Pipelines

A Comparison of Two Common Pipelines Structures

Michael Golden and Trevor Mudge

Electrical Engineering and Computer Science Department

University of Michigan
1301 Beal Avenue
Ann Arbor, Michigan, 48109-2122
email: {mgolden,tnm}@eecs.umich.edu

Abstract

We examine two pipeline structures that are employed in commercial microprocessors. The first is the load-use interlock (LUI) pipeline, which employs an interlock to ensure correct operation during load-use hazards. The second is the address-generation interlock (AGI) pipeline. It eliminates the load-use hazard, but has an address-generation hazard, which requires an address-generation interlock for correct operation. We compare the performance of these two pipelines on existing binaries and on applications that have been recompiled with a local code scheduler that understands the difference in the pipeline structures. Under the assumption of perfect branch prediction, the AGI pipeline outperforms the LUI pipeline on the SPEC92 integer benchmarks, even on binaries that have been compiled for the LUI pipe. When branch prediction is considered, the AGI pipeline performs significantly better than the LUI pipeline if branch prediction is more than 80% accurate and the data cache access time is greater than two cycles. Recompiling the benchmarks with a new local code scheduler optimized for the AGI pipeline provides little additional performance improvement.

Keywords: *cache memory, interlocks, memory system, pipelines, RISC*

1 Introduction

Although pipelining is a widely used technique for speeding up instruction execution, the existence of dependences between instructions means that pipelines cannot run at 100% efficiency. Nevertheless, the improvement in speed through pipelining usually offsets any loss in performance[17].

This paper will examine three types of "hazards" that can reduce the efficiency of a pipeline: branch, load, and address-generation hazards. In particular we will compare two pipeline organizations employed in several commercial machines that make different trade-offs between these three hazards. The first, which we shall refer to as the load-use interlock (LUI) pipeline, issues and

This work was supported by Defense Advanced Research Projects Agency under DARPA/ARO Contract Number DAAL03-90-C-0028.

completes its instructions in-order. It is subject to branch hazards and load hazards, but not address-generation hazards. The second, which we shall refer to as the address-generation interlock (AGI) pipeline, also issues and completes its instructions in-order but differs from the LUI pipeline in that the execute stage is placed later in the pipeline to avoid load hazards. However, this difference results in address-generation hazards and increases the penalty for branch hazards. In this paper we will report on experiments to determine if these penalties are outweighed by the benefits of eliminating load hazards.

The MIPS R2000 and R3000 use a precursor to the LUI pipeline. This precursor does not employ hardware interlocks for loads or branches. Instead, NOPs are inserted after loads and branches, as required, to ensure correct operation. Load interlocks were added in the R6000, a short-lived ECL implementation of the MIPS instruction-set architecture (ISA) [16]. Load interlocks were also subsequently employed in the R4000, R4200, and R4400 [12]. The AGI pipeline is used in the Intel i486 and Pentium and the Cyrix M1[1], as well as in the R8000[2][3][6] [7]. The R8000, which was originally referred to in the literature as the TFP, also implements the MIPS ISA [5] [8]. All four processors with AGI pipelines are designed to preserve *binary compatibility* with earlier LUI microprocessors. A large body of software exists in the form of binaries optimized for the LUI pipeline structure, and it is not known how much performance is degraded when these binaries are run on the rearranged pipeline. To be acceptable, any reduction must be small to avoid the cost of recompiling applications.

There are two questions that this paper attempts to answer:

1. How does the AGI pipeline affect performance on binaries created for an LUI pipeline?
2. Does the AGI pipeline improve performance if the compiler performs local code scheduling specifically for this organization?

This paper is organized as follows. The next section discusses pipeline hazards in more detail and previous work on methods to reducing their negative affect on performance. With this as background, Section 3 describes the LUI and AGI pipeline organizations. The compiler and simulation tools are described in Section 4. Experimental results are presented in Section 5 followed by some concluding remarks in Section 6.

1.The M1 executes the Intel instruction set, but has one extra address calculation stage than the other pipelines.

2 Pipeline hazards and previous work

2.1 Branch hazards

We define the *scope* of a branch to be the number of instructions that can be issued before the branch is resolved. A *branch hazard* occurs when an instruction in the scope of the branch depends on the outcome of the branch. Although a processor may stall the issue of new instructions until it resolves a branch instruction, the introduction of pipeline bubbles caused by this approach can reduce performance to an unacceptable level.

Branch hazards can be eliminated statically by having the compiler schedule independent instructions in the scope of a branch. Alternatively, the pipeline may dynamically eliminate branch hazards by predicting the outcome of the branch, allowing control-dependent instructions to enter the pipeline, and squashing them if the branch has been mispredicted. These approaches are not mutually exclusive, and it is not unusual for some combination to be employed.

Both approaches to removing branch hazards have shortcomings. It is not always possible to eliminate branch hazards by reordering code. It may be necessary to insert NOPs so that any instructions that cause branch hazards are moved beyond the scope of the branch. As noted, this is the solution taken by the R2/3000. However, the presence of NOPs in the execution stream reduces efficiency. Branch prediction can also introduce inefficiency when a prediction fails and instructions that execute as a result of mispredictions must be squashed.

2.2 Load hazards

Load hazards are a result of data dependences rather than control dependences. They occur when the instructions immediately following a load depend on the value retrieved by the load instruction. We define the scope of a load to be the number of instructions that can be issued before the data retrieved from memory by the load becomes available to later instructions. Because the amount of time required to access a value that may reside in any level of a memory hierarchy may vary, the scope of a load instruction may also vary. A load hazard occurs when an instruction in the scope of a load uses directly or indirectly the value read by the load.

In a pipeline that supports out-of-order execution, an instruction that depends on an outstanding load operation can simply be buffered at a reservation station until all of its operands are avail-

able and it can be sent to a function unit. In a pipeline that only allows in-order execution of instructions, there are three approaches to tolerating a load hazard: 1) reorder instructions so that there are no instructions that cause load hazards after the load; 2) stall the pipeline when an instruction that causes a load hazard is fetched until the load is completed (load-use interlock); and 3) use some form of load prediction to prefetch load data and effectively remove dependences that arise from the load.

All three approaches to removing load hazards have shortcomings. It is not always possible to eliminate load hazards by reordering code. It may be necessary to insert NOPs so that dependent instructions that cause hazards are moved beyond the scope of the load. Such a processor must still have some interlock to handle the case when the load instruction misses in the first level of memory and extra cycles are required to fetch the missing data. Once again, this is the solution taken by the R2/3000 and again, the presence of NOPs in the execution stream reduces efficiency. The use of load-use-interlock stalls avoids the code expansion of NOPs, but it too reduces efficiency. Finally, loads are much more difficult to predict and the last method is rarely used [4]. Again these approaches are not mutually exclusive.

2.3 Address-generation hazards

Address-generation hazards occur when a value is computed for a register that is used to form the address of the data retrieved by a load instruction. For the purposes of this discussion we consider only the base-register-plus-offset address mode for load instructions. In this case, the scope of address generation is the number of instruction slots between an instruction that modifies a register and its earliest availability for use as a base register in an address calculation.

As in the case of any data hazard, a machine that supports out-of-order execution of instructions can simply buffer the dependent instruction until all operands become available. For a pipeline that does not allow this model of execution, there are two approaches to tolerating an address-generation hazard: 1) insert instructions so that there is sufficient time to finish modifying the address register before its use by the load instruction; and 2) stall the pipeline until address generation is completed (address-generation interlock). In principle, address generation could also be predicted but it is never done. Removing address-generation hazards by stalling is, as with the other hazards, a source of inefficiency.

2.4 Other hazards

In addition to the hazards that we are concerned with in this paper, there are others that have only a small impact on the performance of LUI or AGI pipelines, or that are avoided altogether in the LUI and AGI pipelines. In the first category are instructions that store values to memory. During a store operation, the memory system does not return a value to the CPU, so subsequent instructions can usually be issued without delay. A hazard can occur if, before the store completes, a load instruction is issued that retrieves data from the memory location that is the target of the store. Microarchitectural features such as write buffers or write caches with hazard detection logic have been used to solve this problem [11]. In this paper, the effects of store hazards are ignored.

In the second category are hazards resulting from true data dependences on instructions that perform ALU operations: if the results of an instruction are required by a succeeding instruction, and if the second instruction issues before the first instruction computes its result, then a hazard occurs. The LUI and AGI pipelines avoid this class of hazards by implementing ALU operations that only require one cycle and by employing *bypass* paths that send a value from one pipeline stage directly to another stage.

Within the scope of this paper, we shall not be concerned with machines that issue more than one instruction at a time, typified by superscalar or VLIW architectures. Of course, their individual execution pipes are likely to be of the LUI or AGI type, and future studies might investigate their relative merits in this setting where the matter of instruction dependence becomes much more complex. For two excellent discussions on this, the reader is referred to [10] and [18].

2.5 Previous work

Previous work has proposed both static and dynamic techniques of eliminating the hazards caused by instructions that are dependent on load instructions. Static techniques involve code scheduling in which the compiler attempts to hide the latency of load instructions by scheduling them well before their results are needed. In [14], Krishnamurthy presents a survey of techniques for local code scheduling. Global code scheduling techniques, such as superblock[9] and hyperblock[15] scheduling, allow code motion between basic blocks.

Austin, et al. classify load instructions into three categories [1]:

1. *Global pointer* references to global variables,
2. *Stack pointer* references to variables on the stack frame, and
3. *General pointer* references which cover all other loads, including references to pointers and arrays.

They notice that the offsets for global and stack pointer references are often quite large. The offsets are small—the most common offset is zero—for general pointer offsets. Negative offsets arise from negative array references and are most unusual. Because of loop optimizations such as strength reduction, most loads that perform array references within loops have zero offsets.

Because many offsets have small values, a logical OR operation will have the same effect as the full address calculation in most cases. Of course, this method fails for negative offsets and whenever there is a carry in any part of the addition. Their method rapidly predicts the set index into the first level cache by OR-ing together the appropriate bits in the base register and the offset instead of performing the full addition. They propose circuitry that will do this during the instruction fetch stage of the "classic five stage pipeline," (see below) so that during the ALU operation stage, the processor can access the cache while it computes the full effective address. The processor saves a cycle if the prediction is correct. If the prediction fails, the load must be reissued.

Because this only works if there are no carries between the bits of the effective address calculation, they propose that the compiler and linker align the global pointer, objects on the heap, and the stack frames to large powers of two. This eliminates carries in many address calculations. These software and hardware optimizations increase data memory allocation up to 20% and memory system usage by up to 50% for some benchmark programs.

Iliffe describes a "forward looking" architecture that immediately issues a memory load whenever the processor forms a potential address instead of waiting for an actual load instruction to be encountered in the instruction stream [13]. In Iliffe's proposal, all registers are tagged. A potential address is created through normal machine instructions that have a destination tagged as an address register. As soon as the processor writes a value to an address register, the machine issues a load to that address.

Sohi and Davidson describe the Structured Memory Access architecture, or SMA[22]. This

machine has an address processing unit that can accept a pattern in memory and issue loads to all addresses in the pattern before the values are actually used. This feature works well to exploit the natural regularity of memory accesses to structures like vectors and multidimensional arrays.

Golden and Mudge propose a microarchitectural cache called a load target buffer (LTB) which is indexed with the program counter during the instruction fetch stage of a pipeline[4]. If the LTB indicates that the current instruction is a load, the processor immediately issues a request to the memory system using a prediction of the required address. The LTB makes this prediction using a history of memory locations accessed by a given load instruction, and can successfully predict the targets of load instructions which have constant-stride reference patterns.

3 Two pipeline organizations

3.1 The load-use interlock pipeline

The LUI pipeline is shown in Figure 1. This has been referred to as the "classic five stage RISC pipeline" [20]. Each box represents a single machine cycle and a list of the functions that are performed during that cycle. Figure 1 labels the five stages with their primary function:

- IF — instruction fetch
- RD — register read and decode
- EX — execute the ALU operation
- MEM — data cache access
- WB — write back to the register file

The bypass paths are also shown. The number of cycles spanned by the path indicates how long the bypass operation takes.

Figure 1 shows that conditional branches are not resolved until the end of the first half of the EX stage. This results in a branch scope of one cycle, during which a branch hazard can occur. This is solved by the inclusion of a *branch-delay slot* in the MIPS ISA. Correct operation requires that the instruction in the branch-delay slot must be able to execute independently of the result of the branch. If an independent instruction cannot be found, a NOP is inserted into the branch-delay slot.

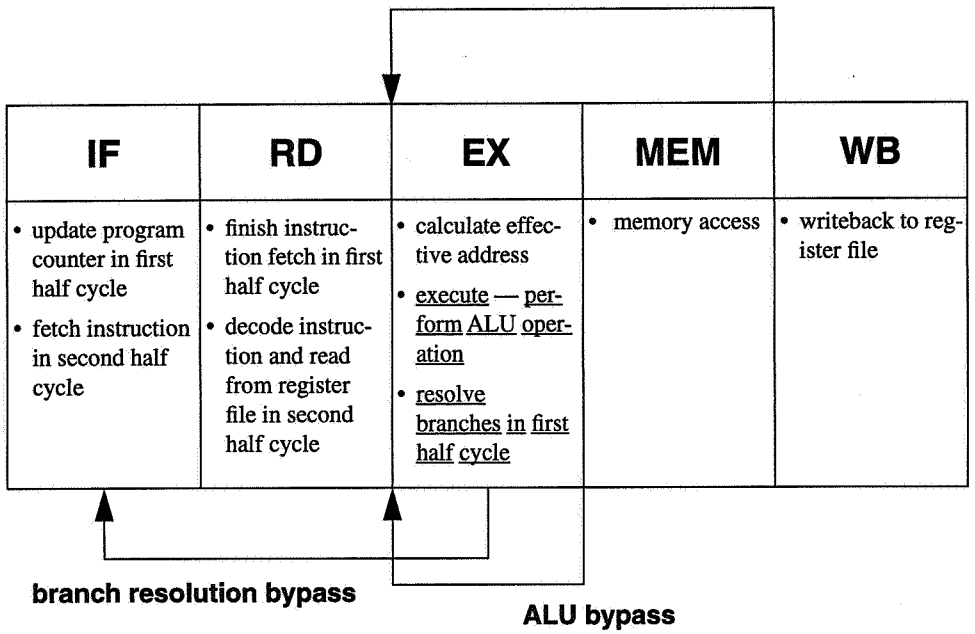

Figure 1: The LUI Pipeline

The five stages and bypass paths are shown. The actions in the EX stage that are underlined are moved into the MEM stage in the AGI pipeline. See Figure 3.

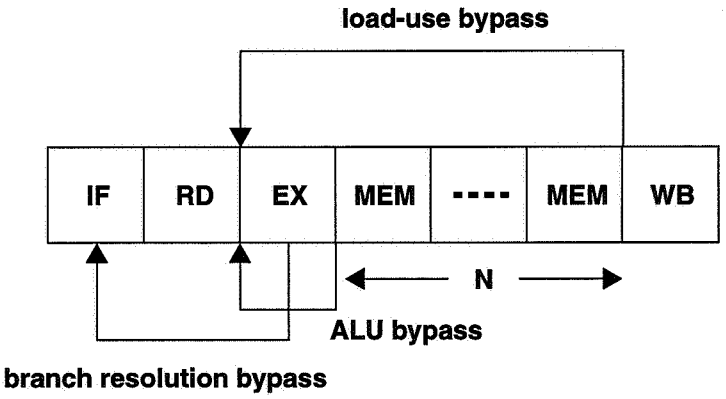

Figure 2: The LUI-N Pipeline

The extra memory cycles and the corresponding increase in the load-use bypass are shown.

During a load instruction, the effective memory address is computed during the EX stage and sent to the memory system. If the request hits in the first level cache, the result is available at the end of the MEM stage, where it may be forwarded back to the EX stage. The forwarding path spans two cycles, indicating that the MEM stage result is not available to the instruction that immediately

follows it in the pipeline, but to the second instruction after the load. Any instruction immediately after a load that uses the result of that load creates a load hazard. In such cases, the pipeline stalls for one cycle. Of course, if the instruction misses in the cache, the delay is much greater and the pipeline stalls for many cycles.

In the early MIPS machines (R2000 and R3000), as noted earlier, the absence of a load-use interlock is handled by requiring that the compiler guarantee that the instruction after a load is not dependent on the load. This instruction occupies the *load-delay slot.* If the compiler cannot find an independent instruction, it puts a NOP instruction in the load-delay slot [12].

In high clock rate microprocessors, even the on-chip primary cache can take more than one cycle to access. This paper will also consider a generalization of the LUI pipeline to systems with multiple-cycle data cache access times. These pipelines will contain additional MEM stages. A data cache with an access time of N cycles will be paired with a LUI pipeline with N MEM stages, and will be referred to as an LUI-N pipeline (see Figure 2). In an LUI-N pipeline, the scope of a load is N instructions and its load-use interlocks can last from 1 to N cycles. If the first dependent instruction in the load scope is k instructions after the load, then the interlock will stall the pipeline for (N-k)+1 cycles.

3.2 The address-generation interlock pipeline

The AGI pipeline is shown in Figure 3. In this pipeline, the load-use interlock has been eliminated by delaying the EX stage by one cycle and combining it with the MEM stage. Combining the EX and MEM stages requires an extra adder, which is dedicated to computing the target address of memory operations. This address calculation is performed in the AD stage before the EX/MEM stage. In contrast, the LUI pipeline has only a single adder in the EX stage, which is used for both integer arithmetic instructions and address calculations. In the AGI pipeline, when an instruction that is dependent upon a load in the previous cycle reaches the EX/MEM stage, the results of the load are available from the ALU bypass. However, branch resolution now occurs one stage later because a conditional branch instruction may require a result from the instruction that immediately precedes it. This result will not be available until the end of the EX stage.

There are two disadvantages to this arrangement. First, an address-generation interlock is re-

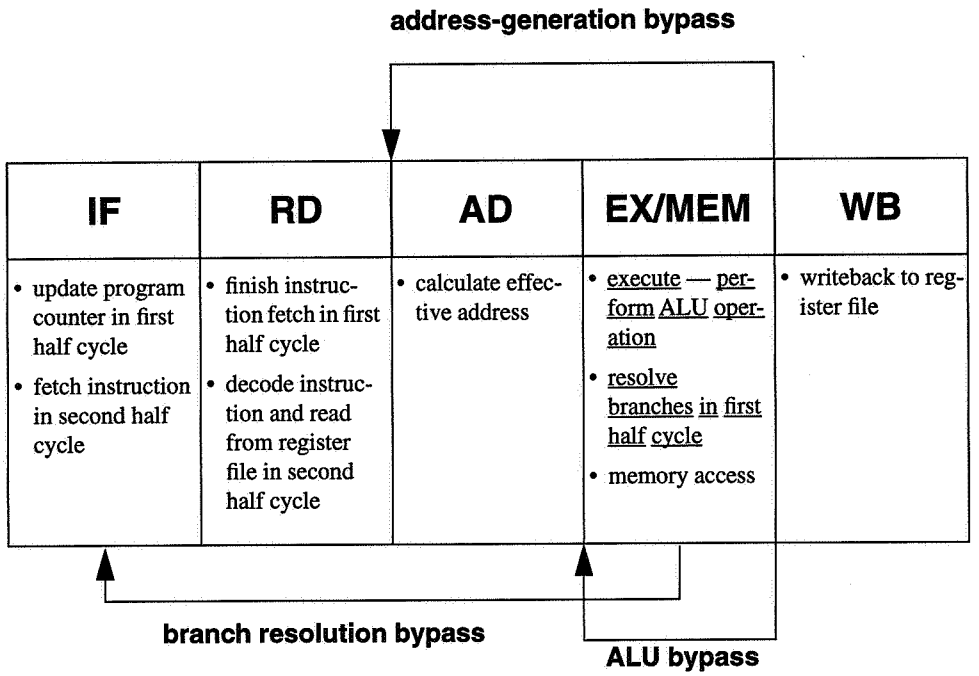

Figure 3: The AGI Pipeline

The five stages and bypass paths are shown. The actions in the EX/MEM stage that are underlined are moved from the EX stage in the LUI pipeline. See Figure 1.

quired when a load instruction requires the register result of an uncompleted instruction to calculate the target address in memory. Second, the branch scope is now two cycles because branch resolution occurs in the first half of the EX/MEM stage of the pipeline. This means that in addition to the branch-delay slot, a second instruction will issue before the branch is resolved. We assume that this instruction is chosen by a prediction scheme, and that it may have to be squashed if the branch has been mispredicted. This contrasts with the LUI pipeline which, because of the branch-delay slot, needs no branch prediction strategy.

As cache access time grows beyond a single cycle, delay stages can be added to the AGI pipeline between the AD and EX/MEM stages. A processor that takes N cycles to access the cache will require N-1 extra MEM stages. We refer to this as an AGI-N pipeline, as shown in Figure 4. In an AGI-N pipe, N instructions must be squashed every time a branch is mispredicted, and address-generation interlocks can last from 1 through N cycles. If the first dependent load instruction is issued k cycles after the instruction that generates its base register, then the interlock will stall the pipeline for (N-k)+1 cycles.

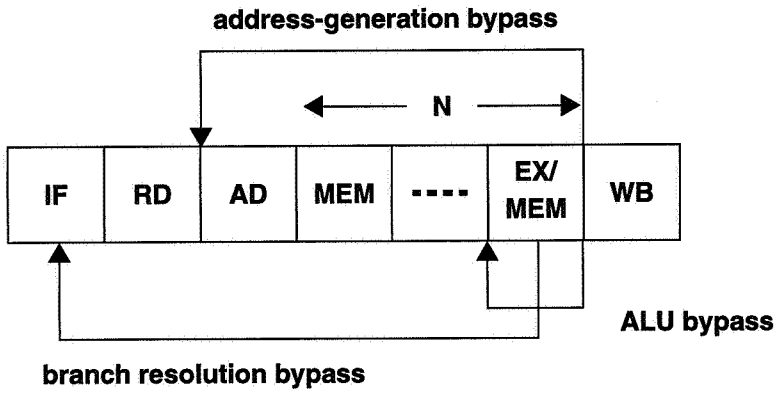

Figure 4: The AGI-N pipeline showing the bypass paths

The extra memory cycles and the corresponding increase in the address-generation bypass are shown.

The code fragment written in MIPS assembly language shown in Figure 5 further illustrates the difference between the two pipeline organizations. NOPs in load-delay slots have been removed—load-use interlocks are modeled instead. The code is taken from the program `eqntott`, a SPEC92 integer benchmark. In this example, instruction I3 depends on instruction I2, which in turn depends on instruction I1. Because the branch instruction I3 depends on I2, a load-use interlock will occur in an LUI pipeline. This interlock does not occur in the AGI pipeline. Instead, an address-generation interlock will stall the pipeline since I1 calculates a value for the base register of the load instruction I2. In addition to the address-generation interlock, the AGI pipeline may face an additional possible performance loss if the branch is mispredicted. In the case of the LUI pipeline, the NOP in the branch-delay slot covers the branch penalty. For every memory access stage in the AGI pipeline, an additional instruction must be squashed after a mispredicted branch. For example, in an AGI-2 pipeline, both I4 and I5 would be squashed if the branch instruction I3 were incorrectly predicted not-taken. Note that for both the LUI and the AGI pipeline, the instruction after the branch occupies a branch-delay slot. Only the additional instructions in the branch scope for the AGI pipeline are speculatively executed.

```
I1:     move    a3,a0                   # move the value in register a0 into register a3
I2:     lw      v1,4(a3)                # use it as the base register to load register v1
I3:     beq     v1,zero,0x400328        # conditionally branch on v1 == 0
        nop
I4:     move    a0,v1                   # v1 != 0, so put the value in v1 into a0
I5:     jal     copy_bnode              # and call copy_bnode(a0)
```

Figure 5: A MIPS assembly language fragment

This code fragment illustrates a load-use hazard and an address-generation hazard.

4 The compiler and simulator

This paper considers programs compiled for the MIPS I instruction set architecture—the version of the architecture that does not support load-use interlocks. This architecture was chosen for several reasons:

- The MIPS architecture has been implemented with a LUI pipeline and with an AGI pipeline. The R series machines all have LUI pipelines and the TFP has an AGI pipeline.
- The Gnu C Compiler (GCC) is available for the MIPS architecture [23]. GCC is in the public domain and the source codes are easily available, so the compiler may be modified.
- The MIPS is a load/store architecture, so all memory operations are contained in explicit load and store instructions. This simplifies the creation of compilers that optimize for the two different pipeline structures.

The experiments use the SPEC 92 integer benchmarks, summarized in Table 1. All of the benchmark programs are executed to completion using one of the "reference" input files provided by SPEC except `xlisp`, which uses the "short" input file. The benchmarks are compiled three times. The MIPS C Compiler creates one version of each program. The MIPS C Compiler heavily optimizes the code and assumes a single load-delay slot. In effect, this provides a binary that is optimized for load instructions that have a scope of one cycle on a cache hit. GCC is used to create two versions of each benchmark: one optimized for the AGI pipeline and one optimized for the LUI pipeline. The versions differ in the cost function given to GCC's scheduling algorithm.

GCC's scheduler assigns a priority to each instruction in a basic block. Instructions with high priorities are scheduled first. Several factors determine the priority of an instruction, but the most

Benchmark Name	Input File	Base Execution Time in Cycles	Average Basic Block Size
compress	reference	79 192 765	5.1
eqntott	reference	1 381 970 038	3.0
espresso	bca.in	493 384 704	5.6
gcc	stmt.i	133 778 490	5.0
sc	loada1	436 172 261	4.6
xlisp	short	1 171 528 797	3.0

Table 1: The SPEC92 integer benchmarks and their characteristics

The reference input files provided by SPEC are used for all of the benchmarks except xlisp, which uses the SPEC-provided short input file due to simulation time considerations. When several SPEC reference input files are available, the experiments use the file listed in the table. The base execution time is the time required to execute the benchmark to completion on a processor with a zero-cycle cache access time.

important is the scope of an instruction. An instruction with a large scope that produces results used by a later instruction is assigned a high priority equal to the number of instructions in its scope. Once the instructions are prioritized, GCC attempts to schedule each instruction so that the pipeline will never interlock. For a discussion of scheduling techniques for pipelined processors, see [14].

To provide a binary that optimizes for load-use hazards, one version of each benchmark is produced in which GCC is told that two instructions are required between a load and its use for interlock-free execution. To create a version optimized to reduce address-generation hazards, the scheduler is told that the scope of address generation is two cycles. The study includes the MIPS C Compiler version because it is the standard compiler for systems using the MIPS processors. Comparing the code produced by GCC with the MIPS C Compiler's version provides a confidence check that the code produced by GCC for the AGI pipeline is equally well optimized.

Each version of the program is then instrumented to produce an instruction and data trace by `pixie`. A simulator based on the `xsim` tool developed by Smith consumes the trace [21]. The simulator models a machine with the following characteristics:

- There are no load-delay slots. Other delay slots, mainly those required by the MIPS architecture for integer multiply and divide instructions, are present in the machine model. This

includes a single branch-delay slot for both the AGI-N and the LUI-N pipeline.

- All operations except data cache accesses complete in a single cycle.
- There is a single execution pipeline.
- All memory references hit in the instruction and data caches.
- Instruction fetch requires a single cycle.

Load-delay slots have been eliminated in newer RISC architectures, such as the Alpha, in favor of load-use interlocks. As cache access times get longer, code expansion caused by NOPs in unfilled delay slots becomes a problem [19]. Typical RISC integer instructions complete in a single cycle, except integer multiplication and division, which usually take more than one cycle. The MIPS ISA requires delay slots in the scope of these instructions, which must be filled by independent instructions or NOPs.

5 Experimental results

5.1 Experiments on an ideal pipeline

In the figures in this section, the x-axis shows the access time of the data cache in cycles. The y-axis shows an execution time that is normalized to the run time of code compiled by the MIPS C Compiler for a machine with an LUI pipeline and a zero-cycle cache access time (N = 0). In other words, all memory references are immediately available so there are no load-use hazards or address-generation hazards in the reference machine. The third column of Table 1 lists these base execution times for each benchmark in cycles. The harmonic means of the experimental results for all benchmarks are presented in Figures 6-9. Results for individual benchmarks are shown in Figures 10-15. High numbers indicate poor performance. When the benchmarks `eqntott` and `xlisp` are simulated for large cache access times, their run times overflow the cycle-counting capabilities of the simulator. Because of this, some of the experiments are missing from Figure 12 and Figure 15 for cache access times of six and seven cycles. To make the comparison between pipelines fair, `xlisp` and `eqntott` are removed from the harmonic mean calculations for these two cache access times.

The first experiment compares how the benchmarks perform on code compiled by the MIPS C Compiler for the MIPS R2000 performs on an LUI and an AGI pipeline for varying cache access

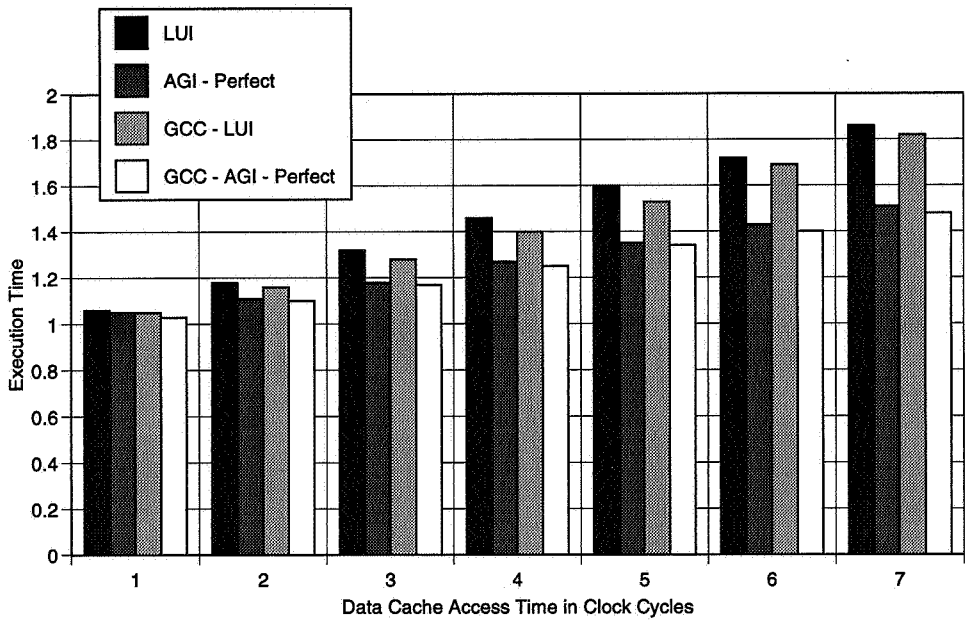

Figure 6: The harmonic mean of all benchmarks — 1-cycle I-Cache

This figure assumes perfect branch prediction. Notice that the improvement between GCC-LUI and GCC-AGI-Perfect is about the same as the improvement between LUI and AGI-Perfect. Informing GCC's local scheduler of the need to avoid address-generation interlocks has little effect. The AGI pipeline shows better performance than the LUI pipeline in all cases.

times. The results assume perfect branch prediction in the AGI case. These bars are labeled "LUI" and "AGI - Perfect" Figure 6. For low cache access times, there is very little difference between the two pipeline organizations. As the access time increases beyond about 3 cycles, the performance benefit of the pipeline with an address-generation interlock begins to appear. The AGI-3 pipeline completes the benchmarks almost 10% faster than the LUI-3 pipeline. The performance gap continues to grow as the cache access time gets larger.

This first experiment answers the question about the performance of existing binaries. For our sample set of benchmarks, the AGI pipeline actually performs slightly better than the LUI pipeline on binaries compiled for an LUI pipeline.

The next set of experiments considers code compiled by GCC for LUI pipelines against code compiled by GCC for AGI pipelines. The programs are run on the pipelines for which they were compiled with the assumption of perfect branch prediction. In Figure 6, these experiments are labeled "GCC-LUI" and "GCC-AGI-Perfect." Once again, a small benefit is seen through the use of AGI pipelines for small cache access times. As cache access times increase, AGI pipelines again provide a larger speedup.

Informing GCC's local scheduler of the new pipeline structure does not seem to affect execution time to a large extent. The percent change between the GCC-LUI experiments and the GCC-AGI experiments are similar to those between the LUI and the AGI-Perfect experiments. This may be because GCC's scheduler works only *within* a single basic block. For the benchmarks under consideration, the basic block size tends to be small, as small as 3 in the case of `xlisp`, so modifying the code scheduling costs may not have a large effect. The limited improvement obtained from the compiler suggests that more aggressive global scheduling techniques may be needed. However, the performance of the Gnu C Compiler versus the MIPS C Compiler—compare LUI vs. GCC-LUI—makes it clear that, for our machine model, GCC is as good as one of the best commercial compilers. This gives support for our remaining results with GCC.

This set of experiments gives a limited answer to the second question posed in the introduction. Simply altering the local scheduling algorithm does not significantly improve the compiler's ability to produce efficient code for the AGI pipeline. However, the performance of the AGI pipeline is already better, as shown above. More sophisticated compiler techniques may provide further improvement.

The final set of results, labeled "GCC-AGI-X%" represent AGI pipelines with X% branch prediction over all branches, including unconditional jumps and calls. These results are summarized in Figure 7. Because the MIPS branch delay slot is included in the simulator, all of the results for LUI pipelines are valid for any branch prediction accuracy. The branch penalty is accounted for by the instruction in the delay slot, which may be a NOP. In contrast, an AGI-N pipeline must squash N extra instructions when a branch is mispredicted. A branch penalty is approximated by assessing a fixed number of cycles for each mispredicted branch and adding it to the total execution time of the benchmark. The penalty for machines with LUI and AGI pipelines are calculated with the following formulas:

$$\text{Penalty}_{LUI} = (N_i - 1) \times (1 - b) \times C_b$$

$$\text{Penalty}_{AGI} = (N_d + N_i - 1) \times (1 - b) \times C_b$$

where N_d is the date cache access time and N_i is the instruction cache access time in machine cycles, b is the branch prediction accuracy expressed as a probability, and C_b is the dynamic

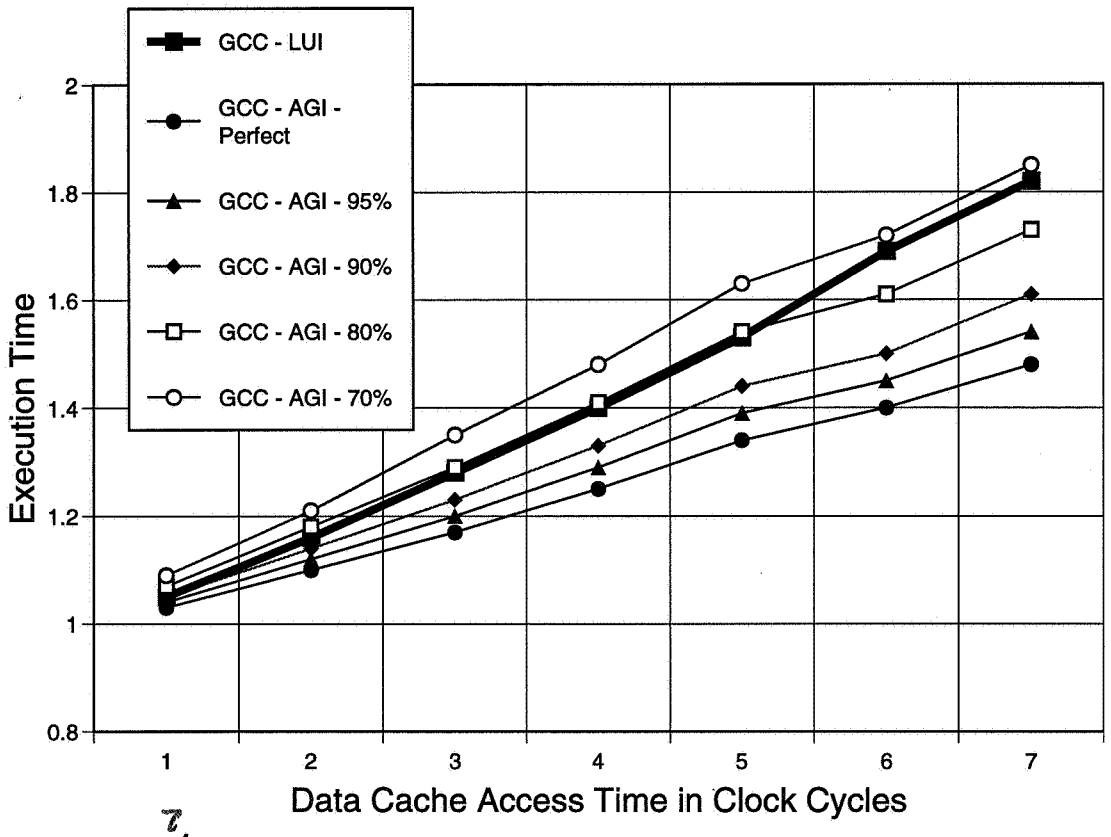

Figure 8: Harmonic mean of all benchmarks. I-cache access time = 1 cycle.

This figure shows the results when branch prediction is taken into account. The AGI pipeline suffers from reduced performance when accurate branch prediction is not available.

branch count of the program. In accordance with the pipeline structures described above, N_i=1 for both pipelines.

For machines with accurate branch prediction, the AGI pipeline still outperforms the LUI pipeline. Once the accuracy of branch prediction drops down to around 80%, the two types of machines perform equivalently. At lower levels of branch prediction accuracy, the early branch resolution of the LUI pipeline allows it to run programs more quickly.

5.2 Pipelines with multi-cycle instruction cache access time

The experiments so far assume that the instruction cache can be accessed in a single cycle—the pipelines described in Section 3 have a single IF stage. As the I-cache latency increases, the penalty for a mispredicted branch increases, because more time is required to fetch the correct instruction from the memory system. In other words, the scope of a branch instruction grows.

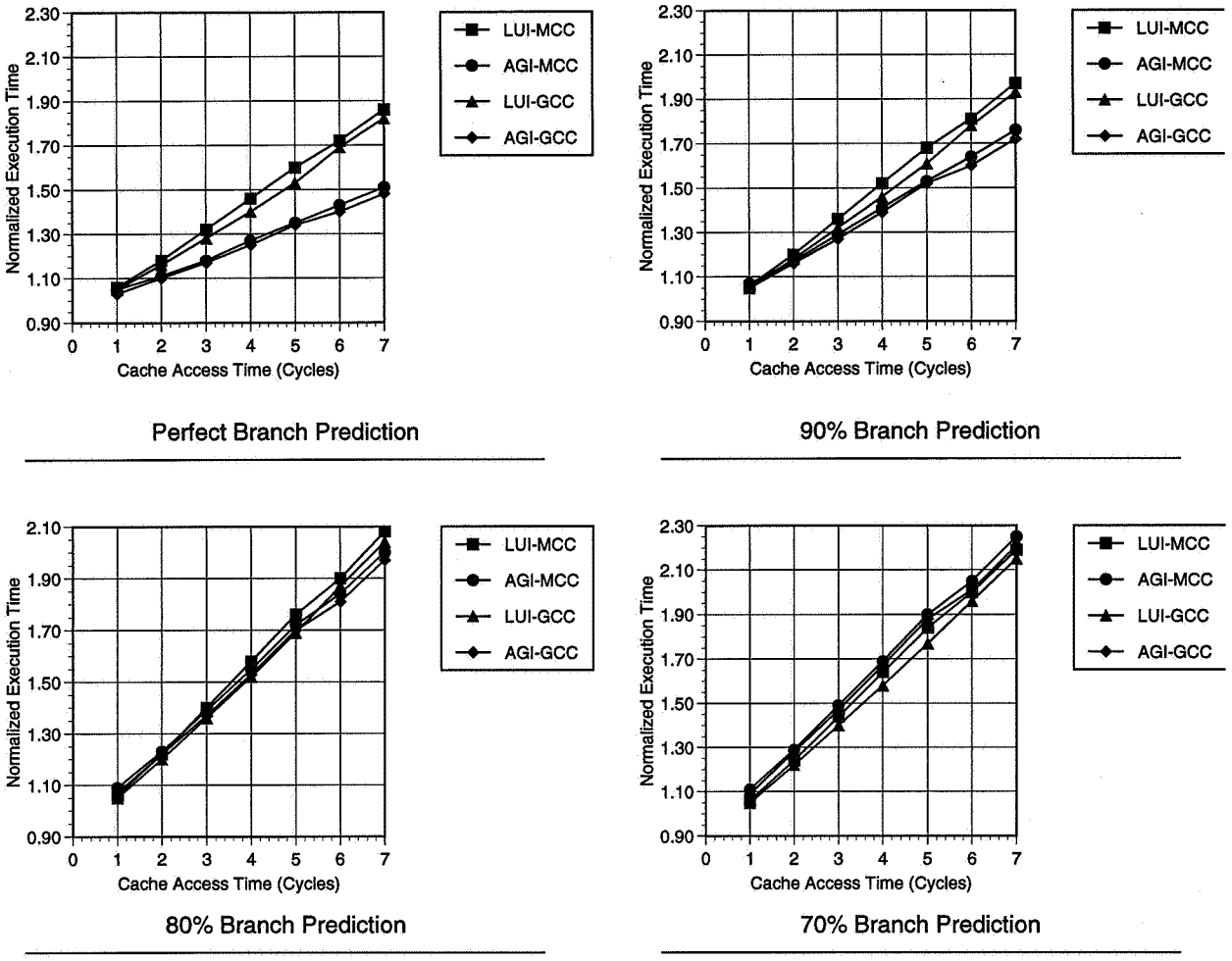

Figure 8: Harmonic mean of all benchmarks. I-cache access time = D-cache access time.

The results labelled MCC have been compiled by the MIPS C Compiler. The results labelled GCC have been compiled by the Gnu C Compiler. The AGI pipeline still requires good branch prediction to outperform the LUI pipeline.

In an LUI system with a multi-cycle I-cache access time, the branch penalty is no longer completely hidden by a single branch delay slot. As a consequence, the requirement that an AGI pipeline have accurate branch prediction to outperform an equivalent LUI pipeline may be eased. Figure 8 shows this is not the case. In this figure, the I-cache access time has been set to equal the D-cache access time. The LUI pipeline experiences a branch penalty in this experiment, but it is less affected by poor branch prediction than the AGI pipeline. Branch prediction still must be better than about 80% accurate for the AGI pipeline to have a performance advantage for machines with slow caches. On machines that have fast caches or poor branch prediction, both pipelines have similar performance.

A chip designer may decide to optimize the speed of the I-cache over the speed of the D-cache in order to ensure a steady supply of instructions. The initial set of experiments represents a machine of this type—the I-cache has an access time of one cycle but the D-cache access time varies. If the I-cache access time is increased to two cycles, the performance penalty for a branch miss on the AGI pipelines increases by one cycle and in the LUI pipeline increases to one cycle because the single branch delay slot cannot hide two cycles of I-cache latency. Figure 9 gives performance for a machine with varying access times for the D-cache but an I-cache access time fixed at two cycles. Once again, the same trend appears. For machines with poor branch prediction or a fast data cache, the LUI pipeline runs faster than the AGI pipeline. When the data cache access time increases and branch prediction becomes more accurate, the AGI pipeline becomes more efficient than the LUI pipeline.

5.3 Pipeline performance on individual benchmarks

Because significant performance improvement is seen in some of the benchmarks, even without sophisticated compiler support, one can examine the properties of the benchmark itself to see where the improvements occur. Figures 10-15 show the results of all experiments on a machine with a 1-cycle I-cache access time for each benchmark. Programs that rely heavily on dynamic data structures see a particularly large benefit from the AGI pipeline. For example, in the benchmark `sc`, which performs spreadsheet calculations, the AGI pipeline outperforms the LUI pipeline even with poor branch prediction. `espresso` and `gcc` also realize significant performance benefits. In these programs, the processor reads from records with many fields. A base register pointing to the beginning of the record needs to be set up, but only once. Once this register is initialized, the values in the fields can be loaded using constant offsets. Only the instruction that sets the base register can cause an address-generation interlock, while each load instruction that follows it has the potential of causing a load-use interlock. Using an AGI pipeline seems to be a good way to increase performance on these "pointer-chasing" benchmarks.

6 Conclusions

A number of processors have recently been announced that eliminate the load-use interlock by overlapping the execute stage of the pipeline with cache access rather than address generation. These AGI machines are designed not only to execute code compiled specifically for them, but

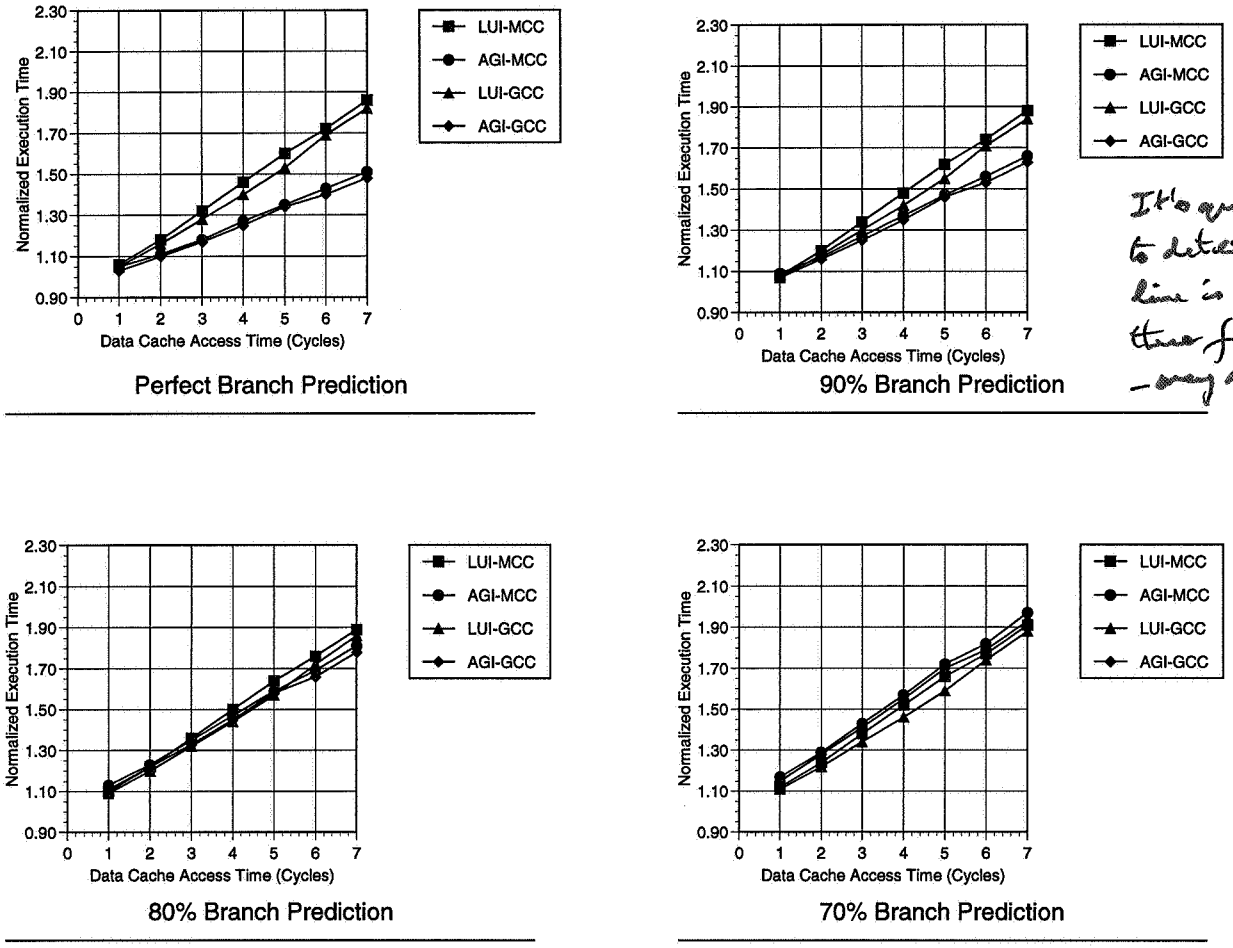

Figure 9: Harmonic mean of all benchmarks. I-cache access time = 2 cycles.

The results labelled MCC have been compiled by the MIPS C Compiler. The results labelled GCC have been compiled by the Gnu C Compiler.

also to run codes compiled for older, LUI, implementations of similar architectures. When good branch prediction methodologies are available, the rearranged pipeline provides improved performance for machines with moderate to large cache access times, even if existing binaries are used. When a branch-delay slot can hide instruction cache latency in an LUI pipeline, high branch prediction accuracy is required for the AGI pipeline to have a performance benefit. As the I-cache access time grows, this trend remains the same.

Simply modifying the compiler's local scheduler shows only a small increase in the benefits of the AGI pipeline. Because basic blocks can be quite short in nonscientific programs, the local

scheduler does not have many instructions to work with. Global scheduling techniques may be able to further improve the performance of the AGI pipeline structure because these methods make more instructions available to be placed between the dependent instructions that cause the interlock.

Two questions remain unanswered. First, in the experiments described in this paper, perfect caches are assumed. In the presence of cache misses, the average time to fetch an instruction and operate on data memory will increase. Cache misses may be distributed such that the effect on these experiments is merely to increase the effective latency to the cache. However, they may be distributed such that pipeline behavior changes noticeably as cache access time and miss rates change.

Second, we have simulated machines that have a single execution pipeline. In a processor with multiple pipelines, each stall cycle can delay the completion of many instructions rather than just one. This may also affect the performance difference between the two pipelines. We leave the study of these two issues as future work.

7 Bibliography

[1] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi, "Streamlining data cache access with fast address calculation," *Proc. 22nd Ann. Int'l. Symp. Computer Architecture*, June 1995, IEEE Computer Society Press.

[2] B. Case, "Intel reveals Pentium implementation details," *Microprocessor Report*, vol. 5, no. 23, pp. 9–17, 1993.

[3] J. H. Crawford, "The i486 CPU: executing instructions in one clock cycle," *IEEE Micro*, pp. 27-36, February, 1990.

[4] M. Golden and T. Mudge, *Hardware support for hiding cache latency*, Technical Report CSE-TR-152-93, The University of Michigan, Department of Electrical Engineering and Computer Science, Ann Arbor, MI, 48109-2122, 1993.

[5] L. Gwennap, "TFP designed for tremendous floating point," *Microprocessor Report*, vol. 7, no. 11, pp. 9–13, August 1993.

[6] L. Gwennap, "Cyrix describes Pentium competitor," *Microprocessor Report*, vol. 7, no. 14, pp. 1,6–10, October 1993.

[7] L. Gwennap, "Intel reveals Pentium implementation details," *Microprocessor Report*, vol. 7, no. 4, pp. 9–17, March 1993.

[8] P. Y. T. Hsu, "Designing the TFP microprocessor," *MICRO*, vol. 14, no. 2, pp. 23–33, April 1994.

[9] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Birmingham, R. G. Oullette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, no. 1/2, pp. 229–248, 1993.

[10] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.

[11] N. P. Jouppi, "Cache write policies and performance," Technical report, Digital Equipment Corporation Western Research Laboratory, 250 University Ave., Palo Alto, CA, 94301, December 1991.

[12] G. Kane and J. Heinrich, *MIPS RISC Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[13] J. K. Iliffe, "A forward looking method of cache memory control," *Computer Architecture News*, vol.15, no. 4, pp. 4-10, September 1987.

[14] S. M. Krishnamurthy, "A brief survey of papers on scheduling for pipelined processors", *SIGPLAN Notices,* vol. 25, no. 7, pp. 97-106, July 1990.

[15] S. A. Mahlke, R. E. Hank, J. E. McCormick, D.I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," *Proc. 22nd Ann. Int'l. Symp. Computer Architecture*, June 1995.

[16] "MIPS chip set implements full ECL CPU," *Microprocessor Report*, vol. 3, no. 12, pp. 1,14–19, December 1989.

[17] O. Olukotun, T. Mudge, and R. Brown, "Performance optimization of pipelined memory caches," in *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 181–190, Gold Coast, Australia, May 1992, IEEE Computer Society Press.

[18] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: history, overview, and perspective," *The Journal of Supercomputing*, vol. 7, no. 1/2, pp. 9–50, 1993.

[19] R. L. Sites, *Alpha Architecture Reference Manual*, Digital press, Maynard, MA, 1992.

[20] J. E. Smith and S. Weiss, "PowerPC 601 and Alpha 21064: a tale of two RISCs," *Computer*, vol. 27, no. 6, pp. 46–58, June 1994.

[21] M. D. Smith, "Tracing with pixie", Center for integrated systems, Stanford University, Stanford CA, 94305-4070, 1.1 edition, April 1991.

[22] G. S. Sohi and E. S. Davidson, "Performance of the structured memory access SMA architecture," *Proc. 1984 Int'l Conf. on Parallel Processing*, pp. 506-513, Bellaire, MI, August 1984.

[23] R. M. Stallman, *Using and Porting GNU CC*, Boston, MA: Free Software Foundation, Inc., 2.4.5 edition, 1993.

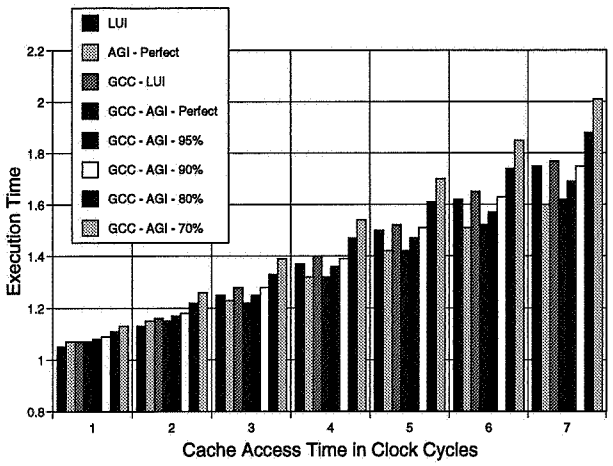

Figure 10: compress

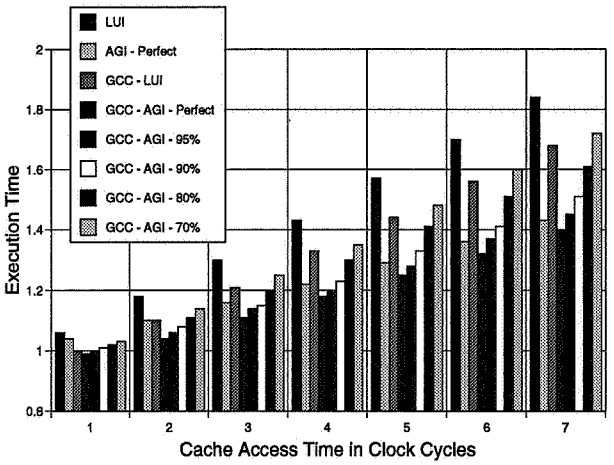

Figure 11: gcc

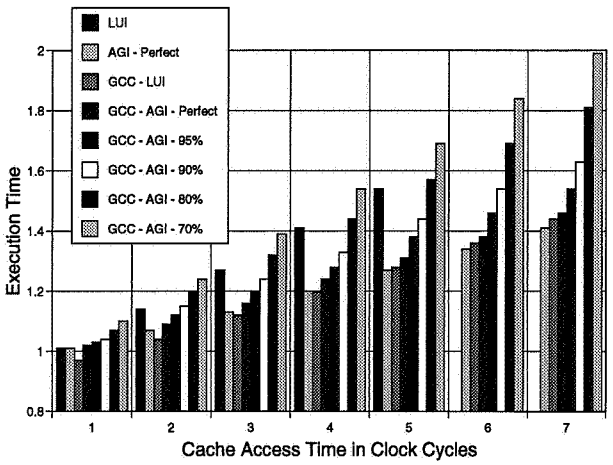

Figure 12: eqntott

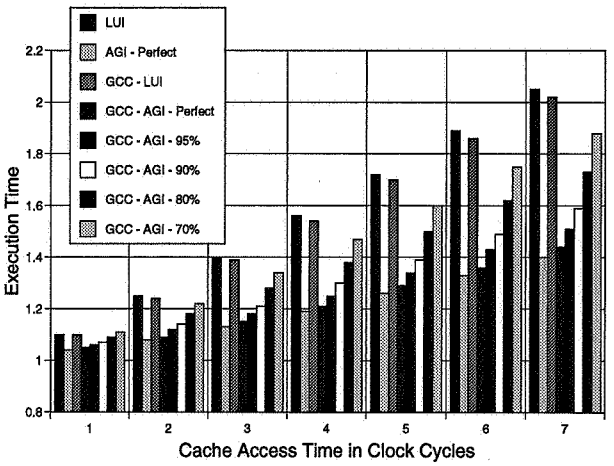

Figure 13: sc

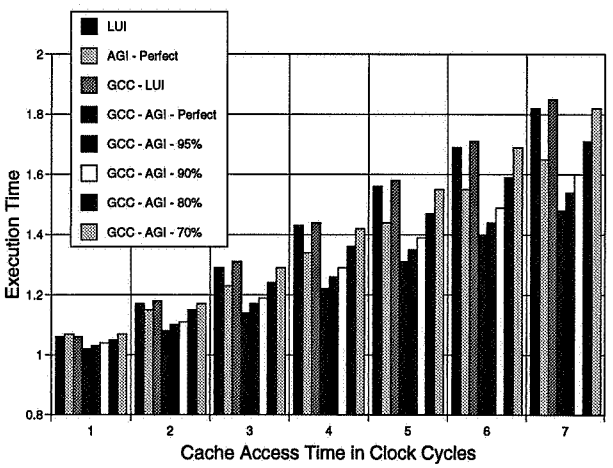

Figure 14: espresso

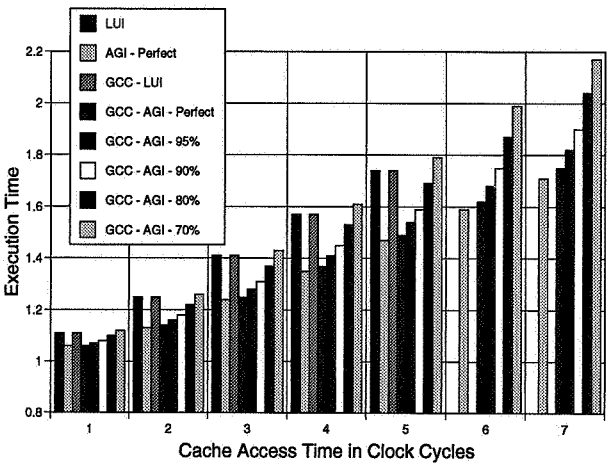

Figure 15: xlisp

In these figures, I-cache access time is one cycle.

Pipeline Architecture

C. V. Ramamoorthy

Computer Science Division, Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California, Berkeley, Berkeley, California 94720

and

H. F. Li

Department of Electrical Engineering and the Coordinated Science Laboratory, University of Illinois at Champaign-Urbana, Urbana, Illinois 61801

Pipelined computer architecture has received considerable attention since the 1960s when the need for faster and more cost-effective systems became critical. The merit of pipelining is that it can help to match the speeds of various subsystems without duplicating the cost of the entire system involved. As technology evolves, faster and cheaper LSI circuits become available, and the future of pipelining, either in a simple or complex form, becomes more promising.

This paper reviews the many theoretical considerations and problems behind pipelining, surveying and comparing various representative pipeline machines that operate in either sequential or vector pipeline mode, the practical solutions adopted, and the tradeoffs involved. The performance of a simple pipe, the physical speed limitation, and the control structures for penalty-incurring events are analyzed separately. The problems faced by the system designers are tackled, including buffering, busing structure, branching, and interrupt handling. Aspects of sequential and vector processing are studied. Fundamental advantages of vector processing are unveiled, and additional requirements (costs) are discussed to establish a criterion for the tradeoff between sequential and vector pipeline processing. Finally, two recent machines (the CRAY-1 and the Amdahl 470 V/6 systems) are presented to demonstrate how complex pipeline techniques can be used and how simple but advantageous pipeline concepts can be exploited.

Keywords and Phrases: computer architecture, pipelining, sequential processing, vector processing
CR Categories: 5.24, 6.33

1. INTRODUCTION

The principle of pipelining has emerged as a major architectural attribute of most present computer systems. In particular, super machines such as the Texas Instruments TI ASC, Burroughs PEPE, IBM System/360 Models 91 and 195, Cray Research CRAY-1, CDC STAR-100, Amdahl 470 V/6, CDC 6600, and CDC 7600 have distinct pipeline processing capabilities, either in the form of internally pipelined instruction and arithmetic units or in the form of pipelined special purpose functional units [1–4].

* Research sponsored by US Army Research Office Contract DA-ARO-D-31-124-73-G157.

CONTENTS

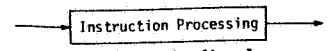

FIGURE 1a. Non-pipelined processor.

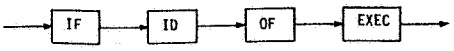

FIGURE 1b. Pipelined processor.

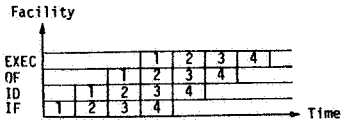

FIGURE 1c. Space-time diagram.

Pipelining is one form of imbedding parallelism or concurrency in a computer system. It refers to a segmentation of a computational process (say, an instruction) into several subprocesses which are executed by dedicated autonomous units (facilities, pipelining segments). Successive processes (instructions) can be carried out in an overlapped mode analogous to an industrial assembly line. So, very loosely, pipelining can be defined as the technique of decomposing a repeated sequential process into subprocesses, each of which can be executed efficiently on a special dedicated autonomous module that operates concurrently with the others.

As an illustration, consider the process of executing an instruction. Normally it involves fetching the instruction, decoding the operations involved, and fetching the operands before it is finally executed. If this process is decomposed into these four subprocesses and executed on the four modules shown in Figure 1(b), four successive independent instructions may be executed in parallel. Specifically, while the EXEC module is executing the first instruction, the Operand Fetch (OF) module fetches the operand needed for the second instruction, the Instruction Decode (ID) module prepares the different operations for the third instruction, and the Instruction Fetch (IF) module fetches the fourth instruction. The overlapped execution among the four modules is best depicted by a space-time diagram. As drawn in Figure 1(c), the horizontal axis represents the time and the vertical axis the space (modules). From this diagram one can observe how independent instructions are executed in parallel in a pipelined processor.

Some theoretical developments and implications of pipelining are reviewed in this section. A top-down, level-by-level characterization of pipeline applications in computers and the associated configuration control are explained in Section 1.2, Pipeline Characteristics. To reveal the fundamental advantages of pipelining, the space-time measure model is employed to illustrate the ideal throughput (performance) of a pipelined system with no external restrictions or dependencies. This pictorial measure applies to a pipeline of any level operating in an ideal environment. Besides the ideal performance, the limitation of this technique to the lowest level in a computer, namely the logic gate level, is surveyed. Here a practical limitation to the ultimate

speed achievable arises because the technique requires the insertion of latches of finite delay. It is shown that this delay plays a significant role in determining the bound on the fastest speed achievable.

On the other hand, when a pipeline operates on tasks with precedence constraints, the space-time measure for the ideal situation is not directly applicable. Section 1.3, Performance Characteristics, analyzes the performance of such a pipe when the precedence relationships are in the form of a tree. Appropriate bounds are provided which reflect that the pipe sometimes has a throughput rate close to its segment time and at other times has a rate close to its flush time. The dominating role played by task relationships in an actual pipeline is thus apparent.

After the analytical evaluation of a pipeline's performance, the various applicable control schemes are classified and compared with respect to the flow of instructions and the resolution of conflicts. This classification covers most of the schemes existing in pipelined systems as well as some theoretically feasible combinations. In Section 1.4, Control Structure, Hazards, and Penalties, three kinds of hazards are formally classified. The detection and resolution techniques for these hazards under either "streamline" or "fully asynchronous" control are analyzed according to the incurred cost in hardware and incurred delay penalties in runtime. Section 1.5, Sequencing Control, presents a simple sequencing control using shift registers as an example of synchronous pipelines whose collisions are predeterminable. This scheme is useful for controlling lower level pipelines such as arithmetic pipes for which external conditions or precedence constraints are rare. Finally, in Section 1.6, Software Aspects, some software problems related to the efficient code generation of a vector pipeline are discussed.

In Section 2, Structure of a Pipelined Processor, the problems and solutions associated with a sequential pipelined system are examined more carefully. Three systems are used as examples to make cross-comparisons in several practical problems. These problems include: 1) buffering for smoothing congestions; 2) busing structure to reduce delay penalties; 3) branch handling to reduce the disruption of flow; and 4) interrupt handling to ensure a proper interrupt response and later recovery. In Section 2.6, Pipeline Processing of Arithmetic Operations, an example of pipelining fast multipliers is provided to illustrate how a lowest level pipeline can be effectively designed. Such pipelines can often use control schemes like the one in Section 1.6.

In Section 3, Vector Processing, many special characteristics associated with a vector pipelined system are analyzed separately. Vector pipelines have become economic ways to achieve high throughput for application with suitable parallelism. Specifically, jobs with identical transformations on a set of data can be carried out with minimal control overhead (instruction) and high speed. Two prominent machines, the TI ASC and the CDC STAR-100, are examined. To provide a clear picture, a detailed example of a typical vector instruction (format and execution) is provided. From it one can realize how to use vector instructions and how to achieve skewing on data elements. An analytical comparison between the performance of a vector pipe and that of a sequential pipe is also furnished. This comparison reveals where vector pipelines bring in speedup; however, the additional demands of vector pipelines for proper instruction sets, proper choices of algorithms, and intelligent compilers are also exposed.

Finally, in Section 4, Overview of Two Recent Machines, the special characteristics of two recent pipelined computers are surveyed. The chaining in the CRAY-1 is an example of pipelining applied between vector instructions. With it a very high throughput can be obtained. It is also interesting to observe that the simple pipeline design for the Amdahl 470 V/6 system has proved to be a success.

1.1 Historical Perspective

Computer designers have exploited the overlapped mode of operations since the late fifties. We recount only some significant milestones in its development. For a

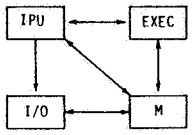

FIGURE 1d. Basic computer structure.

clear understanding of this development, let us model the computer system by the following subsystems (Figure 1(d)):

1) The instruction processing unit (IPU) performs the instructions fetch and decode, and fetches the operands (if any) required for the instructions.
2) The execution unit (EXEC) performs the desired operation on the operands.
3) The input/output unit (I/O) controls the peripheral devices.
4) M represents the primary memory of the computer system.

The earliest use of overlapped operations between the CPU (IPU+EXEC), the memory unit (M), and the input/output unit (I/O) can be found in the UNIVAC I, developed during the early fifties. Here the central processor initiates an I/O process; then the CPU and the I/O proceed concurrently. When the I/O operation is completed, an interrupt signal is issued by the I/O controller to alert the CPU of the completion. This asynchronous I/O processing avoids having the CPU wait for the completion of I/O tasks and improves the throughput.

Another type of pipelining where overlap is achieved between the instruction processing unit and the execution unit, has been exploited by later machines. For example, the IBM 7094 used this type of overlap to its advantage. With a 72-bit-wide memory with a memory cycle time of approximately 2 μsec, it executed on the average an instruction (32 bits) with 32-bit operands in two cycles or 4 μsec [22]. With an interleaved and faster memory [1.4 μsec cycle time], the 7094 II achieved an average execution rate of one instruction per cycle of 2 μsec. The Honeywell H-800 (1959) pioneered in multiprogramming, overlapping of I/O, and concurrent computing among a number of programs resident in the memory. Similarly, the Univac LARC (1961) uses interleaved memory and a four-fold overlap (instruction fetch, indexing, data fetch, and execution) and can run one floating-point add per cycle of 4 μsec [22].

Pipelining within the instruction processing unit was implemented in the IBM 360/91 in the sixties. Some functions of the execution unit were also pipelined—for example, the addition and the division processes.

1.2 Pipeline Characterization

Since pipelining can be applied at more than one level, a top-down, level-by-level characterization of pipelining can be conveniently established for analyzing a pipelined system. A pipe can be further distinguished by its design configurations and control strategies. These two points are elaborated below.

1) *Levels of pipelining:* Pipelining at the system level is exemplified in the design of the instruction processing unit. The IPU can be decomposed into various functional segments—instruction fetch, instruction decode, address generation, etc. (Figure 1(b)). It takes one minor cycle for a task (instruction) to pass through each segment. Thus, after a stream of tasks enters this pipeline, the pipeline starts outputting one task per minor cycle. Microprogram prefetch—that is, overlap of decoding the current microinstruction with fetching the next microinstruction—is another example at this level.

The next level for the application of pipelining is the subsystem level, typical examples of which are the pipelined arithmetic units. Pipelined add, multiply, divide, and square-root functions have been in existence in a number of contemporary computer structures. Figure 1(e) is the conceptual representation of the operation of the divide unit of the IBM 360/91, where, as D_i iteratively approaches 1, N_i approaches N/D, the quotient.

2) *Pipeline configurations:* In addition to the hierarchical levels of pipelining, differ-

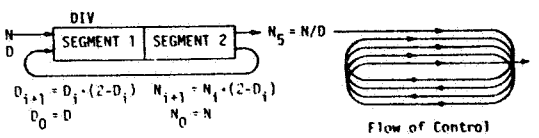

FIGURE 1e. IBM 360/91 divide.

ent design and control strategies classify a pipelined module into one of two forms; it can be either a static or a dynamic pipe. Sometimes a pipelined module only serves a single dedicated function—for example, the pipelined adder or multiplier in the IBM 360/91. Naturally, it can be termed a *unifunctional* pipe with a *static* configuration. On the other hand, a pipelined module can serve a set of functions, each with a distinguishable configuration. For example, in the TI ASC system the arithmetic unit in the processor is a pipe that has different configurations (interconnections of modules) for performing different types of arithmetic operations. Such a pipe is called a *multifunctional* pipe. A multifunctional pipe can be either static or dynamic. In the *static* case, at any time instant only one configuration is active, therefore pipelining (overlapped processing) is permissible only if the tasks (instructions) involve the same configuration. Most, if not all, multifunctional pipes in arithmetic units of existing machines fall into this classification because static pipes are easier to control, as will become clearer later on. *Dynamic* multifunctional pipes permit overlapped processing among several active configurations simultaneously. Throughput may be further enhanced, but more elaborate control and sequencing are required. This classification of static and dynamic pipes will be very useful when we consider and evaluate pipelined processor architecture in subsequent sections.

1.3 Performance Considerations

In this section, the advantages, requirements, and limitations of pipelining are reviewed.

1) *Throughput considerations:* One of the most important performance measures of a system is its throughput rate, defined as the number of outputs (sometimes the number of instructions processed) per unit time. It directly reflects the processing power of a processor system—the higher its throughput rate, the more powerful the system is. Pipelining is one specific technique to improve throughput, as is the use of faster modules.

For this discussion, let us reconsider the example in Figure 1. For a nonpipelined processor, the execution time of an instruction is $T_{np} = t_1 + t_2 + t_3 + t_4$. Therefore, for every T_{np} units of time an instruction is completed; this corresponds to a throughput rate of $1/T_{np}$. In the pipelined case, suppose $t_b = \max\{t_1, t_2, t_3, t_4\}$ = speed of the slowest facility in the pipeline. Then its maximum throughput rate is $1/t_b$, because for every $T_p = t_b$ units of time, an instruction can leave the pipeline after its execution, if instructions are independent. A direct comparison shows that $T_p < T_{np}$; hence the throughput rate of the pipelined processor ($1/T_p$) can be much larger than that of the nonpipelined processor. If $t_1 = t_2 = t_3 = t_4$, then the comparison can show a fourfold throughput improvement. From this result we can anticipate that a high degree of parallelism leads to a high throughput rate.

The throughput of a pipeline is determined by its slowest facility, or "bottleneck." The throughput can be improved by subdivision of the bottleneck element (Figure 1(g)) or by putting facilities in parallel (Figure 1(h)). Both techniques are useful in removing bottlenecks. However, putting facilities in parallel creates more problems in distribution and synchronization of the tasks in the pipeline.

2) *Efficiency considerations:* Another important performance measure for a system is its efficiency, sometimes called its utilization factor. Efficiency also directly re-

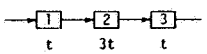

FIGURE 1f. Facility 2 is the bottleneck.

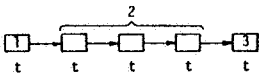

FIGURE 1g. Subdivision of facility 2.

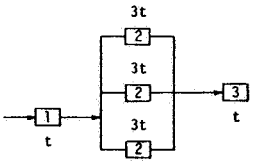

FIGURE 1h. Paralleling of facility 2.

flects how effective a processing scheme is and can be used to indicate how future improvements, such as removal of bottlenecks, should progress. Like most performance measures, it can be evaluated both analytically and experimentally by measurements. Here an attempt is made to illustrate the analytical efficiency of pipeline processing via the space-time relationship introduced earlier.

It is natural to view efficiency as the percentage of busy (productive) periods with respect to a certain time span. Here a slight complication arises because a pipelined processor consists of several modules, some of which may be busy while the others are idle. To evaluate the efficiency of the processor system as an entity, Chen [7] proposes a uniform space-time span index:

$$\text{efficiency of pipeline} = \frac{\text{total space-time span of tasks}}{\text{total space-time span of facilities}}.$$

where the term "task" (process) is used to fit the loose definition of a pipeline. Sometimes the modules in the pipeline are of different natures with different importance (or cost) factors. A refined index which also includes such considerations has been suggested in [8]:

$$\text{efficiency of pipeline} = \frac{\text{total weighted space-time span of } L \text{ tasks}}{\text{total weighted space-time span of } n \text{ facilities}}.$$

For example, for a linear pipeline like the one in Figure 1 (where there is no looping inside the pipeline, so that a task will flow through each facility only once), an analytical efficiency measure can be expressed as follows (assuming the execution time of each module is time invariant):

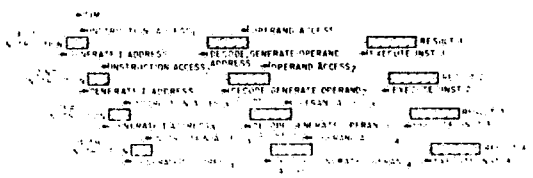

FIGURE 2. IBM 360 Model 91 instruction sequencing illustration.

η = efficiency of linear pipe

$$= \frac{L\left(\sum^{n} \alpha_i t_i\right)}{\sum^{n} \alpha_i \left(\sum^{n} t_i + (L-1)t_j\right)}$$

where t_j is the speed of the slowest facility (bottleneck); t_i is the speed of the ith facility in the pipeline; α_i is the weight associated with the space-time span of the ith facility as determined by its importance, such as cost-speed factor; L is the number of tasks (instructions) pumped into the pipeline in a certain period of time assuming, for highest efficiency, that tasks are pumped in continuously); and η is the total number of facilities in the pipeline. (See Figure 2.)

In the ideal situation in which all modules have the same speed, the equation simplifies to

$$\eta = L/(n + (L-1));$$

so, when L approaches infinity (in the steady state of processing), the efficiency may approach unity. In all others cases, as L approaches infinity, the efficiency approaches

$$\eta \to \left(\sum^{n} \alpha_i t_i\right) \Big/ \left(\sum^{n} \alpha_i\right) t_j < 1.$$

Two observations should be noted at this point. First, this equation holds whether or not there are additional buffers inside the pipeline because of the linearity assumption. As is demonstrated later, buffering is an important tool for increasing throughput in many practical pipeline designs—for example, when two or more EXEC modules are available and one is a bottleneck. Second, in deriving the equation it has been assumed that a continuous supply of tasks (instructions) is available. In reality, execution may be discontinued for such reasons as precedence constraints, branching, interrupts, etc.

3) *Clock rate and maximum speed limitations:* As data and control flow from one pipe segment to another, the propagation delay through each segment and the possible signal skews have to be carefully balanced to avoid any improper gating in a high speed situation. In the maximum speed

pipeline design, all segments have to be synchronized by the same clock for propagating the data through the pipe.

The study of a maximum clock rate serves to place a practical bound on the throughput achievable in a pipeline system limited by the propagation delays of the logic gates used. Several studies have been carried out to examine this problem under various assumptions of timing parameters. In all cases, three necessary conditions of signal balancing exist:

1) The data must be gated by a clock wide enough to insure a properly stabilized output;
2) The clock should not be too wide to allow data to pass through two or more segments within the same clock; and
3) The data that passes through a segment should arrive at the next segment before the next clock begins.

Initially Cotten [27] tested this data rate and latching problem by using a hypothetical circuit as in Figure 3. The clock for various segments may have a skew S_c, defined as the time difference between the arrival of the same pulse at different gates. The latch register is assumed to be composed of two gate levels with feedback connections. Then, under conditions 1) and 2), Cotten's clock requirements are:

$$C_T - S_c \geq 3t_{\max} - t_{\min}, \quad (1)$$

$$C_T + S_c \leq 4t_{\min}, \quad (2)$$

where C_T = clock width, S_c = clock skew, $t_{\max}$ = maximum single gate delay, and $t_{\min}$ = minimum single gate delay.

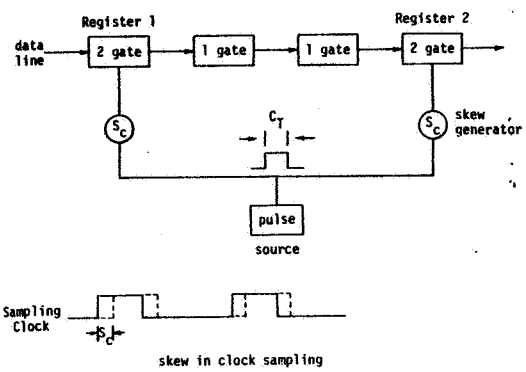

FIGURE 3. Clock rate conditioning.

These two requirements can be combined to form

$$3t_{\max} - t_{\min} + S_c \leq C_T \leq 4t_{\min} - S_c. \quad (3)$$

Although here a segment of the pipe has been assumed to be composed of two gates, one can further the derivation by including condition 3). Then a third constraint exists as

$$C_T + C_{\bar{T}} \geq 2t_{\max} + T_{\max} + S_c, \quad (4)$$

where $C_{\bar{T}}$ = inverted clock width and $T_{\max}$ = maximum propagation delay through the segment (excluding the latch).

Under Cotten's assumptions, the complete set of constraints for a general pipe segment is:

$$3t_{\max} - t_{\min} + S_c \leq C_T \leq T_{\min} + 2t_{\min} - S_c \quad (5)$$

and

$$2t_{\max} + T_{\max} + S_c \leq C_T + C_{\bar{T}}. \quad (6)$$

Consequently the minimum clock period can be derived to be $(C_T + C_{\bar{T}})$, which satisfies the above constraint and also $C_T + C_{\bar{T}} \geq 2 \min C_T$ (that is, the period must be long enough to allow the data to propagate through a latch and then remain stable for min C_T). Under zero skew and $t_{\max} = t_{\min} = t$,

$$C_T + C_{\bar{T}} \geq 4t.$$

This marks the highest frequency possible in an ideally synchronized system. If S_c is nonzero, then $C_T + C_{\bar{T}} \geq 4t + S_c$, and the frequency has to be decreased.

Besides the positive clock skews, other skews may exist, such as skew between C_T and $C_{\bar{T}}$. In [28], Hallin and Flynn propose another set of constraints that includes the skew, called ϵ, and any uncertainty thereof:

$$T_D \geq 2t_M + S_c + \epsilon + U_c \quad (7)$$

$$2t_M + S_c \leq C_T \leq T_D + d \quad (8)$$

where T_D = propagation delay of a segment; t_M = propagation delay of a gate; ϵ = skew between C_T and $C_{\bar{T}}$; U_c = uncertainty in the clock width; and d = minimum length pulse to change a gate output.

While Equations (7) and (8) include the additional terms of ϵ and U_c, the distinctions of the minimum and maximum values of all propagation delay parameters have been ignored. Yet those distinctions are of prime importance in verifying the functioning of the circuit. One can further the derivation of the corresponding constraints, keeping conditions 1), 2), and 3) satisfied, under different sets of skews that may appear in the circuit. In any case, the ($4t$) period always places an absolute lower bound if a register latch is composed of two gates. If a gate delay is 2.5 nsec, the maximum frequency will be 100 megacycles, corresponding to a segment time of 10 nsec.

4) *Design optimization:* Design optimization for pipeline systems shares most of the fundamental difficulties of any system design. One such difficulty is to abstract a proper objective for optimization. No general objective is sufficient to describe individual situations, so individual objectives have to be formulated and solved.

One common approach is to look at the cost-effectiveness, or the cost-speed product. A given processor pipe can be segmented in various ways, resulting in different cost and speed parameters. In a synchronous pipe, such as a multiply or add pipe, a first order model of optimization may be used. The pipe is partitioned into k segments, and the resulting throughput and cost are:

$$\text{segment time} = T/k + \lambda$$

$$\text{cost} = \alpha k + \beta$$

where T = time for the nonpipelined case; λ = latch time; α = cost of each segment (assumed to be the same); and β = initial cost. Thus

$$\text{cost-speed product} = (T/k + \lambda)(\alpha k + \beta).$$

Lemma. Under the first order model, the optimal segmentation for a pipe is $k = (\beta T/\alpha\lambda)^{\frac{1}{2}}$ (assuming a continuous space for optimization).

This result can be derived directly from minimizing the cost-speed product. It is useful for homogeneous straight-line pipes such as a pipelined adder where each segment cost and speed can fit into the characterization. For other cases, the first order model does not apply very well; then the segment speed $f(T, k) + \lambda$ and the cost $g(\alpha, k) + \beta$ for some discrete functions f and g specified by an alternative scheme are available. Consequently, minimizing the cost-speed product here corresponds to minimizing $(\beta f + fg + \lambda g)$, for which an integer programming algorithm is necessary to efficiently enumerate partially all possible schemes. An example of pipelining a processor can be found in [29].

As mentioned earlier, cost-effectiveness may not be a good objective. In some cases, the design objective is to minimize the cost while satisfying some speed constraint or vice versa. This is typically the case because the throughput of a pipe sometimes is not restricted simply by its segment speed, but also by other outside parameters such as memory speed. In other words, the local optimization has to be performed relative to the global system, leading to integer programming problems that involve semi-exhaustive algorithms for optimization. For example, for a linear pipe, a dynamic programming formulation of complexity $O(M^2N)$ is applicable where N is the number of segment nodes and M is the cost constraint. However, for systems that are not linear more complex iterative algorithms are needed.

5) *Bounds on execution time and efficiency:* For the purpose of establishing some upper bounds of a pipe in executing certain typical but related set of operations, the following theoretical model can be used. Here a pipe is characterized by the number of segments it contains, where each segment has the same synchronized speed.

Ideally, when the work to be accomplished has no internal precedence constraints, the maximum throughput can be attained with one output per segment clock. The existence of precedence requirements inhibits the continuous initiation of the pipe, resulting in lower throughput. The most common type of precedence structure is that of a tree.

One special problem of interest is: Given a pipe of m segments, what is the time bound needed to compute the sum (or product) of n numbers, assuming that each segment time is 1 unit? If $n \geq 2m$ and is a power of 2, the pipe is kept busy until

$(m - 1)$ computations are left, with the needed (intermediate) results residing in the m segments. They will take an additional $(m \log_2 m + m - 1)$ units to complete. So altogether, the $(n - 1)$ computations take $(n + m \log_2 m - 1)$ units. On the other hand, if $2 \leq n \leq 2m$, $(n/2 + m \log_2 n - 1)$ units are required. The corresponding efficiencies, defined by the ratio of the total busy segment times to the total segment time span, can then be derived easily as $(n - 1)/((n - 1) + m \log_2 m)$ and $(n - 1)/(m/2 + m \log_2 m - 1)$, respectively. This implies that for $n \gg m$ the pipe of m segments functions almost like a nonpiped processor with speed of one segment time instead of m segment cycles (the total time is $O(n)$, not $O(mn)$). However, for smaller n the time is $O(m \log_2 n)$.

The previous special case assmues a set of uniform operations on a set of data, merging them into one result where the exact order of merging is unimportant. In the case where a specific tree is to be followed (the precedence structure is fixed), other lower bounds can be derived in a similar fashion. It can be shown that, for a general tree (not necessarily binary), if each node i is labeled by $\ell(i)$ corresponding to its distance from the root, then execution of the nodes according to priorities in descending order of $\ell(i)$ in a pipeline environment with identical pipeline characteristics is optimal. Therefore the shortest execution time can be achieved if the nodes are executed according to priorities corresponding to their level labels. However, if the pipes have different structures and/or capabilities, the problem becomes NP-complete. Without going into the latter case, the time bound for the former case can be derived, given a tree structure and a pipe structure (latency and flush time).

First let L_j be the number of nodes of the tree with label j where $l \geq j \geq 0$. For the simple case that there exists a J_0 such that for $\ell \geq j > j_0$, $L_j \geq m$ and for $j \leq j_0$, $L_j \leq m$, the time bound is given by

$$\sum_{j=l}^{j_0+1} L_j + m(j_0 + 1) - 1,$$

and this time bound is exact (from the optimality of the level algorithm). This asserts that once the "critical level" j_0 is reached, the rest of the tree of j_0 levels needs precisely $m(j_0 + 1) - 1$ to flush. So the time complexity is $O(mj_0)$ if the former term is less significant.

Generalization of the simple case will lead to a more complex bound. One way to derive the bound is to chop L_j's into pairs of sections, each of which corresponds to a simple case as depicted in Figure 4. Then if L_j' is the number of nodes left at level j when *all* nodes at level j first become either ready or initiated (since some may have already been computed or initiated), the bound is

$$\sum_{i=1}^{u} \left[\sum_{j=j_{i-1}}^{j_i} L_j' + m(j_i - j_{i+1}) \right] + m - 1.$$

In deriving these bounds, it has been assumed that each node takes the same processing structure and has the same flush time. On the other hand, if more than one pipe exists, the bounds are much more complicated. First, the control of multiple pipelines, specifically the routing of intermediate results, is a practically unsolved problem. While the short-circuit (short-stop) path exists from the output of a pipe to its own input, the disjoint and direct update paths between pipes either incur too much cost or cause too much interference. Aside from this practical restriction, theoretically, with multiple pipes, similar time bounds are derivable. In the case of computing the sum or product of n numbers in a system with p pipes, assume $n = p\ell$ for some ℓ. If $\ell \geq 2m$ (integer powers of 2), then the time bound is $[\ell + m (\log_2 m + \log_2 \ell) - 1]$. If $\ell \leq 2m$, it is $[\ell/2 + m (\log_2 n + \log_2 \ell) - 1]$.

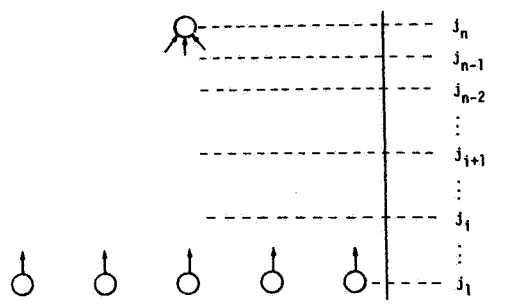

FIGURE 4. Partitioning into simple cases.

Speedup is achievable since $\ell < n$. For a general tree, the routing problem may be more severe and cause more interference. If one ignores these difficulties, bounds similar to those derived for single pipe may be obtained.

1.4 Control Structure, Hazards, and Penalties

The control structure of an overlapped or pipeline system is often overlooked in the literature. Yet it plays such a significant role in characterizing the system under study that it determines the resulting operational efficiency.

In an overlapped pipeline system, two major control structures can be distinguished, and these have been implemented on several systems. The first and simpler kind involves a streamline flow of instructions through the system, with one instruction (task) following another, such that the completion ordering of the instructions is the same as their initiation ordering. Therefore if the system is depicted by a sequence of functional modules, the instructions flow through them one after another, with simple interlocks between two adjacent segments to allow the transfer of data control from one segment to the next. Interlock is necessary because the pipe is asynchronous, and some segments may have speeds different from others or variable depending on the control information. When a bottleneck appears dynamically at a segment, the input will be halted temporarily until the segment is free again.

The second type of control structure is more flexible and powerful, but also more expensive. Here the system can be viewed as fully asynchronous, so that completion ordering of the instructions need not be the same as their initiation ordering. In fact, when an instruction is held up because of some hazard condition, the next instruction may be allowed to go ahead. Such a scheme is desirable whenever the system has multiple (either physical or virtual) execution units or facilities running in parallel (besides the pipelining employed). Then the system resources can be better utilized, despite the occurrence of some undesirable events. Besides, in some cases the execution time of one instruction may be very different from that of another, and it is only natural to allow a subsequent short instruction to finish ahead of a preceding (but independent) long instruction.

The first type of control structure is used by such systems as the IBM 7094 and 360/75, and even the apparently more powerful TI ASC. Representative of the second type of control structure are the CDC 6600, the IBM 360/91, and the STAR-100 systems.

We now look at the fundamental problems to be solved by either type of control as well as the means and complexities involved. For any asynchronous system, three sources of control problems exist: 1) Read after write, 2) write after write, and 3) write after read. Their significance is worth more elaborate explanation.

Read After Write

Because of the simultaneous execution (though in different phases) of several active instructions, the data needed by these instructions has to be guaranteed to be correct. For two "active" instructions, say i and j (j being an immediate successor of i), if i writes into a region and j needs to fetch some control or data from the same location in that region, a "read after write" phenomenon occurs. (The term "region" is a flexible term that refers to any storage element, e.g., a register or main memory, as depicted in Figure 5(a).) To synchronize i and j properly, j has to defer fetching that value until i has completed; otherwise, the wrong information (data or control) is used and the control scheme fails.

Write After Write

Quite analogously, if the instructions i and j write into the same region, even if i completes after j (this may occur when i is a long instruction or when something delays i), the resulting value stored in the region should reflect the result of instruction j, not i. So, to guarantee correct execution, the control structure has to resolve any such occurrence.

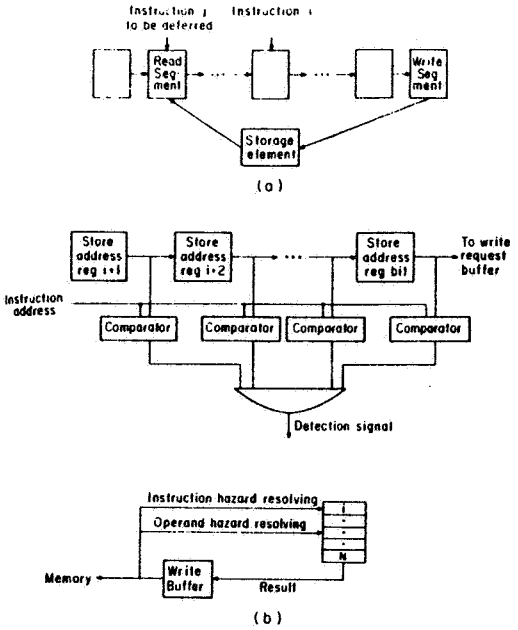

FIGURE 5a. If i writes into the location to be referenced by j, a read-after-write crises arises.
FIGURE 5b. Read after write detection and resolution.

Write After Read

This problem is less severe and rarely occurs except in some special cases. It involves the completion of a read before the next write to the same region takes place. Usually, if a read is initiated (to the memory), even if memory interference delays the actual read, a subsequent write to the same location will still follow the read. For register reference, the interference problem is less severe (resolved faster). A potential situation where such a problem may need further control is when the read and write requests have separate request queues; requests on both queues then have to be synchronized for the write after read to guarantee that the write follows the read.

These three basic problems need specific controls. We explore these problems realizing that the type of control structure of the machine does play a deciding role in determing how they are resolved.

1) *Read After Write*

The read after write problem can be further decomposed into three subproblems, depending on the location of the region and the value type (control or data).

a) *Instruction hazard:* An instruction hazard occurs when the instruction to be initiated (decoded, etc.) is fetched from a location that is yet to be updated by some uncompleted instruction downstream. The instruction initiation must be halted until the read after write problem is resolved.

b) *Register hazard:* In this case the region of conflict is in the register, the contents of which are needed either to compute an effective address or to form one of the operands needed. Again, the instruction (j) involved has to be deferred until the previous write (into the register) is completed.

c) *Operand hazard:* Similarly, if the conflict is at a memory location, an operand hazard occurs. The resolution is to wait until the previous write is completed. Sometimes this process can be speeded up by providing a short-circuit path from the write buffer to the segment needing the operand so that the read from memory is avoided completely. These solutions are explained later.

These three hazards need separate detectors and resolvers. The location of a detector and the complexity of a resolver decide the penalty (time delay in initiation) that is incurred by the hazard. A formalism of such control complexity and penalties is developed in [32]. As an example, suppose that (under a streamline control structure) an instruction hazard is to be detected at segment i of an N-segment pipe such that each segment can hold at most one instruction. To detect the hazard, $N - i$ store address registers have to be installed, one at every segment after i (assuming that a store address has been developed after segment $i + 1$). The instruction to be initiated at segment i has to be checked first by comparing the program counter (the address of that instruction) with the contents of the $N - i$ store address registers. A simple detection and resolution scheme is depicted in Figure 5(b). Upon the detection of that hazard, the instruction (at segment i) is halted while the instructions downstream continue to flow. Finally the detection yields a null signal when the conflicting

(predecessor) instruction exits from the pipe. Then the updated instruction can be fetched from the memory (after store) or directly from the store buffer. In the former case, it incurs an additional penalty, although it is cheaper to control (since the read request is not generated until the write is initiated, as the detection signal switches). In the latter case, a tagging and direct route has to be established from the write buffer to segment i.

The penalty for an instruction hazard, which is the additional delay to the initiation of the instruction at segment i which otherwise would not occur, is

$$\sum_{j=i+1}^{N} t_j + t_{u_I},$$

where t_j is the execution time of segment j and t_{u_I} is the update time for an instruction hazard with one minor cycle for routing using time less than or equal to $t_{u_I} \leq 2t_m$, the memory cycle time.

Similarly, a register hazard must be detected before a register value is used (otherwise, a roll-back scheme of instruction processing is needed). If the hazard is detected at segment k, the detection and resolution scheme is similar except that it now takes $(N - k)$ register-address registers for comparison. The penalty for a register hazard is

$$\sum_{j=k}^{N} t_j + t_{u_R},$$

where t_{u_R} is the update time for a register hazard, usually around one or two minor cycles.

Finally, the operand hazards, similar to instruction hazards, can be detected by comparison with the already existing storage address registers downstream. Usually the location of this detector is further downstream than the one for instruction hazards because operand fetch can be carried out later than instruction control setup. Suppose it is detected at segment ℓ, the penalty is

$$\sum_{j=\ell}^{N} t_j + t_{u_0}$$

where t_{u_0} is the update time for operand hazard, which usually is equal to t_{u_I}.

To sum up, the total control cost of the detector and resolver for all three types of hazards is: $\max(N - i, N - \ell)$ storage address registers; $(N - i) + (N - \ell) + (N - k)$ comparators (parallel comparison); and $N - k$ register address registers for a streamline processing system. The penalties are as previously specified.

In some cases more than one instruction may be allowed to reside in a segment (additional buffering) whose speed is highly variable (in which case buffering will help to smooth out the flow). The extension of the previous lemma will be omitted here. However, similar control and resolution schemes may be used.

For fully asynchronous systems, the detection and resolution control is more complex, since initiation is not halted when a read after write occurs. Rather, the instruction is paused, but subsequent independent instruction(s) may proceed, thus bypassing the instruction that has to wait for the previous write. The ways to accomplish this bypassing can be divided into two strategies: centralized and decentralized. A typical representative of the centralized policy is the scoreboard used in the CDC 6600, which contains information about each execution unit, the operand availability of the registers, and the status of each facility. The decentralized policy is represented by the common data bus (CDB) used by the IBM 360/91, in which tags are used, in addition to the detection necessary for instruction hazard detection at the IPU.

First the instruction hazard can be detected in a similar way by comparing its address with all store addresses yet to be completed. Usually register or operand hazards do not halt the instruction stream; for example, an instruction needing a yet-to-be-written operand can be continually forwarded to an execution unit to wait until the write is completed while subsequent instructions proceed. So, besides the ordinary detection, the resolution needs additional control hardware and time. The CDB of the 360/91 represents such a flexible tagging scheme. A tag is associated with any instruction needing such an operand, and it reflects the source of that operand. By updating the tags and routing operands

according to tag values, the read-after-write can be monitored properly. A CDB can be depicted by S sources (that generate or forward operands) and T sinks (that need the operands). Thus the added control involves $O(T)$ tag registers, each of length $\log_2 S$, in addition to the comparison circuit to route operands (T comparators plus gating control).

A centralized scheme involves similar complexity. The disadvantage of the centralized scheme is that it can reinitiate (update) only one sink at a time so that the delay t_u can be longer than that in the decentralized case with parallel updating (same tag).

The penalty of hazards in fully asynchronous systems is less severe and less well defined. One possible way to view this penalty is to represent it by the waiting time of the instruction causing the hazard, which is a random variable depending on the completion time of its predecessor instruction. Such a stochastic characterization is omitted here.

2) *Write After Write*

The write after write problem does not exist in steamline systems, provided the write buffer is served sequentially. However, in fully asynchronous systems, write after write hurts the processing continuity in some cases, such as in the 6600. It can be detected in a manner similar to that used for the read after write, but resolution differs. In the 6600, after its detection, instruction initiation pauses until the previous write is completed. So no additional hardware is needed. In the 360/91, write after write does not cause a pause, because the decentralized tags used in the CDB will automatically guarantee the precedence of the two writes. Thus, as a byproduct of the solution for read after write, this problem is also resolved.

3) *Write After Read*

As mentioned earlier, the write after read problem occurs only if the read and write queues are not synchronized. If they share the same queue, this problem does not arise.

1.5 Sequencing Control

Pipelines for arithmetic processing such as those used for multiplication and division are characterized by the following attributes: the speed of each segment is fixed; there are no additional buffers between segments; and the execution process requires internal looping, i.e. results being fed back as inputs. The basic sequencing control problem is to determine expedient moments at which to introduce new inputs from an external source so that there will be no collision (two computations attempt to use the same segment) and the throughput rate will be high. Davidson [16, 17] has developed an algorithm to sequence operands properly. A reservation table is used to represent the traversal path of operands through the pipe.

We illustrate this use of a reservation table by means of an example. Figure 6(b) shows the reservation table of a pipeline whose schematic is given in Figure 6(a).

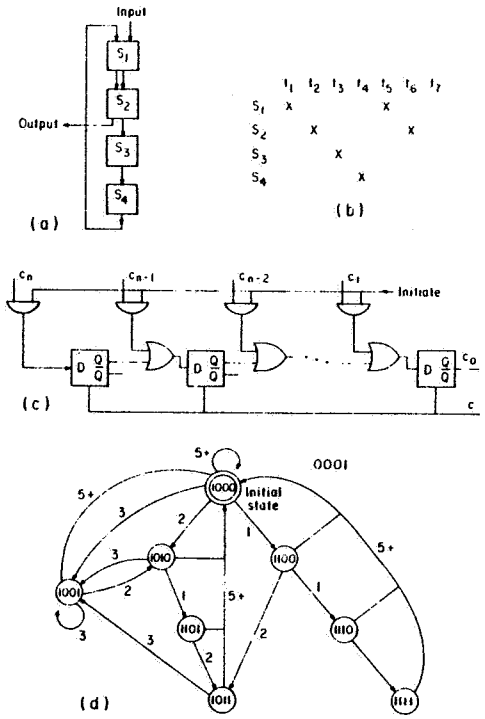

FIGURE 6a. A pipeline.
FIGURE 6b. Reservation table for Fig. 6a.
FIGURE 6c. Shift register controller.
FIGURE 6d. State diagram.

Each segment S_i requires one unit of time for processing. The computational sequence requires the passage of the operands in the order S_1, S_2, S_3, S_4, S_1, and S_2. In the reservation table, an X is placed at the intersection of a segment's row with columns corresponding to each time unit relative to initiation during which that segment is used by the computation. Consider the case in which a computation has just begun. To determine at which future times a new computation may be initiated without causing a collision, one has to analyze the reservation table. One way to determine whether two computations may be initiated K units of time apart is to superimpose the reservation on itself offset by K units of time. If an X falls on top of another, then a collision will occur in that segment, and K is a *forbidden latency*. Otherwise no collision occurs, and K is an *allowable latency*. Thus, if any pair of Xs in any row is K units of time apart, then K is a forbidden latency. Therefore it is simple to construct a *forbidden list*, which is a list of all forbidden latencies for the particular reservation table. From this forbidden list, in which n is the largest element, it is possible to construct the *collision vector*, which is a binary vector of n bits from 1 (rightmost bit) to n (the leftmost bit). Bit i is 1 if and only if it is an element of the forbidden list. Thus, if the collision vector is $C = c_n c_{n-1} \cdots c_2 c_1$, then $c_i = 1$ if i is an element of the forbidden list; otherwise $c_i = 0$. For the degenerate case of a linear (straight through) pipeline, there can be no collisions and the collision vector is empty, i.e., $n = 0$. The forbidden list for the pipeline in Figure 6(a) is (4, 4) and $n = 4$. The collision vector is 1000.

By the use of the collision vector, a simple control mechanism can be used to avoid collisions. Before initiating a new computation the collision vector can be checked to see if there are zeros in every location corresponding to the number of time units that have elapsed since each previous computation was initiated. Davidson devised an ingenious shift register controller ((Fig. 6(c)) for checking this requirement. The shift register controller is a sequential machine and therefore may be conveniently described by its state diagram in Fig. 6(d). It is assumed that computations are initiated only at collision-free opportunities, and only states that are important are represented in the state diagram. Each arc in the state diagram corresponds to the initiation of a computation and is labeled with the number of time units since the previous initiation. The initial node is coded by the collision vector itself and is the state of the shift register after the initiation of the first computation. Every node has an outbound arc for each 0 in the coding of the state and is labeled with the position subscript of its corresponding 0. An arc with label i leaving state S leads to state S', which is S shifted right i positions and OR'd with the collision vector. In addition there is an arc labeled $(n + 1)^+$ (where n is the dimension of the collision vector) leaving every node and leading to the initial node, indicating that, if more than n units of time elapse between the initiations, then the shift register will return to the state represented by the collision vector itself.

Cycles in the state diagram correspond to possible cycles of collision-free initiations in the pipeline. A cycle may be specified completely by the nodes passed through and the latencies of (or the time taken by) the arcs traversed from node to node in sequence. From the state diagram in Figure 6(d), it can be observed that there is a cycle consisting of states (1010), (1101), (1011), and (1001) with latencies of 1, 2, 3, and 2 time units, respectively. This cycle can be entered through the state (1000). At each of these states a new computation with a new set of operands can be initiated. Since four new computations can be initiated during each traversal of the cycle, there is an average latency of two, i.e., one result per two time units. One can find cycles that produce maximum throughput rates (minimum average latency cycles). In this example, the two cycles that produce minimum average latencies are (1000), (1100), (1110), (1111) and (1010), (1101), (1011), (1001), each having a latency of 2 time units.

In general, the problem of efficient sequencing control of a pipeline reduces to the discovery of minimum latency cycles in

the state diagram. In the case of more complex or multifunctional (assuming a certain instruction mix) pipelines, the discovery of the minimum latency cycles becomes quite difficult. Nevertheless, such a shift register control is applicable to properly avoiding any *resource* (facility) conflicts due to the existence of multiple paths or loops, in a completely synchronous sense.

1.6 Software Aspects

Language extensions: FORTRAN has been extended to support vector operations by the inclusion of special primitives such as vector addition and vector multiplication. Examples of such extensions are Lawrence Livermore Laboratory LRL-TRAN for the STAR-100 computer [33] and Texas Instruments ASC-FORTRAN [34] for their ASC computer. Compilers for these FORTRAN extensions perform some parallelism detection and cluster (group) like arithmetic operations and machine dependent code optimization in the object code. The NX compiler for ASC-FORTRAN possesses several of these facilities.

New languages are also being developed to support pipeline processing. SL/1 developed for STAR-100 at the NASA Langley Center is an example [35]. APL has also been tried to support vector operations on some pipeline computers.

Software costs. Software costs consist of three components, viz., 1) the cost of program generation and testing; 2) the cost of compiling; and 3) the cost of program execution. The cost of compiling consists of not only the cost of translation from a high level language to machine code, but also the cost of code optimization of the program for the particular machine. Obviously, efficient code optimization reduces the execution costs.

Machine dependent code optimization is a difficult problem. After the source program has been optimized, the resulting code is often hard to follow. Also because of several nonstandard I/O statement types, it is difficult to optimize I/O oriented codes. To illustrate the peculiarity and machine dependency of the code optimization problem, we provide a few examples.

a) Reducing the number of multifunction pipe reconfigurations by clustering like operations (ASC).

Unoptimized Code		*Optimized Code*	
K=A*B	requires	F=B/C	requires
F=B/C	four recon-	K=A*B	two recon-
L=D+E	figurations	P=F*C	figurations
P=F*C		L=D+E	
H=P+A		H=P+A	

Note that, in a multifunctional pipe, a reconfiguration is required to change its function, which, in turn, involves a time delay.

b) Special machine instructions. A FORTRAN program for polynomial evaluation is shown below:

```
      LIMIT=N+1
      DO 10 J=1,M
      VALUE(J)=X(J)*A(1)
      DO 10 I=2,LIMIT
10    VALUE(J)=VALUE(J) + A(I)*X(J)
```

This is equivalent to one machine instruction in the STAR-100. Therefore the compiler has to "recognize" the high level language statement sequence and replace it with the appropriate machine instruction.

c) Vectorization. Another type of optimization is to recognize sequential program statements that represent vector operations and translate them into powerful vector arithmetic instructions.

FORTRAN program:

```
      DO 10 I=4,100
       :
C(I)=A(I)+B(I-3)
       :
10 CONTINUE
```

Equivalent compiler generated text for pipeline machines:

```
VECT_BEGIN
A,C: VECTOR(4.. 100);
B: VECTOR(1.. 97);
C = A+B
VECT_END.
```

2. STRUCTURE OF A PIPELINED PROCESSOR

In this section, the basic structures of a pipelined processor are examined, with the IBM 360/91 central processor used as the example. The throughput objective of a sequential pipe are uncovered. From the

analysis of its structure, the problems and requirements specific to pipelined processors, outlined in Section 1, become more noticeable. They are discussed, and some solutions in existing processors are also illustrated and compared. Attention to vector processing capabilities is reserved for the next section.

2.1 An Example Sequential Pipelined Processor

To demonstrate the pipeline action in a sequential processor, the IBM 360/91 [9] is used as an illustration. The central processor was designed to upgrade computational performance (throughput) by one or two orders of magnitude compared to the 7090 system by means of pipelining and circuit design.

In order to observe the important problems and characteristics associated with a pipelined processor, the different segments in the pipe for a floating point instruction in the 360/91 are drawn in Figure 7. Basically most segments of the pipe have a cycle time of 60 nsec, with the exception of the storage referencing and execution unit. After decoding, two parallel sequences of operations are initiated. The first sequence includes the effective address calculation and fetch for the operand from memory storage. In calculating this address, the delay time in the segment(s) involved is variable, depending on whether it is indexed or not. The operand access segment again has a random delay, depending on the availability of the memory module to be referenced. The memory system in the 360/91 is interleaved to increase the bandwidth or memory supply rate. However, because of reference conflicts due to requests from other parts of the processor or system (such as instruction fetch or I/O), an operand fetch may have to be delayed for a complete memory cycle or more before it is acknowledged. This variable access time imposes a constraint on the efficiency of the pipelined processor. A completely synchronous operation on the segments may be impossible because of these variable waiting times. The need to be able to reduce the memory access time so as to match the speeds of the other segments in the pipes remains one of the most critical issues in pipelined processor designs. With slow effective memory access time, the memory access segment may be a bottleneck of such a large magnitude that the throughput of the processor is not much improved by pipelining.

The second sequence of operations involves the setting up of operands to be submitted to an assigned execution station in the execution unit. If it is a floating point instruction, it is mapped into a pseudo register-to-register (within the execution unit) instruction and transmitted to the execution unit. The execution unit then waits for the return of the operand from memory. When it returns, the two parallel sequences can merge (join) to initiate the next stage of processing, the actual execution.

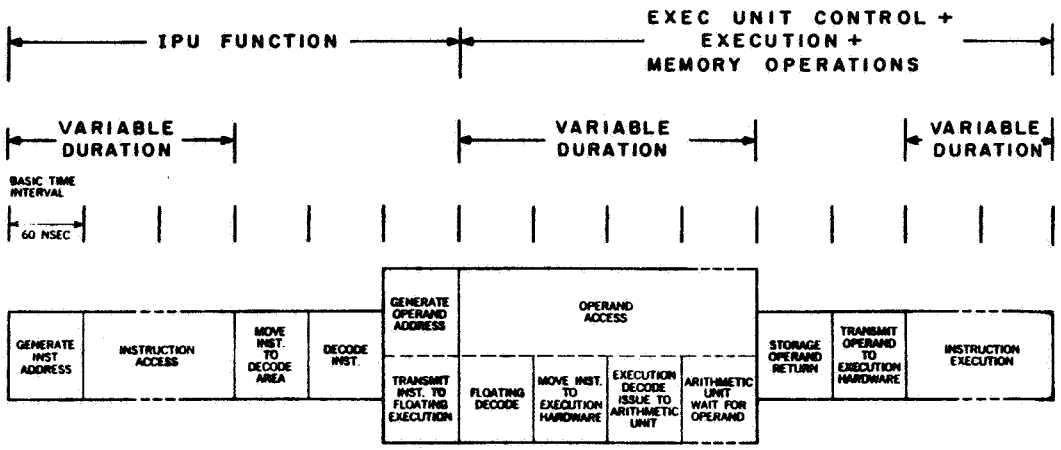

FIGURE 7. Functional segments involved in a floating storage-to-register instruction in Model 91.

The importance of reducing memory access time has been demonstrated. Even after the memory accessing problem has been solved, another bottleneck in the pipeline may emerge. This bottleneck is the execution unit. Usually many arithmetic operations, especially floating point operations, require considerable delay because of their implicit internal circuit delay requirement or iterative characteristics. If there is only one execution station to serve the entire instruction stream coming in, the speed of the execution unit may not be compatible with the input rate, thereby unnecessarily slowing down the computation. One alterative is to provide multiple physical execution stations to perform different types of operations. In the 360/91, there is a fixed point execution area and a floating point execution area. With this arrangement, floating and fixed point operations can be performed asynchronously but in parallel. But within each execution area, the multiplicity of execution stations can be increased. This is equivalent to increasing the throughput of the execution unit as an entity. For example, the floating point area in the 360/91 has two function units: a pipelined adder and a multiply/divide pipe.

We have shown the essential structures of a pipelined processor. Next attention will be paid to studying some design and operational problems associated with a typical pipeline. Included are the following topics:

1) Buffering: the concept and urgency of buffering in a pipeline and the ways it can be accomplished.
2) Busing structure: for communication between segments and operand supply to allow processing to proceed or resume as quickly as possible.
3) Branching: effect of branching in throughput and the ways to alleviate the inefficiency in existing systems.
4) Interrupt handling: how interrupts are handled in sequential and vector pipes.
5) Pipeline processing of arithmetic functions.

Taken together these five topic areas represent the major design constituents to be added to the basic structure already discussed. Their importance and effects actually can decide the efficiency and performance of the resulting design.

2.2 Buffering

Buffering is the process of storing results (outputs) of a segment temporarily before forwarding them to the next segment. It is essential in smoothing out the flow of a computation process when the timing for each processing module (segment) involved is not fixed. The impact of buffering can be visualized in a common assembly line, say in the car industry. Occasionally a station (segment) of the pipe (assembly line) may be slowed down for one of many reasons, which could prevent the continuous input of cars to the next station. If there is sufficient storage space between this segment and its predecessor, the latter can continue its operation on other cars and transfer them to the storage space until it is full. When the station resumes normal service it can try to clear up the cars in the input storage place, perhaps at a faster speed.

Therefore buffering may be needed before or after any segment whose processing speed is not fixed. In a pipelined processor this means 1) memory storage access related stations, including instruction fetch and operand fetch, and 2) execution unit stations. In a typical pipe like the 360/91, the instruction buffer can hold eight words of instructions to be followed in the sequence. In the execution unit, for the fixed point execution area, a buffer of six words of instructions (pseudo) and six words of operands is available, whereas in the floating point area a buffer of six instructions and six operands (from storage) is also provided. These buffers serve the purpose of continuing the supply of instructions or operands to the appropriate units whenever a variable speed occurs. Similar buffers in other pipelined processors can be found. In the STAR-100 system, whose configuration is shown in Figure 8, a 64 quarterword (superword) buffer exists in the stream unit to buffer the data and to align the two operand vectors (in vector processing mode) for streaming in the operations involved. In addition, there is of course the instruction buffer holding four swords of instruc-

tions (each sword is four 128-bit words). One sword in the instruction buffer will be filled by one memory fetch so that the buffer can supply a continuous stream of instructions to be executed even though memory conflicts may occur from time to time. Similarly, in the TI ASC system, whose schematic diagram is shown in Figure 9, sufficient buffers are installed in the IPU and Memory Buffer Unit (MBU). The MBU specifically holds eight-word X, Y, Z (two operands, one result) buffers to serve the arithmetic unit, and its instruction buffer consists of two eight-word fast register files. These examples are typical of the need and magnitude of buffering in a pipelined processor.

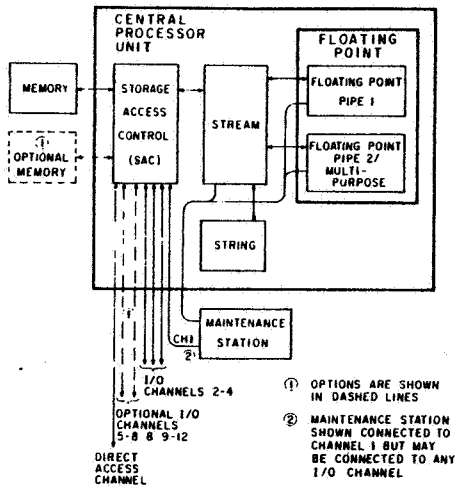

FIGURE 8. Basic CDC STAR-100 configuration.

2.3 Busing Structure

Pipelining requires the concurrent processing of independent instructions though they can be in consecutive stages of execution. With dependent instructions, as discussed in Section 1.4, their input and traversal through the pipe have to be paused until the dependency is resolved. Thus an efficient internal busing structure is needed to route the results to the requesting stations efficiently.

In the 360/91, the common data bus (CDB) was invented (Figure 10). The CDB can transfer data not only to the registers but also to the sink and source registers of all reservation stations (the virtual execution stations). It is fed by all units that can alter a register. To make this process possible, tags (addresses) are assigned to the registers. Then the processing sequence

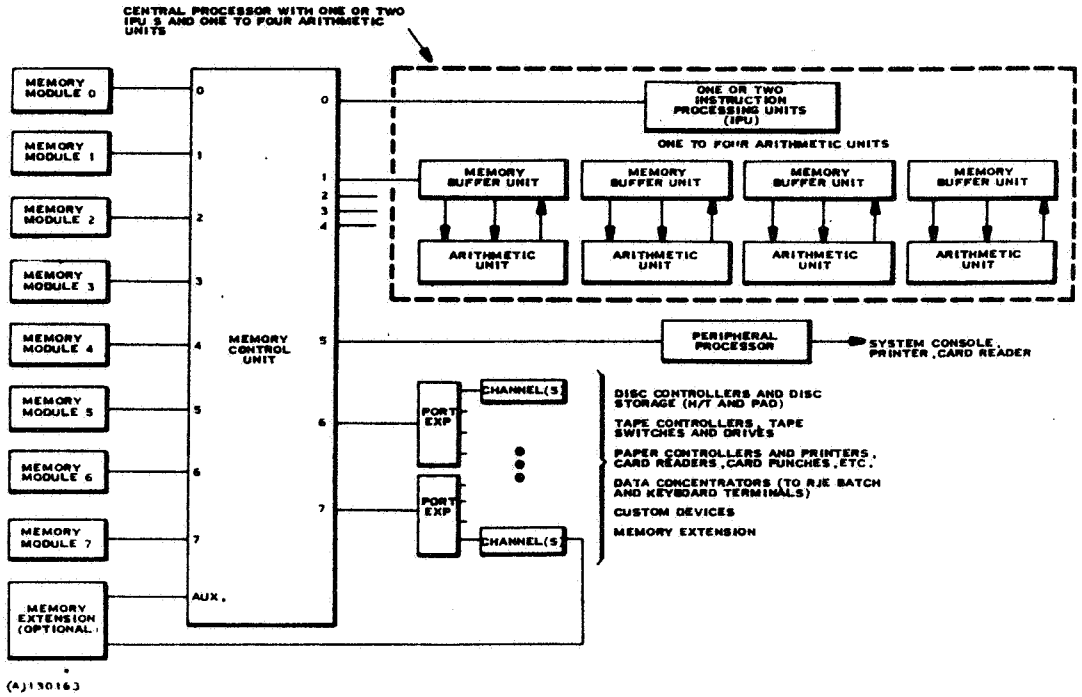

FIGURE 9. ASC system configuration.

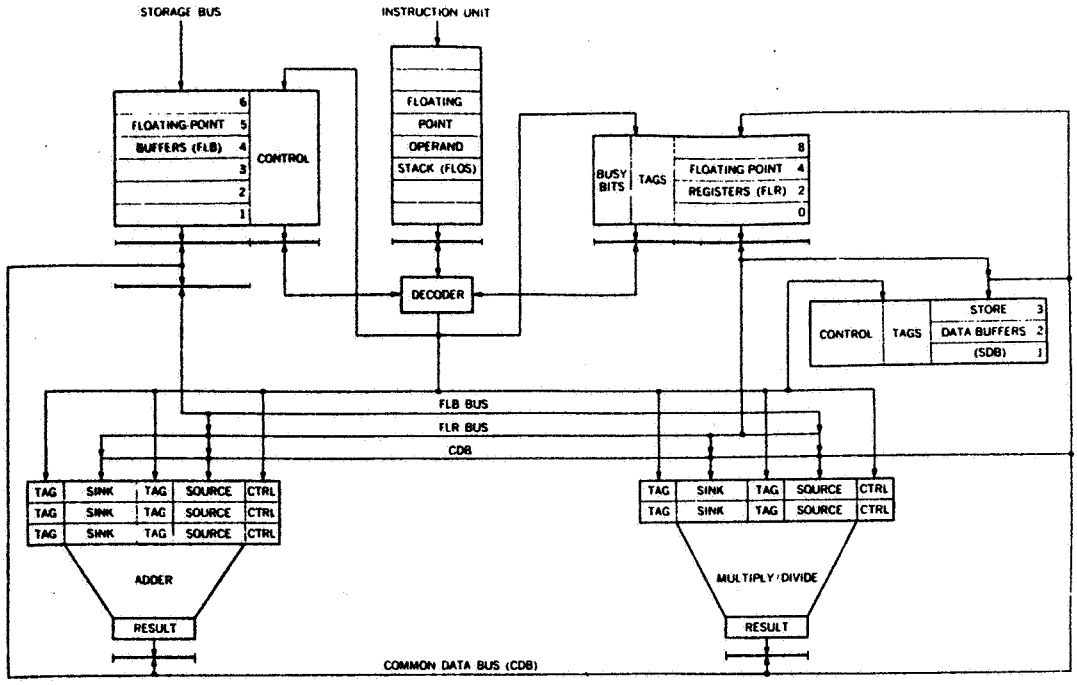

FIGURE 10. Floating point unit of IBM Model 91 with CDB and reservation stations.

can be described as follows. In decoding each instruction, the busy bit of each source register is checked. If it is zero, the independent instruction can be transmitted to a certain execution station, say A1 (virtual adder 1). At the same time, the busy bit of its sink register is set, and the corresponding tag is set to the destination of A1 (so that the sink register will receive the result from A1). If the busy bit is on, instead of waiting for the source operand to be generated and stored in the register, the dependent instruction is issued to an available execution station, say M1 (virtual multiplier 1). However, the tag of the register, rather than its content, is transmitted to the reservation station M1 so that M1 accepts data whose tag matches with its own from the CDB. As an illustration:

ADD F1,FLB1 [(F1) + (FLB1) → (F1)]
MD F1,FLB2 [(F1) × (FLB2) → (F1)]

In executing the ADD, A1 is used, and the tag of F1 is set to 1000 (that of A1) and its busy bit set to 1. In decoding the MD, the busy bit of F1 is 1. So rather than sending (F1) to M1, its tag (1000) is transmitted to M1. In addition, the tag of F1 is changed to 1010 (tag of M1). When the CDB is broadcasting the data tagged with 1000, M1 will succeed in matching the tag and so ingate it to the buffer and resume execution (if FLB2 is available). Notice that the result of ADD is not stored in F1 in reality because that operation is redundant (the tag of F1 is 1010 and not 1000).

A similar busing structure can be found in other pipelined processors such as the TI ASC and CDC STAR-100. In the TI ASC processor [13], an instruction dependency is recognized by hardware which scans the instruction stream and distributes the independent instructions across MBU-AU pairs to ensure proper, yet efficient execution sequences. Update capability is incorporated by allowing the contents of the Z-buffer to be transmitted to the *X*- or *Y*-buffer in the MBU when the latter two buffers are being used as scratch pads in local computation. In the STAR-100 system [14], a more explicit busing structure is maintained because of its different units. In the floating point pipes (whose configurations are drawn in Figure 11), a direct route

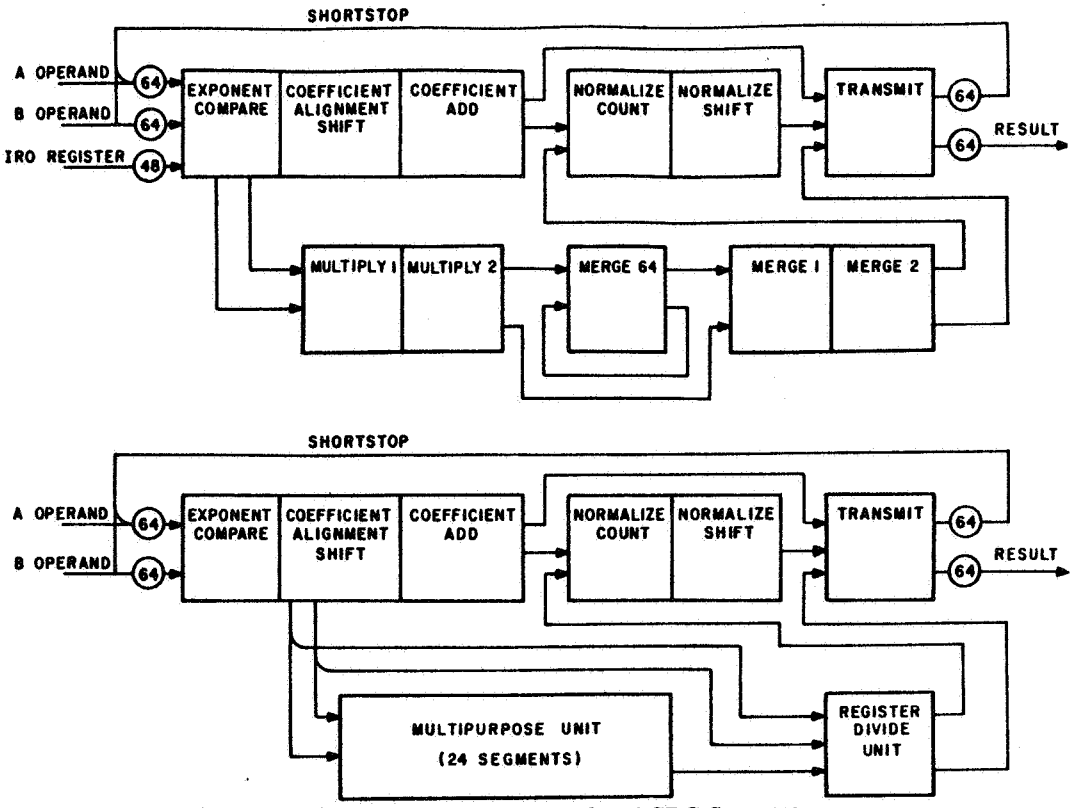

FIGURE 11. Floating point pipes 1 and 2 of CDC STAR-100 system.

called shortstop is established between the output (transmit segment) of each pipe and either of its inputs. This eliminates the time necessary to store the generated result in the register file and then to read it out again. These schemes fall into the control characterization in Section 1.4 very appropriately.

Although an efficient busing structure can reduce the adverse effect of instruction dependency, there is still a great burden on the programmers or the compilers to produce codes that expose sufficient parallelism to allow overlapped processing to become beneficial. If more independent instructions are intermixed appropriately with those dependent ones, more concurrent processing can take place while the dependency is resolved with little incurred time (that is the resolving of dependency is hidden behind other useful processing). This is a very important factor in deciding how efficiently a program or an implemented algorithm can be executed on a pipelined processor. Algorithm efficiency is also dependent on the architectural features of the processor on which it is executed.

2.4 Branching

Branching is more damaging to the pipeline performance than instruction dependency. When a conditional branch is encountered, one cannot tell which sequence of instructions will follow until the deciding result is available at the output. Therefore a conditional branch not only delays further execution but also affects the entire pipe starting from the instruction fetch segment. An incorrect branch of instructions and operands fetched may create a discontinuity of instruction supply.

To remedy the effect of branching, different techniques can be employed to provide mechanisms whereby processing can resume even if an unexpected branch occurs.

In the IBM 360/91 [9], a loop mode and back-eight test are designed with the help of an additional branch target buffer. In the ASC, a load lookahead [15] mechanism (instruction) is explicitly provided, with appropriate hardware and buffer support. Likewise, in the STAR-100 [14], the instruction stack has special branch back capability. We try to explain these schemes in this section.

The branch-on-condition handling is best illustrated by the 360/91. In this processor, upon the decoding of a conditional branch instruction, if the condition code is not yet valid it is assumed that no branch will be taken. However, to guard against an incorrect guess, two instruction doublewords will be fetched from the branch and stored at the branch target buffer. The conditional mode is entered where instructions are forwarded conditionally to later segments for processing. Operands are conditionally set up while actual execution is prohibited. Finally, when the branch should be taken, the conditional instructions are deactivated and processing is resumed using the branch target instructions; otherwise execution continues almost instantaneously. This procedure therefore reduces the waiting time in the average case. To further reduce instruction fetching time, short loops in programs can be fruitfully exploited.

If the instructions are already in the instruction buffer, it is wise not to erase any of them and to assume the branch (repeat loop) will be successful. Then no other memory access for instructions is needed and less memory interference with other parts of the processor will be created. The way to detect these short loops and reserve the instruction loop is by implementing a loop mode and back-eight test.

A sequence of eight instruction doublewords or less is termed a short loop and can be completely stored in the instruction buffer. When a branch (backward) is obtained, the back-eight test is used. If it is satisfied, the loop mode is established. From that point on, the complete loop is fetched into the instruction buffer so that no further fetching is needed until the loop mode is removed by branching out. In conditional branches, the loop mode can be established to replace condition mode once a successful branch results and the back-eight test is satisfied. This method of back-eight test and loop mode is very useful in systems where available memory cycles are precious to the entire system. However, if the memory (cache) access time is not long, the conditional handling may be less useful due to its overhead.

The load lookahead mechanism in the ASC system follows a similar philosophy. The instruction processing unit of the machine contains two instruction address registers (Present Address, PA and Lookahead Address, LA) and two instruction files of eight words each (KA and KB). Each memory reference can fetch an octet (P) of instructions to one of the instruction files. Usually PA contains the starting address of the next octet to be fetched and LA supplies the address of the next octet to be fetched. To accommodate branching for a loop, a branch with lookahead can be set up by placing the branch instruction at the target location of a Load Look-Ahead (LLA) instruction. An LLA enters a count into a Lookahead Count register (LC) and enters the address of the LLA into a branch address register. The count corresponds to the difference of the instruction locations of the LLA and its target branch instruction. The count is decremented by one every time an instruction is executed following the initiation of the LLA. When it has reached a value designating that the branch has already been requested from memory, the control transmits the contents of the PA to the LA. This causes the fetching of the octet containing the LLA and the loop control is reinitialized. In this way, a lookahead loading of instructions in a loop up to 256 instructions is allowed, and instructions will be continuously available for execution before the branch instruction is completed.

The STAR-100 processor has an instruction stack of sixteen 128-bit words. Each quarterword (i.e. four words) is loaded in one minor cycle. Branching is allowed within the instruction stack. The loading and management can be as depicted in Figure

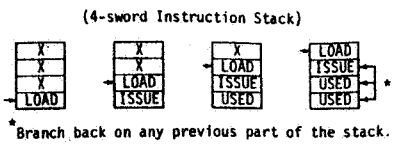

FIGURE 12a. STAR-100 instruction stack loading and issuing with branch tolerance.

12(a). After the stack is loaded any branch within the stack can be honored easily; however, the stack is cleared whenever a branch out of the stack occurs. The reason for this is that the stack can be completely filled by a request to memory (i.e. in one memory cycle).

These methods are useful to help to supply instructions continually to the pipe segments even though branch instructions are inevitable. For fixed (targeted) branches, lookahead strategies can provide the means to continue the instruction sequence. But for conditional branches more elaborate schemes to recover from unexpected branches have to be established (such as the conditional mode).

2.5 Interrupt Handling

Interrupts disrupt the continuity of the instruction stream in a pipeline much as the conditional branches. When an interrupt occurs while instruction i is being executed, the interrupt should be serviced before any action is applied to instruction $i + 1$. This implies that either these two instructions are to be executed sequentially or sufficient information is set aside for the eventual recovery of instruction $i + 1$. The first course defeats the purpose of pipelining. The second approach is taken by some architectural designs when the cost of recovery is not overly substantial.

During vector processing, execution of a vector instruction may take a long time. Therefore, as in the STAR-100 processor, special interrupt counters are available to hold addresses, delimiters, field lengths, etc., which are necessary to restart the vector-type instructions after an interrupt. This represents a recovery mechanism for processing to proceed afterwards when an unpredictable interrupt occurs.

In a more general purpose pipeline, however, many independent instructions can be at various stages of completion in the pipe at the same time. To recover these instructions after the interrupt imposes a complex and costly problem. In the IBM 360/91 two types of interrupts, namely "precise interrupts" and "imprecise interrupts," are used:

1) Precise interrupts are associated with an instruction (like an illegal operation code) and can be uncovered during the decoding stage. This type of interrupt can be treated in the normal fashion. Since decoding is the first stage of the pipe, when an interrupt on instruction i is uncovered, instruction $i + 1$ will be prohibited from entering the pipe; however, instructions which precede instruction i and are uncompleted in the pipe continue to be executed. After all execution activities are completed in the pipe, the processing unit is switched to execute the interrupt routine.

2) Other interrupts which result from storage, address, and execution functions are termed "imprecise." These interrupts usually occur when the instruction is halfway through the pipe and subsequent instructions are already admitted into the pipe. Strict adherence to the normal interrupt processing is therefore difficult. When an interrupt of this kind is encountered, further decoding is prohibited (i.e., no more new instructions are allowed to enter the pipe). But instructions uncompleted inside the pipe, whether they precede or follow the instruction, are completed before the processing unit is switched to service the interrupt.

In both cases the new status word for the interrupt branch is fetched to the branch target buffer while the pipe is being "emptied." Further optimization is possible by starting the fetching of interrupt instructions if it takes a long time to clear the pipe. This imprecise condition due to error interrupts is a disadvantage of overlapped processing when program debugging is considered.

2.6 Pipeline Processing of Arithmetic Operations

One of the most fruitful applications of overlapped processing to improve through-

put has been in the execution of arithmetic operations. In vector arithmetic, for example, the same sequence of operations are executed repetitively, a circumstance most congenial to pipeline implementation.

In the IBM 360/91 and its successors the execution of multiplication and division is pipelined [21]. Algorithms suitable for pipeline execution of binary addition, multiplication, division, and square root have been discussed [22].

A close study of a typical low level pipeline for performing binary multiplication is now presented.

The most common method of multiplication is the pencil and paper algorithm in which the multiplicand is shifted and, if the corresponding bit in the multiplier is 1, added to the partial sum until the multiplier is exhausted. Clearly this is not an effective pipeline algorithm because too many shifting and adding operations (complete additions) are needed. Even if the 0s in the multiplier are skipped, the speed of the multiplier is too slow to match the speed of the other parts of the system. One could try to build a very fast multiplier using Wallace Trees [20] of Carry-Save Adders (CSA). But such an implementation requires too much hardware. Obviously a speed/cost trade-off exists here. The method favored in the IBM 360/91 and other computers is a hybrid method, in which multiples of the multiplicand (summands) corresponding to a group of multiplier bits (generally two or three) are generated iteratively and accumulated by CSAs. During the last iteration, the summand of the last group of multiplier bits and the previously accumulated partial sum are added by using a Full Binary Adder (FBA). Our example system will generate the summands corresponding to each 4-bit group of the multiplier in real time and will use CSAs to accumulate several partial sums before generating the final product. Figure 12(b) shows the flow during the process of multiplication.

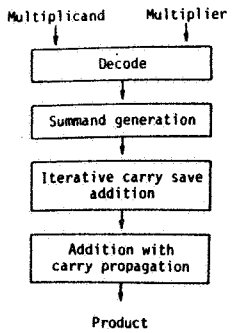

FIGURE 12b. Functions in a multiply pipe.

TABLE 1. MULTIPLIER DECODING

Multiplier Bits	Operation
0 0 0 0	0
0 0 0 1	D
0 0 1 0	2D
0 0 1 1	4D-D, 2D+D
0 1 0 0	4D
0 1 0 1	4D+D
0 1 1 0	8D-2D, 4D+2D
0 1 1 1	8D-D
1 0 0 0	8D
1 0 0 1	8D+D
1 0 1 0	8D+2D
1 0 1 1	16D-4D-D
1 1 0 0	16D-4D
1 1 0 1	16D-4D+D
1 1 1 0	16D-2D
1 1 1 1	16D-D

Decode Phase

The multiplier bits are examined four bits at a time starting with the least significant 4-bit group. The four multiplier bits are expressed into the sum of at most three numbers which are powers of two times the multiplicand. In other words, each 4-bit group of the multiplier is decoded into a maximum of three binary numbers which are powers of two times the multiplicand. For example, if the multiplier bits are 1101, then the decoder generates three numbers, $16D$, $-4D$, and $+D$, where D is the multiplicand which when summed generate the multiple 13 times the multiplicand. Table I provides the decoding table for four bit multiplier summand generation. Note that the decoding process generates at most three numbers using combinatorial logic (in real time) and provides the three inputs needed for a carry save adder.

Generation of Summands

The decoder generates the appropriate multiples of the multiplicand corresponding

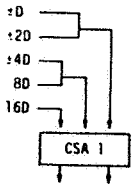

FIGURE 12c. Generation of summands.

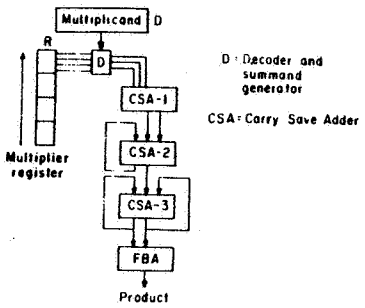

FIGURE 12d. Pipeline multiplication.

to a 4-bit group of the multiplier. The inputs to the first Carry Save Adder (CSA-1) are shown in Figure 12(c).

Operations of the Multiplication Unit

Assume now that we wish to multiply two 16-bit positive numbers. The Carry Save Adders and the Full Binary (carry propagate) Adder are assumed to be 2-word, or 32 bits wide. Initially the adder arrays are cleared.

During each iteration, a 4-bit group of the multiplier in the R register is decoded and the three inputs to CSA-1 are generated. CSA-1 uses the inputs to generate the two outputs (Partial Sum (PS) and Carries Saved (CS)). These are passed on to the next Carry Save Adder, CSA-2. The carry save output of CSA-2 is fed back as an input to itself during the next CSA-2 operation. The PS output of CSA-2 is introduced as an input to CSA-3. Both outputs of CSA-3 are fed back as inputs to itself.

The multiplier bits are decoded four bits at a time starting with the least significant ones. After the CSA-1 receives its inputs, the R register is shifted right four bits, and the decoding of the next group of four bits is initiated. This sequence is continued until the final group of four bits is decoded. As we supply operands to CSA-1, these operands are accumulated.

After supplying the final set of operands (corresponding to the four most significant bits of the multiplier), we have four sets of accumulated operands in the system. Now in the next three cycles (each cycle corresponds to one operation of a CSA), these operands will be accumulated into two operands at the output of CSA-3. Finally these operands are channeled to the FBA to obtain the final product. A timing diagram (reservation table) is provided in Figure 12(e) to elucidate the overlapped operations in the system.

Performance Analysis

Let N be the number of bits in the multiplier, and let t_c be the delay through a CSA. Since the latter can be realized by two levels of combinational logic, t_c will be equivalent to two logic gate delays. The delay through the full binary adder, t_{FBA}, will vary with the size of the operands and its design. Then the total time for multiplication of N bits (from the time the inputs are introduced at the first carry save adder) is

$$t(\text{multiply}) = [N/4]t_c + 4t_c + t_{\text{FBA}}.$$

If t_c is equal to two gate delays of 20 nsec each and t_{FBA} for 32-bit operands using carry look ahead logic is around 70 nsec, then the total multiply time with 16-bit operands to generate a 32-bit product is 270 nsec.

Extensions

The previous procedure using 4-bit multiplier groups can be extended to 8-bit multiplier groups, thereby almost doubling the

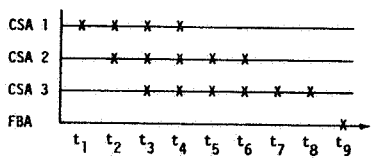

FIGURE 12e. Timing diagram and reservation table for pipeline multiplication.

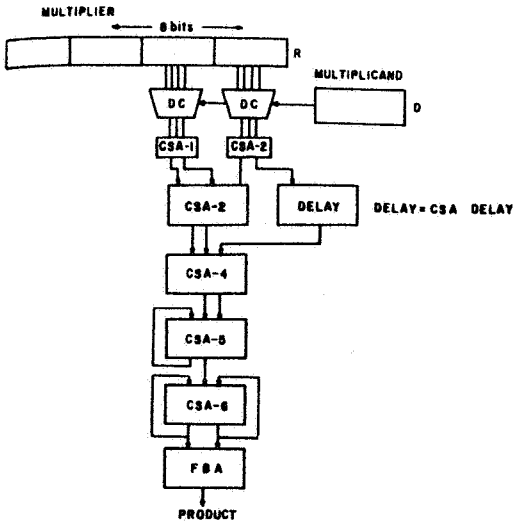

FIGURE 12f. Pipeline multiplication using 8-bit multiplier groups.

throughput rate of the system. Figure 12(f) illustrates the organization. The scheme utilizes two 4-bit decoders, which generate appropriate summands corresponding to two consecutive 4-bit multiplier groups at CSA-1 and CSA-2. The two sets of summands are combined at CSA-3 and CSA-4. CSA-5 and CSA-6 accumulate the summands received from each 8-bit group until all the groups in the multiplier have been processed. In a similar fashion as in the 4-bit group scheme, the full binary addition is performed at the last step.

Several interesting and challenging problems still remain open for investigation. Pipelining of decimal arithmetic functions, radix conversions, and polynomial function evaluation are some of the many useful applications. Also the study of multifunctional pipes with respect to arithmetic expressions deserves attention.

3. VECTOR PROCESSING

One of the main requirements in justifying the pipelining of a process is that the same sequence of operations will be invoked very frequently. Ideally, if a continuous excitation of the pipeline is attained, then the maximum throughput will be reached. For a pipelined processor, this is equivalent to the need for abundant parallelism in the instruction streams to permit almost continuous initiation of independent instructions.

This ideal situation sometimes becomes true when the machine is processing independent vectors, e.g. adding two vectors, element by element, to form a result vector. If each element of a vector has to go through a transformation independent of the transformation of other elements of the vector, then they can be performed in an overlapped mode employing the pipelining characteristics. For machines with multifunctional pipelined execution units, the execution units can establish and retain a static configuration throughout until the entire vector is processed. Hence minimal control, decoding, and reconfiguration overheads may be achieved while the memory operands are supplied to the execution unit in a most efficient way. This will become more apparent as our discussion proceeds.

In this section vector processing in pipelined processors is studied carefully. In Section 3.1, the components of a vector instruction and the ultimate processing procedures are demonstrated and a comparison of two prominent vector machines in this aspect is included. This comparison leads to the revelation and evaluation of the requirements, properties, and tradeoffs in terms of time and space (control hardware) overhead in vector processing as contrasted with sequential pipeline processing. The analysis in Section 3.2 serves to expose the real crux of vector processing.

3.1 Vector Instruction

A vector pipe can be characterized by the existence of one or more multifunctional pipes in the execution unit (arithmetic and logic unit) and the needed control and parameter specifiers in the processor. As mentioned in Section 1, a multifunctional pipe can be either static or dynamic, depending on its reconfiguration control. In the static case, simpler control is required to establish and maintain a desired configuration for processing. There is a fixed route for each operand set to traverse throughout the computation, unless a new

configuration is formed. While in the dynamic case more complicated control and routing overhead is involved, the throughput may be higher because of the simultaneous existence of several configurations. In reality, static vector pipes are more common, as is illustrated in the TI ASC and CDC STAR-100 examples to follow.

For a vector that consists of the two levels of pipeline action, appropriate vector instructions have to be designed and implemented to denote the operations on some ordered data in vector or array form. Generally, in the first level, a vector instruction is fetched, decoded, and the necessary control paths connected before the needed elements of the vector are fetched from consecutive storage locations over a specified address range. The second level execution unit pipe carries out the specified operations on these elements, normally being supervised by a control ROM. Sometimes the results generated are stored back to certain consecutive addresses of a result field, and sometimes other needed indicators are generated and stored in the register file in the processor for future usage. The exact procedures and mechanisms to accomplish all these functions vary from machine to machine. For later comparison and analysis, an example of vector instruction execution is provided here.

Before the execution of a vector instruction starts, certain additional information pertinent to the mode of processing has to be furnished to the system. Such information can be quite varied and detailed, such as the starting (base) address of each source vector and result vector involved (usually two source vectors and one result vector) and the control over what elements of the vectors should be operated upon. The method by which the STAR-100 handles this is demonstrated first. The similarity with the control of an array processing system can be observed. Then similar and different features in the ASC system are noted. Finally the vector processing powers of the two systems are compared.

The schematic diagram of the central processing unit for the STAR-100 system is shown in Figure 8. Basically it consists of four parts: 1) Storage Access Control (SAC), 2) stream, 3) string, and 4) floating point units operating in an overlapped, asynchronous mode. The SAC is responsible for sharing the magnetic core storage among the three read and three write buses shared by the stream and I/O units. The stream unit provides the basic control for the entire processor. Internally it may be regarded as a multisegment pipeline (second level) as it carries out functions which include: 1) memory references; 2) buffering and skewing of operand data; 3) buffering and decoding instructions; 4) setting up control signals for processing the instruction; and 5) performing simple logical and arithmetic operations.

The string unit, as the name implies, is used to process strings of decimal or binary digits. It contains fast half adders and full adders to carry out algorithms for binary arithmetic (add, subtract, divide, and multiply). Finally, the floating point unit consists of two pipes whose configurations are shown in Figure 11. Each pipe is (static) multifunctional as it has different configurations for performing different floating point operations. Pipe 1 performs arithmetic operations on operands in floating point format and address operations on nonfloating point numbers. Pipe 2 performs only two vector address type operations, in addition to other arithmetic operations. Pipe 1 and pipe 2 are quite similar in structure except that the latter has a high speed register divide unit and a multipurpose unit for some special arithmetic such as square root, vector divide, etc. The pipes can take on a certain configuration at any time. For example, to perform floating point addition, pipe 1 configures itself (under microcode control, to be explained later) to activate the path: Expo-

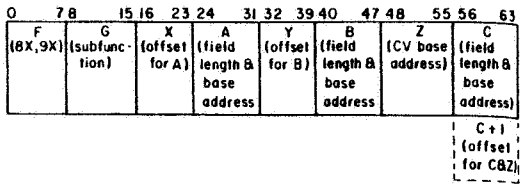

FIGURE 13. Vector instruction format in CDC STAR-100.

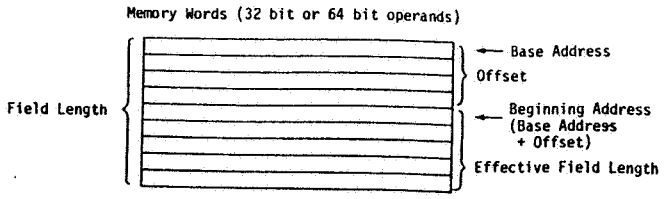

FIGURE 14. Addressing offset for vectors.

nent Compare—Coefficient Align—Coefficient Add—Normalize Count—Normalize Shift—Transmit. With this static configuration, operand pairs can be routed through the pipe at a steady and maximum rate. When the operand pairs can be supplied fast enough and the result stored suitably, an ideal throughput rate will be reached.

Let us now pause to examine a vector instruction before exploring the procedure of its execution. An ordinary vector instruction format in the STAR-100 computer is representable by eight fields as indicated in Figure 13: 1) *F:* function code; 2) *G:* subfunction code; 3) *X*, *Y* specify the registers that hold address offsets for the two corresponding source vectors (the offset operates as depicted in Figure 13 and is useful for skewed vectors); 4) *A*, *B* specify the registers that hold the base addresses *and* field lengths of the two source vectors; 5) Z specifies the register holding the base address of the control vector; 6) *C* specifies the register holding the base address and field length of the result vector; and 7) $C + 1$ then automatically specifies the register holding the offset for the control and result vectors. This automatic assignment is implied to maximize the utilization of each instruction word which has a limited length.

From these registers, the effective starting address and field length of each vector can be calculated. Then the rest of the vector can be referenced sequentially until a termination condition is reached. The control vector is a unique feature introducing the flexibility desired in vector processing. It performs prohibition responsibility, analogous to the control unit in an array processor such as the ILLIAC IV [2]. The control vector in the STAR-100 performs the analogous function, but in a time stretched fashion (compared to the simultaneous inhibition of array elements). Each bit in the control vector is used to specify whether or not the corresponding result element should be stored (for most vector instructions; in some modified cases like macros, it has other duties, as will be explained later). When a bit is set in the control vector, the corresponding element of the result vector will not be modified and stored. Thus the nth bit read from the control vector will be used to control the storing of the nth element generated in processing the vector instruction.

As an illustration consider a vector add instruction:

$$\text{VADD } A, B, C \quad (A + B \rightarrow C)$$

Suppose the instruction format provides the following information:

(A) = content of A register:
field length of A vector = 12 halfwords (32 bits each)
(B) = field length of B vector = 4 halfwords
base address = 20000_{16}
(X) = offset for A vector = 4 halfwords
(Y) = offset for B vector = -4 halfwords
(Z) = base address of control vector = 40004_{16}
(C) = base address of result vector = 30000_{16}
field length = 12 halfwords
$(C+1)$ = control vector and result vector offset = 4 halfwords.

Then the starting address and effective field length of A vector can be calculated as shown in Figure 15.

Note that the addressing used is bit address and a '1' in the control vector permits the storing of the corresponding element in the resulting vector. For example,

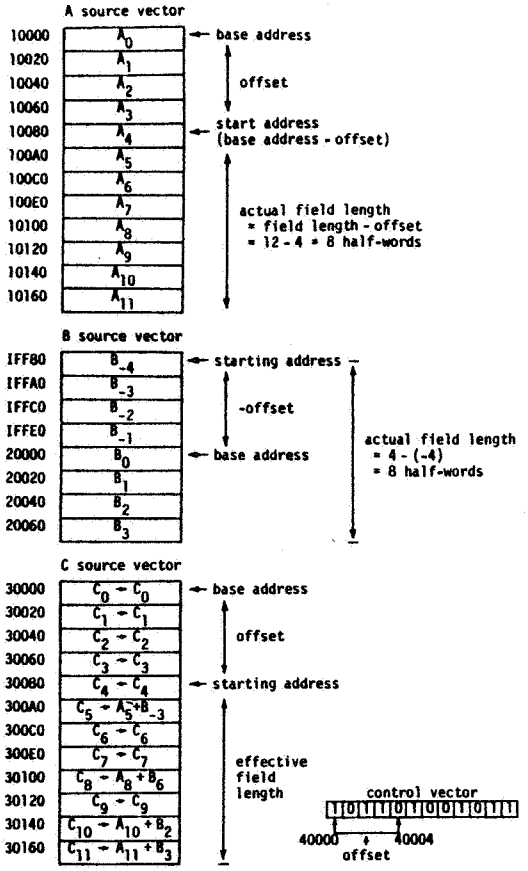

FIGURE 15. Example vector ADD.

40005 stores a '1'; so C_5 is transformed into $A_5 + B_{-3}$. The skewing effect is quite apparent in this example.

The mechanism to generate the desired output has to be explained further. After the instruction has been decoded at the stream unit, the appropriate microcode sequence in the Microcode Unit (MIC) is initiated. This microcode unit resides in the stream unit and is responsible for vector type operations.

When the CPU initiates an instruction requiring microcode control, it sends the F (function) code and a microcode pulse to the MIC. The MIC then takes over control of the start up and termination of the instruction. In the case of interrupts, it also has to branch to save all the operands and parameters necessary to resume execution afterwards. Therefore it is the heart of the vector processing control. In fact, it is the central control once a vector-type instruction has been noticed via decoding. Typically it controls operations including:

1) the reading of addresses from the register file (in the stream unit) for the vector parameters according to the designations specified in the instruction;
2) the calculation of the effective addresses, field lengths, etc. for monitoring the starting of the operations involved in the vector instruction;
3) the setting up of the usage of read/write buses as specified by the G (subfunction) field for the operands and results; and
4) the transfer of addresses and other information to appropriate interrupt count registers whenever needed.

Once the effective addresses are computed, the operand elements are fetched and paired for the operations involved, for example, going through the second level floating point pipe. The static configuration of the execution pipe will remain active until the vector instruction is terminated. A termination is marked by either of the following events:

1) A vector is exhausted (e.g., when the effective field length is zero, or the difference between the effective field length and the number of operand pairs encountered thus far is zero); and
2) Some other data fields or strings have been exhausted.

From the above description, one can see what a vector pipe really includes and how vectors can be processed in an overlapped manner. It is interesting to find some other ways to achieve a vector pipe. So let us examine a similar vector machine, the ASC system. The ASC handles a vector instruction in a similar way, though some additional distinguishing features should be mentioned. To facilitate understanding, the central processor unit composition in the ASC has to be briefly explained. Its schematic diagram is provided in Figure 9. It consists of three main components: 1) Instruction Processing Unit (IPU); 2) Memory Buffer Unit (MBU); and 3) Arithmetic

Unit (AU). The IPU is analogous to the stream unit in the STAR-100; the MBU is analogous to the load/store; and the AU actually processes the data. In vector mode the IPU fetches and decodes the instruction and calculates the effective addresses for the vector fields. After receiving the needed information from the IPU, the MBU starts fetching source operands and pairing those to be sent into an AU pipe (the AU can have one to four identical pipes). Each AU pipe has different configurations for performing different arithmetic operations (including integers) as in a typical static multifunctional pipeline. The two levels of pipeline action are quite apparent in this case.

A vector instruction in the ASC has some outstanding characteristics; the instruction format is depicted in Figure 16. Particular registers for fetching operand address and control information do not have to be specified, however. Some registers in the IPU, forming the Vector Parameter File (VPF), are dedicated to vector processing. The VPF consists of eight 32-bit registers whose individual functions or interpretations have been permanently assigned, as shown in Figure 17. This fixed organization has the advantage that registers can be hardwired to the input of the control ROM or other logic units for fast operation, without having to worry about access conflicts among them. The first register contains the operation code and the type and length of the vector considered (single or two-dimensional). Then the base address and the register containing the index (offset) are specified for each operand vector in the subsequent register in the VPF. The fifth and sixth registers are used to specify the increment for each vector and the number of iterations (field length) in this inner loop. For the outer-loop (two-dimensional vectors), similar information about the increments and number of iterations is included in registers seven and eight. The vector instruction, after having been decoded, will provide the information regarding whether the parameter file has to be loaded from main memory or retain some previous setting for immediate usage. If a load is needed, since the memory is interleaved, one memory cycle is needed for VPF loading. The significance of this and the subsequent additional activities is examined more carefully in the next subsection. Afterwards, the sequence control in the MBU takes over (as does the MIC in STAR-100) the fetching of operands and the routing of operand pairs through the AU pipe.

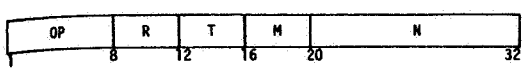

FIGURE 16. Vector instruction format in TI ASC.

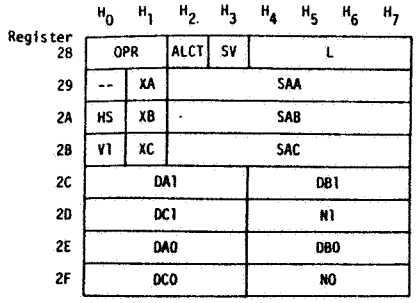

FIGURE 17. Vector parameter file format in TI ASC.

So the ASC has at least two distinguishing features in vector processing: 1) its dedicated use of the vector parameter file; 2) the interpretation and usage of the VPF, allowing variable increments within the different vectors concerned (contrary to the sequential mode in STAR-100), and two-dimensional vectors to be explicitly handled (inner and outer loops).

These features help to execute some vectors more efficiently and reduce the overhead that may have been incurred. Observe that once a vector instruction is initiated, the operand pairs are submitted to the AU continuously—in most cases, once per minor cycle (provided no severe memory interference results from other pipes or parts of the system or processor). Then the maximum throughput rate may be achieved (1 result per minor cycle is equal to 60 nsec.). Also the sequence control for the AU is handled exclusively by the microcode stored in the ROM in the MBU. Therefore the MBU serves as the

unique interface between the IPU and the AU.

From the previous discussions, one can visualize the concept of vector processing and the two ways to achieve high throughput in two similar machines. To bring out more interesting special features in these machines, the vector-type instruction set in the STAR-100 is examined once again. From it a final brief comparison of the two systems, the STAR-100 and the ASC in this respect, is derived.

Generally speaking the STAR-100 has a richer and more powerful vector instruction set. Two outstanding features are: 1) vector macros instructions and 2) sparse vector instructions.

In vector macro instructions, operations are performed on the source vectors except that, in some cases, no result vector is created. Instead, the result is represented and stored in one or two registers as specified by the instruction.

For example, SELECT GE A $\geq$ B, ITEM COUNT TO (C) involves comparing each element of vector field A with the corresponding one in B. The comparison terminates if the condition $A_i \geq B_i$ is met for the current i, or one of the vector fields is exhausted. Then the number of operand pairs encountered thus far is stored in the register specified by C.

In this macro operation, control vectors can be used not only to prohibit the storage of result elements but also to disable the operation on some elements. In the example, even if $A_i \geq B_i$ is true for some i, if that comparison is disabled by the corresponding element in the control vector, execution will not be terminated. Thus, by using this kind of instruction, comparison of ordered vectors (e.g. lexicographic comparison) can be easily handled. The item count will be useful in some cases to indicate at which element the condition is satisfied. On the other hand, ordinary vector compare instructions also exist in the STAR-100 machine. For example, COMPARE GE A $\geq$ B, ORDER VECTOR $\rightarrow$ Z involves 1) comparing the two vectors element by element and 2) storing 1 or 0 at the result vector elements depending on the satisfaction of the comparison condition.

The result of each pairwise comparison is recorded and is available for later use, such as in sorting. Thus ordinary vector and vector macro instructions may form a powerful vector instruction set to be tailored to suit some application as closely as possible. With them, many quite complex sequential algorithms may turn out to be very effective, as is studied later.

The sparse vector instructions in the STAR-100 system further facilitate processing of large vectors with a lot of zero elements because then the vector can be packed easily into a sparse vector to be operated upon later. This packing can save both memory storage space and later effective processing time. A sparse vector can be formed by using the following procedure, as illustrated in Figure 18.

Step 1: Generate an order vector by using a COMPARE instruction to indicate zero elements.

Step 2: Compress the vector into a sparse vector by storing the chosen elements from the former to memory, according to the order vector generated at Step 1. The order vector has to be retained throughout the lifetime of the sparse

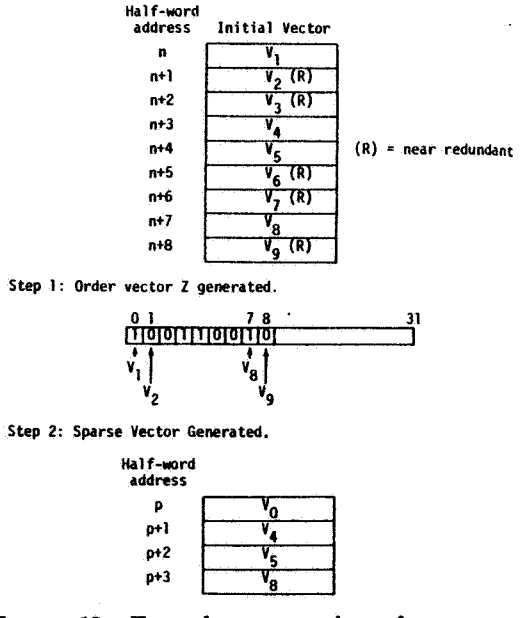

FIGURE 18. Example compression of a vector into a sparse vector field.

vector to specify the positional significance of its elements.

Now the sparse vector can be efficiently operated upon to generate desirable, interpretable results as in other vector instructions, with the help of the order vector. The advantages of sparse vectors should be emphasized:

1) The explicit hardware support for compaction of large vectors reduces memory space needed.

2) If the sparse vector has to go through several operations or computation steps, effective processing time can be saved as well in that the operation on zero elements is no longer necessary.

3) If a variable increment for each vector (as in the ASC) is desired, one way to implement it is to use sparse vector instructions (though a more obvious way is to include the appropriate control vector) for the purpose of saving space and time.

While the ASC does not include sparse vector instructions, its explicit two-dimensional vectors and variable vector increments are good features which promise high vector processing capability. Included in the vector instruction set of both machines are some very interesting and high level instructions, such as vector search, dot product, merge, shift, and order, that allow programmers more power in developing their programs and the system to execute the algorithms implemented with the help of these advanced instructions more efficiently. The ASC has also demonstrated how a 32-bit machine can cope with vector processing by efficiently making use of 8-bit opcode and the other relative fields, together with a dedicated vector parameter file. While the STAR-100 shows a stronger vector instruction set (a vector instruction is composed of 64 bits) because the F (function) and G (subfunction) codes can be used to specify more things, the vector parameters to be used can be assigned to any one of the registers (therefore not dedicated). It is hard to say which scheme is absolutely superior. To summarize, the comparison of the vector processing powers of the ASC and STAR-100 is tabulated in Table 2.

3.2 Implications, Requirements, and Tradeoffs

How vectors are processed has been demonstrated in the previous section. Now a closer look at some hidden or less con-

TABLE 2. COMPARE AND CONTRAST

STAR-100	TI ASC
Vector parameter registers to be specified.	Vector parameter file fixed, therefore easy to reference and store.
Very strong vector instruction set.	Strong vector instruction set.
Sparse vector instruction included.	Sparse vector not included.
Vector increment is fixed.	Variable vector increment allowed.
Control vector introduces flexibility similar to the control unit in array processors. Can be used to implement variable vector increment.	No control vector used.
Explicitly, vectors are only one-dimensional.	Two-dimensional vector explicitly accommodated. Computes 2 level loops effectively.
Use microcode control once a vector instruction is decoded.	Use microcode control to sequence each AU.
String unit and Floating Point unit (2 nonidentical pipes) will be responsible for most of the actual proecessing of data. Therefore concurrency is among different execution units.	Four identical AU-MBU pairs can be installed to carry out all kinds of arithmetic operations (fixed or floating point). Concurrency of execution is among four identical pipes.
Floading point facility more powerful (e.g. Pipe 2 has fast divide, special multipurpose segments).	AU has to be responsible for floating point operations (consists of eight segments).
Requires set up time for vector processing	Also requires set up time (though could be less because the fixed VPF is easier to manage).

spicuous aspects in a vector machine is appropriate. From the previous description, one notices at least four aspects:

1) There is some setup time involved before executing a vector.
2) Additional control in configuring the execution pipe and monitoring operand admission and traversal is needed.
3) Richer instruction sets and intelligent compilers are prerequisites for producing optimized code for vector machines.
4) An intrinsic tradeoff between sequential and vector processing can be derived from the above considerations.

These four observations are discussed in this section.

1) Setup Time and Flush Time

As demonstrated in the ASC and STAR-100 systems, each vector instruction involves a set of vector parameter registers or control vectors to hold the information needed before the instruction can be initiated. The contents of these parameter registers are used to control the addressing operation and storage of result operands, as well as the final termination. In the STAR-100 system, they are used by the MIC and later by other buffers in the stream unit for the continuous initiation of operand fetches and execution until a termination condition is detected by the MIC. In the case of the ASC processor, they are used by the IPU for address calculation, by the MBU for memory references, and by the MIC (in the MBU) for monitoring subsequent execution activities. These parameter registers can be loaded from memory. In doing so, many additional memory fetches (register loading) have to be performed before the vector instruction can be started. These fetches represent an overhead in time—the setup time. If the vector involved has a relatively short field length (the number of iterations to be executed is small), the setup time may be comparable to the actual processing time of the vectors.

Besides the setup time, there is another time measure of interest: the flushing time. The flushing time is the period of time between the initial operation (decode) of the instruction and the exit of the result (for vectors, the first result element) through the entire pipe. Therefore it directly measures the sum of the execution time of all the facilities that the instruction and an operand pair have to go through. Sometimes it is interesting to compare the flush times of a vector pipe to those of a sequential pipe. A vector pipe often has to perform more activities, such as checking the termination condition, checking the control vector, etc. (though some of them can be overlapped with other operations). Therefore it is not surprising to discover that a vector pipe may have a longer flush time than its sequential counterpart.

Here an attempt is made to compare analytically sequential and vector pipeline processing in terms of time efficiency. For a vector pipe, the memory operand supply rate is usually fast enough to meet the speed of the execution pipe(s). For example, in the ASC system, the eight interleaved memory modules can maintain a total data transfer rate of 400M words per second—twice that required to support a central processor with four arithmetic unit pipes when processing vector instructions [13]. Therefore, for an effective vector field length of l, the execution time of the vector instruction can be expressed analytically as (assuming the bottleneck is in execution units):

$$t_{vp} = t_s + t_{vf} + (l - 1)t_e$$

where t_{vp} is the vector instruction processing time; t_s is the setup time; t_{vf} is the vector pipe flush time including decode, address calculation, operand fetch and paired, termination check and execution; and t_e is the speed of the bottleneck segment of the execution unit pipe (in the case of the ASC, all eight segments have the same speed, namely 1 minor cycle = 60 nsec).

The same situation in a sequential pipe can be analogously analyzed. Suppose the same instruction has to be executed on a vector in this case. Without vector processing power, this instruction has to be invoked l times; that is, it must go through the entire pipe l times. Even if the execution unit is fast enough here, it is probable that

the fetching of operands is less efficiently performed. (In vector machines, consecutive storage locations for operands are fetched.) The processing time of the l instructions may be expressed as:

$$t_{sp} = t_{sf} + (l - 1)t_b$$

where t_{sp} is the sequential (pipeline) processing time; t_{sf} is the sequential pipe flush time; and t_b is the speed of bottleneck in the pipe, most likely in fetching operands if the execution unit is fast enough because more interference from unstructured memory references for instructions and operands results.

Comparing t_{vp} and t_{sp} yields:

$$t_s + t_{vf} + (l - 1)t_s \leq t_{sf} + (l - 1)t_b$$

if and only if

$$t_s + t_{vf} - t_{sf} \leq (l - 1)(t_b - t_s).$$

This equation reveals that, if the vector length is reasonably large, vector processing is beneficial, considering the time advantage. If the setup and differential flush times are large compared to the difference of the speeds of the bottlenecks of the two pipes, then a large vector field length is needed to justify processing it in the vector form. Usually $(t_b - t_s)$ has been about a tenth of $t_s + t_{vf} + t_{sf}$; so vector processing provides time efficiency in pipelined processors.

2) *Additional Control and Hardware*

Vector pipes are designed to be cost-effective. They are implemented with sufficient flexibility and power to match the speed of an array processor (which usually is more expensive). For those vector machines with multifunctional pipes, additional control to establish the desirable configurations and routing of the operands between pipe segments are needed. These needs are usually fulfilled by using microcoded control to allow flexibility and simpler circuitry. The hardware and firmware cost so introduced represents a portion of the cost of vector processing. These control functions sometimes are not very conspicuous, but they do require a considerable amount of hardware support.

In addition, some other costs arise indirectly. The vector parameter file or registers represent part of the indirect hardware needed. Larger instruction sets to cope with vector processing also demand longer word lengths—a result that affects the cost throughout the entire system. For smaller word length machines, one can try to get around the problem by using techniques such as dedicated VPF in ASC. Because of its cost-effectiveness and speed advantages, vector processing power may prove adaptable to medium scale systems.

To keep up the execution speed, additional memory buffers (like the MBU) may be necessary to maintain an effective memory supply rate. Memory management problems, though out of the scope of this paper, present a rich area to be explored for vector machines. All this direct and indirect control cost marks the space overhead incurred in vector processing and should be evaluated appropriately in tradeoff considerations.

3) *Richer Instruction Set and Intelligent Compilers*

Once the skeleton processor is assigned, the instruction set has to be designed carefully. As in the case of the STAR-100, suitable higher level vector macro and sparse vector instructions can be implemented (with proper hardware support) so that some application algorithms can be easily handled (fewer instruction and operand fetches and other conflicts). Without such well designed instruction sets, the power of the processor may depreciate many times because inefficient operations, redundant or excessive memory references, and poorly utilized facilities may result.

Since many of the rich instructions are by no means conventional, how to use them effectively in programs becomes a prime concern. For assembly language program writing, the user has to familiarize himself not only with the algorithm he is going to implement, but first with the details of these unconventional instructions [19]. Because of the various architectural aspects involved, he has to choose a suitable algorithm carefully. Often a theoretically fast algorithm

will turn out to be inferior to some normally less effective serial algorithm because of the machine vector characteristic. As a simple example, consider sorting methods. In vector machines, a bubble sort is quite inefficient because of the static multifunctional pipe involved. The bubbling (compare and interchange) of an item incurs too much reconfiguration cost, memory fetch overhead, and setup cost for the pipe. Merge sort algorithms are better because the machine can merge two ordered vectors in one pass without reconfiguration and additional setup. As in the ASC, the instruction vector ORDER A, B, C will try to compare element by element and store the smaller element in C until the entire ordering is accomplished. For example, if A = 1, 3, 4, 5, 7, 8, 9 and B = 2, 3, 5, 8, 10, then C = 1, 2, 3, 3, 4, 5, 5, 7, 8, 9, 10. Therefore only a simple vector instruction is needed to merge sort two ordered vectors. Another good alternative is to find the peak value of an unsorted vector at every iteration, remove and store it at the appropriate place, and repeat until the vector is completely sorted. It is easy to find the peak value of an unsorted vector by using instructions such as SEARCH, and therefore selection sort represents a better strategy (though quite similar) than the conventional bubble sort. This simple example hints how important it is to find the right algorithms to be implemented on vector processors.

Each system requires the installation of intelligent language processors to fully utilize its power. Additional optimization procedures should be incorporated to exploit its vector capability. For example, the optimized FORTRAN compiler for the ASC system was designed to produce highly optimized object code with complete diagnostic and error messages. In general, the additional optimization included is accomplished by analyzing the source program logic and performing optimization on the object code instructions involved. Vector instructions are used wherever feasible, and scalar operations are reordered wherever possible to reduce pipeline reconfiguration and memory reference delays (8-way interleaved memory system). Therefore the compiler can not only recognize array (vector) oriented operations in DO loops but can also reorder some scalar operations generated to meet the architectural characteristics of the machine. Of course the other more conventional optimization procedures are also included, such as elimination of redundant subexpressions, removal of constant assignment statements in a loop, proper register assignment, etc. This burden on compiler designers is quite heavy. Thus the software cost for vector processing is an important item not to be ignored.

4) *Quantitative Comparison of Vector and Scalar Processing*

As mentioned in the other section, the execution time of a vector instruction can be represented by

$$T_{vp} = t_{\text{setup}} + (L - 1)t_e + t_{\text{flush}}.$$

The characteristic is that, without memory interference, operand pairs are accepted at a rate of $1/t_e$ to generate a result. t_e varies from one instruction to another because a loop may exist inside the AU pipe and static control is used. The significance of vector processing is that operand fetch is completely concealed behind actual execution. To achieve this concealment, the memory bandwidth available to the memory buffer unit for fetching operands must be sufficient to sustain that rate. In the ASC, for example, eight operand words (one octet) can be fetched every memory cycle (160 nsec), which is sufficient to yield a bandwidth of 3 words/60 nsec (the basic segment time).

Another advantage is that while a vector instruction is being processed, no additional instruction fetch is needed; memory interference is thus reduced. In fact, because of this, vector instructions can be simulated in [some] systems having an instruction file large enough to eliminate such instruction fetches.

The overhead of a vector instruction includes its setup time. In the ASC, it includes transferring vector parameters to the control (in the MBU) and starting the AU pipe—altogether 27 segment cycles. This setup time may vary in other systems; for

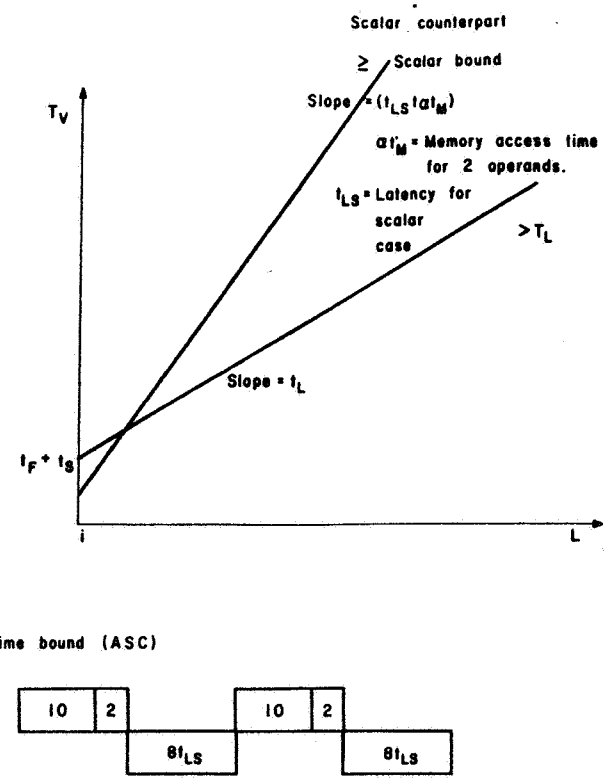

FIGURE 19. Scalar vector timing comparison (TI ASC).

example, the setup time in the STAR-100 can go up to over 100 minor cycles. Without memory interference, the total execution time of a vector instruction may be plotted against the vector length L as in Figure 19. If that vector instruction is replaced by a scalar loop, the resulting execution time is usually much larger, depending on L. A vector instruction can be decomposed into three or more scalar instructions:

Scalar Loop
- [Update Pointer]
- [Basic Instruction] ← (could involve more than one scalar instruction depending on instruction format)
- [Test Pointer and Branch]

Execution of this loop L times requires considerable time. One significant factor is the operand fetch, which is not done (look-ahead) fast enough as in the vector counterpart. So, by the time the operand comes back from memory, several precious processing cycles have been lost (Figure 19).

5) *Tradeoff Summary*

In this section, we have discussed the time and the space overhead needed in vector processing as compared to a *sequential* pipelined processor (such as the IBM 360/91). The advantages of vector processing are its speed improvement for reasonably long vectors and its more orderly management and thus better utilization of the memory system and other resources when dealing with vectors. The costs it incurs are the needed firmward control and additional software facilities to utilize its power. When the latter problems have been solved successfully at less cost, vector processing may be generalized and extended to smaller scale processing systems.

4. OVERVIEW OF TWO RECENT MACHINES

4.1 The Asynchronous CRAY-1 Computer

We describe the vector processing abilities of a new fourth generation pipeline com-

puter CRAY-1 of Cray Research Corporation [26] Several unique features of this machine are explored to supplement the ideas in Section 3 and to illustrate the current trend of progress.

The CRAY-1 design philosophy follows closely the tradition of the CDC 6600 and 7600. The twelve functional units incorporate vector processing capabilities and are "connectable" to form efficient chains, thereby maximizing overlapped vector processing. These units represent a deviation from the universal (multifunctional) pipe approach as adopted by the ASC and STAR-100. However, the tradeoff is quite apparent. The control here is more complex. Some specific features of the CRAY-1 include:

Operating Registers

Figure 20 illustrates the register organization of this computer. The primary operating registers are the scalar and vector registers called S and V registers, respectively. Each of the eight V registers has 64 bits. A scalar instruction may perform some function, such as addition, obtaining operands from two S registers and entering the result into another S register. A vector instruction performs the same function in an analogous fashion, obtaining a new pair of operands each clock cycle of 12.5 nsec from two V registers and storing the result into another V register. The contents of the vector length (VL) register determine the number of operations performed by the vector instruction. Eight 24-bit A registers are used as address registers for memory references and as index registers. The A and S registers are each supported by 64 rapid access temporary storage registers called B and T registers. Data can be transferred between A, B, S, T, or V registers and memory.

Memory

Up to one million 64-bit words are arranged in 16 banks with a bank cycle time of 4 clock periods. The memory is constructed of bipolar 1024-bit LSI chips.

Instruction Buffers

Instructions, which are either 16 or 32 bits, are executed from four instruction buffers, each consisting of 64 16-bit registers. Associated with each instruction buffer is a base address register that is used to determine if the current instruction resides in a buffer. Forward and backward branching within the buffers is possible, and the program segments may be discontinuous in the program buffer. When the current instruction does not reside in a buffer, one of the instruction buffers is filled from memory. Four memory words are read per clock period to the least recently filled instruction buffer. To allow the current instruction to be issued as soon as possible, the memory word containing the current instruction is among the first to be read.

Functional Units

The CRAY-1 CPU has twelve functional units, each of which is independent of the others and therefore capable of parallel operation. A functional unit receives operands from registers and delivers each result to a register when the operation is completed. The functional units retain no information regarding their past operation. The three functional units that provide 24-bit results to A registers are Integer Add, Integer Multiply, and Population Count. The three functional units that provide 64-bit results to the S registers are Integer Add, Shift, and Logical. The three functional units providing 64-bit results to the V registers only are Integer Add, Shift, and Logical. The three functional units that provide 64-bit results to either the S or V registers are Floating Add, Floating Multiply, and Reciprocal Approximation. All functional units are buffered, perform their algorithms in a fixed amount of time, and produce one result per clock period.

Vector Operations

Because of the instruction formats adopted, vector instructions are of four types. One type of vector instruction obtains operands

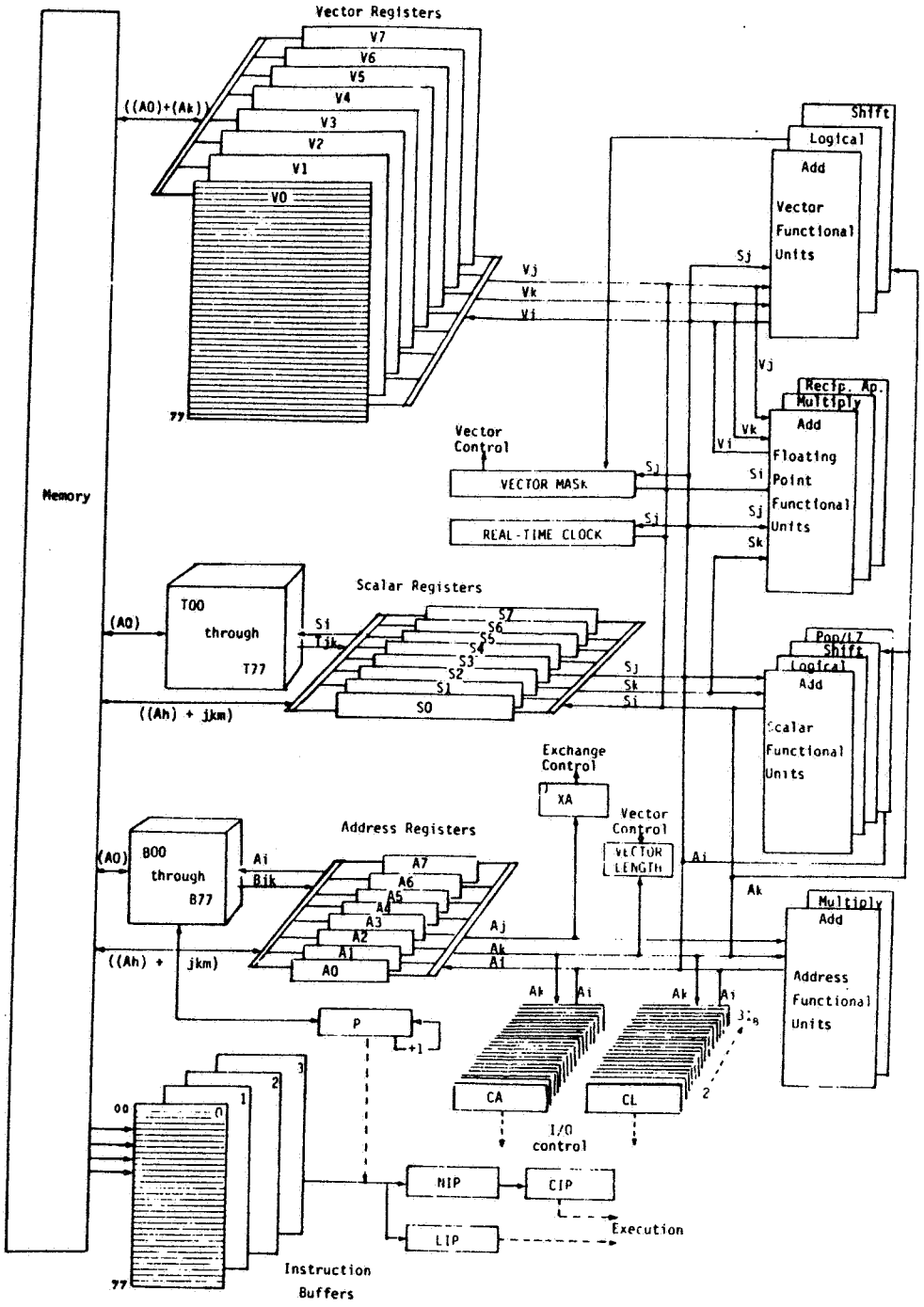

FIGURE 20. Register block diagram-CRAY-1.

from one or two V registers and enters the results into another V register (Figure 21(a)). Successive operand pairs are transmitted from V_j and V_k to the segmented functional unit each clock period, and the corresponding results emerge from the functional unit n periods later, where n is the execution time. The results are entered

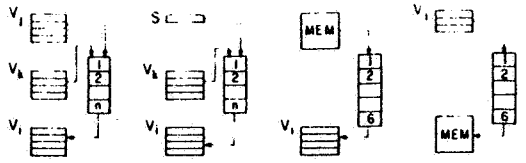

FIGURE 21a. Type I vector instruction.
FIGURE 21b. Type II vector instruction.
FIGURE 21c. Type III vector instruction.
FIGURE 21d. Type IV vector instruction.

into the result register V_i. The contents of the vector length (VL) register determines the number of operand pairs processed by the functional unit.

When vectors contain more than 64 elements, they can be processed by dividing them into vectors of 64 elements (or less).

The second type of vector instruction obtains one operand from an S register and one from a V register (Figure 21(b)). The last two types of vector instructions transmit data between memory and the V registers (Figure 21(c) and Figure 21(d)). The path between memory and the V registers may be considered a functional unit for timing considerations.

The pipelined execution of vector instructions is discussed next. Let VI_j be the jth bit of the vector register VI. Since there are 64 bits in the register (VI_0 through VI_{63}), Figure 22 shows the timing chart for the execution of a floating point addition operation using vector instruction of type I. When the instruction is issued at clock period t_0, the first pair of operands ($V1_0$ and $V2_0$) is transmitted to the add functional unit, where it arrives at time t_1. The function is executed in six clock time periods and the first result exists from the functional unit at clock period t_1. The second pair of operands ($V1_1$ and $V2_1$) arrive at the functional unit at t_2, and so on.

Parallel Operations

When a vector instruction is issued, the required functional unit and the operand registers are reserved for the number of clock periods determined by the vector length. A subsequent vector instruction requiring the same resources (functional units and registers) cannot be executed until the resources are released; however, parallel (simultaneous) execution of neighboring instructions that do not interfere in their resource requirements is permitted.

Chaining

The CRAY machine has the unique ability to combine several pipeline executions in a sequence by chaining. In the chaining process a result register which receives the result of a vector instruction can become the operand register of a succeeding instruction. The succeeding instruction is started as soon as the first result arrives for use as an operand. Figure 23(a) shows a chain of four instructions reading a vector of integers from memory, adding that vector to another, shifting the sum, and finally forming the

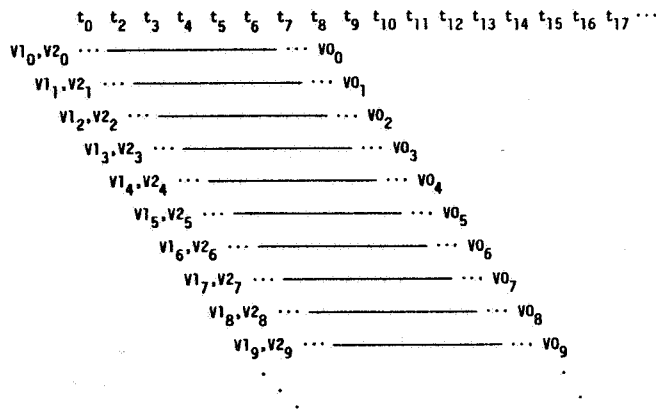

FIGURE 22. Vector instruction timing example (V0 VI V2).

logical product of the shifted sum and a mask vector. The result of the four instructions is placed in vector register V5. Figures 23(b) and 23(c) graphically depict the passage and timing of information through the functional units.

1. V0 ← Memory (Memory Read)
2. V2 ← V0 + V1 (Integer Add)
3. V3 ← V2 < A3 (Left Shift)
4. V5 ← V3 ∧ V4 (Logical Product)

FIGURE 23a. Chaining example.

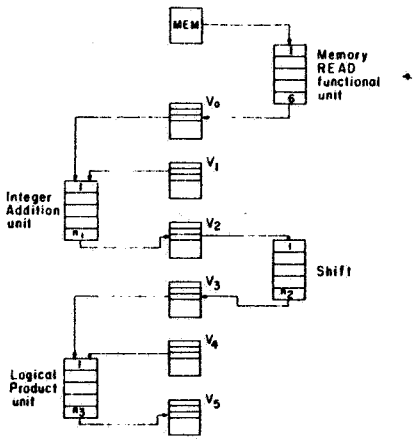

FIGURE 23b. Chaining.

Performance

A performance study of several subroutines for the CRAY-1 FORTRAN library and matrix multiplication illustrates extreme efficiency of the pipeline operations. Vector operations employ algorithms similar to their scalar counterparts. The studies indicate that the vector subroutines outperform the scalar subroutines. Figure 24 illustrates the performance of several library subroutines. The cost (execution time) per result in clock cycles is plotted against the vector length. The cost is constant for scalar subroutines. For vector subroutines the cost drops dramatically and rapidly approaches a lower limit as vector length increases. The performance of matrix multiplication provides yet another illustration of efficiency of pipeline processing in vector operations. Given a matrix $[A]$ of dimension K by N and a matrix $[B]$ of dimension N by M, the element ij of the product matrix $[C]$ is given by

$$c_{ij} = \sum_{n=1}^{N} a_{in} \cdot b_{nj}.$$

Figure 25 shows the execution rate of multiplication of square matrices as a function of matrix dimension. The execution rate is

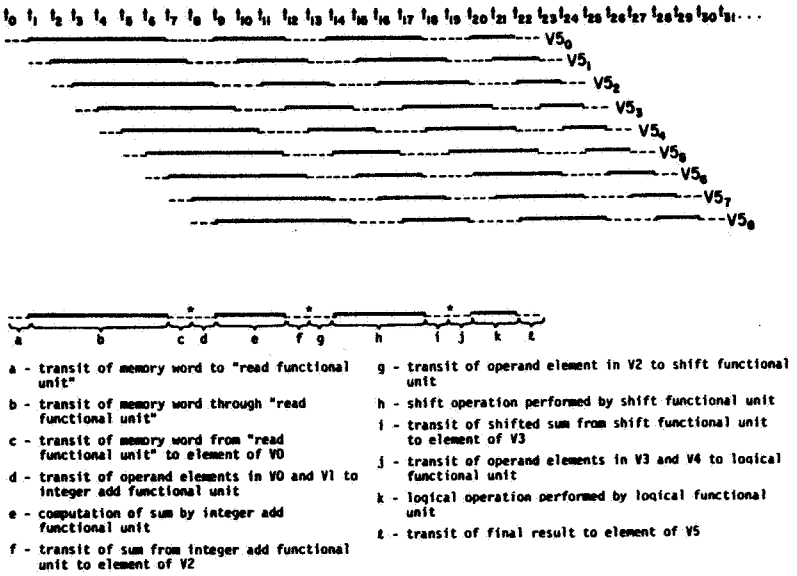

FIGURE 23c. Timing diagram for chaining example.

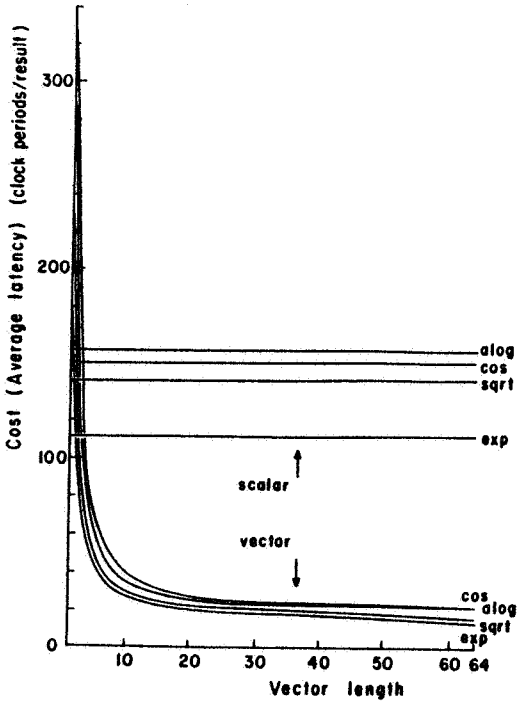

FIGURE 24. Scalar/vector timing comparison.

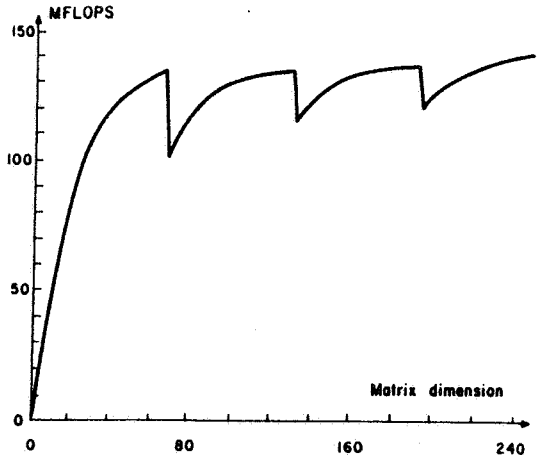

FIGURE 25. Matrix multiplication timing.

defined in terms of "millions of floating point operations per second" ($MFLOP_p$). The number of floating point operations required to multiply two n-dimensional square matrices is $(2n - 1)n^2$, since each of the n^2 elements of the result matrix is formed by summing n products. The fall of throughput rate at discrete time instants is caused by the architectural design in which a vector length of 64 is chosen (for buffering register size). Consequently a vector restart is necessary at those time instances.

4.2 Amdahl 470 V/6

Finally, a few words must be said about the more recent Amdahl 470 V/6 machine [31]. Besides adopting high speed LSI chips for the CPU and most of the channel unit, it employs the technique of pipelining in a reasonably simple way. The CPU instruction execution can be partitioned into six phases, A–F. Phase A consists of instruction decode and reading of general purpose registers (if ready). Phase B calculates the effective address of the memory operand and start fetching (time not fixed). Phase C reads the memory operand into buffer and start execution. Phase D continues the completion of execution (time not fixed). Phase E checks the result generated from the functional unit to see if retry is necessary. If not, phase F writes back the result. Because write back is done at the last phase, software rollback and retry of instruction can be replaced by a simple hardware retry (for most instructions).

The execution unit is decomposed into four subunits: multiplier, adder, shifter, and byte mover. These subunits have a propagation delay of one basic cycle (though many instructions need several iterations and hence several basic cycles). Besides parity check, each functional unit checks for error using residue arithmetic.

Therefore the CPU architecture of this machine is a simple pipeline served by four functional units. With high speed circuits and pipelining, the major problem to be solved is the operand supply rate. Since virtual addressing is implemented, address translation and subsequent memory fetches have to be performed most efficiently in order to be compatible in speed with the pipeline. Two distinct features are used to achieve this. First, the primary memory is buffered by a 16K byte cache that is managed by the set associative scheme (a set of

primary memory blocks is mapped into a corresponding set of cache blocks associatively). To speed up translation, a 256-entry Translation Lookaside Buffer (TLB) is installed for tag (virtual address tags) matching (associatively). To complete the virtual memory mechanism, a Segment Table Origin (STO) stack is used to identify the environments of different TLB entries. With a high hit ratio and possible prefetch of quarter-line segments, each CPU request may take only two cache cycles for completion and hence can be speeded up in a manner compatible with the pipeline flow (c/o Phase B). A comparison of this machine with the IBM 370/168 (a comparably priced third generation computer) shows that the Amdahl 470 V/6 provides three times the performance, yet requires only one third the space of the 370/168 (60 ft.^{3} vs. 200 ft.^{3}) [32]. One other factor that contributes to this comparison is the substantial savings in packaging size with the improvement in LSI technology. The LSI portion of the 470 V/6 takes up 51 cards, each 7 ½ inches square, and 42 chip positions. With simpler and shorter connections, the reliability of the system is unquestionably upgraded.

While the Amdahl 470 V/6 makes use of technology advancement to its great advantage, its comparatively simple pipeline architecture prompts more future design efforts. Specifically, the floating point facilities in the system are rudimentary (the functional units are not designed for floating point operations), and for many applications phase D (execution phase) may become a bottleneck of the pipeline flow. With the success of this machine, the versatility and prospect of pipelining make it an attractive feature in future system design.

5. CONCLUSION

Pipelined processors represent an intelligent approach to speeding up instruction processing when the memory access time has improved to a certain extent. Without having to duplicate the entire processors n times, a throughput rate of close to n times improvement over a nonpipelined case may be achieved. To make this possible, certain problems have to be solved, including: parallelism and busing structure; handling of unexpected events; and efficient sequence control with a well-designed instruction set. Special vector processing capability is one way to specify parallelism in programs easily. These problems and solutions are discussed and solutions in existing machines illustrated. The multilevel application of pipeline discipline is promising in upgrading the performance of a processor, especially from a cost-effective point of view, and certainly deserves future investigation to generalize its application to even smaller scale systems.

ACKNOWLEDGMENT

The authors are grateful to Mr. Gary Ho and Mr. V. Shah for their contributions.

REFERENCES

[1] CHEN, T. C. "Unconventional superspeed computer systems," in *AFIPS 1971 Spring Jt. Computer Conf.*, AFIPS Press, Montvale, N.J., 1971, pp. 365–371.

[2] MCINTYRE, D. "An introduction to the ILLIAC IV computer," *Datamation* (April 1970), 60–67.

[3] EVENSEN, A. J.; AND TROY, J. L. "Introduction to the architecture of a 288-element PEPE," in *Proc. 1973 Sagamore Conf. on Parallel Processing*, Springer-Verlag, N.Y. 1973, pp. 162–169.

[4] RUDOLPH, J. A. "A production implementation of an associative array processor—STARAN," in *AFIPS 1972 Fall Jt. Computer Conf.*, AFIPS Press, Montvale, N.J., 1972, pp. 229–241.

[5] MARVEL, O. E. "HAPPE—Honeywell associative parallel processing ensemble," in *Proc. Symp. on Computer Architecture*, Univ. of Florida, 1973, pp. 261–268.

[6] STANGA, D. C. "Univac 110 multiprocessor system," in *AFIPS 1967 Spring Jt. Computer Conf.*, Thompson Book Co., Washington, D.C., 1967, pp. 67–74.

[7] CHEN, T. C. "Parallelism, pipelining and computer efficiency," *Computer Design* (Jan. 1971), 69–74.

[8] RAMAMOORTHY, C. V.; AND LI, H. F. "Efficiency in generalized pipeline networks," in *AFIPS 1974 National Computer Conf.*, AFIPS Press, Montvale, N.J., 1974, pp. 625–635.

[9] ANDERSON, D. W.; SPARACIO, F. J.; AND TOMASULO, R. M. "IBM System 360 Model 91, machine philosophy and instruction handling," *IBM J. R. and D.* (Jan. 1967), 8–24.

[10] WATSON, W. J. "The TIASC—a highly

modular and flexible super computer architecture," in *AFIPS 1972 Fall Jt. Computer Conf.*, AFIPS Press, Montvale, N.J., 1972, pp. 221–228.

[11] HINTZ, R. G.; AND TATE, D. P. "Control Data STAR-100 processor design," *COMPCON 72*, IEEE, N.Y., 1972.

[12] TOMASULO, R. M. "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. R. and D.* (Jan. 1967), 25–33.

[13] TEXAS INSTRUMENTS INC., *A description of the advanced scientific computer system*, Austin, Texas, April 1973.

[14] CONTROL DATA CORP., *Control Data STAR-100 computer hardware reference manual*, 1974.

[15] TEXAS INSTRUMENTS INC., *The ASC system—central processor*," Austin, Texas, Dec. 1971.

[16] DAVIDSON, E. *The design and control of pipeline function generator*, Stanford Report Stanford University, 1972.

[17] DAVIDSON, E. S.; SHAR, L. E.; THOMAS, A. T.; AND PATEL, J. H. "Effective control for pipelined computers," *COMPCON 75*, IEEE, N.Y., 1975, pp. 181–184.

[18] RAMAMOORTHY, C. V.; AND LI, H. F. "Sequencing control in multifunctional pipelined systems," in *Proc. 1975 Sagamore Conf. on Parallel Processing*, Springer-Verlag, N.Y., 1975.

[19] KISHI, T.; AND RUDY, T. "STAR TREK," in *COMPCON 75*, IEEE, N.Y., 1975, pp. 185–188.

[20] WALLACE, C. S. "A suggestion for a fast multiplier," *IEEE Trans. Computers* EC-13 (1964), 14–17.

[21] ANDERSON, S. F.; EARLE, J. G.; GOLDSCHMIDT, R. E.; AND POWERS, D. M. "The IBM System/360 Model 91: floating-point execution unit," *IBM J. R. and D.* (Jan. 1967), 34–53.

[22] CHEN, T. C. "Overlap and pipeline processing," in *Introduction to computer architecture*, H. S. Stone, (Ed.), Science Research Associates, Chicago, Ill., 1975.

[23] RAMAMOORTHY, C. V.; AND KIM, K. H. "Pipelining—the generalized concept and sequencing strategies," in *AFIPS 1974 National Computer Conf.*, AFIPS Press, Montvale, N.J., 1974, pp. 289–297.

[24] KIM, K. H. "Optimizing architecture in parallel processing," Memorandum RLE-M482, Electronics Research Lab., Univ. of Calif. Berkeley, Nov. 1974.

[25] REDDI, S. S. "Sequencing strategies in pipeline computer systems," PhD Thesis, Univ. of Texas, Austin, August 1972.

[26] CRAY RESEARCH CORP., "Description of CRAY-1", Minn., 1975.

[27] COTTEN, L. W. "Circuit implementation of high speed pipeline systems," in *AFIPS 1965, Fall Jt. Computer Conf.*, Thompson Book Co., Washington, D.C., 1967, pp. 489–504.

[28] HALLIN, T. G.; AND FLYNN, M. J. "Pipelining of arithmetic functions," *IEEE Trans. Computers*, C-21 (August 1972), 880–886.

[29] LARSEN, A. G.; AND DAVIDSON E. S. "Cost-effective design of special purpose processors: a fast Fourier transform case study," in *11th Allerton Conf.*, Univ. Illinois, 1973, pp. 547–554.

[30] LI, H. F. "A structural study of parallel pipelined systems," PhD Dissertation, Univ. of Calif., Berkeley, 1975.

[31] AMDAHL CORP. *Amdahl 470 V/6 machine reference*, Sunnyvale, Calif., 1975.

[32] BEALL, R. J. "Packaging for a super computer," Amdahl Corp., 1975.

[33] MARTIN, J. T.; ZWAKENBERG, R. G.; AND SOLBECK, S. V. "LRLTRAN language used with the CHAT and STAR compilers," LTSS Chapter 207, Lawrence Livermore Lab., Lawrence, Calif.

[34] WEDEL, D. "FORTRAN for the Texas Instruments ASC System," *SIGPLAN 10* (1975), 119–132.

[35] BASILI, V. R.; AND KNIGHT, J. C. "A language design for vector machines," *SIGPLAN 10* (1973), 39–53.