Palo Alto Research Centers

A Retrospective on the Dorado, A High-Performance Personal Computer

Kenneth A. Pier

XEROX

A Retrospective on the Dorado, A High-Performance Personal Computer

Kenneth A. Pier

ISL-83-1 August 1983 [P83-00007]

Abstract: In late 1975, members of the Xerox Palo Alto Research Center embarked on the specification of a high-performance successor to the Alto personal minicomputer, in use since 1973. After four years, the resulting machine, called the Dorado, was in use within the research community at PARC. This paper begins with an overview of the design goals, architecture, and implementation of the Dorado and then provides a retrospective view and critique of the Dorado project as a whole. The major machine architectural features are evaluated, other project aspects such as design automation and management structures are explained, a chronological history with milestones is included, and a variety of accomplishments, red herrings, and shortfalls is discussed. The paper concludes with some speculations on what the project might have done differently and what might be done differently today instead of in the late 1970s.

Although more than a dozen scientists and technicians contributed to the project, the evaluative and speculative parts of this paper are the sole responsibility of the author.

CR categories: B.1.1 [Control Structures and Microprogramming]: Control Design Styles; B.3.2 [Memory Structures]: Design Styles; B.4.3 [Input/Output and Data Communications]: Interconnections (subsystems); C.1.3 [Processor Architectures]: Other Architecture Styles

Key words and phrases: architecture, processor, memory, cache, instruction fetch, emulation, input/output, microprogram, pipeline, retrospective.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Table of Contents

1. Introduction

The machine reviewed in this paper, called the Dorado, was designed and implemented by the Computer Science Laboratory (CSL) of the Xerox Palo Alto Research Center (PARC). It is currently used by researchers in several laboratories working in most areas of computer science, including programming systems, networks, database systems, VLSI design, graphics and imaging, office automation, artificial intelligence, computational linguistics, and analysis of algorithms. These researchers place a heavy emphasis on building usable prototype systems and many such systems, both hardware and software, have been developed over the last ten years. Most systems are part of a personal computing environment which is loosely coupled to other such environments, and to service facilities for storage and printing, by a high-bandwidth communication network [27].

The Dorado provides the hardware base for the current generation of system research within Xerox PARC. It supports a variety of high-level language environments and high-bandwidth I/O devices. Although the Dorado far exceeds the capabilities and cost of what is traditionally thought of as a personal computer, it is personal in the sense that it is designed to be used by a single (expert) user running multiple cooperating processes in an integrated programming environment. It has a microprogrammed processor with a 60 nanosecond microinstruction cycle time, a high-speed cache, memory map, large main memory, an instruction fetch/decode unit, and high-resolution monochrome and color displays. The processor is shared among priority-ordered microcoded *tasks*, performing microcode context switches on demand with no overhead. The memory subsystem is controlled by a seven-stage pipeline. It can deliver a peak main-storage bandwidth of 530 million bits per second to service fast I/O devices and cache misses, and has a cache with hit rates commonly over 99%. The instruction fetch unit (IFU) speeds up the emulation of instructions by fetching, decoding, and preparing later instructions in parallel with the execution of earlier ones. A writeable decoding memory allows the IFU to be specialized to particular instruction sets. There are implementations of instruction sets for the BCPL [27], Mesa [10], Interlisp [26], and Smalltalk [7] languages. The IFU is implemented with a six-stage pipeline, and under favorable conditions can deliver instructions at a peak rate of 16 million instructions per second. The machine is implemented using standard ECL 10K technology.

The Dorado has been extensively documented in three papers [13, 2, 14, collected in 12]. The technical overview in this paper is excerpted from these papers. It begins by sketching the history of the machine's development (Section 2), discusses the design goals for the architecture (Section 3), and then summarizes the most important features of the processor, memory, IFU, and I/O subsystems in turn (Section 4). The retrospective (Section 5) discusses the subsystems, debugging aids and diagnostics, design automation and fabrication issues, and the project management and personnel structure. General observations, comments, and rumination follow (Section 6). A final section offers some speculation based on hindsight, and some attempt at foresight (Section 7).

2. History

The Dorado is a descendant of a small personal computer called the Alto, which was designed and built as an experimental machine during 1973 [27]. The Alto is a fairly simple machine, but it has several features which have turned out to be important:

a microprogrammed processor that is efficiently shared among all the device controllers as well as the virtual machine interpreter;

a high-resolution display system that uses a full bitmap stored in the Alto main memory;

a device for pointing at images on the display;

an interface to a high-bandwidth communication network.

The microarchitecture allows all the device controllers to share the full power of the processor, rather than having independent access to the memory. As a result, controllers can be small and yet the I/O interface provided to programs can be powerful. This concept of processor sharing is fundamental to the Dorado as well.

It was clear by 1975 that a large and rapidly increasing amount of effort was going into surmounting the Alto's limitations of space and speed, and its lack of virtual memory, rather than into trying out research ideas in experimental systems. Researchers therefore began to design a new machine aimed at relieving these burdens. During 1976 and 1977, design work on the Dorado proceeded in the Computer Science Laboratory and the Systems Development Department. Requirements and contributions from parts of Xerox outside of Research affected the design considerably; the memory bandwidth and processor throughput were substantially increased.

In 1977, implementation of the laboratory prototype for the Dorado began in the Computer Science Laboratory. The prototype packaging and a design automation system had already been implemented, and were used for constructing and debugging Dorado "Model 0." A small team of people worked steadily on all aspects of the Dorado system until summer of 1978, when the prototype successfully ran Alto software in an Alto emulation mode. During the summer and fall of 1978 the lessons learned in debugging and microcoding the Model 0, together with the significant improvements made in memory technology since the Model 0 design was frozen, were used to redesign and reimplement nearly every section of the Dorado. Some serious design errors and a number of annoyances to the microcoder were fixed, all the memories of the machine were substantially expanded, and the basic cycle time was decreased. Dorado Model 1 came up in the spring of 1979.

During the next year, several copies of this machine were built in the stitchweld technology used for the prototypes. Stitchwelding works very well for prototypes, but is too expensive for even modest production quantities. Its major advantages are packaging density and signal propagation characteristics similar to those of production technologies, very rapid turnaround during development (three days for a complete 300-chip board, a few hours for a modest change), and complete compatibility with our design automation system.

At the same time, the design was transferred to multiwire circuit boards; the Manhattan wire routing and lower impedance of this technology slowed the machine down by about 15%.

Approximately 25 Dorados were manufactured in this technology. By fall of 1982, a production line was delivering Dorados to the Palo Alto Research Center at the rate of three or four per month, and the production technology had successfully been transferred to multi-layer printed circuit boards that would have been impractical just a few years before.

Since the power of the Dorado has made a substantial impact on the productivity of researchers and system developers, the Computer Science and Imaging Sciences Laboratories have decided to equip all members of the programming staff with personal Dorados.

Appendix A of this paper contains a chronology of the Dorado project, and Appendix B is a tabulation of the people and their contributions.

3. Design Goals

The Dorado is intended to be a powerful but personal computing system supporting a single user within a programming system that extends from the microinstruction level to an integrated programming environment for a high-level language. It should be physically small and quiet enough to occupy space near its users in an office or laboratory setting, and inexpensive enough to be acquired in considerable numbers. These constraints on size, noise, and cost have a major effect on the design. Not all of these design goals were successfully met, as the retrospective part of this paper will discuss.

The Dorado is designed for efficient execution of multiple languages that are compiled into a stream of *byte codes* [24]; this execution is called *emulation*. Such byte code compilers exist for Mesa [10, 17], Interlisp [1, 25] and Smalltalk [7]. The instruction fetch unit (IFU) fetches bytes from such a stream, decodes them as instructions and operands, and provides the necessary control and data information to the processor [14]. Further support for efficient emulation dictates a very fast microcycle, a microinstruction powerful enough to allow interpretation of a simple macroinstruction in a single microinstruction, and a cache with low latency and high throughput.

Very high-bandwidth input/output capability is another major goal for the Dorado. In particular, color graphics monitors, raster scanned printers, and high-speed communications are all part of the research activities at Xerox; these devices typically have bandwidths ranging from 20- to 400-megabits/second. Fast devices should not delay emulation too much, even though the two functions compete for many of the same resources. Relatively slow devices must also be supported, without tying up the high-bandwidth I/O system. These considerations clearly suggest that I/O activity and emulation should proceed in parallel as much as possible. A memory system which supports these requirements allows cache accesses for emulation and main storage references for I/O to proceed in parallel, a cache reference to start in every microinstruction cycle, and a storage reference to start in every main storage cycle. It must also be possible to integrate new device controllers into the Dorado in a relatively straightforward way, by writing microcode and creating a small amount of new device-specific hardware that can be plugged into the existing system.

Relief of the bottlenecks commonly found in memory systems requires hardware support for virtual memory, with a large virtual address space and correspondingly large real address space and storage. In addition, the ability to evolve with advancing memory technologies is called for as memory devices are expected to double and redouble in capacity over the lifetime of the Dorado. However, since the architecture supports a single user using collections of cooperating processes with a high degree of information sharing, the machine deliberately does not support multiple or protected address spaces.

4. Architecture Overview

Figure 1 is a simplified block diagram of the Dorado. Aside from I/O, the machine consists of the processor, the IFU, and the memory system. Both the processor and the IFU can make memory references and transfer data to and from the memory through the cache. Slow (low-bandwidth) I/O devices communicate with the processor, which in turn transfers their data to and from the cache. Fast (high-bandwidth) devices communicate directly with storage, bypassing the cache most of the time.

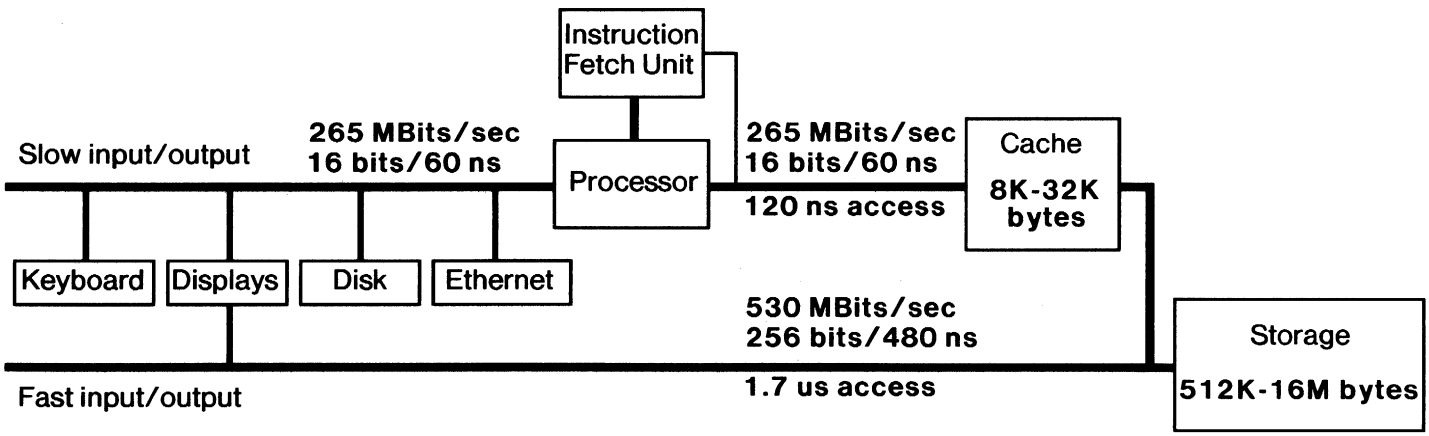


Figure 1: Dorado Block Diagram

For the most part, data is handled sixteen bits (one *word*) at a time. The relatively narrow busses, registers, data paths, and memories which result from this choice help to keep the machine compact. This is especially important for the memory, which has a large number of busses. Packaging, however, is not the only consideration. Speed dictates a heavily pipelined structure in any case, and this parallelism in the time domain tends to compensate for the lack of parallelism in the space domain. There are several pipelines, and they are generally able to start a new operation every cycle. The memory, for instance, has two pipelines, the processor has two, the instruction fetch unit another. Also, there are many independent busses: eight in the memory, half a dozen in the processor, three

in the IFU. These busses increase bandwidth and simplify scheduling. Large I/O bandwidth requires multiple parallel busses in order to allow I/O activity and emulation to proceed in parallel. Keeping the machine physically small also improves the speed, since physical distance (i.e., wire length) accounts for a considerable fraction of the basic cycle time. Finally, performance is often limited by the cache hit rate, which cannot be improved, and may be reduced, by wider data paths (if the number of bits in the cache is fixed).

Rather than putting processing capability in each I/O controller and using a shared bus or a switch to access the memory, the Dorado shares the processor among all the I/O devices and the emulator. This idea originated in the TX-2 computer [6] and is also used in the Alto. This processor sharing is accomplished with 16 hardware-scheduled microcode processes called *microtasks*, or simply *tasks*. Tasks have fixed priority. Most tasks serve a single I/O device, which raises a request line when it wants service from its task. Hardware schedules the processor to serve the highest priority request; control can switch from one task to another on every microinstruction with no overhead. When no device is requesting service, the lowest priority task runs and does high-level language emulation. To eliminate the time cost of multiplexing the processor among the tasks in this way, a number of the machine's working registers are *task-specific*, that is, there is a copy for each task. The implementation typically involves a single physical register, and a 16-element memory which is addressed by the current task number and whose output is held in the register.

When no I/O device wants service, the emulator runs. To execute byte codes, the processor gives the IFU an initial program counter, and subsequently receives a sequence of decoded instructions, which are from sequential bytes except where the IFU has followed a branch. This sequence continues until the processor resets the IFU with another program counter, usually due to a conditional branch that causes the processor to change the locus of program control, or until a fault or interrupt is detected. For each instruction the IFU supplies a microcode dispatch address (into which instructions or exceptions are encoded), some bits of initial state for the processor, a sequence of field data values, and the program counter value for the first byte of the instruction.

Instruction interpretation by the IFU is based on a definite model of how instructions are encoded. Although this model is not specialized to the details of a particular target instruction set, good performance depends on adherence to certain rules. The IFU deals with variable length instructions of up to three bytes in length. Variable length instructions provide code compaction, since frequent operations can be encoded as one-byte instructions. There is also a performance payoff in cache and virtual memory systems, since the compaction enhances locality and thus reduces cache misses and page faulting. Our experience has shown that byte codes provide a flexible format for different languages without favoring a particular one. The choice of eight-bit bytes as the encoding grain is a compromise among optimum encoding, the desire to keep code addresses short, and simplicity of the hardware. A larger grain is highly undesirable, both because more than half the instructions can fit into one byte, and because table lookup as a decoding technique is not feasible

for units much larger than eight bits. A finer grain improves code compactness at the expense of more complex instruction length calculation and memory word disassembly.

The memory system implements paged virtual memory with a maximum virtual address of 28 bits, depending on memory chip density. Memory references specify a 16- or 28- bit *displacement*, and one of 32 *base registers* of 28 bits; the virtual address is the sum of the displacement and the base. Virtual address translation, or *mapping*, is implemented by table lookup in a dedicated memory. *Main storage* is the permanent home of data stored by the memory system. The storage is necessarily slow (i.e., it has long latency, which means that it takes a long time to respond to a request), because of its implementation in cheap but slow dynamic MOS RAMs. To make up for being slow, storage is big, and it also has high-bandwidth, which is more important than latency for sequential references. In addition, there is a *cache* which services non-sequential references with high-speed (low latency), but is inferior to main storage in its other parameters. Deeper layers of virtual memory, such as disk virtual memory, are not implemented in hardware.

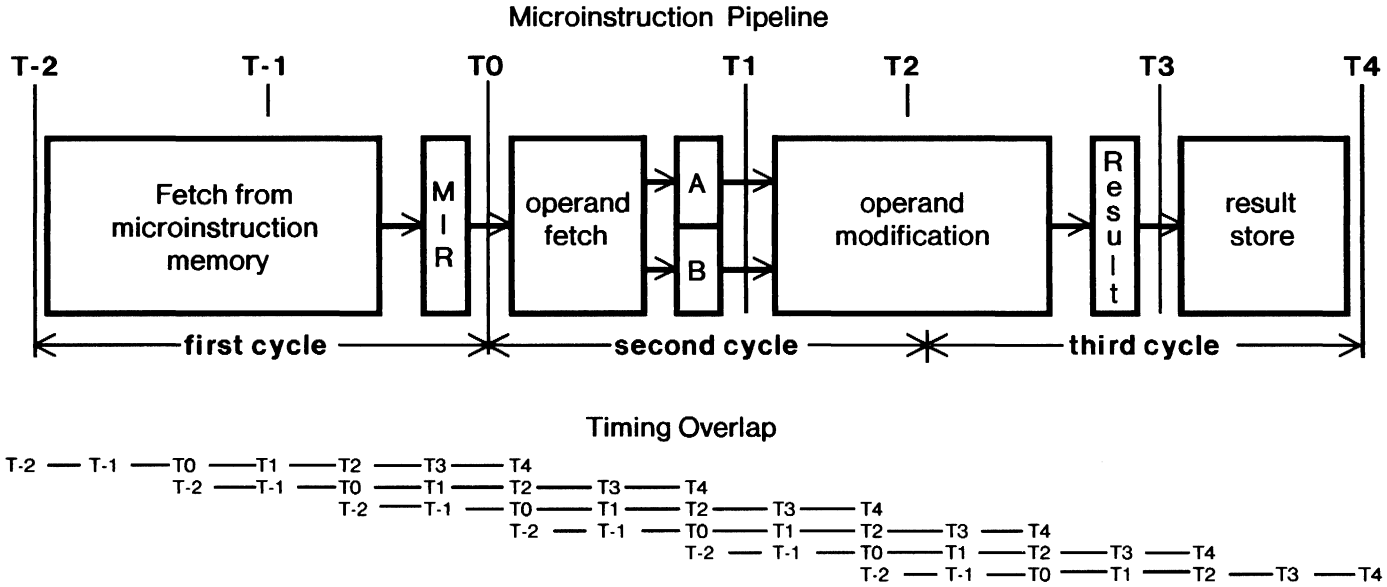
With one exception (the IFU), all memory references are initiated by the processor, which acts as a multiplexor controlling access to the memory and is the sole source of addresses. Once started, a reference proceeds independently of the processor. Each one carries with it the number of its originating microtask, which serves to identify the source or sink of any data transfer associated with the reference. The actual transfer may take place much later, and each source or sink must be continually ready to deliver or accept data on demand. It is possible for a microtask to have several references outstanding, but order is preserved within each type of reference, so that the task number plus some careful hardware bookkeeping is sufficient to match up data with references.

4.1 Processor Details

A detailed description of the processor appears in [13]; this subsection summarizes that material. The processor includes the basic data, arithmetic, and control paths for the machine. It is a microprogrammed unit using writeable control store containing 4096 36-bit microinstructions (34 instruction bits plus two parity check bits). Microinstructions are tightly encoded with subfields often interpreted in several different ways depending on the type of instruction being executed. There is a four-stage microinstruction execution pipeline; a block diagram of this pipeline appears as Figure 2. The pipeline allows a new microinstruction to begin execution every microcycle (60 nanoseconds) and completes execution of a microinstruction in three microcycles.

Figure 3 is a detailed block diagram of the data section of the processor. Its major features include 256 high-speed registers (RM on the diagram) and four 64-deep high-speed stacks (STACK) accessed via the microinstruction. There is a 32-bit-wide shifter (Shifter) and masker for bit field extraction/insertion or for handling bitmapped graphics. The ALU operates on 16-bit quantities. There is also a set of temporary registers (T, Q, COUNT) for intermediate results. Numerous independent busses (memory address, memory data, IO data) allow communication with memory and I/O devices simultaneously.

There are 16 *tasks* (priority levels) associated with microcode execution. Each task is normally associated with some hardware and microcode which together implement a device controller. The tasks have a fixed priority, from task 0 (lowest) to task 15 (highest). Device hardware can request that the processor be switched to the associated task; such a *wakeup request* will be honored when no requests of higher priority are outstanding. The set of wakeup requests is arbitrated within the processor, and a *task switch* from one task to another occurs on demand, typically every ten or twenty microcycles when a high-speed device is running. Task arbitration logic is not shown in Figure 3.



Time between Ts is 30 nsec. T0 marks the time when the microinstruction is latched in the microinstruction register and begins execution.

Figure 2: Microinstruction Pipeline and Timing Overlap

When a device acquires the processor (that is, the processor is running at the requested priority level and executing the microcode for that task), the device will receive service from its microcode. Eventually the microcode will *block*, relinquishing the processor to lower priority tasks until it next requires service. While a given task is running, it has the exclusive attention of the processor. This arrangement is similar in many ways to a conventional priority interrupt system. An important difference is that the tasks are like coroutines or processes, rather than subroutines; when a task is awakened, it continues execution at the point where it blocked, rather than restarting at a fixed point. This ability to capture part of the state in the microprogram counter is very useful.

Task 0 is not associated with a device controller; its microcode implements the emulators currently resident in the Dorado. Task 0 requests service from the processor at all times, but with the lowest priority.

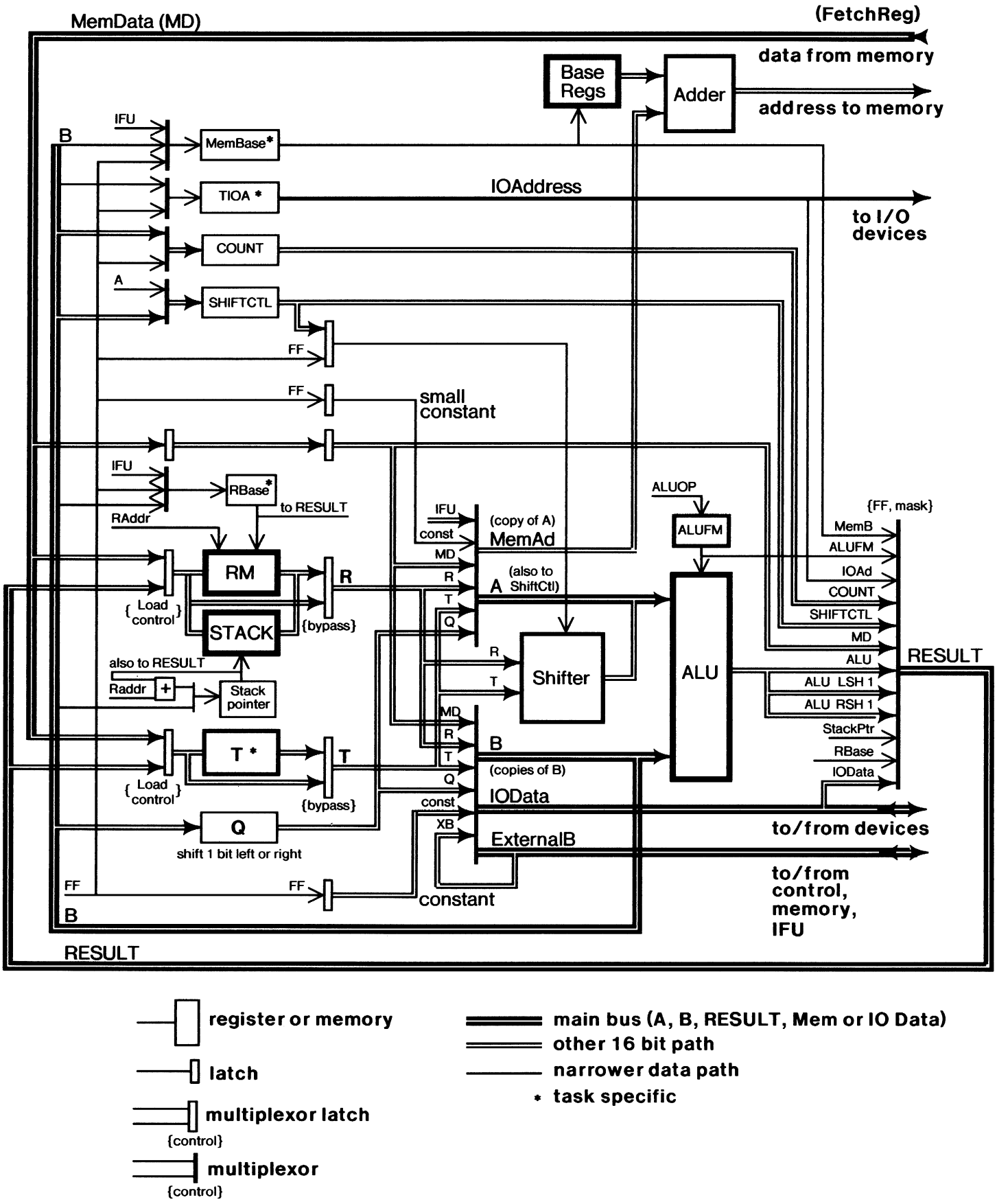


Figure 3: Processor Data Section

In order to allow immediate task switching, the processor must be able to save and restore state with no microinstruction overhead. This is accomplished by keeping volatile state information throughout the processor not in a single rank of registers but in *task-specific* registers. These are implemented with high-speed memory that is addressed by a task number. Examples of task-specific registers are the microcode program counter, the branch condition register, the microcode subroutine link register, the memory data register, and a temporary storage register (T) for each task. The number of the task which will execute in the next microcycle is broadcast throughout the processor and used to address the task-specific registers. Thus, data can be fetched from the high-speed task-specific memories and be available for use in the next cycle. Not all registers are task-specific.

There are two distinct pipelines in the Dorado processor. The main one fetches and executes microinstructions. The other handles task switching, arbitrates wakeup requests, and broadcasts the next task number to the rest of the Dorado. Each pipeline is synchronous, and there is no waiting between stages.

4.2 Memory Details

This subsection is excerpted from [2]. Many interesting problems and solutions within the memory subsystem deal with pipeline management, hardware resource conflict resolution, and processor/memory/IO communication. [2] describes these issues in detail; they are too lengthy to recount here.

Reference	Task	Effect
Fetch(a)	any task	fetches one word of data from virtual address *a* in the cache and delivers it to FetchReg
Store(d, a)	any task	stores data word *d* at virtual address *a* in the cache
I/ORead(a)	I/O task only	reads a 16-word data block at virtual address *a* in storage and delivers it to a fast output device
I/OWrite(a)	I/O task only	takes a data block from a fast input device and writes it at virtual address *a* in storage
Prefetch(a)	any task	forces the data block at virtual address *a* into the cache
Flush(a)	emulator only	removes from the cache (storing if dirty) the data block at virtual address *a*
MapRead(a)	emulator only	reads the map entry addressed by virtual address *a*
MapWrite(d, a)	emulator only	writes *d* into the map entry addressed by virtual address *a*
DummyRef(a)	any task	makes a pseudo-reference guaranteed not to reference storage or alter the cache (useful for diagnostics)

Table 1: Memory Reference Operations

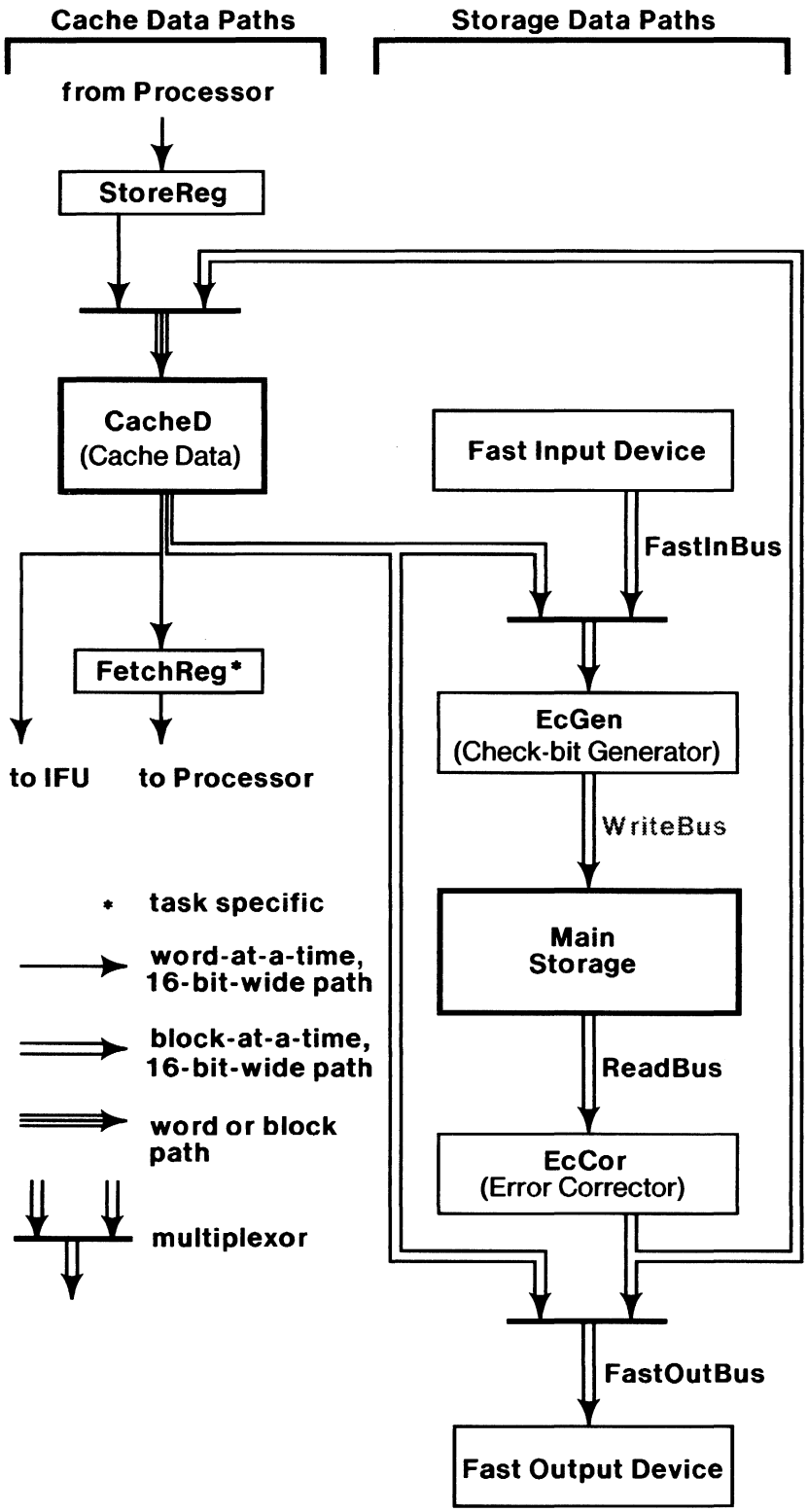


Figure 4: Memory System Data Paths

Recall that the system implements a straightforward paged virtual memory with a single level of mapping implemented via table lookup. All references originate in the processor (except IFU fetches) and deal in virtual addresses. The memory reference operations are shown in Table 1 above.

Figure 4 is a picture of the memory main data paths. A *Fetch* from the cache delivers data to a memory data register called *FetchReg*, from which it can be retrieved at any later time; since FetchReg is task-specific, separate tasks can make their cache references independently. An *I/ORead* reference delivers a 16-word *block* of data from storage to the *FastOutBus* by way of the error corrector, tagged with the identity of the requesting task; the associated output device is expected to monitor this bus and use the data when it appears. Similarly, the processor can *Store* one word of data into the cache, or do an *I/OWrite* reference which demands a block of data from an input device and sends it to storage (by way of the error check-bit generator). There is also a *Prefetch* reference, which brings a block into the cache. *Fetch*, *Store*, and *Prefetch* are called *cache references*. There are special references to flush data from the cache and to allow map entries to be read and written.

A cache reference usually *hits*; it finds the referenced word in the cache. If it *misses*, a *main storage operation* must be done to bring the block containing the requested word into the cache. In addition, I/O references always do storage operations. There are two kinds of storage operations, *read* and *write*. The former transfers a block out of storage to the cache or I/O system; the latter transfers a block into storage from the cache or I/O system.

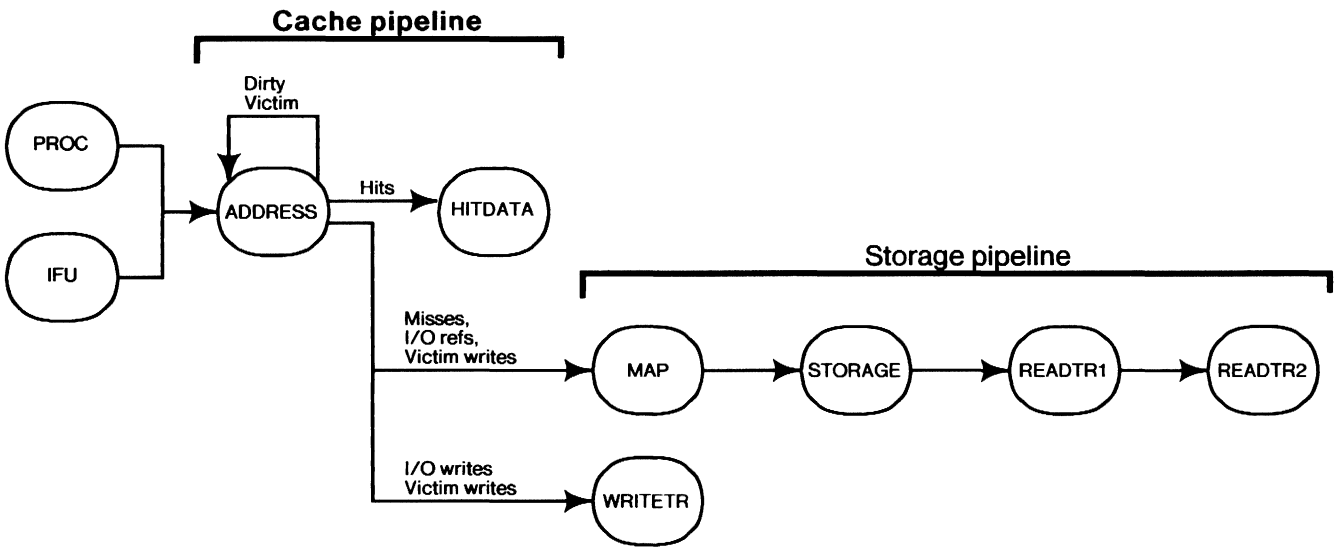


Figure 5: Cache and Storage Pipelines

Figure 5 shows the pipelined organization of the memory system. There are two pipelines. One, consisting of the ADDRESS and HITDATA stages, handles cache references; the other, containing MAP, WRITETR, STORAGE, READTR1, and READTR2 takes care of storage references. A brief explanation of the function of each section follows.

Every reference is first handled by the cache ADDRESS stage, whether or not it involves a cache data transfer. The stage calculates the virtual address and checks to see whether the associated data is in the cache. If it is (a hit), and the reference is a *Fetch* or *Store*, ADDRESS starts HITDATA, which is responsible for the one-word data transfer. On a cache reference that misses, and on any I/O reference, ADDRESS starts MAP.

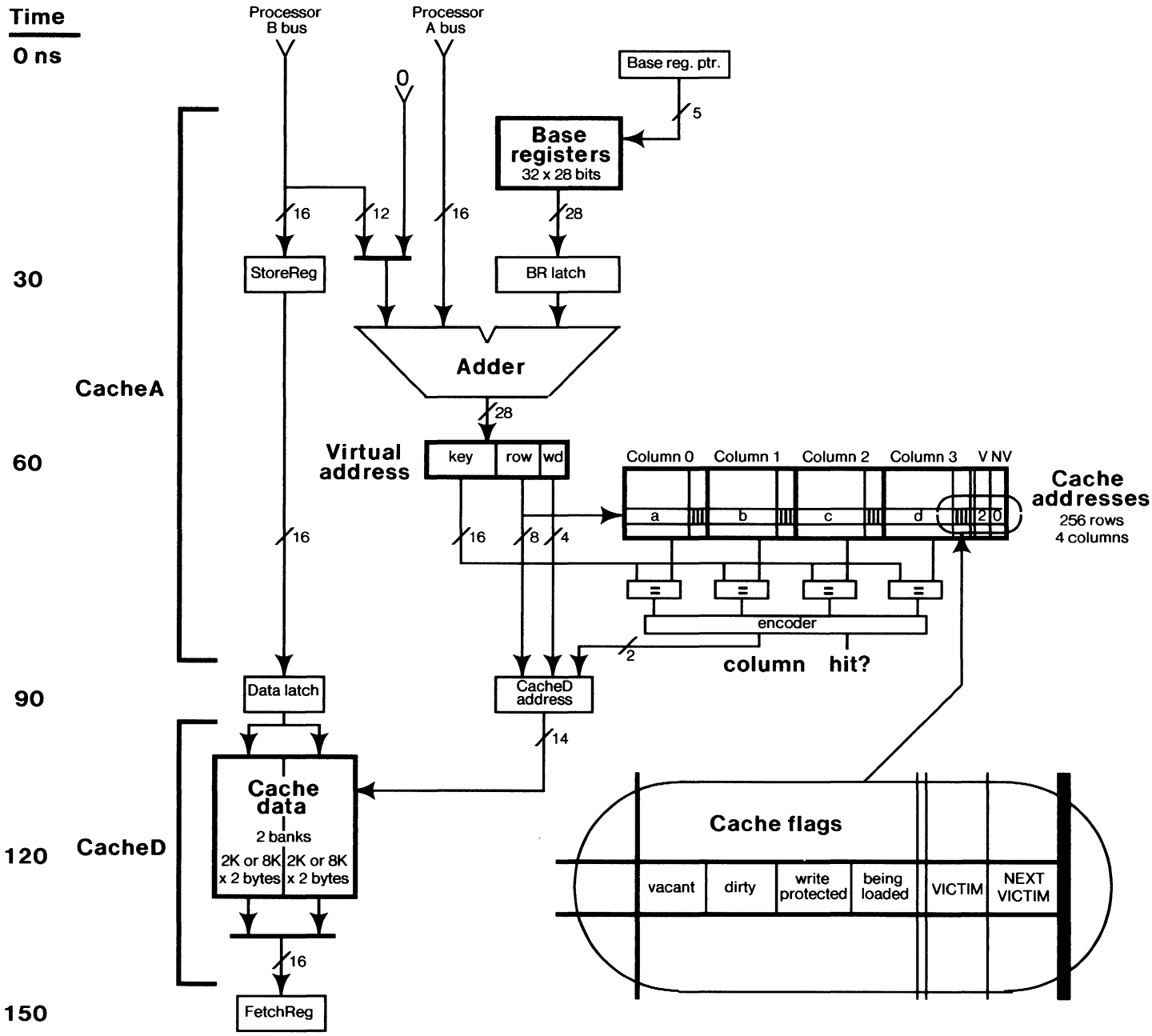


Figure 6: Data Paths in the Cache

The virtual address for the reference is divided into a 16-bit *key*, an 8-bit *row* number, and a 4-bit *word* number; Figure 6 illustrates. This division reflects the physical structure of the cache, which consists of 256 rows, each capable of holding four independent 16-word blocks of data, one in each of four *columns*. A given address determines a row (based on its eight row bits), and it must appear in some column of that row if it is in the cache at all. For each row, the cache address memory, called CacheA, stores the keys of the four blocks currently in that row, together with four flag bits for each block. The Dorado cache is therefore *set-associative* [3]; rows correspond to sets and columns correspond to the elements of a set. The 4-bit word number determines the word desired from the 16-word block.

On a miss, the cache uses a nearly LRU ("least recently used") algorithm to decide which of the four possible column entries to displace with the incoming data block. Each row has two fields, called *Victim* and *NextVictim*, which are managed in hardware by the replacement algorithm. Each field contains a cache column number. When a miss is detected, the data in the column designated by the Victim is written back to storage (if dirty), the NextVictim is promoted to Victim, and a new NextVictim is chosen from one of the other two columns which was neither the Victim nor the NextVictim to begin. The original Victim is overwritten with the incoming block.

The CacheD memory stores the data for the blocks whose addresses appear in CacheA; closely associated with it are the StoreReg and task-specific FetchReg registers which allow the processor to deliver and retrieve its data independently of the memory system's detailed timing. HITDATA obtains the cache address of the word being referenced from ADDRESS, sends this address to CacheD, which holds the cache data, and either fetches a word into the FetchReg register of the task that made the reference, or stores the data delivered by the processor via the StoreReg register. StoreReg is not task-specific because stores are relatively rare (10% to 19% of all cache references) and the cache timing and control are much simpler using a single StoreReg for all tasks. CacheD holds 4-Kwords of data, expandable to 16-Kwords.

Cache misses and fast I/O references use the storage pipeline, shown in Figure 5. Each of the pipeline stages is implemented by a simple finite-state automaton that can change state on every microinstruction cycle. Control is passed from one stage to the next when the first produces a *Start* signal for the second; this signal forces the second automaton into its initial state. Necessary information about the reference type is also passed along when one stage starts another.

The MAP stage translates a virtual address into a real address by looking it up in a hardware table called the MapRAM, and then starts the STORAGE stage. Figure 7 illustrates the straightforward conversion of a virtual page number into a real page number. The low-order bits are not mapped; they address a single word within a page.

Three flag bits are stored in MapRAM for each virtual page:

ref, set automatically by any reference to the page;

dirty, set automatically by any write into the page;

writeProtect, set by memory-management software (using the *MapWrite* reference).

A virtual page not in use is marked as *vacant* by setting both *writeProtect* and *dirty*, an otherwise nonsensical combination. A reference is aborted by the hardware if it touches a vacant page, attempts to write a write-protected page, or causes a parity error in the MapRAM. All three kinds of *map fault* are passed down the pipeline to READTR2 for reporting to the processor.

The map output is a real address in main storage. Main storage consists of from one to four storage modules. Each storage module has a capacity of 2 megabytes using 64K RAM chips or, eventually, 8 megabytes using 256K RAM chips. The two high order bits of the real address select the module. A standard Hamming error-correcting code is used, capable of correcting single errors and detecting double errors in four-word groups.

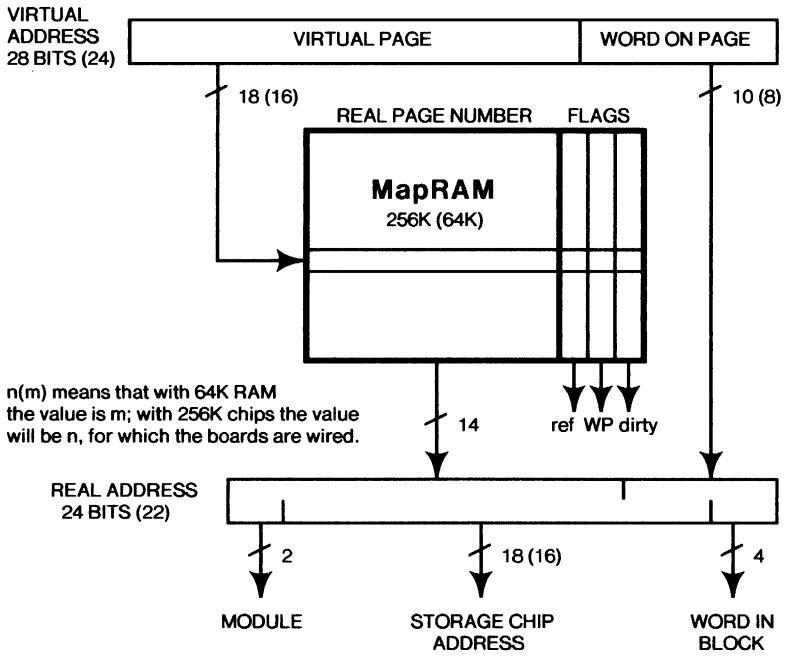


Figure 7: Virtual Address to Real Address Mapping

The Dorado's main storage is controlled by the STORAGE stage. STORAGE is started by MAP, which supplies the real storage address and the operation type (read or write). Storage is organized into 16-word blocks, and the transfer of a block is called a *transport*. All references to storage involve an entire block. Transports into or out of storage take place on word-sized busses called *ReadBus* and *WriteBus* shown in Figure 8. Block-sized shift registers called *ReadReg* and *WriteReg* lie between these busses and storage memory chips. When storage is read, an entire block (256 bits plus 32 error-correction bits) is loaded into ReadReg all at once, and then transported to the cache or to a fast output device by shifting words sequentially out of ReadReg at the rate of two words per microcycle (one word every 30 ns). On a write, the block is shifted a word at a time into WriteReg, and when the transport is finished, the 288 storage chips involved in that block are written all at once.

The WRITETR stage transports a block into WriteReg, either from CacheD or from a fast input device. It uses *ECGen*, the Hamming check bit generator, and WriteBus, and shares WriteReg with STORAGE. It is started by ADDRESS on every write, and synchronizes with STORAGE as needed.

Once ReadReg is loaded by STORAGE, the block is ready for transport to CacheD or to a fast output device. Because it must pass through the error corrector *EcCor*, the first word appears on ReadBus three cycles before the first corrected word appears at the input to CacheD or on the FastOut bus (Figure 4). To match the storage, bus, and error corrector bandwidths, read transport must be controlled by two stages in series; they are called READTR1 and READTR2.

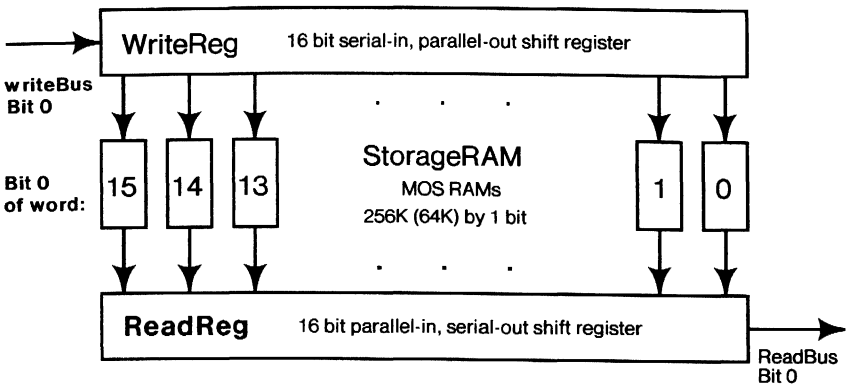


Figure 8: One Bit-slice of Storage RAM and its Data Registers

In fact, these stages run on *every* storage operation, not just on reads. There are several reasons for this. First, READTR2 reports *faults* (page faults, map parity errors, error corrections) and wakes up a fault-handling microtask if necessary; this must be done for a write, as well as for a read. Second, hardware is saved by making all operations flow through the pipeline in the same way. Third, storage latency is in any case limited by the transport time and the storage RAM cycle time. Finishing a write sooner would not reduce the latency of a read, and nothing ever waits for a write to complete.

On a read, STORAGE starts READTR1 just as it parallel-loads ReadReg with a block to be transported. READTR1 starts shifting words out of ReadReg and through the error corrector. On a write, READTR1 is started at the same point, but no transport is done. READTR1 starts READTR2, which shares with it responsibility for controlling the transport and the error corrector. READTR2 reports faults and completes cache read operations either by delivering the requested word into FetchReg (for a fetch), or by storing the contents of StoreReg into the newly-loaded block in the cache (for a store).

As mentioned in the overview, resource management and conflict resolution are major jobs performed by the memory control logic. Most of the difficulty is concentrated in a small section of the logic, which can synchronize and control the other subsystems through the use of a single mechanism called the *Hold* signal. *Hold* is the signal generated by the memory system in response to a processor request that cannot yet be satisfied. Its effect is to convert the microinstruction

containing the request into a jump-to-self; one cycle is thus lost. As long as the same task is running in the processor and the condition causing *Hold* is still present, that instruction will be held repeatedly. However, the processor may switch to a higher priority task which can perhaps make more progress. *Hold* is generated when resources (say, the ADDRESS section or new data in FetchReg) required by the processor are temporarily unavailable because of a previous reference or an incomplete storage read.

4.3 Instruction Fetch Unit Details

The IFU is documented in [14], from which the following is taken. It begins with a general discussion of instruction interpretation and the problems involved with pipelined architectures for instruction decoding. The operation of instruction fetching divides naturally into four stages:

Generating addresses of instruction words in the code, typically by sequentially advancing a program counter, one memory word at a time.

Fetching data from the code at these addresses. This requires interactions with the machine's memory in general, although recently used code may be cached within the IFU. Such a cache looks much like main memory to the rest of the IFU.

Decoding instructions to determine their length and internal structure, and perhaps whether they are branches which the IFU should execute. Decoding changes the representation of the instruction, from one which is compact and convenient for the compiler, to one which is convenient for the processor and IFU.

Formatting the fields of each instruction (addresses, immediate operands, register numbers, mode control fields, or whatever) for the convenience of the processor; e.g., extracting fields onto the processor's data busses.

The Dorado IFU performs all these functions. Figure 9 is a block diagram of the IFU pipeline. Recall that the IFU deals in byte coded instructions sets whose instructions are one, two, or three bytes in length. The first byte of each instruction, called the *opcode*, is decoded by table lookup. It may be followed by as many as two optional *data* bytes (known as *alpha* and *beta*, respectively) that are passed to the processor with only slight reformatting. Of course the processor is free to interpret these bytes as it wishes, but the IFU can only do complex decoding operations on the opcode byte. The limitation to three-byte instructions reduces hardware complexity at a considerable cost in speed for longer instructions; bytes after the third must be fetched explicitly by the processor, and the IFU must restart at the proper point beyond these extra bytes.

The IFU decodes an instruction by looking up its first byte in a 1024 word RAM, called the *decoding table*. The additional two bits of address come from an *instruction-set register*. The IFU can emulate up to four different instruction sets at a time. The contents of the table describe the instruction in sufficient detail for the IFU and the processor to do their jobs, so the opcode byte itself is not passed to the processor. Thus, the table lookup does most of the transformation of the instruction; it also governs some minor transformations of the data bytes such as sign extension.

In addition to decoding instructions, the IFU must deal with certain exception conditions as well. The exceptions may be divided into three classes:

1) The IFU has not finished decoding the next instruction, and hence is not ready to respond to a processor demand;

2) It is necessary to do something different (to handle an interrupt or a page fault);

3) There has been a hardware problem; it is not wise to proceed.

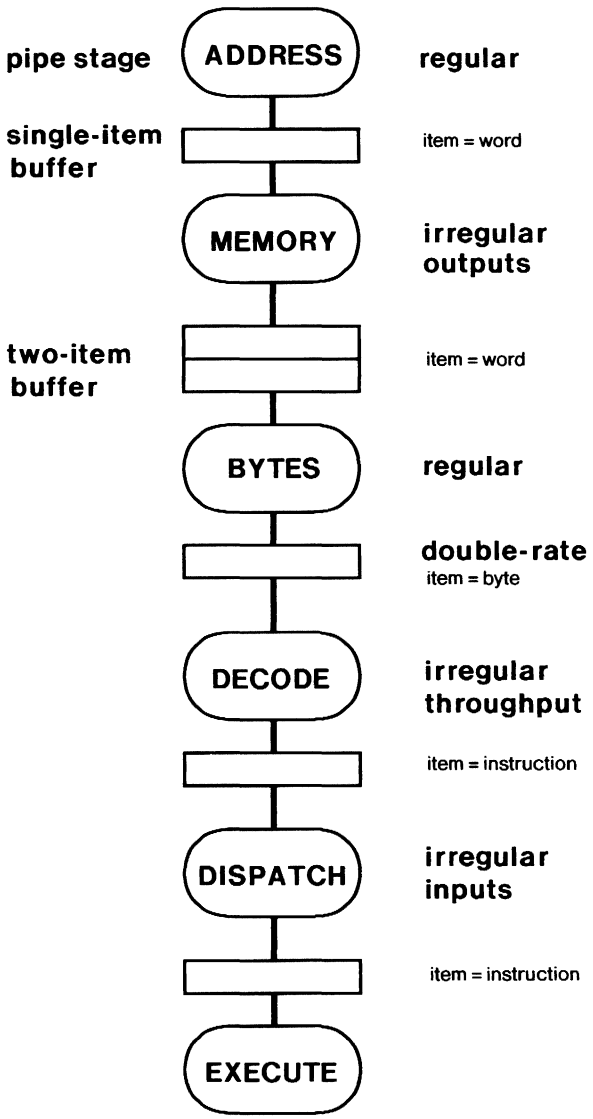


Figure 9: IFU Pipeline Stages

Exception conditions are handled by extending the space of values produced in the IFU and handed off from one stage to the next, rather than by establishing separate communication paths. Thus, for example, a page fault from the memory is indicated by a status bit returned along with

the data word from memory; the resulting "page fault value" is propagated through the IFU pipeline and decoded into a page fault dispatch address which is handed to the processor like any ordinary instruction. Each exception has its own dispatch address. Interrupts cause a slight complication. The IFU accepts a signal called *Reschedule* which means "cause an interrupt"; this signal is actually generated by I/O microcode in the processor, but it could come from separate hardware. The next item leaving DECODE (Figure 9, center) is modified to have a reschedule dispatch address. The microcode at this address examines registers to find out what interrupt condition has occurred. Since the reschedule item replaces one of the instructions in the code, it has a program counter value, which is the address of the next instruction to be executed. After the interrupt has been dealt with, the IFU will be restarted at that point.

Since more than one exception condition may occur at a time, they are arranged in a fixed priority order. Exceptions are communicated only by a dispatch; all exceptions having to do with a particular opcode must be detected before it is handed off to the processor. Thus, all the bytes of an instruction must have been fetched from memory and be available within the IFU before it is handed off.

The IFU takes complete responsibility for keeping track of the emulator program counter. Every item in the pipe carries its program counter value with it, so that when an instruction is delivered to the processor, the program counter is delivered at the same time. The processor actually has access to all the information needed to maintain its own program counter, but the time required to do this in microcode would be prohibitive (at least one cycle per instruction).

The IFU can also follow branches, provided they are program counter-relative, have displacements specified entirely in the instruction, and are encoded in certain limited ways. These restrictions ensure that only information from the code (plus the current program counter value) is needed to compute the branch address, so that no external dependencies are introduced. It would be possible to handle absolute as well as program counter-relative branches, but this did not seem useful, since none of the target instruction sets use absolute branches. The decoding table specifies for each opcode whether it branches and how to obtain the displacement. On a branch, DECODE resets the earlier stages of the pipe and passes the branch program counter back to ADDRESS. The branch instruction is also passed on to the processor. If it is actually a conditional branch which should not have been taken, the processor will reset the IFU to continue with the next instruction; the work done in following the branch is wasted. If the branch is likely not to be taken, then the decoding table should be set up so that it is treated as an ordinary instruction by the IFU, and if the branch is taken after all, the processor will reset the IFU to continue with the branch path; in this case, the work done in following the sequential path is wasted. Even unconditional jumps are passed on to the processor, partly to avoid another case in the IFU, and partly to prevent infinite loops in the IFU without any processor intervention.

The IFU is implemented as a six-stage pipeline (Figure 9). The ADDRESS stage generates the addresses of memory words which contain the successive bytes of code. It increments the program counter by two (there are two bytes per memory word) for each successive reference. ADDRESS

contends with the processor for the memory address bus; since the IFU has lowest priority, it waits until this bus is not being used by the processor. MEMORY is not really a pipeline stage; it is the memory system itself. ADDRESS and MEMORY cooperate to assure that bytes are delivered in the order in which they are requested and that there is always room in the IFU pipeline for data coming from MEMORY. BYTES is a very simple stage which passes the byte stream to DECODE.

The main complications in DECODE are the decoding table, the variable number of bytes required to make up an instruction, the encoding of exceptions, and the execution of jumps. The decoding table is implemented with 1024 x 1 RAMs, which provide room for four instruction sets with 256 opcodes each. The details of the encoding are explained in [14]. DECODE replaces the microcode dispatch address from the table with an exception address, if necessary. If a *Reschedule* is pending, it is treated like any other exception, by replacing the dispatch address of the next instruction item with the reschedule microcode dispatch address. Thus, there is always a valid program counter associated with the exception.

If a *Jump* is decoded, DECODE computes a new program counter by adding an offset to the program counter of the instruction. This offset comes from the alpha byte if there is one, or from the decoding table, sign-extended. The new program counter is sent back to ADDRESS. Jump instructions in which the displacement is not encoded in this way cannot be executed by the IFU, but must be handled by the processor.

The interesting work of DISPATCH is done by the processor, which takes the dispatch address, together with processor state initialization, from the DECODE output buffer. Because *DispatchEmpty* is encoded into a *NotReady* dispatch (another exception), the processor takes no account of whether this stage is empty.

Finally, the EXECUTE stage implements the logic which passes *alpha*, *beta*, and program counter values to the processor as requested. The sequence of data items delivered in response to processor demands is controlled by other fields in the decoding table.

4.4 Input/Output Details

There is no distinct I/O subsystem in the Dorado because the processor and memory implement the functions traditionally provided by DMA controllers. Devices simply conform to the protocols for data and control used over the busses between the processor and controllers. Controllers implement buffers to provide the elasticity needed for asynchronous operation of physical devices and logic for device-specific formatting and control.

The standard Dorado comes with an 80 MByte removable disk which transfers data to the processor at a rate of about 10 MHz over the slow I/O bus, one word at a time. Continuous sequential sector disk transfers require approximately 8% of the processor cycles. Other slow I/O devices include a 3 MHz full duplex Research Ethernet controller, a keyboard, and "mouse" pointing device, and the command and control sections of both high-resolution monochrome and variable-resolution color terminals. Actual transfers of bitmapped images are performed over the

high-bandwidth fast I/O bus to the terminals. A device typically requires, at most, a few hundred words of microcode to implement its function.

4.5 Packaging Overview

The Dorado fits into a very compact package, illustrated in Figure 10. Circuits are mounted on large, high-density logic boards (288 16-pin packages plus 144 8-pin resistor packages per board). The boards slide horizontally into zero-insertion-force connectors mounted in dual backpanels ("sidepanels"). Boards are 0.625 inches apart. This density makes it possible to reconcile the goals of size and capability. Certain sacrifices are made, however. For example, it is not possible to access every signal with an oscilloscope probe for debugging and maintenance. We make up for this by providing sophisticated debugging facilities (Section 5.5), diagnostics, and the ability to incrementally assemble and test a Dorado in steps from the bottom card slot upward.

The entire machine, including disk, displays, and network interfaces, is implemented with approximately 3200 medium-scale integrated components, mostly of the ECL 10K family. In addition there are up to 4 storage modules, each with about 300 64K RAMs and 200 MSI components, for a maximum of 8 megabytes. The total volume, including power and cooling, is about .14 m^3 (4.5 ft^3); this is without any enclosing cabinet, however, and the open machine is quite noisy. Including the 80 megabyte removable disk, it requires about 2.5 KWatts of AC power.

5. Retrospective

This section revisits the processor, memory, IFU, and I/O sections of the machine for comments and evaluation. It then introduces the other major components of the Dorado project: debugging aids and diagnostics, design automation and fabrication, and the management structure, all of which were vital to the success of the project.

5.1 Processor retrospective

We chose a highly encoded ("vertical") microprogrammed processor with fully writeable control store, as opposed to a hardwired processor or one with an unencoded ("horizontal") micro engine. Why? First, control stuctures for high-performance machines tend to be complex and difficult to master. There is very little formal methodology for the design of such systems at their lowest level, and it is difficult to guarantee correctness in any convincing way. One can master this complexity by expressing it in a programming language (microcode) that is powerful and flexible, so that it may be corrected and also evolved and improved over the lifetime of a machine. Second, a machine for multiple high-level languages is much more effective if it allows each language to define its own

target architecture, rather than forcing very different kinds of compilers and interpreters into a mold dictated by a fixed machine language. Microprogrammed architectures fill these needs.

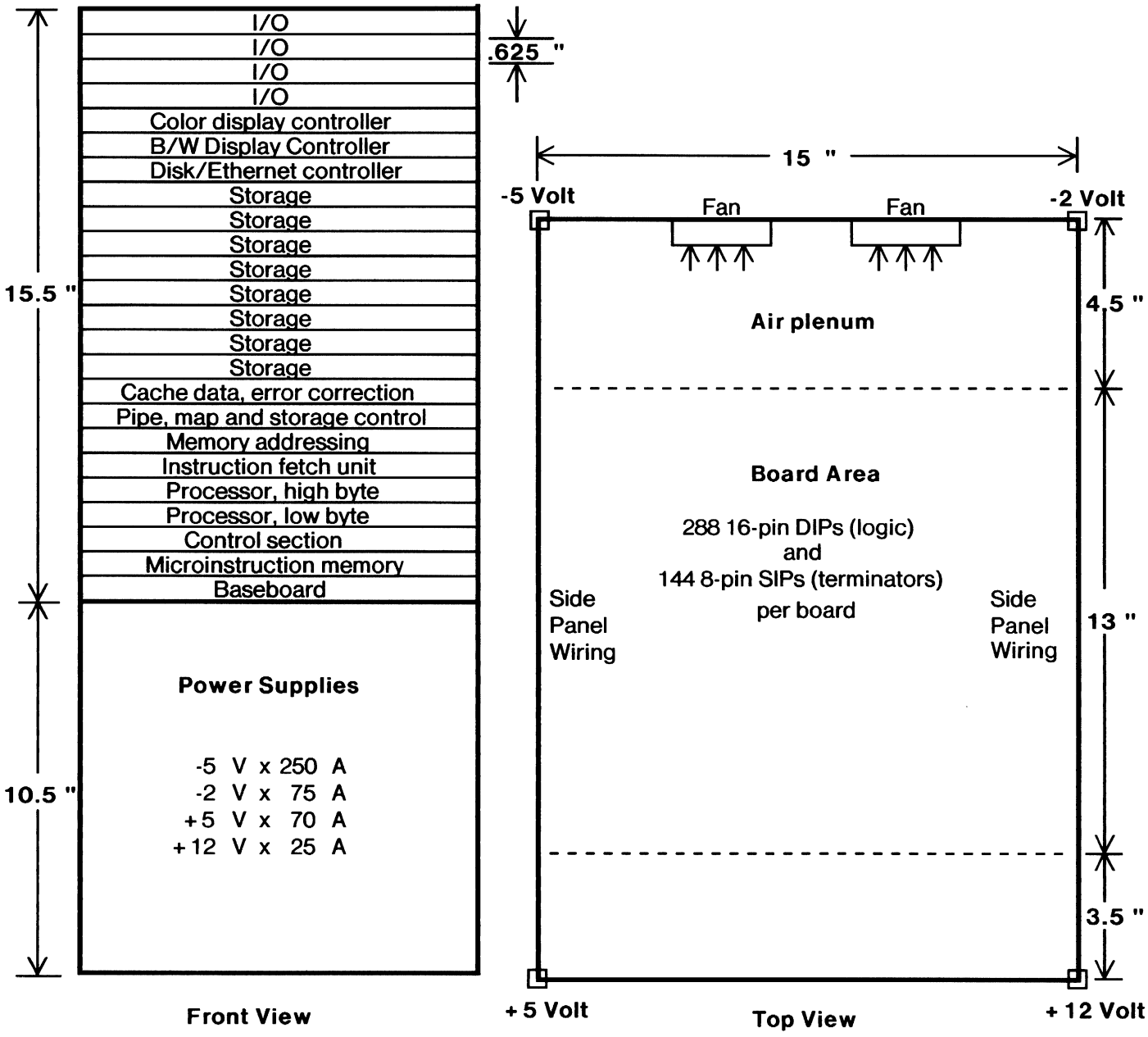


Figure 10: Dorado Chassis

High-performance machines, however, must still be hardware intensive and cannot rely on microcode to make up for a lack of functional richness or operand processing capability. The ratio of microcycles per emulated instruction must be small, and for the simplest macroinstruction a ratio

of 1:1 is desirable. It would have been a mistake to push too much functionality off into the microcode. In fact, as we shall discuss, language emulation could have been substantially speeded up had some more hardware been devoted to managing process context switching, a frequent and somewhat costly event.

The Dorado has a very compact microinstruction, with context dependent interpretation of nearly all fields and a highly encoded field for specifying "next instruction" addresses. There are several motivations for this structure. One is raw cost of components. For very high-speed RAMs, with under 20 nanosecond access times, price is an important factor. In 1977, such RAMs sold for nearly $30 each. The physical size of the RAM array is also important, as additional wiring delay for a large array contributes to overall access time. So, vertical encoding minimizes microstore chip count and cost at the expense of fast microinstruction decoding at execution time. The ECL technology mitigates this problem by allowing the designer to build a decoder that operates in three or fewer stages of combinatorial logic, or around 10 nanoseconds delay.

When encodings are tight and complex they can be hard to deal with. After three major iterations, we were able to invent a very successful encoding which provided both compactness and nearly maximal parallelism on control and data resources available in the processor. However, the complexity of the encoding had to be hidden in a very good microassembler and instruction placer, a loader, and an interactive debugger. We were successful in developing such tools. The tools give the microcoder/debugger the illusion of dealing with a linear, unencoded address space and a language similar to conventional machine languages without regard (most of the time) for details of the encoding. In fact, the Dorado is certainly the easiest processor to microprogram of the several available in the Xerox family of processors.

The data handling section of the processor was redesigned several times to increase quantity of and parallel access to resources. There are 256 general-purpose registers, four hardware stacks, and a number of special-purpose registers commonly used by the microcode. The two ALU inputs are symmetric so that any common source of data can be connected to either side of the ALU. Similarly, the memory address and data registers are coupled tightly to the processor and are treated just like a general register or a temporary, thus bringing memory operations immediately within the scope of the data section.

The multitasking architecture, perhaps the most unique feature of the processor, has proved itself, particularly in the areas of uniformity and flexibility. There is a single microlanguage in which all emulators and device controllers are expressed, and there is a single virtual address space for all devices and language emulators to use for communication and data. By mastering a small set of skills, a designer can easily add new instructions to an instruction set or a new I/O device to the machine. Multitasking and the design of the memory system allows these additions to be made with relatively little concern for impact on already installed devices. Fixed priority of tasks, as opposed to round-robin scheduling, has worked thus far. High priority tasks are very carefully designed and coded to execute only two microinstructions per wakeup (except for the disk. See section 5.4).

Of course, there are possible improvements. We now realize that language emulation could be substantially speeded up by concentrating hardware resources on context switching and on keeping call stacks cached in high-speed registers; the IBM 801 project [21] and the RISC [20] project have emphasized some of these areas and realized significant performance improvements for relatively low cost.

The processor contains a bottleneck as all data must pass through the ALU in order to reach a destination, so there is only one destination site per microcycle. Some other external paths could be useful for, say, saving and restoring stack top entries in registers. A frequently performed operation, especially by I/O microcode, is the generation and use of 16-bit constant values. The processor can generate only an eight-bit immediate operand. It requires two microinstructions to piece together two bytes into a constant before storing or sending it over the I/O command bus, for example. The Alto, to solve this problem, contains a small ROM with two hundred or so "commonly used" constants; such a constant memory did not mesh into the Dorado processor architecture and wasn't used.

The data section occupies two full boards in the Dorado with considerable overhead and chip count (~100 chips) for high-speed communication between boards. It might be possible, by reducing some functionality, to collapse the implementation onto one board and save the overhead and complexity of the larger implementation.

5.2 Memory Retrospective

It is really the memory, rather than the processor, that is the heart of the Dorado. The cache and cache control architecture have been particularly successful. A great deal of effort was expended in the design and implementation which produced a pipeline with only two cycles of latency (one for CacheA, the second for CacheD) and a single cycle of throughput to the processor, an essentially optimal design. For the byte-coded target languages emulated, this cache not only functions almost as fast as internal registers, but also provides hit rates of greater than 99% in ordinary cases. It has a write-back rather than write-through scheme for managing dirty blocks of data, and this scheme reduces the bandwidth required between storage and the cache by a factor of about seven for write operations.

Because of the resource management and contention resolution performed by the memory control logic, there is essentially no effort required on the part of microcoders to deal with contention, latency, implementation quirks, or error conditions. For example, once a Fetch has been executed by the microcode, *any* subsequent microinstruction may request that the fetched data be used. If the data is not available for any reason, the *Hold* mechanism will be invoked and the processor delayed only until the data arrives. Contrast this to common microprogrammed architectures where it is either forbidden that data be accessed before a certain (perhaps worst case) time or required that data be accessed at exactly a certain cycle after the request is made. It is also possible to write the various I/O driver microcode segments without regard for what the emulator or other I/O

microcode is doing with the memory; since the processor switches tasks completely asynchronously and microcode can make arbitrarily sequenced accesses, this reduces a potentially exhausting problem to a simple one.

Another desirable feature of the architecture is the ability for all clients of the memory to deal only in virtual addresses. It is common in many systems for I/O devices to deal explicitly in real addresses and to negotiate with memory management software for both buffer management and explicit translation from virtual to real addresses. As the processor initiates all transfers and everything goes through the Map, these issues disappear in the operating systems for the Dorado.

The virtual memory address size, the MapRAM, and main storage are all implemented with an eye towards growing chip technology. This was relatively easy to do and is important over the lifetime of the Dorado, helping to extend its useful life in the face of voracious software systems. In fact, the original Dorado memory modules were constructed using 4K X 1 MOS RAMs in 1976. Evolution has since produced the 16K (1977) and 64K (1981) chips, and modules are upgraded by simply plugging in the higher density devices and reconfiguring a few jumper wires.

But, everything has its price. The control structures for the memory are tightly woven from logical steel wool, a necessary evil if one wants both compactness and speed without custom LSI components. As a result, memory control logic had numerous obscure and nasty design bugs that the original implementors wrestled with. Such bugs often have delayed effects due to pipelining; an error can manifest itself long after its cause has disappeared (a feature of pipelined systems in general). Although production machines seem to contain no design errors, difficulties in debugging new machines result from the complexity of this logic. Further, with any system containing dynamic RAMs, it isn't possible to single step clocks and maintain refreshed data. The diagnostic system has several options for single stepping clocks with and without refresh enabled; this helps find control failures where data values are not critical.

One aspect of the architecture, memory latency on cache misses, could be substantially improved. Ideally, once CacheA detects a miss, the desired word should be read and delivered to the processor within the memory access, bus transit, and error correction time. Had more resources been devoted to the storage-cache transport mechanism, this ideal could have been achieved. As implemented, an entire 16-word block of data (on a 16-word address boundary) is transported, in order, up the bus and into the cache. After the block is in place, the cache is accessed again to fetch the desired word and deliver it to the processor. This method results in about nine additional microcycles of miss latency over the ideal scheme. We should note that this inefficiency does not degrade performance as much as it might seem, since only 8% of the total available cycles is spent in memory wait (Hold) time.

The storage modules are now very reliable and easily manufactured; they were converted early in 1979 to printed circuit board technology because of their regular arrangement. However, the original storage modules, manufactured on a custom stitchweld board, were a major headache due to careless distribution of different logic families (ECL, TTL, and MOS) on a single board. Crosstalk between ECL and TTL clock signals and badly distributed sheet currents in the ground plane made

these modules unreliable. Major redesign by an expert electronic engineer was necessary to reform and lay out the board to eliminate problems due to logic family mixing.

The very high I/O bandwidth (530 Mbits/sec) is a major factor in the cost of the memory system. The many busses that operate in parallel, the fully segmented pipeline with its "extra" automata stages designed to keep up with the data transport rate, the separation of the error correction code generator and error checking logic to enable them to run in parallel, and the ability of the cache to handle multiple misses simultaneously (for the emulator and for an I/O task) are all motivated by the I/O service requirements. Less stringent requirements would have made the memory system quite a bit smaller and simpler.

5.3 IFU Retrospective

The IFU produces a doubling of emulation speed over the same machine with emulation done strictly in microcode. In fact, the Dorado is carefully partitioned so that the IFU is not required for emulation. It is possible to load a microcoded interpreter and do conventional instruction fetching and decoding without hardware assist, unlike some modern architectures in which the instruction fetch unit and the execution unit are not fully separated [11]. Dorado Model 0 ran exactly that way, without an IFU.

It is the IFU, in conjunction with the processor, that fulfills the design objective of being able to execute a simple byte code in a single microinstruction. There is a separate high-speed bus (the IFUData bus) between the processor and the IFU, over which *alpha* and *beta* bytes move to the processor in parallel with register and memory accesses. The processor uses this data like any other operand source without needing to store IFUData into a register. Another parallel bus, IFUAddress, is the dispatch address source for the microstore and is used directly to indicate "next dispatch address" information for emulators. Second, a technique called *instruction forwarding*, detailed in Section 4 of [14], allows certain cleanup actions to be left over from one instruction to another, thus saving microcycles. For example, to push a variable onto the stack top, it needs to be fetched, then pushed. However, if all emulators have multiple entry points so that emulation microcode can deal with the top of stack located either on the stack itself or in the memory data register (*FetchReg*), the extra push instructions can be eliminated (forwarded). This technique allows simple instructions to require a single microinstruction for emulation and saves approximately 8% of the execution time in straight line code without much cost in added microinstructions for multiple entry points.

A major advantage of the IFU is its programmability. All decoding takes place via table lookup, and these tables are implemented in RAMs which are loaded by initialization microcode and can even be modified "on the fly" if needed. As a result, instruction sets can evolve, and new instruction sets for experimental languages can be derived. In fact, after approximately four years and hundreds of thousands of lines of Mesa code were completed for the Xerox 8010 Star Network Systems product, the Mesa instruction set was redesigned and achieved a 20% improvement in overall code compaction [23]. Note that this revision required nothing from the large Mesa client community

other than recompilation of sources once the microcode, IFU, and compiler updates were done by the Mesa system developers.

Unfortunately, the IFU is also constructed of logical steel wool, like the memory control. Its complexity is due to the variable length instructions, to the need for a global pipeline architecture, and to the desire to deliver one byte code per cycle on demand. Also, board site limitations required that the IFU be squeezed onto a single board, since the processor and memory subsystems had already overflowed onto more boards than originally planned. As a result, there is not enough inter-stage buffering to smooth out the pipeline flow irregularities due to memory response time and to jump handling.

The six-stage pipeline in the IFU is too long to handle frequent code branches well, since they cause pipeline flushes to occur. It is common for highly structured code to consist of branches every five instructions; even with good branch prediction, long delays are introduced while the pipe is refilled. Without the aforementioned board site restrictions, it would have been possible to reduce the length and complexity of the pipeline by adding more buffering, and perhaps to enhance branch prediction capability by dynamically recording branch results and following previous paths [29].

The IFU offers little help for long, complex instructions such as process switch or RasterOp [19]. Even subroutine calls in Mesa require multiple microinstructions to interpret [15]. It is even less effective for languages like Smalltalk, whose target language typically requires thirty to forty microinstructions to execute a macroinstruction; in this case the IFU offers marginal speed-up, and instruction forwarding to save a single microinstruction is essentially useless.

5.4 I/O Retrospective

There is little to be said about the I/O subsystem. Device controllers are first-class citizens, serviced on demand from the processor via the virtual memory system. High-performance devices make good use of the microtasking scheme and the memory bandwidth available. A high-resolution, large format (1024- X 808- pixels) binary terminal display and a full color (640- X 480- pixels) display are supported, without special frame buffers, with little degradation of emulator performance because the bandwidth for refreshing the displays would be essentially unused were they not present. These fast I/O devices require only two microinstructions per transfer to manage 16-word block transfers over the I/O busses, and these busses interact with main storage only, bypassing the cache whenever possible. Future controller development is simplified by the ability to borrow standard "canned" interface logic from existing controllers. Memory bandwidth is still available for future controllers.

The only drawback to the existing set of controllers is the disk controller, which was one of the earliest designs completed and runs on the slow I/O system instead of the fast I/O system. It is odd to call a 10 MHz device "slow," but the processor is capable of handling the disk over the word-by-word I/O bus. This does use the cache and degrades cache performance some (we have no hard data on how much), and unfortunately requires that the disk controller run as the highest priority I/O device in the machine in order to access sequential disk sectors without missing

revolutions. The disk microcode takes up several hundred microinstructions and is convoluted. Data transfers over the fast I/O bus with control over the slow I/O bus would be a much cleaner implementation, but is complicated by disk format requirements for small fields (as small as two words) in the disk format. No resources have been expended to remedy this situation.

5.5 Debugging and Diagnostic Aids

The Alto was a natural choice to help us bootstrap to the next generation. We attached an Alto, via a simple parallel interface, to a special port on the Dorado processor and wrote a display oriented "console program" called *Midas*. A small amount of dedicated hardware in the processor allows Midas to completely control the Dorado via its umbilical cord. Midas controls the starting, stopping, single stepping, and running rates of clocks in the Dorado, and it can cause any microinstruction to be placed in the microinstruction register and executed by the processor. As a result, all registers and memories accessible to the microcode are accessible through the console program as well, which presents to the Midas user a screen full of fields in which any named register or memory location can be displayed and/or written at the user's discretion. Several formats of readout and typein are available, including symbolic format for microcode disassembly when examining or modifying the microstore.

In addition to memories and registers, Midas has read-only access to two thousand hand-selected (hardwired) "vision" or "scan" signals. There is a 2000 element serial multiplexor spread across all the boards in the machine, and the Midas Alto can scan out all multiplexed signals in under half a second. The Dorado scanning system is described in [5]. Typically, these signals are used to display the states of automata and critical control signals in the hardware, or to directly read out bus contents.

The combination of single stepping, scan signals, and microcode jamming allows Midas to provide a predictive simulator and a history list for the hardware. The simulator is executed within the Alto program and the states of as many of the scan signals, memories and registers as possible are calculated for the next microinstruction to be executed. Midas then causes that instruction to be executed and reads out the hardware state, comparing it to the simulated state and announcing any discrepancies. The user can also single step and symbolically observe the state of internal signals, a technique which proved very useful for initial checkout of undebugged prototype subsystems.

Midas serves many other functions. It loads microcode into the microstore and allows the user to set microcode breakpoints and to single step or run through microcode routines as needed. Other features provided by Midas are a suite of very low-level diagnostic and exerciser routines which execute via the umbilical, and error reporting for low-level unrecoverable errors detected by the hardware.

Originally, Dorados were completely dependent on Midas for bootstrapping, including clock speed adjustment, setting of all registers and memories with good parity data, and downloading microcode. Dorados can now do this independently because another logic board, called the *baseboard*, has been installed in the machine. The baseboard has a microcomputer on it which is programmed

to simulate Midas and to execute a standard bootstrapping sequence on command or at power-on time. The baseboard also does environmental monitoring of temperatures, voltages, and currents to prevent failure due to meltdown or overload.

It would be difficult to overemphasize how important the Midas system was (and is) to the commissioning and maintaining of Dorados. Software and system expertise were concentrated on Midas instead of on building either a logic level or microcode level simulator for the Dorado. Basically, it was decided to build and debug the hardware in place rather than simulate it first. That seems to have been a good decision, because the tools and skills developed for the prototype apply to both reproduction of Dorados and to new projects.

As the machine was designed and implemented, sets of microdiagnostics were specified and microcoded. These diagnostics are aimed at incremental assembly of the Dorado hardware, from the baseboard on up. As each board was built, it went into the prototype chassis as the uppermost board in the system, fully accessible to logic analyzers and oscilloscopes. A diagnostic was prepared to exercise that fraction of the machine installed and available at that point. As the machine grew, more and more elaborate diagnostics could be run, relying on the already debugged lower sections of the machine to exercise the undebugged new boards in the upper chassis. These diagnostics are still used to debug new Dorados, but the production line does not normally use the incremental assembly scheme. Instead, a working Dorado station debugs one new board at a time within a fully functioning system.

System level diagnostics for such areas as instruction set verification or exhaustive memory tests have yet to emerge. Diagnostics for the disk, Research Ethernet network, and display terminal systems still run in Alto emulation mode and were borrowed from the Alto world. There are a number of problems with this diagnostic gap, and they all stem from the nature of digital system *collapse*.

> *Digital system collapse*: The set of states attained by a digital programmed system in which nothing but the diagnostic programs will run to completion.

Diagnostics at the microcode level are a maintenance headache. There is one person responsible for nearly sixteen thousand lines of microcode. Technicians were not accustomed to dealing with microdiagnostics for repair of machines; the designers had to teach machine architecture and debugging techniques before Dorados could be produced. New systems, both hardware and software, constantly appear to challenge the technical support and maintenance people.

5.6 Design Automation and Prototype Fabrication

A new design automation (DA) system had to be in place before the project could proceed, and it is this system that really made the actual implementation and commissioning of the Dorado prototype possible. Although by today's standards the DA system has shortfalls, it was a marvel in 1976. As the project evolved so did the DA system, and every effort to refine or enhance the functionality of this system has paid off. Originally intended for use with a custom, point-to-point wiring technique called *stitchweld*, the system today handles stitchweld, Multiwire, and multi-layer

printed circuit boards for a variety of digital environments. The DA system insures that, at essentially all times during a project, all logic design documentation is current, accurate, and available on-line to anyone on the design team. It also provides automatic version and revision control, and, most important, a uniform syntax and semantics for expressing hardware designs and implementations across projects and designers.

The sole user input to the DA system consists of stylized logic diagrams, at the chip level, of logic designs. Diagrams are created via an interactive graphics illustrator running on any Alto (or Alto emulator). By following a simple set of conventions designers quickly build, from prestored libraries of chip macros, their logic designs. Once the diagrams and board layouts are complete, all syntax checking, semantic interpretation, and generation of intermediate files, net lists, wire lists, and other documentation are done automatically by the DA system. There is no intervention or additional information supplied by the designer. This system works both for the creation of new logic boards and for incremental upgrading of existing boards during debugging. The user simply revises the logic diagrams using the illustrator, and the DA system automatically compares the new design against the existing one to generate "Delete/Add" instructions for the stitchwelder.

Coupled with the Midas debugger, this system can be compared in style to high-level language programming systems. The interactive illustrator acts like a text editor for a high-level description of a logic board (a "program module"); the DA system then "compiles" this description into lower levels of abstraction, without user intervention, culminating in a wirelist to drive a semi-automatic wiring machine and a stuffing list for component type and location specifications. Midas then acts as a symbolic level debugger/interpreter when the physical board is installed and checked out.

There are two major shortfalls of the DA system. One is the lack of hierarchically organized graphics for defining logical pieces of a system, which can then be hooked together to form larger pieces. All registers, for example, are explicitly drawn by the designer instead of expanded by the system from a prototypical register "bit-slice." The SCALD (Structured Computer Aided Logic Design) [28] system designed for the S1 computer project incorporates such hierarchical graphics which yield a much more compact, structured representation of a digital system. Automatic hierarchical construction from prototypes markedly reduces specification errors. Many of our initial Dorado bugs were simply "typographical" in nature. A second shortfall is the lack of timing analysis tools that give a true picture of the expected performance of a logic board once its design and layout is complete [16]. Although we made some attempt to implement such an analysis tool, the attempt was not very successful or complete.

5.7 Project Organization

Exactly how was the Dorado specified, designed, implemented, and commissioned? And, why was it done in a research environment? The second question is easy to answer: the research laboratory determined that it needed a Dorado class machine to pursue software systems research over the coming five to ten years, and there were no alternatives. The Dorado project began at PARC in

1975, then moved to the Systems Development Department, which was formed in order to transfer distributed computing technology into future Xerox product lines. Systems Development decided, after design sketches for the Dorado were completed, that a smaller, less expensive, less powerful machine would be required for the envisioned systems applications, and the Dorado was shelved in favor of other processor design projects. By this time, Research had already made plans to employ Dorados, and after many meetings and much soul searching within the Computer Science Laboratory, a small group of people made the commitment, in June of 1977, to complete the Dorado implementation.

This team of about ten individuals was led by Butler Lampson, who had been one of the two system architects (with Chuck Thacker) since the beginning of the project. Lampson continued as the senior architect and also implemented the control section of the memory system. The team members were mostly veterans of one or more major systems (hardware and software). Lampson, Ed McCreight, Gene McDaniel, and several others had worked on the MAXC time sharing system [4] and the Alto at PARC, both predecessors of the Dorado effort.

Team members proved highly motivated once their commitment to the project was made. Subsystem teams (processor, memory, storage, IFU, I/O devices, microcode, diagnostics, Midas, design automation) formed naturally according to the skills and interests of the people involved, with many members participating on multiple subteams. As initial pieces of the machine came into existence their designers worked on later pieces, eventually building up the global overview of the entire system necessary for system integration. There was an adequate mix of skills and experience to eventually solve or finesse the myriad of problems that arose over the next several years.

Weekly meetings were held at which the week's progress was reviewed and planning of "open" schedules was done. There were no hardline management decisions or deadlines imposed. McCreight and Severo Ornstein served as co-managers of the project in addition to making technical contributions. It was the attitude of the laboratory manager (Bob Taylor) that the team be free to manage itself, and to make even major decisions, like the complete redesign from Model 0 to Model 1, without external pressure. This attitude was consistent with the style of research projects within the laboratory (small groups proceeding independently) and kept the team members from feeling isolated from their colleagues. In addition, all activities and personnel were located in the same building, within a few steps of one another. Team members were in constant communication personally, via electronic mail, and via the on-line DA system. Design and implementation details were thrashed out on a daily basis, and all logic designs were subject to careful peer review before being welded into a prototype.

Nearly all efforts were applied to the actual prototype itself, or to the tools and diagnostics directly needed to construct and debug the prototype. In particular, no code simulators were built, and microcode was debugged directly on the prototype hardware via the Midas interface. A software simulator specifically for the IFU was coded in order to test and debug implementation alternatives. Since much of the microcode has real-time applications for I/O controllers, simulation is less useful in this microtasking architecture than in a conventional system.

6. Observations and Commentary

The Dorado is not the machine that would have been designed just for research. Its architectural demands for both high-performance emulation and high I/O bandwidth are inherited from the Systems Development Department specification of a high-performance processor for imaging systems (composers and printers) and for office information systems. Although we take advantage of this duality in the Dorado to easily integrate imaging systems with information processing systems, they could have been more traditionally separated. We are pleased that we do not have to make that distinction.

One major impact on the daily working life of the research staff, as Dorados became available, was the breaking of a "psycho-computational" barrier that has plagued expert system implementors until recently. It has always been the case that system experts have felt hampered by their tools; no matter how good the batch, interactive remote job entry, timesharing, or small-scale personal machine was, it wasn't enough. Far too much real time was expended waiting for the machine to compile or load or respond in some other way, and far too much programming effort was expended to try to overcome resource limitations within machines. With the advent of the Dorado, experimentors took a large step towards becoming weary from concentrating on their work instead of becoming frustrated by their tools. This really changed the scope and style of research attempted and accomplished at PARC. It seemed as if a few hours on a Dorado were more productive than an entire day on a smaller machine running an identical programming environment [22]. With so much processing power available, very large systems (notably Smalltalk personal environments like PIE [8]) that were previously barely demonstrable became viable for daily use. Fully integrated programming environments in three major languages (Interlisp, Smalltalk-80, and an extension of Mesa called Cedar) were produced and today are the development environments of choice for nearly a hundred scientists and engineers. An informal survey of Cedar users [9] found that expert users claimed subjective productivity improvement factors ranging from two to five over Cedar or similar programming environments running on less powerful machines.

The Dorado came into service quickly because of nearly complete "backward emulation" of the Alto. Backward emulation provided a stable, working software environment to implement as a first goal. Having a working language (BCPL), operating and executive systems, and many applications as tests guided the hardware/microcode effort on the project from the start. Once the Alto emulation really worked, we could proceed to develop new programming environments with the confidence that a very large fraction of the hardware was operational. Contrast this to the prospect of developing all new hardware, system software, and applications contemporaneously, without a stable platform to rely on.

The experience gained on the original Model 0 prototype was crucial to the success of the project. The design team realized that the lessons learned were too valuable not to be immediately applied. We were willing to discard the functioning prototype and work an additional 14 months (5 months of which overlapped completion of Model 0) to produce the Model 1. Although physically

nearly identical to Model 0, Model 1 has much larger memories and register banks, a more versatile microinstruction encoding, a higher performance cache, and the IFU.

There were a few worries that at least the author had at the inception of the project that turned out to be groundless. The first was a common misconception that ECL logic, because of its very high-speed, would be difficult to work with. All of the team members had experience with TTL based systems only. ECL is in fact easier to work with than Schottky TTL in a properly designed environment with controlled impedances, adequate grounding, and termination resistors. Except where we carelessly mixed logic families on the original storage boards, we never had any problems with ECL logic; the "digital illusion" model of perfect components and binary logic always worked. ECL 10K has other advantages, such as uniform 16-pin DIP packages and consistent pinouts, which earlier TTL families lacked.

High component density and limited access to boards in the board stack was also a gamble that worked. Recall that the physical structure of the machine is such that it is easy to get at only the top board and the two side panels with instrument probes. Other components are hidden, and can be accessed only very painfully, essentially one pin at a time, by power cycling the machine and removing boards to attach low-profile probes to particular pins. This procedure was used a few times, notably during attempts to speed up the basic cycle time, but was by and large unnecessary. By careful use of the information provided via Midas, construction of a useful suite of diagnostics, and some cleverness on the part of debuggers to deduce information from side panel signals, it was possible to live with the limited access and still work effectively. Packaging technologies are available today that combine high-density board stacks, cooling, and access to every pin by "unfolding" a machine, but these techinques still don't allow the compactness available in the Dorado system. Incidentally, we weren't able to design any kind of extender board that would allow the system to continue to function.

Perhaps the most pleasant surprise for the design team was the success in turning over the Dorado to manufacturing and maintenance personnel. Would every new instance of a Dorado have to be brought up with some help from designers, due to the overwhelming complexity of boards in the system? Fortunately, no. Although one member of the team continued to serve as liaison for manufacturing and to assist with the toughest problems, all the other team members quickly dispersed to new jobs (both within and outside the company). Dozens of Dorados have been manufactured and installed, working perfectly, with no intervention from the designers. This is a tribute to the skill and enthusiasm of the production and maintenance people at PARC.

Everything did not come up roses, however. There are two major disappointments with the existing Dorado, and other disappointments with "the machine that might have been." The existing machine fails to run at design speed and fails to work reliably in office environments. The original design specification called for a 40 nanosecond microcycle. Although sections of the machine can execute at a 40- nanosecond clock speed, fully functional prototypes operated at 50- nanoseconds, and production machines operate at 60. In choosing a very simple model which tried to simplify timing delays introduced by wires and board transitions, we were naive. A further introduction of

time penalties occurred after changing the point-to-point prototype wiring technology (stitchweld) to a Manhattan routing multi-layer production technology (MultiWire and printed circuit). Also, the complexity and cleverness employed in the logic implementation were heavily influenced by the very high-speed goal. If we had realized the limitations on clock speed earlier, the machine would have been simpler, with equivalent functionality. A really first-class timing analyzer, such as the one eventually developed for the S1 Mark IIA project [16], is crucial for accurately designing and implementing a machine out of a high-speed technology like ECL.

The Dorado, in spite of its compactness, turned out to be too big, too hot, and too noisy. We were unable to solve power distribution and air circulation problems within a noise-abating enclosure well enough to maintain the PARC style of physically distributed personal computers located in offices and powered by conventional 110V AC wall circuits. After numerous attempts, we simply finessed these problems by restructuring machine rooms previously devoted to time-sharing systems. Dorados at PARC are now rack mounted and installed in "free air" in machine rooms with cables strung to terminals in individual offices. This actually improves the quality of life in offices considerably, but makes cable installation and machine room maintenance an ongoing headache. It also discourages the development of experimental devices and controllers, since it is much more difficult to attach a new device, say an input scanner or digitizing tablet, to a machine that is physically far away. Commercial devices are usually designed for shorter cable lengths than we now require.

With respect to the machine that might have been, Lampson claims that by reworking the processor to enhance high-level language context switiching and by reducing the I/O bandwidth, the Dorado would be half its physical size and have twice the emulation performance. Such a machine might be a better match for the software intensive research activities at PARC.

Dorados, we had hoped, would begin production by mid-1978. Two factors made the project considerably later. First, the redesign to Model 1 added about nine months to the prototype production stage. Second, moving to production technologies involved letting go of complete in-house facilities and contracting with outside vendors. It took over a year to get boards rolling successfully off external production lines, due primarily to very long turnaround times (3-4 months) per iteration, and to the "pioneering" nature of the work (larger, denser circuit boards with more layers than had been previously implemented by the vendor). This was not fun.

In summary, the overall architecture of the machine has been successful, and the flexibility and extensibility of the machine is actively being exploited. Microcode support for floating point operations and microcode/hardware for a full-color display system were added after the machine was in service. Recently, new instruction sets have been implemented resulting in enhanced functionality, increased code density, and faster execution. New boards for implementing data encryption services, random number generation from white noise sources, and small amounts of "stable storage" used to implement transaction based systems [18], have been designed. The design automation system has been used for dozens of projects throughout the Xerox Corporation, and has been extended to encompass automatic generation of PC boards.

7. Speculation

What are some of the things we might have done differently? We have already noted the possibility of reducing resources devoted to high-bandwidth I/O capability and directing those resources (chip count, designer attention, etc.) to support for language emulation. Although we deliberately avoided special-purpose hardware targeted at a specific language, most high-level languages share certain ideas that could benefit from hardware support. For example, array bounds are frequently checked in microcode, and NIL dereferencing causes spurious page faults; some hardware assistance in the form of bounds registers or comparators could help. Except for the *ref* and *dirty* bits in the Map, the hardware doesn't enhance automatic storage management schemes. Some provision for more memory protection in hardware, or a small number of isolated address spaces might have made implementing debuggers or automatic allocators (garbage collectors) easier. This is a complex issue, involving tradeoffs between the advantages of isolation and the difficulties such isolation produces with respect to sharing and cooperating processes, beyond the scope of this discussion.

Perhaps we should have built and used a viable timing analysis program early in the project. I say perhaps, because it is conceivable that the project would not have proceeded had the "bad news" been known early on. It is difficult to build analysis programs that will handle non-trivial systems; we implemented one to analyze Model 1, with limited success. Fortunately, the Dorado clock system was simple and uniform throughout the machine, so we were able to simply lower the clock rate until reliable performance was achieved, after burning out on speed enhancement efforts.

There was relatively little we could do about the problems that were a result of inexperience, such as the packaging and cooling problems or the production technology transfer difficulties. Those were essentially painful learning experiences which in the future we will try not to repeat. They were costly in time and effort.

What might we do today, were we embarking on a similar processor design project? It's certain we would go to a full 32-bit architecture for both data and address spaces. Technologies for implementation could be semi-custom ECL gate arrays, commercially available, and a mix of ECL 10K and ECL 100K logic. An alternative would be fully custom VLSI chips. Timing and logic verification extensions to the design automation system should be developed, although it is possible today, as opposed to just a few years ago, to purchase commercial services that fulfill these functions.

Processor/memory design should emphasize context switching speed and "lightweight" processes, as larger and larger software systems use more and more processes and procedures. Stack caching schemes such as those proposed for the RISC architecture [20] and high-speed, low-overhead control transfer functions [15] should be supported in the processor.

Should the architectural style of the Dorado, founded on the Alto, be continued? The answer is not clear. When memory and processors were expensive the multitask sharing was justified. With the advent of VLSI, it is less important to keep a processor busy most of the time, and one can justify constructing multiprocessor systems in which some of the processors do relatively little, but

the structure of the system is simple and expandable. It is more important to develop high-performance hierarchical memory systems (caches) and high-performance instruction decoder units that can be instantiated multiple times using VLSI. Memory components are cheaper now, but large multi-port memories are still expensive due to multiple busses, switches, and contention problems that can result in poor performance. It is likely that the Dorado is the last machine of its line but, like most dinosaurs, it will have a very long tail.

Acknowledgments

The project could not have been done without the support of the Computer Science Laboratory and the patience of its members. Appendix B lists the personnel directly involved with the Dorado project and their main areas of contribution. Each member of the team should be commended for high professional standards and a spirit of cooperation vital to completing a project of this difficulty with a minimal size staff. Finally, special thanks go to the people who took up where the design and commissioning team left off and who are responsible for the ongoing success of the Dorado.

Appendix A: Dorado Project Chronology

1975 **January-December**: Initial concept and staffing.

1976 **January-December**: Concept refinement, design automation system initial implementation, logic design sketches for microcontroller, processing unit, instruction fetch unit, storage module.

1977 **January-June**: Project re-evaluation. Some logic design sketches for display controller and memory system.

June: Project moves from Systems Development Department to Computer Science Lab. Dorado team formed.

June-December: Model 0 prototype implementation progresses on control, processor, and memory.

1978 **January-June**: Complete Model 0. Alto system emulator without IFU running. Begin Model 1 specification and logic design.

June-December: Refine Model 0 and continue Model 1. Virtual memory support, page fault and error handling hardware completed. First Mesa, Smalltalk, and Interlisp emulators.

1979 **January-March**: Debugging of Model 1, with baseboard, control, processor, new memory system, and IFU design in progress. Released Model 0 to users. Extensive microdiagnostics developed; training of technicians begins. I/O controllers debugged. Model 1 language emulators installed. Major Mesa system executes successfully.

March-June: Microdiagnostics completed. First 16K RAM chips installed in memory Map and storage modules. Second prototype commissioning begins.

July-August: Attempts to speed up system clocks and improve cooling in individually packaged Dorados. First Model 1 delivered to Laboratory.

August-December: Continued wrestling with power, cooling, and packaging problems. Technology transfer for manufacturing begins.

1980 **January-June**: Full-color display system implemented. Manufacturing begins at approximately one Dorado per month. Machine room installation prepared.

1981 Manufacturing continues; multi-layer PC board versions implemented.

1982 Manufacturing continues; multi-layer PC board versions tested.

1983 Manufacturing successful. Approximately 75 Dorados in service. Technology transfer to printed circuit boards completed. All programming staff to have personal Dorados.

Appendix B: Dorado Project Personnel

Early System Architects:

Butler Lampson, Chuck Thacker, Ron Rider

Systems Development Department, early sketches:

Chuck Thacker, Brian Rosen, Don Charnley, Tom Chang, Ken Pier

CSL Dorado Team:

Baseboard, storage modules, and Friend of the Electron: Ed McCreight
Control and Data Processor: Roger Bates, Ed Fiala, Ken Pier
Memory: Butler Lampson, Doug Clark, Ken Pier
IFU: Butler Lampson, Gene McDaniel, Severo Ornstein, Will Crowther
Disk Controller: Roger Bates, Willie-Sue Haugeland
Ethernet Controller: Ed Taft, David Boggs
Display Controllers: Ken Pier
Debugging and Diagnostics: Gene McDaniel, Ed Fiala
MicroAssembler and Placer: Peter Deutsch, Ed Fiala
Language Emulators: Willie-Sue Haugeland, Ed Taft, Peter Deutsch, Nori Suzuki, Bruce Horn, Larry Masinter
Design Automation: Chuck Thacker, Roger Bates, Ed McCreight, Bob Sproull, Martin Kay
Technical Support: Mike Overton, Charlie Sosinski, Herb Yeary
Manufacturing: Larry Clark, Tim Diebert

References

[1] Cognitive and Information Sciences Group. Papers on Interlisp-D. Technical Report CIS-5, Xerox Palo Alto Research Center, 1981.

[2] Clark, D.W., Lampson. B.W., and Pier, K. A. The Memory System of a High-Performance Personal Computer. *IEEE Transactions On Computers* **C-30**, 10, Oct. 1981, 715-733. Also in Technical Report CSL-81-1, Xerox Palo Alto Research Center, Jan. 1981.

[3] Conti, C.J. Concepts for Buffer Storage. *IEEE Computer Group News* **2**, March 1969, 9-13.

[4] Fiala, E.H. The Maxc Systems. *Computer* **11**, 5, May 1978, 57-67.

[5] Frank, E. H. and Sproull, R.F. Testing and Debugging Custom Integrated Circuits. *Computing Surveys ACM* **13**, 4, Dec. 1981, 425-451.

[6] Forgie, J.W. The Lincoln TX-2 Input-Output System. *Proc. Western Joint Computer Conference*, Los Angeles, Feb. 1957, 156-160.

[7] Goldberg, A., and the Smalltalk Language Group. The Smalltalk-80 System. *BYTE Magazine*, June 1981, 36-48.

[8] Goldstein, I.P. PIE: A Network Based Personal Information Environment. *Proc. of the Office Semantics Workshop*, Chatham, Mass., June 1980.

[9] Horning, J. G. Private Communication.

[10] Johnsson, R.K. and Wick, J.D. An Overview of the Mesa Processor Architecture. *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, SigArch/SigPLAN, Palo Alto, March 1982, 20-29.

[11] Kidder, T. *The Soul of a New Machine.* Little Brown, 1981.

[12] Lampson, B.W., Clark, D.W., McDaniel, G.A., Ornstein, S.M., and Pier, K.A. The Dorado: A High-Performance Personal Computer-Three Papers. Technical Report CSL-81-1, Xerox Palo Alto Research Center, Jan. 1981.

[13] Lampson, B.W. and Pier, K.A. A Processor for a High-Performance Personal Computer. *Proc. 7th Int. Symp. Computer Architecture*, SigArch/IEEE, La Baule, May 1980, 146-160. Also in Technical Report CSL-81-1, Xerox Palo Alto Research Center, Jan. 1981.

[14] Lampson, B.W., McDaniel, G.A., and Ornstein, S.M. An Instruction Fetch Unit for a High-Performance Personal Computer. Technical Report CSL-81-1, Xerox Palo Alto Research Center, Jan. 1981. Submitted for publication to IEEE Transactions on Computers.

[15] Lampson, B.W. Fast Procedure Calls. *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, SigArch/SigPLAN, Palo Alto, March 1982, 66-76.

[16] McWilliams, T.M. Verification of Timing Constraints on Large Digital Systems. *J. Digital Syst. (USA)* **5**, 4, Winter 1981, 401-427.

[17] Mitchell, J.G., Maybury, W., and Sweet, R.E. *Mesa Language Manual*, Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[18] Needham, R.M., Herbert, A.J., and Mitchell, J.G. How to Connect Stable Memory to a Computer. SigOps Newsletter, to appear.

[19] Newman, W.M. and Sproull, R.F. *Principles of Interactive Computer Graphics*, 2nd ed. McGraw-Hill, 1979.

[20] Patterson, D. A. and Sequin, C. H. RISC-I: A Reduced Instruction Set Vlsi Computer. *Proc. 8th Int. Symp. Computer Architecture*, SigArch/IEEE, May 1981.

[21] Radin, G. The 801 Minicomputer. *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, SigArch/SigPLAN, Palo Alto, March 1982, 39-47.

[22] Sturgis, H. Private Communication.

[23] Sweet, R. E. and Sandman, J. G. Empirical Analysis of the Mesa Instruction Set. *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, SigArch/SigPLAN, Palo Alto, March 1982, 158-166.

[24] Tanenbaum, A.S. Implications of Structured Programming for Machine Architecture. *Comm. ACM* **21**, 3, March 1978, 237-246.

[25] Teitelman, W. *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Oct. 1978.

[26] Teitelman, W. and Masinter, L. The Interlisp Programming Environment. *Computer* **14**, 4, April 1981, 25-33.

[27] Thacker, C. P. *et. al.* Alto: A Personal Computer. In *Computer Structures: Readings and Examples*, 2nd edition, Sieworek, Bell and Newell, eds., McGraw-Hill, 1981. Also in Technical Report CSL-79-11, Xerox Palo Alto Research Center, August 1979.

[28] Widdoes, L. C. *et. al.* SCALD: Structured Computer Aided Logic Design. *Proc. 15th Annual Design Automation Conference*, SigDA, June 1978, 271-284.

[29] Widdoes, L. C. The S-1 project: Developing High Performance Digital Computers. *Proc. IEEE Compcon*, San Francisco, Feb. 1980, 282-291.