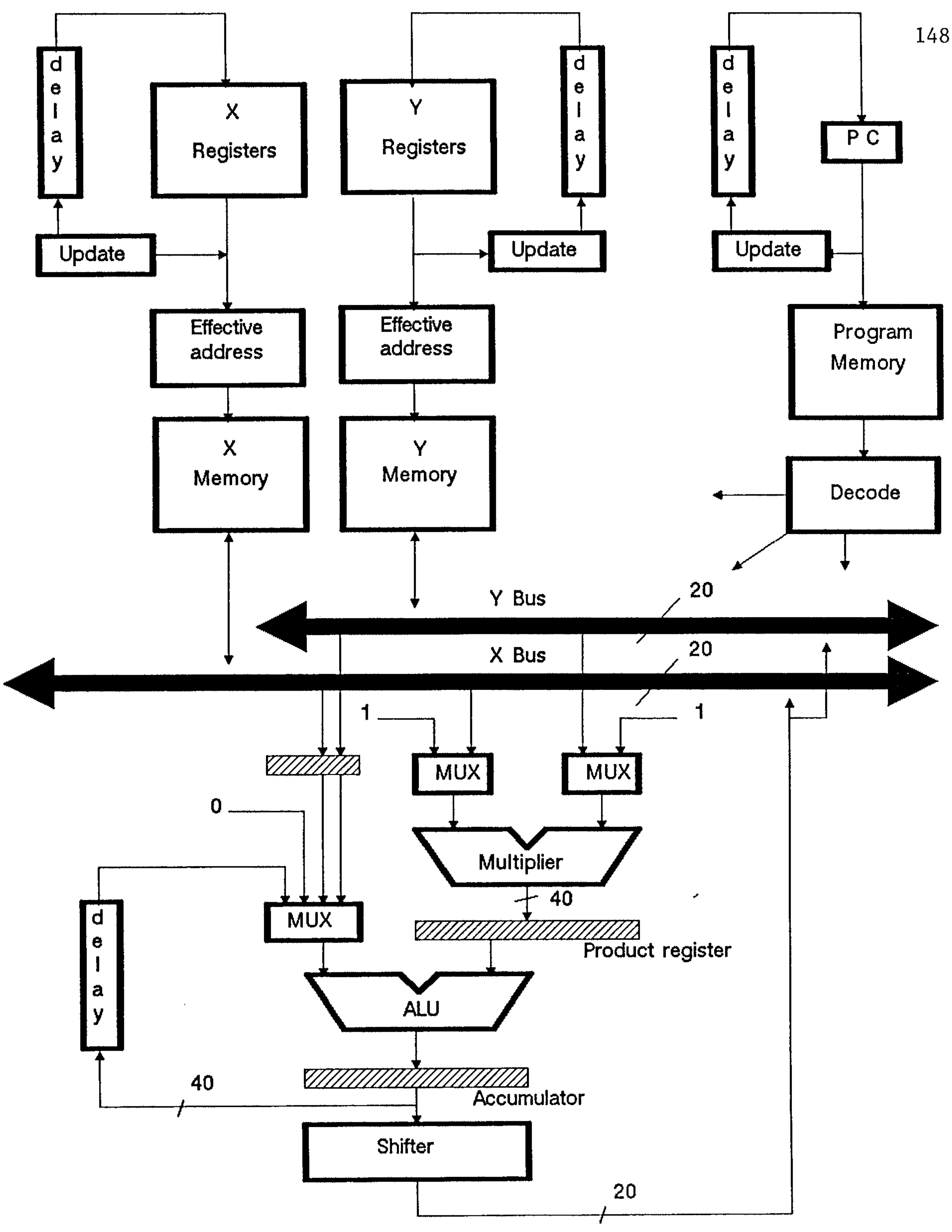


Figure 4-9: Block diagram of the LM-π.

specifically to permit additional pipelining using either of the techniques in figure 4-7. The reservation table, showing how the hardware resources are used by an instruction, is shown below.

RESERVATION TABLE FOR THE LM-π									
Cycle	1	2	3	4	5	6	7	8	9
P-Mem	O				X				
Decode		O				X			
X-Mem			O				X		
Y-Mem			O				X		
Mult				O				X	
ALU					O				X
Shifter		O				X			

Two instructions in the same π slice are illustrated, one labeled with an "O" and the other with an "X". There are three vacant cycles between the two instructions shown, implying a total of four π slices for totally invisible pipelining. The amount of pipelining can be doubled or tripled to get 8 or 12 π slices, for example. Details of the reservation table will become evident in the following subsections.

4.5.1. Memory Organization

As shown in figure 4-9, the LM-π has three separate memories, one for instructions and two for data; each memory is accessed once per instruction cycle. Arithmetic instructions, therefore, can have two operands, and instructions will be executed at top speed if the operands are fetched from each of the two memories. A multiply and accumulate instruction, therefore, may be written

$$a = a + xop * yop$$

where "a" is the accumulator, *xop* is an X memory operand and *yop* is a Y memory operand.

Without additional pipelining or memory interleaving, the instruction cycle time is a single access time (read *or* write) of the memories. Contrast this with the TMS32010 and TMS32020, which have a read/write cycle in the data memory per instruction cycle, and the DSP 1, DSP 20, and DSP 32 which have two accesses per memory per instruction cycle. For most applications, the performance *per instruction* will be similar to that of the Bell Labs device despite the reduction in the number of memory accesses because of the hardware support for circular buffers, discussed below.

The three memory, single access per instruction configuration has the advantage that higher throughput can be achieved for a given memory bandwidth. Also, strict separation of data memories from program memories means that the word size of the instructions need not match the word size of the data. This permits much more flexibility in the instruction set design.

The address space is assumed to be 64K for the X and Y memories and 16K for the program memory. These figures are somewhat arbitrary, and are easily modified, with consequent hardware cost or benefit.

4.5.2. Registers

The LM-π register set is listed in the following table.

LM-π REGISTER SET		
Name	Length	Description
rx1	16	X memory address register
rx2	16	X memory address register
ry1	20	Y memory address register
ry2	20	Y memory address register
ry3	20	Y memory address register
ry4	20	Y memory address register
pc	14	program counter
ret1	14	return from subroutine address
ret2	14	return from subroutine address
i	5	auto increment/decrement index
b	7	base register for Y memory
lc	8	loop counter
a	40	accumulator
cc	2	condition codes

The total state of an LM-π slice is 216 bits, which is multiplied by the number of π slices to determine the total amount of memory dedicated to storing the processor state.

The two rx registers are used to address the X memory, and the four ry registers are used to address the Y memory. There are more ry registers because the Y memory supports circular buffers, and will be used more because of the SDF programming. The registers are used as in the following example

a = a + (*rx1++)*(*ry1++);

where the notation "*rx1++" means that the "rx1" register provides the operand address and is auto-incremented by one after the operand is fetched. This style of instruction description is borrowed from the Bell Labs DSP 32[Kers85a]. In figure 4-9, a box labeled "delay" accomplishes the interleaving. The delay can be implemented simply as a clocked delay line, although depending on the VLSI process and the size of the register set, it may be more compact as a RAM.

The pc register is the program counter. The two registers ret1 and ret2 store the return addresses of subroutines, which can be nested two deep. The implementation of SDF proposed in the following chapter uses one level of subroutine calls for control flow, and the second level is available to the user. The restriction of the user to one level is not severe because the structure of the program is not oriented around subroutines, but rather around data flow blocks which can be nested arbitrarily.

The index register i allows the user to auto increment or decrement by some number other than one using expressions like "(*rx1++i)". The base register b is used to support circular buffers, discussed below. The loop counter lc simply provides convenient loop control; such registers are popular in DSP architectures. The choice of a 40 bit accumulator assumes fixed point arithmetic with 20 bit precision. Floating point arithmetic can be implemented, as done in the DSP32[Kers85a] for example, with perhaps additional pipeline stages required to support the more complicated arithmetic. We assume fixed point arithmetic only to simplify the discussion.

4.5.3. Arithmetic Unit

The data path of the device is simple. Every arithmetic instruction uses both the multiplier and the ALU. If no multiplication is required, a multiplication by unity is performed; this is the reason for the multiplexers at the inputs to the multiplier in figure 4-9. Similarly, if no ALU operation is required, the product is added to zero. With this in mind, the following arithmetic instructions are supported.

a = a + *xop* * *yop*	Mult. and ALU operation
a = *xop* + *yop*	ALU only
a = a + *op*	ALU only, with a as one operand
a = a + *di*	ALU only, with direct address
a = ALU(a)	Negate, complement, absolute value, etc.
op	Operand only, to do auto-incr. or decr.

In this table, the symbol "+" designates any ALU operation, which could be addition, subtraction, or bitwise logical operations. The symbol *xop* is an X memory operand, "(*rx1++i)" for example, *yop* is a Y memory operand, and *op* is either. The symbol *di* represents a direct memory address for either the X memory or the Y memory. An instruction consisting of only an operand simply performs the auto-increment or decrement specified by the operand.

4.5.4. Data Transfer

An instruction set is not complete without data transfer instructions. The following instructions are supported.

op = *reg*	register to memory
di = *reg*	register to memory
reg = *op*	memory to register
reg = *di*	memory to register
reg = *reg*	register to register
reg = *data*	immediate data to register

To be able to transfer immediate data from instruction memory to a register in one cycle, we must assume that the instruction word is significantly wider than the data word. We have made this assumption, but in a hardware implementation this may be undesirable, in which case a direct memory address should be used to load immediate data. Instructions that would be useful but cannot be supported in a single instruction cycle without augmenting the pipeline include

memory to memory transfers. In this architecture, such transfers must go through a register.

A barrel shifter is provided for transferring any portion of the 40 bit accumulator onto the 20 bit busses. Let the symbol *sha* denote an operand such as "a << 2", which represents the accumulator shifted to the left two bits, or "a >> 5", which represents the accumulator shifted right five bits. Then instructions using the barrel shifter are listed below.

op = *sha*	shifted a to memory
di = *sha*	shifted a to memory
reg = *sha*	shifted a to register
a = a + *sha* * *yop*	shifted a to multiplier
a = a + *xop* * *sha*	shifted a to multiplier
a = *sha* + *yop*	shifted a to ALU
a = *xop* + *sha*	shifted a to ALU
a = a + *sha*	shifted a to ALU

The reservation table shows that an instruction uses the barrel shifter in its second cycle, at the same time it is being decoded. This permits instructions like "*op* = a << 2" because the shift operation is complete before memory cycles become available to the π slice. This arrangement implies that the barrel shift operation must begin before the instruction is decoded. This is not a problem, because a shift operation can be started and if decoding determines that no barrel shift operation is required, the result is discarded.

4.5.5. Control Flow

The only control flow instructions we use in our examples below are subroutine call instructions and unconditional branches, but a full complement of conditional and unconditional branch instructions are easily supported. Note

from the reservation table that the decode hardware has the condition codes available from the previous ALU operation in the same π slice, so conditional branches are no more difficult than unconditional branches. Hardware for low overhead branching, like that in the Motorola DSP56000[Moto86a] is becoming increasingly popular, and should probably also be supported.

4.5.6. Addressing Modes

We have mentioned direct addressing, designated by the symbol *di*, and indirect addressing, designated by the symbols *xop* *yop* and *op*. Indirect addresses use either the X registers or the Y registers with four types of auto-increment/decrement, listed in the following table, which uses the register rx1 as an example.

rx1	no increment/decrement
rx1++	increment by one
rx1--	decrement by one
rx1++i	increment by the contents of the i register

To support synchronous data flow programming, however, it is also necessary to efficiently support circular buffers. Circular buffers require an addressing mode where the auto-increment/decrement is modulo the length of the buffer. For subtle reasons that become clear in the next chapter, it is advantageous that all the information about a circular buffer be contained in one word in memory or one register. Unfortunately, this was not done in the Motorola DSP56000. This device easily supports large circular buffers, as long as there are not many. We need to support many small circular buffers. The mechanism we describe supports only small buffers, but a practical architecture would

probably require some support (perhaps less efficient support) for larger buffers. In the LM-π, modulo-mode addressing is supported in the Y registers, thus confining the circular buffers to the Y memory. Using circular buffers should not slow a program at all. One bit (the high order bit) in the Y address registers is reserved to indicate whether the register is used to access a circular buffer. If not, then the low order 16 bits are used directly as a memory address. Otherwise, the bits of the register are divided into four fields, shown in the following table.

Symbol	Field Name	Field Length	Description
c	circular	1 bit	When set, address circular buffer
s	start	9 bits	Start position of buffer (with b reg.)
f	offset	5 bits	Current position in buffer
m	mod	5 bits	Length of buffer

All fields are unsigned. The total length of the registers is 20 bits. The registers can be initialized using the assembler mnemonic **circ**, as shown in the following example,

ry1 = **circ**(*start*,*offset*,*modulo*);

The right hand side is converted into a 20 bit constant by the assembler with the high order bit set.

Suppose that register ry1 has a one in the c bit position. Then an operand like "(*ry1++)" results in the address

High order 8 bits	Low order 8 bits
b	s + f mod 256

where b is the base register. The offset field is then incremented by one, modulo the contents of the mod field,

$$f_{n+1} = f_n + 1 \bmod m.$$

One possible hardware configuration to support this is shown in figure 4-10. The divider is a nontrivial circuit, but is small because of the small number of bits in its inputs. Suitable divider circuits are discussed in[Hwan79a]. Existing signal processors that implement modulo-mode addressing restrict the mod to be a power of two, obviating the need for a divider[Mint83a, Shiv82a, Qure84a]. However, to support synchronous data flow programming without squandering memory, it is deemed important to support more general modulo-mode addressing.

Circular buffers of length greater than 32 are not supported in the LM-π. If longer buffers are desired, they must be implemented by the user or made by cascading shorter buffers. Details of how these circular buffers are used are given in the following chapter.

4.5.7. Asynchronous Protocols

In chapter 2 it is argued that most communication between π slices will be completely synchronous; a compiler determines when data from one slice will be ready to be used in another slice and ensures that the slice programs are appropriately synchronized. Sometimes, however, a program in a slice may have a data dependent execution time, or may put a data dependent number of samples into a circular buffer. To support this, the LM-π architecture has a semaphore mechanism in the Y memory. An extra bit indicates whether a memory location is full or empty. The bit is set with every write to a location, and is only reset by one of the following instructions:

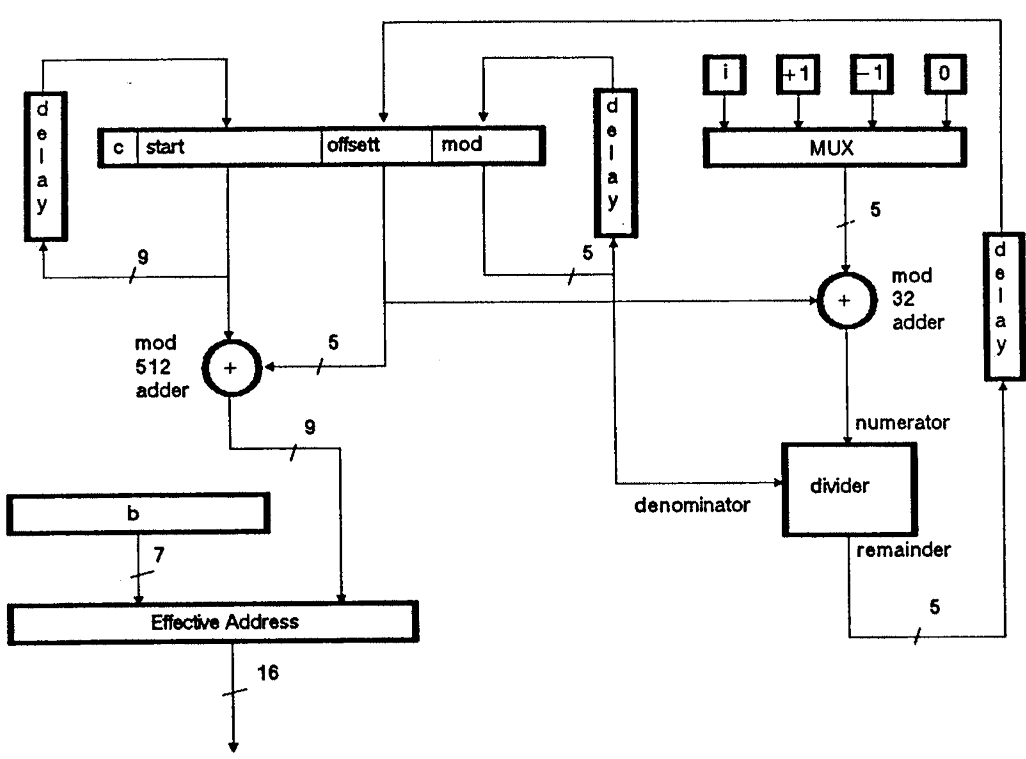


Figure 4-10: Hardware support for modulo-mode auto-increment/decrement.

empty(*di*);

empty(*yop*);

To conditionally access a memory location, the expressions

waitf(*yop*)

waitf(*di*)

can be used instead of *yop* in arithmetic or data transfer instructions. The π slice will repeat the memory access until it reads a *full* memory location. Obviously, this should only be used if another π slice or I/O circuit is expected to fill the memory location. To wait until a memory location is *empty* before proceeding, the following instructions are used

waite(*yop*);

waite(*di*);

The next instruction presumably writes into this memory location.

Often the entire π processor should be synchronized to the inputs or outputs. If instruction count is used to synchronize π slices, then uncertainty in the timing of I/O operations can be fatal. To prevent this, the expressions

synf(*yop*)

synf(*di*)

can be used to halt *the whole machine* (except I/O hardware) until a location is full, or the instructions

syne(*yop*);

syne(*di*);

can be used to halt the whole machine (except I/O hardware) until a location is empty. These instructions can only be used in connection with I/O, discussed in the next subsection.

4.5.8. Input and Output

The I/O mechanism for the LM-π must match the requirements of the SDF programming. Furthermore, a single set of I/O ports should be accessible to all π slices. Also, it must be easy to interconnect multiple π processors.

The LM-π has two serial ports, one for input and one for output. For now we assume that samples are clocked into an input buffer **ibuf** by external circuitry, and outputs are shifted out whenever the internal output buffer **obuf** is written. The mnemonic **ibuf** may be used in place of any Y operand, and the mnemonic **obuf** in place of any destination. To synchronize to external inputs, use the expression **synf(ibuf)**, which causes the machine to halt until the input buffer is written. The expression **empty(ibuf)** then marks the input buffer empty. The following two instructions write an input sample to the location pointed to by ry1,

```
*ry1 = synf(ibuf);
empty(ibuf);
```

The full or empty status of the input buffer is available to outside circuitry, thus completing the synchronization. Synchronizing to outputs is accomplished with instructions like

```
syne(obuf);
obuf = *ry1;
```

This mechanism is adequate for many synchronous applications, but often some form of asynchronous I/O is required. In particular, synchronous data flow programming is considerably simplified if an autonomous I/O mechanism writes to and reads from buffers in memory without interfering with the execution of the program. Two mechanisms are under investigation. The first is interrupt driven I/O in which the interrupt service routine writes input data into a circular buffer or collects output data from a circular buffer. The user interface to I/O is through these buffers. Synchronization is accomplished using semaphores in the Y memory. It is not obvious, however, how to implement interrupts in a π processor. Five methods for interrupting non-interleaved pipelined processors are given by Smith[Smit85a]. A method suitable for the π uses an extra π slice that is created when an interrupt occurs. Its state (all the registers) simply get inserted into the interleaved instruction stream when it gets activated, and get put aside otherwise. It runs only to serve interrupt requests. Execution in all π slices is slightly slower when the interrupt slice is active, but they remain synchronized to one another.

A second I/O alternative is a DMA circuit that steals unused cycles in the Y memory to write input data into a circular buffer or read output data from a circular buffer. This method is probably more complicated, but it has the advantage that the π slices are not slowed at all by I/O operations. A more common DMA mechanism requires the CPU to steal unused cycles from the DMA circuit, giving I/O priority, but with such an approach, the π slices will be slowed somewhat by I/O. The difficulty with our approach is the need to be sure for each program that unused cycles occur with sufficient frequency to reliably manage the I/O. This is guaranteed to be the case if a program is

synchronized to the I/O and must wait for each new sample to be written into the memory before proceeding. The user interface is identical to that in the interrupt approach.

One possible DMA mechanism for bit serial I/O in the LM-π is shown in figure 4-11. Two address registers (which are not replicated for each π slice), designated ri and ro, specify where the next input or output sample, respectively, should be read or written in the Y memory. The read or write occurs whenever a cycle of the Y memory is not used by any one of the slices. Normally, these registers address circular buffers, so they take the same format as Y registers addressing circular buffers. A base register designated iob is used instead of the slice dependent base registers used when Y registers address circular buffers. The program simply accesses a designated buffer for input data and writes to a designated buffer for output data. Double buffering and synchronization become particularly simple. Other details of the I/O mechanism are not important to this thesis.

4.5.9. Exceptions

Interestingly, in the current generation of monolithic DSP architectures, automatic exception handling is almost entirely absent. There is no mechanism for automatically managing arithmetic overflows or pointer limit overflows, for example. The programmer may insert code to test for such conditions, of course, if the condition can cause a critical error. The main reason for this is probably that monolithic DSPs usually operate without a human user, so that it is not clear what an exception handling routine would be able to do that would be useful.

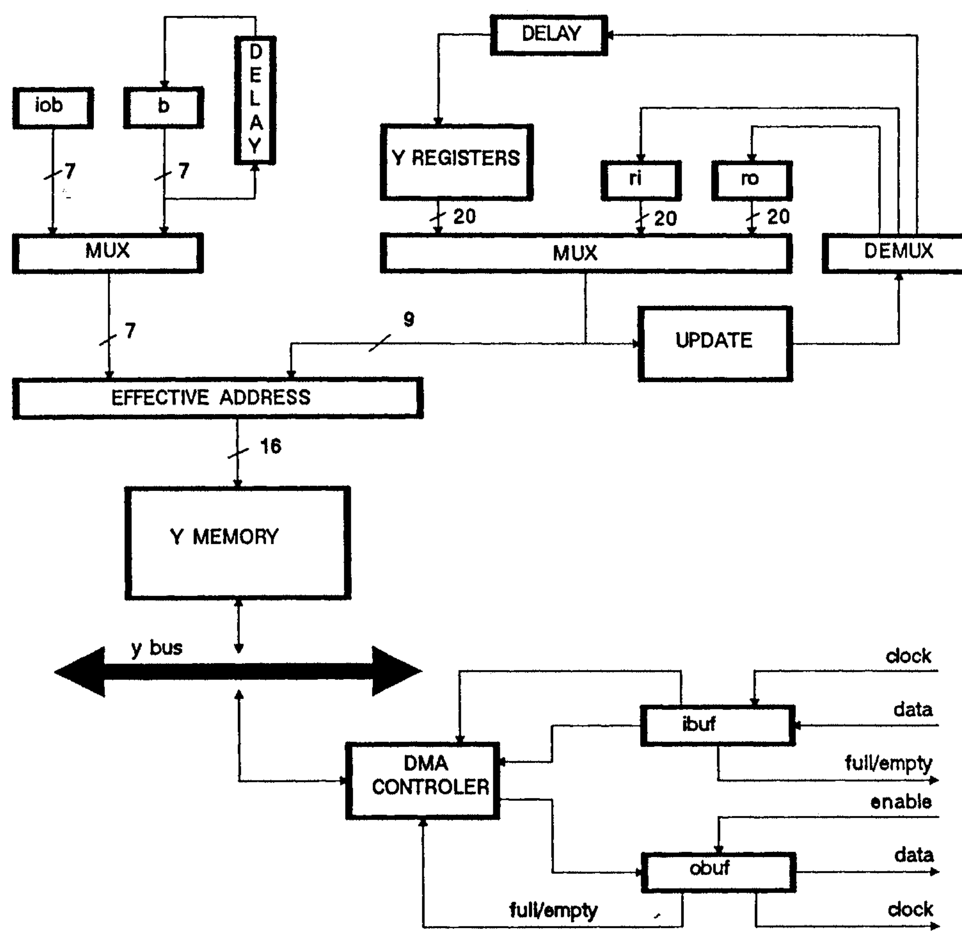


Figure 4-11: One possible DMA I/O scheme.

4.6. PROGRAMMING EXAMPLES

In this section two important examples, FIR filters and IIR filters, illustrate the programming of the LM-π. These programs are a straightforward use of the architecture, without SDF programming. The synchronization of π slices and passing of data between them is much simpler when synchronous data flow programming is used because a compiler takes care of the details.

4.6.1. FIR filters

As an programming example, consider a 16-tap FIR filter implemented on a four slice LM-π processor as shown in figure 4-12. Each slice will implement four of the taps. A single 16-tap delay line is implemented with a circular buffer initialized as shown in figure 4-13. So slice 1 is responsible for the newest samples and slice 4 for the oldest. Logically, the first slice is responsible for collecting input and the fourth slice is responsible for disposing of the output. The program for each slice is divided into an initialization phase and a main loop. The program for the first slice is shown in the following table.

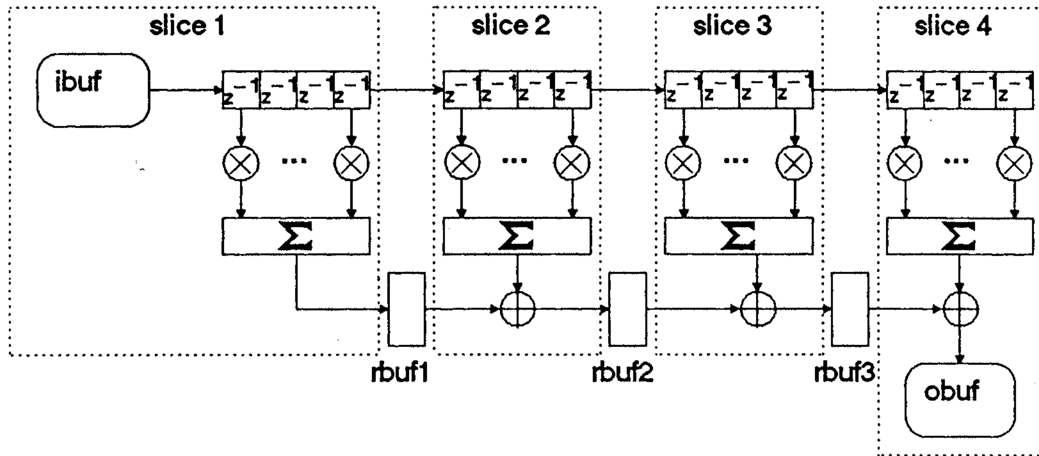


Figure 4-12: A 16-tap FIR filter implemented in four slices. The delay line is a single circular buffer shared by the slices.

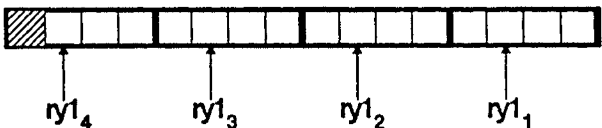


Figure 4-13: The delay line for the 16-tap FIR filter example is implemented as a single, shared circular buffer initialized as shown. The shaded cell is empty and will be the one written to. The subscripts indicate which slice the registers belong to.

16-Tap FIR filter (slice 1)		
	label	**instruction**
1	start:	ry1 = **circ**(*delay_line*,13,16);
2		ry2 = &*rbuf1*;
3		rx1 = **circ**(*coef1*,0,4);
4		i=-2;
5	loop:	a = (*rx1++)*(*ry1++);
6		a = a + (*rx1++)*(*ry1++);
7		a = a + (*rx1++)*(*ry1++);
8		*ry1 = **waitf(ibuf)**;
9		**empty(ibuf)**;
10		a = a + (*rx1++)*(*ry1++i);
11		*ry2 = a;
12		goto loop;

The initialization phase is the first four instructions. The first instruction sets up the pointer to the delay line. The second instruction sets ry2 to point to *rbuf1*, the result buffer for the first slice. The third instruction sets rx1 to point to a circular list with four filter coefficients. Instructions 5-7 multiply three data samples by filter coefficients. Instructions 8 and 9 collect an input, halting *this slice only* until an input appears, and instruction 10 multiplies that input by a filter coefficient. Instruction 11 writes the accumulated result to the *rbuf1* buffer, and instruction 12 restarts the loop. Notice that after one pass through the loop, rx1 points to beginning of the coefficient list and ry1 points to the location one higher in the delay line than before.

The second and third π slices have similar programs.

16-Tap FIR filter (slice 2)		
	label	**instruction**
1	start:	ry1 = **circ**(*delay_line*,9,16);
2		ry2 = &*rbuf2*;
3		ry3 = &*rbuf1*;
4		rx1 = **circ**(*coef2*,0,4);
5		i=-2;
6	loop:	a = (*rx1++)*(*ry1++);
7		a = a + (*rx1++)*(*ry1++);
8		a = a + (*rx1++)*(*ry1++);
9		a = a + (*rx1++)*(*ry1++i);
10		a = a + **waitf**(*ry3);
11		**empty**(*ry3);
12		*ry2 = a;
13		goto loop;

On the first pass through the loop, instruction 10 will cause this slice only to wait until memory location *rbuf1* is written into by the first slice. Thereafter, no wait occurs because the loop has the same number of instructions (8) as the loop in the first slice. Synchronization could also have been done by padding the code with **nops**, but in this case it is essential that the first slice use **synf(ibuf)** instead of **waitf(ibuf)** so that all slices are halted when the first slice has to wait for an input. The example in the next subsection is synchronized this way, to illustrate the technique of *instruction-count synchronization*. The third π slice is virtually identical, so its code is not shown.

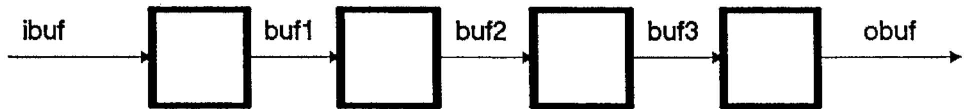
The fourth π slice writes to an output buffer, but we assume that the machine is synchronized to the input, and outputs are simply written when ready. The program is

16-Tap FIR filter (slice 4)		
	label	instruction
1	start:	ry1 = circ(*delay_line*,1,16);
2		ry3 = &*rbuf3*;
3		rx1 = circ(*coef4*,0,4);
4		i=-2;
5	loop:	a = (*rx1++)*(*ry1++);
6		a = a + (*rx1++)*(*ry1++);
7		a = a + (*rx1++)*(*ry1++);
8		a = a + (*rx1++)*(*ry1++i);
9		a = a + waitf(*ry3);
10		empty(*ry3);
11		obuf = a;
12		goto loop;

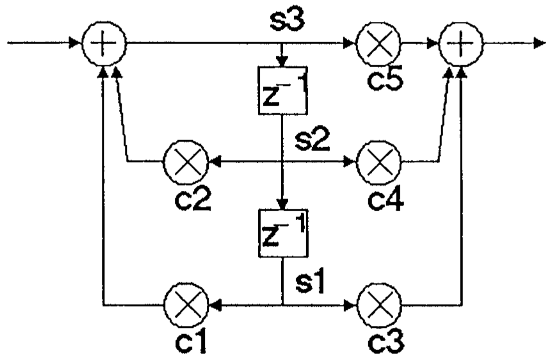
If the filter order is greatly increased, the percentage of time spent on overhead decreases, of course.

4.6.2. IIR Filters

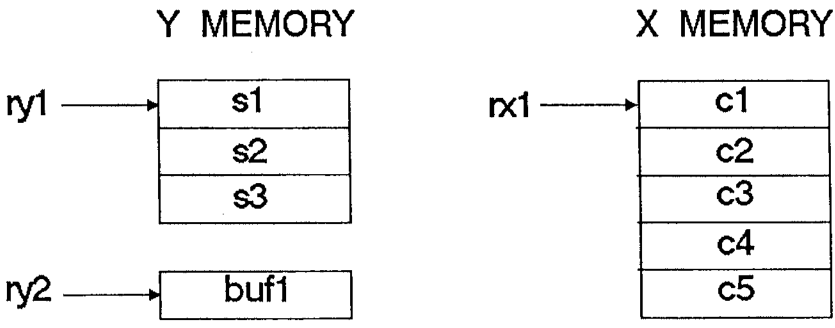
As a second programming example, consider putting four cascaded IIR biquad filters into each of four LM-π slices. In this case, we will use instruction count synchronization. That is, the whole machine is synchronized to the input, but relative to one another, the π slices are running open loop. The cascade of four biquads is illustrated in figure 4-14(a) with the buffers used to pass data between them named. A detail of a biquad is shown in figure 4-14(b) with the coefficients labeled *c1*, ... ,*c5* and the state variables labeled *s1*, ... ,*s3*. The variable *s3* is actually a temporary, not a state variable. The coefficients and state variables are implemented using circular buffers, initialized for the first π slice as shown in figure 4-14(c). The code for the first π slice is



(a) Four cascaded biquads



(b) One of the biquads



(c) Memory allocation and initial pointer locations.

Figure 4-14.

Biquad in the first slice			
	label	**instruction**	**comment**
1	start:	ry1 = **circ**(*s1*,1,3);	buffer starting at *s1*
2		ry2 = &*buf1*;	output buffer
3		rx1 = **circ**(*c1*,0,5);	coefficient list
4	loop:	a = (*rx1++)*(*ry1++);	c1*s1
5		a = a + (*rx1++)*(*ry1++);	c2*s2
6		a = a + **synf(ibuf)**;	synchronize to the input
7		**empty(ibuf)**;	mark ibuf empty
8		*ry1++ = a;	write to s3
9		a = (*rx1++)*(*ry1++);	c3*s1
10		a = a + (*rx1++)*(*ry1++);	c4*s2
11		a = a + (*rx1++)*(*ry1--);	c5*s3
12		*ry2 = a;	write the output
13		**goto** loop;	

Note that after one pass through the loop, rx1 points to c1 and ry1 points to s2. This effectively accomplishes the sample shift in the delay line.

The second slice contains similar code. For clarity, we have identified the state variables and coefficients using the same names as in the first slice.

Biquad in the second slice			
	label	instruction	comment
1	start:	ry1 = circ(*s1*,1,3);	buffer starting at *s1*
2		ry2 = &*buf2*;	output buffer
2		ry3 = &*buf1*;	input buffer
4		rx1 = circ(*c1*,0,5);	coefficient list
5		**nop**;	
6		**nop**;	
7		**nop**;	
8		**nop**;	
9		**nop**;	
10		**nop**;	
11	loop:	a = (*rx1++)*(*ry1++);	c1*s1
12		a = a + (*rx1++)*(*ry1++);	c2*s2
13		a = a + *ry3;	collect the input
14		*ry1++ = a;	write to s3
15		a = (*rx1++)*(*ry1++);	c3*s1
16		a = a + (*rx1++)*(*ry1++);	c4*s2
17		a = a + (*rx1++)*(*ry1--);	c5*s3
18		*ry2 = a;	write the output
19		**nop**;	
20		**goto** loop;	

The **nops** in the initialization are required because the input to the biquad is read in the seventh working (non-**nop**) instruction that the second π slice executes, but is written by the first π slice in the 12th instruction. Thus, six **nops** are required to ensure that the input isn't read until the 13th instruction. An alternative approach would be to allow the second π slice to read garbage (or a zero sample) from the input buffer on the first time through the loop. This is equivalent to pipelining the biquads. Note that the sequence of **nops** can be made more compact with a repeat (**rpt**) instruction like that used in the TMS-32020. Unlike in the TMS-32020 architecture, however, there is no other advantage to a repeat instruction. (The TMS-32020 architecture saves memory cycles by using the repeat instruction, and actually puts those memory cycles to work). The extra **nop** in the loop is to make the loop the same length as the

loop in the first π slice, to ensure that the π slices remain synchronized.

The third π slice has code very similar to the second. The fourth is only slightly different.

	Biquad in the fourth slice		
	label	**instruction**	**comment**
1	start:	ry1 = **circ**(*s1*,1,3);	buffer starting at *s1*
2		ry3 = &*buf1*;	input buffer
3		rx1 = **circ**(*c1*,0,5);	coefficient list
4		**nop**;	
		...	
22		**nop**;	19 nops total
23	loop:	a = (*rx1++)*(*ry1++);	c1*s1
24		a = a + (*rx1++)*(*ry1++);	c2*s2
25		a = a + *ry3;	collect the input
26		*ry1++ = a;	write to s3
27		a = (*rx1++)*(*ry1++);	c3*s1
28		a = a + (*rx1++)*(*ry1++);	c4*s2
29		a = a + (*rx1++)*(*ry1--);	c5*s3
30		**obuf** = a;	write the output
31		**nop**;	
32		**goto** loop;	

If we compare the performance of the four slice LM-π to an unpipelined processor with otherwise the same architecture, we get nearly four times the throughput. If we compare it with an equivalent pipelined processor with no interleaving, we benefit from being able to write code that does not suffer because of the pipelining. An important advantage is that the code we have illustrated for four π slices will work unmodified if there are eight, where the other four can be dedicated to other tasks or to increasing the order of the filter implemented. However, as this example illustrates, dividing a task into parallel programs to run on the π slices is not as easy as we would like. Synchronization must be carefully controlled. It is for this reason that SDF programming is

so important in this application.

4.7. THE COLUMBIA ARCHITECTURE

The architecture proposed by Cohn[Cohn83a] at Columbia in his PhD thesis is particularly interesting. Cohn also uses pipelining and interleaving, and considers the programmability of the machine, but because of the general purpose nature of the proposed machine, the techniques he proposes are quite different. He describes a Generalized Machine (GEM) that can allegedly do what any other SPIN (Sequential Processors & Interconnection Network) machine can do, and is apparently incarnate in the CHoPP (Columbia Homogeneous Parallel Processor). It is a shared memory system with an omega network interconnection between the processors and the memories. There are several important aspects to the GEM architecture. It is programmed using a large grain data flow paradigm with dynamic control, but each block in the data flow graph is actually a process. Corresponding to each block is a full set of registers, the entire state of a processor slice. This has significant advantages for dynamic data flow control because a block can be suspended at any time due to insufficient data on its input buffers, and when it is restored, the entire processor state is restored. Furthermore, each processor is interleaved, so data flow blocks are scheduled onto slices of processors. The important details of the architecture are outlined below. (Note that most of the acronyms below, VP, PS, & OS are mine, not Cohn's).

4.7.1. Conflict-Free Memory System.

This is a pipelined omega network in which each node recognizes simultaneous accesses to identical memory locations and combines the accesses into one, so no delay occurs. Simultaneous accesses to different memory locations in the same memory module still cause delays, but a hashing of memory addresses reduces the frequency of such conflicts. This is a rather complex and expensive patented memory scheme. But it is not clear that it is worth the cost.

4.7.2. Virtual Processors (VP).

This is Cohn's term for processes with interdependencies (data flow blocks). Each VP has a register set stored in main memory (or cache) called a *state vector*. Cohn assumes there are an infinite number of VPs available, that each one runs until it gets blocked (needs a result from another VP that has not been computed), and that a list is maintained of runnable but not running VPs. The naive scheduling, Cohn argues, never costs more than a factor of two in efficiency[Grah69a], rarely costs that much in an actual application[II83a], and doesn't actually cost anything in a general purpose computing environment where user's jobs can be intermingled, the supply of jobs can be deemed infinite, and the efficiency is measured over months, not milliseconds. Finally, each VP can run on any PS (see below).

4.7.3. Processor Slices (PS)

These are interleaved processes like those in the π processor used to compensate for latency in the pipelined memory accesses. One VP that can be run is an Operating System (OS) VP associated with each processor slice to handle I/O

requests or the management of blocked VPs. Thus, the OS VP in each slice does the dynamic scheduling. Each processor slice has three hardware registers associated with it: a register pointing to the current VP state vector in memory, a one bit register to specify whether the current VP is running or the operating system VP is running, and a register through which a VP can send messages to the OS VP. The state vector of the OS VP resides at a prespecified memory location. One way of handling the scheduling is to let the OS VP select the next VP from a list of runnable VPs when the current one gets blocked. The list of runnable VPs is maintained by separate hardware associated with each processor (processing element with a set of PSs) to run in the background maintaining the scheduling lists. It will ideally always have on hand a state-vector address for the next VP to run when the current one gets blocked. Each PS has its own cache to reduce traffic on the network. However, each PS can access the cache of each other PS on the same PE.

4.7.4. VP Synchronization

The synchronization and communication between VPs is done via "blackboards" of which there are two types. The first type is a list of runnable but not running VPs. If VP1 produces data that it knows is needed by VP2, then it can put VP2 on the list. If VP2 actually also needs data from another process, it can run until it gets blocked again by trying to access that data. This blackboard is easily fractured into many smaller blackboards; the OS VP selects a blackboard at random, until it finds one unlocked, and locks it while it accesses it. The fracturing reduces conflicts. The second kind of blackboard gets associated with a particular event that can block other VPs. A list of VPs blocked by

this event is stored here. Any VP depending on this event checks this blackboard to see if the event has occurred, and if not, blocks, and adds its own state vector address to the blackboard. Cohn says a waiting queue is needed for this blackboard in the event of simultaneous accesses.

4.7.5. Comments on Relationship with the π Processor

The blackboard method of synchronization would, in principle, work with the π processor, but it requires relatively expensive run-time support. In a general purpose computing environment this is appropriate, because the distinction between run-time and program development is not as drastic as in the π processor context, where every effort is made to eliminate all run time overhead due to the interleaving. The synchronous nature of most signal processing programs makes such overhead unnecessary, as shown in the previous chapters, and their computational demands makes it imperative.

Cohn's interleaving requires less memory to replicate the state because each slice only has three hardware registers. These registers point to the processor state in memory. However, the disadvantage of this approach is that the memory used to store the processor state for all the VPs could easily become the speed bottleneck in the system. A final disadvantage to Cohn's approach makes it ill suited for hard real-time applications. The use of caches and conflict-free memories afford only probabilistic speedup, and hard real-time applications require performance in the worst case.

4.8. CONCLUSIONS

We have proposed a pipelined and interleaved (π) approach for the design of high performance programmable DSPs. Deep pipelining is used to gain maximum hardware advantage, but the problems generally associated with such pipelining are avoided by making the pipelining invisible to the programmer. Instead, the programmer sees a set of parallel processors (π slices) that share memory. An implementation of synchronous data flow (SDF) programming is described in the next chapter.

The main hardware cost of interleaving is the additional registers required to replicate the processor state. However, these registers can be implemented as dynamic shift registers, which require considerably less VLSI area than randomly placed registers implemented as sets of flip-flops. The main hardware advantage is the ability to increase the amount of pipelining without exacerbating the programming. Additional hardware features that have been identified as helpful to support SDF programming are

- modulo auto-increment/auto-decrement addressing modes (for circular buffers);
- data independent execution times for all instructions; and
- memory semaphores with a full/empty discipline (for asynchronous systems).

A specific architecture has been outlined that contains these features.

5

PROGRAMMING A π PROCESSOR USING SDF GRAPHS

In the last chapter we use a pipelined and interleaved (π) approach for the design of high performance programmable DSPs. Deep pipelining is used to gain maximum hardware advantage, and the problems generally associated with such pipelining are avoided by making the pipelining invisible to the programmer. Instead, the programmer sees a set of parallel processors (π slices) that share memory. A specific architecture, designated the LM-π, is outlined in that chapter. In this chapter, programming of the LM-π using the synchronous data flow paradigm is discussed. Thus, this chapter should serve to weld the theoretical results of chapters 2 and 3 with the architecture of chapter 4.

The implementation of synchronous data flow requires

(1) a way to specify the topology of an application (the SDF graph);

(2) a way to specify the functions associated with each block;

(3) a systematic method for mapping these specifications onto π slices or other parallel processor architectures.

The SDF graph can be generated using a graphical interface[Hait85a], making prototyping DSP systems that use only standard blocks particularly easy. In this chapter we assume that the code defining the function of a block is written in the assembly language of the π processor. In section 5.1, the mechanics of buffering data between SDF blocks is considered. The design of a suitable high level language and graphical interface is briefly considered in section 5.2. In section 5.3, a voiceband data modem example is considered in detail.

Implementing the signal processing system described by a SDF graph requires *buffering* the data samples passed between blocks and *scheduling* blocks so that they are executed when data is available. Our goal is a *compiler* that translates block definitions and a SDF graph into efficient sequential code for a parallel processor. Note that our compiler is not translating a high level language into machine code, although this could ultimately be part of its function.

It begins by scheduling using the techniques in chapter 2 and proceeds with code generation. Criteria for the *correctness* of a SDF graph are given in chapter 2. In particular, for a graph to be correct, it must have a *periodic admissible schedule*. A periodic schedule is said to be *admissible* if the amount of data in the buffers remains bounded and non-negative with infinite repetition of the schedule. A graph does not have a periodic admissible schedule if it has inconsistent sample rates or directed loops with insufficient delays. Necessary and sufficient conditions for correctness of a SDF graph are given. A broad class of

algorithms designated *class S* (for sequential) algorithms will find a periodic admissible schedule if one exists, solving the single processor scheduling problem. The π processor is a multiprocessor, however. For the multiprocessor case, a class S algorithm is given in chapter 2 for translating a SDF graph into an *acyclic precedence graph* for one or more periods of a periodic schedule. Such a graph can be used to construct blocked schedules for multiple processors. Indeed, given such an acyclic precedence graph, the block scheduling problem reduces to the well studied assembly line problem. General schedule length minimization algorithms are *NP-complete*, but a large family of *critical path methods* offer simple heuristics that can be shown to perform extremely well. These methods give preference to the path through the precedence graph with the most computation.

Mainly by means of examples, the practical aspects of code generation are considered in this chapter. The performance of the scheduling algorithm of chapter 2 is evaluated by considering in detail a complicated real-world example, a voiceband data modem. A program (written in Lisp) called *Gabriel* is used to build the examples and implement the scheduling algorithms.

5.1. BUFFERS AND DELAYS

Each arc in the SDF graph corresponds to a buffer that can be implemented as a circular list. Circular lists are easily supported using the modulo addressing described in chapter 4. The size of each buffer depends on the schedule. To see this, consider the trivial system shown in figure 5-1. Assume for simplicity that we are to schedule this system onto a single π slice; the buffering problem is the same in the multiprocessor case. If the schedule is a periodic repetition of

Figure 5 – 1: A trivial synchronous data flow graph describing a four tap FIR filter with input and output.

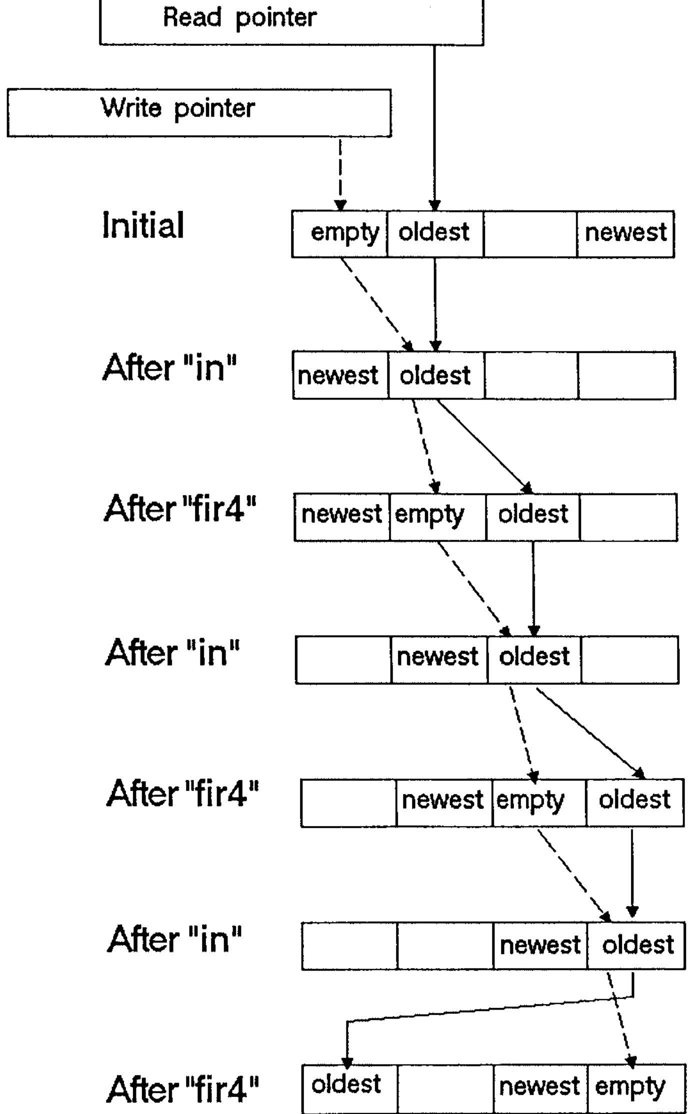


Figure 5 – 2: A buffer can serve as the delay line of an FIR filter. This figure shows how the read and write pointers change as the "in" and "fir4" blocks in Fig. 5 – 1 are repeatedly executed.

the sequence {**in**, **fir4**, **out**}, then the buffers need only store one new sample at a time, and can therefore be of length one. If the schedule is a periodic repetition of {**in**, **in**, **fir4**, **fir4**, **out**, **out**}, then buffers of length two are required.

It is desirable to be able to access not just the new samples in a buffer, but also past samples[Mess84a]. The buffer "buf1" can then be used to implement the delay line of the four tap FIR filter. The programmer therefore does not need to implement data structures that can easily be provided by the circular buffering mechanism.

Each buffer is a set of contiguous memory locations plus two pointers. Both pointers are set up by the scheduler for modulo-mode auto-increment/decrement, as described in chapter 4. The *write* pointer is used by the block that puts data into the buffer. Before this block is invoked, the write pointer should point to the next location to be written (an empty location in the buffer). The *read* pointer is used by the block taking data from the buffer. Before that block is invoked, the read pointer by convention points to the *oldest* sample of interest in the buffer. Thus, in figure 5-1, before invoking **fir4**, the read pointer of "buf1" points to the last sample in the delay line of the FIR filter. The code within a block must increment all read or write pointers that it uses to indicate consumption or production of data samples.

This use of the buffer is illustrated in figure 5-2. A buffer of length four and the schedule {**in**, **fir4**, **out**} are assumed. The first buffer state, labeled *initial*, shows the write pointer pointing to the first location in the buffer and the read pointer to the second. We assume the buffer is initially full of zero samples, so the FIR filter delay line is initialized to contain zeros. Invoking the **in**

block causes a new sample to be written into the first location. The pointer is incremented by one, indicating that one sample has been produced, as shown in the figure. Now we can run **fir4**. The **fir4** block accesses the four samples in the buffer, incrementing the read pointer (modulo four) until it accesses the newest sample. After accessing the newest sample, it decrements the pointer by two so that the net update of the pointer is one, as shown in the third buffer configuration. This procedure can be repeated indefinitely; the periodic schedule ensures the integrity of the data.

Having illustrated the buffering operation, we illustrate the code for the LM-π architecture described in section 4.5 implementing the block **fir4.** The block subroutine expects the buffer pointers to be in pre-agreed registers, and updates the registers before returning. This convention implies that all blocks must have implicit self-loops, because if two successive invocations of a block were to run simultaneously, only one of the blocks will successfully modify the buffer pointers. This limitation is discussed further in the final chapter. The main part of the code is the following subroutine.

```
fir4:   a = (*rx1++)*(*ry1++);        coefficient times oldest data
        a = a + (*rx1++)*(*ry1++);    coefficient times second oldest data
        a = a + (*rx1++)*(*ry1++);    coefficient times third oldest data
        i=-2;                         So that we can auto-decrement by 2.
        a = a + (*rx1)*(*ry1++i);     coefficient times newest data
        *ry2++ = a;                   write to output buffer
        return;
```

This is all the code written by the user defining the block. It assumes that on entry to the subroutine,

(1) the address register ry1 contains the read pointer for the input buffer and ry2 contains the write pointer for the output buffer;

(2) ry1 points to the sample in the input buffer four iterations old;

(3) ry2 points to the next empty location in the output buffer;

(4) rx1 points to the coefficient for the oldest sample.

These assumptions are part of the definition of the block. Hence, in addition to the subroutine above, declarations like the following are required as part of the definition of the block:

> **input** *name*,ry1;
> **output** *name*,ry2;
> **param** coefficients,rx1,4;

The names can be used to distinguish inputs and outputs when there are more than one of each. Notice that the definition of the block is not affected by the length of the buffers, and therefore is independent of the schedule. This same block can be used in the SDF graph shown in figure 5-3, where because of the interpolator, a buffer of length four at the input to **fir4** will not work. If the schedule is {**in**, **interp**, **fir4**, **fir4**, **out**, **out**}, then a buffer of length five is required.

To initialize the system, the buffers must be defined by allocating memory and setting the read and write pointer to their initial values. Our compiler, after scheduling to determine the length of the buffers, generates code to initialize the buffers "buf1" and "buf2" in figure 5-1. Suppose that the schedule is {**in**, **fir4**, **out**}. Then the following LM-π code will set up the buffers:

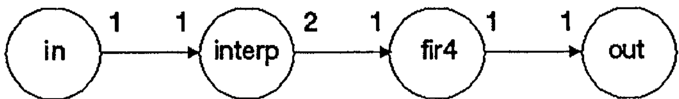


Figure 5 – 3: A simple multiple sample rate system.

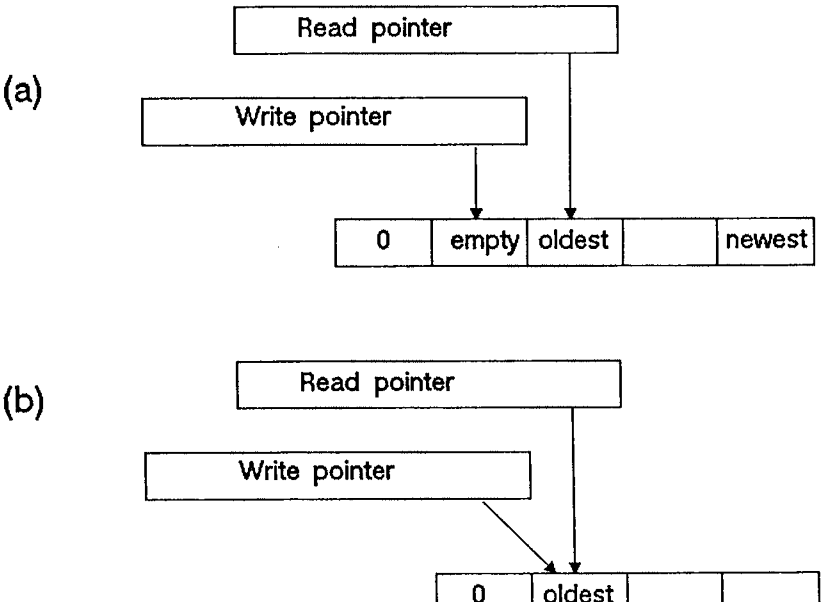


Figure 5 – 4: Two possible ways of initializing a length four buffer when the corresponding arc has a unit delay:

(a) the source block will run first, or

(b) the destination block will run first.

```
buf1:         ram(4);                  Pseudo-op to allocate four memory locations.
buf1_write:   ram(1);                  Pseudo-op to allocate one memory location.
buf1_read:    ram(1);                  Pseudo-op to allocate one memory location.
buf2:         ram(1);                  Pseudo-op to allocate one memory location.
buf2_write:   ram(1);                  Pseudo-op to allocate one memory location.
buf2_read:    ram(1);                  Pseudo-op to allocate one memory location.

              a = circ(buf1, 0, 4);    Point to first location of length 4 buf.
              buf1_write = a;          Write to buffer write pointer location.
              a = circ(buf1, 1, 4);    Point to second location of length 4 buf.
              buf1_read = a;           Write to buffer read pointer location.
              a = circ(buf2, 0, 1);    Point to length 1 buf.
              buf2_write = a;          Write to buffer write pointer location.
              a = circ(buf2, 0, 1);    Point to length 1 buf.
              buf2_read = a;           Write to buffer read pointer location.
```

This code is generated by the compiler, and executed only once. Notice that this code can be trimmed down by observing that the read/write pointers to a length one buffer are always the same. We will ignore such obvious optimizations for now.

When the program is running and collecting input samples, to invoke the block **fir4**, the registers rx1, ry1, and ry2 must be set. Each invocation of **fir4** therefore proceeds as follows (this code is also generated by the compiler):

```
ry1 = buf1_read;                  Set read pointer.
ry2 = buf2_write;                 Set write pointer.
rx1 = circ(coefficients,0,4);     Set parameter pointer.
call fir4;                        Call the subroutine.
buf1_read = ry1;                  Update the read pointer.
buf2_write = ry2;                 Update the write pointer.
```

The two memory writes at the end are required to record the number of samples consumed or produced. Actually, because "buf2" has unity length, the last write is not required, but we will assume the compiler blindly ignores such optimizations; we therefore get a worst-case bound on the overhead introduced

by the programming method.

Each invocation of the block **fir4** requires the six instructions above generated by the scheduler plus the seven instructions of the **fir4** subroutine, for a total of 13 instruction cycles. This is considerably more than the seven instruction loop in the four tap FIR filter example described in chapter 4, but, as with virtually all high level programming methodologies, the advantages of the method only become evident when the system gets relatively complicated. We describe the voiceband data modem example in the next section.

In chapter 2, it is observed that delays are managed quite simply in SDF. A delay is a property of the arc connecting two blocks. That is, if there is a unit delay on the arc connecting block **A** to block **B**, then the nth sample consumed by **B** will be the $(n-1)^{th}$ sample produced by **A**. The first sample consumed by **B** is therefore not produced by A at all, but is rather part of the initial state of the buffer. *A delay on an arc is exactly equivalent to an initial sample on the arc*, implying that delays are simply introduced in the initialization of the buffer.

Consider again the example in figure 5-1. A unit delay on the arc from **in** to **fir4** implies that **fir4** can be invoked before **in**. Indeed, on a single processor, we now have three possible schedules with unit blocking factor, {**in**, **fir4**, **out**}, {**fir4**, **in**, **out**}, and {**fir4**, **out**, **in**}. Depending on whether the scheduler selects to run **fir4** before **in**, the buffer will be initialized in one of the two states shown in figure 5-4, where the zero indicates the initial sample put there by the compiler.

Having illustrated the mechanics of buffering, we consider the interface by which a system designer specifies an implementation.

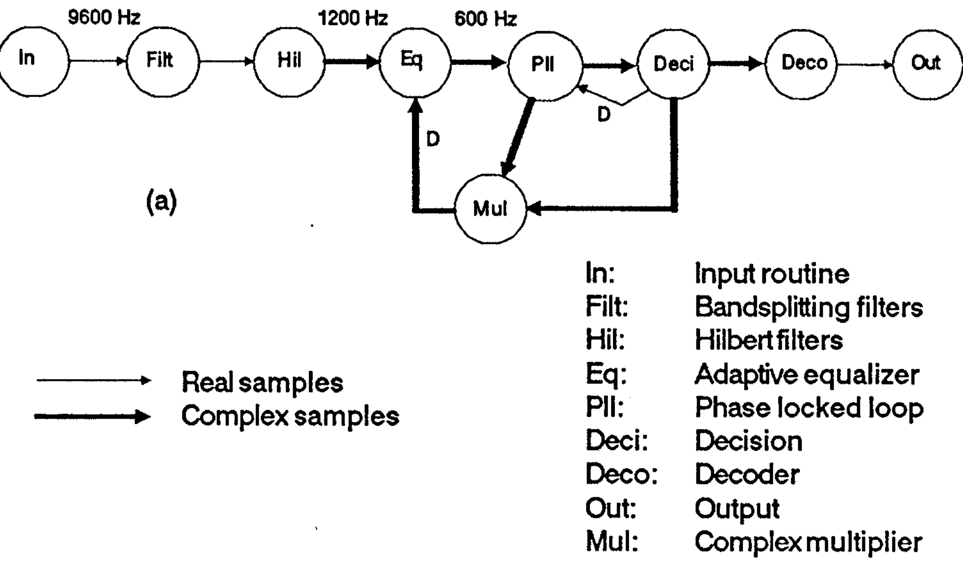
5.2. LANGUAGE DESIGN

To retain the appeal of a block diagram specification of a DSP application, the granularity of the SDF description must be flexible. For this reason, we divide the language problem into three parts, *topology definition*, *block definition*, and *function definition*.

5.2.1. Topology Definition

The is the outermost layer of the user interface, and therefore must be the easiest to use and the simplest. The graphical nature of the block diagrams suggests a graphical interface. A prototype graphics interface running on Sun workstations for the large grain data flow simulator called BLOSIM is under development[Hait85a]. A similar interface is suitable for SDF. Standard blocks in a library are represented by icons which are interconnected using a mouse. A system that can be constructed only with standard blocks can therefore be rapidly prototyped. Furthermore, the time required to learn to use the system at this level should be negligible, allowing users to immediately construct useful programs. Most programming will be done at this level, so the topology definition level must be carefully and aesthetically designed, with extensive use of menus and help screens.

A voiceband data modem block diagram is illustrated in figure 5-5(a). The sample rates are shown explicitly, and complex signal paths are illustrated with bold lines. The equivalent SDF graph is shown in figure 5-5(b). Complex



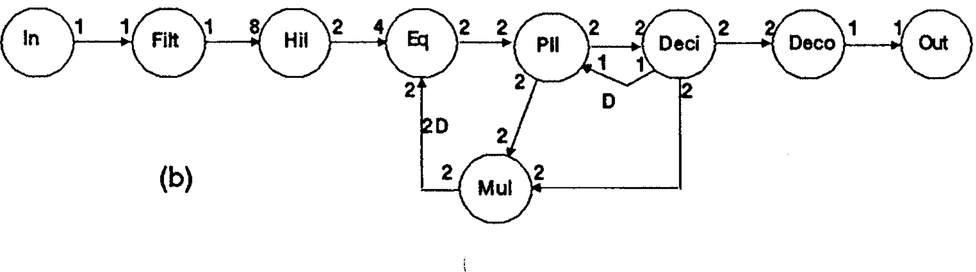


Figure 5 – 5: (a) A block diagram of a 2400 bit per second, 600 baud modem. Three sample rates are evident. The first, 9600 Hz, is Nyquist rate samples of the data bearing signal, the second is twice baud rate samples feeding a fractionally spaced equalizer, and the third, 600 Hz, is the baud rate. (b) A SDF graph description of the same modem.

numbers are communicated between blocks as two successive samples, so many of the blocks produce and consume two samples on each input and output. A suitable graphical interface would permit the user to begin with the construction of this graph, initially defining each node to have a null function, so that the application can be built top-down. A block may then be selected and specified in greater detail. For example, the PLL block is expanded in figure 5-6, where icons are used to represent blocks that are likely to be part of a standard library. The biquads, adder, multiplier, and complex conjugate are easily recognized from the icon. Two two-way forks are shown; they simply replicate each input sample on each output path. The **sin&cos** block computes a polynomial approximation to the sin and cosine (a table lookup could be used instead). The next level of detail is the block definition.

5.2.2. Block Definition

Inevitably, even a rich set collection of standard blocks will not adequately meet the needs of all applications. Serious users will have to define their own. A block has the following features:

- inputs and outputs;
- parameters, which are constants that may be different in distinct instances of a block (for example, the coefficients of a second order section are parameters);
- state variables (variables local to each instance of the block);
- a run-time routine, or a program segment that takes samples from the inputs and produces samples on the outputs; and
- an initialization routine, if one is required.

Each input and output definition gives the input or output signal a symbolic name, so that it can be referenced in the code by name, and specifies the number of samples produced or consumed in each invocation of the block. Also specified

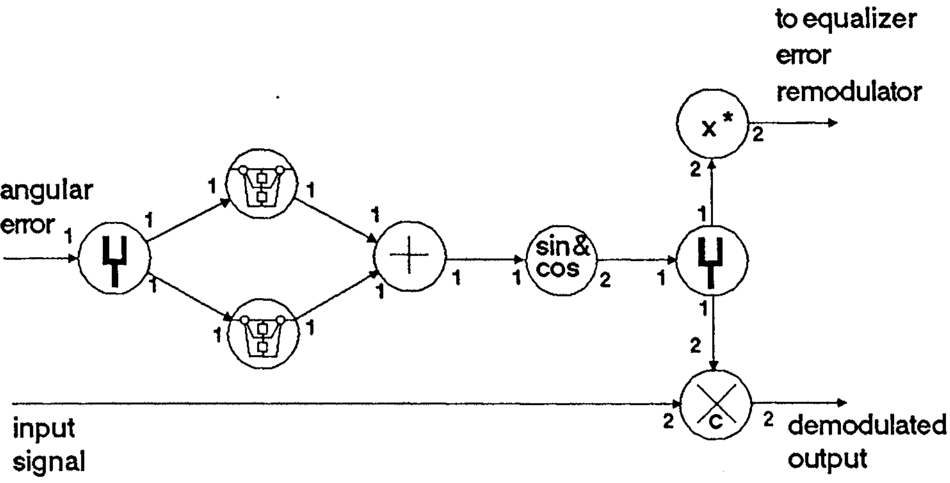


Figure 5 – 6: Detail of the PLL block in the voiceband data modem. The next level of detail is assumed to be assembly language.

is the number of past samples required so that input and output buffers can be used as tapped delay lines. For example, an FIR filter requires the current input sample plus N past samples, where N is the order of the filter. Finally, for efficient scheduling, the scheduler needs to know the run time of each block. It may be possible to obtain much information automatically from the function definition.

One possible syntax for specifying this information is based on LISP and has been implemented in an experimental synchronous version of BLOSIM called *Gabriel*. A second order section is given by

```
(def_block biquad                      ; Give the block a name.
   (input in)                          ; Define the input with name "in".
   (output out)                        ; Define the output with name "out".
   (param coefs [default values])      ; Filter coefficients.
   (state fred 3)                      ; The state "fred" is a length 3 ...
                                       ; circular buffer.
   (function biquadsub)                ; Name of run time subroutine.
   (runtime runtime)                   ; Execution time of run time subroutine
)
```

Gabriel keywords are shown in bold type, names in roman type, and values to be supplied in italics. Information that is lacking, such as the number of samples consumed on the input named "in" defaults to the most common value, unity. The only part of the block definition remaining is the definition of the subroutine called "biquadsub". This is deliberately considered a separate issue to maintain independence at the block level from the language used to program the target processor. Definition of the subroutine is *function definition*.

5.2.3. Function Definition

We assume throughout this chapter that the run-time subroutine corresponding to a block is defined in the assembly language of the LM-π. An obvious alternative is a high level language, preferably one specialized to signal processing. An interesting possibility is an applicative language called *SILAGE*[Hilf84a]. The applicative language paradigm shares some features with data flow; in particular, much of the inherent concurrency in an algorithm is evident, so it may be practical to exploit concurrency inside blocks defined this way. Inherent in the SDF paradigm, of course, is the ability to have *mixed mode* implementations, where different blocks in the same SDF graph are defined in different languages. For greatest efficiency, the most commonly used blocks could be written in assembly language. When the program structure of a block is dependent on a parameter of the block, a code generator may be desirable. For example, and FIR filter with a parameter specifying the order of the filter may be best implemented using in-line code for small filters and a loop for large filters. A code generator could make this decision. A fixed order FIR filter block has been illustrated already for the LM-π.

5.3. A VOICEBAND DATA MODEM EXAMPLE

Just as pipeline bubbles degrade the performance of a pipelined processor, scheduling imperfections will degrade the performance of a synchronous data flow program on a π processor. The amount of degradation depends on the amount of concurrency in the graph. Smaller granularity will generally lead to enhanced concurrency, but smaller granularity may also require more overhead. The optimal tradeoff is application dependent. Algorithms with feedback loops

can also limit the amount of concurrency available, as shown in chapter 3.

To verify the efficacy of the programming methodology, we examine here a real application, a voiceband data modem, rather than relying on standard DSP benchmarks. The standard benchmarks, IIR and FIR filters and FFTs, are not a good test of the programming method. Its benefits are most evident in complicated, relatively unstructured applications. We selected the modem application because we have previously implemented and fully tested such a modem using the Bell Labs DSP-20. By using essentially the same algorithms, we avoid the need for real-time testing to verify the validity of the implementation. Since the LM-π has not been implemented in hardware, real-time testing is not possible.

Figure 5-5 illustrates the SDF graph describing a 2400 BPS voiceband data modem. The full detail is shown in figure 5-7. Although the programmer is unlikely to ever wish to see the whole modem at this level of detail, for the purposes of exposition, it is instructive to depict the complete complexity of the application. An inventory of the blocks is given in figure 5-8, with the second column indicating which blocks are likely to be part of a standard library. The run time, measured in instructions, for each block includes the setup time, the subroutine call, and the subsequent writes. The run time for a four tap FIR filter would therefore be 13. The run times represent only one of many possible implementations and can undoubtedly be improved, but the specific numbers are not as important as the knowledge that we are testing a real (as opposed to contrived) practical example. The numbers are computed by translating the DSP20 code for the modem into code for the LM-π. We expect similar performance for

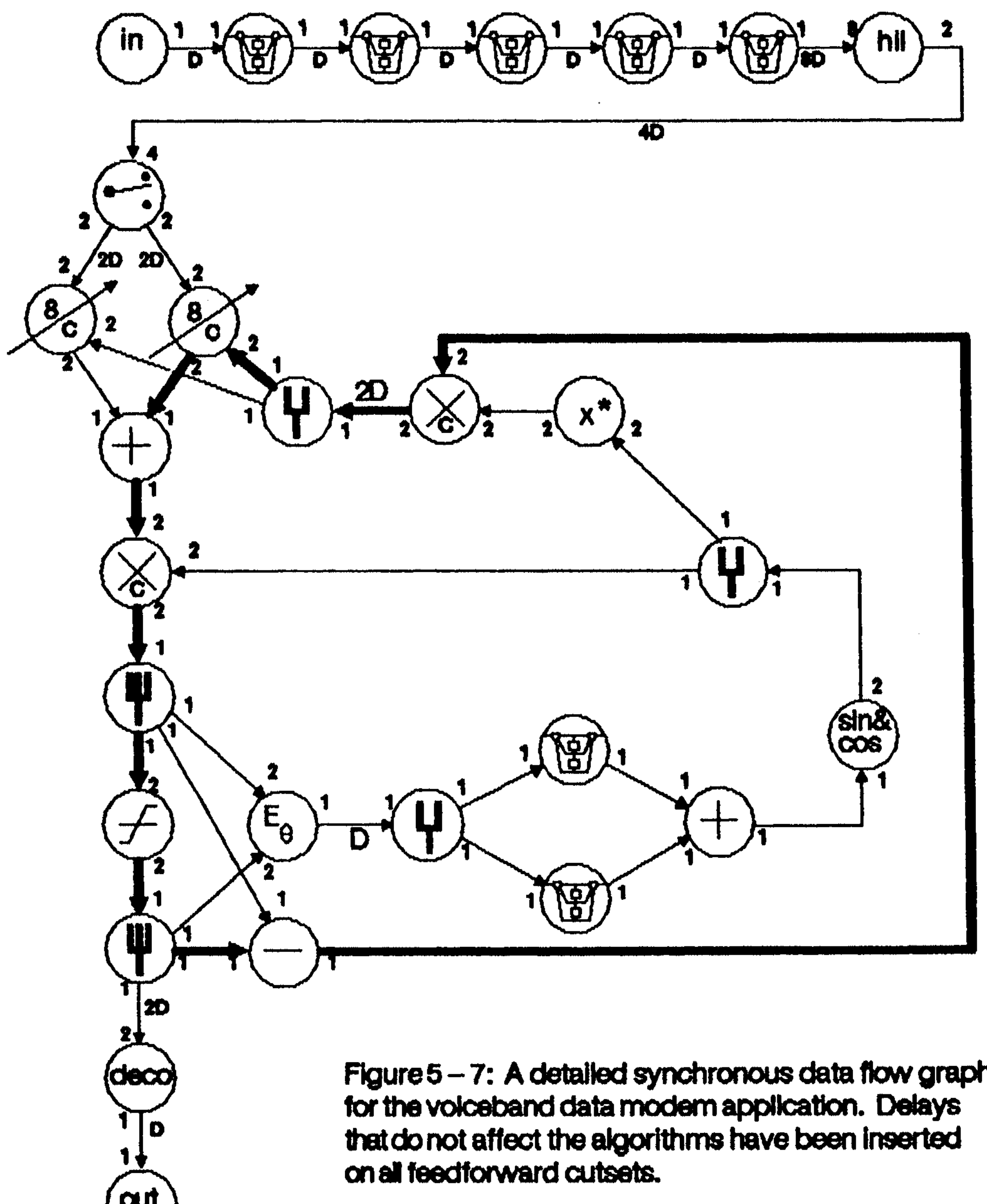


Figure 5 – 7: A detailed synchronous data flow graph for the voiceband data modem application. Delays that do not affect the algorithms have been inserted on all feedforward cutsets.

Symbol	Standard?	Function	Run Time
in	yes	Input a sample from outside the system	5
	yes	Biquad	16
hil	no	Hilbert filters	43
	yes	Distributor (alternate outputs)	16
8 c	yes	8 tap complex adaptive filter	110
+	yes	adder	11
c	yes	complex multiplier	16
	yes	3 way fork (copy the input on all three outputs)	16
	yes	2 way fork	11
	no	Slicer (modem decision device)	16
deco	no	Decoder (convert a symbol into bits)	25
out	yes	Output a sample to the outside	5
—	yes	Subtractor	11
E_θ	no	Angular difference between complex inputs	14
sin& cos	yes	Compute the sin and cos of the input	71
x^*	yes	Conjugate a complex input	11

Figure 5 – 8: An inventory of blocks used in the voiceband data modem. The second column indicates whether the block is likely to be standard in a programming system.

different implementations.

The total amount of computation is 2062 instructions in one period. Notice that one period processes 16 input samples and produces one output sample (which in this case contains four bits of interest). Using the methods described in chapter 2 we constructed the acyclic precedence graph, and a schedule for various numbers of processors. With no additional delays, beyond those in the feedback loops in figure 5-5, the length of the critical path in the acyclic precedence graph is 642 instructions for unity blocking factor. Hence, the minimal schedule period is 642. The periods achieved for various numbers of processors are listed in the following table. Also listed is the percent of available processing resources used by the schedule.

Scheduling Results		
No feedforward cutset delays, and no extra loop delays		
Minimal period is 642 instruction cycles		
No. Processors	Period	% Utilization
2	1105	93%
3	817	84%
4	716	72%
5	669	62%
6	643	53%

The utilization figure should be interpreted cautiously. It is the utilization of the processors during one period of a periodic schedule, and does not reflect any idle time that may occur between periods. Such idle time occurs because the throughput of the modem is determined not by the speed of the processors, but rather by the real time constraints of the modem. Nonetheless, the utilization figure is an indication of how many processors can be usefully used, which in turn tells us that the modem algorithms can be run at a higher rate or slower

processors can be used.

For five or more processors, the period is within 5% of the minimal schedule period. For five processors, however, the processor utilization is a mediocre 62%.

The processor utilization and/or the iteration period for a fixed number of processors can be improved by using a blocking factor greater than unity. A more effective method is to put delays in the feedforward cutsets, as discussed in chapter 3. Figure 5-7 shows such delays. With these delays, the performance of the scheduler is dramatically improved. The periods achieved for various numbers of processors are listed in the following table. Also listed is the percent of available processing resources used by the schedule.

Scheduling Results		
Some feedforward cutset delays, and no extra loop delays		
Minimal period is 277 instruction cycles		
No. Processors	Period	% Utilization
1	2062	100%
2	1031	100%
4	520	99%
6	348	99%
7	303	97%
8	282	91%
10	280	74%
12	277	62%
14	277	53%

The minimum schedule period is reduced from 642 to 277, for unity blocking factor. For seven or more processors, the achieved period is within 5% of this minimum period. In figure 5-9 we show the schedule for eight processors.

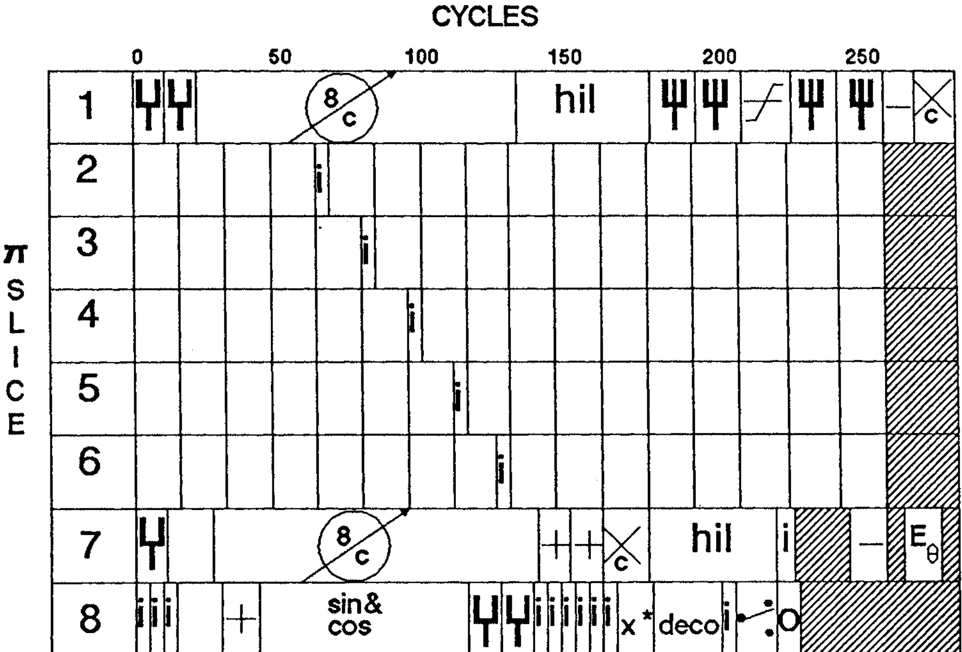


Figure 5 – 9: An 8 processor schedule for a voiceband data modem. The shaded areas are idle, the unlabled boxes are biquads, the boxes labled "i" are inputs, and the boxes labled "o" are outputs. All other symbols are those used before. The total period is 282 cycles and the utilization is 91%.

In figure 5-10, the processor utilization as a function of the number of π slices is shown. Again, the utilization figure should be interpreted cautiously. Essentially full utilization is maintained with up to seven π slices. Recall from chapter 4 that the computation per unit time of the π processor increases somewhat less than linearly with the number of π slices, suggesting that the maximum throughput is achieved for some number of slices greater than seven.

It is not obvious how much delay to put on each feedforward cutset. In figure 5-7, only enough delay is added to be able to run each block once at the beginning of the schedule. To decouple a block completely from its predecessor, however, more delay than this may be required. Each biquad in the front end bandpass filter is invoked 16 times in one period. Thus to completely decouple the second biquad from the first, a delay of 16 samples is required between them. This is costly in memory, however, because it amounts to double buffering the data between successive blocks. We found that with this maximal delay on feedforward cutsets, the schedule performs no better than with the delays in figure 5-7. This follows because the with the delays of figure 5-7, the iteration bound is quickly reached.

The iteration bound, found using the techniques of chapter 3, is 277. The critical loop is the tap update loop, shown in bold in figure 5-7. For more than seven processors, this bound is closely approximated with unity blocking factor and only the feedforward cutset delays of figure 5-7. Larger blocking factors will not improve the schedule period. Furthermore, there is no benefit from *retiming*, a technique discussed in chapter 3 of moving the delays around to try to reduce the length of the critical path for a fixed block size.

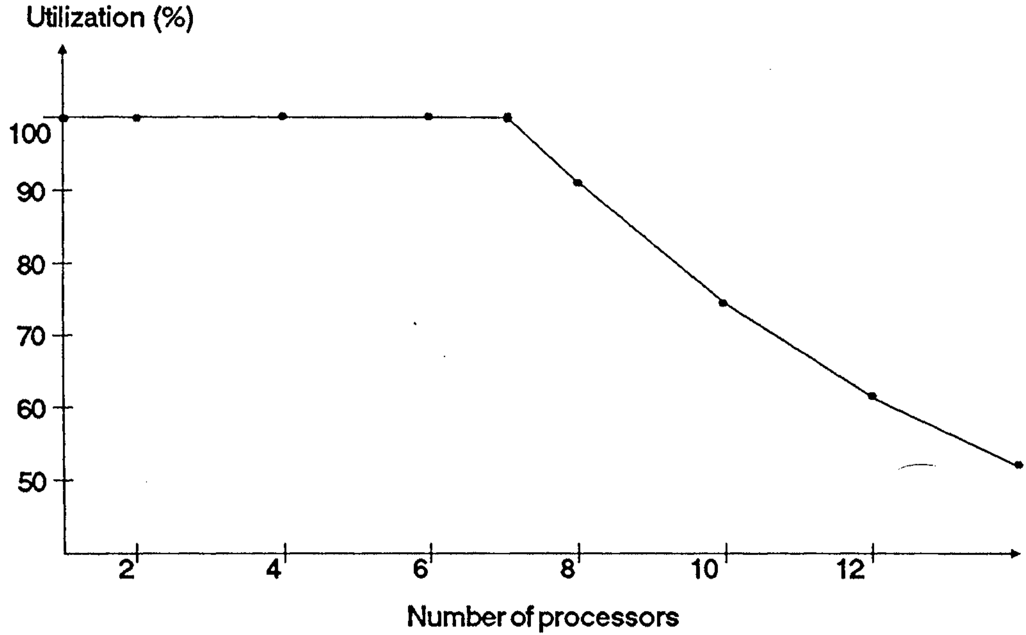


Figure 5 – 10: Processor utilization vs. the number of processors for the voiceband data modem.

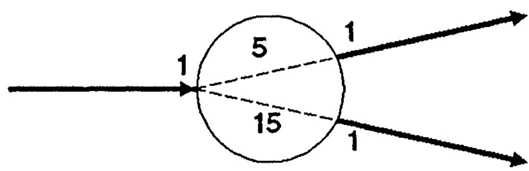


Figure 5 – 11: Partial run times are the number of processor cycles after consuming an input before a node produces an output. The example shown here has partial run times 5 and 15. The Hu level scheduling algorithm can use this information to improve the schedule.

An obvious way to reduce the iteration bound would be to increase the delay in the critical loop. This alters the algorithm, but it has been observed that up to about three samples of delay can be inserted in the tap update loop of such an equalizer without measurably affecting the performance of the modem[Falc82a]. In this case, the position of any extra delays is important, if unity blocking factor is to be used. Putting simply a two sample delay on each arc out of the adaptive filter blocks, we were able to reduce the critical path to 256 instruction cycles. The critical loop becomes the 16 iterations of each biquad, which has a computation time of 16. More delays will not help. To get the iteration bound lower, we need to relax the implicit self loops of the biquads, requiring much more elaborate techniques. Such techniques would lower the iteration bound to 241, determined by the demodulation loop. The demodulation loop is not likely to tolerate additional delays without performance degradation of the modem. Even a heroic effort to bring the iteration bound down, therefore, will only lower it 13%, from 277 to 241. Of course, the 241 figure can be reduced by using a table lookup instead instead of a polynomial approximation for the sine and cosine, assuming enough memory is available.

The iteration bound could be further lowered by introducing *partial run time dependencies*, illustrated in figure 5-11. In that figure, a node with one input and two outputs computes one of the outputs significantly earlier than the other, implying that any node waiting for the earlier output can be started earlier than implied by the model with only one run time. In the figure, a run time of five is associated with the upper output and a run time of 15 with the lower. Such a generalization is easily managed by the Hu level scheduling

algorithm. The only difficulty is in the human interface; the more information required about a block, the more tedious defining a block will be, and specifying a set of run times may be more trouble than it is worth. This objection, of course, becomes invalid if the information can be extracted automatically by a compiler.

5.4. CONCLUSIONS

The programming of the architecture of chapter 4 under the synchronous data flow paradigm has been described, with one elaborate example used to illustrate the efficacy of the technique. A voiceband data modem is shown to be naturally described using SDF, and an SDF description exhibits enough concurrency to keep seven π slices completely busy, assuming the modem sample rate is the maximum that the implementation can handle. The Hu level scheduling algorithm of chapter 2 is shown to construct schedules that closely approximate the theoretical iteration bound imposed by the structure of the algorithm.

Some aspects of the human interface for SDF programming have been discussed. The advantages of SDF are

- it is natural for digital signal processing;
- it is modular and hierarchical;
- mixed-mode description is easily supported;
- the same description can used for simulation and implementation;
- concurrency is explicit;
- concurrency can be automatically enhanced; and
- multiple sample rates can be supported.

In addition, extensions to support limited asynchrony are possible.

The main disadvantage of a synchronous data flow programming paradigm is that processing resources may be lost due to scheduling inefficiencies. This is a fundamental problem with parallel implementations. The amount of the loss is dependent on the granularity and structure of the application.

Hardware features that have been included in the LM-π to support this type of programming are

- modulo auto-increment/auto-decrement addressing modes (for circular buffers);
- data independent execution times for all instructions; and
- memory semaphores with a full/empty discipline (for asynchronous systems).

It should be noted that the implementation of these features described in chapter 4 is well suited to the voiceband modem application, but may not be as well suited to other applications. For example, only circular buffers of length less that or equal to 32 are supported efficiently. Of course, it can be argued that short buffers are by far the most common, so some inefficiency can be tolerated in implementations requiring longer buffers. Also, the limit to 32 is a consequence of the 20 bit wordsize. An additional limitation is imposed by the convention that buffer pointers are passed to the SDF block subroutines in registers. The number of registers is necessarily small because of the requirement that every register be replicated for each π slice. Therefore, blocks with many input or output paths will require extra overhead to fetch and store buffer pointers. This scenario does not arise in the voiceband modem example, because no block has more than a total of four inputs and outputs.

6

FURTHER WORK

Synchronous data flow is a natural and appealing paradigm for the specification of DSP algorithms for implementation. Algorithms are described as block diagrams and can be efficiently mapped onto single or parallel hardware. The programmer need not be concerned with synchronization of the parallel processors, communication, scheduling, or deadlock avoidance. Given such a programming paradigm, parallel architectures become much less formidable. The π processor architecture takes advantage of this using a pipelined interleaved structure. Extensive pipelining is used for efficiency, and multiple programs are interleaved in the pipeline so that instead of programming a single deeply pipelined processor, the user programs a set of virtual parallel processors. An SDF compiler takes care of the more difficult details of such programming.

Although much theory and design has been developed, the work in this thesis probably suggests more new problems than it solves. This section is dedicated to cataloguing these problems for the benefit of those who wish to explore

further. Some of the further work suggested centers around implementation of the ideas described in the rest of the thesis, and some is almost entirely new ground. Frequently, in this work we avoided the temptation to pursue optimality questions and chose instead to propose a practical system solution, so several of the open questions center on optimization.

The large number of open questions is a consequence of limited time. We divide the open questions into four categories, *fundamentals*, *hardware*, *software*, and *applications*. A section is dedicated to each. The final section is an admission that the most challenging task remains undone: implementation of the π processor in hardware, and a practical, usable SDF programming system.

6.1. FUNDAMENTALS

The theory of synchronous data flow developed in this thesis can probably be expanded considerably, increasing the range of applications. Four fundamentally new areas are mentioned in this section, *asynchrony*, *real-time constraints*, *non-homogeneous parallel processors*, and *silicon compilation*.

6.1.1. Asynchrony

The solid theoretical framework of synchronous data flow suggests that certain types of asynchronous systems can perhaps be handled efficiently. Put another way, a middle ground may exist between the generality of data flow and the determinism of SDF. The basic mechanism we propose is to divide a data flow graph into synchronous subgraphs. Synchronous subgraphs are either disconnected or connected by cutsets where all arcs in the cutset represent an asynchronous link. Disconnected SDF graphs are a trivial form of asynchrony

that is handled easily by scheduling the pieces of the graph onto disjoint sets of processors. This may not always be possible, however.

Synchronous subgraphs connected by asynchronous links are also easily handled if the synchronous subgraphs can be scheduled onto disjoint sets of processors. Asynchronous communication between processors is required. In chapter 4, a full-empty discipline in memory is proposed for the LM-π precisely so that such cases can be easily managed. This solution suggests that asynchronous systems are more easily implemented on a π processor than on a conventional single-process DSP chip. The most obvious approach is to construct maximal throughput schedules for each synchronous subgraph, taking as many processors as required, but this solution is not likely to be efficient. It is unlikely that all synchronous subgraphs can run at top speed without waiting for data from other synchronous subgraphs. Furthermore, if the number of processors is limited, deciding on the number of processors devoted to each synchronous subgraph is difficult. Without information about the relative timing of synchronous subgraphs, we cannot do anything systematic.

In many applications, however, an asynchronous arc may not be completely general. A signal processing system with asynchrony may arise when separate subsystems are controled by different clocks, but the clock rates may match within a close tolerance, so the asynchrony may not be completely general. While we cannot specify the exact number of samples consumed or produced on an asynchronous arc, it may be possible to specify a range. For example, a block may consume one, two, or three input samples, making the range one to three. Given a range, bounds on the relative sample rates of synchronous

subgraphs can be computed, and schedules that are well matched can be constructed, in principle.

A more difficult problem arises if the synchronous subgraphs have to be scheduled onto the same processors. One approach is to construct static schedules for each of the synchronous subgraphs and dynamically schedule the subgraphs. Again, it is not clear how many processors each synchronous schedule should use. A synchronous subgraph can be invoked only when it has input data *and* processors are available to run it. Minimizing the makespan of the synchronous subgraph by using multiple processors results in complicated processor requirements each time the subgraph is invoked, making dynamic scheduling more difficult. Some work has been done on scheduling tasks that require more than one processor[Blaz86a], but more needs to be done in the SDF context.

6.1.2. Real-Time Constraints

The scheduling method described in chapter 2 concentrates on maximizing the throughput subject to a constraint on the number of processors. This criterion makes some sense for fixed hardware topologies, such as the π processor, where the entire system is to be dedicated to one signal processing task. A more generally useful criterion, however, is to satisfy a real-time constraint while minimizing the amount of hardware dedicated to the task. Optimal throughput is not usually necessary, albeit theoretically interesting. This new criterion changes the scheduling problem. The new problem can be solved by iteratively increasing the number of processors until the real-time constraint is satisfied. The Hu level scheduling algorithm is sufficiently simple that for modest

numbers of processors this solution is satisfactory, but more direct methods would be appealing.

6.1.3. Non-Homogeneous Parallel Processors

The π processor has the advantage that each π slice is identical to other π slices. Such a parallel architecture is said to be *homogeneous*. For some applications, more cost effective implementations may use several types of processors. Not all parts of an algorithm require a hardware multiplier, for example, so a realization using a DSP and a microprocessor may be more cost effective than a realization using two (or more) DSPs.

One way to deal with this problem is to specify a set of run times, one for each processor type, for each node in an SDF graph. Unfortunately, the Hu level scheduling algorithm breaks down because the level is not well defined for each node in the acyclic precedence graph until its descendants have been scheduled. This problem is related to problems in flexible manufacturing systems, where a shop with robots is to be used in a manufacturing sequence. Each robot can perform each stage of the assembly with varying degrees of efficiency, and the problem is to schedule the steps in the assembly onto the robots.

6.1.4. Silicon Compilation

Consider a VLSI design system with large parametrized macrocells that can be combined on a chip. An example is Lager[Pope84a] in which each macrocell is a simple programmable processor with variable wordsize. Given a SDF graph with each node specified in a language that can be translated into machine language for the programmable processor, we wish to decide how many

processors we need to meet some throughput, what the processors' wordsizes should be, and what the interconnection topology needs to be. We also need a schedule, of course. The problem can be generalized so that a set of non-homogeneous processor types are available; for example, one processor type may have a hardware multiplier and one may have bit serial arithmetic. The fundamental task is to select the processors and construct a schedule such that some throughput constraint is met and the cost (VLSI area) is minimized. This is also related to the flexible manufacturing systems problem mentioned in the previous section, except that we have not yet bought the robots.

6.2. HARDWARE

The only completely reliable proof of the π architecture is probably a hardware prototype. This suggests substantial further work to realize a π processor as a chip. Unfortunately, such a project is large enough that an academic environment may not be the right setting for it. A breadboard made with commercial parts, however, may be adequate proof, or a fast simulation made with dedicated processors. Aside from the obvious need to prove the architecture with hardware, several architectural questions remain unanswered.

6.2.1. Indivisible Buffer Increments

In many applications, the implicit self-loop for some node is the critical loop. If the self-loop can be eliminated, faster implementations may be possible.

If the node has no state variables, the self-loop is an artifact of the buffer implementation. In the scheme described in chapter 5, buffer pointers are loaded

into registers (or a pre-agreed memory location) before the node is invoked, the node alters the registers to indicate that samples are consumed or produced, and the registers are stored after the node terminates, becoming the new buffer pointers. This procedure precludes simultaneously running more than one invocation of the same node because if two nodes run simultaneously, only the second one to finish will affect the buffer pointers. An alternative scheme uses *indivisible buffer access operations*. A node directly fetches a sample from a buffer and the buffer pointer is incremented in a single, indivisible operation. No other node (or other invocation of the same node) can access the buffer while this operation occurs. Such a scheme is worth exploring for its impact on hardware complexity and flexibility.

6.2.2. Number of π Slices

In this thesis we have carefully avoided speculation about the optimum number of π slices for maximum throughput. This issue is highly technology dependent, but important. Cappello, LaPaugh, and Steiglitz have studied the problem for certain classes of circuits, and perhaps their conclusions are applicable[Capp84a].

It has been suggested that *variable* numbers of slices may be advantageous. Not all applications will be able to make maximal use of the number of slices provided. Reducing the number of slices below the minimum required to prevent pipelining conflicts means that interlocking or some other mechanism to prevent pipeline hazards must be implemented. Increasing the number of slices implies a small hardware cost to store additional processor states, but may improve the efficiency of a given application, particularly one with considerable

asynchrony.

6.2.3. Memory

It is anticipated that for existing VLSI technology the limitation on the amount of pipelining in a π processor, and hence the limitation on the throughput, will be imposed by the memory. Fast memory is costly, suggesting that methods should be found to pipeline slow memories. An alternative to pipelined memories is interleaved memories, but interleaved memories divide the π slices into clusters, and communication between clusters may be more awkward than communication within clusters. This suggests that if memory cannot be pipelined, and fast memory is too costly, then effective methods for communicating between clusters must be found.

6.3. SOFTWARE

Further work in the software arena divides into three categories, *optimization*, *user interface*, and *application to other architectures*.

6.3.1. Optimization

The scheduling algorithm given in chapter 2 is a heuristic that approximately minimizes the makespan of a single period of a periodic schedule. This is related to, but not identical with, maximizing the throughput. Maximizing the throughput would generally require schedules that are not blocked, such as the cyclo-static schedules of Schwartz[Schw85a]. Such methods should be assessed for complexity and benefit in the context of synchronous data flow.

Many methods remain unstudied for increasing the throughput of an implementation. Perhaps the most promising uses *partial computation times* to overlap execution of nodes with dependencies. This method is discussed in chapter 5 and is illustrated in figure 5-11. *Retiming*, a technique used by Leiserson[Leis83a] to increase the throughput of digital circuits, can be applied to increase the throughput of blocked schedules with a fixed blocking factor. For blocked schedules, an adequate systematic method for finding the optimal blocking factor has not been found. Finally, a systematic method for determining the best amount of delay on feedforward cutsets would be useful.

Most applications, however, do not require maximal throughput. Instead, a real-time constraint must be met for minimum cost. Algorithms performing this optimization should be investigated.

6.3.2. User Interface

This problem is described at some length in chapter 5. Perhaps the most interesting aspect is the design of the graphical interface for topology definition. Workstation technology has reached the point that high resolution bit-mapped graphical displays are accessible to many more designers than before. Another interesting problem is the design of a language for defining the nodes. The *BLOSIM* and *Gabriel* systems are a first attempt. Closely related to this is the design of a language for specifying the functions of the nodes. Ideally, such a language would offer portability among a wide variety of programmable architectures.

6.3.3. Application to Other Architectures

Synchronous data flow is such an appealing paradigm for the description of signal processing processing algorithms that it may be useful for programming general multiprocessor systems. The main challenge is to incorporate in the scheduling algorithms information about the interconnection topology and communication delays. The π processor is a shared memory machine without contention, which is the most desirable interconnection topology. Such a topology is not usually practical for machines built with multiple physically separate processors.

6.4. APPLICATIONS

In chapter 5 we describe one promising application for the π processor and SDF programming. One application, however, does not demonstrate much versatility. The methodology should be investigated in the context of image processing, numerical simulation (e.g. circuit simulation), speech recognition, bandwidth compression, coding, robotics, seismic signal processing, and communication protocol design and implementation. Such investigation, however, requires hardware for implementation; software simulations of the π processor are far too slow for realistic testing of application algorithms. This suggests that the most important immediate challenge we face is bringing the ideas expressed in this thesis to fruition as usable hardware and software techniques.

6.5. THE COUNCIL HELD BY THE RATS

Old Rodilard, a certain cat,
 Such havoc of the rats had made,
 With nature's debt unpaid.

The few that did remain,
 To leave their holes afraid,
From usual food abstain,
 Not eating half their fill;
 No wonder no one will.

That one who made of rats his revel,
With rats pass'd not for cat, but devil.
Now on a day, this dread rat-eater,
Who had a wife, went out to meet her;
And while he held his caterwauling,
The unkill'd rats, their chapter telling,
Discuss'd the point, in grave debate,
How they might shun impending fate.
 Their dean, a prudent rat,
Thought best, and better soon than late,
 To bell the fatal cat;
That, when he took his hunting round,
The rats, well caution'd by the sound,
Might hide in safety underground;

Indeed he knew no other means.
 And all the rest
 At once confess'd
Their minds were with the dean's.
No better plan, they all believed,
Could possibly have been conceived;
No doubt the thing would work right well,
If anyone could hang the bell.
 But one by one, said every rat,
 "I'm not so big a fool as that."
The plan, knock'd up in their respect,
The council closed without effect.

And many a council I have seen,
Or reverend chapter with its dean,
 That thus resolving wisely,
 Fell through like this precisely.

Moral

To argue or refute
 Wise counsellors abound;
The man to execute
 Is harder to be found.

- Aesop c. 550 B.C.

SDF GLOSSARY

data flow graph: A graph where each node represents a function, each arc represents the path of data samples. A node is invoked whenever it has a sufficient number of input data samples to perform its function.

synchronous data flow (SDF) graph: A data flow graph which statically specifies the number of samples produced or consumed on each output or input path of each node each time the node is invoked.

homogeneous SDF graph: A SDF graph where the number of samples produced or consumed on each arc is unity.

sequential schedule (ϕ): A list of nodes. The order indicates in which order the nodes should be invoked on a single processor.

parallel schedule ($\phi = \{\psi_i\}$): A set of sequential schedules ψ_i for a set of parallel processors.

admissible schedule: A sequential or parallel schedule that does not deadlock and keeps buffer sizes bounded.

blocked schedule: An infinite periodic schedule where each cycle must be finished before the next cycle can begin.

PASS (ϕ): A periodic admissible sequential schedule. The list ϕ represents one cycle.

PAPS ($\phi = \{\psi_i\}$): A periodic admissible parallel schedule. The lists ψ_i represent one cycle in each parallel processor.

acyclic precedence graph: The precedence graph for one cycle of a periodic schedule.

critical path: The path through the precedence graph with the greatest total computation time.

blocking factor (J): Admissible periodic schedules have a minimum number of invocations of each node in a cycle. Any periodic admissible schedule invokes each node an integer multiple J of this minimum number in one cycle. The integer multiple J is the blocking factor.

schedule period $S_J(\phi)$: The computation time required for one cycle of a blocked schedule ϕ with blocking factor J.

iteration period ($T_J(\phi)$): The schedule period divided by the blocking factor.

computation rate ($1/ T_J(\phi)$): The reciprocal of the iteration period.

minimum iteration period (T_J): The smallest iteration period over all periodic admissible blocked schedules with blocking factor J.

iteration bound (T_∞): The minimum (over all blocking factors J in the extended positive integers) of T_J. This is the reciprocal of the maximum achievable computation rate.

Renfors and Neuvo bound (T_0): The maximum (over all directed loops) of the total computation time in the loop divided by the number of delays in the loop. It is equal to the iteration bound.

critical loop: A loop achieving the above maximum.

sample period (T): For an implementation of a homogeneous data flow graph, the average amount of time between the production of samples on any arc.

logical delay (D): A property of an arc in a SDF graph specifying an offset in the samples produced and consumed on that arc. A logical delay of ND on an arc means that the Mth sample consumed from that arc is the $M-N^{th}$ produced on that arc. The buffer associated with the arc must be initialized with N initial samples. A logical delay is identical to a z^{-1} operator in signal processing.

shimming delay: The actual *time* delay for each logical delay in an implementation of a SDF graph.

reachable graphs: Non-standard term used in [Schw85] to refer to SDF graphs with directed paths from a unique

starting node to all other nodes.

equivalent graphs: Two graphs with the same nodes (but possibly different arcs) such that any schedule that is admissible for one is admissible for the other.

REFERENCES

Acke82a.

Ackerman, William B., "Data Flow Languages," *Computer* 15(2)(Feb., 1982).

Adam74a.

Adam, T. L., Chandy, K. M., and Dickson, J. R., "A Comparison of List Schedules for Parallel Processing Systems," *Comm. ACM* 17(12) pp. 685-690 (Dec., 1974).

Ager79a.

Agerwala, Tilak, "Putting Petri Nets to Work," *Computer*, p. 85 (December, 1979).

Ahme82a.

Ahmed, Hassan M., Delosme, Jean-Marc, and Morf, Martin, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *IEEE Computer* 15(1)(Jan. 1982).

Alle75a.

Allen, J., "Computer Architecture for Signal Processing," *Proceedings of the IEEE* 63(4)(April, 1975).

AMI,a.

AMI, Inc.,, *Signal Processing Peripheral,* Data Sheet for the S28211

Babb84a.

Babb, Robert G., "Parallel Processing with Large Grain Data Flow Techniques," *Computer* 17(7)(July, 1984).

Barn82a.

Barnwell, Thomas P., Hodges, C. J. M., and Randolf, Mark, "Optimum Implementation of Single Time Index Signal Flow Graphs on Synchronous Multiprocessor Machines," *Proceedings of the Int. Conf. on Acoustics, Speech, and Signal Processing*, (May 3-5, 1982).

Barn83a.

Barnwell, Thomas P. and Schwartz, D. A., "Optimal Implementation of Flow Graphs on Synchronous Multiprocessors," *Proc. 1983 Asilomar Conf. on Circuits and Systems*, (Nov., 1983).

Blah85a.

Blahut, Richard E., *Fast Algorithms for Digital Signal Processing*, Addison-Wesley Publishing Co., Reading, MA (1985).

Blaz86a.

Blazewicz, J., Drobowski, M., and Weglarz, J., "Scheduling Multiprocessor Tasks to Minimize Schedule Length," *IEEE Trans. on Computers* C-35(5)(May 1986).

Braf78a.

Brafman, J. P., Szczupak, J., and Mitra, S. K., "An Approach to the Implementation of Digital Filters using Microprocessors," *IEEE Trans. on Acoustics, Speech, and Signal Processing* ASSP-26(5) pp. 442-446 (Oct. 1978).

Capp83a.

Cappello, Peter R. and Steiglitz, Kenneth, "Completely Pipelined Architectures for Digital Signal Processing," *IEEE Transactions on ASSP* **ASSP-31,** No. 4(August, 1983).

Capp84a.

Cappello, Peter R., LaPaugh, Andrea, and Steiglitz, Kenneth, "Optimal Choice of Intermediate Latching to Maximize Throughput in VLSI Circuits," *IEEE Trans. on ASSP* 32 p. 28 (February, 1984).

Chap81a.

Chapman, R. C., Editor, "Digital Signal Processor," *Bell System Technical Journal* 60(7)(September, 1981). Special Issue, Part 2

Chas84a.

Chase, M., "A Pipelined Data Flow Architecture for Signal Processing: the NEC uPD7281," in *VLSI Signal Processing*, IEEE Press, New York (1984). [NEC Electronics, Inc.]

Cohn83a.

Cohn, Leonard Allen, *A Conceptual Approach to General Purpose Parallel Computer Architecture*, Columbia University, New York (1983). PhD Thesis

Comm71a.

Commoner, F. and Holt, A. W., "Marked Directed Graphs," *Journal of Computer and System Sciences* 5 pp. 511-523 (1971).

Croc75a.

Crochiere, R. E. and Oppenheim, A. V., "Analysis of Linear Digital

Networks," *Proceedings of the IEEE* 63(4) pp. 581-595 (April, 1975).

Crys74a.

Crystal, T. and Kulsrud, L., *Circus*, Institute for Defense Analysis, Princeton, NJ (Dec., 1974). CRD Working Paper

Davi73a.

Davidson, E. S. and Larson, A. G., "Pipelining and Parallelism in Cost-Effective Processor Design," *Res. Rep., Digital Systems Lab., Stanford University*, (1973).

Davi78a.

Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proc. Fifth Ann. Symp. Computer Architecture*, pp. 210-215 (April, 1978).

Dene81a.

Denelcor, Inc.,, *Heterogeneous Element Processor*, Pub. 9 000 001 1981.

Denn80a.

Dennis, J. B., "Data Flow Supercomputers," *Computer* 13(11)(Nov., 1980).

Dert69a.

Dertouzous, M., Kaliske, M., and Polzen, K., "On line simulation of block-diagram systems," *IEEE Trans. on Computers* C-18(4)(April, 1969).

Dipaa.

Dipartimento di Elettronica, Politecnico di Torino,, *TOPSIM III - Simulation Package for Communication Systems - User's Manual*.

Falc76a.

Falconer, D. D., "Jointly Adaptive Equalization and Carrier Recovery in Two-Dimensional Digital Communication Systems," *Bell System Technical Journal* 55(3)(March 1976).

Falc82a.

Falconer, D. D., "Adaptive Reference Echo Cancellation," *IEEE Transactions on Communications* COM-30(9)(Sept. 1982).

Fett76a.

Fettweis, A., "Realizability of Digital Filter Networks," *Arch. Elek. Ubertrangung* **30** pp. 90-96 (Feb. 1976).

Fish84a.

Fisher, Joseph A., "The VLIW Machine: A Multiprocessor for Compiling Scientific Code," *Computer* 17(7)(July, 1984).

Flyn70a.

Flynn, M. J. and et., al., "A Multiple Instruction Stream Processor with Shared Resources," *Proceedings of the Conference on Parallel Processing*, pp. 251-286 Spartan Press, (1970).

Flyn72a.

Flynn, M. J., "Some Computer Organizations and their Effectiveness," *IEEE-C21*, (Sept. 1972).

Gitl81a.

Gitlin, R. D. and Weinstein, S. B., "Fractionally-Spaced Equalization: An Improved Digital Transversal Equalizer," *Bell System Technical Journal* 60(2)(February, 1981).

Gitl82a.

Gitlin, R. D., Meadors, H. C. Jr., and Weinstein, S. B., "The Tap-Leakage Algorithm: An Algorithm for the Stable Operation of a Digitally Implemented, Fractionally Spaced Adaptive Equalizer," *Bell System Technical Journal* 61(8)(October, 1982).

Gold69a.

Gold, B. and Rader, C., *Digital Processing of Signals*, McGraw-Hill (1969).

Golu80a.

Golumbic, Martin C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York (1980).

Grah69a.

Graham, R. L., "Bounds on Multiprocessing Time Anomalies," *SIAM J. on Applied Mathematics* 17(2) pp. 416-429 (March, 1969).

Hait85a.

Hait, David J., "The BLOSIM Simulation Program," *Master's Report*, (Nov. 11, 1985). U. C. Berkeley

Henk75a.

Henke, W., "MITSYN - An Interactive Dialogue Language for Time Signal Processing," *MIT Research Laboratory of Electronics memo. no. RLE-TM-1*, (Feb. 1975).

Hilf84a.

Hilfinger, Paul, "SILAGE: A Language for Signal Processing," *Private Communication*, (October, 1984).

Hu61a.

Hu, T. C., "Parallel Sequencing and Assembly Line Problems," *Operations Research* 9(6) pp. 841-848 (1961).

Hwan79a.

Hwang, Kai, *Computer Arithmetic*, John Wiley & Sons, Inc. (1979).

II83a.

II, L. F. Horney, *Job Scheduling in a Distributed System*, Columbia University, New York (1983). PhD Thesis

Jhon85a.

Jhon, C. S., Sobelman, G. E., and Krekelberg, D. E., "Silicon Compilation Based on a Data-Flow Paradigm," *IEEE Circuits and Devices Magazine* 1(3)(May 1985).

John84a.

Johnson, O. G., "Three-Dimensional Wave Computations on Vector Computers," *Proceedings of the IEEE* 72(1)(January, 1984). In a Special Issue on Supercomputers

Jord84a.

Jordan, H. F., "Experience with Pipelined Multiple Instruction Streams," *Proceedings of the IEEE* 72(1)(January, 1984). In a Special Issue on Supercomputers

Jump78a.

Jump, J. Robert and Ahuja, Sudhir R., "Effective Pipelining of Digital Systems," *IEEE Trans. on Computers* C-27(9)(Sept., 1978).

Kahr84a.

Kahrs, M., "Silicon Compilation of a Very High Level Signal Processing Language," in *VLSI Signal Processing*, IEEE Press, New York (1984). [University of Rochester]

Kara65a.

Karafin, B., "The new block diagram compiler for simulation of sampled-data systems," *AFIPS Conference Proceedings* 27 pp. 55-61 (1965). Spartan Books

Karp66a.

Karp, R. M. and Miller, R. E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal* **14** pp. 1390-1411 (November, 1966).

Karp67a.

Karp, R. M., Miller, R. E., and Winograd, S., "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM* **14** pp. 563-590 (1967).

Karp69a.

Karp, Richard M. and Miller, Raymond E., "Parallel Program Schemata," *Journal of Computer and System Sciences* **3** pp. 147-195 (1969).

Kell61a.

Kelly,, Lochbaum,, and Vyssotsky,, "A Block Diagram Compiler," *BSTJ* **40**(3)(May, 1961).

Kers85a.

Kershaw, R. N., Bays, L. E., Freyman, R. L., Klinikowski, J. J., Miller, C.

R., Mondal, K., Moscovitz, H. S., Stocker, W. A., and Tran, L. V., "A Programmable Digital Signal Processor with 32b Floating Point Arithmetic," *ISSCC 85 Digest of Technical Papers*, (Feb. 13, 1985). AT&T Bell Labs WE DSP32 First Announcement

Kogg81a.

Kogge, Peter M., *The Architecture of Pipelined Computers*, Hemisphere Publishing Co., McGraw Hill Book Co., New York (1981).

Kohl75a.

Kohler, W. H., "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. on Computers*, pp. 1235-1238 (Dec., 1975).

Kope80a.

Kopec, G., *The Representation of Discrete-Time Signals and Systems in Programs*, MIT PhD Thesis (May, 1980).

Kope84a.

Kopec, Gary E., "The Integrated Signal Processing System ISP," *IEEE Trans. on ASSP* **32**(4)(August, 1984).

Kope85a.

Kopec, Gary E., "The Signal Representation Language," *IEEE Trans. on ASSP* **ASSP-33**(4)(August 1985).

Korn77a.

Korn, G., "High-speed block-diagram languages for microprocessors and minicomputers in instrumentation, control, and simulation," *Computers in Electrical Engineering* **4** pp. 143-159 (1977).

Kung80a.

Kung, H. T. and Leiserson, Charles E., "Algorithms for VLSI Processor Arrays," in *Mead and Conway, Introduction to VLSI Systems*, Wesley Publishing Co., Reading, MA (October, 1980).

Kung83a.

Kung, S. Y. and Hu, Yu Hen, "A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems," *IEEE Transations on ASSP* **ASSP-31, No. 1**(February, 1983). Looks Good.

Kung84a.

Kung, S. Y., "On Supercomputing with Systolic/Wavefront Array Processors," *Proceedings of the IEEE* 72(7)(July 1984).

Leis83a.

Leiserson, Charles E. and Rose, Flavio M., "Optimizing Synchronous Circuitry by Retiming," *Third Caltech Conference on VLSI*, (March, 1983).

Lu85a.

Lu, Hui-Hung, Lee, Edward A., and Messerschmitt, David G., "Fast Recursive Filtering with Multiple Slow Processing Elements," *IEEE Transactions on Circuits and Systems*, (November, 1985).

Maga85a.

Magar, Surendar, Essig, Daniel, Caudel, Edward, Marshall, Steve, and Peters, Roger, "An NMOS Digital Signal Processor with Multiprocessing Capability," *ISSCC 85 Digest of Technical Papers*, (Feb. 13, 1985). Texas Instruments TMS32020 First Announcement

Maga82a.

Magar, S., Caudel, E., and Leigh, A., "A Microcomputer with Digital Signal Processing Capability," *International Solid-State Circuits Digest of Technical Papers*, pp. 32-33 (Feb., 1982).

Mess84a.

Messerschmitt, David G., "Structured Interconnection of Signal Processing Programs," *Proceedings of Globecom 84*, (Dec., 1984).

Mess84b.

Messerschmitt, David G., "A Tool for Structured Functional Simulation," *IEEE Journal on Selected Areas in Communications* **SAC-2**(1)(January, 1984).

Mint83a.

Mintzer, F., Davies, K., Peled, A., and Ris, F. N., "The Real-Time Signal Processor," *IEEE Trans. on Acoustics, Speech, and Signal Processing* **ASSP-31**(1) p. 83 (Feb. 1983).

Moto86a.

Motorola,, *DSP56000 Digital Signal Processor User's Manual.* 1986.

NEC.....

NEC Electronics U.S.A. Inc.,, *Digital Signal Processor*, Data sheet for the uPD7720 Signal Processor Interface (SPI)

Padu80a.

Padua, D. A., Kuck, D. J., and Lawrie, D. H., "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. on Computers* **C-29**(9) pp. 763-776 (Sept. 1980).

Patt81a.

Patterson, D. A. and Sequin, C. H., "RISC I: A Reduced Instruction VLSI Computer," *Proceedings of the 8th Symposium on Computer Architecture*, pp. 443-457 (May 1981). ACM SIGARCH CAN

Paul80a.

Paul, D. B., Feldman, J. A., and Sferrino, V. J., "A design study for an easily programmable, high-speed processor with a general purpose architecture," *MIT Lincoln Lab. Tech. Note 1980-50*, (1980).

Pete77a.

Peterson, James L., "Petri Nets," *Computing Surveys* 9(3)(September, 1977).

Pete81a.

Peterson, James L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall Inc., Englewood Cliffs, NJ (1981).

Pope84a.

Pope, S., Rabaey, J., and Brodersen, R. W., "Automated Design of Signal Processors Using Macrocells," in *VLSI Signal Processing*, IEEE Press, New York (1984). [U. C. Berkeley]

Qure84a.

Qureshi, S. U. H. and Ahmed, H. M., "A Custom Chip Set for Digital Signal Processing," in *VLSI Signal Processing*, IEEE Press, New York (1984). [Codex Corp.]

Rao85a.

Rao, Sailesh K., *Regular Iterative Algorithms and their Implementations on Processor Arrays*, Information Systems Laboratory, Stanford University

(October, 1985). PhD Dissertation

Reit67a.

Reiter, Raymond, *A Study of a model for Parallel Computations*, University of Michigan (1967). Doctoral Dissertation

Reit68a.

Reiter, Raymond, "Scheduling Parallel Computations," *JACM*, (14) pp. 590-599 (1968).

Renf81a.

Renfors, Markku and Neuvo, Yrjo, "The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints," *IEEE Trans. on Circuits and Systems* CAS-28(3)(March 1981).

Rumb77a.

Rumbaugh, J., "A Data Flow Multiprocessor," *IEEE Trans. on Computers* C-26(2) p. 138 (Feb. 1977).

Schw85a.

Schwartz, David A., "Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs," *Georgia Institute of Technology Technical Report DSPL-85-2*, (July 1985). PhD Dissertation

Shar74a.

Shar, Leonard E. and Davidson, Edward S., "A Multiminiprocessor System Implemented Through Pipelining," *Computer* 7(2)(Feb., 1974).

Sher84a.

Sherburne, Robert Warren Jr., *Processor Design Tradeoffs in VLSI*, PhD Dissertation, U.C. Berkeley, Berkeley, CA (1984).

Shiv82a.

Shively, Richard R., "Architecture of a Programmable Digital Signal Processor," *IEEE Trans. on Computers* c-31(1)(Jan., 1982).

Sing85a.

Singer, Elliot, "Comparative Architectures for a Multiple Function Speech Processor," *MIT Lincoln Labs Report TR-712*, (1985). In publication.

Smit78a.

Smith, Burton J., "A Pipelined, Shared Resource MIMD Computer," *Proc. of the 1978 Int. Conf. on Parallel Processing*, pp. 6-8 (1978).

Smit85a.

Smith, James E., "Implementation of Precise Interrupts in Pipelined Processors," *SIGARCH Newsletter* 13(3)(June 1985). Conference Procedings for the 12th Annual International Symposium on Computer Architecture

Snyd84a.

Snyder, Lawrence, "Parallel Programming and the Poker Programming Environment," *Computer* 17(7)(July, 1984).

Texa83a.

Texas Instruments, Inc.,, *TMS32010 User's Guide*. 1983.

Thom77a.

Thompson, C. D. and Kung, H. T., "Sorting on a Mesh-Connected Parallel Computer," *Comm. ACM* 20(4) pp. 263-271 (april, 1977).

Thor64a.

Thornton, J. E., "Parallel Operation in the Control Data 6600," *Proc. FJCC* **26, part 2** p. 33 (1964).

Thor70a.

Thornton, J. E., *The Design of a Computer*, Scott Foreman & Co. (1970).

Trel81a.

Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P., *Data Driven and Demand Driven Computer Architecture*, University of Newcastle upon Tyne, Newcastle upon Tyne, England (1981). Technical Report

Tsud83a.

Tsuda, T., Mochida, Y., Murano, K., Unagami, S., Gambe, H., Ikezawa, T., Kikuchi, H., and Fujii, S., "A High performance LSI Digital Signal Processor for Communication," *Proceedings of IEEE International Conference on Communications*, (June 19, 1983). Fujitsu Labs, Ltd.

Unge76a.

Ungerboeck, Gottfried, "Fractional Tap-Spacing and Consequences for Clock Recovery in Data Modems," *IEEE Transactions on Communications*, (August, 1976).

Unge85a.

Ungerboeck, G., Maiwald, D., Kaeser, H. P., Chevillat, P. R., and Beraud, J. P., "Architecture of a digital signal processor," *IBM Journal of Research and Development* 29(2)(March 1985).

Wats82a.

Watson, I. and Gurd, J., "A Practical Data Flow Computer," *Computer* 15(2)(Feb. 1982).

Yama85a.

Yamauchi, Horonori and et. al.,, "An 18-bit Floating-point Signal Processor

VLSI with on-chip 512W Dual-port RAM," *Proceedings of ICASSP 85*, p. 204 (March, 1985). Tampa, Fla.

Zema83a.

Zeman, J. and Moschytz, G. S., "Systematic Design and Programming of Signal Processors, Using Project Managment Techniques," *IEEE Trans. on Acoustics Speech and Signal Processing* ASSP-31(6)(December 1983).