

The ZS-1 Central Processor

J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski,
D. L. Fowler, K. R. Scidmore, and J. P. Laudon

Astronautics Corporation of America
Madison, Wisconsin

Abstract

The Astronautics ZS-1 is a high speed, 64-bit computer system designed for scientific and engineering applications. The ZS-1 central processor uses a decoupled architecture, which splits instructions into two streams—one for fixed point/memory address computation and the other for floating point operations. The two instruction streams are then processed in parallel. Pipelining is also used extensively throughout the ZS-1.

This paper describes the architecture and implementation of the ZS-1 central processor, beginning with some of the basic design objectives. Descriptions of the instruction set, pipeline structure, and virtual memory implementation demonstrate the methods used to satisfy the objectives. High performance is achieved through a combination of static (compile-time) instruction scheduling and dynamic (run-time) scheduling. Both types of scheduling are illustrated with examples.

1. Introduction

1.1. System Overview

The Astronautics ZS-1 is a high-speed, computer system targeted at scientific and engineering applications. It is built around a high performance, 64-bit central processor that executes compiled FORTRAN at about a third the rate of a CRAY-1. In its maximum configuration, the ZS-1 contains 256 Megabytes of central memory, and an I/O system consisting of up to 32 input-output processors. UNIX¹ is the primary operating system. The ZS-1 is constructed primarily of off-the-shelf Advanced Schottky SSI and MSI logic circuits and 256k dynamic MOS memory integrated circuits.

As shown in Fig. 1, the ZS-1 is divided into four major subsystems: the Central Processor, the Memory system, the I/O system, and the Interconnection Network. This paper will concentrate on the architecture and implementation of the Central Processor.

¹ UNIX is a registered trademark of AT&T Bell Laboratories

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

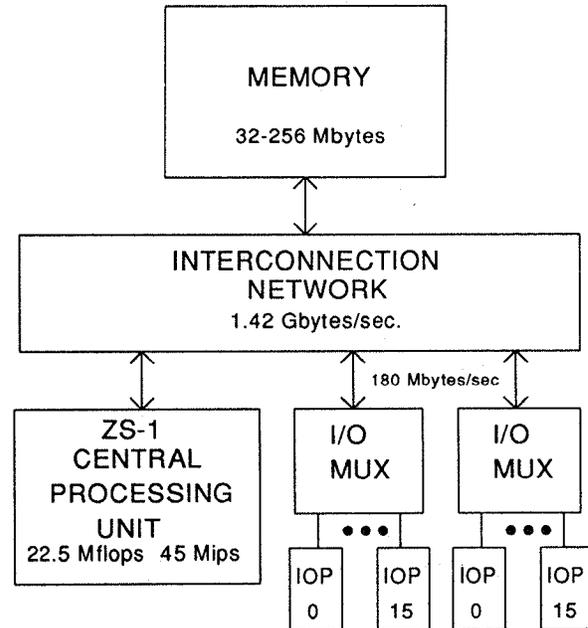


Fig. 1. Overall Diagram of the ZS-1 System.

1.2. CPU Design Objectives

In a system such as the ZS-1 there are many (sometimes conflicting) design objectives. The following four are particularly relevant in this paper.

1) A well-balanced general purpose scientific processor should provide high performance on programs with relatively little or hard-to-find parallelism [WOR81]. To satisfy this objective, we use a decoupled architecture [SMIT86], a high degree of pipelining with a fast clock, and a simple, CDC6600/7600-style instruction set [THOR70].

2) Because of the considerable resources being directed at developing vector-oriented algorithms, the processor should be able to provide performance on parallel programs similar to that achievable by a vector processor.

3) The processor should be designed in conjunction with the compilers. The two should work together well. In the ZS-1 this is most apparent in the way instruction scheduling is shared between the compiler and the hardware.

4) The processor should be able to support a modern programming environment, while maintaining high performance. A primary example is the support of virtual memory; the virtual memory architecture must be carefully designed in order to keep performance losses at a minimum.

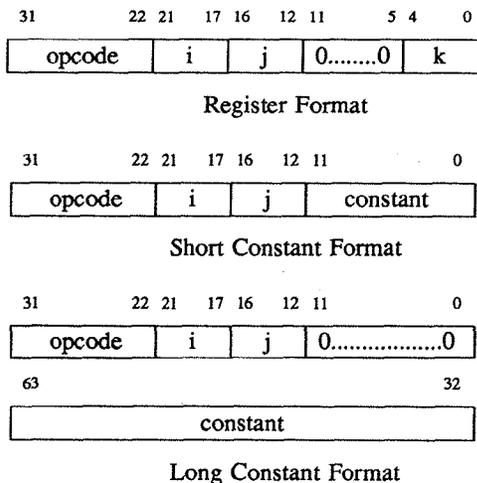
2. Architecture

The decoupled architecture of the ZS-1 provides two sets of operating registers for holding operands, addresses, and control information in the CPU. 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.

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.

2.1. Instruction Set

The three ZS-1 instruction formats are shown in Fig. 2.



opcode: The opcode specifies the operation to be performed.

i, j, and k: The j and k fields specify source operands, and the i field specifies the destination.

constant: In the short and long constant formats a 12-bit or 32-bit constant is used in place of the k operand.

Fig. 2. ZS-1 Instruction Formats

Operands specified by i, j, and k fields 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 to be used as a source operand. A designator of 31 in the i field indicates that the result is to be placed into the store queue. The opcode determines whether A registers and queues or X registers and queues are to be operated upon. All the long constant instructions must be aligned on a word boundary.

2.2. Data Types and Operations

All major data paths, floating point registers, and functional units are 64 bits wide. The ZS-1 uses IEEE standard 32-bit and 64-bit floating point formats, but the arithmetic is simplified in the interest of high performance. In addition to the full and half-precision floating point data, 32-bit, 16-bit, and 8-bit fixed point data types are implemented. Memory is addressed with 32-bit virtual addresses that have byte resolution.

Fixed point operations include unsigned and integer arithmetic, logical operations, and addressing for loads and stores to and from all the queues. Operations defined on the floating point data types are floating point arithmetic, and logical operations.

2.3. Load and Store Instructions

There are load and store instructions that use each of the three instruction formats. In each case, the effective address is computed by adding the contents of register Aj and register Ak (or a constant). The computed effective address is placed in Ai. Therefore, all loads and stores can be "auto-incrementing".

For load instructions, the computed effective address is used to load the appropriate queue with data from memory. For store instructions, the computed effective address is used to store the first element of the appropriate store queue to memory.

2.4. Conditional Branches

A subset of the compare instructions explicitly set a conditional Branch Flag (B) held in the instruction fetch unit. The *Branch on Condition True* and *Branch on Condition False* instructions test the value in B. If the specified branch condition holds, control is transferred to the instruction at the address contained in the constant field of the instruction.

2.5. Example

Fig. 3a contains a simple FORTRAN loop, and Fig. 3b contains a compilation into ZS-1 assembly language.

2.6. Virtual Memory Architecture

In the ZS-1, each process has access to a 4 Gigabyte virtual memory which is addressed with a 32-bit virtual address. This virtual address space is partitioned into two halves: user space (in which each process has its own independent copy) and kernel space (which is shared among all the processes). Both real and virtual memory are divided into 64 Kilobyte pages, the basic unit for memory management and protection.

```

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

```

(a) FORTRAN source.

A5 = -100	.negative loop count
A6 = A - 8	.load initial pointer to A
A7 = B - 8	.load initial pointer to B
A8 = C - 8	.load initial pointer to C
A9 = D - 8	.load initial pointer to D
loop: A5 = A5 + 1	.increment A5
B, A0 = (A5 == 0)	.compare =, set branch flag
XLQ = (A7 = A7 + 8)	.load next element of B
XLQ = (A8 = A8 + 8)	.load next element of C
XLQ = (A9 = A9 + 8)	.load next element of D
X2 = XLQ	.copy B element into X2
X3 = X2 * XLQ	.multiply B and C
XSQ = XLQ + X3	.add D; result to XSQ
(A6 = A6 + 8) = XSQ	.store result into A
JMPF loop	.branch on false to "loop"

(b) Assembly language version of the FORTRAN loop.

Fig. 3. A FORTRAN Loop and Its ZS-1 Compilation.

All memory accesses in the ZS-1 are done with 32-bit virtual addresses. En route to memory loads, stores, and instruction fetches pass through an address translation pipeline which contains an address Translation Table. The Translation Table contains 4096 translation descriptors. Hence, a total of 256 Megabytes can be mapped automatically by the hardware at any given time. If there is a miss in the Translation Table, then a trap to the operating system is generated.

3. Decoupled Implementation

A block diagram of the ZS-1 CPU is shown in Fig. 4. Instruction words are fetched by the *Instruction Fetch Unit* which contains a 16K byte instruction cache. Instructions read from the cache are placed in the output register of the *Instruction Fetch Unit*, which is referred to as the "splitter". Here, the 64-bit 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.

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 simultaneously. 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.

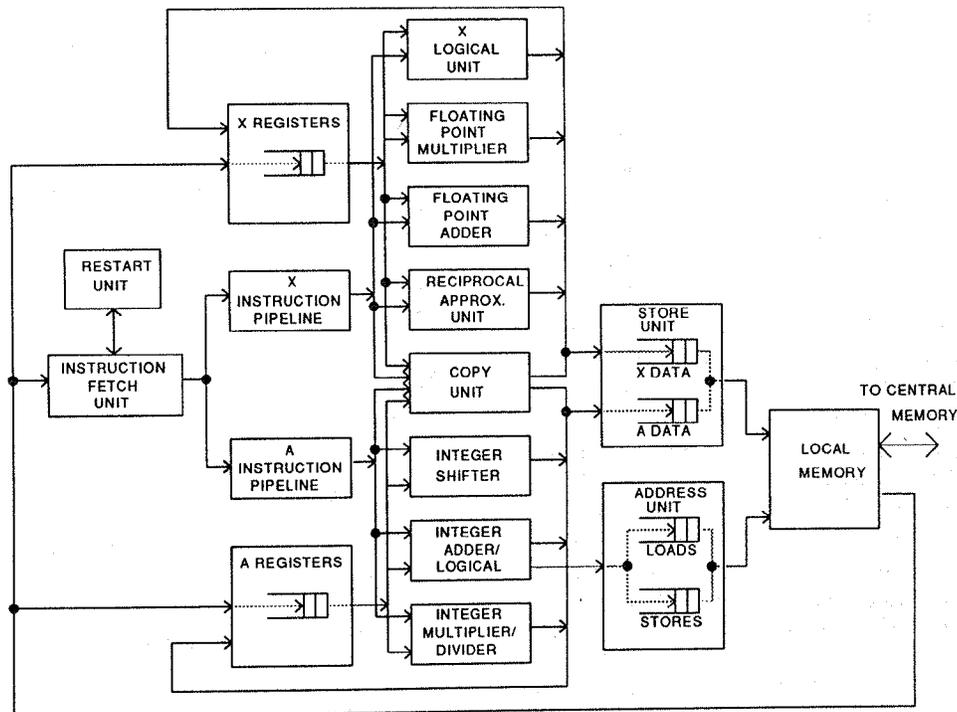


Fig. 4. The ZS-1 Processor Block Diagram.

In the instruction pipelines, instructions are read from the instruction buffers, decoded, and issued to functional units for execution. It is at the issue stage where conflict conditions and data dependencies are detected; for example, if an instruction uses the result of a previously issued but unfinished instruction, it waits at the issue register until the previous instruction completes.

At the time an instruction is issued from one of the pipelines, operand data is read from the appropriate registers and/or queues which are held in the *A Register* and *X Registers*.

After being issued, the instruction begins execution in one of the parallel functional units. The primary functional units for the A instructions are the *Shifter*, the *Integer Adder/Logical Unit*, and the *Integer Multiplier/Divider*. The primary functional units for the X instructions are the *X Logical Unit*, the *Floating Point Adder*, the *Floating Point Multiplier*, and the *Reciprocal Approximation Unit*. Data can be copied between A and X registers via the *Copy Unit*.

Load and store instructions compute their effective addresses in the Integer Adder, and then pass them to the *Address Unit*. The Address Unit contains the translation table and associated logic to translate a virtual address into a physical address. Then the address is placed into a load address queue, or a store address queue, depending on the type of memory operation. Store addresses wait until a corresponding data item appears in the appropriate (A or X) store data queue. 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. All load addresses stay in order with respect to one another, as do store addresses.

When a functional unit instruction completes, the result is fed back to the register file via a result bus. Store data is passed from functional units into the *Store Unit*. The Store Unit contains the store data queues for both 32-bit and 64-bit data. Data waits in the queues until the corresponding store address is ready in the Address Unit.

Memory loads and stores are processed in the *Local Memory*. It is organized as a 128 Kilobyte write-back cache, enhanced with instructions to explicitly manage it. The cache line size is 128 bytes.

To help with interrupts and traps, the *Restart Unit* contains interrupt and trap registers and mask registers. It also contains the control logic for context switching.

Branches and system call/return instructions are detected and held in the splitter. These instructions use the long constant format, so they never share the splitter register with another instruction and are more easily detected.

Conditional branches need to test the Branch Flag (B). Compare instructions which set B are detected at the splitter as they pass through, and B is marked as "busy". The compare instruction modifying B proceeds up one of the instruction pipelines, issues, completes, and sets B. At that time, B is marked as "not busy". Meanwhile, if a conditional branch instruction encounters a busy Branch Flag, it must wait. When B is not busy, the branch decision is made, and fetching either resumes with the next sequential instruction fetch

address, or with the conditional branch target, depending on the outcome.

Example

Fig. 5 illustrates the processing of two iterations of the loop in Fig. 3b. Only the instructions within the loop body are shown. Time, in clock periods, runs across the page from left to right. The letters to the right of the instruction sequence indicate the pipeline stages that hold the corresponding instruction during each clock period. Lower case letters are used for A instructions and branches; upper case letters are used for X instructions. In the timing diagram, instruction processing starts at the splitter; it is assumed that all instruction fetches hit in the instruction cache. The letters labeling pipeline stages have the following meaning:

S or **s** indicates the instruction is being processed at the splitter.

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

D or **d** indicates the instruction is being decoded.

I or **i** indicates the instruction is being issued for execution.

E or **e** indicates the instruction is being executed.

Periods are used to indicate that instructions are being held in a stage, possibly due to delays in preceding instructions, and are simply waiting for the next pipeline stage to become available.

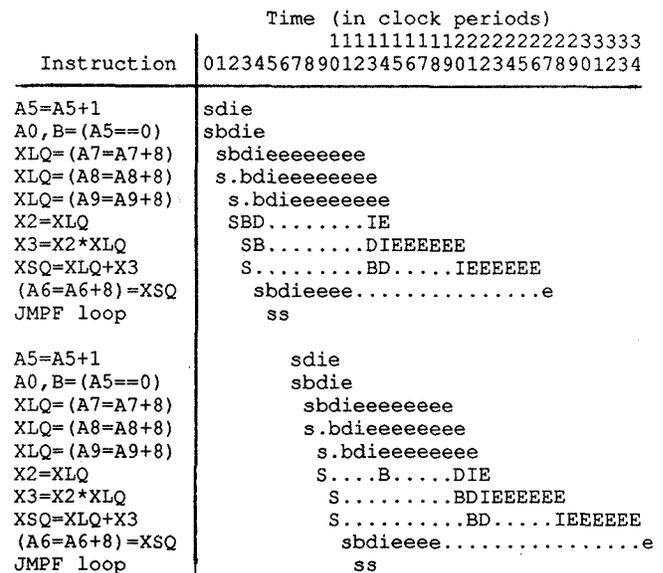


Fig. 5. The Processing of Two Iterations of the Loop in Fig. 4.

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 then must 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.

The second and all subsequent loop iterations are similar to the first. However, the A instructions have moved ahead of the X stream, so that the waits for memory data are eliminated, and the X instructions requiring data from memory issue as soon as they are decoded. In steady state, dependencies involving the floating point operations determine overall performance. By extrapolating from the diagram it can be seen that up to four iterations of the loop can be in some phase of processing simultaneously. During many clock periods eight or more instructions are being processed in parallel (not counting those being blocked).

The example just given is intended to illustrate aspects of the ZS-1 implementation. In fact, the compilers automatically unroll loops. When this is done, and instructions are rescheduled using the resulting larger basic blocks, floating point operations are interleaved so that they can be issued at a maximum rate. The same loop, unrolled four times, is illustrated in Fig. 6. The bottleneck becomes the memory access path. The A instruction stream is able to issue a load or store nearly every clock period. Hence, the memory path is saturated and performance comparable to a vector processor is achieved. In the example, as many as 20 instructions are processed in parallel.

4. Instruction Scheduling

A very important aspect of pipelined processing is the ordering or scheduling of instructions to increase overlap. This scheduling can be done by the compiler (static scheduling) and at runtime by the hardware (dynamic scheduling). These two types of scheduling are by no means mutually exclusive, and in the ZS-1 both are used extensively.

Because of recent successes with simplified instruction sets and hardware, combined with increased emphasis on compiler technology, there has been a recent movement toward eliminating all hardware interlocks [HENN83, FISH83]. However, we do not take such an extreme view; the advantages to hardware interlocks are too great to discard them.

First, consider arguments against dynamic scheduling. One is that dynamic scheduling requires additional hardware.

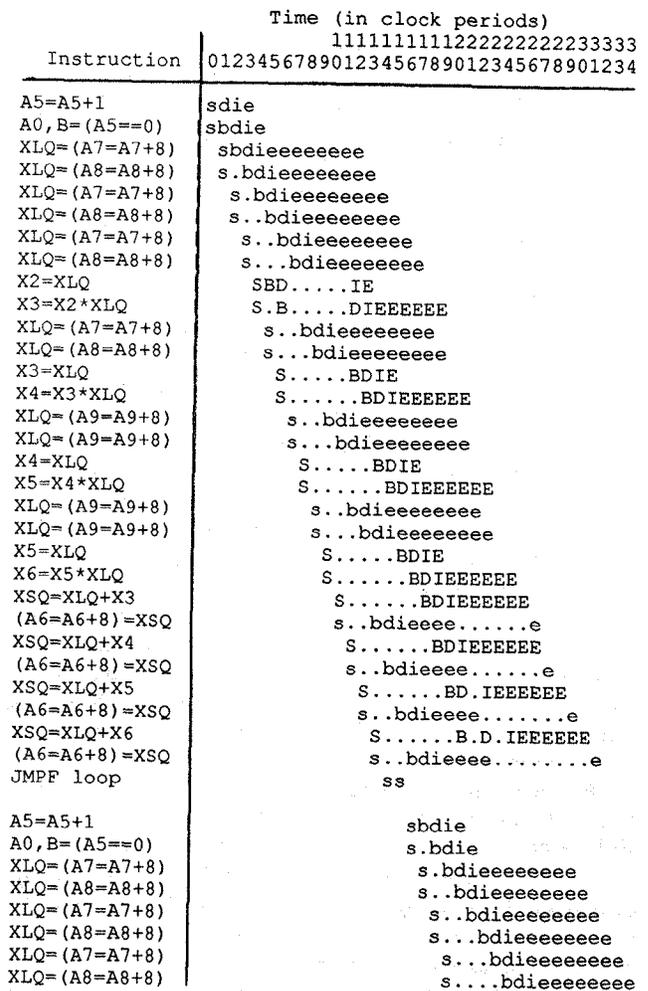


Fig. 6. The Processing of an Unrolled Loop

In a 64-bit processor which has an architecture designed for high overlap, the hardware devoted to pipeline interlocks is rather small. A more important potential disadvantage is delay added to critical timing paths which can increase the clock period. In the ZS-1, the design is done in such a way that the required interlocks are not the most critical paths. In fact, the tightest paths are in the data portion of the integer adder.

The example in Fig. 5 illustrates the way that static and dynamic scheduling complement each other in the ZS-1 design. Each of the two pipelines processes its instructions strictly in the order in which they are received. This is to keep the instruction issue logic simple. The issue logic must check register, queue, functional unit, and result bus conflicts. By doing as much preprocessing as possible during the decode pipeline phase, the issue phase can be done in the time required to do a multiplex and a NAND function. The remainder of the issue clock period can be used for the large fanout required of the "issue" control signal. Because each of the streams is executed in order, static scheduling is important for keeping the separate instruction streams highly overlapped within themselves.

On the other hand, the two instruction streams do not have to be executed in lockstep. The data access stream can move ahead of the floating point execution stream. This is made possible by the splitting of the instruction streams, the large buffers at the beginning of the X instruction pipeline, and the hardware in the Address Unit that allows loads to bypass stores. This is a form of dynamic scheduling that allows instructions from one basic block to be issued while floating point instructions from a preceding basic block are still awaiting execution. While static scheduling can be used to modify the size and contents of basic blocks (as is done in loop unrolling, for example), it cannot provide the capability of spanning basic blocks at run time.

Performance is also improved because dynamic scheduling decisions are made at run time based on actual data and addresses. Consider vectorizing compilers where array references must be checked for memory hazards involving loads and stores. If store addresses in one loop iteration are different from the load addresses of subsequent iterations, then a loop may be vectorized. However, the sets of addresses may not be known at compile time, for example when there are complex subscript calculations, subscripted subscripts, or subscript values passes as subroutine arguments. In these cases a compiler must often make the worst-case assumption that there will be conflicts. In the ZS-1, load addresses in the Address Unit are allowed to pass preceding stores, provided their addresses do not conflict. Memory hazard decisions are made at run time with the actual addresses. Hence, the ZS-1 can achieve a vector level of performance, on what might well be non-vectorizable code.

A mix of static and dynamic scheduling is also used for overlapped execution of conditional branches. Static scheduling is used to push the compare instruction ahead of the branch as far as possible so that the branch decision can be made in advance of when the branch instruction enters the splitter. Because branch instructions are executed at the splitter, instead of one of the instruction pipelines, the branch can be executed ahead of both A and X instructions that precede it. This dynamic scheduling allows the branch instruction to be executed early, and instruction fetching can proceed while instructions from previous basic block(s) are still buffered in the A and X Instruction pipelines.

5. Implementation of Virtual Memory

In a processor as highly parallel as the ZS-1, the implementation of virtual memory presents some problems. Perhaps the most significant problem is in saving and restoring a context properly when a virtual memory trap condition is detected during instruction processing.

To simplify the problem, a trap is generated when there is a miss in the Translation Table held in the CPU. The hardware does not go to page tables in memory as is the case in a conventional virtual memory implementation. Instead, operating system software is responsible for searching page tables and updating the Translation Table via special instructions. One advantage of using this type of virtual memory translation method is that it reduces control hardware needed

to perform address translation. A more significant advantage is that it is always known exactly 4 clock periods after a load or store instruction issues whether it will get a virtual memory trap. This, short constant delay allows precise restartable traps to be implemented more easily because less state information needs to be buffered because of pending instructions that may trap.

There are also some software advantages to implementing virtual memory in this way. For example, the software implementation of page tables provides greater flexibility because the operating system can use page table structures that the system software writers deem optimal.

The most significant disadvantage of this virtual memory method is the potential for added delays because there is a trap for every Translation Table miss, rather than for every page fault as in a conventional system. We have attempted to reduce the performance impact by using a large translation table that can map 256 Mbytes of virtual memory at any given time. There is also a special vectored location set aside for translation traps.

6. Summary

The ZS-1 is a scalar processor that provides high performance on inherently scalar code, and performance similar to that provided by a vector processor on vectorizable code. This is achieved by using a mix of static and dynamic instruction scheduling. The compilers automatically unroll loops to increase the size of basic blocks. Basic blocks are statically scheduled to minimize delays. Dynamic scheduling allows runtime re-ordering of instructions beyond the basic block boundaries provided by the compiler. Furthermore, because runtime scheduling by the hardware is done with actual rather than worst-case data and address values, it can make scheduling decisions not possible in the compiler.

Virtual memory is provided in order to speed program development and permit efficient memory management by the operating system. Virtual memory, however, has potential for reducing performance. To reduce performance losses, the ZS-1 virtual memory architecture is specially designed for a pipelined implementation.

References

- [FISH83] J.A. Fisher, "Very Long Instruction Word Architectures and the ELI-512", *10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, pp.140-150, June 1983.
- [HENN83] J.L. Hennessy, et al., "Design of a High Performance VLSI Processor," *3rd Caltech Conference on VLSI*, pp. 33-54, Mar. 1983.
- [SMIT86] J.E. Smith, S. Weiss, and N. Y. Pang, "A Simulation Study of Decoupled Architecture Computers," *IEEE Transactions on Computers*, pp. 692-702, Aug. 1986.
- [THOR70] J. E. Thornton, *Design of a Computer - The Control Data 6600*, Scott, Foresman and Co., Glenview, IL, 1970.
- [WOR81] J. Worlton, "The Philosophy Behind the Machines," *Computer World* Nov. 9. 1981.