On the Design and Performance of Pipelined Architectures

TR87-022

August, 1987

N.P. Topham, A. Omondi, R.N. Ibbett**

The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27514

**Department of Computer Science, University of Edinburgh*

On the Design and Performance of Pipelined Architectures

Nigel P. Topham

Department of Computer Science,
University of Edinburgh,
James Clerk Maxwell Building,
The Kings Buildings,
Mayfield Road,
Edinburgh EH9 3JZ,
Scotland, U.K.

Amos R. Omondi

Department of Computer Science,
University of North Carolina
Sitterson Hall 083,
Chapel Hill, NC 27514,
U.S.A.

Roland N. Ibbett

Department of Computer Science,
University of Edinburgh.

ABSTRACT

Pipelining is a widely used technique for implementing architectures which have inherent temporal parallelism when there is an operational requirement for high throughput. Many variations on the basic theme have been proposed, with varying degrees of success. The aims of this paper are twofold. The first is to present a critical review of conventional pipelined architectures, and put some well known problems in sharp relief. It is argued that conventional pipelined architectures have underlying limitations which can only be dealt with by adopting a different view of pipelining. These limitations are explained in terms of discontinuities in the flow of instructions and data, and representative machines are examined in support of this argument. The second aim is to introduce an alternative theory of pipelining, which we call *Context Flow*, and show how it can be used to construct efficient parallel systems.

Keywords : **computer architecture, pipelining, multiprocessing, micromultiprogramming, context flow.**

1 Introduction: A Review of Pipelining Principles

Pipelining generally refers to the exploitation of temporal parallelism as a means of achieving high performance. In its simplest form, it is the decomposition of a function into sub-functions, coupled with the provision of segmented hardware to process all sub-functions in parallel. A typical example, *instruction pipelining*, is the processing of instructions in which the typical task constituents are the phases of instruction execution: Fetch Instruction, Decode, Generate Operand Addresses, Fetch Operands, Execute and Store Result. Similarly, in *arithmetic pipelining*, an arithmetic operation such as floating point addition can be subdivided into the following subtasks: Subtract Exponents, Align Mantissae, Add Mantissae and Normalise Result. The corresponding pipelines are illustrated in Figure 1. *Vector pipelines* are an important class of pipelines designed to process vector instructions by attempting to stream the elements of a vector through pipelined arithmetic units.

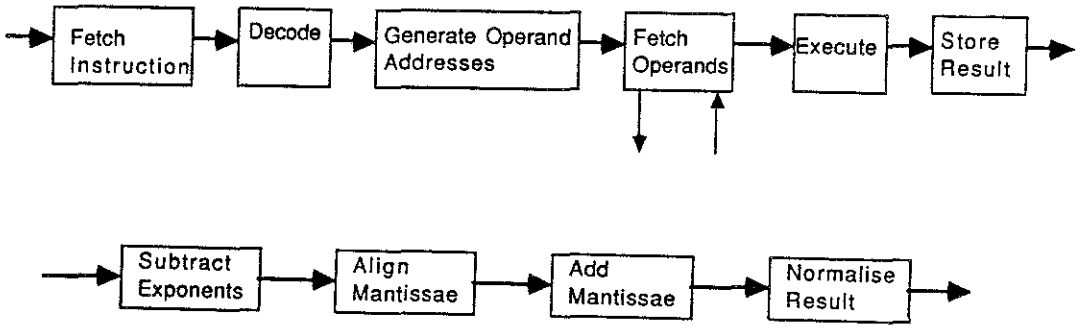

Figure 1: Typical Pipeline Structures

The performance gained through the use of pipelining may be determined by considering the processing of N tasks in a pipeline of n stages with a beat time of t_p. The total time required is given by:

$$T_p = n\,t_p + (N-1)t_p \tag{1}$$

The first term in this expression is the *start-up time*, that is, the time required for the first set of data to propagate through the pipeline. The second term is the time required to stream the remaining $N-1$ sets of data through the pipeline.

This can be compared with the time required to process the same data using non-pipelined logic (T_{np}) by assuming the same end-to-end latency in both cases. Hence, $T_{np} = n\,N\,t_p$, and we can say that:

$$T_p = T_{np}\,\frac{(n+N-1)}{n\,N} \tag{2}$$

the speedup over the non-pipelined operation is then

$$\frac{T_{np}}{T_p} = \frac{nN}{(n+N-1)} \tag{3}$$

and

$$\lim_{N\to\infty} \frac{T_{np}}{T_p} = n \tag{4}$$

As the speedup tends to n, for large N, the more stages there are in a pipeline, the better its peak performance will be, provided the end-to-end latency of the computation remains constant. In practice technology limits how far the beat time can be reduced, and other factors also constrain the number of stages that can be employed usefully within a range of possible beat times [Kunkel and Smith, 1986].

The equations above represent a simplification (for the purposes of demonstrating the general benefits of pipelining in a simple form); in practice T_p is largely determined by three factors: the time to get the first set of data to the first stage of the pipeline (this includes address generation time, store access time, etc.), the end-to-end latency, and the time to propagate the remaining data sets through the pipeline. Thus:

$$T_p = \alpha\, t_a + \beta\, t_p + \gamma\, t_p \tag{5}$$

Where t_a is the memory access time and the parameters α, β and γ vary according to the nature of the pipeline and the characteristics of the application.

Pipelines may be classified in a number of ways depending on their temporal and logical control characteristics. For example, the transfer of data between pipeline stages may be synchronous or asynchronous. Pipeline functionality may be fixed, or it may be either statically or dynamically reconfigurable.

In any pipeline, all actions should be naturally sequential, all stages should take the same time to execute and there should be a continuous flow of information between stages. This means that input should be available to a stage when it is required and output should be taken from a stage as soon as it is produced. The computation performed at each pipeline stage should be dependent only upon the information passed to it by the preceeding stage. The equations of throughput given above assume that these conditions are satisfied and whilst these conditions are normally satisfied within arithmetic pipelines, instruction pipelines frequently suffer from their inability to meet them. More thorough discussions of the fundamentals of pipelining may be found in [Hwang and Briggs, 1984,Ibbett, 1982,Rammamoorthy and Li, 1977].

1.1 Aims and Outline of the paper

The aims of this paper are twofold. The first is to present a critical review of what has been accomplished in the design of pipelined computers, to put some well known problems of pipelining in sharp relief, and to argue that conventional implementations of pipelined architectures have underlying limitations that can only be dealt with by the adoption of a different view of pipelining. We discuss these problems within a uniform framework and examine representative machines in detail. The second is to present a design alternative, *Context Flow*, and discuss the extension of this to larger systems of communicating processors.

In section 2 the limitations of conventional pipelined machine design are discussed in the light of the inherent discontinuities in the flow of instructions and data and their adverse effects on the performance and complexity of hardware. In the third section the background to *micro-multiprogramming* is discussed, together with its suitability as an alternative implementation technique. A more general implementation technique, called *Context Flow*, which regards computation as a set of logical transformations on process contexts, is then presented and the design of a simple parallel processing element is outlined. This concept is extended in section 4, which describes how an arbitrary number of these processing elements could be connected together using a multistage Context Flow network.

Section five summarises the main topics of the paper, and outlines the current state of the authors' research in this area.

2 Limitations of Conventional Pipeline Designs

Although the perfect instruction pipeline will achieve a throughput of one instruction per pipeline beat, in practice this has proved difficult to achieve due to the dependencies which exist between instructions and data under a sequential model of computation. Previous attempts to solve this problem have produced complex hardware structures that fail to provide an effective cost/performance benefit. The first order of business, therefore, is to examine these issues and to appreciate the fact that a different view of pipelining is necessary.

2.1 Discontinuities in the Flow of Instructions

Data and instruction access times are normally much longer than the pipeline beat time. If the pipeline is halted when these accesses occur then its thoughput will be significantly reduced. Two partial solutions exist;

1. Prefetch data and instructions, and hence ensure that access times will fall within the pipeline beat time.

2. Allow "out of sequence" processing of instructions following a high-latency operation.

The processing of "out of sequence" instructions may seem to be a sensible solution; however, it requires complex control mechanisms to enforce the inherent data dependence constraints. The inter-dependence of textually close instructions also limits amount of "out of sequence" processing to relatively small numbers of instructions.

The prefetching of instructions relies upon being able to predict the future sequence of instructions, before they are issued and before it is known that they will be issued. Conditional control transfers create difficulties here, because of the unavailability of the decision variable at the moment when prediction occurs. Even with special prediction hardware there is a finite probability that an incorrect prediction will be made. When the false prediction is noticed, a number of cycles later, the *start-up* time of the pipeline will create gap in the flow of instructions with a corresponding degradation in performance.

The extent to which control transfer instructions are detrimental depends on several factors, the most important being the position of the Control Point, the length of the pipeline, and the extent to which techniques for minimizing these problems are effective. The Control Point is the point at which the Program Counter is altered and it can be regarded as the point at which branch instructions are executed.

The effect of control transfers on pipeline performance can be approximated by assuming that a proportion m $\{0 \leq m \leq 1\}$ of all instructions are control transfers. Then, assuming that these instructions occur in sequence:

$$T_p' = (m\,N + 1)t_a + n(m\,N + 1)t_p + [(1 - m)N - 1]\,t_p \qquad (6)$$

The main objective is therefore to mask out the effects of the terms in m so that T_p' approaches T_p. This is the rationale behind the techniques described in the remainder of this section.

2.1.1 The Pipeline Length

According to Equation 4 the length of the pipeline determines the maximum throughput. However, a long pipeline also exhibits a long start-up time, resulting in a large number of instructions being discarded and a longer pipeline-refilling time when the flow of instructions becomes disrupted.

It is precisely for this reason that the Reduced Instruction Set Computers such as the Berkley RISC [Sequin, 1983], Stanford MIPS [Hennessy, 1984], and the IBM 801 [Radin, 1983]

have opted for fairly short and simple pipelines. Similar reasoning underlies the design of MU6-G [Edwards et al, 1980], a successor to MU5. Nonetheless, performance degradations caused by control transfers have failed to disappear completely even in these machines.

2.1.2 Position of the Control Point

The Control Point is the stage in the pipeline containing the instruction addressed by the Program Counter. As each instruction passes the Control Point the Program Counter is updated, and the instruction is considered to have been executed. No irrecoverable action must be taken during partial execution of instructions before the Control Point. The Control Point is also the stage at which control transfers are executed, and since control transfers are normally followed down the pipeline by incorrect sequences of instructions, which must be discarded, the delay incurred while waiting for new instructions depends upon the distance between the store and the Control Point. This suggests placing the Control Point early in the pipeline.

However, conditional control transfers depend on a value or condition evaluated in an arithmetic unit, normally at the end of the pipeline, and the delay incurred while waiting for the condition to be evaluated depends upon the distance between the Control Point and the arithmetic unit. This suggests placing the Control Point late in the pipeline. In fact, for conditional control transfers that branch, the total delay depends not on the position of the Control Point, but on the delays inherent in the number of stages between the store and the Control Point *plus* the number of stages between the Control Point and the arithmetic unit, i.e. on the *total* pipeline length. Therefore techniques to minimise both these delays must be incorporated into a high-performance pipeline. A more detailed discussion of this issue may be found in [Ibbett, 1982].

2.1.3 Conventional Techniques used to Limit the Effect of Branches

Various means have been employed to limit the effects of Branch instructions on performance. These can broadly be divided into two categories: those implemented in software and those implemented in hardware. The former generally involve the use of optimising compilers to generate more congenial code than would otherwise be dictated by the normal flow of events and are typified by *delayed branching* [Sequin, 1983] and other similar techniques [Goodman, 1985]; small systems have tended to rely solely on such techniques. Conversely, the larger machines have typically employed hardware mechanisms to minimize the gaps created, either by trapping loops or by attempting to predict branch destinations. The following subsections examine these techniques in some detail.

Hardware Techniques To Limit the Effect of Branches

The problems involved in supplying a constant stream of instructions from store to an instruction pipeline are ameliorated to a large extent by the fact that most instructions are obeyed sequentially and that the main store word size is normally such that one word fetched from main store can contain several instructions. Furthermore, with an interleaved store, successive accesses for sequential instructions reference each stack in turn and are not held up by cycle time effects. Store requests can therefore be made in advance of the corresponding instruction being required and the replies buffered until they are needed for execution. This *pre-fetching* technique is used in almost all high performance pipelined processors. A significant proportion of instructions result in the transfer of control, however, and each such transfer requires a request to be made to the store for a new sequence of instructions. So, although the accessing rate for instructions can normally be matched satisfactorily to the processing rate, the access time for the first instruction of a new sequence can result in a long delay to the processor. Techniques for overcoming this problem rely on the fact that the cause of many control transfers is a branch back from the end to the start of a loop of

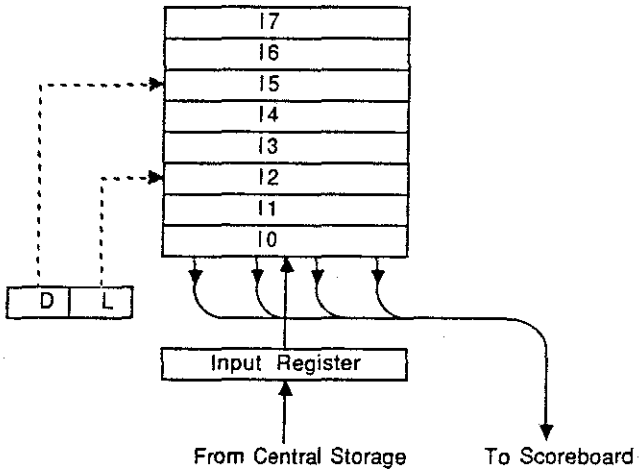

Figure 2: The CDC 6600 Instruction Stack

instructions, and *loop catching* buffers are incorporated into a number of processors. One of the earliest examples of such a system was that used in the CDC 6600 [Thornton, 1970].

The CDC 6600 Instruction Stack

In the CDC 6600 instructions are fetched from store and placed in an Instruction Stack before being decoded and issued by the Scoreboard (control unit) to the appropriate function unit. The Instruction Stack itself consists of eight 60-bit registers (I0-I7 in Figure 2) which operate as a stack and which can contain instruction loops. Programs are initiated in the 6600 by an *Exchange Jump* in which the contents of all addressable registers are interchanged with the contents of a designated store area. Following such an Exchange Jump the new contents of the program address register are used to access the first instruction word. This word is received from store into an Input Register and then loaded into the bottom register of the Instruction Stack.

Instruction words are made up of four 15-bit *parcels* and as the first instruction word enters the bottom register of the stack (I0), the first two parcels within the word (to allow for 30-bit instructions) are transferred into a series of instruction registers within the Scoreboard. As further instructions are fetched the old instructions ripple upwards through the stack.

Information about the contents of the stack is contained in two registers, D and L. The D(epth) register measures the number of valid instruction words in the stack, and L(ocation) register specifies the location in the stack of the instruction word currently in use. During execution of a loop held entirely in the stack, the instructions remain in fixed locations and the program address register can point to any one of the stack registers within a distance D from the bottom. D is re-set to zero whenever a branch out of the stack is taken, and is incremented by one for every new instruction word brought in. When the stack is full, D remains equal to seven.

When a conditional branch is decoded a test for *jump within stack* is made. This involves subtracting the current program address from the branch address. If the absolute value of the result

is less than seven words, and if the values in D and L indicate that the branch is to a location within the stack, no further store accesses are made for instruction words until instruction parcels are again taken from I0. Thus a branch may jump forwards or backwards within the stack and loops may be held in the stack in various forms.

A very similar Instruction Stack was used in the STAR-100 computer, CDC's first commercially produced vector processor. The STAR-100 had a much longer instruction format than the 6600 so that its Instruction Stack was larger, being made up of sixteen 128-bit registers, but it used essentially the same mechanisms. Both these systems are relatively simple and make no attempt to deal with the delays caused by conditional control transfers. As we saw earlier, the delays incurred by conditional control transfers which branch depend on the number of stages both before and after the Control Point. However, if correct instruction sequences can be supplied to the pipeline behind control transfers, then the delay depends only on the number of stages beyond the Control Point. Furthermore, if recoverable instructions can be sent out beyond the Control Point, then in some cases the delays incurred beyond the Control Point can also be overcome. The instruction buffering system in the IBM System/360 Model 195 attempted to do this.

The IBM System/360 Model 195 Instruction Processor

The IBM System/360 Model 195 [Murphy and Wade, 1970] central processor consists of an Instruction Procesor and Fixed and Floating-point Execution Units. The Instruction Processor is concerned with fetching and buffering instructions from store, fetching the operands which those instructions specify, issuing instructions to the appropriate execution unit, handling interrupts, and executing all branching (control transfer), status switching and input/output instructions.

Instructions fetched from store are buffered in an Instruction Stack (Figure 3) made up of eight 64-bit registers. The instruction fetching mechanism is controlled by three registers, the Instruction Register (IR) which addresses the instruction currently being decoded, the Upper Bound Register (UB) which points to the most recent word brought into the stack, and the Lower Bound Register (LB) which points to the earliest word in the stack. During normal operation the stack contains the current instruction word, some words ahead of the current instruction and a copy of some instructions which have already been issued.

Pre-fetching of instructions is controlled by the UB register. When instruction fetching is initiated following an interrupt, for example, the Instruction Stack is declared empty and the store address of the first instruction word is loaded into UB and LB. The instruction fetching mechanism associated with UB then accesses this word and loads it into the location in the Instruction Stack addressed by the three least significant word address bits in UB. Initially this location is also addressed by IR, which selects each instruction in sequence for decoding and processing. After an instruction has been decoded and passed to the next stage in the processor pipeline, IR is incremented by the number of half-words in that instruction and the next instruction selected.

Once the first instruction access has been sent to store, the instruction fetching mechanism increments UB and continues to make sequential store accesses until prevented from doing so either because the address in UB is seven words higher than that in IR (and any further accesses would cause instructions not yet decoded to be overwritten), or because the Instruction Processor has detected a condition giving rise to a change in the instruction sequence (a branch instruction or an interrupt, for example).

During normal operation the instruction fetching mechanism continually attempts to increment UB and fetch instruction words from store, while the instruction decoding mechanism continually increments IR as instructions are decoded and passed along the processor pipeline. Once IR has been incremented beyond the address in LB, instructions in the first word fetched into the stack can be overwritten with new information. LB and UB are then incremented together and at each instruction access the oldest word in the stack is replaced by the latest word fetched from store.

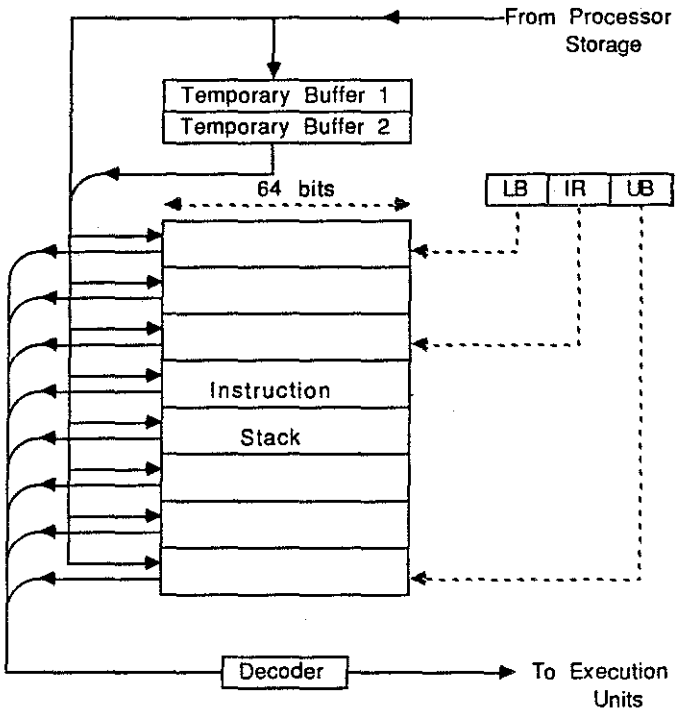

Figure 3: The IBM System/360 Model 195 Instruction Buffer

Use of this pre-fetching mechanism allows a continuous sequence of instructions to be supplied to the processor at a rate approaching one per machine clock cycle, and thus roughly matching the instruction execution rate. When a new sequence of instructions is required as a result of the branch being taken in a control transfer instruction, however, the start-up delay is of the order of six clock cycles, and in the absence of some additional technique the average performance of the processor would be seriously degraded. Conditional branches cause even further problems, of course, since the branch decision depends on the outcome of a previously issued, but not necessarily completed arithmetic instruction, and an additional delay may be incurred in awaiting this outcome. In the Model 195 two techniques are used to ameliorate the problems caused by branches, one involving a *Conditional Mode* of operation, and the other a *Loop Mode.*

Conditional Mode

Conditional branch instructions interrogate a 2-bit Condition Code at their point of execution in order to determine whether or not the branch is to be taken. The Condition Code is set by a variety of instructions, but only the last of these issued before a conditional branch must be allowed to affect its outcome. This is accomplished by tagging each instruction which will set the Condition Code as it leaves the Instruction Unit. At the same time a signal is forwarded through the pipeline to remove the tags from any previously issued but uncompleted instructions. Only a tagged instruction may set the Condition Code, at which point its tag is removed, and a conditional branch instruction can only execute when there are no outstanding tags in the processor.

The Condition Code will normally be invalid when a conditional branch is decoded, and so the hardware always assumes this to be the case and establishes Conditional Mode. In Conditional Mode further sequential instruction accesses are inhibited, but rather than hold up further activity entirely, processing of the remaining instructions in the Instruction Stack beyond the branch proceeds as far as possible, with the instructions being marked as *conditional.* When conditional instructions are decoded, their operand fetches are initiated, and they are forwarded beyond the Control Point to the relevant execution units in the normal way. The conditional tag inhibits the execution units from actually completing them, however, and once the first such instruction reaches the point of execution, further processing is held up until the Condition Code is set and the branching action determined. If the branch is not taken, the conditional tags are re-set and the pipeline re-started.

If the branch is taken, the conditional instructions must be abandoned and a fresh start made with a new sequence. The delay incurred in refilling the pipeline from the decoder onwards is unavoidable, but the delay in accessing the first instruction at the target address of the new sequence is minimised in the Model 195 because the hardware assumes at the start of Conditional Mode that either outcome is equally likely and fetches the first two instruction words at the branch target address immediately. These two words are loaded into the two Temporary Buffers shown in figure 3, in order that the Instruction Stack remain unaffected if the branch is not taken. If the branch is taken, the access time for the target instructions will have been overlapped with the wait for the Condition Code. In the case of an unconditional branch to an instruction not in the Instruction Stack, there is, of course, no need to wait for the Condition Code to become valid. As in the conditional case, the target instruction sequence is requested immediately, but unless the execution unit pipelines are also held up (as a result of divide operations, for example) the six clock start-up delay inevitably causes a gap to occur in the instruction processing sequence.

Loop Mode

Without the use of branch target instruction pre-fetching in Conditional Mode, the time lost when the branch is taken would be roughly equal to the sum of the time spent waiting for the Condition Code to be set and the store access time, i.e. equivalent to the full length of the pipeline. With pre-

fetching the time lost becomes equal to only the greater of these two, but even so, where the branch is closing a short loop of instructions, this loss can severely limit overall processor performance. Thus for short loops a different philosophy is adopted whereby the entire loop is contained within the Instruction Stack and store accesses are avoided altogether until the program exits from the loop. Clearly, the longer the loop, the smaller the proportion of time lost as a result of the branch, and the choice of eight words as the capacity of the stack represented a compromise between hardware cost and performance in Loop Mode.

Loop Mode is entered whenever a branch backwards is taken to a target address within eight words of the current instruction. The Instruction Stack is immediately re-initialised to contain the appropriate eight words, after which instruction fetching ceases and the address path to store is fully available for operand fetching throughout execution of the loop. Loop Mode is controlled by two additional registers, one containing the loop target address (SLT) and the other the value of IR corresponding to the loop closing instruction (SLCIR). Once in Loop Mode the address of any branch instruction being decoded is compared with that in SLCIR, and if it is the same the branch is made immediately to the target address held in the other. Thus the asumption built into Conditional Mode is reversed, since it is assumed that the branch will be taken, and instructions are therefore decoded from the target path rather than the straight through path. Furthermore, no fetches are made to the Temporary Buffers. Loop Mode is turned off when an exit is taken from the loop.

The main drawback of both the IBM System/360 Model 195 and the CDC 6600 instruction buffers is that where the total number of instructions being obeyed in a loop will fit into the stack, but the code is actually made up of a number of non-contiguous segments (as in figure 4, for example), the loop may not be caught in the stack. With machine code programming this situation can normally be avoided, but it is a common occurrence in compiler generated code and increasing emphasis on high-level language programming caused processor designers to seek alternative solutions. The CDC 7600 [CDC, 1977]and CYBER 205 [CDC, 1981], for example (successors, respectively, to the 6600 and STAR-100 [Hintz and Tate, 1972]), both use associatively addressed buffers, with the CYBER 205 Instruction Stack again being correspondingly larger than that in the 7600.

The CDC 7600 Instruction Stack

The CDC 7600 was designed to be machine code upward compatible with the 6600, but to provide a substantial increase in performance. The central processor is very similar to that of the 6600; it contains nine parallel function units, a scratch pad of eight X registers, eight A registers and eight B registers, and an Instruction Stack. The 7600 Instruction Word Stack is made up of twelve 60-bit registers, however, compared with the eight used in the 6600, and each register also has its own 18-bit associative address register in an Instruction Address Stack (figure 5).

The Instruction Stack is filled two words ahead of the instruction currently being executed, thus giving a greater degree of pre-fetching than in the 6600. Furthermore, instructions are obeyed from a Current Instruction Word (CIW) register, rather than from the bottom stack register, and a complete 60-bit word is transferred from the Instruction Stack into this register whenever the word address changes in the program address counter. This transfer can be made from any of the twelve registers in the Instruction Word Stack, allowing a considerable degree of flexibility in pre-fetching and loop catching. Whenever a new word is required in the CIW register, the address in the program address counter is compared with the entries in the Instruction Address Stack, and if a coincidence occurs for any of these entries, the content of the corresponding register in the Instruction Word Stack is transferred into the CIW register.

When obeying sequential code the required word will normally be in one of the bottom two registers. When a branch instruction is executed and the branch is taken, the required word may already be in one of the top ten registers, obviating the need for a store access, and giving improved

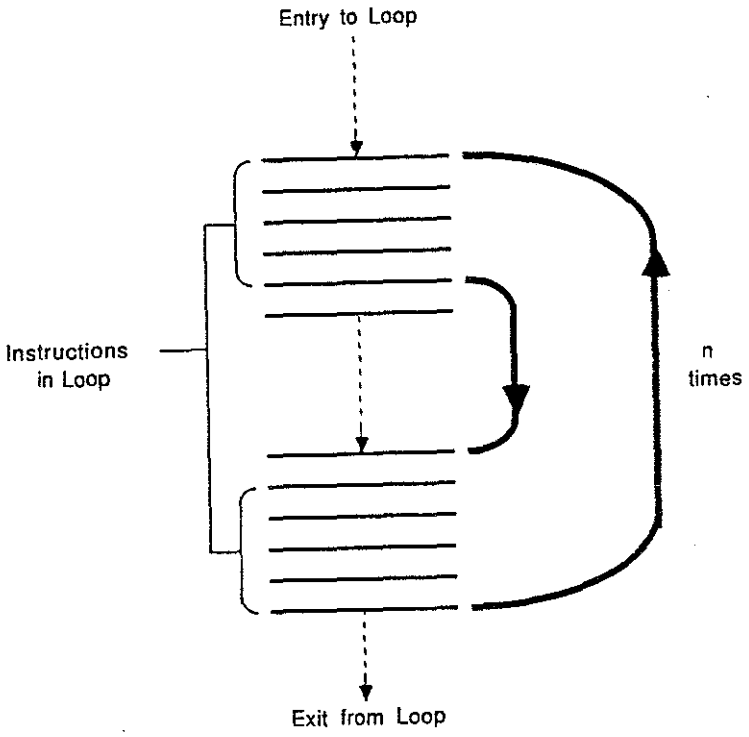

Figure 4: A Non-contiguous Instruction Loop

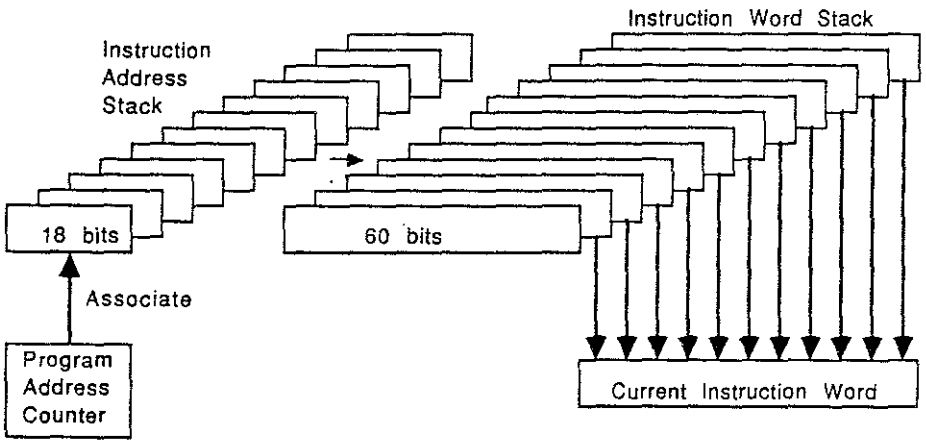

Figure 5: The CDC 7600 Instruction Stack

performance. If the required word is not in the stack, the first two words at the target address are immediately requested from store and instruction execution continues when the first of these is received. Whenever an instruction word is received from store all the entries in the Instruction Word Stack and the Instruction Address Stack are simultaneously moved up one position, with the new address and instruction word being entered at the bottom of the stack and the oldest entry being lost. Entries in the stack are only invalidated by the execution of a subroutine call or Exchange Jump, and not by normal branch instructions, so that a program may branch back and forth between short sequences of non-contiguous code held in the stack.

Although this stack is larger than that of the 6600, it is still relatively small, and considerable effort is frequently required to reduce the amount of code in program loops in order that they may fit into it. A different scheme from those previously considered is required if loops of unrestricted size are to be accommodated. One such scheme involves the use of a *Branch Target Buffer*, first introduced in the MU5 computer.

The MU5 Instruction Buffer Unit

In the MU5 computer instructions are pre-fetched from a four-way interleaved Local Store and buffered in an Instruction Buffer Unit. This Instruction Buffer Unit (figure 6) contains three 128-bit buffer registers through which instructions flow on their way to the Primary Operand Pipeline (PROP). The necessary store requests are made by the Store Request System, which issues store addresses formed by a counter at a rate matched to the maximum rate at which instructions can be taken from the buffers by PROP.

This system operates satisfactorily until a branch occurs (as a result of either an unconditional control transfer instruction, or a conditional control transfer instruction for which the condition is met). Then all the pre-fetched instructions have to be abandoned, and the branch target instruction cannot be executed until the store has been accessed, using the new control address, and the new instruction stream has passed through both the buffers and the PROP pipeline. As a result the total time between the execution of the control transfer and the first instruction of a new sequence is 1350 ns.

In order to reduce the number of occasions on which this delay is incurred, MU5 incorporates a Jump Trace (*Branch Target Buffer*) which attempts to predict the outcome of an impending control transfer. This is effective because a significant proportion of control transfers occur at the end of program loops, and under these circumstances the branches are normally taken. By predicting the outcome of a control transfer at pre-fetch time, rather than trying to contain loops in buffers, loop size is virtually unrestricted. (There would be a problem with very small loops, were it not for the fact that they can be caught in the very limited amount of assembly buffering in the IBU.)

The Jump Trace is implemented using an eight-line associatively addressed store. Whenever a new instruction address is generated by the pre-fetching mechanism within the IBU it is presented to the associative *jump-from* address store before being sent to store. If an equivalence is found, this address is replaced by the corresponding *jump-to* address, so that pre-fetching of the new sequence takes place instead.

When the control transfer instruction which gave equivalence in the Jump Trace is sent to PROP, it is accompanied by a bit indicating that the instructions following it are *out of sequence*. This bit is used in PROP to determine the action after execution of the control transfer. If the following instructions have been correctly predicted, execution of instructions continues uninterrupted. If the instructions are not out of sequence, but should have been, a store request is made for the instructions at the *jump-to* address, and at the same time a line in the Jump Trace is loaded with the *jump-from* and *jump-to* addresses. (The line used in the Jump Trace is selected according to a cyclic replacement algorithm and as each line is overwritten its *use* digit is set. The use digits are normally only re-set, and the Trace thereby cleared, at a process change.) When the *jump-from*

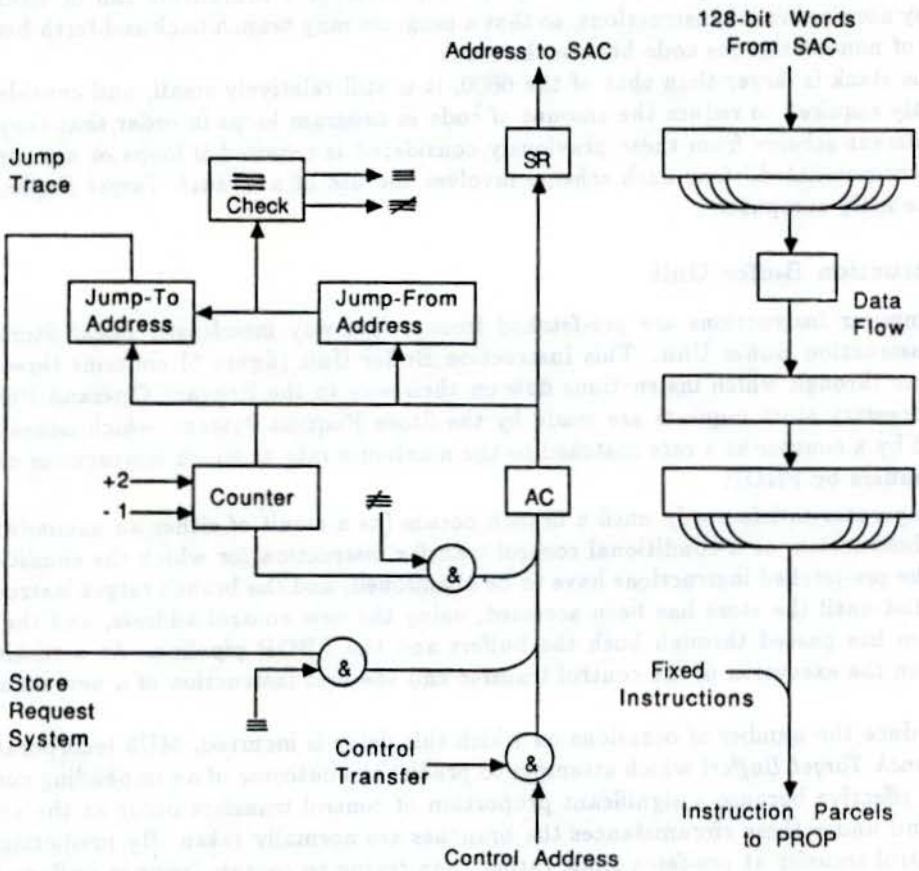

Figure 6: The MU5 Instruction Buffer Unit

Instruction Address	Normal Branch		Delayed Branch		Optimized Delayed Branch	
100	LOAD	X, A	LOAD	X, A	LOAD	X, A
101	ADD	1, A	ADD	1, A	JUMP	105
102	JUMP	105	JUMP	106	ADD	1, A
103	ADD	A, B	NO-OP		ADD	A, B
104	SUB	C, B	ADD	A, B	SUB	C, B
105	STORE	A, Z	SUB	C, B	STORE	A, Z
106			STORE	A, Z		

Figure 7: The *Delayed Branch* Technique

address is subsequently generated by the pre-fetching mechanism within the IBU, the instructions at the *jump-to* address are automatically pre-fetched.

Measurements made with the MU5 hardware performance monitor indicated that without the use of the Jump Trace, around 20% of control transfers were followed through the pipeline by the correct sequence of instructions during program execution. Using the Jump Trace, this figure rose to around 65% [Holgate and Ibbett, 1980].

The Branch Target Buffer technique has been evaluated consistently as the most effective branch prediction strategy [Lee and Smith, 1984], [McFarling and Hennessy, 1986]; one observes that even in RISC machines, where the use of software techniques has been prevalent, the use of Branch Target Buffers has not been completely ruled out [Patterson, 1983]. It has become the preferred prediction strategy in more recent high performance machines such as the IBM 3090 [Tucker, 1986] and Fujitsu's VP Series machines [Miura, 1986].

Software Techniques to Limit the Effects of Branches

Software techniques are increasingly being used as a means to deal with the branching problem in small machines where hardware is a particularly scarce resource; this is typical of single chip VLSI pipelines in which only so much can fit on the chip. The most popular by far is the *delayed branch* in which code is reorganized so that a control transfer takes effect in the second instruction slot after the one in which it is defined. This is illustrated in the example of Figure7. Essentially the code is rearranged so that the next instruction is always prefetched during the execution of the current one. This allows the hardware to execute one instruction in every beat although it may require the insertion of *No-Operation* (NOP) instructions in order to achieve this. Optimization by the compiler can be quite successful in removing a large proportion of these [Sequin, 1983] although this is only likely to be true for unconditional control transfers; conditional control transfers are generally difficult to handle and point to the weakness of such techniques. Thus, in RISC I, the designers cite the proportion of removed NOPs as 90% for unconditional branches and 25% for conditional branches; for the IBM 801 [Radin, 1983] the figures are much lower, with only 60% removal for unconditional branches.

Another software technique that can be regarded as an extension of the delayed branch is the *branch-preparation* technique used in PIPE [Goodman, 1985]. The main difference between this and the delayed branching of the Berkley RISC is that whereas the latter always requires a fixed number of instructions to be executed before the branch takes effect, and hence in some cases will require the use of NOP instructions, in PIPE the number of instructions is variable. Thus the *Prepare To Branch* instruction used in PIPE, in addition to specifying a branch condition, also specifies the number of instructions which must be executed irrespective of the branch condition.

The techniques described above are not new; they are essentially derived from similar techniques used in microprogramming and in the larger pipelined machines where they have been used to supplement hardware techniques. Their weakness is that not only do they require considerable ingenuity on the part of the compiler writer but also that in spite of this they are not always applicable; it is sometimes the case that compiler-generated code simply cannot be manipulated in the manner indicated by the above example. Indeed the designers of the Berkley RISCs have not found the use of hardware techniques, in the form of an instruction cache with some branch prediction, to be unreasonable [Patterson, 1983].

2.1.4 Commentary

Among the points that should be drawn from the above discussion is the observation that trying to deal with branches in the context of single instruction streams is unlikely to be entirely successful. A much better approach is to accept that such discontinuities are inevitable and to look for some means of masking them and hence avoiding the associated loss in performance. One such technique consists of concurrently maintaining several active instruction streams, switching to another stream whenever a control transfer is decoded in the current one, and returning to processing the (temporarily) abandoned stream only when it can be guaranteed that executable instructions are available; such an approach, as well as a practical refinement of the general idea, is described in sections 3 and 4.

2.2 Discontinuities in the Flow of Data

In the same way that a control transfer introduces increased entropy, or disordered arrangement of information, the necessity to fetch operands from outside the pipeline can also be regarded as a manifestation of high entropy, and various mechanisms have been developed to reduce its effects. These mechanisms may be *explicit* or *implicit*, but all depend on the *locality of data* phenomenon: at any one time during the execution of a program the majority of accesses which it makes are to a relatively small and slowly changing subset of its total data-set.

Explicit mechanisms usually involve the use of programmable registers in which the programmer or compiler may keep frequently used variables, while implicit mechanism involve the use of block organised cache stores such as that first introduced in the IBM System/360 Model 85 or selective word organised buffer stores such as the MU5 Name Store. Discontinuities in the flow of data can also be caused by data dependencies, however, and the problems which these can create can rapidly lead to complex hardware structures or poor performance.

The CDC 6600 and its successors (the CDC 7600 and the Cray-1) are interesting examples of machines with explicit mechanisms. All arithmetic and logical operations use source and destination operands held in computational registers, while transfers between these registers and store are effected by instructions which load corresponding address registers. Thus, by judicial coding, operands can be brought from store to the computational registers ahead of their being required, and the inherent latency of the store accesses can be programmed out. Of course, to be effective for high-level languages, this requires a considerable degree of compiler optimisation.

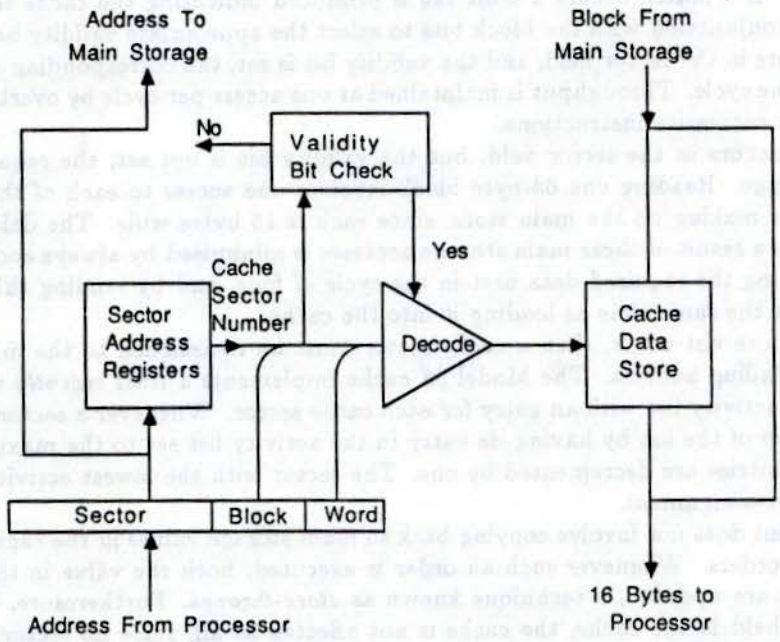

Figure 8: Cache Organisation in the IBM System/360 Model 85

Cache stores

In the System/360 Model 85 all program-generated addresses are real addresses referring to main (core) store locations, and the semiconductor cache store which is invisible to the programmer is used to hold the contents of those portions of main storage currently in use by the program. The cache mechanism operates by dividing both cache and main storage into logical sectors, each consisting of 1024 contiguous bytes starting on 1 Kbyte boundaries. During operation, cache sectors are assigned to main storage sectors in current use, and a sector address register associated with each cache sector contains the 14-bit address of the main storage sector to which it is assigned (figure 8). Since the number of cache sectors (16 or 32) is smaller than the number of main storage sectors (512 or 4096), most main storage sectors do not have a cache sector assigned to them. However, the localised nature of store accessing exhibited by most programs means that most processor accesses are handled by the cache (which operates at the 80 ns processor rate in the Model 85) rather than by the (1.04 microsec cycle time) main store.

Each sector within the cache is made up of 16 blocks of 64 bytes and each block is marked with a *validity* bit. Whenever a cache sector is re-assigned to a different main storage sector, all its validity bits are re-set, and the block containing the required store word in the new sector is accessed from main storage. The validity bit for this block is then set and the sector address register updated. Further blocks are accessed and their validity bits set as required.

The sector address registers constitute an associative store. Whenever an address is generated

which requires an operand to be fetched from store, the sector bits within the address are presented for association. If a match occurs a 4-bit tag is produced indicating the cache sector address, and this is used in conjunction with the block bits to select the appropriate validity bit for examination. If a match occurs in the sector field, and the validity bit is set, the corresponding data is read out in the next machine cycle. Throughput is maintained at one access per cycle by overlapping association and reading for successive instructions.

If a match occurs in the sector field, but the validity bit is not set, the required block is read from main storage. Reading one 64-byte block involves one access to each of the four interleaved storage modules making up the main store, since each is 16 bytes wide. The delay experienced by the processor as a result of these main storage accesses is minimised by always accessing the storage module containing the required data first in the cycle of four, and by sending this data directly to the processor at the same time as loading it into the cache.

If a match does not occur, then a cache sector must be re-assigned to the main storage sector containing the failing address. The Model 85 cache implements a *least recently used* algorithm by maintaining an activity list with an entry for each cache sector. Whenever a sector is referenced it is moved to the top of the list by having its entry in the activity list set to the maximum value, while all intervening entries are decremented by one. The sector with the lowest activity list value is the one chosen for re-assignment.

Re-assignment does not involve copying back to main storage values in the cache updated by the action of write orders. Whenever such an order is executed, both the value in the cache and that in main storage are updated, a technique known as *store-through*. Furthermore, if the word being updated is not held in the cache, the cache is not affected at all, since no sector re-assignment or block fectching takes place under these circumstances. While the store-through technique has the advantage of not requiring any copying back of cache values at a sector re-assignment, it also has the disadvantage of limiting the execution rate of a sequence of write orders to that imposed by the main storage cycle time.

The MU5 Name Store

A quite different approach to high-speed buffering was taken in the design of the MU5 computer [Ibbett, 1982], [Morris and Ibbett, 1979]. An examination of the operands used in high level languages, and studies of programs run on Atlas (the predecessor to MU5 at the University of Manchester), had indicated that over a large range of programs, 80% of all operand accesses were to named scalar variables, of which only a small number was in use at any one time. In a register machine these variables would be kept in the fast programmable registers in order to achieve high performance. However, this sort of hardware feature causes considerable compiler complexity, complexity which the designers of MU5 were seeking to avoid. The alternative scheme adopted in MU5 was to use a small associatively addressed buffer store containing only named scalar variables. MU5 instructions contain information about the operand type and accesses to non-scalar variables such as data structure elements are made via descriptors, which are themselves named variables, eligible to be stored in the Name Store. Because of this scalar variables and data structure elements can be buffered separately.

Addresses presented to the Name Store are all virtual addresses and the Name Store forms part of a one-level store with the Local Store of the processor. Simulation studies indicated that a hit-rate of around 99% would be obtained with 32 words of store, a number which it was technologically and economically feasible to construct and operate at a 50 ns rate. The address and value fields of the Name Store (Figure 9) form two adjacent stages of the Primary Operand Unit (PROP) pipeline. A virtual address generated in the previous two stages of the pipeline is copied into the Interrogate Register (IN), and concatenated with the contents of the Process Number register (PN), for presentation to the address field of the Name Store. A full virtual address in MU5 consists of a

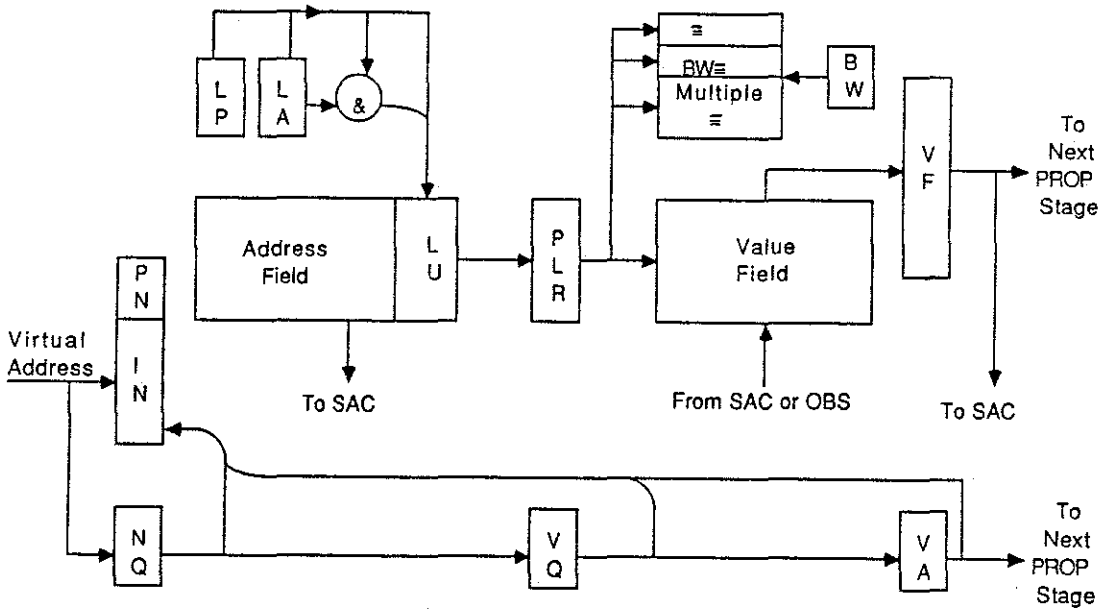

Figure 9: The MU5 PROP Name Store

4-bit Process Number, a 14-bit Segment and 16 bits which identify a 32-bit word within a segment. Only 15 of the word address bits are presented for association (thus referring to 64-bit words), a 32-bit operand being selected from within a 64-bit word in a later stage of the pipeline.

If the presented virtual address gives a match in the associative store, and the corresponding Line Used digit (equivalent to the *validity bit* in an IBM cache store) is set to 1, then an equivalence has occurred, and on the next pipeline beat a digit is set in the PROP Line Register, PLR. The digit in PLR then selects a register in the Value Field, and the 64-bit word is read out and copied into the Value Field Register (VF) by the next beat. At the same time, a check is made to determine whether an equivalence actually occurred. If no digit is set in PLR, this indicates non-equivalence, and the Name Store is updated by transferring a new word into it and discarding an old one.

When a Name Store entry is replaced the hardware must take into account the effect of store orders. To maintain the speed advantage of the Name Store, store orders only update the value of an operand in a Name Store, rather than operating on the store-through principle used in IBM cache stores. (more recent IBM machines, e.g. the 3090 [Tucker, 1986], have abandoned the store-through principle). Thus the content of a word may have to be read out and copied back to the Local Store before it is overwritten. The choice of which line to replace is made on the basis of a simple cyclic replacement algorithm, which requires a minimum of additional hardware for its implementation.

Performance

The performance of a buffer store may be characterized by two parameters: *hit rate* and *effectiveness*. The hit rate h is equal to the proportion of accesses which find their operand in the buffer store. If the access time to the buffer is t_b and the access time to main store is t_s, then the average access time t_a is given by

$$t_a = h\,t_b + (1-h)\,t_s$$

and the effectiveness e is given by

$$e = \frac{t_s}{t_a}$$

Simulation studies of the Model 85 cache carried out before it was built showed an average hit rate over 19 different programs of 0.986. With a main store access time of around 660 ns, this gave an effectiveness of 81%. There do not appear to have been any subsequent measurements on real Model 85 systems to confirm or deny these predictions, but certainly a different arrangement was introduced in the System/360 Model 195 [Murphy and Wade, 1970] and carried through into the System/370 Models 165 and 168. In these machines the cache is organised as a large number of small blocks, and a more complex, set associative, addressing mechanism is used. Measurements made on the Model 195 showed an average hit rate of 99.6% over a range of 17 job segments covering a wide variety of processing activities.

Performance measurements of the MU5 Name Store performance were made for a set of 95 programs containing both Fortran and Algol jobs ranging in complexity from simple student exercises to large scientific programs. For most programs it was found that around 80% of operand accesses were to named variables, that no more than 120 names were used in any one program, and that in all programs 95 per cent of name accesses were to fewer than 35 per cent of the names used. These figures confirmed the Atlas results which inspired the idea of using a Name Store, but the figures for hit-rates were not as good as had been anticipated. The hit rate measurements are complicated by the fact that a second Name Store was sited near the floating-point arithmetic unit. The hardware attempted to keep operands being used as floating-point variables in this second Name Store, while keeping operands involved in address calculations in the first Name Store. Thus although 96.1% of name accesses found their operands in one or other Name Store, only 86% of name accesses found their operands in the correct Name Store.

2.3 Function Execution

Having dealt with the problems of instruction and operand fetching, the next set of problems arise from function execution. It is convenient to consider an instruction pipeline as being that section of the processor which prepares functions and operands for execution in the function execution section. However, functions generally fall into two categories; computational functions and organisational functions. Computational functions, typified by floating-point operations, can be carried out independently of the instruction pipeline. Organisational functions, on the other hand, are typified by control transfers and addressing register manipulation functions which affect the pipeline itself. Thus, before considering the problems involved in achieving high floating-point performance, we examine problems internal to the instruction pipeline.

2.3.1 Organisational Functions

The effects of control transfers have already been dealt with in Section 2.1.3. Here we examine the problems associated with functions which operate on addressing registers and problems associated with multilength and multicycle instructions.

Addressing Register Problems

Operations on addressing registers may occur explicitly, as a result of the execution of functions which operate on base address registers, or implicitly, as a result of stacking or unstacking operations. The MU5 computer shows clear examples of both types. In both cases the registers concerned are located in the first stage of the Primary Operand Unit (PROP) pipeline (Figure 9). A function which operates on one of these registers must be completed before the next instruction in sequence attempts to make use of the register, but the instruction must itself proceed to the end of the PROP pipeline in order to pick up its operand. A hold-up is therefore required on the next instruction. This hold-up is could, in principle, be selective on the particular base register, but in practice in MU5 there is a general hold-up on all orders, because the *name + base* adder is also used to execute the function. Such orders therefore incur a delay equivalent to the length of the pipeline.

One of the addressing registers in MU5 is the Stack Front (SF) register, which may not only be operated on explicitly, but may also be implicitly incremented by a function which stacks an operand, or implicity decremented by an operand specification which unstacks an operand. These orders use the *name + base* adder to operate on SF, but unlike the explicit base register manipulation orders, do not need to traverse the PROP pipeline in order to obtain an operand. They can therefore alter SF *on the fly*. This updating of SF occurs before the order passes the Control Point, however, and so no irrevocable change to SF can be made. The problem could be solved by invoking a hold-up, as in the case of explicit operations, but this option was rejected by the designers on performance grounds. Some mechanism was therefore required to restore SF to its correct value should a control transfer occur before an order which implicitly altered it passed the Control Point. The technique used in MU5 is to build extra registers into the pipeline to carry any new value of SF through to the Control Point, and to preserve the new value with the Program Counter when the latter is updated for the order. After a control transfer this preserved value is used to restore SF to its proper value.

Multilength and Multicycle Instructions

Multilength and multicycle instructions degrade performance by requiring more than a single pipeline beat (the clock period in a synchronous machine) to go through any pipeline segment. The two categories may be distinguished as follows: in the first, technological/cost considerations have affected the design in such a fashion that it is not possible to transfer an instruction from one stage to the next within a single beat; the adverse effect on performance then arises from the fact that several beats are required to build the instruction at the succeeding stage and a gap inevitably appears within the pipeline. In the second, it may be possible that, due to the nature of the instruction being processed, a particular stage simply cannot complete its phase of the processing within a short beat time; once again, several beats are necessary to complete the instruction and a gap is inevitable in the instruction flow.

It is important to consider both of these types of instructions since they appear to be just as crucial, it not more so, with the newer technologies such as VLSI. Discussions of these must also be related to the issue of order codes; the case of multilength instructions in effect states that one should strive for short instruction lengths while the case of multicycle instructions implies that there are operations that are, ideally, not suited for implementation as single instructions.

MU5 again provides examples; the basic instruction parcel is 16 bits and the interface between the IBU and PROP was implemented for this width only. As a consequence, whenever 2, 3 or 5 parcel instructions are being processed there are instances where the first stage of PROP cannot complete its action in one beat. In these cases *dummy* orders are propagated forwards until a sufficient number of instruction parcels are available and, as would be expected, this is accompanied by a drop in performance. The obvious solution to this sort of problem is to increase, at added hardware costs, the width of the interface; this indeed was considered, but rejected by the designers of MU5,

who felt that the frequency of such instructions would not make it worthwhile. With hindsight this was probably the wrong decision; any solution that relies on there being a low frequency of long instructions is bound not to meet the goal of one completed instruction per clock and may suffer from poor performance when programming practices change.

The other solution is to strive for an order code in which all instructions have the same, short, width. The implications are that register-register instructions are to be preferred, with 2-register instructions considered better than 3-register instructions, and that instructions that reference store should, ideally, be limited to one store reference per instruction. Thus a complex order code such as that of the IBM System/360, which was not designed with pipelining in mind, cannot be considered as being conducive to pipelining, as witnessed by some of the complexity in machines such as the Model 195.

The situation is somewhat less straightforward in considering multicycle operations. Clearly a pipeline is only as fast as its slowest stage and one should therefore strive for single cycle instructions, as far as possible. On the other hand, the need to reduce the gap between high-level languages and machine-level instructions points to the need for relatively complex instructions[1]. Since some fundamental instructions (e.g. subroutine calling instructions) are intrinsically multicycle, what needs to be considered is not just the particular instructions that are to be included in the order code, but also the ease and effectiveness of *simulating* basic multicycle instructions within the chosen order code.

2.3.2 Computational Functions

The activities carried out in any instruction pipeline normally include fixed-point address arithmetic calculations, and the pipeline clock period is therefore of the same order as the time required for such an operation. Floating-point arithmetic operations take longer, however, so that in order to consume operands at the same rate as that at which the instruction pipeline can produce them, some means must be found to speed up these operations. Pipelining of the arithmetic unit is one answer to this problem, but one of the problems facing the designers of high-speed computer systems is the difficulty of achieving the fastest possible execution times for a particular technology in universal execution units. Circuitry designed to carry out both multiplication and addition, for example, will do neither as fast as two units each limited to one kind of operation.

Thus in some systems separate function units are used for different types of arithmetic/logical operation, and these units are then operated in parallel. The CDC 6600 and the IBM System/360 Model 91 were among the first computers to adopt this arrangement. The arithmetic units may themselves be pipelined, of course. (They were not in the CDC 6600, although pipelining was used subsequently in the CDC 7600 as a means of providing greater performance in an upwardly compatible system.) Pipelined or not, the time taken to complete different operations may be different, and this immediately leads to problems. Some instructions may require as inputs the results of previous instructions which have not yet completed, while others may produce outputs which will overwrite values required by instructions which have not yet started. Such dependencies between instructions are examples of the general problem of data dependencies in instruction pipelines.

2.3.3 Data Dependencies among Computational Functions

In processing instructions from a single stream, it is inevitable that data dependencies will occur between different instructions. Maintaining data consistency in the face of such dependencies results in performance degradations and/or complexities in the hardware, most frequently both. Hardware complexity is moreover aggravated if it is possible to execute instructions out of their initiation order, as happens with parallel function units.

[1] This point of view is argued at length by Myers in [Myers, 1982].

Essentially three types of data dependencies (hazards) are possible: *Read-after-Write* in which an instruction needs data that is to be produced by an instruction that precedes it in the stream; *Write-after-Read* in which an instruction attempts to overwrite data that has yet to be consumed by an instruction that appears earlier in the stream; and *Write-after-Write* in which an instruction attempts to overwrite data that has yet to be overwritten by an earlier instruction. The occurrence of a *Write-after-Write* to the same variable normally implies, of course, that there has not been an intervening Read of that variable, so that the first Write is in fact unnecessary. Where a *Write-after-Write* could in principle occur in hardware, however, the designers have generally displayed less than complete faith in software writers and have ensured that the hardware appears to maintain proper sequentiallity.

In any of these situations maintaining consistency clearly requires that the second instruction in question be prevented from completing before the first one. Solutions tend to fall into two categories: in machines which issue and execute instructions in strict order of initiation, issue of the second instruction is simply held up until the first instruction completes with a resulting degradation in performance in most instances. In machines where instructions may be completed out of their initiation order it is possible to alleviate performance loss by continuing to issue independent instructions whilst constructing an internal (hardware) data flow graph for instructions with dependencies. The latter case typically reverts to the former, when the quantity of low-level parallelism reaches a hardware limit determined by physical space available to hold the internal data flow graph. In the remainder of this section we describe the solutions adopted in the CDC 6600, the IBM System/360 Model 91 and MU5, discuss their weaknesses, and compare them.

The CDC 6600 Scoreboard

In the CDC 6600 instructions are taken in sequence from the Instruction Stack and issued by the Scoreboard to the appropriate execution unit (Figure 10). Each unit takes its input operands from among the 24 scratch-pad registers (eight 60-bit X (operand) registers, eight 18-bit A (address) registers, and eight 18-bit B (index) registers) and returns its result to one of these registers. The maximum rate of issue is one instruction per minor clock cycle (100 ns), while the units take typically 300 or 400 ns to complete their operations. In order to take advantage of the multiple function units, instructions may be executed out of issue order, and may be issued before obtaining their operands if a function unit is available. The Scoreboard is responsible for maintaining the dataflow graph necessary to ensure sequential consistency. Detection and resolution of hazards is performed via *reservations* which are placed on registers at the time of instruction issue. To permit out-of-sequence-execution, various *flags* and *function designators* are also used for each function, identified by a unique number. An example is illustrated in Figure 11.

The F registers identify the sink and source registers, the Q registers identify the function units producing the two inputs, and the Read Flags indicate the availability of the inputs. Each register also has reservation bits identifying the function unit that has reserved it as a sink. Assuming that there is a function unit available, instruction issue is preceded by the setting of the F registers (from the instruction), the Q registers (from the reservation bits of the appropriate computational registers), and the reservation bits of the sink register. Once an instruction has been issued it may proceed to execute only if the Read Flags for its function unit are set and it may store its result only if its sink register is not reserved. On completion, the function unit releases the reservation it has on the sink register and broadcasts to other function units which in turn set their Read Flags if they had been waiting for the output of the former function unit; additional details on the Scoreboard operation may be found in [Thornton, 1970]. In effect, the Scoreboard issues instructions as fast as it can and constructs a data flow graph (with function units as nodes and registers as edges) for instructions with data dependencies.

The Read-after-Write in the 6600 occurs when an instruction requires the result of a previously

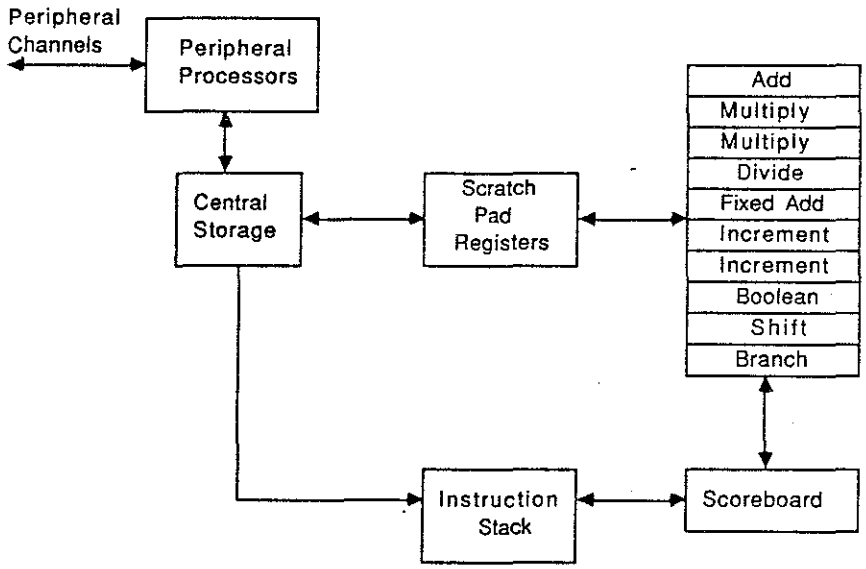

Figure 10: The CDC 6600 Central Processor

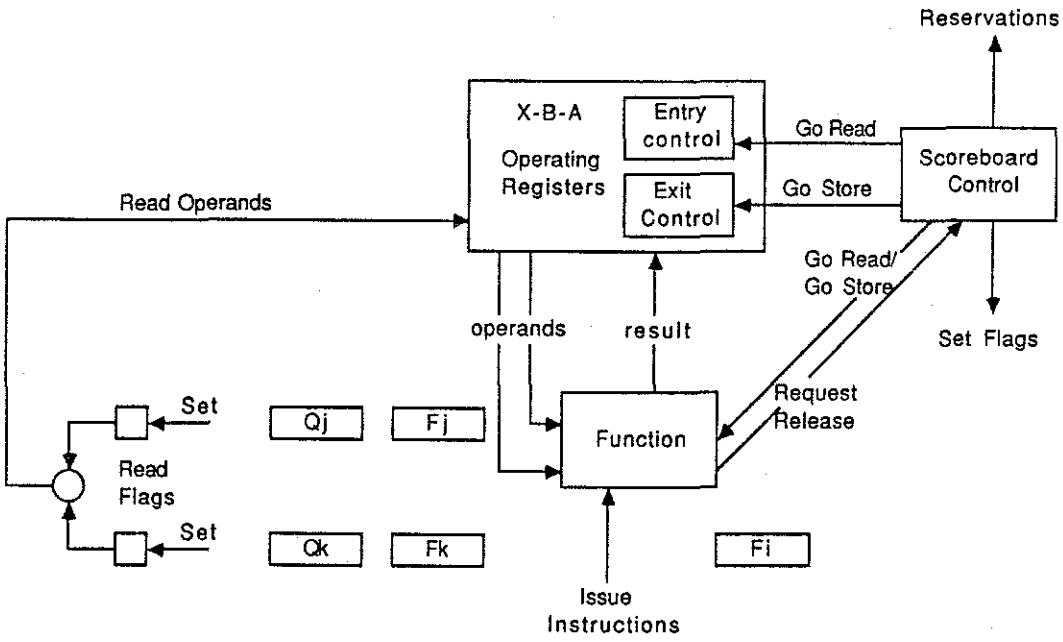

Figure 11: CDC 6600 - Functional Unit Reservation Designators

issued, but as yet uncompleted instruction, as input. In this case the function is issued but its Read is held up in the function unit until the Write completes and releases the reservation it has on the register in question. Because there are several function units, a number of instructions may be issued following the Write and hence the removal of the reservation may permit the register to be read from several function units.

The Write-after-Read occurs when an instruction needs to store its result in a register that is to be used as input by a previously issued, but as yet unstarted instruction. In the case of an unstarted instruction both instructions are issued but the Write is held up until the reservation placed on behalf of the Read has been removed.

The Write-after-Write occurs when an instruction requires the use of the same result register as a previously issued but as yet uncompleted instruction. In such a case, the reservation placed on behalf of the first instruction is detected in time for the issuing of the second instruction to be held up. This is a more severe form of hold-up than that used for a *Write-after-Read*, but as the result of the first Write is clearly not used, it is an unlikely event, so the length of the delay is irrelevant.

The IBM 360/91 Common Data Bus and Reservation Stations

Because all computational instructions in the 6600 involve register-register operations, all dependencies occur between registers and function units, rather than memory locations. Hence the task of resolving these dependencies remains tractable. In the IBM System/360 Model 91, computational instructions can also be store-register, but because operands from store are temporarily buffered in a special set of registers, all instructions can, in fact, be regarded as two-address register-register, and again the problem of resolving dependencies remains tractable. The sink specifies both an input and output register while the source specifies an output register, and it therefore suffices for our purposes to consider only register-register operations.

The organisation of the Model 91 floating-point unit [Tomsulo, 1967] is shown in Figure 12. Instructions are prepared for this unit by the Instruction Unit pipeline and entered in sequence, at a maximum rate of one per clock cycle, into the Floating-point Operand Stack (FLOS). Instructions are taken from the FLOS in the same sequence, decoded, and routed to the appropriate execution unit. The Instruction Unit maps both storage-to-register and register-to-register instructions into a pseudo-register-to-register format, in which the equivalent of the R1 field always refers to one of the four Floating-point Registers (FLR), while R2 can be a Floating-point Register, a Floating-point Buffer (into which operands are received from store), or a Store Data Buffer (from which operands are written to store). In the first two cases R2 defines the source of an operand; in the last case it defines a sink.

The most significant feature of this floating-point system is the *Common Data Bus* (CDB). The CDB is fed by all units which can alter a register, and itself feeds the floating-point registers, the store data buffers and all units which can have a register as an input operand. The latter connections allow data produced as the result of any operation to be forwarded directly to the next execution unit without first going through a floating-point register, thus reducing the effective pipeline length for *Read-after-Write* dependencies, as found, for example, in the scalar product loop. The running total in this loop would not actually appear in a floating-point register in the Model 91 until the last execution of the loop.

The operation of the CDB is controlled by the use of tags. A tag is a 4-bit number generated by the CDB control logic to identify separately each of the eleven sources which can feed the CDB. Thus there are six floating-point buffers, three parallel *reservation stations* (containing input buffer registers) associated with the adder, and two parallel reservation stations associated with the multiplier/divider. Tag registers are associated with each of the four floating-point registers, with the source and sink input registers of each of the five reservation stations, and with each of the three store data buffers. There is also a busy bit associated with each of the floating-point registers. This

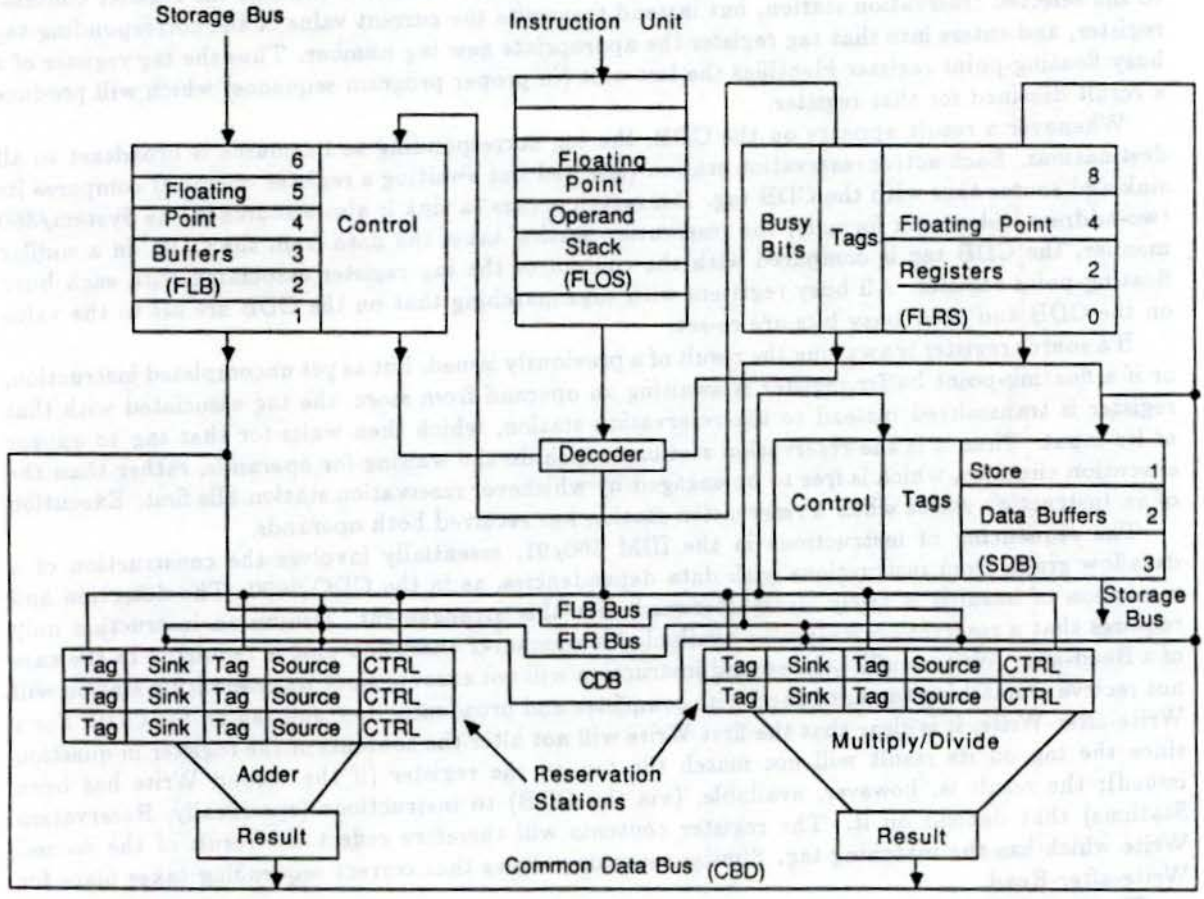

Figure 12: Organization of the IBM S/360 Model 91 Common Data Bus

bit is set whenever the FLOS issues an instruction designating the corresponding register as a sink, and is re-set when a result is returned to the register.

Whenever the FLOS decodes an instruction it checks the busy bit of each of the specified floating-point registers. If the bit is zero, the content of the register is sent to the selected reservation station via the Floating-point Register (FLR) Bus. On issuing the instruction the FLOS sets the busy bit of the designated sink register, and enters into its tag register the tag number of the selected execution unit. If the FLOS finds a busy bit set, however, it does not transmit the register contents to the selected reservation station, but instead transmits the current value of the corresponding tag register, and enters into that tag register the appropriate new tag number. Thus the tag register of a busy floating-point register identifies the last unit (in proper program sequence) which will produce a result destined for that register.

Whenever a result appears on the CDB, the tag corresponding to its source is broadcast to all destinations. Each active reservation station (selected but awaiting a register operand) compares its sink and source tags with the CDB tag. If a match occurs (a sink is also a source in the System/360 two-address instruction format), the reservation station takes the data from the CDB. In a similar manner, the CDB tag is compared with the content of the tag register associated with each busy floating-point register. All busy registers with tags matching that on the CDB are set to the value on the CDB and their busy bits are re-set.

If a source register is awaiting the result of a previously issued, but as yet uncompleted instruction, or if a floating-point buffer register is awaiting an operand from store, the tag associated with that register is transmitted instead to the reservation station, which then waits for that tag to appear at its input. Thus it is the reservation stations which do the waiting for operands, rather than the execution circuitry, which is free to be engaged by whichever reservation station fills first. Execution of an instruction starts when a reservation station has received both operands.

The sequencing of instructions in the IBM 360/91, essentially involves the construction of a dataflow graph from instructions with data dependencies, as in the CDC 6600. The detection and resolution of hazards is fairly straightforward with this arrangement. Issuing an instruction only requires that a reservation station be available for whichever execution unit is required. In the case of a Read-after-Write conflict,the second instruction will not execute since its reservation station will not receive a matching tag until the Write completes and broadcasts its result tag on the CDB. For a Write-after-Write, it is clear that the first Write will not alter the contents of the register in question since the tag on its result will not match the tag on the register (if the second Write has been issued); the result is, however, available, (via the CDB) to instructions (specifically, Reservation Stations) that depend on it. The register contents will therefore reflect the result of the second Write which has the matching tag. Similar reasoning shows that correct sequencing takes place for Write-after-Read.

The use of the CDB and the associated tagging mechanism has been shown to reduce the execution times of the inner loops of programs used to solve partial differential equations, for example, by about one-third.

The MU5 Name Store

In MU5, write operations to scalar variables have direct effect only on the Name Store, with main store being updated only when a word in the Name Store is to be overwritten, thus all dependencies must be resolved at the Name Store. A *Read-after-Write* dependency occurs in the following circumstances. When an order which writes to store the content of the B (index) register, for example, arrives at the Name Store, the value in B will not be correct because other orders which can alter the value may not yet have been executed. In order not to cause an immediate hold-up, a copy of the Line Register is preserved (in a register designated BW), a B⇒ OUTSTANDING digit is set, and the order proceeds to the B-unit. A Read-after-Write dependency occurs if a subsequent

instruction attempts to read from the word addressed by BW, before the required value of B has been returned to the Name Store. This occurs when the order is executed in the B-unit. The B value is sent to PROP, the PROP pipeline is held up and the value is written in to the line addressed by BW. The B⇒ OUTSTANDING digit is then reset and the pipeline is restarted. A Write-after-Write dependency occurs if a second Write instruction to the line indicated by BW arrives at the Name Store stage while the B⇒ OUTSTANDING digit is set. Write-after-Read cannot occur since the Read instruction arrives at the name store (and hence reads the line to be overwritten by the Write) in advance of the Write; writing is therefore guaranteed to be safe. In the first two cases the following instructions simply cause a hold-up of the preceding pipeline stages; in MU5 there is in fact a slight overkill since *any* Write entering the pipeline while there is an outstanding Write will cause a pipeline hold-up; we will refer to this situation, where there is no direct conflict on one line, as a *pseudo Write-after-Write.*

2.3.4 Commentary

Complexity of the three mechanisms discussed above clearly place the MU5 approach as the simplest, the IBM 360/91 as the most complex, and the CDC 6600 as in between. Beyond this there are several other bases for comparison.

- **Code-Generation by Compilers**
  From a compiler-writer's viewpoint, the MU5 arrangement is clearly the best since the absence of addressable general purpose registers relieves one of the burden of having to manage these.

- **Order of instruction issuing**
  In the MU5 instructions are issued only after they have obtained their operands and are executed in strict sequence; clearly with only two, essentially unrelated function units, there would little advantage in being able to execute instructions out-of-order. Instruction execution in the CDC 6600 on the other hand necessarily has to be out of order if full advantage is to be taken of the multiple function units. However, in the case of the IBM 360/91 it is doubtful that the ability to issue instructions out of order confers any worthwhile advantages with only two function units employed; the architecture, however, is easily extensible to accommodate more units.

- **Hazard Detection and Resolution**
  In terms of how the three machines detect and resolve hazards the following can be observed. Write-After-Read does not occur in the MU5 since instructions are only issued after they have obtained there operands; it appears in the CDC 6600 and in the IBM 360/91 because these issue instructions prior to operand fetching. For Read-after-Write, MU5 has a pipeline hold-up on the Write since operands must be read before instructions are issued whereas in the CDC 6600 and IBM 360/91 the Read (in fact several Reads) may be issued prior to the completion of the Write; however, once again MU5 is not necessarily at a disadvantage since it does not have multiple function units to make effective use of such a capability; thus the complexity is of limited value in the IBM 360/91 but finds more justification in the CDC 6600. Write-after-Write is handled in essentially the same manner in both MU5 and the CDC 6600; that is, the second Write is not issued and causes a pipeline hold-up. In the IBM 360/91 there is no hold-up and in fact any number of subsequent Writes may be issued as long as there are reservation stations available.

While there is no doubt that the designers of all three machines attempted to achieve the best design possible at that time and for the particular machine and no doubt considered many alternatives and *improvements*, a useful point of comparison is to speculate on the ease with which the

various mechanism could be extended in redesigned machines and whether such extensions would be worthwhile. Such speculations must also take into account the fact that there is more scope for change in the simplest systems and very little in the more complex ones.

The only straightforward addition that could be made to the IBM Common Data Bus would be an extention of the maximum size of the internal dataflow graph. This could be done by adding reservation stations and function units. Although, in principle, this seems quite straightforward, there is the added cost of storing and comparing larger tags. If such a system were extended to its logical extreme the result would be a dynamic Dataflow architecture.

Modifying the MU5 Name Store in order to handle Read-after-Write conflicts along the same lines as in the IBM 360/91 would clearly be a far from straightforward task; although avoiding a hold-up in the case of pseudo Write-after-Write conflicts seems relatively straightforward. It ought to be possible to do away with the BW register and instead add another status bit to be set whenever the line is to be reserved for writing and examined whenever the line is addressed; this would permit several B-Writes to progress through the pipeline as long as their targeted lines were distinct. Some other alternative must now be found to serve the BW register's role of line identification; one straightforward possibility would be for each Write instruction to carry along the address of its target line. Allowing more then one Write to be issued in the case of genuine Write-after-Write conflicts would require something along the lines of several BW bits per line. With such a scheme each Write would set one of the BW bits of its target line as it left the stage and a hold-up would be necessary only if all the BW bits for a line were set. Such a scheme, however, is not likely to be great value unless more than one function unit is employed; with multiple function units sequencing becomes a bigger issue.

In certain RISC processors a technique known as *delayed loading* is used to overcome the Read-after-Write problem. It is a direct analog of the *delayed branch* technique, used to overcome the control transfer discontinuity in RISC machines. In a normal sequence of instructions we might expect to find a LOAD instruction, followed by an instruction which uses the data just read from memory. In a pipelined implementation, the dependent instruction must be held up until the data has arrived. If this delay is predictable, then a totally independent instruction could overtake the dependent instruction and hence inprove throughput.

Clearly, this can only occur if a suitable instruction can be found for the otherwise "dead" cycle following a LOAD instruction. This is another example of how it is possible to migrate the responsibility, for ensuring integrity, from the hardware into the software. As with the delayed branch technique, the probalility of filling the pipeline after a LOAD instruction, with totally independent instructions, decreases as the pipeline length increases. Effectively, the ability to exploit the hardware parallelism decreases as the degree of parallelism increases. We conclude that this technique has little to offer in the long term, although it may provide short term performance improvements for architectures with relatively short pipelines.

2.3.5 Data Flow in Vector Pipelines

Dealing with speed disparities between processor and store speed in vector pipelines simply requires the extension of the mechanisms used in instruction pipelines to handle vectors instead of simple scalars; thus the use of scalar registers easily extends to vector registers (e.g. in the CRAY systems) and the use of the caches easily extends to vector caches (e.g. in the vector processing system proposed in [Lin, 1986]). It is worth noting, however, that vector operations in the top-end CDC machines, such as the CYBER 205, are store-to-store operations; the implications of this are discussed below. Beyond this there are two main issues that have to be considered in the design of vector pipelines; these are the start-up time and the granularity of synchronization. For the former, the main problem is simply one of trying to mask unavoidable gaps (in the flow of results) that are inherent in pipelining; for the latter, the problem is one of trying to minimise overall processor

idle-times. If synchronization is performaed on entire vectors then idle-time is a function of the average vector length; if it is performed on individual vector elements then it is determined only by algorithmic considerations [Topham, 1986]. Machines such as the CRAY Series provide fine-grain instruction interlock through the provision of a chaining mechanism, and in the case of the CDC Series, through the provision of a short-stopping mechanism and programmer-controlled linking.

The main consideration when incorporating fine-grain synchronization into a vector machine is the hardware cost incurred. Simple schemes which synchronize on whole vectors are relatively inexpensive, but more complex schemes which synchronize a number of processors on an element by element basis can be costly.

In the following subsections, we consider the design alternatives that have been taken in three machines, the CRAY-1, the CDC CYBER 205, and MU6-V. These are fairly representative; indeed some of the recent Japanese supercomputers, for example the Fujitsu [Miura, 1986], NEC [Watanabe et al, 1986] and Hitachi machines [Odaka et al, 1986], have architectures that are somewhat reminiscent of the CRAY-1.

The effect on performance of the architectural techniques used in each of these machine classes can be approximated by considering Equation 5 again. Let us first consider machines with register-to-register vector operations, typified by the CRAY Series. If the length of each vector register is l (elements) then a vector operation of length N must be partitioned into $\lceil \frac{N}{l} \rceil$ sequential instructions, so:

$$T_{p_l} = \left\lceil \frac{N}{l} \right\rceil t_a + \left\lceil \frac{N}{l} \right\rceil n\, t_p + \left\lceil \frac{N}{l} \right\rceil (l-1)\, t_p \tag{7}$$

So, to lower the threshold value (N_0) of N beyond which the last term dominates and Equation 4 holds, we must either reduce t_a (which largely corresponds to a faster store) or increase l.

Let us now consider machines with store-to-store vector instructions, typified by the CDC Cyber 205. T_{p_l} is now given by given by:

$$T_{p_l} = t_a + n\, t_p + (N-1)\, t_p \tag{8}$$

Thus, to get the last term to dominate, for as low a value of N_0 as possible, t_a and/or $n\, t_p$ must be kept low.

The CRAY-1 Vector System

The organization of the CRAY-1 is illustrated in Figure 13. The newer CRAY systems [Thompson, 1986] have essentially the same basic organisation, with replication in the multiprocessor systems. The vector operation subsystem consists of four vector functional units (integer add, shift, logical, and population count) and eight 64-element (each of 64 bits) vector registers; the floating point units are also heavily pipelined and are used in vector mode operation as well as in scalar mode. In the normal mode of vector operation, operands are taken from two vector registers (or a scalar register and a vector register) and the result is returned to vector register, while in chained mode intermediate results can be transmitted directly between functional units. Consequently data dependencies are resolved at the registers. The latter relies on the use of a reservation system, that is similar to, but simpler than, that of the CDC 6600. At the point of instruction issue reservations are placed on the input operand registers, the output register, and the selected functional unit.

The CRAY-1, and all machines with explicit vector registers, suffer from performance penalties when operating on vectors longer than a single vector register (in the case of the CRAY-1 longer than 64 elements) since interactions with the store are not always overlapped with arithmetic operations. In fact, it has been pointed out [Hockney and Jesshope, 1981] that the CRAY-1 has only one sixth of the store-processor bandwidth that is required to match the processing rate of the vector

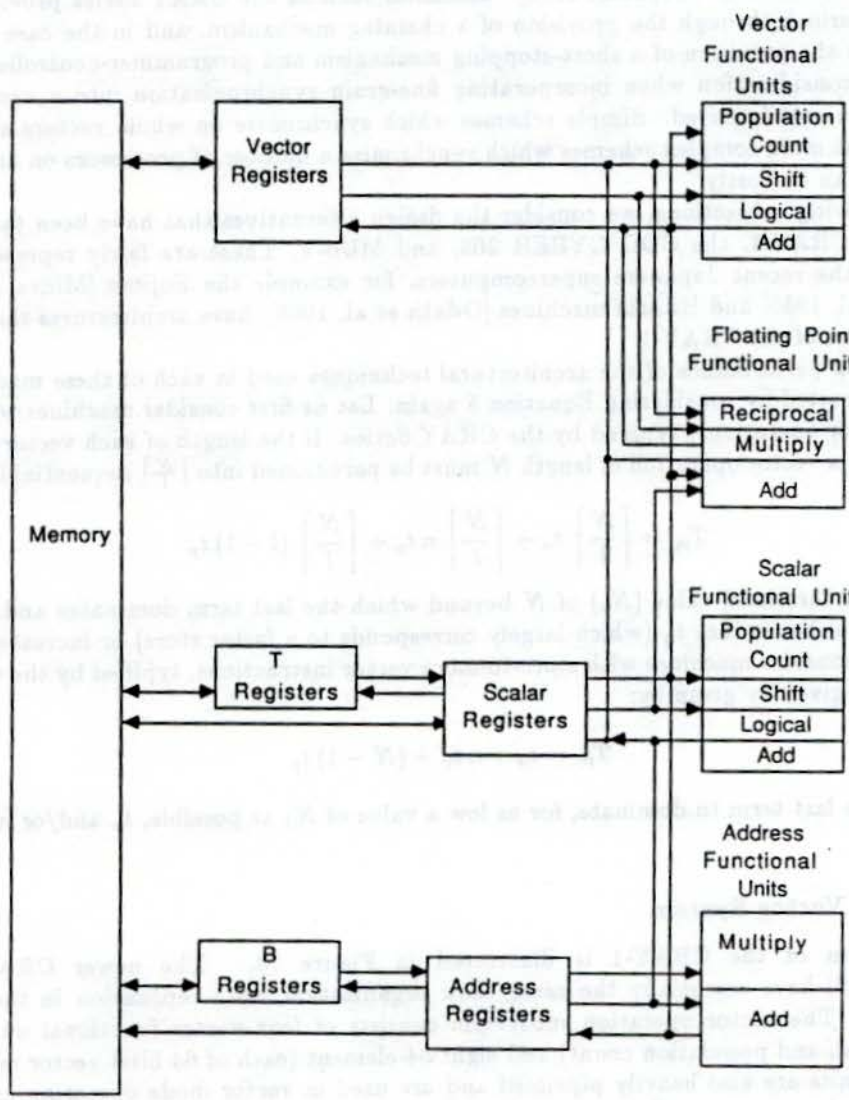

Figure 13: Organisation of the Cray-1

processing unit. As a consequence of this compilation is made harder by the need to minimize store traffic. Although the use of chaining mitigates the (relatively) poor performance from using a large synchronization granularity, it is a technique that is generally not applicable. An example of such an instance is the case where a vector operation operates on two vectors, one of which is the result of producing the element of a matrix in a column-wise fashion and the other is the result of producing the element of a matrix in a row-wise fashion; in general, the problem is one where the order of production does not match the order of consumption. For such a computation, even though the source operands may be produced at the rate of one element per clock, it is clear that the output cannot be produced at an equivalent rate. This sort of problem is easily overcome by the use of micromultiprogramming [Omondi and Brock, 1987] which would permit other vector operations to be executed on the same functional unit during what would otherwise be long gaps in the production of results. This also applies to the problems that occur with the use of vector registers. Also from Equation 7, it can be seen that vector register length plays an important role; therefore for a machine in which l is fixed it may be necessary for the programmer/compiler to either mask out the first term in that equation (by attempting to preload registers ahead of their use) or to supply vectors of appropriately large N (vectorization) since N is a program dependent parameter; both require considerable compiler sophistication. Although l is usually fixed by the hardware, it is worth noting that at least one class of machines (the Fujitsu VP series) permits l to vary although only within a small range.

The CYBER 205 Vector System

The organisation of the CYBER 205 Vector Processor is illustrated in Figure 14. A typical vector operation starts with the receipt, at the Vector Control Unit (which initiates and controls the execution of all vector instructions), of some vector function from the scalar processor. The relevant addressing information is then forwarded to the Stream Addressing Unit which thereafter generates the addresses of the required vector elements as well as the addresses of the results; this unit, therefore, interacts quite closely with the Input and Output Stream Units. The Input Stream Unit receives data from central store and performs any required alignment of the operands prior to forwarding these to the Floating Point Pipeline or String Unit. The Floating Point Pipeline consists of individual pipelined units to perform addition/subtraction, multiplication, and logical operations; logic is also provided for division and square root operations.

These units are capable of operating concurrently, and some vector operations may be chained (*linked triadic* operations in CDC terminology) as in the CRAY-1 but with some restrictions. A useful feature of this pipeline is the provision of a direct data path, a *shortstop*, from the outputs to the inputs of the individual units; this removes the need to store and refetch intermediate results and hence eliminates some conflicts that would otherwise occur. Another useful facility is the ability of an arithmetic pipeline to operate as either one 64-bit pipeline or two independent 32-bit pipelines. The String Unit processes strings of bits and characters (bytes); such processing includes the execution of editing instructions as well as arithmetic and logical instructions. Lastly, the Output Stream Unit buffers and realigns results, from the Floating Point Pipeline and String Unit, prior to writing this to the store.

Almost all criticism of the CYBER 205 has centered around its long start up time, about 1 μS. This is a problem that has plagued CDC machines for a long time and in fact the CYBER 205 itself is the result of re-engineering the STAR 100 [Hintz and Tate, 1972], which had a startup time of some 3-7 μS. This relatively long startup time is a consequence of operating directly on memory-based vector operands and, although it finds justification in avoiding some of the problems observed with the use of vector registers, the use of memory-based operands requires extremely high store bandwidths in order to minimize the first term in Equation 8. There seems to have been little difficulty in providing this although it necessitated some restrictions on permissible vector

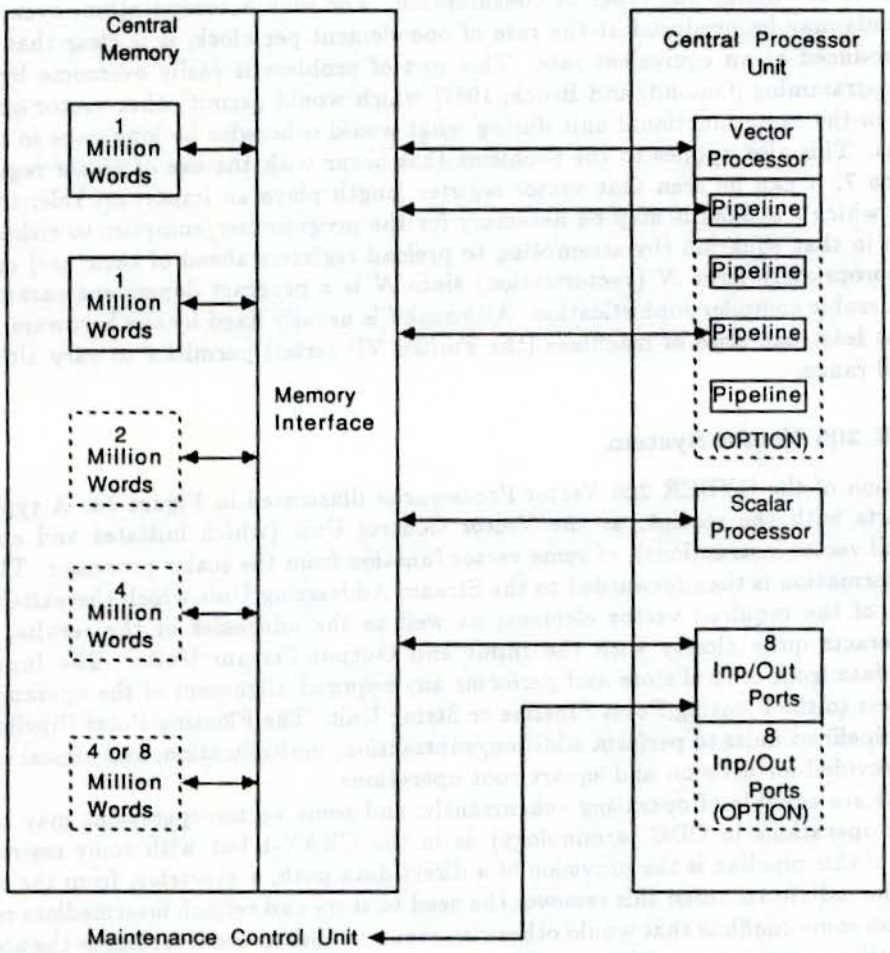

Figure 14: Organisation of the CYBER 205

strides. Whilst the store bandwidth on the CYBER 205 exceeds what is nominally required by the computational units, it clearly represents a potential bottlneck in trying to design even faster machines. Also, in examining Equation 8 it may be observed that reducing both t_a and t_p (as CDC did in going from the Star-100 to the Cyber 205) may fail to yield significant improvements (in terms of lowering N_0), particularly if n is not reduced (but such reduction would be contrary to Equation 4). A vector cache such as that suggested by Lin [Lin, 1986] might have been useful for this machine.

Yet another source of problems has been the use of virtual storage, an unusual feature for a supercomputer. Although this decision has been justified to some extent [Lincoln, 1982], the fact of the matter is that, all other things being equal, a machine with virtual storage will be slower than one without. We believe that many of performance problems in the machine can be overcome through the use of micromultiprogramming; one can in fact observe that one of the techniques that improves the performance of the CYBER 205 (over that of the STAR 100), that of overlapping the start of one vector operation with the finish of another, is a small step towards micromultiprogramming. Essentially the effect of full microprogramming would be to mask out the effect of the first two terms in Equation 5.

The MU6-V System

A major objective in the design of MU6-V [Ibbett et al, 1985] was to have a vector processing system in which performance could be increased arbitrarily by the addition of more vector units. The two factors that would limit this in a machine like the CRAY-1 or the CYBER 205 are the processing power of the scalar unit and the required bandwidth of a centralized store [Ibbett, 1981]. In MU6-V, therefore, scalar processing capabilities are provided as part of each vector processor and all store is distributed among the processors; this gives the structure illustrated in figure 15. The bandwidth required of the shared highway is, in general, substantially less than would be required of the store is a conventional shared-store multiple vector processor. Virtual addressing is achieved by a mechanism in which the *vector name* is analogous to a *segment number*. This makes translation by direct table look-up feasible, resulting in highly efficient memory management.

Another objective was to avoid the performance problems that arise when the techniques for communication and sychronization between processors rely on shared variables. The solution in MU6V involves a data-driven global-update protocol, with the granularity of communication under program control. To communicate a value, a processor broadcasts it on the common highway and any processors that need the value pick it up; processors, however, always wait for values to be broadcast, rather than attempting to read them. This results in a data driven system whose advantages (over a system in which processors make explicit requests for data) are that less time is required to communicate a value (one highway cycle instead of two) and store latency is eliminated; also several processors can receive a common value at the same time rather than making individual requests and contending for the bus. The synchronization required to ensure the correct ordering of events is fairly straightforward and may be inhibited whenever it is not required. Every element has a synchronization bit associated with it; this bit is set on writing and optionally reset on reading but the element cannot be overwritten by globally communicated data if the bit is set. Once a processor has consumed its local copy of value any attempt to access the (new) value causes the process to suspend until the new value has been broadcast and the synchronization bit set. In the event that some processor has computed a new value but not all processors holding copies are ready to update them, the producer processor postpones its broadcast and retries at a later time.

Although both results from a small prototype and analytical measurements [Topham, 1987] have shown that this system is capable of linear speedup, some rewriting of algorithms may be necessary in order to avoid frequent halting of the processors; in the absence of this, linear speedup may be limited mostly to compute-bound problems that perform a large number of operations before halting.

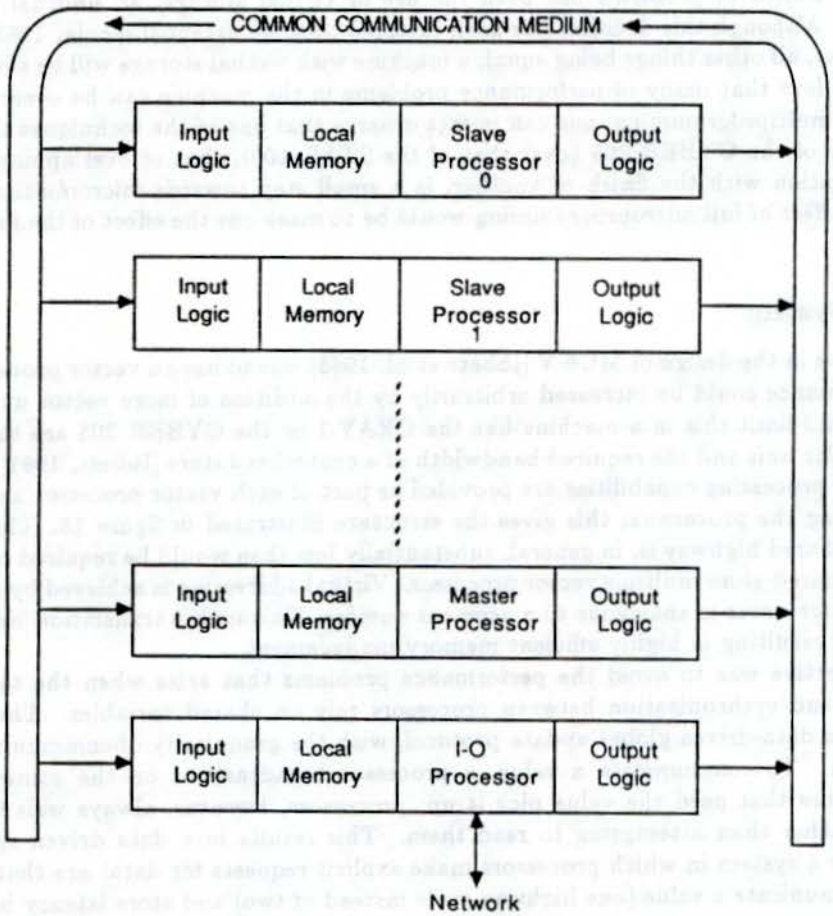

Figure 15: The MU6-V Vector Processor

Once again, such problems might be avoided completely if it were possible to micromultiprogram the vector processor.

3 Micromultiprogramming

For many years multiprogramming has been used as a technique for optimising the utilization of a shared CPU, and for providing an interactive multi-user capability. To get reasonable utilization each time-slice allocated to each process must be large in comparison with the time taken to change context. Therefore, multiprogramming at the operating system level is only useful for reducing inefficiencies caused by relatively infrequent discontinuities, typically I/O and memory management.

However, as we have seen, there are discontinuities with much finer levels of granularity than this. For example, when an instruction within a pipeline needs an operand from main store a pipeline discontinuity normally occurs. The latency associated with this discontinuity is normally of the order of several clock periods, and frequent operand accesses result in poor performance. It is often impossible to continue processing instructions from the same instruction stream when this happens and so an alternative strategy must be adopted.

One strategy that has been suggested, and used in at least one commercial machine, is *micro-multiprogramming*. This is based on a recognition of the fact that a high-latency memory cycle and a high-latency I/O operation are simply two examples of discontinuities within a sequential instruction stream, and suggests the use of a common strategy to deal with both. The micro-multiprogramming strategy addresses these two problems by initiating a context switch whenever a high-latency operation is encountered.

3.1 Previous Work

The term micro-multiprogramming (μ-multiprogramming) was, to the authors' knowledge, first introduced by T.C. Chen [Chen, 1971], to describe a mechanism for sharing the hardware resources of a parallel or pipelined system in a self-optimising way. Chen described how the throughput of an $n-$way interleaved store could be improved by a factor of $\sqrt{n}$ by micro-multiprogramming the memory requests using memory request queues. A novel form of micro-multiprogramming was suggested by Flynn in 1970 [Flynn, 1970], in which highly pipelined function execution units were shared between 32 "skeleton" processors. Flynn's idealised system is shown in figure 16. The system was never constructed, although extensive simulation results are available.

This technique was also used, to a limited degree, in the Xerox Alto [Thacker et el, 1982] and Dorado [Pier, 1983] machines and the Symbolics 3600 [Moon, 1985]. These machines used multiple microcode contexts to implement the sharing of CPU resources by different I/O device adapters. As Arvind and Iannuci [Arvind, 1983] point out, such a limited use of multiprogramming will not solve the problems of memory latency and process synchronization in a multiprocessor system. However, they go on to further hypothesise that these problems cannot be solved in a von Neumann style of architecture. They maintain that the number of low-level contexts required to sustain performance must grow, as the system is scaled, in order to match the increased memory latency. When describing their dataflow model they omit to mention that dataflow machines exhibit similar behaviour. This arises because the level of instantaneous parallelism within a piece of dataflow code must match the level of parallelism within the hardware. Naturally the node-to-node latency will grow as the dataflow machine is scaled, resulting in a requirement for greater application parallelism if performance is to be maintained. This problem appears to be universal, arising from fundamental register-transfer-level constraints, rather than being specific to certain types of architecture.

3.1.1 The Denelcor HEP

One of the best examples of μ-multiprogramming to date is the Denelcor *Heterogeneous Element Processor* (HEP) [Smith, 1985]. The architecture of this machine is illustrated in figure 17. The

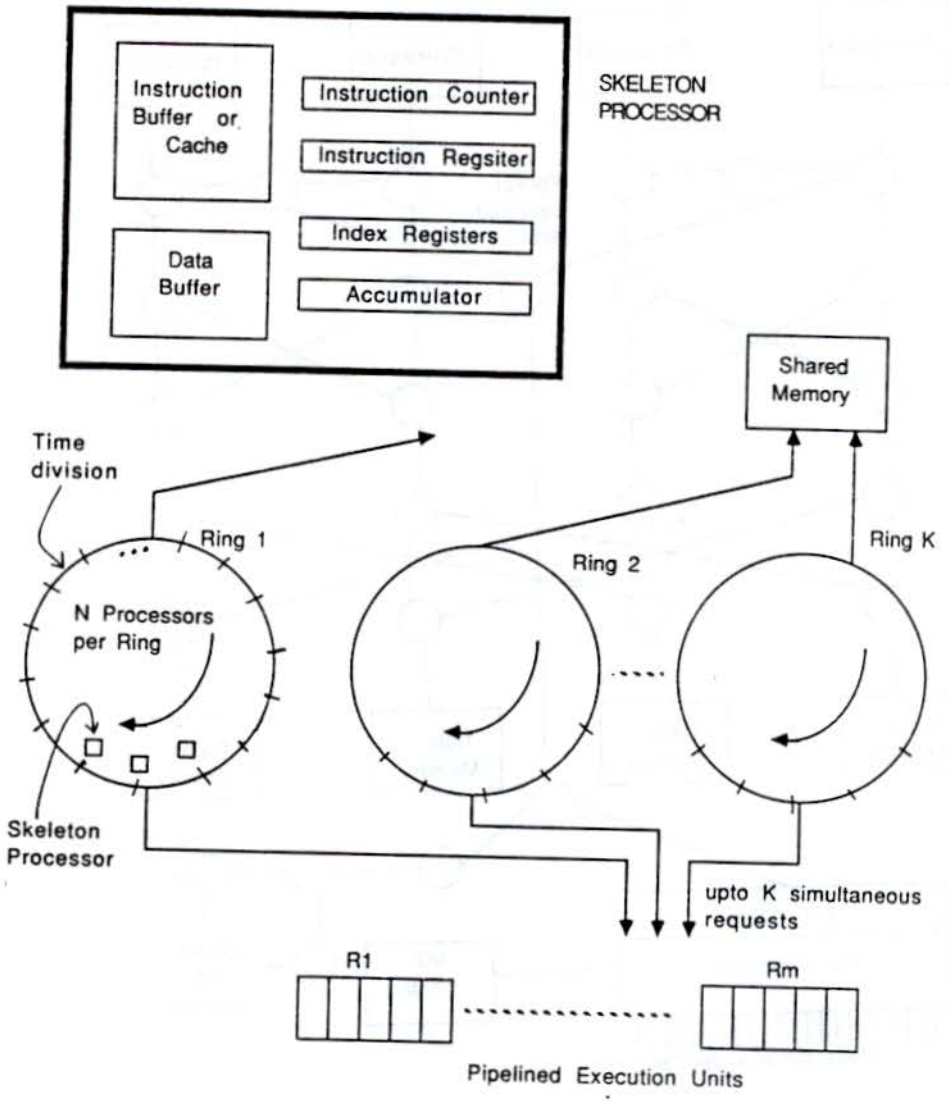

Figure 16: Flynn's 1970 Proposal

HEP system comprises up to 16 Process Execution Modules (PEMs) and up to 128 Data Memory Modules (DMMs), connected via a novel high-speed multi-stage packet switch.

As far as this paper is concerned, the most important feature of the HEP is the way in which a number of distinct processes are μ-multiprogrammed within each PEM. This is achieved by issuing instructions from a different process, on each clock cycle, which generates an implied context switch between every pipeline stage.

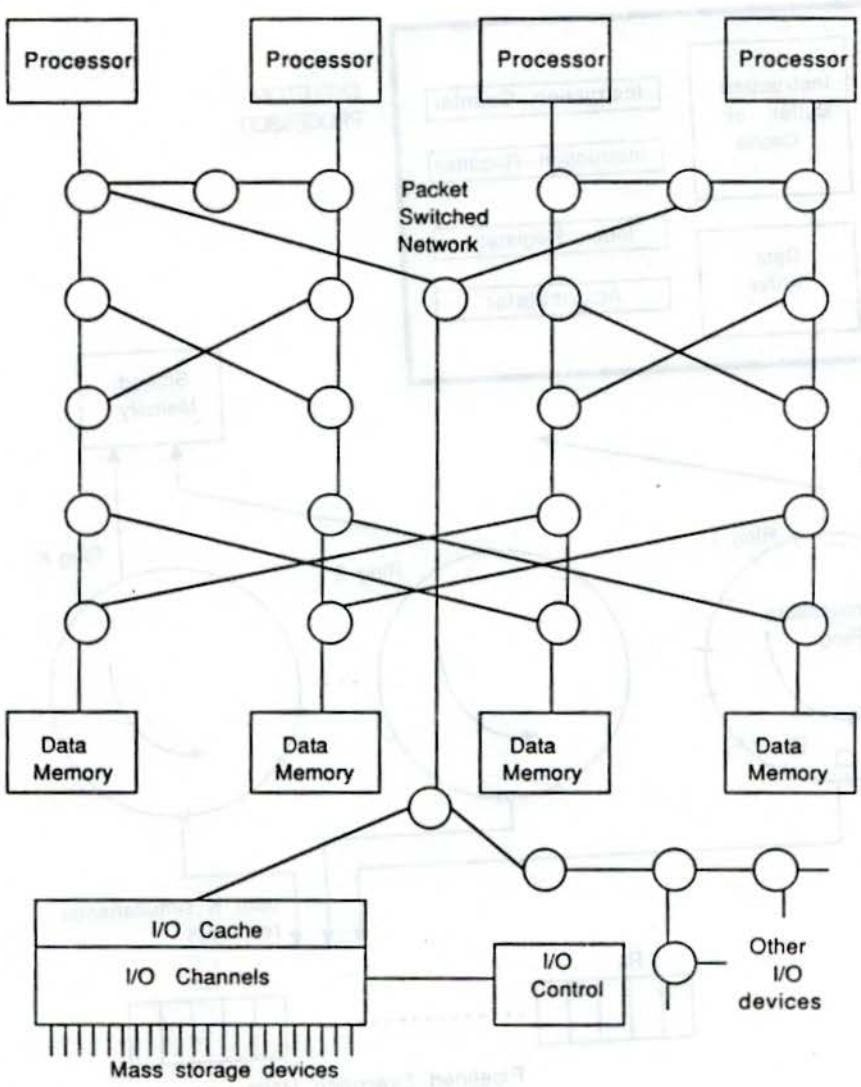

Figure 17: The HEP Computer System

Whereas previous instruction pipelines, such as the Primary Operand Pipeline in MU5 [Morris and Ibbett, 1979], had pipeline stages which evaluated a function f, such that :-

$$f : Instruction \mapsto Partial\ Result$$

The HEP pipeline stages evaluate a function g, such that :-

$$g : Instruction \times Process\ Tag \mapsto Partial\ Result \times Process\ Tag$$

In order that the domain of each process can be kept separate the HEP has a large partitioned register file; effectively a *virtual* register space. Each PEM also contains a number of parallel function units, some program memory, some constant memory, a set of control units and an interface to the interconnection network.

Within the HEP, work is divided into independent tasks each of which may contain up to 64 processes. A Task Status Word (TSW), associated with each task, identifies the relocation and protection domain, and the degree of privilege for processes within that task. The μ-multiprogramming feature requires a hardware scheduling mechanism, and the basic design criteria of this mechanism are :-

- It must be fair to all tasks.
- It must be fair to all processes within a task.
- It must be able to schedule (and de-schedule) processes at the pipeline clock rate.

The mechanism used in the HEP is shown in figure 18. The operation of this mechanism is described in detail in [Hwang and Briggs, 1984], the main points to note are that this mechanism ensures each process has at most one instruction in a state of partial execution at a time, and that this results in the optimal utilization of the instruction pipeline.

If an instruction needs to reference a data memory module (DMM) the process tag and an operation descriptor are sent to the Storage Function Unit (SFU). The SFU queues incoming requests in a secondary set of task queues. These requests are scheduled in the same way as instruction issue requests, and for each request the SFU sends a packaged memory request into the HEP Switch. Memory acknowledgements returning from the Switch are routed back to the operand fetch section, in the case of successful memory operations, or routed back into the SFU Queue in the case of unsuccessful memory operations that need to be repeated.

Unsuccessful memory operations occur when an attempt is made to operate on a "locked" memory location. This locking mechanism is used to ensure data integrity, and hence communication, in the HEP's shared-memory environment.

The processing of memory operations does not require any intervention from the execution section of the PEM, and this means that the processing of high-latency (non-local) memory requests does not degrade the throughput of each PEM, assuming that sufficient process-parallelism exists. The major flaw in the design of the HEP is the low level IPC mechanism, which is based on data sharing. In consequence the possibility of a "busy-waiting" condition exists, wherein blocked memory requests are re-issued at intervals which depend upon the loading of the SFU queue.

This places a certain pressure on the user of such a mechanism to ensure that the wait-times are always relatively short. In section 4.1 we present an alternative mechanism which corrects this flaw, and opens up some exciting possibilities for highly efficient interprocessor communication protocols. We believe that in order to achieve efficient parallel processing a process must not consume any computation or communication resources under the following conditions.

- when it is *idle*
- during *suspension* and *continuation*

3.1.2 The Cyclic Parallel Computer

The Cyclic Parallel Computer proposal from the University of Tokyo [Goto et al, 1986] is another example of how μ-multiprogramming could provide very high performance. This machine proposal

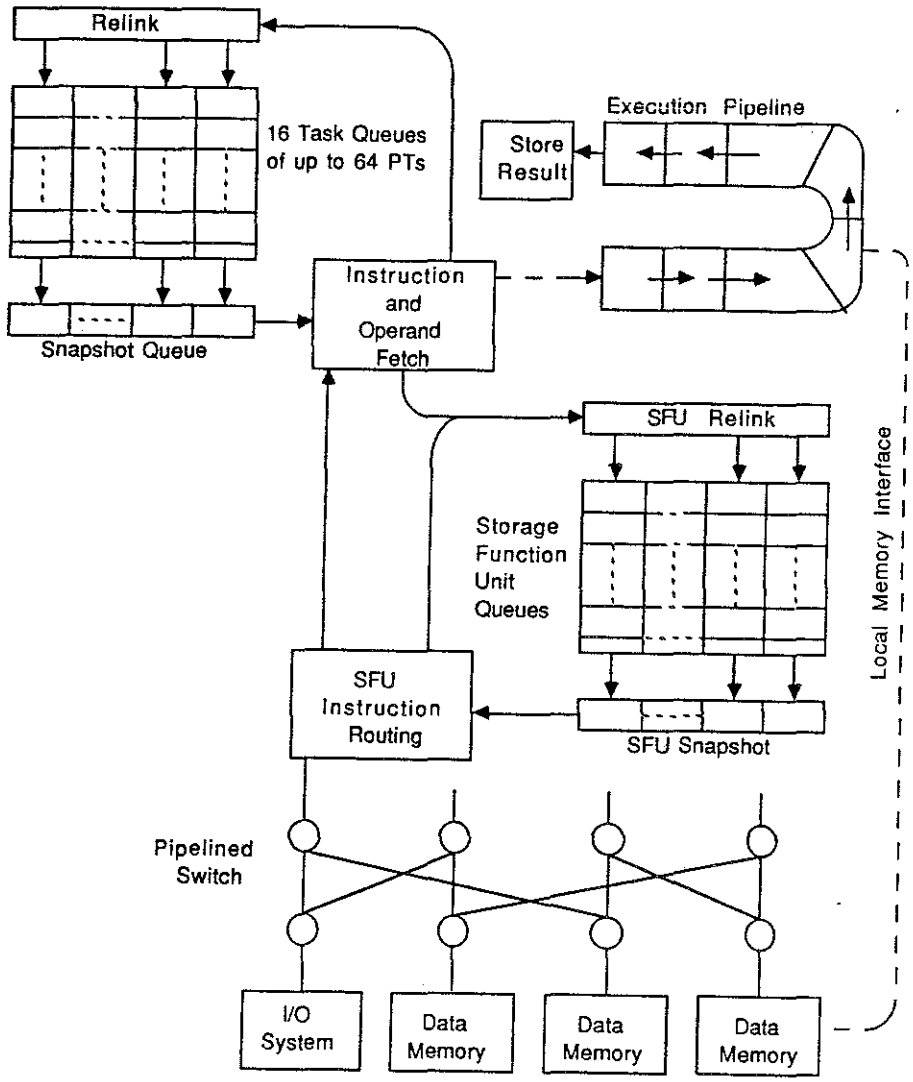

Figure 18: The HEP Scheduling Mechanism

combines a cyclic MIMD pipeline with Josephson technology to produce a quoted potential performance of 10 GFLOPS[2]

Josephson technology has the advantage that each logic device may act as a latch, enabling shallow-logic pipelining to be practised without the penalty of the extra cost and time delay that are required when placing pipeline registers in Si logic. Consequently all logic, both processor and

[2]The CPC's high clock rate of 10 GHz requires a physically small design. The CPC would fit inside a large coffee mug !

memory, can be naturally pipelined. The CPC proposal suggests the use of a pipelined memory in which successive read/write operations can be carrried out at intervals of the pipeline pitch time without causing conflicts. The CPC can be regarded as a scalar multi-process system with a common shared memory, in which interprocess synchronization is implemented by conventional mutual exclusion primitives. Waiting processes remain idle, which again assumes that wait times will always be relatively short.

The CPC architecture shown in figure 19 assumes a single μ-multiprogrammed processor, with the degree of parallelism fixed at around 40 processes. The CPC proposal is therefore of great interest in respect of its novel logic and packaging technology, but of less significance in respect of the extensibility of the architecture and the generality and efficiency of its interprocess communication mechanisms.

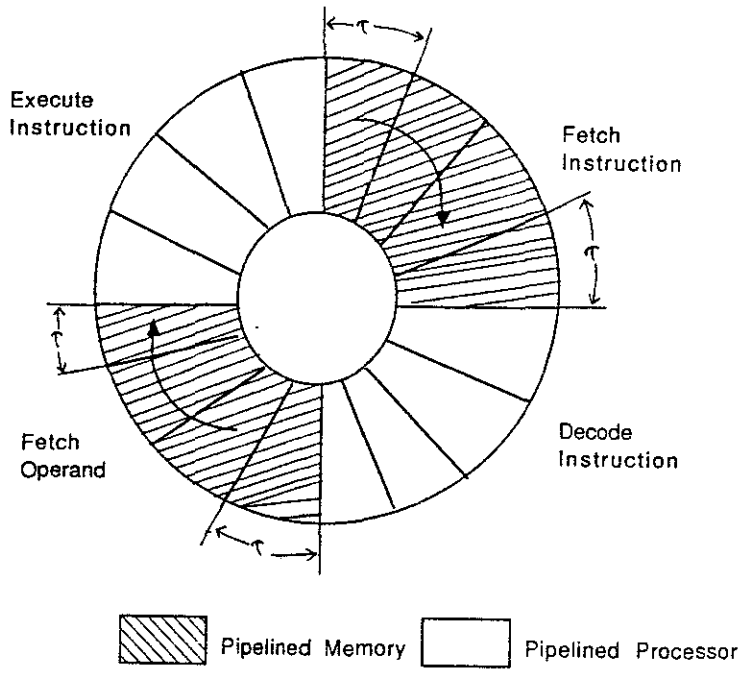

Figure 19: The Tokyo Cyclic Parallel Computer

3.1.3 The Circulating Context Multiprocessor

A machine, very similar in design to the HEP, has been proposed by Staley and Butner at the University of California, Santa Barbara [Stanley and Butner, 1986]. This system, shown in figure 20, embodies a conventional von Neumann programming model, but is implemented as packets of executable context in a tightly-coupled shared memory environment.

In common with the HEP, the CCMP uses FIFO queues to smooth the flow of instructions between pipeline stages, and consequently is capable of sustaining comparable throughput.

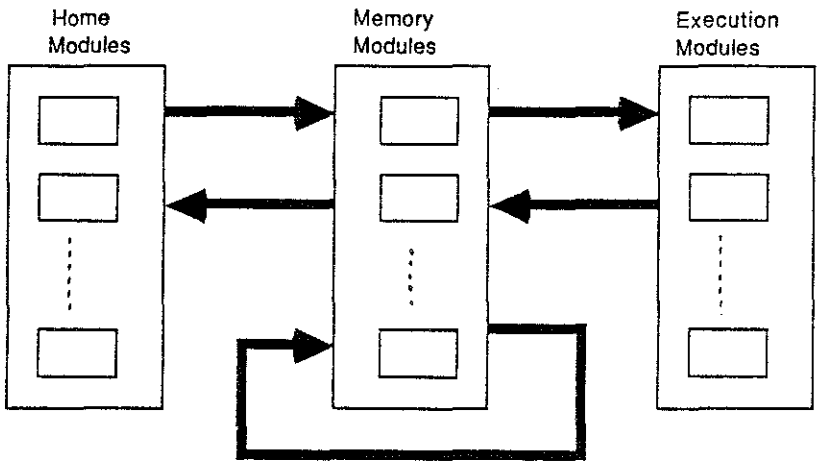

Figure 20: The Circulating Context Multiprocessor

3.2 The Rationale for μ-multiprogramming

The primary motivation behind the design of micromultiprogrammed machines has been efficiency; in such machines the relative independence of instructions in different instruction streams is exploited in order to use up the otherwise wasted processor cycles that occur during:

- operand accessing,
- control transfers, and
- inter-instruction data dependencies.

A rule is enforced, in all of these machines, that each process may have one, and only one, instruction in a partial state of execution at any instant. This contrasts sharply with conventional instruction pipelines, where as many instructions as possible are kept active for a single process.

The ability to change context at the pipeline beat frequency relies on having a minimum of volatile context associated with each process, and hence a minimal context-change overhead. In the case of the HEP, this volatile context consists of a single Process Status Word (PSW).

Although μ-multiprogramming is capable of reducing the overhead caused by general context switch requirements, this technique is particularly useful for eliminating instruction and data discontinuities in pipelined machines. As pipeline beat frequencies increase the discontinuity problems become more acute. It therefore seems inevitable that the use of this technique will become more

widespread. The commercial trend in advanced microprocessor architecture is currently retracing the evolution of the present-day mainframe, through the introduction of integrated virtual memory, instruction prefetch and buffering, and data caching mechanisms The combination of these techniques will provide a partial solution to the data-latency problem, however, they will also introduce the possibility of additional data-oriented discontinuities[3]. It is the authors' belief that these techniques merely address the symptoms of the problem, rather than the disease itself.

It is clear that there are now a significant number of machine designs and proposals which use the μ-multiprogramming technique, although each machine appears to incorporate the technique for a different reason. Furthermore, experience so far suggests that this technique could be exploited in the search for general-purpose parallel VLSI architectures.

The goal of this paper is to address the problem of producing a generic architectural form in which discontinuities, from whatever source, do not affect the overall throughput of the system unless the degree of multiprogramming falls below a fixed level.

If this were achievable then all problems associated with discontinuities would be replaced by the problem of partitioning an application code into concurrent processes. At this point the interprocess communication (IPC) mechanism becomes a possible source of discontinuities. In section 4.1 we explain how IPC can be implemented without interrupting the smooth flow of computation and without introducing any computing overhead.

3.3 Context Flow - A canonical form of μ-multiprogramming

In this section we attempt to formalize the notion of μ-multiprogramming as this will provide us with a tool for implementing a range of parallel architectures as well as a means for identifying architectures that cannot be implemented efficiently. Central to this theme is the concept of a sequential process [Hoare, 1985] with a referentially transparent context [Burstall et al, 1980], [Keller and Sleep, 1981].

We define a *context* to be the unique locus of control for a sequential process and all its volatile information. At the hardware level each context may be further subdivided into a *dynamic* context and a *static* context, the relevance of which will be discussed later.

We represent a hardware structure for evaluating a machine instruction set as a graph in which each node represents an atomic transformation applied to the context of a sequential process. We call this a *Context Flow Graph.* In effect, the nodes in a Context Flow Graph (CFG) are directly equivalent to the pipeline stages in a conventional (ad hoc) machine implementation. The arcs in a CFG represent the highways connecting the pipeline stages, and during each *graph cycle period* a context flows from the source node to the destination node. In cases where the producing node does not produce a valid context a null context flows instead.

A CFG is almost certain to contain cycles, in the same way that circular control pathways exist in all practical computers.

In general a context C is a tuple $\langle f, s \rangle$, where f identifies the next function to be applied to the process state variable s. At each node in a CFG the aggregate of the information stored at the node and the context C will be sufficient to evaluate f.

The volume of information in s will generally be large. For example, a conventional process may have several Megabytes worth of context information. This naturally precludes the movement of a complete process context between every node (and hence between every pipeline stage). This is where the concept of a dynamic and a static context become useful. We define the dynamic context of a process, C_d, to be $\langle f, s_d \rangle$, and the static context of a process to be $\langle i, s_s \rangle$ where $i \in \{0 \ldots N\}$ for an N-node CFG. We thus divide the context into a small partition containing frequently referenced

[3] A data-oriented discontinuity may involve a cache-miss, a TLB non-equivalence or consistency synchronization delay

information (s_d) and a large partition containing less frequently used information (s_s). The static state is then held in a memory associated with node i of the graph, and for any exchange of information between C_d and C_s, C_d must be within node i.

Thus, a CFG is a directed cyclic graph representation of an abstract machine evaluation algorithm. A CFG provides a method of representing, and hence exploiting, any parallelism which may exist within the evaluation algorithm. This is analogous to the way in which a Dataflow graph (DFG) is able to identify and extract parallelism within an application algorithm. However, in contrast to dataflow machines, application parallelism is not implicitly identified. Consequently no asynchronous matching of function parameters is required, a common source of implementation problems for dataflow machines.

3.4 Context Flow Principles

A Context Flow graph for an arbitrary evaluation algorithm can be constructed from three primitive node types. These are:

- The Transformation Node
- The Branch Node
- The Merge Node

3.4.1 The Transformation Node

A Transformation node consists of a single registered input path, some evaluation logic and a single output path, as shown in figure 21. The functions evaluated by each transformation node would normally be distinct, and each node may even implement several variants of a *class* of functions. For example, a single node may evaluate a group of arithmetic functions, one of which would be selected by a field within the context at that node.

It is easy to imagine how a group of Transformation nodes might implement a conventional instruction pipeline.

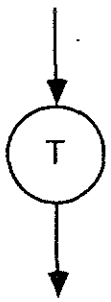

Figure 21: The Transformation Node

Node 1 : Fetch Instruction This node transforms the incoming context by appending the instruction at the location identified by the program counter within the context.

Node 2 : Access Operands This node transforms the incoming context by replacing the operand specification(s) with the corresponding data.

Node 3 : Evaluate Instruction The transformation applied here involves replacing the operands $x_1 \ldots x_n$ with the result, $f(x_1, \ldots, x_n)$.

Node 4 : Store Result At this node the result data is removed, and left within the node for subsequent retrieval. Normally an explicit address would identify its location.

Node 5 : Increment PC Here the address of the next instruction is computed. Conditional control transfers could also be implemented at this stage.

In a very simple architecture these nodes could be connected in a ring, as shown in figure 22. The throughput of such a simple architecture would be proportional to the number of concurrent contexts, up to a maximum of five. A Transformation node may contain static data, in the form of

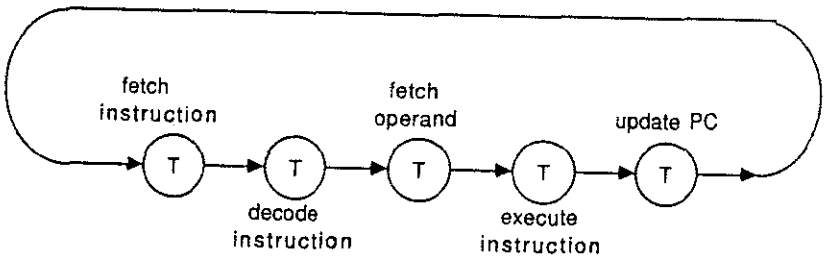

Figure 22: A Ring-Structured Context Flow Graph

an associated block of memory. Functional transformations for reading and writing to the memory can be defined very simply. The **read** transformation would use an address field in the input context and append the data at that address in memory to the output context. The **write** transformation would write a piece of data from the input context to a memory location identified by an address in the input context. In both cases the resultant context may contain a status bit to indicate that the transformation took place successfully. If a virtual memory scheme were adopted, a TLB non-equivalence could be signalled in the same way.

The data for each context may be distinct or shared. Thus, implementations of global memory architectures and referentially-transparent parallel combinator reduction architectures are both possible.

3.4.2 The Branch Node

In order to introduce spatial, as well as temporal, parallelism a means of composing parallel sequences of Transformation nodes is required. This is achieved through the use of Branch and Merge nodes. The informal specification of a Branch node is shown in figure 23, from which it can be seen that a Branch node has one incoming arc and two outgoing arcs. It operates by examining a decision variable within the incoming context. If is is true the context is passed to the left output arc, and if false it is passed to the right output arc. The output arc not receiving the incoming context receives a null context. This provides a choice mechanism, enabling more elaborate algorithms to be implemented. It is also possible to define a **fork** node which dynamically creates a parallel context,

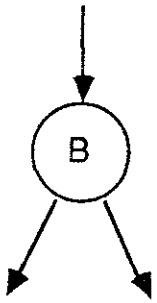

Figure 23: The Branch Node

one context being output on each arc. The dynamic deletetion of a context is trivial to implement; the context is simply routed to a node with no output connection.

3.4.3 The Merge Node

A Context Flow graph may be *open* or *closed*. An open graph is one which contains at least one node with an unconnected output arc. Any context flowing along this arc is effectively terminated. In a closed graph all output arcs are connected, and consequently some mechanism for *merging* parallel streams of contexts is required. The informal specification for the Merge node is shown in figure 24,

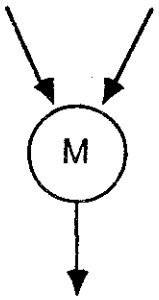

Figure 24: The Merge Node

from which it can be seen that the Merge node has two input arcs and a single output arc. The two preceeding nodes will both output a context on every graph cycle, and therefore the Merge node must be capable of accepting two contexts every graph cycle. As the Merge node can only output a single context per graph cycle it must be capable of buffering the incoming contexts until they can be output. Several buffering disciplines are possible, but the only sensible scheme is to forward incoming contexts on a First-In-First-Out (FIFO) basis. This implies the existence of a linear queue of contexts within each Merge node.

The simple example can now be extended to include a realistic memory interface, by defining appropriate Branch and Merge nodes. This is shown in figure 25.

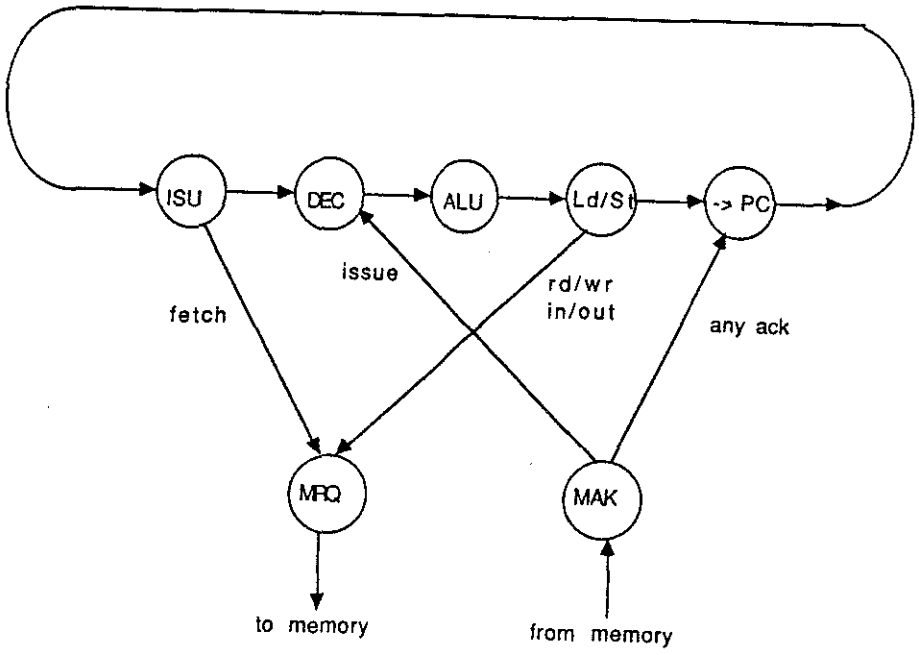

Figure 25: Context Flow Processor with Memory Interface

3.5 Graph Evaluation Rules

The rules for evaluating a Context Flow graph are relatively simple, and are independent of the abstract machine that is being implemented.

- Each "process" has a single unique context.
- A single context flows along every arc in the graph during each system clock period. Contexts may be null.
- Each **T** and **B** node may hold a single context, but **M** nodes may queue contexts for the purpose of matching the average input and output flow rates of non-null contexts.
- Cycles are permitted.
- Any node operations which are not free from side-effect may produce indeterminate results, and are discouraged. Both referentially-transparent and opaque abstract machines can be implemented.
- The **B** node decides on which arc to output its context, based solely on state information within that context.
- The **M** node merges two streams of contexts, with the output order based on a "first-come-first-served" priority.

These rules make no mention of how a context is created, or how a graph is initialised. These are relatively minor issues, and would normally be implementation specific.

4 Context Flow Multiprocessors

Conventional multiprocessor systems, typified by systems such as the Sequent Balance [Fielland and Rogers, 1984], the BB & N Butterfly [Rettberg and Thomas, 1986], the iNTEL iPSC [Intel] and many others, all suffer to varying degrees from performance problems associated with interprocessor communication. These problems limit such machines to tasks involving relatively coarse-grained parallelism, and are caused by the fixed computational overhead imposed on the system by each communication event. The effect of this can be seen from the graphs in figures 26 and 27. The curve in figure 26 shows how the actual performance of a multiprocessor system increases as parallelism is introduced, when the total amount of work is constant and the amount of communication per process is constant. Figure 27 shows the catastrophic performance curve for a multiprocessor system in which the total amount of work is constant and the amount of communication per process is proportional to the number of cooperating processes.

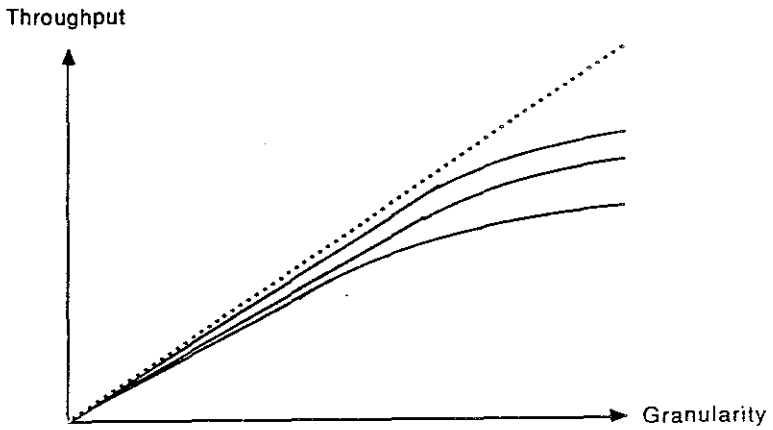

Figure 26: Processor throughput vs. Granularity : non-global communications

If the physical implementation of *any* communication protocol, in a multiprocessor system, requires the exchange of information with a group of processes then an optimum degree of parallelism will exist — beyond which performance will decrease. If the communication protocol requires a non-zero quantity of CPU time, on the sending and receiving processors, for the purpose of switching context (in order to multiprogram the waiting processor) then the system performance will not scale linearly for a fixed problem size.

If it were possible to devise a communication mechanism in which the communication between processes consumed no more time than communication between processors and memories, then the communication overhead could be said to be zero. This is one of the primary goals of Context Flow Architecture, and one which can apparently be attained relatively easily.

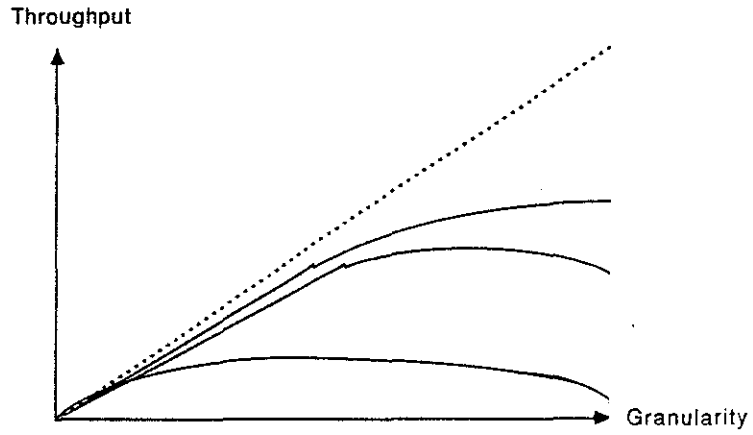

Figure 27: Processor Througput vs. Granularity : global communications

4.1 Context Flow Communication

There are two possible ways in which process communication links can be defined; via a communication channel or by explicitly naming the source and destination processes. Each has its advantages and disadvantages. For example, explicit process naming removes the need for declaring channels but requires the passing of parent process names to sub-processes when they are created [Hoare, 1985, page 239]. There is also a potential implementation problem when communicating non-deterministically between groups of explicitly named processes, where the cyclic polling of processes becomes necessary. This can be avoided by using communication channels. We now propose a mechanism for interprocessor communication in a Context Flow environment, capable of supporting non-determinsism efficiently.

For two processes to communicate their contexts must somehow meet. According to the graph-evaluation rules, outlined above, the contexts must be in the same node at the same time. Thus, we define a communication channel as the **queue** of contexts belonging to those processes waiting to communicate through that channel. Such a queue has several useful properties. Firstly, the queue can be stored in ordinary memory — attached to a Transformation node somewhere in the machine. Therefore, to communicate, a context must contain the address of the channel through which it wishes to communicate as it flows through the node containing the queue.

Two simple transformations, **send** (**S**) and **receive** (**R**) can be defined in much the same way that **write** and **read** are defined. Both **S** and **R** operate in similar ways, the only difference between them being in the direction of data transfer. When a context $\langle f, s_d \rangle$ is encountered, and $f = \mathbf{X}(ch, var)$ such that $X \in \{\mathbf{S}, \mathbf{R}\}$ then some communication event is required. If the channel queue is empty then the current context is queued, and the node outputs a null context. If the channel is not empty it will contain one or more contexts of the form $\langle \mathbf{Y}(ch, var), s_d \rangle$. If $X = Y$ then the current context is added to the end of the queue and a null context is output. If $X \neq Y$ then one X event and at least one Y event are in the same node, and a communication event can occur. This is achieved simply by removing the element at the front of the queue, exchanging any information required by the communication protocol and outputing both contexts. The production

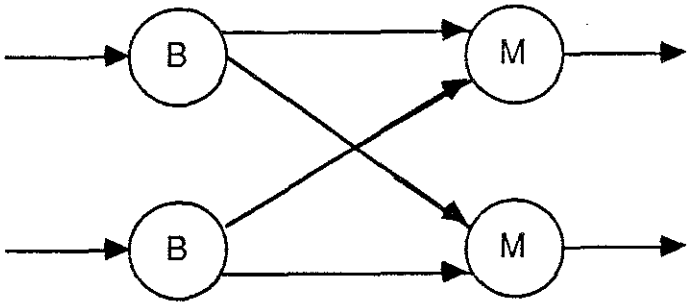

Figure 28: A 2×2 Router for CF Multiprocessors

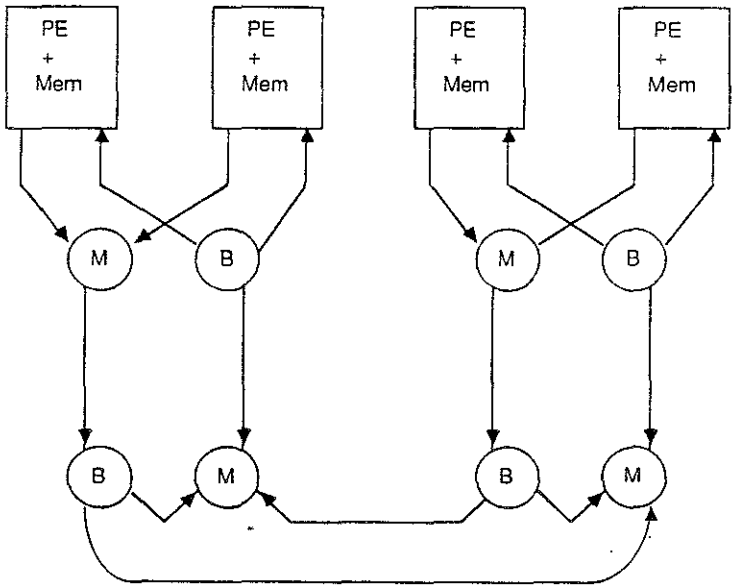

Figure 29: A 4-Processor Configuration

can say that the abstract machine is relatively efficient. The actual efficiency of any abstract machine depends upon the power of the abstract machine instruction set, and so to produce an efficient computing engine we must consider the implications of high-level languages, abstract machines and hardware implementation efficiency as one problem.

5 Summary

High performance computers require special implementation techniques, and the most widely used of these techniques is pipelining. Pipelining promises performance gains in two respects. Firstly,

it promises increased instruction processing throughput; secondly, it promises a more effective use of the hardware resources available. A wealth of experience exists in the design and evaluation of pipelined machines, and for the most part the results are particularly disappointing. Consider a pipeline with a maximum parallelism of n. Now in most machines the value of n will be somewhere between five and twenty, so the extent of parallelism is hardly massive. If the limited extent of parallelism is further reduced by a poor utilization factor, caused by the program discontinuities discussed earlier in this report, then the positive advantages of pipelining could be marginal.

Early machines, such as the CDC 6600 and MU5, incorporated complex control mechanims in an attempt to maximize the pipeline throughput. Studies of MU5 revealed that average instruction times were between 0.2 and 0.4μS, compared with a pipeline beat time of 50 nS. Whilst the mechanisms designed to maximize throughput are interesting, their cost-effectiveness must be questioned.

This report has surveyed a number of proposals for radically different styles of pipeline. Most notable amongst these are the HEP, the CPC and the CCMP. These machines all use a technique known as μ-multiprogramming which, in effect, implements a trade-off between high-latency operations and multiprogramming.

The disparate theoretical bases for these architectures are brought together in the form of a graphical implementation technique called Context Flow. We have shown how a pipelined processor can be contructed using this technique, and how it can then be extended to a multiprocessor system. One of the primary aims of Context Flow architecture is to express both the logical and the spatial manipulations required to produce a solution, in a form which explicitly identifies the logical and spatial complexity of the problem.

A mechanism for synchronously transfering information between the contexts of two processes has been outlined. The communication protocol can support guarded input or output constructs for non-deterministic processing, and has a constant time-complexity. In addition, the communication protocol is implemented as two special memory functions, rather than as a sequence of processing element instructions, and consequently consumes zero CPU time.

It is expected that this technique will become more widely exploited, as the use of VLSI in high-performance systems increases, for it is only through the use of integrated processing elements that this technique can achieve its full potential.

The authors' active work in this area can be divided into three areas. Firstly, we are investigating the possibilities for the design of VLSI microprocessors which are revealed by this technique. Secondly, we are investigating the implementation of an abstract machine for functional languages using Context Flow ideas. Thirdly, the referential transparency of Context Flow building blocks, and the relative ease with which they can describe real systems, leads us to believe that Context Flow could represent a very useful tool for describing parallel hardware at a high level. One possible use for this could be as a description language for a Silicon Compiler. It might then be possible to translate from a formal description of an abstract machine, in the form of an annotated Context Flow graph, to a correct implementation in silicon. The graphical description would also contain enough information to drive a behavioural simulation, firstly to verify the specification and secondly to gauge the efficiency of parallel processing within the system as a whole.

ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under grant number DCR-8603609.

References

[Arvind, 1983] Arvind and R A Iannuci. A Critique of Multiprocessing von Neumann Style. In *Proceedings, 10th Annual International Symposium on Computer Architecture*, pages 426–436, 1983.

[CDC, 1977] *Control Data 7600/ Cyber 70 Model 76 Computer Systems: Hardware Reference Manual.* Control Data Corporation, 1977.

[CDC, 1981] *Control Data Cyber 200 Model 205: Hardware Reference Manual.* Control Data Corporation, 1981.

[Chen, 1971] T C Chen. Parallelism, Pipelining, and Computer Efficiency. *Computer Design*, January 1971, pp 69–74.

[Edwards et al, 1980] D B G Edwards A E Knowles and J V Woods. MU6-G: A New Design to Achieve Mainframe Performance from a Mini-Sized Computer. In *Proceedings, 7th Annual International Symposium on Computer Architecture*, pages 161–167, 1980.

[Feng, 1981] T Y Feng. A Survey of Interconnection Networks. *IEEE Computer*, 12–27, Dec. 1981.

[Flynn, 1970] M J Flynn. A Multiple Instruction Stream Computer with Shared Resources. In L C Hobbs et al, editor, *Parallel Processor Systems, Technologies, and Applications*, pages 251–286, Spartan Books, Washington, D.C., 1970.

[Fielland and Rogers, 1984] G Fielland and D Rogers. 32-bit computer systems shares load equally among up to 12 processors. *Electronic Design*, September 1984.

[Goodman, 1985] J R Goodman et al. PIPE: a VLSI decoupled architecture. In *Proceedings, 12th Annual International Symposium on Computer Architecture*, pages 20–27, 1985.

[Goto et al, 1986] E Goto K Shimizu and S Ichikawa. *CPC (Cyclic Pipelined Computer) - An Architecture Suited for Josephson and Pipelined Machines.* Technical Report 86-19, University of Tokyo, Department of Information Science, November 1986.

[Hwang and Briggs, 1984] K Hwang and F A Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, NY, 1984.

[Hennessy, 1984] J L Hennessy. VLSI Processor Architecture. *IEEE Transactions on Computers*, C-29(4):1221–1246, 1984.

[Holgate and Ibbett, 1980] R W Holgate and R N Ibbett. An Analysis of Instruction-Fetching Strategies in Pipelined Computers. *IEEE Transactions on Computers*, C-29(4):325–329, 1980.

[Hockney and Jesshope, 1981] R W Hockney and C R Jesshope. *Parallel Computers.* Adam Hilger Ltd., Bristol, U.K., 1981.

[Hoare, 1985] C A R Hoare. *Communicating Sequential Processes.* Prentice-Hall International, U.K., 1985.

[Hintz and Tate, 1972] R G Hintz and D P Tate. Control Data STAR-100 Processor Design. In *FALL COMPCON*, pages 1–4, 1972.

[Ibbett, 1981] R N Ibbett. Vector Processing. In *Proceedings, International Computing Symposium*, pages 337–341, 1981.

[Ibbett, 1982] R N Ibbett. *The Architecture of High Performance Computers.* Springer-Verlag, New York, NY, 1982.

[Ibbett et al, 1985] R N Ibbett P C Capon and N P Topham. MU6-V: A Parallel Vector Processing System. In *12th Annual International Symposium on Computer Architecture*, pages 136–144, 1985.

[Intel] *iPSC Product Summary.* intel, Pubn. no. 280101-002.

[Keller and Sleep, 1981] R M Keller and M R Sleep. Applicative Caching. In *ACM Conference on Functional Programming Languages and Computer Architecture*, pages 131–140, 1981.

[Kunkel and Smith, 1986] S R Kunkel and A J Smith. Optimal Pipelining in Supercomputers. In *Proceedings, 13th Annual International Symposium on Computer Architecture*, pages 404–411, 1986.

[Lee and Smith, 1984] J F K Lee and A J Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1):6–22, 1984.

[Lincoln, 1982] N R Lincoln. Technology and design tradeoffs in the creation of a modern supercomputer. *IEEE Transactions on Computers*, C-31(5):349–362, 1982.

[Lin, 1986] Q Lin. Design of a Vector Processor. *Journal of Computer Science and Technology*, 1(1):26–34, 1986.

[Mead and Conway, 1980] C Mead and L Conway. *Introduction to VLSI Systems.* Addison Wesley, 1980.

[McFarling and Hennessy, 1986] S McFarling and J Hennessy. Reducing the cost of branches. In *Proceedings, 13th Annual International Symposium on Computer Architecture*, pages 396–403, 1986.

[Morris and Ibbett, 1979] D Morris and R N Ibbett. *The MU5 Computer System.* Springer-Verlag, New York, NY, 1979.

[Miura, 1986] K Miura. Fujitsu's Supercomputer: Facom Vector Processor System. In S Fernbach, editor, *Supercomputers: Class VI Systems, Hardware and Software*, pages 137–152, North-Holland, Amsterdam, The Netherlands, 1986.

[Moon, 1985] D A Moon. Architecture of the Symbolics 3600. In *Proceedings, 12th Annual International Symposium on Computer Architecture*, pages 76–83, 1985.

[Burstall et al, 1980] R M Burstall D B MacQueen and D T Sanella. HOPE: an experimental applicative language. In *LISP Conference Record*, pages 187–194, 1980.

[Murphy and Wade, 1970] J O Murphy and R M Wade. The IBM 360/195. *Datamation*, 72–79, April 1970.

[Myers, 1982] G J Myers. *Advances in Computer Architecture.* John Wiley and Sons, New York, NY, 1982.

[Odaka et al, 1986] T Odaka S Nagashima and S Kawabe. Hitachi Supercomputer S-810 Array Processor System. In S Fernbach, editor, *Supercomputers: Class VI Systems, Hardware and Software*, pages 113–136, North-Holland, Amsterdam, The Netherlands, 1986.

[Omondi and Brock, 1987] A R Omondi and J D Brock. *Micromultiprogramming a Vector Pipeline.* Techical Report (In Preparation), Department of Computer Science, University of North Carolina - Chapel Hill, 1987.

[Patterson, 1983] D A Patterson et al. Architecture of a VLSI instruction cache for a RISC. In *Proceedings, 10th International Symposium on Computer Architecture*, pages 108–118, 1983.

[Pier, 1983] K A Pier. A retrospective on the dorado : a high performance personal computer. In *Proceedings, 10th International Symposium on Computer Architecture*, pages 252–269, 1983.

[Radin, 1983] G Radin. The IBM 801 Minicomputer. *IBM Journal of Research and Development*, 27(3):237–246, 1983.

[Rammamoorthy and Li, 1977] C V Rammamoorthy and H F Li. Pipeline Architecture. *ACM Computing Surveys*, 9(1):61–102, March 1977.

[Rettberg and Thomas, 1986] R Rettberg and R Thomas. Contention is no obstacle to shared-memory multiprocessing. *Comm. ACM*, 29:1202–1212, Dec. 1986.

[Sequin, 1983] C A Sequin. Design and Implementation of RISC I. In *VLSI Architecture*, pages 276–298, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[Siegel, 1979a] H J Siegel. A Model of SIMD Machines and a Comparison of Various Interconnection Networks. *IEEE Trans. Comput.*, C-28(12):907–917, Dec. 1979.

[Siegel, 1979b] H J Siegel. Interconnection Networks for SIMD Machines. *IEEE Computer*, 12(6):57–65, 1979.

[Smith, 1985] B Smith. The Architecture of HEP. In *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pages 41–55, MIT Press, Cambridge, MA, 1985.

[Stanley and Butner, 1986] C A Stanley and S F Butner. A feasibility study and simulation of the circulating context multiprocessor. In *Proc. IEEE Conf. Parallel Processing*, pages 455–462, 1986.

[Thacker et el, 1982] C P Thacker et al. Alto: A Personal Computer. In D P Siewiorek C G Bell and A Newell, editors, *Computer Structures: Principles and Examples*, pages 549–572, McGraw-Hill, New York, NY, 1982.

[Thornton, 1970] J E Thornton. *Design of a Computer: The Control Data 6600.* Scott, Foresman and Co, Glenview, IL, 1970.

[Thompson, 1986] J R Thompson. The CRAY-1, the CRAY-XMP, the CRAY-2 and Beyond: the supercomputers of Cray Research. In S Fernbach, editor, *Supercomputers: Class VI Systems, Hardware and Software*, pages 169–82, North-Holland, Amsterdam, The Netherlands, 1986.

[Tomsulo, 1967] R M Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[Topham, 1986] N P Topham. *A Parallel Machine Description.* Internal Report, Department of Computer Science, University of Edinburgh, 1986.

[Topham, 1987] N P Topham. Performance Analysis of a Data-Driven Multiple Vector Processing System. In G L Reijns and M H Barton, editors, *Highly Parallel Computers*, pages 111–125, North-Holland, Amsterdam, The Netherlands, 1987.

[Tucker, 1986] S G Tucker. The IBM 3090: an overview. *IBM Systems Journal*, 25(1):4–19, 1986.

[Watanabe et al, 1986] T Watanabe H Katayama and A Iwaya. Introduction of the NEC Supercomputer SX System. In S Fernbach, editor, *Supercomputers: Class VI Systems, Hardware and Software*, pages 153–168, North-Holland, Amsterdam, The Netherlands, 1986.