Dynamic Instruction Scheduling and the Astronautics ZS-1

James E. Smith
Astronautics Corporation of America

Pipelined instruction processing has become a widely used technique for implementing high-performance computers. Pipelining first appeared in supercomputers and large mainframes, but can now be found in less expensive systems. For example, most of the recent reduced instruction set computers use pipelining.[1,2] Indeed, a major argument for RISC architectures is the ease with which they can be pipelined. At the other end of the spectrum, computers with more complex instruction sets, such as the VAX 8800,[3] make effective use of pipelining as well.

The ordering, or "scheduling," of instructions as they enter and pass through an instruction pipeline is a critical factor in determining performance. In recent years, the RISC philosophy has become pervasive in computer design. The basic reasoning behind the RISC philosophy can be stated as "simple hardware means faster hardware, and hardware can be kept simple by doing as much as possible in software." A corollary naturally follows, stating that instruction scheduling should be done by software at compile time. We refer to this as *static instruction scheduling*, and virtually every new computer system announced in the last several years has followed this approach.

Dynamic instruction scheduling resolves control and data dependencies at runtime. This extends performance over that possible with static scheduling alone, as shown by the ZS-1.

Many features of the pioneering CDC 6600[4] have found their way into modern pipelined processors. One noteworthy exception is the reordering of instructions at runtime, or *dynamic instruction scheduling*. The CDC 6600 *scoreboard* allowed hardware to reorder instruction execution, and the memory system *stunt box* allowed reordering of some memory references as well. Another innovative computer of considerable historical interest, the IBM 360/91,[5] used dynamic scheduling methods even more extensively than the CDC 6600.

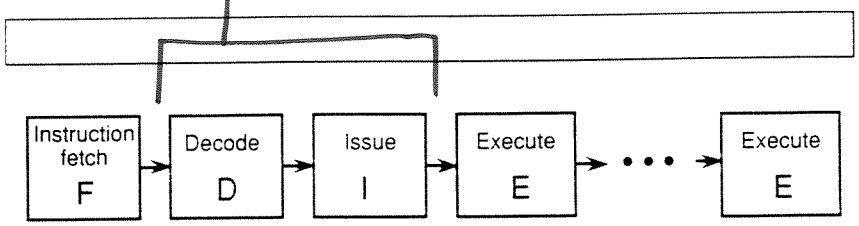
As the RISC philosophy becomes accepted by the design community, the benefits of dynamic instruction scheduling are apparently being overlooked. Dynamic instruction scheduling can provide performance improvements simply not possible with static scheduling alone.

This article has three major purposes:

(1) to provide an overview of and survey solutions to the problem of instruction scheduling for pipelined computers,
(2) to demonstrate that dynamic instruction scheduling can provide performance improvements not possible with static scheduling alone, and
(3) to describe a new high-performance computer, the Astronautics ZS-1, which uses new methods for implementing dynamic scheduling and which can outperform computers using similar-speed technologies that rely solely on state-of-the-art static scheduling techniques.

0018-9162/89/0700-0021$01.00 ©1989 IEEE

MIPS: combines these



(a)

time →

inst. 1	FDIEEEE
inst. 2	FDIEEEE
inst. 3	FDIEEEE
inst. 4	FDIEEEE

(b)

Figure 1. Pipelined instruction processing: (a) a typical pipeline; (b) ideal flow of instructions through the pipeline.

```
R1 ← (Y)          Load register R1 from memory location Y
R2 ← (Z)          Load register R2 from memory location Z
R3 ← R1 +f R2     Floating add registers R1 and R2
(X) ← R3          Store the result into memory location X
R4 ← (B)          Load register R4 from memory location B
R5 ← (C)          Load register R5 from memory location C
R6 ← R4 *f R5     Floating multiply registers R4 and R5
(A) ← R6          Store the result into memory location A
```

(a)

time →

```
R1 ← (Y)          FDIEEEE
R2 ← (Z)           FDIEEEE
R3 ← R1 +f R2       FD...IEEE
(X) ← R3             F...D..IEEEE
R4 ← (B)                F..DIEEEE
R5 ← (C)                   FDIEEEE
R6 ← R4 *f R5               FD...IEEE
(A) ← R6                      F...D..IEEEE
```

(b)

Figure 2. Pipelined execution of X=Y+Z and A=B*C: (a) machine code; (b) pipeline timing.

Introduction to pipelined computing

Pipelining decomposes instruction processing into assembly line-like stages. Figure 1 illustrates a simple example pipeline. In Figure 1a, the pipeline stages are:

(1) Instruction fetch -- for simplicity assume that all instructions are fetched in one cycle from a cache memory.

(2) Instruction decode -- the instruction's opcode is examined to determine the function to perform and the resources needed. Resources include general-purpose registers, buses, and functional units.

(3) Instruction issue -- resource availability is checked and resources are reserved. That is, pipeline control interlocks are maintained at this stage. Assume that operands are read from registers during the issue stage.

(4) Instruction execution -- instructions are executed in one or several execution stages. Writing results into the general-purpose registers is done during the last execution stage. In this discussion, consider memory load and store operations to be part of execution.

Figure 1b illustrates an idealized flow of instructions through the pipeline. Time is measured in clock periods and runs from left to right. The diagram notes the pipeline stage holding an instruction each clock period. *F* denotes the instruction fetch stage, *D* denotes the decode stage, *I* denotes the issue stage, and *E* denotes the execution stages.

In theory, the clock period for a *p*-stage pipeline would be $1/p$ the clock period for a nonpipelined equivalent. Consequently, there is the potential for a p times throughput (performance) improvement. There are several practical limitations, however, on pipeline performance. The limitation of particular interest here is instruction dependencies.

Instructions may depend on results of previous instructions and may therefore have to wait for the previous instructions to complete before they can proceed through the pipeline. A *data dependence* occurs when instructions use the same input and/or output operands; for example, when an instruction uses the result of a preceding instruction as an input operand. A data dependence may cause an instruction to wait in the pipeline for a preceding instruction to complete. A *control dependence* occurs when control decisions (typically as conditional branches) must be made before subsequent instructions can be executed.

Figure 2 illustrates the effects of dependencies on pipeline performance. Figure 2a shows a sequence of machine instructions that a compiler might generate to perform the high-level language statements $X = Y + Z$ and $A = B * C$. Assume load and store instructions take

four execution clock periods while floating-point additions and multiplications take three. (These timing assumptions represent a moderate level of pipelining. In many RISC processors fewer clock periods are needed. On the other hand, the Cray-1 requires 11 clock periods for a load, and floating-point additions take six. Cray-2 pipelines are about twice the length of the Cray-1's.)

Figure 2b illustrates the pipeline timing. A simple in-order method of instruction issuing is used; that is, if an instruction is blocked from issuing due to a dependence, all instructions following it are also blocked. The same letters as before are used to denote pipeline stages. A period indicates that an instruction is blocked or "stalled" in a pipeline stage and cannot proceed until either the instruction ahead of it proceeds, or, at the issue stage, until all resources and data dependencies are satisfied.

The first two instructions issue on consecutive clock periods, but the add is dependent on both loads and must wait three clock periods for the load data before it can issue. Similarly, the store to location X must wait three clock periods for the add to finish due to another data dependence. There are similar blockages during the calculation of A. The total time required is 18 clock periods. This time is measured beginning when the first instruction starts execution until the last starts execution. (We measure time in this way so that pipeline "fill" and "drain" times do not unduly influence relative timings.)

```
R1 ← (Y)          Load register R1 from memory location Y
R2 ← (Z)          Load register R2 from memory location Z
R4 ← (B)          Load register R4 from memory location B
R5 ← (C)          Load register R5 from memory location C
R3 ← R1 +f R2     Floating add X and Y
R6 ← R4 *f R5     Floating multiply B and C
(X) ← R3          Store R3 to memory location X
(A) ← R6          Store R6 to memory location A
```

(a)

time →

```
R1 ← (Y)          FDIEEEE
R2 ← (Z)           FDIEEEE
R4 ← (B)            FDIEEEE
R5 ← (C)             FDIEEEE
R3 ← R1 +f R2         FD.IEEE
R6 ← R4 *f R5          F.D.IEEE
(X) ← R3                  F.DIEEEE
(A) ← R6                     FD.IEEEE
```

(b)

Figure 3. Reordered code to perform X=Y+Z and A=B*C: (a) machine code; (b) pipeline timing.

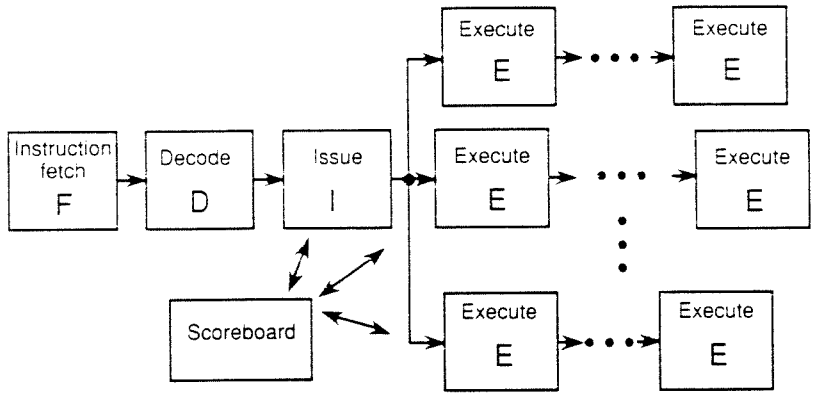


Figure 4. Block diagram of the CDC 6600-style processor.

Instruction scheduling

An important characteristic of pipelined processors is that using equivalent, but reordered, code sequences can result in performance differences. For example, the code in Figure 3a performs the same function as that in Figure 2a except that it has been reordered, or "scheduled," to reduce data dependencies. Furthermore, registers have been allocated differently to eliminate certain register conflicts that appear to the hardware as dependencies. Figure 3b illustrates the pipeline timing for the code in Figure 3a. Note that there is considerably more overlap, and the time required is correspondingly reduced to 11 clock periods.

The above is an example of *static instruction scheduling*, that is, the scheduling or reordering of instructions that can be done by a compiler prior to execution. Most, if not all, pipelined computers today use some form of static scheduling by the compiler. Instructions can also be reordered after they have entered the pipeline, which is called *dynamic instruction scheduling*. Used in some high-performance computers over 20 years ago, it is rarely used in today's pipelined processors.

Dynamic instruction scheduling: scoreboards. The CDC 6600[4] was an early high-performance computer that used dynamic instruction scheduling hardware. Figure 4 illustrates a simplified CDC 6600-type processor. The CDC

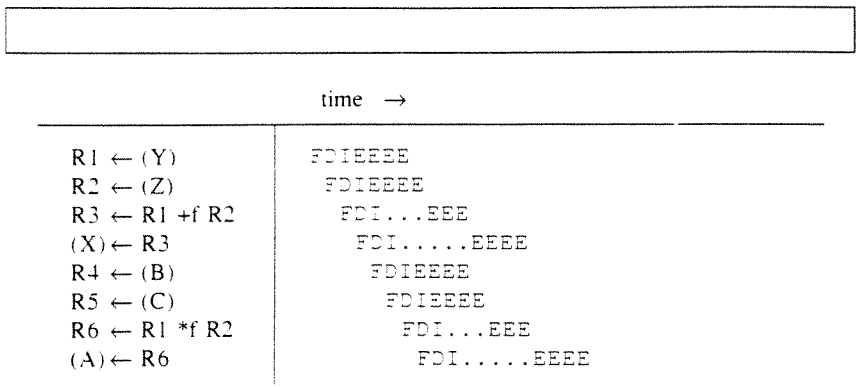


Figure 5. Pipeline flow in a CDC 6600-like processor.

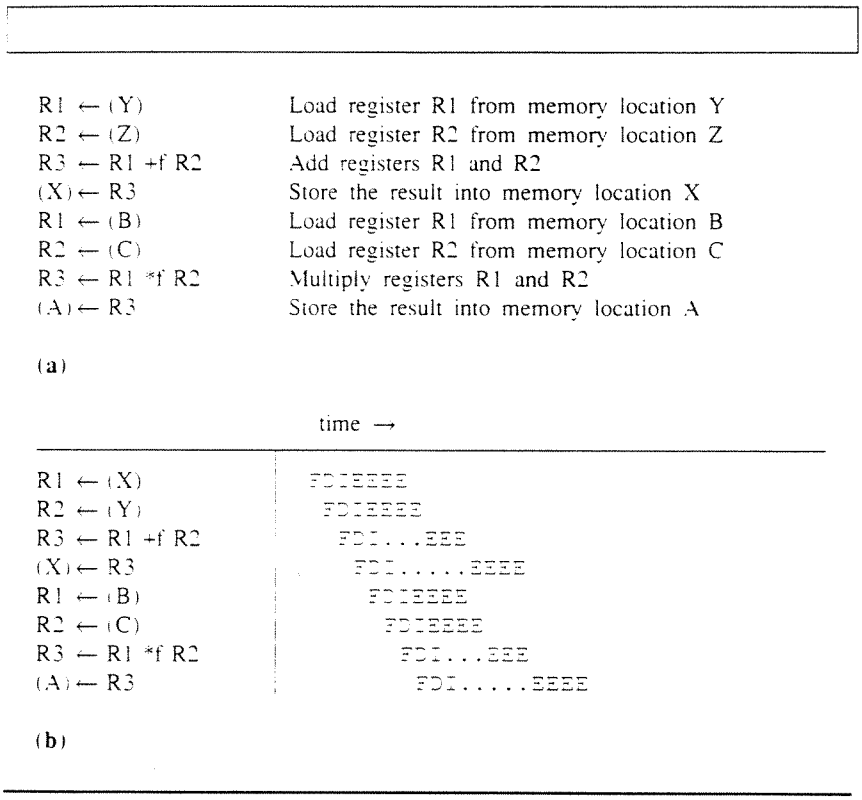


Figure 6. Pipelined execution of X=Y+Z and A=B*C using Tomasulo's algorithm: (a) minimal register machine code; (b) timing for pipeline flow.

6600 was pipelined only in the instruction fetch/decode/issue area. It used parallel execution units (some duplicated) to get the same overlapped effect as pipelining. In addition, parallel units allowed instructions to complete out of order, which, in itself, is a simple form of dynamic scheduling.

The CDC 6600 had instruction buffers for each execution unit. Instructions were issued regardless of whether register input data were available (the execution unit itself had to be available, however). The instruction's control information could then wait in a buffer for its data to be produced by other instructions. In this way, instructions to different units could issue and begin execution out of the original program order.

To control the correct routing of data between execution units and registers, the CDC 6600 used a centralized control unit known as the *scoreboard*. The scoreboard kept track of the registers needed by instructions waiting for the various functional units. When all the registers had valid data, the scoreboard issued a series of "go" signals to cause the data to be read from the register file, to send data to the correct functional unit, and to start the unit's execution. Similarly, when a unit was about to finish execution, it signaled the scoreboard. When the appropriate result bus was available, the scoreboard sent a "go" signal to the unit, and it delivered its result to the register file.

The original CDC 6600 scoreboard was a rather complicated control unit. In recent years, the term "scoreboard" has taken on a generic meaning: any control logic that handles register and result bus reservations, including methods that are not as sophisticated as the 6600's scoreboard.

Figure 5 illustrates the example of Figure 2 executed with a pipeline using scoreboard issue logic similar to that used in the CDC 6600. The pipeline latencies are the same as those in previous examples. The add instruction is issued to its functional unit before its registers are ready. It then waits for its input register operands. The scoreboard routes the register values to the adder unit when they become available. In the meantime, the issue stage of the pipeline is not blocked, so other instructions can bypass the blocked add. Performance is improved in a way similar to the static scheduling illustrated in Figure 3. Here, it takes 13 clock periods to perform the required operations.

Developed independently, and at roughly the same time as the CDC 6600, the IBM 360/91 floating-point unit[6] used a similar but more elaborate method of dynamically issuing floating-point instructions. Instructions were issued to *reservation stations* at the inputs of the floating-point units. The reservation stations held not only control information, but also operand data. The operands were given *tags* that associated the data buffers in the reservation stations with functional units that would supply the data. The data would then be automatically routed via a common data bus to the appropriate reservation stations as they became available. Any instruction in a reservation station that had received all its input operands was free to issue.

Tomasulo's algorithm,[6] used in the IBM 360/91, differed from the CDC 6600 method in two important ways:

(1) There were multiple reservation stations at each unit so that more than one instruction could be waiting for operands. The first one with all its input operands could proceed ahead of the others.

(2) The use of tags, rather than register designators, allowed a form of dynamic register reallocation, as well as instruction reordering. This feature was particularly important in the 360/91 because the IBM 360 architecture had only four floating-point registers.

Figure 6 shows the example of Figure 2 with a minimal register allocation. The pipeline timing with Tomasulo's algorithm appears in Figure 6b. Here, the timing is essentially the same as with the CDC 6600 scoreboard in Figure 5. Note, however, that the registers are automatically reassigned by the hardware. If the CDC 6600 were given the same code as in Figure 6, its timing would be similar to the slower timing shown in Figure 2 because it would be unable to reassign registers "on-the-fly" as the 360/91 does.

Control dependencies. It may appear from the previous examples that, while the dynamic scheduling methods are very clever, a compiler could be used to arrange the code as well as the hardware can (see Figure 3). One need only consider programs containing conditional branches, especially loops, to see that this is not the case.

For static code scheduling, instruction sequences are often divided into *basic blocks*. A basic block is a sequence of code that can only be entered at the top and exited at the bottom. Conditional branches and conditional branch targets typically delineate basic blocks. To phrase it differently, control dependencies break programs into basic blocks. Dynamic scheduling allows instruction reordering beyond machine-code-level basic-block boundaries. This ability to schedule past control dependencies is demonstrated in later examples.

Some advanced static scheduling methods are also directed at minimizing the effects of control dependencies. Loop unrolling and vectorizing start with high-level language basic blocks and produce machine-code-level basic blocks containing more operations. Performance is improved by reducing the number of control dependencies, but static scheduling still cannot look beyond the larger machine-code-level blocks. Dynamic scheduling can be used in conjunction with loop unrolling and vectorizing to further enhance performance by looking beyond the remaining control dependencies.

Trace scheduling[7] is based on statically predicting conditional branch outcomes. These predictions are based on compile-time information deduced from the source code or from compiler directives. Instructions can then be reordered around control dependencies, but instructions must also be inserted to "undo" any mistaken static branch predictions. Incorrect branch predictions result in:

(1) instructions executed that should not be,
(2) additional instructions to undo the mistakes, and
(3) execution of the correct instructions that should have been done in the first place.

With dynamic resolution of control dependencies, there are no mistakes, and no "undoing" is necessary.

Some types of control constructs inherently make certain static scheduling techniques extremely difficult. For example, "while" loops (in Fortran, Do loops with early exit conditions) severely restrict loop unrolling and vectorizing techniques because the exact number of loop iterations is only discovered at the time the loop is completed.

As the following example illustrates, however, there are other advantages to dynamic resolution of control dependencies associated with ordinary Fortran Do loops. Figure 7 shows a Fortran loop and a compilation into machine instructions. The loop code has been statically scheduled. Its execution for two iterations on a standard pipelined machine is shown in Figure 7c (left pipeline flow), and its execution with a Tomasulo-like dynamic hardware scheduler is shown in Figure 7c (right pipeline flow).

Performance with the dynamic scheduling algorithm is improved because the conditional branch at the end of the loop issues and executes before some of the preceding instructions have begun execution. After the conditional branch is out of the way, instructions following the branch instruction can also be fetched and executed before all the instructions from the previous basic block(s) have executed.

Note that, in the above example, no special techniques are used to improve the performance of the conditional branch instruction. That is, there is no branch prediction or delayed branching. Only after the branch is executed can instruction fetching of the target instruction begin. Dynamic scheduling reduces the delay between the branch and preceding instructions. Therefore, performance improvements for conditional branches accrue regardless of whether a particular branch is taken or not. More advanced branching methods tend to improve performance after the branch execution by reducing the hole in the pipeline between the branch and the following instruction. Consequently, special branching techniques can be successfully used to supplement a dynamic scheduling algorithm.

Quantitatively, with static scheduling only, each loop iteration in Figure 7 executes in 18 clock periods. With dynamic scheduling it takes only 11 clock periods. This is an improvement of about 60 percent. A more complete performance comparison can be found in Weiss and Smith,[4] where the Cray-1 architecture is used as a common base for comparing various instruction issue methods, including Tomasulo's algorithm. Instructions are statically scheduled prior to the performance analysis, and Tomasulo's algorithm is shown to provide a 60-percent performance improvement beyond that provided by static scheduling.

Data dependencies. Techniques (both static and dynamic) that reduce the performance impact of control dependencies tend to expose data dependencies as the next major performance obstacle.

(1) Dynamic scheduling around control dependencies causes instructions in the basic block following a branch to be eligible for issue and execution before all the instructions in the previous block have finished. Instructions from the two different basic blocks may have data dependencies that a static scheduler cannot resolve.

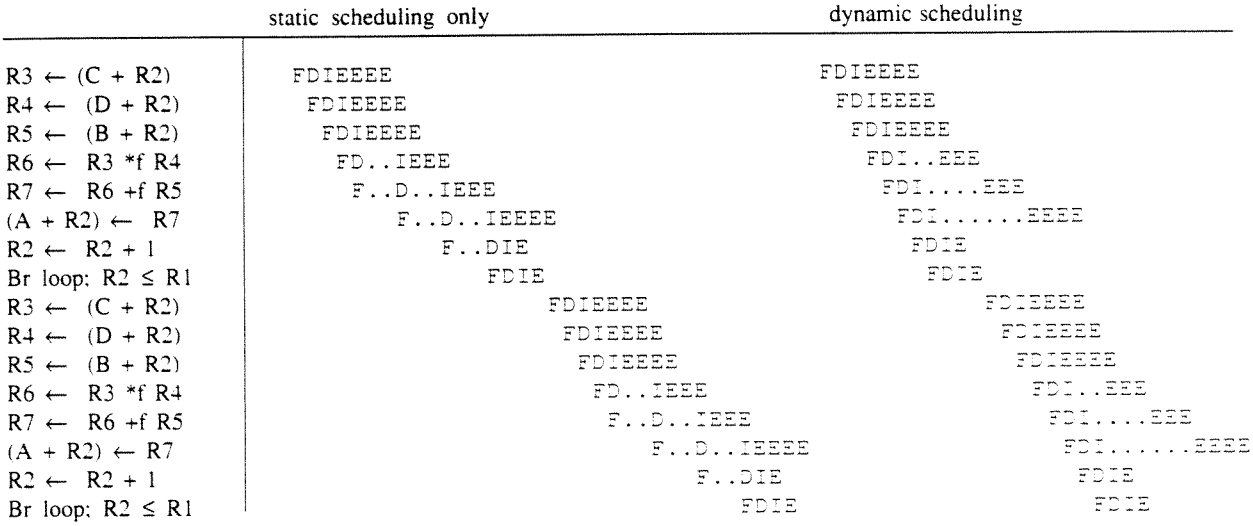
(2) Static scheduling methods that increase basic block size, such as trace scheduling and loop unrolling, increase the scheduling possibilities within the

```
      Do 1 I = 1,100
1     A(I) = B(I) + C(I)*D(I)
```

(a)

```
        R1 ← 100
        R2 ← 1
loop:   R3 ← (C + R2)
        R4 ← (D + R2)
        R5 ← (B + R2)
        R6 ← R3 *f R4
        R7 ← R6 +f R5
        (A + R2) ← R7
        R2 ← R2 + 1
        Br loop; R2 ≤ R1
```

(b)



(c)

Figure 7. Loop execution with and without dynamic instruction scheduling: (a) Fortran loop; (b) machine instructions; (c) pipeline flows.

basic block. Indeed, this is a major justification for using such methods. In the process, however, the number and variety of data dependencies is increased as well.

Data dependencies may involve data held either in registers or memory. There are three varieties of data dependencies[9]:

(1) flow dependencies, when a result operand of one instruction is used as a source operand for a subsequent one,
(2) output dependencies, when a result operand of one instruction is also used as a result operand for a subsequent one, and
(3) antidependencies, when a source operand of one instruction is used as a result operand for a subsequent one.

All three types of data dependencies can occur for either register data or memory data. Flow dependencies are inherent in the algorithm and cannot be avoided. Within a basic block, output and antidependencies involving registers can be avoided by static methods, specifically by reallocating registers (assuming that enough registers are available). However, dynamic scheduling of branch instructions may result in situations where output and antidependencies involving register instructions in different basic blocks must be resolved. For example, in Figure 7, one can conceive of dynamic issuing methods where the load from memory into $R3$ is ready to issue for the second loop iteration before the instruction using the old value of $R3$ (i.e., $R6 = R3$ *f $R4$) has issued in the first loop iteration. (While this may at first seem farfetched, the dynamic issuing method used by the ZS-1 frequently causes such a situation.) Dynamic register reallocation such as that provided by Tomasulo's algorithm can effectively deal with such situations.

Probably more difficult to deal with than dependencies involving registers are memory data dependencies. All three forms of data dependencies that can

occur with registers can also occur with memory data. For example, a load instruction following a store to the same memory address is a flow dependence, and the load may not be reordered ahead of the store instruction. On the other hand, if the load and store are to different addresses, there is no problem with reordering them.

Load and store instructions to scalar variables produce dependencies that are relatively simple to resolve. Furthermore, data dependencies involving scalars can often be removed by optimizing compilers.

Memory dependencies involving arrays and other data structures are much more difficult to schedule. The major problem with scheduling such memory dependencies lies in detecting them. Accesses involving arrays and other data structures use index values that can change at runtime, so complete information regarding the index values may be difficult or impossible for the compiler to discern. This is because the only address information available to the compiler is in symbolic form (such as array subscript expressions expressed in a high-level language).

Trace scheduling and loop unrolling combine high-level language basic blocks into larger machine-language ones. Significant performance advantages accrue if operations from the original basic blocks can be "blended" together as the instructions are scheduled. A good schedule tends to have load instructions near the top of a basic block with functional operations (adds and multiplies) in the middle and stores near the bottom. To achieve this with trace scheduling or loop unrolling, load instructions must often be scheduled ahead of stores belonging to different loop iterations. Consequently, the ability to determine data dependencies involving loads and stores from different HLL basic blocks is of critical importance. In the trace scheduling method, this process, known as *memory disambiguation*,[10] is a very important component. Memory disambiguation manipulates array subscript expressions in the form of systems of integer equations and inequalities to determine if load and store addresses to the same array can ever coincide in such a way that it is unsafe to interchange the load and store instructions.

Vectorizing compilers also concentrate on this kind of dependence, using very similar methods.[11] In fact, resolving memory data dependencies involving array references is a primary feature of vectorizing methods. Because of the excellent literature available on vectorizing compilers, the remaining discussion on memory data dependencies is phrased in terms of vectorizing compilers.

If the loop statement $A(I) = A(I) + C(I)$ is vectorized,

(1) the elements of $A(I)$ are loaded as a vector,
(2) the elements of $C(I)$ are loaded as a vector,
(3) there is a vector add of $A(I)$ and $C(I)$, and
(4) $A(I)$ is stored as a vector.

The vector load of the $A(I)$ followed later by the vector store amounts to a massive reordering of the loads and stores. That is, loads from $A(I)$ for later loop iterations are allowed to pass stores for earlier iterations. The loop $A(I) = A(I-1) + C(I)$ cannot be vectorized as just described, however, because the loads of $A(I-1)$ involve values just computed during the previous loop iteration. (Some simple linear recurrences may vectorize on certain computers that have special recurrence instructions. Throughout, this article uses simple examples for illustrative purposes; more complex linear and nonlinear recurrences could just as easily be used to prove the point.)

In two important classes of problems the static data dependence analysis done by compilers cannot achieve performance as high as dynamic runtime analysis. The first class involves analyses of such complexity that the compiler cannot safely determine if the subscripts are independent. These are referred to as *ambiguous subscripts*. The second class consists of code that has true load/store conflicts, but only for a few iterations of a loop.

An example of ambiguous subscripts is shown below:

```
      NL1=1
      NL2=2
      Do 1 I=1,N
      Do 2 J=1,N
2     A(J,NL1)=A(J-1,NL2)+B(J)
      NTEMP=NL1
      NL1=NL2
1     NL2=NTEMP
```

In the above example, the array A is split into two halves; old values in one half are used to compute new values in the other. When this process is finished, the two halves are switched (by interchanging $NL1$ and $NL2$). This kind of construct can be found in many Fortran programs. In fact, the original program from which Lawrence Livermore Kernel 8[11] was extracted used this kind of technique. A vectorizing compiler must determine that $NL1$ is never equal to $NL2$ to vectorize the loop. It is doubtful that any current vectorizing compiler has this capability. (However, a compiler might compile the example code as both vector and scalar and use a runtime test whenever the loop is encountered to determine which version should be used.) In general, $NL1$ and $NL2$ could be arbitrarily complex functions of any number of variables. This suggests the theoretical undecidability of vectorizable loops.

As another example, consider array references that frequently occur when handling sparse matrices. For example, $A(MAP(I)) = A(MAP(I)) + B(I)$, where MAP is an integer array of indices. This code can be vectorized (with vector gather/scatter instructions) if the compiler can determine that none of the $MAP(I)$ are equal. In most cases this is beyond a vectorizing compiler's capabilities.

As mentioned above, the second important class of problems where static analysis is inferior to dynamic analysis occurs when loops have true data dependencies, but for relatively few of the loop iterations. For example, consider the following nested loop:

```
      DO 1 J=1,N
      DO 1 I=1,N
1     X(I) = X(J) + Y(I)
```

For a particular execution of the inner loop, the value of $X(J)$ changes partway through, when $I=J$. The store into $X(I)$ when $I=J$ must follow all the loads of $X(J)$ for preceding inner loop iterations. This store must precede all subsequent loads of $X(J)$.

Another example is the case where subscripted subscripts are used; for example, the inner loop statement $X(M(K)) = X(N(K)) + W(K)$, where M and N have a small nonempty intersection. In this case, many, but not all, of the loads may pass stores. (The ones that may not pass occur when $M(I1) = N(I2)$ for $I2 < I1$.)

Dynamic scheduling: stunt boxes. The IBM 360/91 memory system[12] al-

```
      DO 1 J=1,N
      DO 1 I=1,N
1     X(I) = X(J) + Y(I)
```

(a)

```
R3 ← (R1)                 FDIEEEE
R4 ← (Y + R2)              FDIEEEE
R6 ← (Y+1 + R2)             FDIEEEE
R5 ← R3 +f R4                FDI..EEE
(X + R2) ← R5                 FDI....EEEE
R7 ← (R1)                      FDI....EEEE
R8 ← R6 +f R7                   FDI.......EEE
(X+1 + R2) ← R8                  FDI.........EEEE
R2 ← R2 + 1                       FDIE
Br loop; R2 ≤ R1                   FDIE
R3 ← (R1)                               FDI....EEEE
R4 ← (Y + R2)                            FDI....EEEE
R6 ← (Y+1 + R2)                           FDI....EEEE
R5 ← R3 +f R4                              FDI......EEE
(X + R2) ← R5                               FDI........EEEE
R7 ← (R1)                                    FDI........EEEE
R8 ← R6 +f R7                                 FDI..........EEE
(X+1 + R2) ← R8                                FDI............EEEE
R2 ← R2 + 1                                     FDIE
Br loop; R2 ≤ R1                                 FDIE
```

(b)

	no memory conflicts	with memory conflict

```
R3 ← (R1)                 FDIEEEE                    FDIEEEE
R4 ← (Y + R2)              FDIEEEE                    FDIEEEE
R6 ← (Y+1 + R2)             FDIEEEE                    FDIEEEE
R5 ← R3 +f R4                FDI..EEE                   FDI..EEE
(X + R2) ← R5                 FDI....EEEE                FDI....EEEE
R7 ← (R1)                      FDIEEEE                    FDI....EEEE
R8 ← R6 +f R7                   FDI...EEE                  FDI.......EEE
(X+1 + R2) ← R8                  FDI.....EEEE               FDI.........EEEE
R2 ← R2 + 1                       FDIE                       FDIE
Br loop; R2 ≤ R1                   FDIE                       FDIE
R3 ← (R1)                               FDIEEEE                    FDIEEEE
R4 ← (Y + R2)                            FDIEEEE                    FDIEEEE
R6 ← (Y+1 + R2)                           FDIEEEE                    FDIEEEE
R5 ← R3 +f R4                              FDI..EEE                   FDI..EEE
(X + R2) ← R5                               FDI....EEEE                FDI....EEEE
R7 ← (R1)                                    FDIEEEE                    FDIEEEE
R8 ← R6 +f R7                                 FDI...EEE                  FDI...EEE
(X+1 + R2) ← R8                                FDI.....EEEE               FDI.....EEEE
R2 ← R2 + 1                                     FDIE                       FDIE
Br loop; R2 ≤ R1                                 FDIE                       FDIE
```

(c)

Figure 8. The effects of load/store reordering: (a) a nonvectorizable loop with a memory hazard; (b) pipeline execution without dynamic memory reordering; (c) pipeline execution with dynamic memory reordering.

lowed load instructions to pass store instructions after they had been issued. The memory system allowed the dynamic reordering of loads and stores as long as a load or a store did not pass a store to the same address. Store instructions could issue before their data were actually ready. There was a queue for store instructions waiting for data. Load instructions entering the memory system had their addresses compared with the waiting stores. If there were no matches, the loads were allowed to pass the waiting stores. If a load matched a store address, it was held in a buffer. When the store data became available, they were automatically bypassed to the waiting load as well as being stored to memory.

The CDC 6600 memory system was simpler than in the IBM 360/91 and allowed some limited reordering of memory references to increase memory throughput in a memory system that used no cache and had a relatively slow interleaved main memory. In the CDC 6600, the unit used for holding memory references while they were waiting for memory was called the *stunt box*. The CDC 6600 stunt box did not actually allow loads to pass waiting stores, but just as the term "scoreboard" has taken on a more general meaning than the way it was used in the 6600, the term "stunt box" has come to mean any device that allows reordering of memory references in the memory system.

Figure 8a shows a Fortran loop, unrolled twice, that has a memory data dependence that inhibits vectorization. Because of this, the compiler is restricted from moving certain loads above stores that may potentially be to the same address. Figure 8b shows pipeline usage with dynamic scheduling but with no provision for loads to pass stores via a stunt box. Figure 8c shows pipeline usage with dynamic scheduling and a memory stunt box. The left pipeline flow is for a sequence of code where $I<J$ so there are no dependencies involving $X(I)$ and $X(J)$. The right pipeline flow includes the case where I "passes" J so that there is temporarily a memory hazard preventing a load from $X(J)$ from passing a store to $X(I)$ when $I=J$. This results in a temporary glitch in the execution of memory instructions, but there is no overall time penalty as the code sequence continues past the point where $I=J$. By inspecting the timing diagrams, the pipeline without dynamic load/store reordering can execute one loop iteration (two iterations in the original rolled loop) every 17 clock periods. With dynamic load/store reordering, it executes an iteration every 13 clock periods. Without any dynamic scheduling at all, each loop iteration takes 25 clock periods (this case is not illustrated). Using full dynamic scheduling results in an almost two-times performance improvement over static scheduling alone.

The viability of dynamic scheduling

As pointed out, dynamic scheduling was used in large-scale computers in the 1960s and has not been used to any appreciable extent since. One can only speculate about the reasons for abandonment of dynamic scheduling in production high-performance machines. Following are some of the possible reasons:

(1) Increased difficulty in hardware debugging: a hardware failure can cause errors that are highly dependent on the order of instruction issuing for many clock periods prior to the actual detection of the error. This makes error reproducibility difficult. The fault diagnosis problem was compounded in the IBM 360/91 and CDC 6600 because discrete logic was used, and diagnostic resolution had to be very fine.

(2) Longer clock period: dynamic instruction issuing can lead to more complex control hardware. This carries with it potentially longer control paths and a longer clock period.

(3) Advances in compiler development: initially, dynamic scheduling permitted simple compilers that required relatively little static scheduling capability. However, improved compilers with better register allocation and scheduling could realize some (but certainly not all) of the benefits of dynamic issuing.

Why consider dynamic scheduling today, when it was passed by years ago? First, very large scale integration parts and extensive use of simulation in today's computers alleviate many of the debugging and diagnosis problems present 20 years ago. Simulation can be used to find design errors, and hardware faults need only be located to within a (large) replaceable unit. That is, if a fault is detected in a CPU's instruction issue logic, the entire CPU, or at least a large part of it, can be replaced; there is no need for extensive and detailed fault location methods.

Second, it is possible to use methods that selectively limit the generality of dynamic scheduling so that significant performance benefits can be realized while keeping the control logic simpler than in the CDC 6600 and IBM 360/91. Techniques of this type have been successfully used in the ZS-1 and are described in later sections.

Third, both compiler scheduling and processor design are much more mature, and most of the big performance gains in these areas have probably been made. Consequently, the gains achievable by dynamic scheduling may appear more attractive today than they did 20 years ago.

The ZS-1

The Astronautics ZS-1 is a recently developed, high-speed computer system targeted at scientific and engineering applications. The ZS-1 central processor is constructed of transistor-transistor logic-based technology and has no vector instructions, but makes extensive use of dynamic instruction scheduling and pipelining to achieve one-third the performance of a Cray X-MP1.

A block diagram of a ZS-1 system is shown in Figure 9. The ZS-1 is divided into four major subsystems: the central processor, the memory system, the I/O system, and the interconnection network. In its maximum configuration, the ZS-1 contains one gigabyte of central memory, and an I/O system consisting of up to 32 input-output processors. Unix is the primary operating system.

The ZS-1 uses a decoupled architecture[13] that employs two instruction pipelines to issue up to two instructions per clock period. One of the instruction streams performs the bulk of fixed-point and memory addressing operations, while the other performs floating-point calculations.

To support the two instruction streams, the decoupled architecture of the ZS-1 provides two sets of operating registers. A set of thirty-one 32-bit A registers is used for all memory address computation and accessing, and a second set of thirty-one 64-bit X registers is used for all floating-point operations. The A and X registers provide fast temporary storage for 32-bit integers and 64-bit floating-point numbers, respectively.

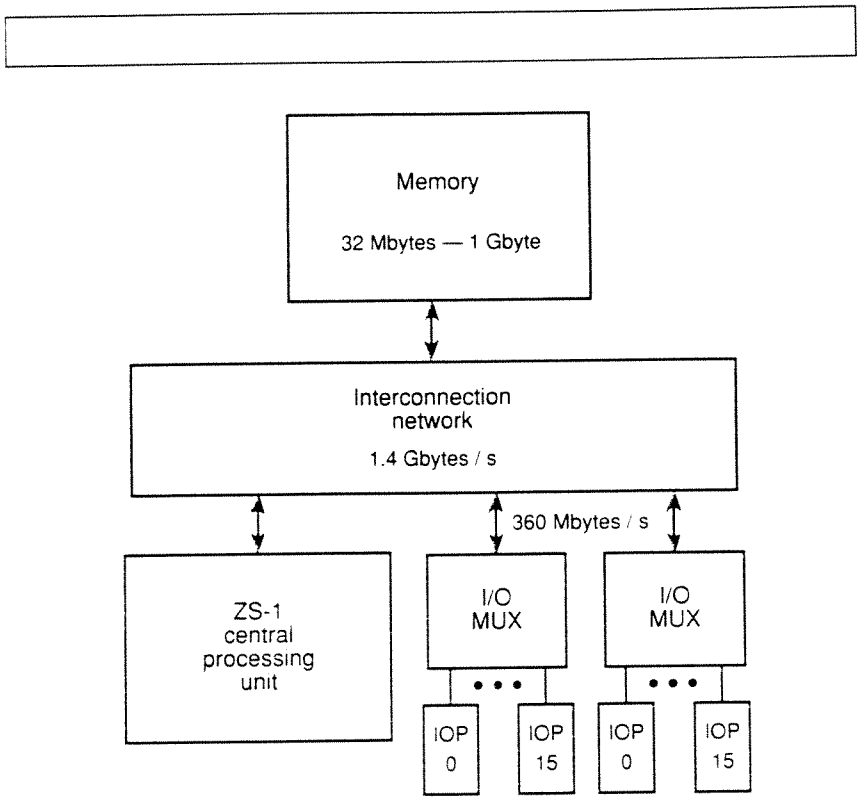


Figure 9. Overall diagram of the ZS-1 system.

```
      Do 10 I = 1, 100
10    A(I) = B(I)*C(I) + D(I)
```

(a)

```
S1:         A5 ← 0                      .loop count
S2:         A6 ← A - 8                  .load initial pointer to A
S3:         A7 ← B - 8                  .load initial pointer to B
S4:         A8 ← C - 8                  .load initial pointer to C
S5:         A9 ← D - 8                  .load initial pointer to D
S6: loop:   A5 ← A5 + 1                 .increment A5
S7:         B, A0 ← (A5 == 100)         .compare =, set Branch Flag
S8:         XLQ ← (A7 = A7 + 8)         .load next element of B
S9:         XLQ ← (A8 = A8 + 8)         .load next element of C
S10:        XLQ ← (A9 = A9 + 8)         .load next element of D
S11:        X2 ← XLQ                    .copy B element into X2
S12:        X3 ← X2 *f XLQ              .multiply B and C
S13:        XSQ ← XLQ +f X3             .add D; result to XSQ
S14:        (A6 ← A6 + 8) = XSQ         .store result into A
S15:        Br loop; B==0               .branch on false to "loop"
```

(b)

Figure 10. A Fortran loop and its ZS-1 compilation: (a) Fortran source; (b) machine language version of the loop.

A distinctive feature of the ZS-1 is the use of architectural queues for communication with main memory. There are two sets of queues. One set consists of a 15-element A load queue (ALQ) and a 7-element A store queue (ASQ). These A queues are used in conjunction with the 32-bit A registers. The other set of queues consists of a 15-element X load queue (XLQ) and a 7-element X store queue (XSQ). These X queues are used in conjunction with the 64-bit X registers.

Instructions. Instruction formats are reminiscent of those used in the CDC 6600/7600 and Cray-1. There is an opcode, and operands specified by i, j, and k fields. The j and k fields typically specify input operands and the i field specifies the result. The i, j, and k operands may be either general-purpose registers or queues. A designator of 31 in the j or k field indicates that the first element of the load queue is used as a source operand. A designator of 31 in the i field indicates that the result is placed into the store queue. In this way, queue operands can be easily intermixed with register operands in all instruction types. The opcode determines whether A registers and queues or X registers and queues are to be operated upon.

The ZS-1 architecture is best understood by examining a sequence of machine code. Figure 10a contains a simple Fortran loop, and Figure 10b contains a compilation into ZS-1 machine instructions. The instruction *S1* initializes fixed-point register A5, which is used as the loop counter. Then instructions *S2* through *S5* initialize A6 through A9 to point to the arrays accessed in the loop. The pointers are offset by -8 because byte addressing is used, and the load and store instructions use pre-autoincrementing.

In the loop body, instructions *S6* and *S7* increment the loop counter and test it to see if it has reached the upper limit. The test is done by a compare instruction, which generates a Fortran Boolean result that is placed in A0 (because A0 is defined to always hold constant 0, this is equivalent to discarding it), and it also sets the architectural branch flag, *B*, to the result of the comparison. The branch flag will be tested by the conditional branch instruction that terminates the loop.

Instructions *S8* through *S10* load the elements from arrays *B*, *C*, and *D*. These memory operands are automatically

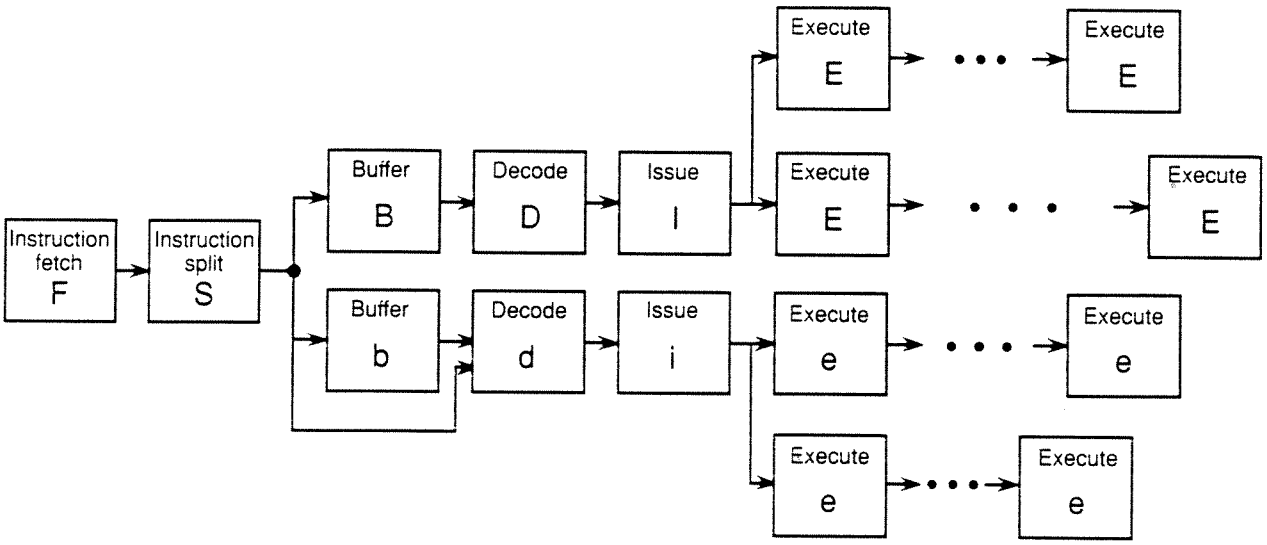


Figure 11. The ZS-1 processor pipelines.

placed in the XLQ. Because the destination of the load instructions are queues implied by the opcode, the *i* field of load and store instructions may be used to specify a result register for the effective address add. This makes autoincrementing memory operations particularly easy to implement.

Then instructions *S*11 through *S*13 copy the memory data from the XLQ and perform the required floating-point operations. These are generic floating-point instructions with register designator 31 used wherever there is a queue operand. The floating-point add is of particular interest because it not only uses the XLQ as its *j* operand, but it also uses the XSQ for its result. The store instruction, *S*14, generates the store address in array *A*.

Decoupled implementation

A simplified block diagram of the ZS-1 CPU pipelines is shown in Figure 11. The pipeline segments include an instruction fetch stage where instruction words are read from a 16-kilobyte instruction cache. An instruction word may contain either one 64-bit instruction (used for conditional branches and loads and stores with direct addressing) or two 32-bit instructions (by far the most common case). The next pipeline segment is the "splitter" stage, where instruction words are split into instructions and sent to the two instruction pipelines. In one cycle, the instruction word is examined by the A instruction pipeline and the X instruction pipeline to see whether it contains one or two instructions and to determine whether the instructions are:

(1) X unit instructions,
(2) A unit instructions,
(3) branch instructions or system call/return instructions.

Branch and system call/return instructions are held and executed in the splitter stage. Instructions belonging to the first two classes are sent to an instruction buffer at the beginning of the appropriate instruction pipeline. Up to two instructions are forwarded to the instruction pipelines per clock period.

The instruction buffer in the X instruction pipeline can hold 24 instructions. The buffer in the A instruction pipeline is four instructions deep and can be bypassed. The very deep X instruction buffer allows the A instruction pipeline to issue many instructions in advance of the X instruction pipeline. The A instruction buffering is intended to reduce blockages of the splitter. The bypass allows instructions to move quickly up the A pipeline when it is empty; for example, following some branches.

In the instruction pipelines, pipeline segments are:

(1) the buffer stage where instructions are read from the instruction buffers,
(2) the decode stage where instructions are decoded, and
(3) the issue stage where instructions are sent to functional units for execution.

At the issue stage a simple Cray-1-like issuing method allows instructions to begin execution in strict program sequence. For example, if an instruction uses the result of a previously issued, but unfinished, instruction, it waits at the

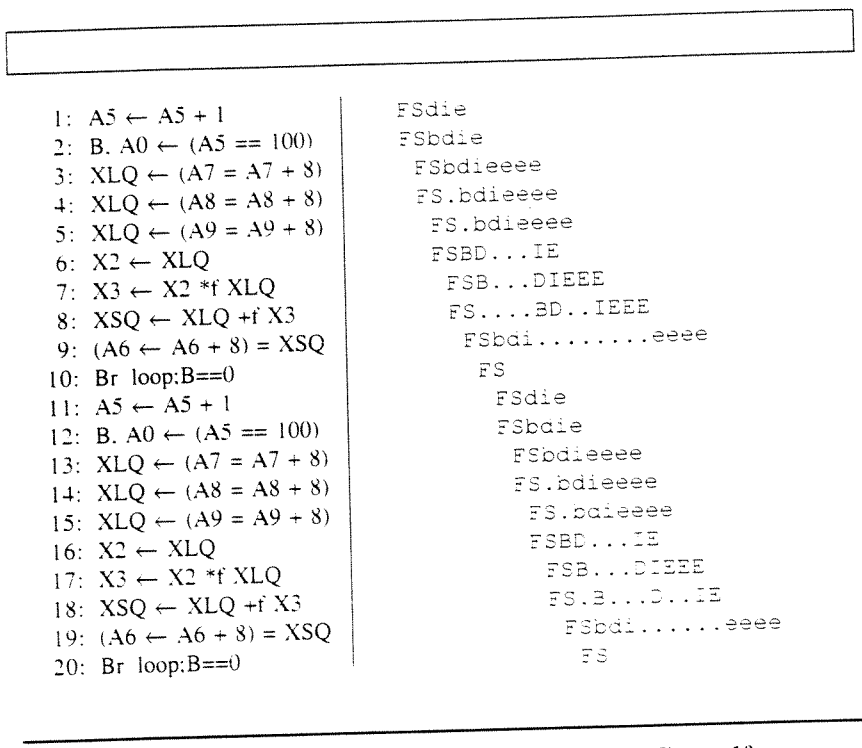


Figure 12. The processing of two iterations of the loop in Figure 10.

issue register until the previous instruction completes.

At the time an instruction is issued from one of the pipelines, operand data are read from the appropriate register files and/or queues. After issue, the instruction begins execution in one of the parallel functional units. The primary functional units for the fixed-point A instructions are: a shifter, an integer adder/logical unit, and an integer multiplier/divider. The primary functional units for the floating-point X instructions are: an X logical unit, a floating-point adder, a floating-point multiplier, and a floating-point divider. Data can be copied between A and X registers via the copy unit.

The ZS-1 uses several dynamic scheduling techniques:

(1) The architectural instruction stream is split in two, with each resulting stream proceeding at its own speed. This not only permits the ZS-1 to sustain an instruction issue rate of up to two instructions per clock period, but it allows the memory access instructions to dynamically schedule ahead of the floating-point instructions. In addition, the scoreboard issue logic at the end of each of the instruction pipelines remains as simple as in the Cray-1; no instruction reordering is done within a single pipeline.

(2) Branch instructions are held and executed at the splitter. This is accomplished by decomposing branch operations into their two fundamental components: comparing register values and transferring control. Compare instructions are detected in the splitter and set the branch flag to "busy" as they pass through. If a branch instruction encounters a "busy" branch flag, it waits in the splitter until the flag is set by the compare instruction. Compare instructions are subsequently issued from the appropriate instruction pipeline, depending on whether fixed- or floating-point data are tested. A comparison sets the branch flag when it completes. Consequently, branch instructions do not directly read register values, they simply test the branch flag. This mechanism allows branches to be executed dynamically ahead of instructions that precede them in the instruction stream. Furthermore, executing branches very early in the pipeline reduces, and in some cases eliminates, any resulting "hole" in the pipeline.

(3) Using queues for memory operands provides an elastic way of joining the memory access and floating-point functions. This elasticity allows the memory access function to schedule itself dynamically ahead of the floating-point operations. This can also be viewed as a way of achieving dynamic register allocation. That is, each load or store instruction dynamically allocates a new value of "register" 31.

(4) Store instructions merely generate store addresses; they do not wait for the store data before issuing. In the memory system is a stunt box containing two queues, one for load addresses and the other for store addresses. Store addresses wait in their queue until a corresponding data item appears in a store data queue (one for fixed-point data, one for floating-point). Load addresses may pass store instructions that are waiting for their data. Memory hazards are checked by comparing load and store addresses so that loads do not pass stores to the same address.

Figure 12 illustrates the processing of two iterations of the loop in Figure 10. As in earlier examples, only the instructions within the loop body are shown. Many of the pipeline stages are the same as in previous examples, and there are two new stages. One of the new stages is the splitter stage, the other is the buffer stage at the beginning of each of the instruction pipelines (although the buffer in the fixed-point pipeline can be bypassed). Because there are actually two distinct instruction pipelines following the splitter, for all stages after the splitter lowercase letters are used to denote fixed-point instructions, and uppercase letters are used for floating-point instructions. Pipeline lengths are the same as in the previous examples to more clearly demonstrate the principles at work. To summarize, the letters labeling pipeline stages have the following meanings:

F denotes the instruction is in the fetch stage,

S indicates the instruction word is processed at the splitter,

B or *b* indicates the instruction is read from an instruction buffer,

D or *d* indicates the instruction is decoded,

I or *i* indicates the instruction is issued for execution,

E or *e* indicates the instruction is executed.

The first instruction (A5 = A5 + 1) is split at time 0, decoded at time 1 (the buffer is bypassed), issued at time 2, and executed at time 3.

The second instruction is split at the same time as the first and is read from the buffer at time 1. Note that this second instruction sets the branch flag. The next three instructions follow a similar sequence for processing.

The sixth instruction is the first X instruction. It is split at time 2, is read from the X instruction buffer at time 3, and is decoded at time 4. It must then wait for data from the XLQ before continuing.

The seventh and eighth instructions perform the required floating-point operations in sequence, with the eighth putting its result in the XSQ for storage to memory.

The ninth instruction generates the store address for the preceding one. It is an A instruction that issues at time 7. It passes through four clock periods of execution while the address is generated and translated. It then waits while the preceding floating-point addition completes. Then the result is stored to memory.

The tenth and final instruction in the loop body is the conditional branch. It is detected and executed in the splitter stage. Note that, in this example, all but one of the clock periods required for the conditional branch are hidden; instruction issuing proceeds without interruption in the fixed-point pipeline.

The second loop iteration follows the first in Figure 12, and all subsequent loop iterations are similar to it. In this example, steady state performance is determined by the rate at which the fixed-point operations can be issued. In cases where floating-point dependencies are more severe, steady state performance is determined by the floating-point pipeline.

By extrapolating data from the diagram we can see that up to three iterations of the loop are in some phase of processing simultaneously. This is a clear example of the ability of dynamic scheduling to fetch and execute instructions beyond basic-block boundaries. During many clock periods eight or more instructions are processed in parallel (not counting those blocked in the pipeline).

The example just given is intended to illustrate dynamic scheduling aspects of the ZS-1 implementation. In fact, the ZS-1 compilers automatically unroll loops. The degree of unrolling is a function of

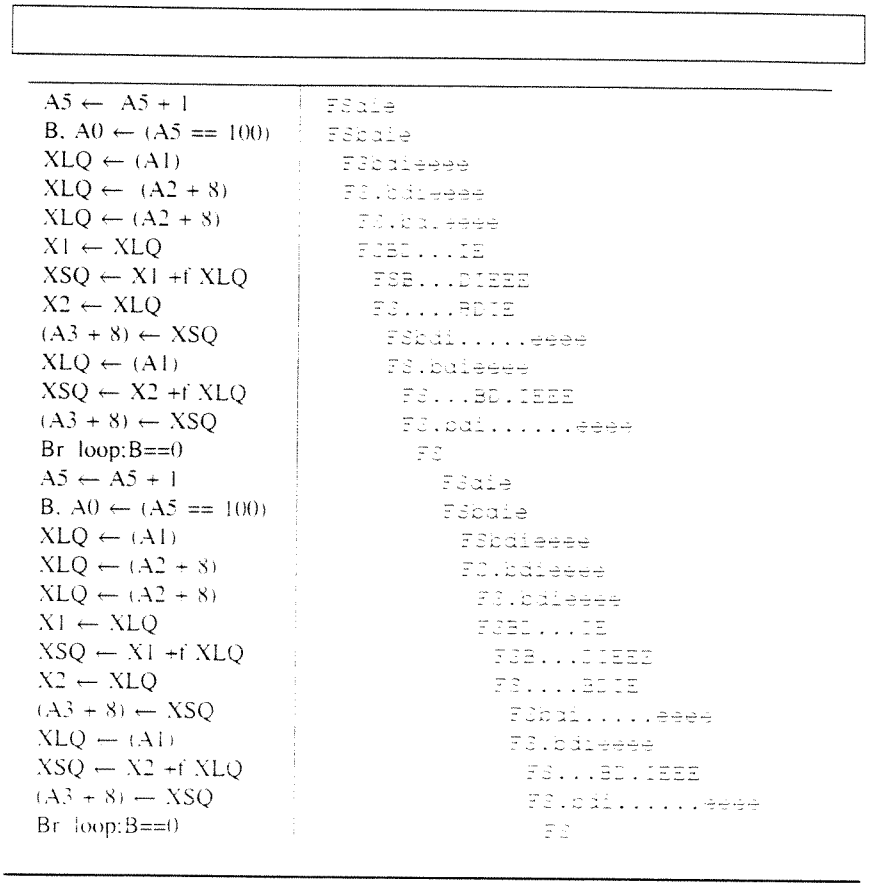


Figure 13. ZS-1 execution of a nonvectorizable loop.

the size of the loop: for a simple loop as illustrated in Figure 12, the Fortran compiler unrolls the loop body eight times. When this is done, and instructions are rescheduled using the resulting larger basic blocks, vector levels of performance can be achieved. For example, if the loop of Figure 12 is unrolled for the ZS-1, a load or store instruction issues during 94 percent of the clock periods. In other words, the memory path is busy 94 percent of the time. Many vector processors, including the Cray-1 and Cray-2, have their vector performance limited by their ability to perform only one load or store operation per clock period. For practical purposes, this is the same bottleneck that ultimately limits ZS-1 performance, and vector instructions would provide no significant performance benefit.

On the other hand, when faced with loops like those in Figure 8 that do not vectorize, the ZS-1 can achieve vector performance levels. This is illustrated in Figure 13, where the loop is unrolled two times to permit comparison with the earlier example.

This example illustrates the iterations where no memory conflict exists. When *I=J* and there is a memory conflict, a slight perturbation that lasts for only two loop iterations occurs. The delays caused by this perturbation are completely hidden by the dynamic scheduling, however. The total time to execute an inner loop in this example is nine clock periods. Because this loop is not vectorizable, we saw earlier that it would take a static-scheduled Cray-1-like machine 17 clock periods to execute the inner loop once. Note also that this example illustrates a situation where instruction issuing in the fixed-point pipeline is completely uninterrupted by the conditional branch executed at the splitter.

ZS-1 performance

All the examples in this article have used a fixed set of pipeline lengths to make comparisons possible. The production ZS-1 models operate at a 45-nanosecond clock period and use VLSI floating-point chips that provide pipeline lengths of three clock periods for 64-bit floating-point multiplication and addi-

tion. (The prototype systems described in an earlier work[14] used standard TTL-based floating-point units with latencies about twice as long. In addition, the production systems use a true divide algorithm as opposed to the reciprocal approximation method described in that work.[14]) The memory pipeline is conservatively designed and consumes eight clock periods for data found in the 128-kilobyte data cache. The decision to use a data cache was made relatively late in the design process. Consequently, the cache was added in series with the address translation unit, with a board-crossing between. A more parallel address translation/cache system would probably have only half the latency, or about four clock periods.

For performance comparisons, we use the 24 Livermore Kernels[11] because they are extracted from real programs and contain a realistic mix of both vector and scalar code. To summarize performance in millions of floating-point operations per second (Mflops), we use the harmonic mean, which is the total number of operations (scaled to be the same for each kernel) divided by the total time. For the 24 double-precision kernels, using the original Fortran (with no added compiler directives, as are often used to assist vector machines), the ZS-1 performs at 4.1 Mflops.

The Multiflow Trace 7/200[10] is constructed of similar-speed technology (about 3.5 nanoseconds per gate) as the ZS-1 and uses state-of-the-art trace scheduling compiler technology. On the 24 Livermore Kernels the Multiflow Trace operates at 2.3 Mflops. As a final comparison, the Cray X-MP can execute the Livermore Kernels at 12.3 Mflops.[11]

Of course, all the above performance numbers are very much a function of the compiler used. It is expected that later compiler versions for any of the machines, including the ZS-1, could lead to improved performance.

Additional reading on dynamic instruction scheduling

The book by Thornton[4] describing the CDC 6600 is a classic, but unfortunately it is out of print. Considerable detail on the scoreboard design can be found in the Thornton and Cray patent, U.S. patent no. 3,346,851. The IBM 360/91 is described in a series of papers in the January 1967 issue of the *IBM Journal of Research and Development*. A recent book by Schneck[15] contains a discussion of several pipelined machines, including both the CDC 6600 and the IBM 360/91. The book by Kogge[16] is another excellent reference. Recent research in pipelined computers has concentrated on static scheduling, rather than dynamic scheduling. A notable exception is work being undertaken by Hwu and Patt[17] that contains interesting enhancements to Tomasulo's algorithm.

Dynamic instruction scheduling improves performance by resolving control and data dependencies at runtime using real data.

Static scheduling must rely on predictions or worst-case assumptions made at compile time. Consequently, there will always be situations where runtime scheduling can outperform static scheduling.

It is generally true that simple, streamlined instruction sets reduce hardware control complexity and tend to produce faster pipelined implementations than complex instruction sets. However, it does not logically follow that less hardware control complexity leads to better performance. When complexity is directed toward greater instruction functionality, performance often does suffer, but carefully chosen control complexity can also be directed toward greater performance.

There is a performance/complexity trade-off curve, but it is not clear that the maximum performance point occurs at the minimal complexity end of the curve. It may very well be that some additional control complexity can be effectively used to increase performance. The risk, of course, in increasing control complexity is that performance advantages can be offset by slower control paths and an increased clock period. The ZS-1 is a successful attempt at achieving improved performance levels by using an architecture that naturally leads to multiple instruction streams and dynamic instruction scheduling. Despite using dynamic scheduling, the ZS-1's 45-nanosecond clock period is the fastest we know of for standard TTL-based machines.

The result is an architecture that can achieve vector levels of performance on highly parallel, vectorizable code. Furthermore, and more importantly, similar performance levels can be achieved with many less parallel, nonvectorizable codes. This is done by mixing advanced static scheduling techniques, based on loop unrolling (and simple forms of trace scheduling), with advanced dynamic scheduling techniques. It is important to note the "orthogonality" of advanced static scheduling techniques and dynamic scheduling. Static scheduling can go a long way toward high performance, but this article has shown that dynamic scheduling can extend performance beyond that achievable with static scheduling alone.

A final observation is that compiler complexity and compilation times are often considered to be of little consequence when discussing hardware control complexity/performance trade-offs. This is not absolutely true, however. Mature compilers take considerable time to construct, and in many program development environments compilation times using advanced static scheduling methods can become excessively long. Using dynamic scheduling provides good performance on nonoptimum code. This means that immature compilers, or very fast compilers with reduced optimization, can come close to achieving the full potential of a computer that uses dynamic scheduling methods. □

Acknowledgments

The ZS-1 has resulted from the hard work o many people; only a few are acknowledge here. The original architecture was specifie by Greg Dermer, Tom Kaminski, Michae Goldsmith, and myself. The processor desig was completed by Brian Vanderwarn, Stev Klinger, Chris Rozewski, Dan Fowler, Keit Scidmore, and Jim Laudon. Other principals i the hardware design effort were Bob Niem Harold Mattison, Tom Staley, and Jerry Rab In the software area, Don Neuhengen manage the operating system development, Greg Fi cher managed compiler development, ar Kate Murphy managed applications develo ment. Finally, I would like to acknowledge th constant interest and encouragement provid by Ron Zelazo, president of the Astronauti Corporation of America.

References

1. J. Hennessy, "VLSI Processor Architecture," *IEEE Trans. on Computers*, Vol. 33, Dec. 1984, pp. 1,221-1,246.

2. D.A. Patterson, "Reduced Instruction Set Computers," *Comm. ACM*, Vol. 28, Jan. 1985, pp. 8-21.

3. D.W. Clark, "Pipelining and Performance in the VAX 8800 Processor," *Proc. 2nd Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS II), Oct. 1987, pp. 173-177.

4. J.E. Thornton, *Design of a Computer -- The Control Data 6600*, Scott, Foresman and Co., Glenview, Ill., 1970.

5. D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM J. Research and Development*, Jan. 1967, pp. 8-24.

6. R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Development*, Jan. 1967, pp. 25-33.

7. J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, Vol. C-30, July 1981, pp. 478-490.

8. S. Weiss and J.E. Smith, "Instruction Issue Logic in Pipelined Supercomputers," *IEEE Trans. Computers*, Vol. C-33, Nov. 1984, pp. 1,013-1,022.

9. D.A. Padua and M.J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Comm. ACM*, Vol. 29, Dec. 1986, pp. 1,184-1,201.

10. R.P. Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. Computers*, Vol. 37, Aug. 1988, pp. 967-979.

11. F.H. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," Research Report, Lawrence Livermore Laboratories, Dec. 1986.

12. L.J. Boland et al., "The IBM System/360 Model 91: Storage System," *IBM J.*, Jan. 1967, pp. 54-68.

13. J.E. Smith, S. Weiss, and N. Pang, "A Simulation Study of Decoupled Architecture Computers," *IEEE Trans. Computers*, Vol. C-35, Aug. 1986, pp. 692-702.

14. J.E. Smith et al., "The ZS-1 Central Processor," *Proc. ASPLOS II*, Oct. 1987, pp. 199-204.

15. P.B. Schneck, *Supercomputer Architectures*, Kluwer Acacemic Publishers, Norwell, Mass., 1988.

16. P.M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, New York, 1981.

17. W. Hwu and Y.N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," *Proc. 13th Ann. Symp. Computer Architecture*, June 1986, pp. 297-307.

James E. Smith has been with the Astronautics Corporation Technology Center in Madison, Wis., since 1984. He is system architect for the ZS Series of computer systems and has performed research and development activities primarily directed toward the ZS-1, a large-scale scientific computer system. He is currently involved in the specification and design of follow-on products.

Smith received BS, MS, and PhD degrees from the University of Illinois in 1972, 1974, and 1976, respectively. Since 1976, he has been on the faculty of the University of Wisconsin at Madison, where he is an associate professor (on leave) in the Department of Electrical and Computer Engineering. He is a member of the IEEE Computer Society, IEEE, and ACM.

Readers may contact the author at Astronautics Corp. of America, Technology Center, 4800 Cottage Grove Rd., Madison, WI 53716-1387.