A Processor for a High-Performance Personal Computer

by Butler W. Lampson and Kenneth A. Pier

January 1981

ABSTRACT

This paper describes the design goals, microarchitecture, and implementation of the microprogrammed processor for a compact high performance personal computer. This machine supports a range of high level language environments and high bandwidth I/O devices. It also has a cache, a memory map, main storage, and an instruction fetch unit these are described in other papers. The processor can be shared among 16 microcoded tasks, performing microcode context switches on demand with essentially no overhead. Conditional branches are done without any lookahead or delay. Microinstructions are fai tightly encoded, and use an interesting variant on control field sharing. The processor implements a large number of internal registers, hardware stacks, a cyclic shifter/maske and an arithmetic-logic unit, together with external data paths for instruction fetching memory interface, and I/O, in a compact, pipelined organization.

The machine has a 60 ns microcycle, and can execute a simple macroinstruction in one cycle; the I/O bandwidth is 530 megabits/sec. The entire machine, including disk, displa and network interfaces, is implemented with approximately 3000 MSI components, mostly ECL 10K; the processor is about 35% of this. In addition there are up to 4 storage modules, with about 300 16K or 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). A number of machines are currently running.

A version of this paper appeared in Proc. 7th Symposium on Computer Architecture, SigArch/IEEE, La Baule, May 1980, 146-160.

CR CATEGORIES

6.34, 6.21

KEY WORDS AND PHRASES

architecture, controller, emulation, input/output, microprogram, pipeline, processor.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

1. Introduction

The machine described in this paper, called the Dorado, was designed by and for the Comp
Science Laboratory (CSL) of the Xerox Palo Alto Research Center. CSL has approximately f
people doing research in most areas of computer science, including VLSI design, communica
programming systems, graphics and imaging, office automation, artificial intelligence, c
linguistics, and analysis of algorithms. There is a heavy emphasis on building usable p
systems, and many such systems, both hardware and software, have been developed over the
seven years. Most are part of a personal computing environment which is loosely coupled
such environments, and to service facilities for storage and printing, by a high bandwid
communication network [8].

The Dorado provides the hardware base for the next generation of system research in CSL.
machines have limitations on virtual address size, real memory size, memory bandwidth, a
processor speed that severely hamper our work. The size and speed of the Dorado minimiz
limitations.

The paper has six sections. We begin by sketching the history of the machine's developm
Then we discuss the design goals for the Dorado (¶ 3), and explain how these goals and t
available technology determine the high level processor architecture (¶ 4). Next, we pr
most important details of the processor architecture (¶ 5) and some interesting aspects
implementation (¶ 6). A final section describes the machine's performance (¶ 7).

2. History

The Dorado is a descendant of a small personal computer called the Alto, which was desig
built as an experimental machine in CSL during 1973 [8]. The Alto was a fairly simple ma
it had several features which turned out to be important:

- a microprogrammed processor that is efficiently shared among all the device contro
  well as the virtual machine interpreter;
- a fairly high resolution display system that uses a full bitmap stored in the Alto
  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 p
rather than having independent access to the memory. As a result, controllers can be sm
the I/O interface provided to programs can be powerful. This concept of processor sharin
fundamental to the Dorado as well, and is more fully explained in ¶ 4.

Although there are now many hundreds of Altos at work within Xerox and elsewhere, and th
formed the hardware base for CSL until mid-1980, it was clear by 1976 that a large and ra
increasing amount of effort was going into surmounting the Alto's limitations of space a
rather than trying out research ideas in experimental systems. CSL therefore began to de
machine aimed at relieving these burdens. During 1976 and 1977, design work on the Dora
proceeded in CSL and the System Development Department. Requirements and contributions f
parts of Xerox outside of CSL affected the design considerably, as did the tendency towar
grandiosity well known in second systems. The memory bandwidth and processor throughput
substantially increased.

In 1977, implementation of the laboratory prototype for the Dorado began. 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 al
aspects of the Dorado system until summer of 1978, when the prototype successfully ran a
Alto software. During the summer and fall of 1978 we used the lessons learned in debugg

microcoding the Model 0, together with the significant improvements in memory technology the Model 0 design was frozen, to redesign and reimplement nearly every section of the D We fixed some serious design errors and a number of annoyances to the microcoder, substa expanded all the memories of the machine, and speeded up the basic cycle time. Dorado M came up in the spring of 1979.

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

At the same time, the design was transferred to multiwire circuit boards; the Manhattan routing and lower impedance of this technology slowed the machine down by about 15%. Do are now assembled with very little in-house labor, since boards and backpanels are manuf and loaded by subcontractors. We do 100% continuity testing of the boards both before a they are loaded with components and soldered. Checkout of an assembled machine is still trivial, but is a fairly predictable operation done entirely by technicians.

3. Goals

This section of the paper describes the overall design goals for the Dorado. The high l architecture of the processor, described in the next section, follows from these goals a characteristics of the available technology.

The Dorado is intended to be a powerful but personal computing system. It supports a si within a programming system which may extend from the microinstruction level to a fully integrated programming environment for a high-level language; programming at all levels relatively easy. The machine must be physically small and quiet enough to occupy space users in an office or laboratory setting, and cheap enough to be acquired in considerabl These constraints on size, noise, and cost have a major effect on the design.

In order for the Dorado to quickly become useful in the existing CSL environment, it had compatible with the Alto software base. High-performance Alto emulation is not a requir however; since the existing software is also obsolescent and due to be replaced, the Dor needs to run it somewhat faster than the Alto can.

Instead, the Dorado is optimized for the execution of languages that are compiled into a byte codes; this execution is called emulation. Such byte code compilers exist for Mesa Interlisp [2, 7] and Smalltalk [4]. An instruction fetch unit (IFU) in the Dorado fetche such a stream, decodes them as instructions and operands, and provides the necessary con data information to the processor; it is described in another paper [5]. Further suppor comes from a very fast microcycle, and a microinstruction powerful enough to allow inter of a simple macroinstruction in a single microinstruction. There is also a cache which of two cycles, and can deliver a word every cycle. The goal of fast execution affects t implementation technology, microstore organization, and pipeline organization. It also number of specific features, for example, stacks built with high speed memory, and hardw registers for addressing software contexts.

Another major goal for the Dorado is to support high-bandwidth input/output. In particu monitors, raster scanned printers, and high speed communications are all part of the res activities within CSL; one of these devices typically has a bandwidth of 20 to 400 megabi Fast devices should not slow down the emulator too much, even though the two functions c for many of the same resources. Relatively slow devices must also be supported, without the high bandwidth I/O system. These considerations clearly suggest that I/O activity an should proceed in parallel as much as possible. Also, it must be possible to integrate

undefined device controllers into the Dorado system in a relatively straightforward way. memory system supports these requirements by allowing cache accesses and main storage re to proceed in parallel, and by fully segmented pipelining which allows a cache reference every cycle, and a storage reference to start in every storage cycle; this system is des another paper [1].

Any system for experimental research should provide adequate resources at many levels. processor, this means plenty of high speed internal storage as well as ample speed. Har support for handling arbitrary bit strings, both large and small, is also necessary.

4. High level architecture

We now proceed to consider the major design decisions which shaped the Dorado processor. the most part these were guided by the goals set out above, the available implementation technology, and our past experience. In this section we stay at a high level, reserving the architecture for the next.

The Dorado fits into a very compact package, illustrated in Figure 1a; Figure 1b is a hi block diagram. Circuits are mounted on large, high density logic boards (288 16-pin DIP packages plus 144 8-pin SIP resistor packages per board). The boards slide horizontally insertion-force connectors mounted in dual backpanels ("sidepanels"); they are .625 inch This density makes it possible to reconcile the goals of size and capability. Certain s made, however. For example, it is not possible to access every signal with a scope prob debugging and maintenance. We make up for this by providing sophisticated debugging fac diagnostics, and the ability to incrementally assemble and test a Dorado from the bottom

The entire machine, including disk, display and network interfaces, is implemented with approximately 3000 MSI components, mostly ECL 10K; the processor is about 35% of this. I addition there are up to 4 storage modules, each with about 300 16K or 64K RAMS and 200 M components, for a maximum of 8 megabytes. The total volume, including power and cooling about .14 m^3 (4.5 ft^3); this is without any enclosing cabinet, however, and the open mach noisy. Including an 80 megabyte removable disk, it requires about 2.5 Kw of AC power.

Most data paths are sixteen bits wide. The relatively small busses, registers, data pat memories which result help to keep the machine compact. Packaging, however, is not the consideration. CSL has a large class of applications where doubling the data path width performance only a little, because some of the bits contain type codes, flags or whateve be examined before an entire datum can be processed. Speed dictates a heavily pipelined in any case, and this parallelism in the time domain tends to compensate for the lack of in the space domain. Keeping the machine physically small also improves the speed, sinc distance accounts for a considerable fraction of the basic cycle time. Finally, perform limited by the cache hit rate, which cannot be improved, and may be reduced, by wider da (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 t to access the memory, the Dorado shares the processor among all the I/O devices and the e This fundamental concept of the architecture, which motivates much of the processor desi first tried in the Alto. It works for two main reasons.

- First, unless a system has both multiple memory busses (i.e., multi-ported memorie multiple memory modules which can cycle independently, the main factor governing processor throughput is memory contention. Put simply, when I/O interfaces make me references, the emulator ends up waiting for the memory. In this situation the pr might as well be working for the I/O device.

<==<ProcFig1.press<

- Second, when the processor is available to each device, complex device interfaces implemented with relatively little dedicated hardware, since most of the control d have to be duplicated in each interface. For low bandwidth devices, the force of argument is reduced by the availability of LSI controller chips, but for data rates megabit/second no such chips exist as yet.

Of course, to make this sharing feasible, switching the processor must be nearly free of and devices must be able to make quick use of the processor resources available to them.

Many design decisions are based on the need for speed. Raw circuit speed is a beginning the Dorado is implemented using the fastest commercially available technology which has reasonable level of integration and is not too hard to package. In 1976, the obvious ch ECL 10K family of circuits; probably it still is. Secondly, the processor is organized a pipelines. One allows a microinstruction to be started in each cycle, though it takes t complete execution. Another allows a processor context switch in each cycle, though it cycles to occur. Thirdly, independent busses communicate with the memory, IFU, and I/O s so that the processor can both control and service them with minimal overhead.

Finally, the design makes the processor both accessible and flexible for users at the mi so that when new needs arise for fast primitives, they can easily be met by new microcod particular, the hardware eliminates constraints on microcode operations and sequencing o in less powerful designs, e.g., delay in the delivery of intermediate results to registe calculating and using branch conditions, or pipeline delays that require padding of micr sequences without useful work. We also included an ample supply of resources: 256 gener registers, four hardware stacks, a fast barrel shifter, and fully writeable microstore, Dorado reasonably easy to microcode.

5. Low level architecture

This section describes in some detail the key ideas of the architecture. Implementation and details are for the most part deferred to the next section; readers may want to jump see the application of these ideas in the processor. Along with each key idea is a refe places in the processor where it is used.

5.1 Tasks

There are 16 priority levels associated with microcode execution. These levels are call or simply tasks. Each task is normally associated with some hardware and microcode whic together implement a device controller. The tasks have a fixed priority, from task 0 (l 15 (highest). Device hardware can request that the processor be switched to the associa such a wakeup request will be honored when no requests of higher priority are outstandin of wakeup requests is arbitrated within the processor, and a task switch from one task t occurs on demand, typically every ten or twenty microcycles when a high-speed device is

When a device acquires the processor (that is, the processor is running at the requested level and executing the microcode for that task), the device will presumably receive ser microcode. Eventually the microcode will block, thus relinquishing the processor to lowe tasks until it next requires service. While a given task is running, it has the exclusi the processor. This arrangment is similar in many ways to a conventional priority inter An important difference is that the tasks are like coroutines or processes, rather than when a task is awakened, it continues execution at the point where it blocked, rather th at a fixed point. This ability to capture part of the state in the program counter is v

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

5.2 Task scheduling

Whenever resources (in this case, the processor) are multiplexed, context switching must happen when the state being temporarily abandoned can be restored. In most multiplexed microcoded systems, this requires the microcode itself to explicitly poll for requests, restore state, and initiate context switches. A certain amount of overhead results. Fu the presence of a cache introduces large and unpredictable delays in the execution of mi (because of misses). A polling system would leave the processor idle during these delay though the work of another task can usually proceed in parallel. To avoid these costs, does task switching on demand of a higher priority device, much like a conventional inte system. That is, if a lower priority task is executing and a higher priority device req the lower priority task will be preempted; the higher priority device will be serviced w consent or even the knowledge of the currently active task. The polling overhead is abs the hardware, which also becomes responsible for resuming a preempted task once the proc relinquished by the higher priority device.

A controller will continue to request a wakeup until notified by the processor that it i receive service; it then removes the request, unless it needs more than one unit of serv the microcode is done, it executes an operation called Block which releases the processor effect is that requesting service is done explicitly by device controllers, but scheduli task is invisible to the microcode (and nearly invisible to the device hardware).

5.3 Task specific state

In order to allow the immediate task switching described above, the processor must be ab and restore state within one microcycle. This is accomplished by keeping the vital stat throughout the processor not in a single rank of registers but in task specific register actually implemented with high speed memory that is addressed by a task number. Example task specific registers are the microcode program counter, the branch condition register microcode subroutine link register, the memory data register, and a temporary storage re each task. The number of the task which will execute in the next microcycle is broadcas throughout the processor and used to address the task specific registers. Thus, data ca from the high speed task specific memories and be available for use in the next cycle.

Not all registers are task specific. For example, COUNT and Q are normally used only by t However, they can be used by other tasks if their contents are explicitly saved and rest

5.4 Pipelining

There are two distinct pipelines in the Dorado processor. The main one fetches and exec microinstructions. The other handles task switching, arbitrates wakeup requests and bro next task number to the rest of the Dorado. Each structure is synchronous, and there is between stages.

The instruction pipeline, illustrated in Figure 2, requires three cycles (divided into s completely execute a microinstruction. The first cycle is used to fetch it from microst t_0). The result of the fetch is loaded into the microinstruction register MIR at t_0. The is split; in the first half, operand fetches (as dictated by the contents of MIR) are per results latched at t_1 in two registers (A and B) which form inputs to the next stage. In half cycle, the ALU operation is begun. It is completed in the first half cycle of cycle result is latched in register RESULT (at t_3). The second half of cycle three (t_3 to t_4) is results from RESULT into operand registers.

<==<ProcFig2.press<

<==<ProcFig3.press<

The figure also shows how the pipeline overlapping is achieved. A new microinstruction every cycle time. The operand registers are used in the first half cycle of every cycle operands for the current instruction (during t_0 t_1). The second half of every cycle is u results for the previous instruction (during t_3 t_4).

Figure 3 shows the task arbitration pipeline. This pipeline is two stages long, and als cycle per stage. At the beginning of the pipeline (t_0), wakeup requests from device cont latched into the WAKEUP register. During the first half cycle (t_0 t_1), arbitration is per the highest priority task determined. During the second half cycle (t_1 t_2), the micropro address for the highest priority task is fetched from the task specific program counter number, its TPC, and the command to switch tasks (if the highest priority task is higher currently executing task) are loaded into registers at t_2. In the second pipe cycle, the fetch the next microinstruction from the microstore, the entire processor uses the selec

number to fetch the appropriate task specific information, and device controllers are to
will have the processor next. Finally, at t_4 the task switch is complete, and the new ta
control of the processor; this time corresponds to t_0 of the first microinstruction execu
new task.

5.5 Microinstruction format

One of the key decisions made in the design of any microprogrammed processor is the form
semantics of the microinstruction. The Dorado's demand for compactness and power are at
this case. Compactness dictates that an essentially vertical structure be used, with en
specifying many functions in a few bits. The details of the microinstruction format app
The major features of interest here are the choice of successor instruction encoding, an
specification of a large number of functions which may be executed by the processor.

In a classical microprogrammed processor, each instruction carries with it the address o
successor, NEXTPC; this address is latched with the rest of the instruction, and then use
address the microstore for fetching the next instruction. NEXTPC may be modified by stat
the processor during execution, but the basic idea is that enough bits must be present i
microword to address the whole microstore. This results in a uniform structure for addr
allows the next instruction fetch to proceed without any delay for decoding; it has the
of increasing the size and cost (and reducing the speed) of the microstore. The lack of
decoding time also makes it impossible to specify a subroutine return or other major cha
sequencing, and have it take effect immediately (branches can still use the scheme descr

The alternative, used in the Dorado, is to divide the microstore into pages, use a few b
a next address within the current page, and have a type field which can specify branches
returns, transfers to another page, or whatever. At the start of a microcycle, the proc
the type field and accesses other information (such as the current page number or the re
to compute NEXTPC. In addition, some types cause side effects such as the loading the re
The net result is substantially fewer bits to control microsequencing than a horizontal
require (in the Dorado, 8 bits instead of about 16). The disadvantages are, of course,
time for decoding this field, and the additional complexity of an assembler which can fi
instructions onto pages appropriately.

Conditional branching is always a problem with pipelined instruction execution. Most de
one of the following two schemes, and tolerate its drawbacks. The first requires that a
specified one (or more) instructions before it is taken. Although this simplifies and s
hardware, it imposes severe constraints on the microcode organization, and often forces
instructions to be executed. The second scheme detects the branch and inserts asynchron
or an extra cycle to allow time for the new instruction to be fetched. This obviously s
the machine.

Conditional branching in the Dorado is handled by allowing one of eight branch condition
modify the low order bit of NEXTPC. This modification (Boolean or into the low order bit
place about half way into the instruction fetch cycle. The microstore is organized so t
does not change the chip address, but instead selects a different chip from a set of chi
outputs are tied directly together. Since access time from the chip select is considera
from the address, the late arriving branch condition does not increase the total cycle t
to work, the assembler must place each false branch target at an even address, and the
corresponding true branch target at the next higher odd address. An annoying consequenc
several conditional branches cannot have same target; when this case arises the target m
duplicated. Everything has its price.

Another tradeoff occurs in the mechanism for controlling the functions of the processor
microcycle. The Dorado encodes most of its operations (other than register selection, AL
operations, storing results, and memory references) in an eight bit function field calle

quickly decoded at the beginning of every microinstruction execution cycle (during t_0-t_1) used to invoke all of the less frequently used operations that the processor can do: con I/O busses, reading and setting state in the memory and IFU, extracting an arbitrary fiel word, reading and loading most registers, non-standard carry and shift operations, and l values into small registers. FF can also serve as an eight bit constant or as part of a address. This encoding saves many bits in the microinstruction, at the expense of allow FF-specified operation to be done in each cycle, even though the data paths exist for do such operations in parallel.

5.6 Data bypassing

Recall that a microinstruction is initiated at the beginning of every cycle, but takes o instruction fetch and two cycles for execution. If an instruction uses a result generat immediate predecessor, it needs to get that result from an operand register before the p has actually delivered the result to that register. Rather than forbidding such use of delaying execution until the register has been loaded, we solved this problem with a tec called bypassing. The hardware detects that an operand specified in the current instruc actually the result of the previous instruction. Rather than obtaining the operand from source in a RAM, the processor takes it directly from the input to the RAM, which is the previous instruction. Figure 4 illustrates the scheme. This costs extra hardware for mu bypass detection logic, but the result is much smaller and faster microcode in many comm In the Model 0 Dorado, we omitted bypassing logic in a few places, and required the micr avoid these cases. The result was a number of subtle bugs and a significant loss of per

<==<ProcFig4.press<

5.7 Memory delays

Pipelining and bypassing are effective ways to reduce delay and increase throughput with processor. Interactions with the memory, however, pose different problems. Once a memo reference has been made, there must be some way to tell when the memory system has deliv the requested data. Two simple techniques are to wait a fixed (unfortunately, maximum) time before using the data, or to explicitly poll the memory system. Neither is satisfa high performance machine. First, the difference between the best case (cache hit) and t (cache miss plus memory system resource contention) is more than an order of magnitude. useful work can often be performed by a given task before it uses the requested memory d Third, even if a given task must wait for memory data before it can proceed, higher prio may very well be able to do useful work in the meantime.

The Dorado manages this problem by making the memory keep track of when data is ready, a allowing the processor to keep executing instructions. Only instructions which use memo

start memory references can be affected by the state of the memory. When such an instru
executed, the memory checks to see whether it can be allowed to proceed. If so, no acti
But if the memory is busy, or the data being used is not ready, the memory responds by a
the signal Hold. The effect of Hold is to stop any state changes specified by the curren
However, all the clocks in the system keep running. This is important, because task swi
not be inhibited during memory delays. In effect, Hold converts the currently executing
into a "no operation, jump to self" instruction. If no task switch occurs, the instruct
again, and a new calculation is made to see whether it can proceed. Meanwhile, the memo
pipeline is running, and sooner or later, the need for Hold will be gone as the pipeline

Note that if a task switch occurs while an instruction is held, the state is such that t
instruction may simply be restarted when the lower priority task is resumed by the proce
Cycles which would otherwise be dead time are consumed instead by higher priority tasks
useful work.

5.8 Separate external interfaces

If most macroinstructions (byte codes) are to execute in a small number of cycles, hardw
be provided to make communication among processor, IFU, and memory very quick in the comm
cases. The Dorado provides a number of data paths and control structures for this purpo
detailed in the block diagrams, Figures 5 and 6. All the busses are a full word wide an
accessed in one cycle or less. The B input to the ALU is extended to the remainder of th
(except I/O devices, which have their own busses) for the transfer of status and control
processor and the other subsystems. The memory address bus is a copy of the A side ALU i
Memory data comes directly into the processor and is routed to a variety of destinations
simultaneously, to make such operations as field manipulations and indirect addressing f
IFU can directly supply operand data to the processor, and any microinstruction can speci
the last of a macroinstruction, in which case the successor address is supplied by the IF
requires a microstore address bus and operand data bus directly from the IFU to the proce

It is also desirable to make I/O transfers through the processor fast. To this end there
address bus and an I/O data bus for direct access to I/O controllers. The data bus can t
word per cycle, or 265 megabits/second, and both the memory reference and the I/O transfe
be specified in a single instruction, so that it is possible to move a sequence of words
cache and a device at this rate. However, this subsystem is called the slow I/O system.
also a more direct memory access I/O subsystem, the fast I/O system; it allows data to mo
between storage and I/O devices, in blocks of 16 words, without polluting the cache. Fig
shows a display controller that uses both slow and fast I/O systems.

5.9 Constants

Notice that there is no source for 16 bit constants within the processor. Such constant
necessary, particularly in device controller microcode where they often are used as comm
addresses or literal data. It would be possible to include a constant box, addressed pe
FF function, as a source for constants. However, such a box would have a limited size an
experience tells us, would not hold enough constants to satisfy a growing world.

Fortunately, a large fraction of the constants used in microcoding are either small posi
negative (2's complement) integers, or sparsely populated bit vectors, with the property
the two eight bit fields in the constant is all zeroes or all ones. Thus a useful subse
can be specified using the eight bits of FF for one byte of the constant and two other bi
the other byte value and position. Using this technique, most 16 bit constants can be s
one microinstruction, and any constant can be assembled in two microinstructions. (The
two bits come from the BSelect field in the microword).

6. Implementation

In this section we describe, at the block diagram level, the actual implementation of th
processor. There is only space to cover the most interesting points and to illustrate t
from ¶ 5.

6.1 Clocks

The Dorado has a fully synchronous clock system, with a clock tick every 30 nanoseconds.
consists of two successive clock ticks; it begins on an even tick, which is followed by
and completes coincident with the beginning of a new cycle on the next even tick. Even
be labeled with names like t_{-2}, t_0, t_2, t_4 to denote events within a microinstruction exec
pipeline, relative to some convenient origin. Odd ticks are similarly labeled t_{-1}, t_1, t_3

6.2 The control section

The processor can be divided into two distinct sections, called control and data. The c
fetches and broadcasts the microinstructions to the data section (and the remainder of t
handles task switching, maintains a subroutine link, and regulates the clock system. It
interface to a console and monitoring microcomputer which is used for initialization and
of the Dorado. Figure 5 is a block diagram of the control section.

6.2.1 Task pipeline

The task pipeline consists of an assortment of registers and a priority encoder. All th
loaded on even clocks. Wakeup requests are latched at t_0 in WAKEUP, one bit per task; REA
corresponding bits for preempted and explicitly readied tasks. The requests in WAKEUP an
compete. A task can be explicitly made ready by a microcode function. The priority enc
produces the number of the highest priority task, which is loaded into BESTNEXTTASK and als
to read the TPC of this task into BESTNEXTPC; these registers are the interface between the
stages in this pipeline. The NEXT bus normally gets the larger of BESTNEXTTASK and THISTASK.
THISTASK is loaded from NEXT, and LASTTASK is loaded from THISTASK, as the pipeline progresses

This method of priority scheduling means that once a task is initiated, it must explicit
the processor before a lower priority task can run. A bit in the microword, Block, is us
indicate that NEXT should get BESTNEXTTASK unconditionally (unless the instruction is held)

Note that it takes a minimum of two cycles from the time a wakeup changes to the time th
change can affect the running task (one for the priority encoding, one to fetch the micr
This implies that a task must execute at least two microinstructions after its wakeup is
before it blocks; otherwise it will continue to run, since the effects of its wakeup wil
cleared from the pipe. The device cannot remove the wakeup until it knows that the task
(by seeing its number on NEXT). Hence the earliest the wakeup can be removed is t_0 of th
instruction (NEXT has the task number in the previous cycle, and the wakeup is latched at
the grain of processor allocation is two cycles for a task waking up after a Block.

Some trouble was taken to keep the grain small, for the following reason. Since the mem
heavily pipelined and contains a cache which does not interact with high bandwidth I/O, t
microcode often needs to execute only two instructions, in which a memory reference is s
a count is decremented. The processor can then be returned to another task. The maximu
which storage references can be made is one every eight cycles (this is the cycle time o
storage RAMs). A two cycle grain thus allows the full memory bandwidth of 530 megabits/s
to be delivered to I/O devices using only 25% of the processor.

A simpler design would require the microcode to explicitly notify its device when the wa
should be removed; it would then be unnecessary to broadcast NEXT to the devices. Since
notification could not be done earlier than the first instruction, however, the grain wo
cycles rather than two, and 37.5% of the processor would be needed to provide the full m
bandwidth. Other simplifications in the implementation would result from making the pip
longer; in particular, squeezing the priority encoding and reading of TPC into one cycle
difficult. Again, however, this would increase the grain.

6.2.2 Fetching microinstructions

Refer to the right hand side of Figure 5. At t_0 of every instruction, the microinstructi
MIR is loaded from the outputs of IM, the microinstruction memory, and the THISPC register
loaded with IMADDRESS. The NEXTPC is quickly calculated based on the NextControl field in MI
which encodes both the instruction type and some bits of NEXTPC; see Figure 7 for details
calculation produces THISTASKNEXTPC, so called because if a task switch occurs it is not us
next IMADDRESS. Instead, the BESTNEXTPC computed in the task pipeline is used as IMADDRESS.

<==<ProcFig7.press<

TPC is written with the previous value of THISTASKNEXTPC every cycle (at t_3), and read for t
in BESTNEXTTASK every cycle as well. Thus, TPC is constantly recording the program counter
for the current task, and also constantly preparing the value for the next task in case
switch.

6.2.3 Miscellaneous features

There is a task specific subroutine linkage register, LINK, shown in Figure 5, which is l
the value in THISPC+1 on every microcode call or return. Thus each task can have its own
microcoded coroutines. LINK can also be loaded from a data bus, so that control can be s
arbitrary computed address; this allows a microprogram to implement a stack of subroutin
for example. In addition to conditional branches, which select one of two NEXTPC values,
also eight-way and 256-way dispatches, which use a value on the B bus to select one of e
of 256 NEXTPC values.

Since the Dorado's microstore is writeable, there are data paths for reading and writing
paths allow reading and writing TPC. These paths (through the register TPIMOUT) are folde
already existing data paths in the control section and are somewhat tortuous, but they a
infrequently and hence have been optimized for space. In addition, another computer (ei
separate microcomputer or an Alto) serves as the console processor for the Dorado; it is
via the CPREG and a very small number of control signals.

6.3 The data section

Figure 6 is a block diagram of the data section, which is organized around an arithmetic
(ALU). It implements most of the registers accessible to the programmer and the microcod
functions for selecting operands, doing operations in the ALU and shifter, and storing re
calculates branch conditions, decodes MIR fields and broadcasts decoded signals to the re
Dorado, supplies and accepts memory addresses and data, and supplies I/O data and address

6.3.1 The microinstruction register

MIR (which actually belongs to the control section) is 34 bits wide and is partitioned in
following fields:

RAddress	4	Addresses the register bank RM.
ALUOp	4	Selects the ALU operation or controls the shifter.
BSelect	3	Selects the source for the B bus, including constants.
LoadControl	3	Controls loading of results into RM and T.
ASelect	3	Selects the source for the A bus, and starts memory references.
Block	1	Blocks an I/O task, selects a stack operation for task 0.
FF	8	Catchall for specifying functions.
NextControl	8	Specifies how to compute NEXTPC.

6.3.2 Busses

The major busses are A, B (ALU sources), RESULT, EXTERNALB, MEMADDRESS, IOADDRESS, IODATA,
IFUDATA, and MEMDATA .

The ALU accepts two inputs (A and B) and produces one output (RESULT). The input busses ha
variety of sources, as shown in the block diagram. RESULT usually gets the ALU output, bu
also sourced from many other places, including a one bit shift in either direction of th
A copy of A is used for MEMADDRESS; two copies of B are used for EXTERNALB and IODATA.
MEMADDRESS provides a sixteen bit displacement, which is added to a 28 bit base register i
memory system to form a virtual addresses. EXTERNALB is a copy of B which goes to the con
memory, and IFU sections, and IODATA is another copy which goes to the I/O system; the sou

B can thus be sent to the entire processor. Both are bidirectional and can serve as a s
well. IOADDRESS is driven from a task specific register; it specifies the particular devi
which should source or receive IODATA.

IFUDATA and MEMDATA allow the processor to receive data from the IFU and memory in parallel
with other data transfers. MEMDATA has the value of the memory word most recently fetched
the current task; if the fetch is not complete, the processor is held when it tries to u
IFUDATA has an operand of the current macroinstruction; as each operand is used, the IFU p
the next one on IFUDATA.

6.3.3 Registers

Here is a list and brief description of registers seen by the microprogrammer. All are
bits) wide.

RM: a bank of 256 general purpose registers; a register can be read onto A, B, or
shifter, and loaded from RESULT under the control of LoadControl. Normally, th
same register is both read and loaded in a given microinstruction, but loadi
different register can be specified by FF.

STACK: a memory addressed by the STACKPTR register. A word can be read or written,
and STACKPTR adjusted up or down, in one microinstruction. If STACK is used in
microinstruction, it replaces any use of RM, and the RAddress field in the mi
tells how much to increment or decrement STACKPTR. The 256 word memory is
divided into four 64 word stacks, with independent underflow and overflow
checking.

T: a task specific register used for working storage; like RM, it can be read or
or the shifter, and loaded from RESULT under the control of LoadControl..

COUNT: a counter; it can be decremented and tested for zero in one microinstruction
only the NextControl or FF field. It is loaded from B or with small constants
FF.

SHIFTCTL: a register which controls the direction and amount of shifting and the width
and right masks; it is loaded from B or with values useful for field extracti
FF.

Q: a hardware aid for multiply and divide instructions; it can be read onto A o
loaded from B, and is automatically shifted in useful ways during multiply an
divide step microinstructions.

The next group of registers vary in width. They are used as control or address register
dynamically but infrequently by microcode.

RBASE: RM addressing requires eight bits. Four come from the RAddress field in the
microword, and the other four are supplied from RBASE. It is loaded from B o
and can be read onto RESULT.

STACKPTR: an eight bit register used as a stack pointer. Two bits of STACKPTR select a
and the least significant six bits a word in the stack. The latter bits are
incremented or decremented under control of the RAddress field whenever a sta
operation is specified.

MEMBASE: a five bit register which selects one of 32 base registers in the memory to
for virtual address calculation. It is loaded from FF field or from B, and c
loaded from the IFU at the start of a macroinstruction.

ALUFM: a 16 word memory which maps the four-bit ALUOp field into the six bits requir
to control the ALU.

IOADDRESS: a task specific register which drives the IOADDRESS bus, and is loaded by I/O
microcode to specify a device address for subsequent Input and Output operati
It may be loaded from B or FF.

6.3.4 The shifter

The Dorado has a 32 bit barrel shifter for handling bit-aligned data. It takes 32 bits RM and T, performs a left cycle of any number of bit positions, and places the result on ALU output may be masked during a shift instruction, either with zeroes or with data from MEMDATA.

The shifter is controlled by the SHIFTCTL register. To perform a shift operation, SHIFTCTL (in one of a variety of ways) with control information, and then one of a group of "shif microoperations is executed.

6.4 Physical organization

Once the goal of a physically small but powerful machine was established, engineering de material lead times forced us to develop the Dorado package before the implementation wa than partially completed, and the implementation then had to fit the package. The data partitioned onto two boards, eight bits on each; the boards are about 70% identical. Th section divides naturally into one board consisting of all the IM chips (high speed 1K x RAMS) and their associated address drivers, and a second board with the task switch pipel NEXTPC logic, and LINK register.

The sidepanel pins are distributed in clusters around the board edges to form the major The remaining edge pins are used for point to point connections between two specific boa I/O busses go uniformly to all the I/O slots, but all the other boards occupy fixed slots wired for their needs. Half the pins available on the sideplanes are grounded, but wire not controlled except in the clock distribution system, and no twisted pair is used in t except for distribution of one copy of the master clock to each board.

We were very concerned throughout the design of the Dorado to balance the pipelines so t one pipe stage is significantly longer than the others. Furthermore, we worked hard to longest stage (which limits the speed of this fully synchronous machine) as short as pos longest stage in the processor, as one might have predicted, is the IMADDRESS calculation microinstruction fetch in the control slice. There is about a 50 nanosecond limit for r operation in a stitchwelded machine, and 60 ns in a multiwired machine. There are pipe about the same length in the memory and IFU.

We also worked hard to get the most out of the available real estate, by hand tailoring integrated circuit layout and component usage, and by incrementally adding function unti the entire board was in use. We also found that performance could be significantly impr careful layout of critical paths for minimum loading and wiring delay. Although this wa labor intensive operation, we believe it pays off.

7. Performance

Four emulators have been implemented for the Dorado, interpreting the BCPL, Lisp, Mesa an Smalltalk instruction sets. A typical microinstruction sequence for a load or store ins only one or two microinstructions in Mesa (or BCPL), and five in Lisp. The Mesa opcode c a 16 bit word to or from memory in one microinstruction; Lisp deals with 32 bit items an its stack in memory, so two loads and two stores are done in a basic data transfer opera complex operations (such as read/write field or array element) take five to ten microins Mesa and ten to twenty in Lisp. Note that Lisp does runtime checking of parameters, whi Mesa most checking is done at compile time. Function calls take about 50 microinstructi Mesa and 200 for Lisp.

The Dorado supports raster scan displays which are refreshed from a full bitmap in main this bitmap has one bit for each picture element (dot) on the screen, for a total of .5

(more for gray-scale or color pictures). A special operation called BitBlt (bit boundary transfer) makes it easier to create and update bitmaps; for more information about BitBlt where it is called RasterOp. BitBlt makes extensive use of the shifting/masking capabilit processor, and attempts to prefetch data so that it will always be in the cache when nee Dorado's BitBlt can move display objects around in memory at 34 megabits/sec for simple o erasing or scrolling a screen. More complex operations, where the result is a function object, the destination object and a filter, run at 24 megabits/sec.

I/O devices with transfer rates up to 10 megabits/sec are handled by the processor via th and IOADDRESS busses. The microcode for the disk takes three cycles to transfer two words way; thus the 10 megabit/sec disk consumes 5% of the processor. Higher bandwidth device the fast I/O system, which does not interact with the cache. The fast I/O microcode for takes only two instructions to transfer a 16 word block of data from memory to the devic can consume the available memory bandwidth for I/O (530 megabits/sec) using only one qua the available microcycles (that is, two I/O instructions every eight cycles).

Recall that the NEXTPC scheme (¶ 5.5 and ¶ 6.2.2) imposes a rather complicated structure microstore, because of the pages, the odd/even branch addresses, and the special subrout locations We were concerned about the amount of microstore which might be wasted by aut placement of instructions under all these constraints. In fact, however, the automatic 99.9% of the available memory when called upon to place an essentially full microstore.

Acknowledgements

The early design of the Dorado processor was done by Chuck Thacker and Don Charnley. Th data section was redesigned and debugged by Roger Bates and Ed Fiala. Peter Deutsch wro microcode assembler and instruction placer, and Ed Fiala wrote the Dorado assembler macr microprogram debugger, and the hardware manual. Willie-Sue Haugeland, Nori Suzuki, Bruc Horn, Peter Deutsch, Ed Taft and Gene McDaniel are responsible for production and diagno microcode.

References

1. Clark, D.W. et. al. The memory system of a high-performance personal computer. Technical Report CSL-81-1, Xer Alto Research Center, January 1981. Revised version to appear in IEEE Transactions on Computers.
2. Deutsch, L.P. Experience with a microprogrammed Interlisp system. Proc. 11th Ann. Microprogramming Works Grove, Nov. 1979.
3. Geschke, C.M. et. al. Early experience with Mesa. Comm ACM 20, 8, Aug 1977, 540-552
4. Ingalls, D.H. The Smalltalk-76 programming system: Design and implementation. 5th ACM Symp. Principles of Programming Languages, Tucson, Jan 1978, 9-16.
5. Lampson, B.W. et. al. An instruction fetch unit for a high-performance personal computer. Technical Repo Xerox Palo Alto Research Center, Jan. 1981. Submitted for publication.
6. Mitchell, J.G. et. al. Mesa Language Manual, Technical Report CSL-79-3, Xerox Palo Alto Research Center, A
7. Teitelman, W. Interlisp Reference Manual, Xerox Palo Alto Research Center, Oct. 1978.
8. Thacker, C.P. et. al. Alto: A personal computer. In Computer Structures: Readings and Examples, 2nd editio Bell and Newell, eds., McGraw-Hill, 1981. Also in Technical Report CSL-79-11, Xerox Palo Alto Research Cen 1979.
9. Newman, W.M. and Sproull, R.F. Principles of Interactive Computer Graphics, 2nd ed. McGraw-Hill, 1979.