The Dorado:
A High-Performance Personal Computer

Three Papers

CSL-81-1 January 1981

ABSTRACT

This report reproduces three papers on the Dorado personal computer. Each has been, or will be, published in a journal or proceedings.

> **A Processor for a High-Performance Personal Computer**, by Butler W. Lampson and Kenneth A. Pier. Appeared in *Proc. 7th Symposium on Computer Architecture*, SigArch/IEEE, La Baule, May 1980, 146-160.
>
> **An Instruction Fetch Unit for a High-Performance Personal Computer**, by Butler W. Lampson, Gene A. McDaniel, and Severo M. Ornstein. Submitted for publication.
>
> **The Memory System of a High-Performance Personal Computer**, by Douglas W. Clark, Butler W. Lampson, and Kenneth A. Pier. A revised version will appear in *IEEE Transactions on Computers.*

The first paper describes the Dorado's micro-programmed processor, and also gives an overview of its history and physical construction. The second discusses the instruction fetch unit, which prepares program instructions for execution, and the third deals with the cache, map and main storage of the Dorado's memory system.

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

This page should not appear in the published report.

Please note that the pages are numbered continuously for all three papers, including the title page for each paper, but excluding the title page for the entire report. Pages have been laid out with the assumption that an even page and the next odd page will form a double-page spread. All the title pages are odd numbered, and the next (even) page is the first text page, which should appear on the reverse of the title page. A blank page should appear on the reverse of the title page for the entire report. Page 20, at the end of the first paper, is blank so that the title page of the second paper will fall on an odd page.

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 fairly 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/masker, 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, display 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, each 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 Computer Science Laboratory (CSL) of the Xerox Palo Alto Research Center. CSL has approximately forty people doing research in most areas of computer science, including VLSI design, communications, programming systems, graphics and imaging, office automation, artificial intelligence, computational linguistics, and analysis of algorithms. There is a heavy emphasis on building usable prototype systems, and many such systems, both hardware and software, have been developed over the last seven years. Most 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 [8].

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

The paper has six sections. We begin by sketching the history of the machine's development (¶ 2). Then we discuss the design goals for the Dorado (¶ 3), and explain how these goals and the available technology determine the high level processor architecture (¶ 4). Next, we present the most important details of the processor architecture (¶ 5) and some interesting aspects of the 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 designed and built as an experimental machine in CSL during 1973 [8]. The Alto was a fairly simple machine, but it had several features which 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 fairly 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, and is more fully explained in ¶ 4.

Although there are now many hundreds of Altos at work within Xerox and elsewhere, and they formed the hardware base for CSL until mid-1980, it was clear by 1976 that a large and rapidly increasing amount of effort was going into surmounting the Alto's limitations of space and speed, rather than trying out research ideas in experimental systems. CSL therefore began to design a new machine aimed at relieving these burdens. During 1976 and 1977, design work on the Dorado proceeded in CSL and the System Development Department. Requirements and contributions from parts of Xerox outside of CSL affected the design considerably, as did the tendency toward grandiosity well known in second systems. The memory bandwidth and processor throughput were 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 all aspects of the Dorado system until summer of 1978, when the prototype successfully ran all the Alto software. During the summer and fall of 1978 we used the lessons learned in debugging and

microcoding the Model 0, together with the significant improvements in memory technology since the Model 0 design was frozen, to redesign and reimplement nearly every section of the Dorado. We fixed some serious design errors and a number of annoyances to the microcoder, substantially expanded all the memories of the machine, and speeded up the basic cycle time. 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 worked very well for prototypes, but is too expensive for even modest quantities. Its major advantages are packaging density and signal propagation characteristics very similar to those of the production technology, 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%. Dorados are now assembled with very little in-house labor, since boards and backpanels are manufactured and loaded by subcontractors. We do 100% continuity testing of the boards both before and after they are loaded with components and soldered. Checkout of an assembled machine is still non-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 level architecture of the processor, described in the next section, follows from these goals and the characteristics of the available technology.

The Dorado is intended to be a powerful but personal computing system. It supports a single user 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 must be relatively easy. The machine must be physically small and quiet enough to occupy space near its users in an office or laboratory setting, and cheap enough to be acquired in considerable numbers. 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 to be compatible with the Alto software base. High-performance Alto emulation is not a requirement, however; since the existing software is also obsolescent and due to be replaced, the Dorado only 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 stream of byte codes; this execution is called *emulation*. Such byte code compilers exist for Mesa [3, 6], Interlisp [2, 7] and Smalltalk [4]. An instruction fetch unit (IFU) in the Dorado fetches bytes from such a stream, decodes them as instructions and operands, and provides the necessary control and data information to the processor; it is described in another paper [5]. Further support for this goal comes from a very fast microcycle, and a microinstruction powerful enough to allow interpretation of a simple macroinstruction in a single microinstruction. There is also a cache which has a latency of two cycles, and can deliver a word every cycle. The goal of fast execution affects the choices of implementation technology, microstore organization, and pipeline organization. It also mandates a number of specific features, for example, stacks built with high speed memory, and hardware base registers for addressing software contexts.

Another major goal for the Dorado is to support high-bandwidth input/output. In particular, color monitors, raster scanned printers, and high speed communications are all part of the research activities within CSL; one of these devices typically has a bandwidth of 20 to 400 megabits/second. Fast devices should not slow down the emulator 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. Also, it must be possible to integrate as yet

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

Any system for experimental research should provide adequate resources at many levels. For the processor, this means plenty of high speed internal storage as well as ample speed. Hardware 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. For 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 details of the architecture for the next.

The Dorado fits into a very compact package, illustrated in Figure 1a; Figure 1b is a high-level block diagram. Circuits are mounted on large, high density logic boards (288 16-pin DIP logic packages plus 144 8-pin SIP resistor packages per board). The boards slide horizontally into zero-insertion-force connectors mounted in dual backpanels ("sidepanels"); they are .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 a scope probe for debugging and maintenance. We make up for this by providing sophisticated debugging facilities, diagnostics, and the ability to incrementally assemble and test a Dorado from the bottom up.

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. In addition there are up to 4 storage modules, each 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); this is without any enclosing cabinet, however, and the open machine is quite 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 paths, and memories which result help to keep the machine compact. Packaging, however, is not the only consideration. CSL has a large class of applications where doubling the data path width increases performance only a little, because some of the bits contain type codes, flags or whatever which must be examined before an entire datum can be processed. 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. Keeping the machine physically small also improves the speed, since physical distance 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 fundamental concept of the architecture, which motivates much of the processor design, was first tried in the Alto. It works for two main reasons.

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

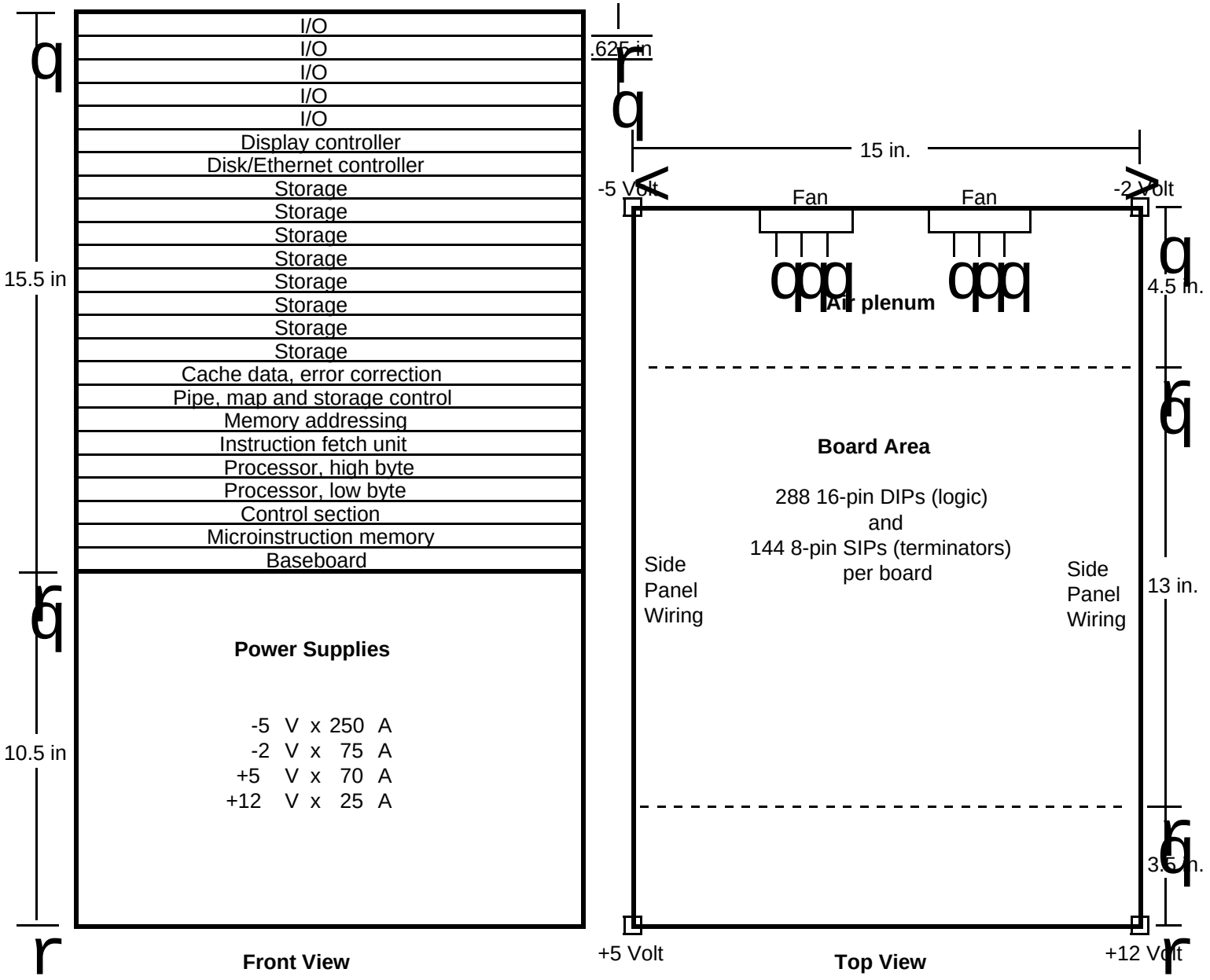


Figure 1a: Dorado chassis

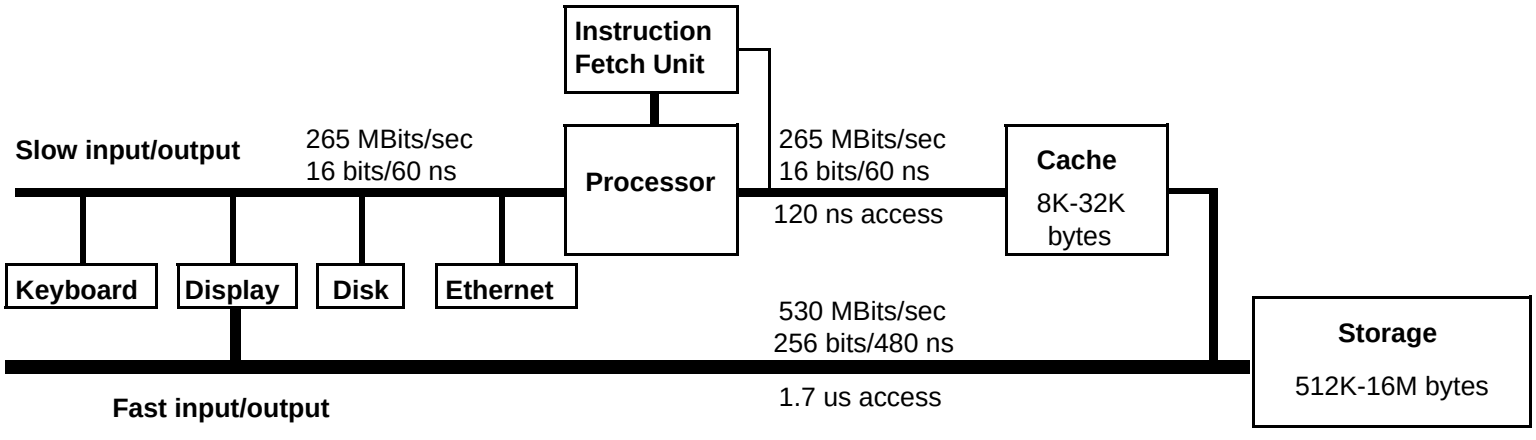


Figure 1b: Dorado block diagram

- Second, when the processor is available to each device, complex device interfaces can be implemented with relatively little dedicated hardware, since most of the control does not have to be duplicated in each interface. For low bandwidth devices, the force of this argument is reduced by the availability of LSI controller chips, but for data rates above one megabit/second no such chips exist as yet.

Of course, to make this sharing feasible, switching the processor must be nearly free of overhead, 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. Thus, the Dorado is implemented using the fastest commercially available technology which has a reasonable level of integration and is not too hard to package. In 1976, the obvious choice was the ECL 10K family of circuits; probably it still is. Secondly, the processor is organized around two pipelines. One allows a microinstruction to be started in each cycle, though it takes three cycle to complete execution. Another allows a processor context switch in each cycle, though it takes two cycles to occur. Thirdly, independent busses communicate with the memory, IFU, and I/O systems, 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 microcode level, so that when new needs arise for fast primitives, they can easily be met by new microcode. In particular, the hardware eliminates constraints on microcode operations and sequencing often found in less powerful designs, e.g., delay in the delivery of intermediate results to registers or in calculating and using branch conditions, or pipeline delays that require padding of microinstruction sequences without useful work. We also included an ample supply of resources: 256 general registers, four hardware stacks, a fast barrel shifter, and fully writeable microstore, to make the Dorado reasonably easy to microcode.

5. Low level architecture

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

5.1 Tasks

There are 16 priority levels associated with microcode execution. These levels are called *microtasks*, or simply *tasks*. 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.

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 presumably receive service from its microcode. Eventually the microcode will *block*, thus 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 arrangment 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 program counter is very powerful.

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.

5.2 Task scheduling

Whenever resources (in this case, the processor) are multiplexed, context switching must only 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, save and restore state, and initiate context switches. A certain amount of overhead results. Furthermore, the presence of a cache introduces large and unpredictable delays in the execution of microcode (because of misses). A polling system would leave the processor idle during these delays, even though the work of another task can usually proceed in parallel. To avoid these costs, the Dorado does task switching on demand of a higher priority device, much like a conventional interrupt system. That is, if a lower priority task is executing and a higher priority device requests a wakeup, the lower priority task will be *preempted*; the higher priority device will be serviced without the consent or even the knowledge of the currently active task. The polling overhead is absorbed by the hardware, which also becomes responsible for *resuming* a preempted task once the processor is relinquished by the higher priority device.

A controller will continue to request a wakeup until notified by the processor that it is about to receive service; it then removes the request, unless it needs more than one unit of service. When the microcode is done, it executes an operation called *Block* which releases the processor. The effect is that requesting service is done explicitly by device controllers, but scheduling of a given 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 able to save and restore state within one microcycle. This is accomplished by keeping the vital state information throughout the processor not in a single rank of registers but in *task specific* registers. These are actually 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 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. For example, COUNT and Q are normally used only by task 0. However, they can be used by other tasks if their contents are explicitly saved and restored.

5.4 Pipelining

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 structure is synchronous, and there is no waiting between stages.

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

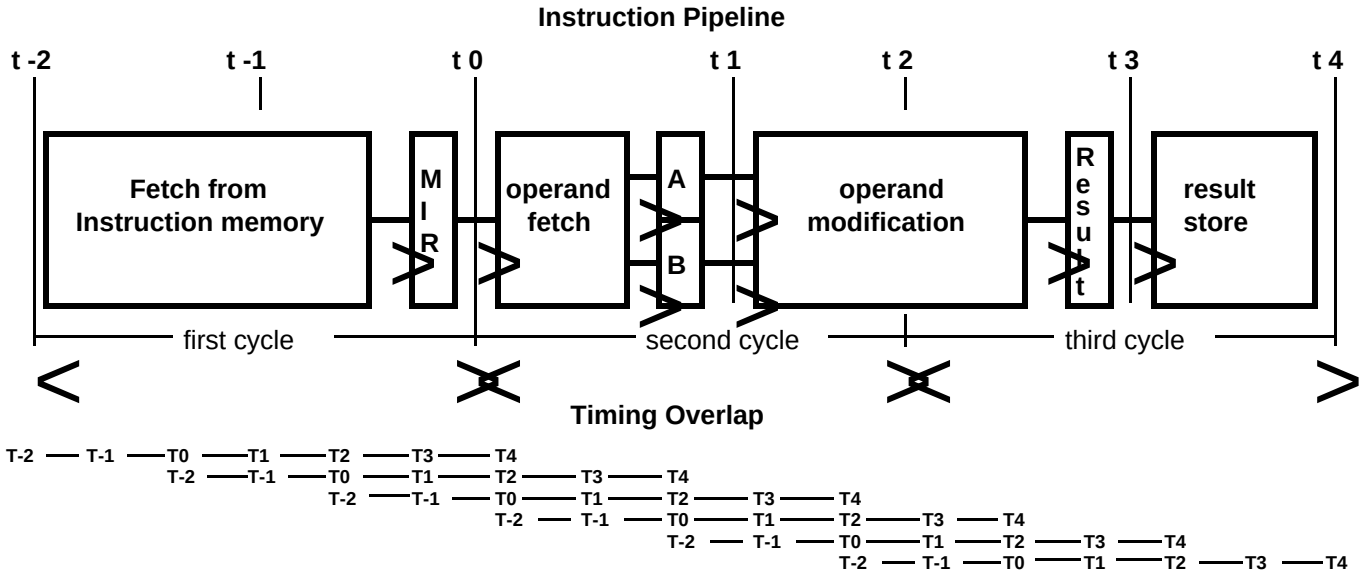


Figure 2: Instruction pipeline and timing overlap

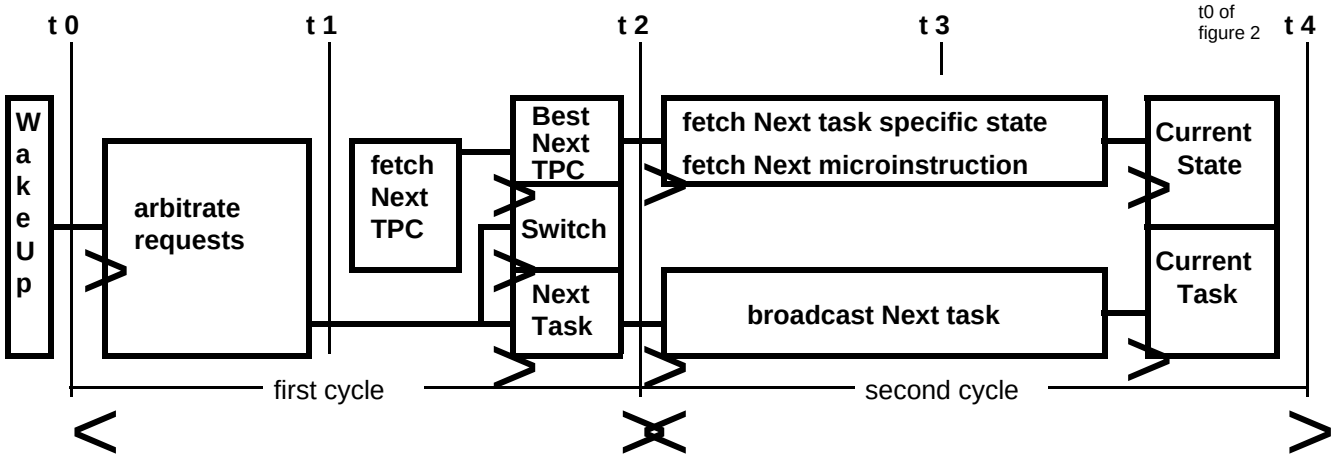


Figure 3: Task arbitration pipeline

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

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

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

5.5 Microinstruction format

One of the key decisions made in the design of any microprogrammed processor is the format and semantics of the microinstruction. The Dorado's demand for compactness and power are at odds in this case. Compactness dictates that an essentially vertical structure be used, with encoded fields specifying many functions in a few bits. The details of the microinstruction format appear in ¶ 6. The major features of interest here are the choice of successor instruction encoding, and the 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 of its successor, NEXTPC; this address is latched with the rest of the instruction, and then used directly to address the microstore for fetching the next instruction. NEXTPC may be modified by state within the processor during execution, but the basic idea is that enough bits must be present in each microword to address the whole microstore. This results in a uniform structure for addressing, and allows the next instruction fetch to proceed without any delay for decoding; it has the disadvantages of increasing the size and cost (and reducing the speed) of the microstore. The lack of any decoding time also makes it impossible to specify a subroutine return or other major change in sequencing, and have it take effect immediately (branches can still use the scheme described below).

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

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

Conditional branching in the Dorado is handled by allowing one of eight branch conditions to modify the low order bit of NEXTPC. This modification (Boolean **or** into the low order bit) takes place about half way into the instruction fetch cycle. The microstore is organized so that this bit does not change the chip address, but instead selects a different chip from a set of chips whose outputs are tied directly together. Since access time from the chip select is considerably faster than from the address, the late arriving branch condition does not increase the total cycle time. For this 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 consequence is that several conditional branches cannot have same target; when this case arises the target must be duplicated. Everything has its price.

Another tradeoff occurs in the mechanism for controlling the functions of the processor at each microcycle. The Dorado encodes most of its operations (other than register selection, ALU operations, storing results, and memory references) in an eight bit function field called *FF*. This is

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

5.6 Data bypassing

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

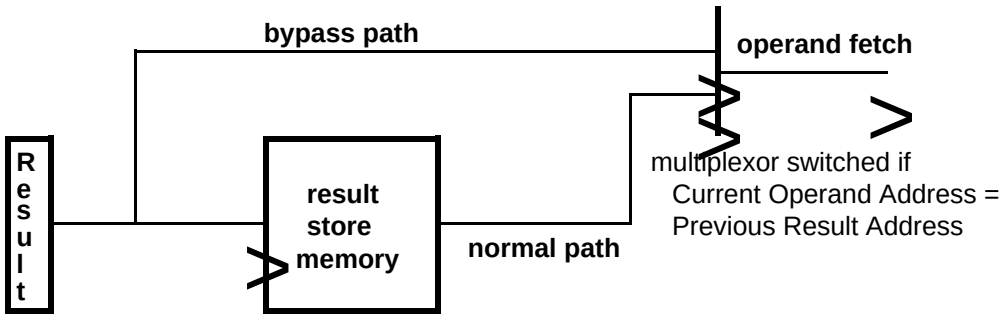


Figure 4: Bypassing example

5.7 Memory delays

Pipelining and bypassing are effective ways to reduce delay and increase throughput within the processor. Interactions with the memory, however, pose different problems. Once a memory reference has been made, there must be some way to tell when the memory system has delivered the requested data. Two simple techniques are to wait a fixed (unfortunately, maximum) amount of time before using the data, or to explicitly poll the memory system. Neither is satisfactory for a high performance machine. First, the difference between the best case (cache hit) and the worst (cache miss plus memory system resource contention) is more than an order of magnitude. Second, useful work can often be performed by a given task before it uses the requested memory data. Third, even if a given task must wait for memory data before it can proceed, higher priority tasks 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, and allowing the processor to keep executing instructions. Only instructions which use memory data or

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

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

5.8 Separate external interfaces

If most macroinstructions (byte codes) are to execute in a small number of cycles, hardware must be provided to make communication among processor, IFU, and memory very quick in the common cases. The Dorado provides a number of data paths and control structures for this purpose, detailed in the block diagrams, Figures 5 and 6. All the busses are a full word wide and can be accessed in one cycle or less. The B input to the ALU is extended to the remainder of the Dorado (except I/O devices, which have their own busses) for the transfer of status and control between the processor and the other subsystems. The memory address bus is a copy of the A side ALU input. 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 fast. The IFU can directly supply operand data to the processor, and any microinstruction can specify that it is the last of a macroinstruction, in which case the successor address is supplied by the IFU. This requires a microstore address bus and operand data bus directly from the IFU to the processor.

It is also desirable to make I/O transfers through the processor fast. To this end there is an I/O address bus and an I/O data bus for direct access to I/O controllers. The data bus can transfer one word per cycle, or 265 megabits/second, and both the memory reference and the I/O transfer can be specified in a single instruction, so that it is possible to move a sequence of words between the cache and a device at this rate. However, this subsystem is called the *slow* I/O system. There is also a more direct memory access I/O subsystem, the *fast* I/O system; it allows data to move directly between storage and I/O devices, in blocks of 16 words, without polluting the cache. Figure 1b 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 constants are necessary, particularly in device controller microcode where they often are used as commands, addresses or literal data. It would be possible to include a *constant box*, addressed perhaps with an FF function, as a source for constants. However, such a box would have a limited size and, 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 positive or small negative (2’s complement) integers, or sparsely populated bit vectors, with the property that one of the two eight bit fields in the constant is all zeroes or all ones. Thus a useful subset of constants can be specified using the eight bits of *FF* for one byte of the constant and two other bits to specify the other byte value and position. Using this technique, most 16 bit constants can be specified in one microinstruction, and any constant can be assembled in two microinstructions. (The "other" two bits come from the *BSelect* field in the microword).

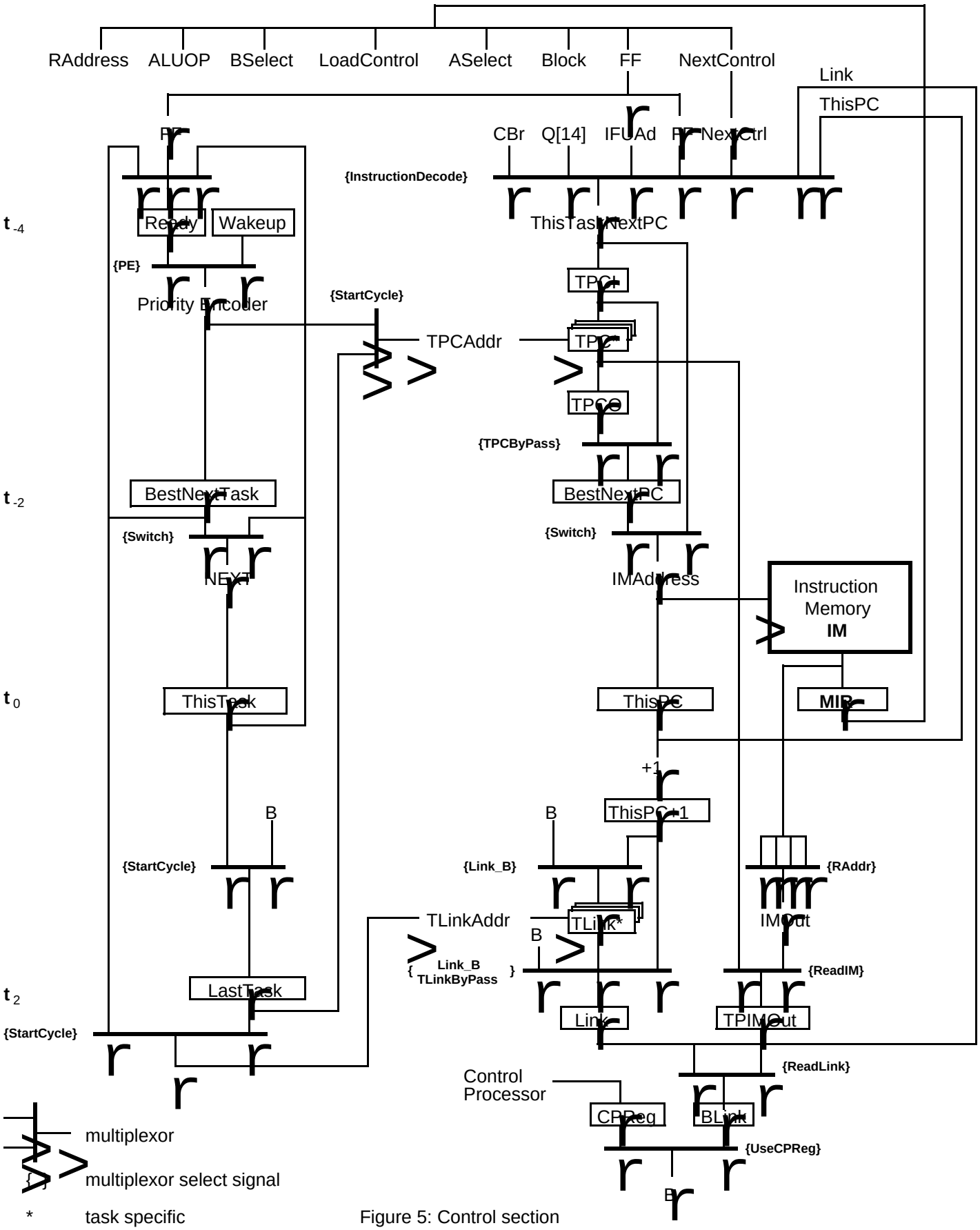


Figure 5: Control section

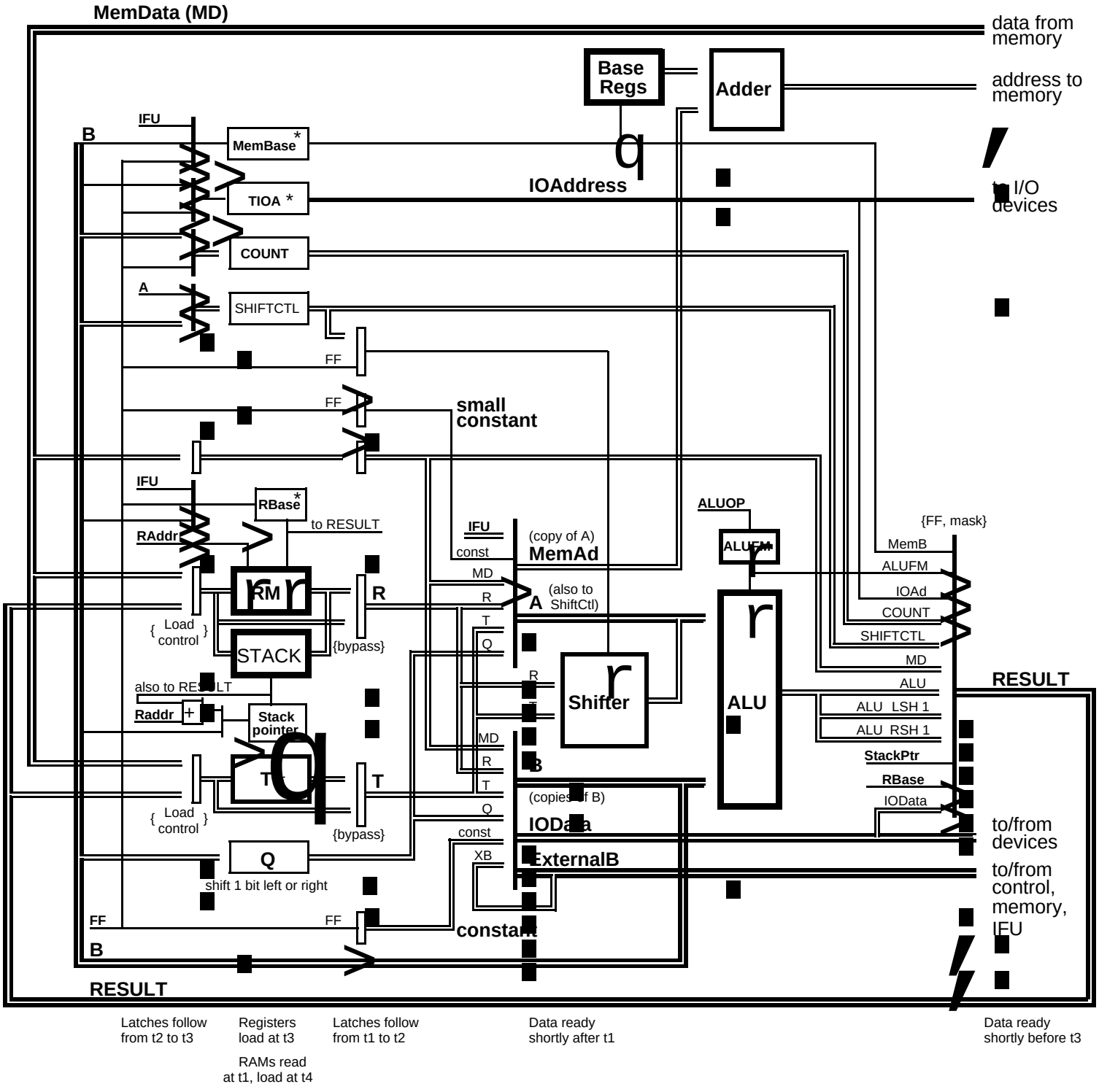


Figure 6: Data section

6. Implementation

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

6.1 Clocks

The Dorado has a fully synchronous clock system, with a clock *tick* every 30 nanoseconds. A *cycle* consists of two successive clock ticks; it begins on an *even* tick, which is followed by an *odd* tick, and completes coincident with the beginning of a new cycle on the next even tick. Even ticks may be labeled with names like t_{-2}, t_0, t_2, t_4 to denote events within a microinstruction execution or a 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 control section fetches and broadcasts the microinstructions to the data section (and the remainder of the Dorado), handles task switching, maintains a subroutine link, and regulates the clock system. It also has an interface to a console and monitoring microcomputer which is used for initialization and debugging 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 the registers are loaded on even clocks. Wakeup requests are latched at t_0 in WAKEUP, one bit per task; READY has corresponding bits for preempted and explicitly readied tasks. The requests in WAKEUP and READY compete. A task can be explicitly made ready by a microcode function. The priority encoder produces the number of the highest priority task, which is loaded into BESTNEXTTASK and also used to read the TPC of this task into BESTNEXTPC; these registers are the interface between the two 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 explicitly relinquish the processor before a lower priority task can run. A bit in the microword, *Block*, is used to indicate that NEXT should get BESTNEXTTASK unconditionally (unless the instruction is held).

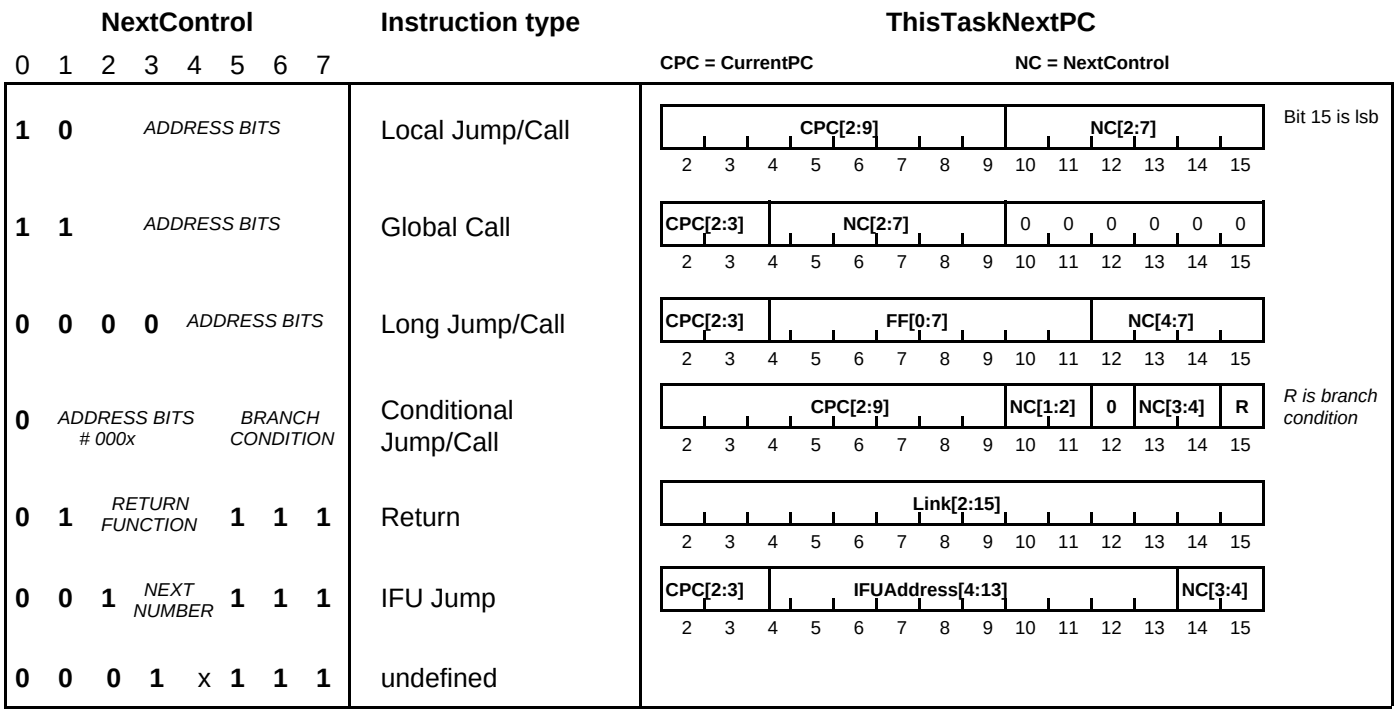
Note that it takes a minimum of two cycles from the time a wakeup changes to the time this change can affect the running task (one for the priority encoding, one to fetch the microinstruction). This implies that a task must execute at least two microinstructions after its wakeup is removed before it blocks; otherwise it will continue to run, since the effects of its wakeup will not have been cleared from the pipe. The device cannot remove the wakeup until it knows that the task will run (by seeing its number on NEXT). Hence the earliest the wakeup can be removed is t_0 of the first instruction (NEXT has the task number in the previous cycle, and the wakeup is latched at t_0); thus 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 memory is heavily pipelined and contains a cache which does not interact with high bandwidth I/O, the I/O microcode often needs to execute only two instructions, in which a memory reference is started and a count is decremented. The processor can then be returned to another task. The maximum rate at which storage references can be made is one every eight cycles (this is the cycle time of the main storage RAMs). A two cycle grain thus allows the full memory bandwidth of 530 megabits/second 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 wakeup should be removed; it would then be unnecessary to broadcast NEXT to the devices. Since this notification could not be done earlier than the first instruction, however, the grain would be three cycles rather than two, and 37.5% of the processor would be needed to provide the full memory bandwidth. Other simplifications in the implementation would result from making the pipeline longer; in particular, squeezing the priority encoding and reading of TPC into one cycle is quite 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 microinstruction register MIR is loaded from the outputs of IM, the microinstruction memory, and the THISPC register is loaded with IMADDRESS. The NEXTPC is quickly calculated based on the *NextControl* field in MIR, which encodes both the instruction type and some bits of NEXTPC; see Figure 7 for details. This calculation produces THISTASKNEXTPC, so called because if a task switch occurs it is not used as the next IMADDRESS. Instead, the BESTNEXTPC computed in the task pipeline is used as IMADDRESS.



Conditional Branch

NC[5:7]	-or-	FF	Branch condition
0		60	ALU = 0
1		61	ALU < 0
2		62	Carry'
3		63	Count=0 (& Count_Count-1)
4		64	R < 0
5		65	R odd
6		66	IOAtten' (non-emulator)
--		67	Overflow'

A long, local or conditional branch is a CALL if, before any modification by branch conditions or dispatches, ThisTaskNextPC[12:15]=0; otherwise it is a jump.

Loaded into Link by Call, Return, or IFUJump

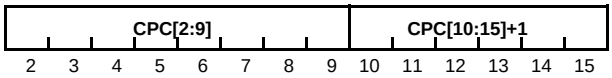


Figure 7: Next address formation

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

6.2.3 Miscellaneous features

There is a task specific subroutine linkage register, LINK, shown in Figure 5, which is loaded with 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 sent to an arbitrary computed address; this allows a microprogram to implement a stack of subroutine links, for example. In addition to conditional branches, which select one of two NEXTPC values, there are also eight-way and 256-way *dispatches*, which use a value on the B bus to select one of eight, or one of 256 NEXTPC values.

Since the Dorado's microstore is writeable, there are data paths for reading and writing it. Related paths allow reading and writing TPC. These paths (through the register TPIMOUT) are folded into already existing data paths in the control section and are somewhat tortuous, but they are used infrequently and hence have been optimized for space. In addition, another computer (either a separate microcomputer or an Alto) serves as the console processor for the Dorado; it is interfaced 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/logic unit (ALU). It implements most of the registers accessible to the programmer and the microcode functions for selecting operands, doing operations in the ALU and shifter, and storing results. It also calculates branch conditions, decodes MIR fields and broadcasts decoded signals to the rest of the Dorado, supplies and accepts memory addresses and data, and supplies I/O data and addresses.

6.3.1 The microinstruction register

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

RAddress	4	Addresses the register bank RM.
ALU*Op*	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 have a variety of sources, as shown in the block diagram. RESULT usually gets the ALU output, but it is also sourced from many other places, including a one bit shift in either direction of the ALU output. 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 in the memory system to form a virtual addresses. EXTERNALB is a copy of B which goes to the control, memory, and IFU sections, and IODATA is another copy which goes to the I/O system; the sources of

B can thus be sent to the entire processor. Both are bidirectional and can serve as a source for B as well. IOADDRESS is driven from a task specific register; it specifies the particular device and register 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 by the current task; if the fetch is not complete, the processor is held when it tries to use MEMDATA. IFUDATA has an operand of the current macroinstruction; as each operand is used, the IFU presents the next one on IFUDATA.

6.3.3 Registers

Here is a list and brief description of registers seen by the microprogrammer. All are one word (16 bits) wide.

RM: a bank of 256 general purpose registers; a register can be read onto A, B, or the shifter, and loaded from RESULT under the control of *LoadControl*. Normally, the same register is both read and loaded in a given microinstruction, but loading of a 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 a microinstruction, it replaces any use of RM, and the *RAddress* field in the microword 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 onto A, B, 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, using only the *NextControl* or *FF* field. It is loaded from B or with small constants from *FF*.

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

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

The next group of registers vary in width. They are used as control or address registers, changed 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 or *FF*, and can be read onto RESULT.

STACKPTR: an eight bit register used as a stack pointer. Two bits of STACKPTR select a stack, 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 stack operation is specified.

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

ALUFM: a 16 word memory which maps the four-bit ALU*Op* field into the six bits required 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* operations. 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 of input from RM and T, performs a left cycle of any number of bit positions, and places the result on A. The 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 is loaded (in one of a variety of ways) with control information, and then one of a group of "shift and mask" microoperations is executed.

6.4 Physical organization

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

The sidepanel pins are distributed in clusters around the board edges to form the major busses. The remaining edge pins are used for point to point connections between two specific boards. The I/O busses go uniformly to all the I/O slots, but all the other boards occupy fixed slots specifically wired for their needs. Half the pins available on the sideplanes are grounded, but wire lengths are not controlled except in the clock distribution system, and no twisted pair is used in the machine 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 that no one pipe stage is significantly longer than the others. Furthermore, we worked hard to make the longest stage (which limits the speed of this fully synchronous machine) as short as possible. The longest stage in the processor, as one might have predicted, is the IMADDRESS calculation and microinstruction fetch in the control slice. There is about a 50 nanosecond limit for reliable operation in a stitchwelded machine, and 60 ns in a multiwired machine. There are pipe stages of 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 the integrated circuit layout and component usage, and by incrementally adding function until nearly the entire board was in use. We also found that performance could be significantly improved by careful layout of critical paths for minimum loading and wiring delay. Although this was a very labor intensive operation, we believe it pays off.

7. Performance

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

The Dorado supports raster scan displays which are refreshed from a full *bitmap* in main memory; this bitmap has one bit for each picture element (dot) on the screen, for a total of .5 1 megabits

(more for gray-scale or color pictures). A special operation called *BitBlt* (bit boundary block transfer) makes it easier to create and update bitmaps; for more information about *BitBlt* consult [9], where it is called *RasterOp*. *BitBlt* makes extensive use of the shifting/masking capabilities of the processor, and attempts to prefetch data so that it will always be in the cache when needed. The Dorado's *BitBlt* can move display objects around in memory at 34 megabits/sec for simple cases like erasing or scrolling a screen. More complex operations, where the result is a function of the source 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 the IODATA and IOADDRESS busses. The microcode for the disk takes three cycles to transfer two words in this way; thus the 10 megabit/sec disk consumes 5% of the processor. Higher bandwidth devices use the fast I/O system, which does not interact with the cache. The fast I/O microcode for the display takes only two instructions to transfer a 16 word block of data from memory to the device. This can consume the available memory bandwidth for I/O (530 megabits/sec) using only one quarter of 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 on the microstore, because of the pages, the odd/even branch addresses, and the special subroutine call locations We were concerned about the amount of microstore which might be wasted by automatic placement of instructions under all these constraints. In fact, however, the automatic placer can use 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. The data section was redesigned and debugged by Roger Bates and Ed Fiala. Peter Deutsch wrote the microcode assembler and instruction placer, and Ed Fiala wrote the Dorado assembler macros, the microprogram debugger, and the hardware manual. Willie-Sue Haugeland, Nori Suzuki, Bruce Horn, Peter Deutsch, Ed Taft and Gene McDaniel are responsible for production and diagnostic microcode.

References

1. Clark, D.W. *et. al*. The memory system of a high-performance personal computer. Technical Report CSL-81-1, Xerox Palo 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 Workshop*, Pacific 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 Report CSL-81-1, 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, April 1979.

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 edition, Sieworek, Bell and Newell, eds., McGraw-Hill, 1981. Also in Technical Report CSL-79-11, Xerox Palo Alto Research Center, August 1979.

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

An Instruction Fetch Unit for a High-Performance Personal Computer

by Butler W. Lampson, Gene A. McDaniel and Severo M. Ornstein

January 1981

ABSTRACT

The instruction fetch unit (IFU) of the Dorado personal computer speeds up the emulation of instructions by pre-fetching, decoding, and preparing later instructions in parallel with the execution of earlier ones. It dispatches the machine's microcoded processor to the proper starting address for each instruction, and passes the instruction's fields to the processor on demand. A writeable decoding memory allows the IFU to be specialized to a particular instruction set, as long as the instructions are an integral number of bytes long. There are implementations of specialized instruction sets for the Mesa, Lisp, and Smalltalk languages. The IFU is implemented with a six-stage pipeline, and can decode an instruction every 60 ns. Under favorable conditions the Dorado can execute instructions at this peak rate (16 mips).

This paper has been submitted for publication.

CR CATEGORIES

6.34, 6.21

KEY WORDS AND PHRASES

cache, emulation, instruction fetch, microcode, pipeline.

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

1. Introduction

This paper describes the instruction fetch unit (IFU) for the Dorado, a powerful personal computer designed to meet the needs of computing researchers at the Xerox Palo Alto Research Center. These people work in many areas of computer science: programming environments, automated office systems, electronic filing and communication, page composition and computer graphics, VLSI design aids, distributed computing, etc. There is heavy emphasis on building working prototypes. The Dorado preserves the important properties of an earlier personal computer, the Alto [13], while removing the space and speed bottlenecks imposed by that machine's 1973 design. The history, design goals, and general characteristics of the Dorado are discussed in a companion paper [8], which also describes its microprogrammed processor. A second paper [1] describes the memory system.

The Dorado is built out of ECL 10K circuits. It has 16-bit data paths, 28 bit virtual addresses, 4K-16K words of high-speed cache memory, writeable microcode, and an I/O bandwidth of 530 Mbits/sec. Figure 1 shows a block diagram of the machine. The microcoded processor can execute a microinstruction every 60 ns. An *instruction* of some high level language is performed by executing a suitable succession of these microinstructions; this process is called *emulation*.

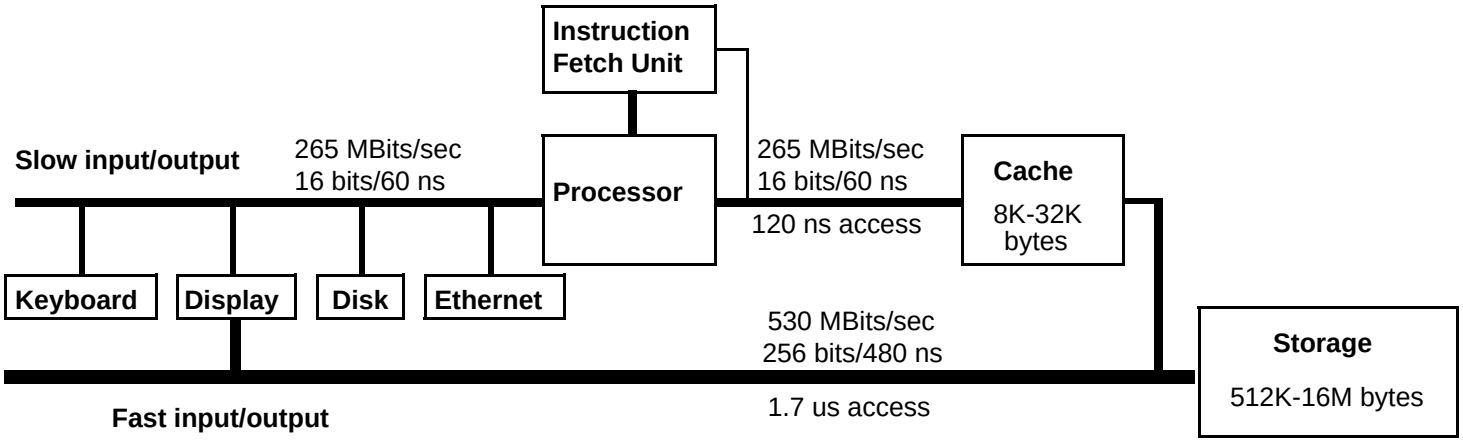


Figure 1: Dorado block diagram

The purpose of the IFU is to speed up emulation by pre-fetching, decoding, and preparing later instructions in parallel with the execution of earlier ones. It dispatches the machine's microcoded processor to the proper starting address for each instruction, supplies the processor with an assortment of other useful information derived from the instruction, and passes the instruction's various fields to the processor on demand. A writeable decoding memory allows the IFU to be specialized to a particular instruction set; there is room for four of these, each with 256 instructions.

There are implementations of specialized instruction sets for the Mesa [9], Lisp [12], and Smalltalk [5] languages, as well as an Alto [13] emulator. The IFU can decode an instruction every 60 ns, and under favorable conditions the Dorado can execute instructions at this peak rate (16 MIPS).

Following this introduction, we discuss the problem of instruction execution in general terms and outline the space of possible solutions (¶ 2). We then describe the architecture of the Dorado's IFU (¶ 3) and its interactions with the processor which actually executes the instructions (¶ 4); the reader who likes to see concrete details might wish to read these sections in parallel with ¶ 2. The next section deals with the internals of the IFU, describing how to program it and the details of its pipelined implementation (¶ 5). A final section tells how large and how fast it is, and gives some information about the effectiveness of its various mechanisms for improving performance (¶ 6).

2. The problem

It has long been recognized that the algorithm for executing an *object program* can be most easily described by another program, called an *interpreter*, which treats both the instructions and the data of the object program as its own data. The simplest microprogrammed computers actually do execution in just this way; the microinstructions can specify only general-purpose data manipulations, and all the knowledge about the instructions being emulated is expressed in the microprogram.

We illustrate this point with the following fragment of an emulator for a stack-based instruction set. The fragment includes the basic instruction fetch operation and code for two instructions: *PushConstant*, which pushes the next instruction byte onto the stack, and *PushLocalVar*, which pushes the contents of the local variable addressed by the next byte (relative to a pointer in the register *localData*). The notation is self-explanatory for the most part. Microinstructions are separated by semicolons, and parallel operations in the same microinstruction by commas. This code uses no special-purpose operations, except that we have compressed the details of the stack manipulation into a *Push* operation.

```
Registers: PC, localData, opcode, temp

GetInstruction:                          -- Top of the microcode instruction emulation loop.
    Fetch[PC];                           -- Start a memory fetch from address in PC; data arrives later.
    PC _PC+1;                            -- Increment PC register for next instruction.
    if interruptPending then goto processInterrupt
    opcode _ memoryData;                 -- Use the memory data we previously fetched
    goto opcode;                         -- The opcode value is the starting microcode address.

PushConstant:                            -- Dispatch address for the PushConstant instruction.
    Fetch[PC];                           -- PC points to the next instruction byte.
    PC _ PC+1;                           -- Increment PC register for next instruction.
    Push[memoryData], goto GetInstruction;

PushLocalVar:                            -- Dispatch address for the PushLocalVar instruction.
    Fetch[PC];                           -- Fetch the next instruction byte, which is the index in the local data for the
                                             variable to be pushed.
    PC _ PC+1;
    temp _ memoryData;
    temp _ temp+localData;               -- Now temp is the address of the local variable.
    Fetch[temp];
    Push[memoryData], goto GetInstruction;
```

In order to make this emulator run faster (given a fixed time for each primitive operation, presumably established by circuit speeds), it is necessary to do more of the operations concurrently. One possibility is to enhance the processor, so that it can do several operations in a single microinstruction. For instance, the first two microinstructions might be replaced by

```
    Fetch[PC], PC _ PC+1;                -- Start a memory fetch from address in PC; data arrives later. Increment PC
                                             for next instruction.
```

This approach is fine as far as it goes, but it is limited to combining independent operations. A *Fetch* and the following retrieval of data, for example, cannot be combined without making the microinstruction slower, since the memory takes time to respond.

A second approach is to make several copies of the entire processor, and let them work on several instructions at once. With *n* copies, this would run *n* times as fast if there were no synchronization problems; it would also be very simple to implement (though perhaps not cheap). Unfortunately, a program written in a conventional language and encoded into a conventional instruction set typically has a great deal of interaction between the successive instructions. For instance, consider the instruction sequence *PushConstant*, *PushLocalVar*, *Add*. We see from the microcode above that all three instructions need to reference the stack; this is *contention* for the same resource. Furthermore, the *Add* instruction needs the contents of the stack *after* both the previous instructions are finished; this is not only contention, but *dependency* of one instruction on the results of another.

In spite of these problems. this approach can be made to work, especially for numeric computations, and in conjunction with a sympathetic compiler. Indeed, it is used in high-performance machines such as the CDC 6600 [14] and 7600, the IBM 360/91 [16], the MU5 [4], and the Cray-1 [10]; typically only part of the processor is duplicated, often into specialized devices called *functional units*. However, with 1977 technology this approach is too expensive for a personal machine, and hence was not considered for the Dorado.

A third possibility (often combined with the second) is to *pipeline* the execution of an instruction by dividing it into parts, each one to be performed by a separate processor or *stage*. Different stages can operate concurrently on successive instructions. In this example, we might have one stage for fetching the instruction (*GetInstruction*), and another for executing it (*PushConstant* and *PushLocalVar*). Successive instructions can then execute as follows (where each line represents a "major cycle").

GetInstruction[1]				
Execute[1]	*GetInstruction*[2]			
	Execute[2]	*GetInstruction*[3]		
		Execute[3]	*GetInstruction*[4]	. . .

Each instruction spends the same amount of time executing as before, but the throughput is doubled.

2.1 About pipelines

An ideal pipeline has *no* communication between the stages except when work is passed from one stage to its successor. The unit of work which is passed between stages is called an *item*. The crucial problems in designing a pipeline are:

hand-off of items from one stage to the next;

buffering of items within a stage;

contention among stages for resources (a form of communication);

dependency of one stage on the activity of another (also a form of communication). Particularly troublesome is *backward* dependency, in which an early stage depends on the results of a later one (e.g., a conditional branch);

irregularity in the flow of items through the pipe. This can arise from variations in the rate of:

processing items in the different stages (e.g., memory fetches may be slow, or variable in rate, or both);

input (e.g., fetch requests to a memory pipe);

output (e.g., decoded instructions from an IFU pipe).

The main performance parameters of a pipeline are:

throughput or *bandwidth* the rate at which items are processed to completion when there are no dependencies (let t be the time to complete one item);

latency the time for one item to traverse the entire pipeline when it is otherwise empty (let l be the latency);

elasticity the ability of the pipe to deliver results at full bandwidth in spite of irregularity. More buffering means more elasticity, more bits of storage in the pipe, and perhaps more latency.

A *synchronous*, *uniform* pipeline is one in which each stage takes the same amount of time. With n stages we have $l=nt$, where t is the time of each stage. With many small stages, t can be made small and the throughput high, at the expense of the latency. The only absolute limit to this process is the cost of synchronization between stages (which is a lower bound on t; in a synchronous pipeline this is the time to pass through a register).

The minimum time to do the smallest indivisible piece of work (e.g., to read from an internal RAM) tends to be a practical limit also. This limit can be evaded, however (at some cost), by making n copies of the hardware, assigning the work to them in round-robin fashion, and selecting the results by the same round-robin rule. If a single stage has $t=s$, such a *duplicated* stage has $t=s/n$ plus the time for multiplexing the results. When this method is used, the copies are usually called *functional units*.

Usually the main goal is to maximize the throughput; in the absence of dependencies latency is unimportant. As dependencies increase, however, latency becomes more important. To see why this is true, consider the backward dependency caused by a conditional branch. Strictly speaking, when a branch instruction is encountered, fetching cannot proceed until the result of the branch is known. When it is, the target instruction of the branch must traverse the pipe before any more instructions can be completed. If w is the fraction of branch instructions, the average completion time will be $t+wl$. Thus if $l=5t$ (a five stage uniform pipe), a w of 20% will halve the throughput. In this example, of course, it is sensible to make a guess and follow one path, so that w is the fraction of instructions for which a wrong guess is made; note that $w=20\%$ is fairly accurate prediction. Following a guessed path is easy because there are no forward dependencies (program state is never changed by instruction fetching), so that a wrong path can be abandoned with no ill effects. However, no such shortcut is possible in the case of the *Add* instruction mentioned earlier, because it isn't practical to guess the result of the *PushLocalVar*.

2.2 Pipelining instruction execution

Let us now see how to apply these ideas to instruction execution. Following many earlier designs (e.g., [4, 16]), we can divide this task into four stages:

instruction fetching and preparation;

operand preparation: address calculation, fetching and reformatting;

computation;

result storage.

Each of these in turn may be divided into sub-stages. We observe that in any conventional architecture there are many dependencies among the last three stages, because results are constantly being stored into memory or register locations from which operands are fetched. Furthermore, if *every* store operation is regarded as a dependency, there could never be much concurrency. Hence it is necessary to compare the address of each location modified by a store with all the addresses referenced by earlier stages. Even these dependencies are common enough to be painful; hence provision is usually made in such a pipeline for modifying the actions of earlier stages when operands are changed by stores. As a result of all this, pipelining the last three stages of instruction is a complex and expensive business. A fast multi-port cache inside the processor makes the problem much easier, but is not feasible with this technology. An interesting but untried idea is to impose programming restrictions which forbid harmful dependencies; if all the code is generated by compilers this is quite feasible.

Hardly any of these problems arise, however, in separating instruction fetching from the rest. If we assume that execution cannot modify the code being executed, there are *no* dependencies except those arising from branches. If this assumption is unacceptable, then checks must be made for such modifications, but since they are rare in practice, the checks can be at a very coarse grain, and fairly drastic resetting actions can be taken. The absence of forward dependencies means that instruction fetching activities can be abandoned without any communication to other parts of the machine.

The function of an instruction fetching and preparation stage or IFU, then, is to hand off to the rest of the machine the relevant information for each instruction, conveniently formatted for later use. Whether the rest of the machine is a single microcoded processor, an operand preparation stage in a pipeline, or a collection of functional units which can operate concurrently is unimportant to the IFU, except as it affects the meaning of "conveniently formatted." We will call this part of the machine the *execution unit* or EU, and will not be much concerned with its internal structure.

The EU demands instructions from the IFU at an irregular rate, depending on how fast it is able to absorb the previous ones. A simple machine must completely process an instruction before demanding the next one. In a machine with multiple functional units, on the other hand, the first stage in the EU waits until the basic resources required by the instruction (adders, result registers, etc.) are available, and then hands it off to a functional unit for execution. Beyond this point the operation cannot be described by a single pipeline, and complete execution of the instruction may be long delayed, but even in this complicated situation the IFU still sees the EU as a single consumer of instructions, and is unaware of the concurrency which lies beyond.

Under this umbrella definition for an IFU, a lot can be sheltered. To illustrate the way an IFU can accommodate specific language features, we draw an example from Smalltalk [5]. In this language, the basic executable operation is applying a function *f* (called a *method*) to an object *o*: *f*(*o*, . . .). The address of the code for the function is not determined solely by the static program, but depends on a property of the object called its *class*. There are many implementation techniques for finding the class and then the function from the object. One possibility is to represent a class as a hash table which maps function names (previously converted by a compiler into numbers) into code addresses, and to store the address of this table in the first word of the object. The rather complex operation of obtaining the hash table address and searching the table for the code address associated with *f*, is in the proper domain of an IFU, and removes a significant amount of computation from the processor. No such specialization is present in the Dorado's IFU, however.

2.3 Pipelining instruction fetches

For the sake of definiteness, we will assume henceforth that

> the smallest addressable unit in the code is a byte;
>
> the memory delivers data in units called *words*, which are larger than bytes;
>
> an instruction (and its addresses, immediate operands, and other fields) may occupy one or more bytes, and the first byte determines its essential properties (length, number of fields, etc.).

Matters are somewhat simplified if the addresssable unit is the unit delivered by the memory or if instructions are all the same length, and somewhat complicated if instructions may be any number of bits long. However, these variations are inessential and distracting.

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

Buffering may be introduced between any pair of these stages, either the minimum of one item required to separate the stages, or a larger amount to increase the elasticity. Note that an item must be a *word* early in the pipe (at the interface to the memory), must be an *instruction* late in the pipe (at the interface to the EU), and may need to be a *byte* in the middle.

There are three sources of irregularity (see ¶ 2.1) in the pipeline, even when no wrong branches are taken:

The instruction length is irregular, as noted in the previous paragraph; hence a uniform flow of instructions to the EU implies an irregular flow of bytes into the decoder, and vice versa.

The memory takes an irregular amount of time to fetch data; if it contains a cache, the amount of time may vary by more than an order of magnitude.

The EU demands instructions at an irregular rate.

These considerations imply that considerable elasticity is needed in order to meet the EU's demands without introducing delays.

2.4 Hand-off to the EU

From the IFU's viewpoint, handing-off an instruction to the EU is a simple producer-consumer relationship. The EU demands a new instruction. If one is ready, the IFU delivers it as a pile of suitably formatted bits, and forgets about the instruction. Otherwise the IFU notifies the EU that it is not ready; in this case the EU will presumably repeat the request until it is satisfied. Thus at this level of abstraction, hand-off is a synchronized transfer of one data item (a decoded instruction) from one process (the IFU) to another (the EU).

Usually the data in the decoded instruction can be divided into two parts: information about what to do, and parameters. If the EU is a microprogrammed processor, for example, what to do can conveniently be encoded as the address of a microinstruction to which control should go (a *dispatch* address), and indeed this is done in the Dorado. Since microinstructions can contain immediate constants, and in general can do arbitrary computations, it is possible in principle to encode all the information in the instruction into a microinstruction address; thus the instructions *PushConstant*(3) and *PushConstant*(4356) could send control to different microinstructions. In fact, however, micro-instructions are expensive, and it is impractical to have more than a few hundred, or at most a few thousand of them. Hence we want to use the same microcode for as many instructions as possible, representing the differences in *parameters* which are treated as data by the microcode. These parameters are presented to the EU on some set of data busses; ¶ 4 has several examples.

Half of the IFU-EU synchronization can also be encoded in the dispatch address: when the IFU is not ready, it can dispatch the EU to a special *NotReady* location. Here the microcode can do any background processing it might have, and then repeat the demand for another instruction. The same method can be used to communicate other exceptional conditions to the EU, such as a page fault encountered in fetching an instruction, or an interrupt signal from an I/O device. The Dorado's IFU uses this method (see ¶ 3.4).

Measurements of typical programs [7, 11] reveal that most of the instructions executed are simple, and hence can be handled quickly by the EU. As a result, it is important to keep the cost of hand-off low, since otherwise it can easily dominate the execution time for such instructions. As the EU gets faster, this point gets more important; there are many instructions which the Dorado, for instance, can execute in one cycle, so that one cycle of hand-off overhead would be 50%. This point is discussed further in ¶ 3 and 4.

2.5 Autonomy

Perhaps the most important parameter in the design of an IFU is the extent to which it functions independently of the execution unit, which is the master in their relationship. At one extreme we can have an IFU which is entirely independent of the EU after it is initialized with a code address (it might also receive information about the outcome of branches); this initialization would only occur on a process switch, complex procedure call, or indexed or indirect jump. At the other extreme is an IFU which simply buffers one word of code and delivers successive bytes to the EU; when the buffer is empty, the IFU dispatches the EU to a piece of microcode which fetches another memory word's worth of code into the buffer. The first IFU must decode instruction lengths, follow jumps, and provide the program counter for each instruction to the EU (e.g., so that it can be saved as a

return link). The second leaves all these functions to the EU, except perhaps for keeping track of which byte of the word it is delivering. One might think that the second IFU cannot help performance much, but in fact when working with a microcoded EU it can probably provide half the performance improvement of the first one, at one-tenth the cost in hardware. The reason can be seen by examining the interpreter fragment at the beginning of ¶ 2; half a dozen micro-instructions are typically consumed in the clumsy *GetInstruction* operation, and things get worse when instructions do not coincide with memory words.

When deciding what trade-offs to make, one important parameter is the speed of the EU. It is pointless to be able to execute most instructions in one or two cycles, if several cycles are consumed in *GetInstruction*. Hence a fast EU must have an autonomous IFU. An important special case is the speed of the memory relative to the microinstruction time. If several microinstructions can be executed in the time required to fetch the next instruction from memory, the processor can use this time to hold the IFU's hand, or to perform the *GetInstruction* itself. On the Dorado, the cache ensures that memory data arrives almost immediately, so there is no free time for handholding.

An autonomous IFU must do more than simply transforming instructions into a convenient form for the EU. There are two natural ways in which its internal operation may be affected by the instruction stream: decoding instruction lengths, and following branches. Any IFU which handles more than one instruction without processor intervention must calculate instruction lengths. Following branches is desirable because it avoids the cost of a start-up latency at every branch instruction (typically every fifth instruction is a branch). However, it does introduce potential complications because a conditional branch must be processed without accurate information (perhaps without any information) about the actual value of the condition; indeed, often this value is not determined until the processor has executed the preceding instruction. A straightforward design decides whether to branch based on the opcode alone, and the processor restarts the IFU at the correct address if the decision turns out to be wrong.

The branch decision may be based on other historical information. The S-1 [17], for instance, keeps in its instruction cache one bit for each instruction, which records whether the instruction branched last time it was executed. This small amount of partial history reduces the fraction of incorrect branch decisions to 5% [Forest Baskett, personal communication]. The MU5 [4] remembers the *addresses* of the last eight instructions which branched; such a small history leaves 35% of the branches predicted wrongly, but the scheme allows the prediction to be made *before* the instruction is fetched. More elaborate designs [16] follow both branch paths, discarding the wrong one when the processor makes the branch decision. Each path may of course encounter further branches, which in turn may be followed both ways until the capacity of the IFU is exhausted. If each path is truly followed in parallel, then following n paths will in general require n times as much hardware and n times as much memory bandwidth as following one path. Alternatively, part or all of the IFU's resources may be multiplexed between paths to reduce this cost at the expense of bandwidth.

2.6 Buffering

As we saw in ¶ 2.2, a pipeline with any irregularities must have buffering to provide elasticity, or its performance at each instant will approximate the performance of the slowest stage at that instant; this maximizing of the worst performance is highly undesirable. From the enumeration in ¶ 2.3 of irregularities in the IFU, we can see that to serve the EU smoothly, there should be a buffer between the EU and any sources of irregularity, as shown in Figure 2. Similarly, to receive words from the irregular memory, there should be a buffer between the memory and any sources of irregularity. Because of the irregularity caused by variable length instructions, a single buffer cannot serve both functions. Note that additional regular stages (some are shown in the figure) have no effect one way or the other.

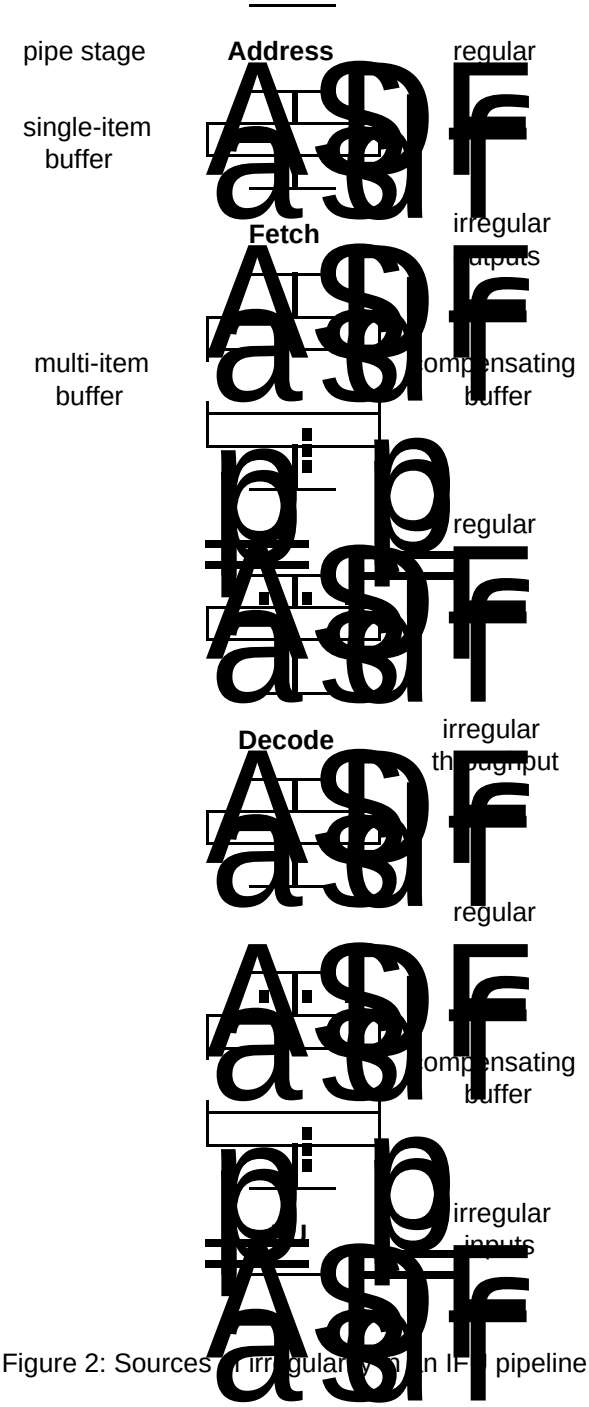


Figure 2: Sources of irregularity in an IFU pipeline

The cost of introducing a buffer (in the ECL 10K MSI technology) is the RAM storage to implement it, a multiplexor to bypass it when it is empty, and its control; see Figure 6 for details. The bypass ensures that the buffer does not increase the latency. In addition, there is typically a very minor performance penalty: when the pipe is reset, any external resources (the memory in the case of the IFU) which have been used to fill the buffers are wasted. If some other processor could make better use of the resources, something has been lost.

3. Architecture of the Dorado IFU

We now turn from our discussion of general principles to the actual IFU of the Dorado. Its structure follows from the principles of the previous section, though we must admit that the design in fact proceeded less from general principles than from the goal of delivering one decoded instruction per microcycle. This performance requirement dictates an autonomous IFU, and it also requires careful attention to the details of IFU-EU hand-off. In the Dorado the EU is a microcoded processor with a number of data paths, and a pipelined implementation which allows it to execute a microinstruction every 60 ns; in order to remind the reader of this implementation, we use the word "processor" to denote the Dorado's EU. The processor does not have any significant concurrency visible to the microprogram, however. In particular, all the work done in a given cycle is specified directly by the microinstruction executed in that cycle, although memory references are done by an autonomous unit which in fact is shared with the IFU; see Figure 1.

The processor gives the IFU an initial program counter (PC), 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 PC, unless a fault or interrupt is detected. For each instruction the IFU supplies a microcode dispatch address (into which *NotReady* and all other exceptions are encoded), some bits of initial state for the processor, a sequence of *field* data values, and the PC value for the first byte of the instruction. The uses made of this information are described in ¶ 4.

3.1 Byte codes

The IFU's interpretation of the code is based on a definite model of how instructions are encoded. Although this model is not specialized to the details of a particular instruction set, good performance depends on adherence to certain rules. The IFU deals only with instructions encoded as variable length byte sequences *byte codes* [3, 11]. Variable length instructions provide code compaction, since frequent instructions can be small. 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 bits as the 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 somewhat at the expense of more complex length calculation and word disassembly.

The first byte of each instruction, called the *opcode*, is decoded by full 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, which also must restart the IFU at the proper point.

3.2 The decoding table

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 27-bit contents of the table describe the instruction in sufficient detail for the IFU and the processor to do their jobs, and 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.

This method of instruction decoding has a number of advantages. It makes the decoder completely programmable in a very simple and economical way. It also allows any substructure of the opcode

(e.g., register or mode fields) to be extracted with complete flexibility. Indeed, it is not necessary for such fields to exist explicitly. If single-byte *PushConstant* instructions for values 0-4 are desired, any five opcode values can be assigned for this purpose, and the table can produce the values 0-4. Furthermore, no sharp distinction is needed between "control" and "data" in the instruction encoding, since both control information and data values are produced by the same table lookup.

Of course nothing is perfect. This scheme may fail when an instruction has many small fields, especially if they cross byte boundaries. The PDP-11 and Nova instruction sets are interesting borderline cases: it works quite well to look up the first byte and use the result to select either a second lookup in an *alternate* table lookup, or treatment of the next byte as data. A convenient way to describe this is to have the first byte specify either a two byte instruction, or a one byte instruction which switches the "instruction set" temporarily for decoding the next byte.

This facility of modifying the instruction set register on the fly is not implemented in the Dorado, since it is not very useful for the instruction sets we actually use. It is simple, however, and could easily be added; the only delicate point is that the instruction set register must be saved on an exception, or else exceptions must be prohibited before instructions which are decoded with an alternate table. Currently only the processor can change the instruction set, and it normally does so only when switching from one language to another. This facility is used in the Interlisp implementation, for example, since the nucleus of this system is written in BCPL and compiled into a different instruction set than the one used for Lisp.

Multiple decoding tables have other uses. In fact, the IFU can be viewed as a rather general byte-stream processor. For example, consider the problem of generating halftone values for a grey scale image: The task is to transform a sequence of grey pixels (p_g bits each, at a resolution of r_g pixels/inch), into a sequence of binary pixels (one bit each, at a resolution of r_b pixels/inch). Both sets of pixels are packed into words, $16/p_g$ per word and 16 per word respectively. Thus as each binary pixel is generated, it is necessary to keep track of whether a new binary word must be started (once every 16 binary pixels), and whether a new grey pixel is needed (once every r_b/r_g binary pixels); in the latter case, a new grey word may be needed. Typical algorithms use a single scan-line buffer containing an error value which must be compensated at each binary pixel. The IFU can be used to fetch values from this buffer in parallel with the processor. Special pseudo-opcode values can be used to mark the points which require one or more of the special actions above. The decoding table will dispatch the processor to the special code for these functions without any processor overhead. A trial implementation using this idea was about twice as fast as one without the IFU.

3.3 Pipeline stages and buffering

Figure 3 shows the pipeline stages in the IFU. An item varies in size, but all stages except one operate in a single 60 ns cycle. For the most part all state is held in the buffers between the stages, which themselves are purely functional or combinatorial.

At the beginning of the pipe, PC values are generated and put on the memory address bus (ADDRESS), and the corresponding 16-bit words are returned from the memory (MEMORY), at a peak rate of one per cycle. If there are no cache misses and no collisions with the processor, the memory can accept an address in every cycle and return data words at the same rate two cycles later. Thus under these ideal conditions the memory is not irregular. A double-rate (30 ns) stage (BYTES) delivers bytes to the decoder (DECODE), which can accept one opcode byte and one operand byte in a single cycle, though it requires a full cycle to process an instruction. This arrangement allows two-byte instructions to pass through the pipe at the rate of one per cycle; longer instructions require two cycles, but are rare. Because DECODE requires a full cycle, the peak rate for one byte instructions is still one per cycle. Note that the processor cannot demand instructions faster than this anyway.

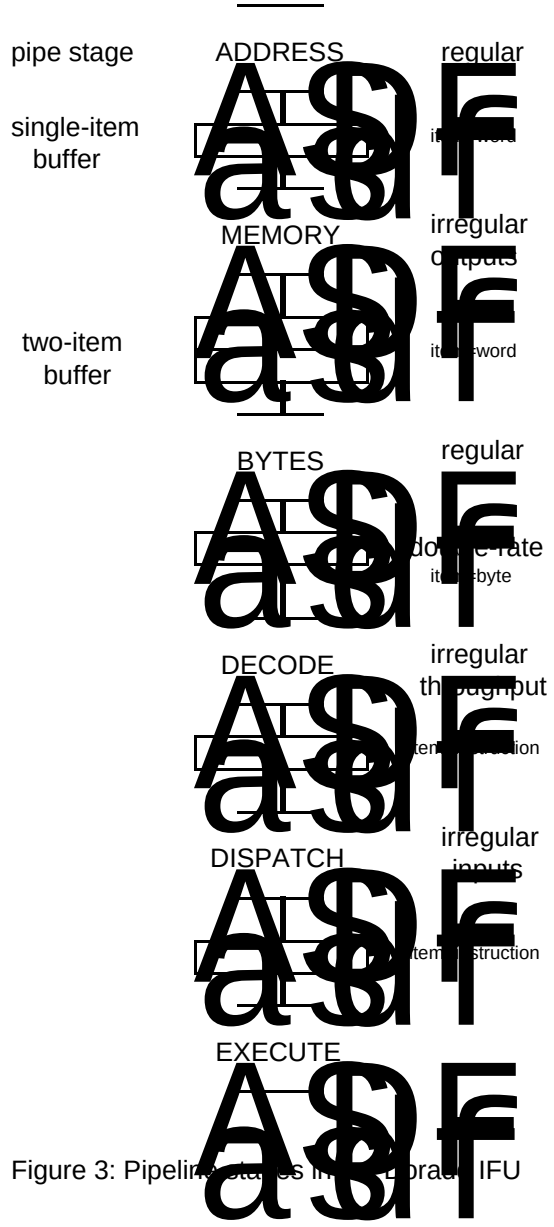


Figure 3: Pipeline stages in the broad IFU

From DECODE on, an item is an instruction; one of these items is held in a buffer from which it is handed off to the processor (DISPATCH). It turns out that the processor proper requires some of the decoded instruction before it executes the first microinstruction (the dispatch address and other initial state; see ¶ 4.2), but consumes the field data later, one byte at a time. The physical IFU also contains a logical extension of the processor (EXECUTE), which holds this deferred information and doles it out on demand.

There are two words of buffering after MEMORY, but there is no other buffering except for the minimum single item between stages, contrary to the arguments of ¶ 2.6. This design was adopted partly to save space, and partly because we did not fully understand the issues in maintaining peak bandwidth. Fortunately the peak bandwidth of the IFU is substantially greater than what the processor is likely to demand for more than a very short interval (see ¶ 6), so that not much useful throughput is lost because of the inadequate buffering.

3.4 Exceptions

Exception conditions are handled by extending the space of values stored in an item 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; the resulting "page fault value" is propagated through the pipe 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 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 PC 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.

The exceptions may be divided into three classes:

1) the IFU has not (yet) 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.

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

3.5 Contention and dependencies

There is no contention for resources within the IFU, and the only contention with the rest of the Dorado is for access to the memory. The IFU shares with the processor a single address bus to the Dorado’s cache, but has its own bus for retrieving data. The processor has highest priority for the address bus, which can handle one request per cycle. Thus under worst-case conditions the IFU can be locked out completely; eventually, of course, the processor will demand an instruction which is not ready and stop using the bus. Actual address bus conflicts are not a major factor (see ¶ 6.3).

Although ideally the MEMORY stage is regular, in fact collisions with the processor can happen; these irregularities are partially compensated by the two words of buffering after MEMORY. In addition cache misses, though very rare, cost about 30 cycles when they do occur.

There is only one dependency on the rest of the execution pipeline: starting the IFU at a new PC. Since no attempt is made to detect modifications of code being executed, or to execute branches which depend on the values of variables, the only IFU-processor communication is hand-off synchronization and resetting of the PC, and these are also the only communication between the IFU stages. The IFU is completely reset when it gets a new PC; no attempt is made to follow more than one branch path, or to cache information about the code within the IFU. The shortage of buffering makes the implementation of synchronization rather tricky; see ¶ 5 .

The IFU takes complete responsibility for keeping track of the PC. Every item in the pipe carries its PC value with it, so that when an instruction is delivered to the processor, the PC is delivered at the

same time. The processor actually has access to all the information needed to maintain its own PC, 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 PC-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 PC 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 PC-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 PC 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 pased 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.

4. IFU-processor hand-off

With a microcoded execution unit like the Dorado's processor, efficient emulation depends on smooth interaction between the IFU and the processor, and on the right kind of concurrency in the processor itself. These considerations are less critical in a low-performance machine, where many microcycles are used to execute each instruction, and the loss of a few is not disastrous. A high-performance machine, however, executes many instructions in one or two microcycles. Adding one or two more cycles because of a poorly chosen interface with the IFU, or because a very common pair of operations cannot be expressed in a single microinstruction, slows the emulator down by 50-200%. The common operations are not very complex, and require only a modest amount of hardware for an efficient implementation. The examples in this section illustrate these points.

Good performance depends on two things:

> An adequate set of data busses, so that it is physically possible to perform the frequent combinations of independent data transfers in a single cycle. We shall be mainly concerned with the busses which connect the IFU and the processor, rather than with the internal details of the latter. These are summarized in Figure 4.
>
> A microinstruction encoding which makes it possible to specify these transfers in a single microinstruction. A horizontal encoding does this automatically; a vertical one requires greater care to ensure that all the important combinations can still be specified.

We shall use the term *folding* for the combination of several independent operations in a single microinstruction. Usually folding is done by the microprogrammer, who surveys the operations to be done and the resources of the processor, and arranges the operations in the fewest possible number of microinstructions.

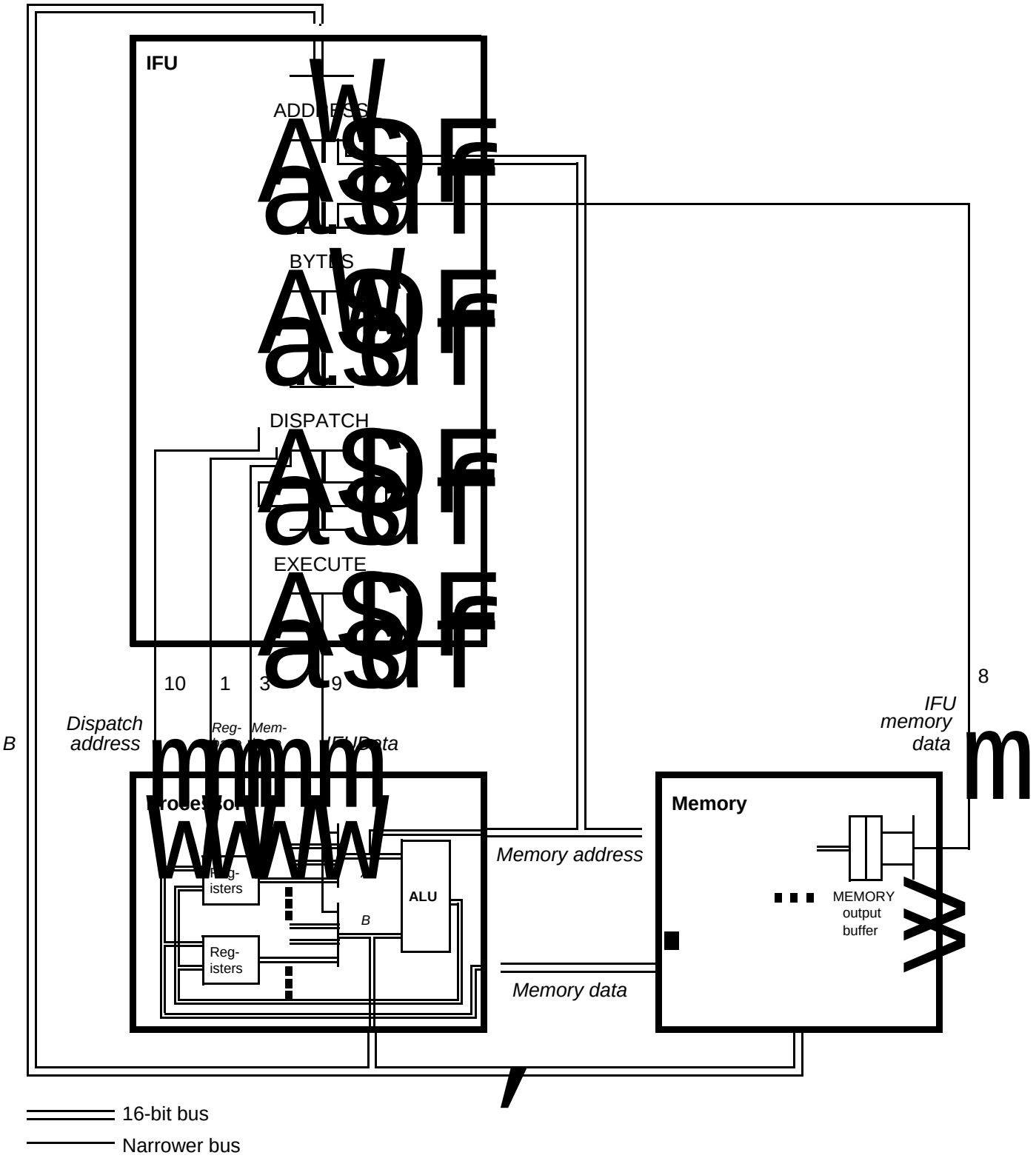


Figure 4: Busses between the IFU and the processor/memory

4.1 How the processor sees the IFU

The processor has four main operations for dealing with the IFU. Two are extremely frequent:

IFU*Jump*: The address of the next microinstruction is taken from the IFU; a ten bit bus passes the dispatch address to the processor's control section. In addition, parts of the processor state are initialized from the IFU, and other parts are initialized to standard values (see ¶ 4.2). IFU*Jump* causes the IFU to hand off an instruction to the processor if it has one ready. Otherwise the IFU dispatches the processor to the *NotReady* location. The microcode may issue another IFU*Jump* at that point, in which case the processor will loop at *NotReady* until the IFU has prepared the next instruction. An IFU*Jump* is coded in the branch control field of the microinstruction, and hence can be done concurrently with any data manipulation operation.

IFU*Data*: The IFU delivers the next field datum on the IFU*Data* bus, which is nine bits wide (eight data bits plus a sign). Successive IFU*Data*'s during emulation of an instruction produce a fixed sequence of values determined by the decoding table entry for the opcode, and chosen from:

a small constant *N* in the decoding table entry;
the alpha byte, possibly sign extended;
either half of the alpha byte;
the beta byte;
the instruction length.

IFU*Data* is usually delivered to the A bus, one of the processor's two main input busses, from which it can be sent through the ALU, or used as a displacement in a memory reference. In this case it is encoded in the microinstruction field which controls the contents of this bus, and hence can be done concurrently with all the other operations of the processor. IFU*Data* can also be delivered to B, the other main input bus, from which it can be shifted, stored, sent to the other ALU input, or output. This operation is encoded in the special function field, where it excludes a large number of relatively infrequent operations as well as immediate constants and long jumps, all of which also use this field. For the details of the processor and its microinstructions, see [8].

The other two IFU-related operations are less frequent, and are also coded in the special function field of the microinstruction:

PC: The IFU delivers the PC for the currently executing instruction to the B bus.

PC_: resets the IFU and supplies a new PC value from the B bus. The IFU immediately starts fetching instructions from the location addressed by the new PC.

In addition there are a number of operations that support initialization and testing of the hardware.

Strictly speaking, the IFU*Data* and PC operations do not interact with the IFU. All the information the IFU has about the instruction is handed off at the IFU*Jump*, including the field data and the PC (about 40 bits). However, these bits are physically stored with the IFU, and sent to the processor busses incrementally, in order to reduce the width of the busses needed (to 9 bits, plus a 16 bit bus multiplexed with many other functions). From the microprogrammer's viewpoint, therefore, the description we have given is natural.

We illustrate the use of these operations with some examples. First, here is the actual microcode for the *PushConstant* instruction introduced in ¶ 2.

```
PushConstantByte:
   Push[IFUData], IFUJump;                    -- Reduced from 9 microinstructions to 1!
```

To push a 16 bit constant, we need a three byte instruction; alpha contains the left eight bits of the constant and beta the right eight bits.

```
PushConstantWord:
   temp _ LeftShift[IFUData, 8];          -- put alpha into the left half of temp
   Push[temp or IFUData], IFUJump;        -- or in beta, push the result on the stack, and dispatch to the next instruction
```

Notice that the first microinstruction uses the IFU to acquire data from the code stream. Then the second microinstruction simultaneously retrieves the second data byte and dispatches to the next instruction. These examples illustrate several points.

> Any number of microinstructions can be executed to emulate an instruction, i.e., between IFU*Jump*s.
>
> Within an instruction, any number of IFU*Data* requests are possible; see Table 3 for a summary of the data delivered to successive requests.
>
> IFU*Jump* and IFU*Data* may be done concurrently. The IFU*Data* will reference the current instruction's data, and then the IFU*Jump* will dispatch the processor to the first microinstruction of the next instruction (or to *NotReady*).

Suppose analysis of programs indicates that the most common *PushConstant* instruction pushes the constant 0. Suppose further that 1 is the next most common constant, and 2 the next beyond that, and that all other constants occur much less frequently. A lot of code space can probably be saved by dedicating three one-byte opcodes to the most frequent *PushConstant* instructions, and using a two-byte instruction for the less frequent cases, as in the *PushConstantByte* example above, where the opcode byte designates a *PushConstantByte* opcode and alpha specifies the constant. A third opcode, *PushConstantWord*, provides for 16-bit constants, and still others are possible.

Pursuing this idea, we define five instructions to push constants onto the stack: *PushC0*, *PushC1*, *PushC2*, *PushCB*, *PushCW*. Any five distinct values can be assigned for the opcode bytes of these instructions, since the meaning of an opcode is completely defined by its decoding table entry. The entries for these instructions are as follows: (*N* is a constant encoded in the opcode, *Length* is the instruction length in bytes, and *Dispatch* is the microcode dispatch address; for details, see ¶ 5.4).

```
Opcode    Partial decoding table contents          -- Remarks

PushC0    Dispatch_PushC, N_0, Length_1            -- push 0 onto the stack
PushC1    Dispatch_PushC, N_1, Length_1            -- push 1 onto the stack
PushC2    Dispatch_PushC, N_2, Length_1            -- push 2 onto the stack
PushCB    Dispatch_PushC, Length_2                 -- push alpha onto the stack
PushCW    Dispatch_PushCWord, Length_3             -- push the concatenation of alpha and beta onto the stack
```

Here is the microcode to implement these instructions; we have seen it before:

```
PushC:                                     -- PushC0/1/2,  (ifuData=N), PushCB, (ifuData=alpha)
   Push[IFUData], IFUJump;

PushCWord:                                 -- PushCW,
   temp _ Lshift[IFUData, 8];              -- (IFUData=alpha here)
   Push[temp or IFUData], IFUJump;         -- (IFUData=beta here)
```

Observe that the same, single line of microcode (at the label *PushC*) implements four different opcodes, for both one and two byte instructions. Only *PushConstantWord* requires two separate microinstructions.

4.2 Initializing state

A standard method for reducing the size and increasing the usefulness of an instruction is to *parameterize* it. For example, we may consider an instruction with a base register field to be parameterized by that register: the "meaning" of the instruction depends on the contents of the register. Thus the same instruction can perform different functions, and also perhaps can get by with a smaller address field. This idea is also applicable to microcode, and is used in the Dorado. For example, there are 32 memory base registers. A microinstruction referencing memory does not specify one of these explicitly; instead, there is a *MemBase* register, loadable by the microcode, which tells which base register to use. Provided the choice of register changes infrequently, this is an economical scheme.

For emulation it presents some problems, however. Consider the microcode to push a local variable; the address of the variable is given by the alpha byte plus the contents of the base register *localData*, whose number is *localDataRegNo:*

```
PushLocalVar:
   MemBase _ localDataRegNo;          -- Make memory references relative to the local data.
   Fetch[IFUData];                    -- Use contents of PC+1 as offset.
   Push[memoryData], IFUJump;         -- Push variable onto stack, begin next instruction
```

This takes three cycles, one of which does nothing but initialize *MemBase*. The point is clear: such parametric state should be set from the IFU at the start of an instruction, using information in the decoding table. This is in fact done on the Dorado. The decoding table entry for *PushLocalVar* specifies *localData* as the initial value for *MemBase*, and the microcode becomes:

```
PushVar:
   Fetch[IFUData];                    -- IFU initializes MemBase to the local data
   Push[memoryData], IFUJump;         -- Push variable onto stack, begin next instruction
```

One microinstruction is saved. Furthermore, the *same* microcode can be used for a *PushGlobalVar* instruction, with a decoder entry which specifies the same dispatch address, but *globalData* as the initial value of *MemBase.* Thus there are two ways in which parameterization saves space over specifying everything in the microinstruction: each microinstruction can be shorter, and fewer are needed. The need for initialization, however, makes the idea somewhat less attractive, since it complicates both the IFU and the EU, and increases the size of the decoding table.

A major reduction in the size of the decoding table can be had by using the opcode itself as the dispatch address. This has a substantial cost in microcode, since typically the number of distinct dispatch addresses is about one-third of the 256 opcodes. If this price is paid and parameterization eliminated, however, the IFU can be considerably simplified, since not only the decoding table space is saved, but also the buffers and busses needed to hand off the parameters to the processor, and the parameterization mechanism in the processor itself. On the Dorado, the advantages of parameterization were judged to be worth the price, but the decision is a fairly close one. The current memory base register and the current group of processor registers are parameters of the microinstruction which are initialized from the IFU. The IFU also supplies the dispatch address at the same time. The remainder of the information in the decoding table describes the data fields and instruction length; it is buffered in EXECUTE and passed to the processor on demand.

4.3 Forwarding

Earlier we mentioned folding of independent operations into the same microinstruction as an important technique for speeding up a microprogram. Often, however, we would like to fold the emulation of two successive instructions, deferring some of the work required to finish emulation of one instruction into the execution of its successor, where we hope for unused resources. This cannot be done in the usual way, since we have no *a priori* information about what instruction comes next. However, there is a simple trick (due to Ed Fiala) which makes it possible in many common cases.

We define for an entire instruction set a small number *n* of *cleanup actions* which may be *forwarded* to the next instruction for completion; on the Dorado up to four are possible, but one must usually be the null action. For *each* dispatch address we had before, we now define *n* separate ones, one for *each* cleanup action. Thus if there were *D* addresses to which an IFU*Jump* might dispatch, there are now *nD*. At each one, there must be microcode to do the proper cleanup action in addition to the work required to emulate the current instruction. The choice of cleanup action is specified by the microcode for the previous instruction; to make this convenient, the Dorado actually has four kinds of IFU*Jump* operations (written IFU*Jump*[*i*] for *i*=0, 1, 2, 3), instead of the one described above. The two bits thus supplied are ORed with the dispatch address supplied by the IFU to determine the microinstruction to which control should go. To avoid any assumptions about which pairs of successive instructions can occur, all instructions in the same instruction set must use the same cleanup actions and must be prepared to handle all the cleanup actions. In spite of this limitation, measurements show that forwarding saves about 8% of the execution time in straight-line code (see ¶ 6.4); since the cost is very small, this is a bargain.

We illustrate this feature by modifying the implementation of *PushLocalVar* given above, to show how one instruction's memory fetch operation can be finished by its successor, reducing the cost of a *PushLocalVar* from two microinstructions to one. We use two cleanup actions. One is null (action 0), but the other (action 2) finds the top of the stack not on the hardware stack but in the *memoryData* register. Thus, any instruction can leave the top of stack in *memoryData* and do an IFU*Jump*[2]. Now the microcode looks like this:

```
PushLocalVar[0]:
    Fetch[IFUData], IFUJump[2];          -- this entry point assumes normal stack, and leaves top of stack in
                                            memoryData.

PushLocalVar[2]:
    Push[memoryData], Fetch[IFUData], IFUJump[2]; -- this entry point assumes top of stack is in memoryData and leaves it there.
```

In both cases, the microcode executes IFU*Jump*[2], since the top of stack is left in the *memoryData* register, rather than on the stack as it should be. In the case of *PushLocalVar*[2], the previous instruction has done the same thing. Thus, the microcode at this entry point must move that data into the stack at the same time it makes the memory reference for the next stack value. The reader can see that successive *Push* instructions will do the right thing. Of course there is a payoff only because the first microinstruction of *PushLocalVar*[0] is not using all the resources of the processor.

It is instructive to look at the code for *Add* with this forwarding convention:

```
Add[0]:
    temp _ Pop[];                        -- this entry point assumes  and leaves normal stack
    StackTop _ StackTop+temp, IFUJump[0];

Add[2]:
    StackTop _ StackTop+memoryData, IFUJump[0]; -- this entry point assumes top of stack is in memoryData, leaves normal
                                                   stack.
```

This example shows that the folding enabled by forwarding can actually eliminate data transfers which are necessary in the unfolded code. At *Add*[2] the second operand of the *Add* is not put on the stack and then taken off again, but is sent directly to the adder. The common data bus of the 360/91 [15] obtains similar, but more sweeping, effects at considerably greater cost. It is also possible to do a cleanup after a *NotReady* dispatch; this allows some useful work to be done in an otherwise wasted cycle.

4.4 Conditional branches

We conclude our discussion of IFU-processor interactions, and give another example of forwarding, with the example of a conditional branch instruction. Suppose that there is a *BranchNotZero* instruction that takes the branch if the current top of the stack is not zero. Assume that its decoding table entry tells the IFU to follow the branch, and specifies the instruction length as the first IFU*Data* value. Straightforward microcode for the instruction is:

```
BranchNotZero:                           -- IFU jumps come here.  IFU assumed result#0.
    if stack=0 then goto InsFromIFUData, Pop;  -- Test result in this microinstruction.
    IFUJump;                             -- Result was non-zero, IFU did right thing.

InsFromIFUData:                          -- Result was zero. Do the instruction at PC+IFUData.
    temp _PC+IFUData;                    -- PC should be PC+Instruction length.
    PC _ temp;                           -- Redirect the IFU
    IFUJump;                             -- This will be dispatched to NotReady, where the code will loop until the IFU
                                            refills starting at the new location.
```

The most likely case (the top of the stack non-zero) simply makes the test specified by the instruction and does an IFU*Jump* (two cycles). If the value is zero (the IFU took the wrong path), the microcode computes the correct value for the new PC and redirects the IFU accordingly (four cycles, plus the IFU's latency of five cycles; guessing wrong is painful). If we think that *BranchNotZero* will usually fail to take the branch, we can program the decoding table to treat it as

an ordinary instruction and deliver the branch displacement as IFU*Data*, and reverse the sense of the test.

A slight modification of the forwarding trick allows further improvement. We introduce a cleanup action (say action 1) to do the job of *InsFrom*IFU*Data* above (it must be action 1 or 3, since a successful test in the Dorado **or**s a 1 into the next microinstruction address). Now we write the microcode (including for completeness the action 2 of ¶ 4.3):

```
BranchNotZero[0]:                    -- IFU jumps come here.  Expect result#0.
    Test[stack=0], Pop, IFUJump[0];  -- Test result in this microinstruction; if the test succeeds, we do IFUJump[1].
BranchNotZero[2]:
    Test[memoryData=0], IFUJump[0];

EveryInstruction[1]:                 -- Branch was wrong. Do the instruction at PC+IFUData.
    temp _PC+IFUData;
    PC _ temp;                       -- Redirect the IFU
    IFUJump[0];                      -- This will be dispatched to NotReady, where the code will loop until the IFU
                                        refills starting at the new location.
```

Now a branch which was predicted correctly takes only one microinstruction. For this to work, the processor must keep the IFU from advancing to the next instruction if there is a successful test in the IFU*Jump* cycle. Otherwise, the PC and IFU*Data* of the branch instruction would be lost, and the cleanup action could not do its job. Note that the first line at *EveryInstruction*[1] must be repeated for each distinct dispatch address; all these can jump to a common second line, however.

5. Implementation

In this section we describe the implementation of the Dorado IFU in some detail. The primary focus of attention is the pipeline structure, discussed within the framework established in ¶ 2 and ¶ 3.3, but in addition we give (in ¶ 5.4) the format of the decoding table, which defines how the IFU can be specialized to the needs of a particular instruction set. Figure 3 gives the big picture of the pipeline. Table 1 summarizes the characteristics of each stage; succeeding subsections discuss each row of the table in turn. The first row gives the properties of an ideal stage, and the rest of the table describes departures from this ideal. This information is expanded in the remainder of this section; the reader may wish to use the table to compare the behavior of the different stages.

The entire pipe is synchronous, running on a two-phase clock which defines a 60 ns cycle; some parts of the pipe use both phases and hence are clocked every 30 ns. An "ideal" stage is described by the first line of the table. There is a buffer following each stage which can hold one item (b=1), and may be *empty* (represented by an *empty* flag); this is also the input buffer for the next stage. The stage takes an item from its input buffer every cycle (t_{input}=1) and delivers an item to its output buffer every cycle (t_{output}=1); the item taken is the one delivered (l=1). The buffer is loaded on the clock edge which defines the end of one cycle and the start of the next. The stage handles an item if and only if there is space in the output buffer for the output at the end of the cycle; hence if the entire pipe is full and an item is taken by the processor, every stage will process an item *in that cycle*. This means that information about available buffer space must propagate all the way through the pipe in one cycle. Furthermore, this propagation cannot start until it is known that the processor is accepting the item, and it must take account of the various irregularities which allow a stage to accept an item without delivering one or vice versa. Thus, the pipe has *global* control. Note that a stage delivers an output item whether or not its input buffer is empty; if it is, the special *empty* item is delivered. Thus the space bookkeeping is done entirely by counting *empty* items.

Implementing global control within the available time turned out to be hard. It was considered crucial because of the minimal buffering between stages. The alternative, much easier approach is *local* control: deliver an item to the buffer only if there is space for it there at the *start* of the cycle. This decouples the different stages completely within a cycle, but it means that if the pipe is full (best case) and the processor suddenly starts to demand one instruction per cycle (worst case), the pipe can only deliver at half this rate, even though each stage is capable of running at the full rate;

Stage	*Size*	*Input*	*Output*	*Reset*	*Remarks*
"ideal"		*t*=1; takes one item if output is possible	*t*=*l*=1; delivers one item if buffer *will be* empty; *b*=1	Clears buffer to empty on PC_ . . .	All state is in the buffer after the stage.
ADDRESS	word	No input	Not if paused, MAR contention, or mem busy; OK if space in *any* later buffer.	and jump; also accepts new PCvalue	Pass PC by incrementing; a source, hence has state (PC).
MEMORY	word	Internal complications	*l*>2; output is unconditional; *b*=2	and jump; discards output of fetches in progress	Must enforce FIFO; not really part of IFU; has state of 0-2 fetches in progress
BYTES	byte	*t*=.5	*t*=*l*=.5	and jump	Break byte feature.
DECODE	instr	*t*>.5; rate depends on instruction length		only	Recycling to vary rate; splits beta byte; encodes exceptions; does jumps.
DISPATCH	instr	On IFU*Jump*		only	*NotReady* is default delay; IFUHold is panic delay.
EXECUTE	byte	On IFU*Data*	No output buffer	Reset unnecessary	

Table 1: Summary of the pipeline stages

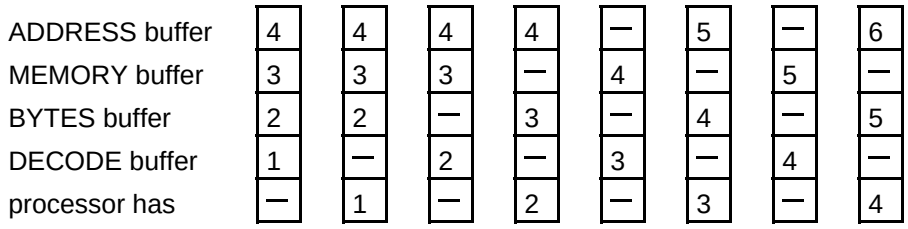


Figure 5a: Cogging with local control and one item buffering

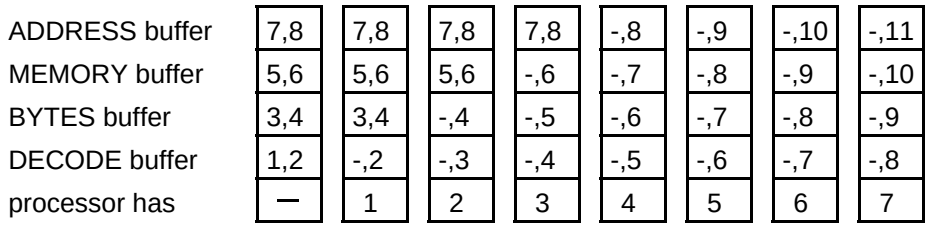


Figure 5b: Smooth operation with local control and two item buffering

Figure 5*a* illustrates this *cogging*. Figure 5*b* shows that with two items of buffering after each stage, local control does not cause cogging. The Dorado has small buffers and global control partly because buffers are fairly costly in components (see below), and partly because this issue was not fully understood during the design. Note that it is easy to implement global control over a group of consecutive stages which have no irregularities, since *every* stage can safely advance if there is room in the buffer of the *last* stage. In this IFU, alas, there are no two consecutive regular stages.

Unfortunately, the cost of buffering is not linear in the number of items. A two item buffer costs more than three times as much as a one item buffer; this is because the latter is simply a register, while the former requires two registers plus a multiplexor to *bypass* the second register when the buffer is empty, as shown in Figure 6. Without the bypass a larger buffer increases the latency of the pipe, which is highly undesirable since it slows down every jump which the IFU doesn't predict successfully. Once the cost of bypassing is paid, however, a multi-item buffer costs only a little more, since a RAM can be used in place of the second register. Although there are no such buffers in the Dorado, it is interesting to see how they are made.

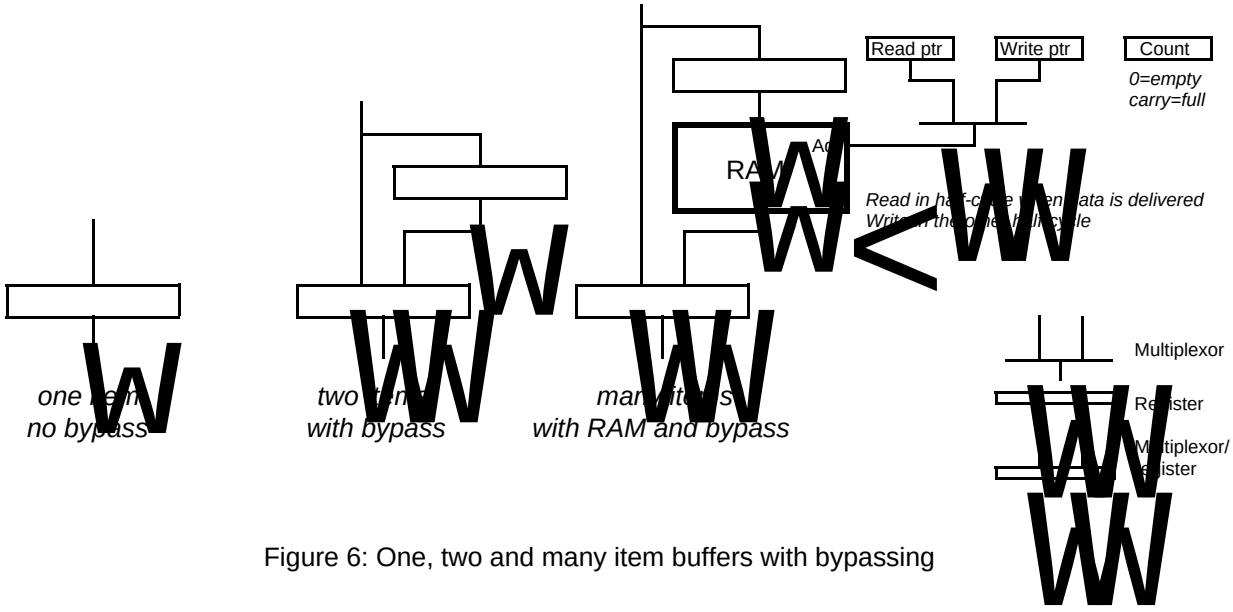


Figure 6: One, two and many item buffers with bypassing

The RAM requires two counters to act as read and write pointers, and a third to keep track of the number of items in the buffer. In addition, it must be effectively two-ported, since in a single cycle it is necessary to write one item and read an earlier one. In the Dorado two-port RAMs are used in many places; since no such part is available, they are implemented by running an ordinary RAM at twice the machine cycle (both 16x4 and 256x4 RAMs are available which can be read or written in 10 ns), and using a multiplexor to supply the read address in one half-cycle and the write address in the other. Figure 6 shows this arrangement in a slightly simplified form.

A normal stage has no state which changes as instructions are executed; all the state is represented in the items as they are stored in the inter-stage buffers. As a consequence, resetting the pipe is done simply by filling all the buffers with *empty* items.

Every item carries with it a PC, which is the address in the code from which its first byte was fetched. It is the IFU's handling of jumps which makes this necessary; otherwise it would suffice to remember the initial PC at the end of the pipe, and to increment it by the instruction length as each instruction goes by. Since no jumps can be executed between the ADDRESS and BYTES states, this method is in fact used there. It is especially convenient because BYTES handles one byte at a time, so that the PC can be held in a counter which is incremented once per item; later in the pipe an

adder would be needed to handle the variable instruction lengths, and it would cost about four times as much.

Every item also carries a *status* field, which is used to represent various values that do not correspond to ordinary instructions: empty, page fault, memory error. These are converted into unique dispatch addresses when the item is passed to the processor, as discussed in ¶ 3.4.

5.1. ADDRESS stage

This stage generates the addresses of memory words which contain the successive bytes of code. Unlike the other stages, it has no ordinary input, but instead contains a PC which it increments by two (there are two bytes per memory word) for each successive reference. The PC can also take on a *pause* value which prevents any further memory references until the processor resupplies ADDRESS with an ordinary PC value. This *pause* state plays the same role for ADDRESS that an empty input buffer plays for the other stages; hence it is entered whenever this stage is reset. That happens either because of a processor *Reset* operation (which resets the entire IFU pipe, and is not done during normal execution), or because of a *Pause* signal from DECODE. Correspondingly, a new PC can be supplied either by a processor PC_ operation, or by a *Jump* signal from DECODE when it sees a jump instruction. Any of these operations resets the pipe between ADDRESS and DECODE; the processor operations reset the later stages also.

ADDRESS makes a memory reference if the memory is willing to accept the reference; this corresponds to finding space in the buffer between ADDRESS and MEMORY, although the implementation is quite different because the memory is not physically part of the IFU. In addition, 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. Finally, it is necessary to worry about space for the resulting memory word: the memory, unlike ordinary IFU stages, delivers its result unconditionally, and hence must not be started unless there is a place to put the result. ADDRESS surveys the buffering in the rest of the pipe, and waits until there are at least two free bytes guaranteed; it isn't necessary for these bytes to be in the MEMORY output buffer, since data in that buffer will advance into later buffers before the memory delivers the data. It is, however, necessary to make the most pessimistic assumptions about instruction length and processor demands. On this basis, there are seven bytes of buffering altogether: four after MEMORY, two after BYTES, and one after DECODE.

5.2 MEMORY stage

This stage has several peculiarities. Some arise from the fact that most of it is not logically or physically a part of the IFU, but instead is shared with the processor and I/O system. As we saw in the previous section, the memory delivers results unconditionally, rather than waiting for buffer space to be available; ADDRESS allows for this in starting MEMORY. Furthermore, the memory has considerable internal state and cannot be reset, so additional logic is required to discard items which are inside the memory when the stage is reset.

Other problems arise from the fact that the memory's latency is more than one cycle; in fact, it ranges from two to about 30 cycles (the latter when there is a cache miss). To maintain full bandwidth, the IFU must therefore have more than one item in the MEMORY stage at a time; since l=2 when the cache hits, and this is the normal case, there is provision for up to two items in MEMORY. A basic principle of pipeline stages is that items emerge in the order they are supplied. A stage with fixed latency, or one which holds only one item, does this automatically, but MEMORY has neither of these properties. Furthermore, its basic function is random access, with no sequential relationship between successive references. Hence if one reference misses and the next one hits, the memory is happy to deliver the second result first. To prevent this from happening, the IFU notifies the memory that it has a reference outstanding when it makes the second one, and the memory rejects the second reference unless the first one is about to complete.

The irregularity of the memory also demands more than one word of buffering for its output, and in fact two are provided. They are physically packaged with the cache data memory, as is the BYTES stage multiplexing required to produce individual bytes. As a result, a one-byte bus suffices to deliver memory data to the IFU.

5.3 BYTES stage

This is a very simple stage, which consists only of the multiplexors just mentioned. It does, however, run twice as fast as the other stages, so that it can deliver two-byte instructions at the full rate of one per cycle. This means that the multiplexors must look at both words of the MEMORY output buffer, which runs only at the normal rate.

BYTES also includes a provision for replacing the first byte coming from memory with a byte taken from a *substitute* register within the stage. This feature makes it convenient to proceed after a breakpoint without removing the one-byte breakpoint instruction from the code; instead the opcode byte displaced by the breakpoint is loaded into the substitute register (by the microcode) and substituted for the break instruction. Since the substitution is done only once, the break is executed normally when control returns to it. The substitute register is also a convenient way to address the decoding table for loading and testing it.

5.4 DECODE stage

The main complications in this stage 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 1kx1 RAMs, which provide room for four instruction sets with 256 opcodes each. It takes about two-thirds of a cycle to read these RAMs, with consequences which are described below. The form of an entry is outlined in Table 2; parity is also stored. Most of this information is passed on directly to the DECODE buffer. The last three fields, however, affect the IFU's handling of subsequent instructions.

Name	*Size*	*Function*
Dispatch	10	The starting microcode address for the instruction
MemBase	3	Selects one of eight memory base registers.
RBase	1	Selects one of two processor register groups.
SplitAlpha	1	Split the first data byte into two four-bit data items.
N	4	Encoded constant.
Sign	1	Extend sign of the first datum provided to the processor.
Length	2	The length of the instruction; also supplied as a datum.
Jump	1	Indicates a jump; DECODE computes a new PC from PC plus N (if *length*=1) or alpha (if *length*=2).
Pause	1	Indicates that ADDRESS should be reset.

Table 2: Fields of a decoding table entry.

The instruction length determines the treament of both this and later instructions; the fact that it isn't known until late in the DECODE cycle causes serious problems. A simple implementation of DECODE addresses the decoding table directly from the input buffer. If the instruction turns out to be one byte long, it is delivered to the output buffer in the normal way. If it is longer, the decoded output is latched and additional bytes are taken from BYTES until the complete instruction is in DECODE ready to be delivered; see Figure 7*a*. Unfortunately, the length must be known before the middle of the cycle to handle two-byte instructions at full speed. Figure 7*b* shows how this

problem can be attacked by introducing a sub-stage within DECODE; unfortunately, this delays the reading of the decode table by half a cycle, so that its output is not available together with the alpha byte. To solve the problem it is necessary to provide a second output buffer for BYTES, and to feed back its contents into the main buffer if the instruction turns out to be only one byte long, as in Figure 7*c*. Some care must be taken to keep the PCs straight. This ugly backward dependency seems to be an unavoidable consequence of the variable-width items.

In fact, a three-byte instruction is not handled exactly as shown in Figure 7. Since the bandwidth of BYTES prevents it from being done in one cycle anyway, space is saved by breaking it into two sub-instructions, each two bytes long; for this purpose a dummy opcode byte is supplied between alpha and beta. Each sub-instruction is treated as an instruction item. The second one contains beta and is slightly special: DECODE ignores its dummy opcode byte and treats it as a two-byte instruction, and DISPATCH passes it on to EXECUTE after the alpha byte has been delivered.

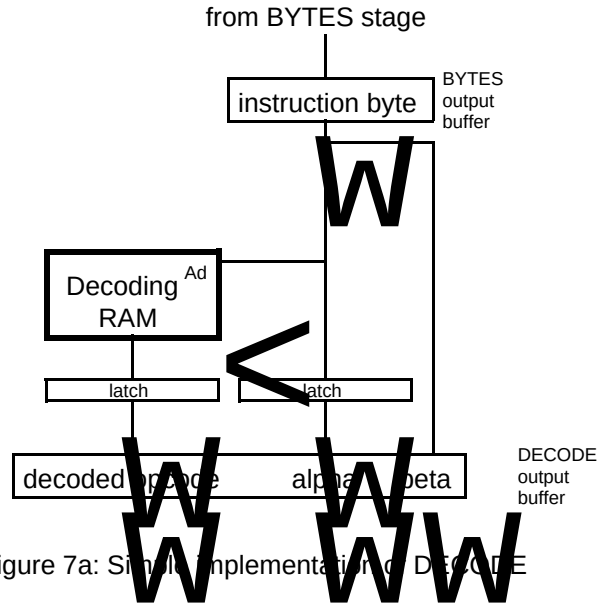


Figure 7a: Simple implementation of DECODE

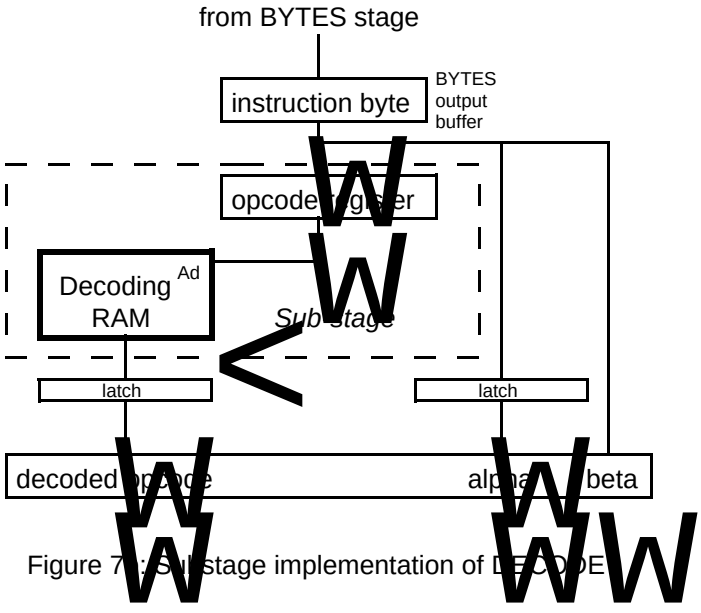


Figure 7b: Sub-stage implementation of DECODE

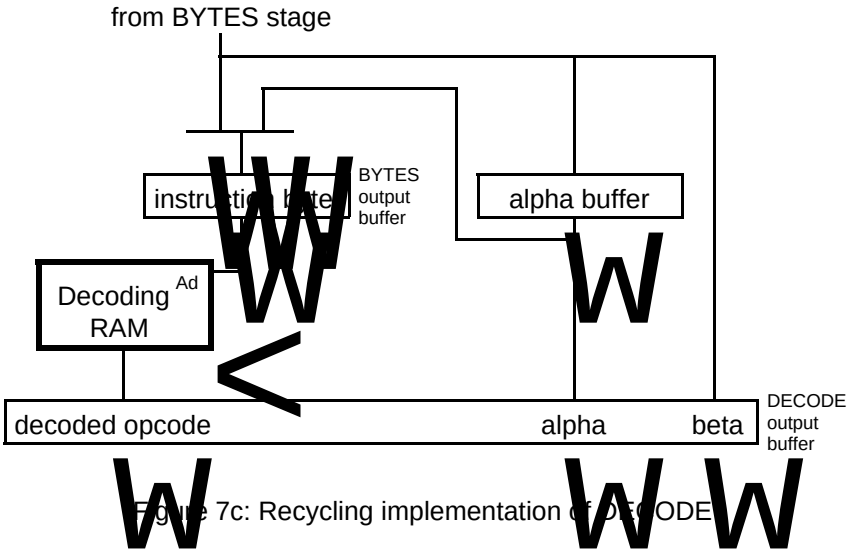


Figure 7c: Recycling implementation of DECODE

DECODE replaces the dispatch address from the table with an exception address if necessary. In order to obey the rule that exceptions must all be captured in the dispatch address, the exception values of all the instruction bytes are merged into its computation. For three-byte instructions, this requires looking back into BYTES for the state of the beta byte. If any of the bytes is *empty*, DECODE keeps the partial instruction item when it delivers an *empty* item with a *NotReady* dispatch into its output buffer. If a *Reschedule* is pending, it is treated like any other exception, by converting the dispatch address of the next instruction item into *Reschedule*. Thus there is always a meaningful PC associated with the exception.

If the *Jump* field is set, DECODE computes a new program counter by adding an offset to the PC of the instruction. This offset comes from the alpha byte if there is one, otherwise from *N* and *SplitAlpha*; it is sign-extended if *Sign* is true. The new PC is sent back to ADDRESS, as described in ¶ 5.1, where *Pause* is also explained. 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.

5.5 DISPATCH stage

The interesting work of this stage is done by the processor, which takes the dispatch address, together with the state initialization discussed in ¶ 4.2, from the DECODE output buffer when it executes an IFU*Jump*. Because *empty* is encoded into a *NotReady* dispatch, the processor takes no account of whether the buffer is empty. There are some ugly cases, however, in which DECODE is unable to encode an exception quickly enough. In these cases DISPATCH asserts a signal called *Hold* which causes the processor to skip an instruction cycle; this mechanism is rather expensive to implement, and is present only because it was essential for synchronization between the processor and the memory [1]. Once implemented, however, it is quite cheap for the IFU to use. The *NotReady* dispatch is still preferable, because it gives the microcode an opportunity to do some useful work while waiting.

5.6. EXECUTE stage

This stage implements the IFU*Data* function; as we have already seen, it is logically part of the processor. The sequence of data items delivered in response to IFU*Data* is controlled by *Jump*, *Length*, *N*, and *SplitAlpha* according to Table 3; in addition, *alpha* is sign-extended if *Sign* is true. EXECUTE also provides the processor with the value of the PC in response to a different function.

Jump	*Length*	*N*	*SplitAlpha*	IFU*Data*
Yes				*Length*, . . .
No	1	No		*Length*, . . .
No	1	Yes		*N, Length*, . . .
No	2	No	No	*alpha, Length*, . . .
No	2	No	Yes	*alphaHigh, alphaLow, Length*, . . .
No	2	Yes	No	*N, alpha, Length*, . . .
No	2	Yes	Yes	*N, alphaHigh, alphaLow, Length*, . . .
No	3	No	No	*alpha, beta, Length*, . . .
No	3	No	Yes	*alphaHigh, alphaLow, beta, Length*, . . .
No	3	Yes	No	*N, alpha, beta, Length*, . . .
No	3	Yes	Yes	*N, alphaHigh, alphaLow, beta, Length*, . . .

Table 3: Data items provided to IFU*Data*

6. Performance

The value of an instruction fetch unit depends on the fraction of total emulation time that it saves (over doing instruction fetching entirely in microcode). This in turn clearly depends on the amount of time spent in executng each instruction. For a language like Smalltalk-76 [5], a typical instruction requires 30-40 cycles for emulation, so that the half-dozen cycles saved by the IFU are not very significant. At the other extreme, an implementation language like Mesa [9, 11] is compiled into instructions which can often be executed in a single cycle; except for function calls and block transfers, no Mesa instruction requires more than half a dozen cycles. For this reason, we give performance data only for the Mesa emulator.

The measurements reported were made on the execution of the Mesa compiler, translating a program of moderate size; data from a variety of other programs is very similar. All the operating system functions provided in this single-user system are included. Disk wait time is excluded, since it would tend to bias the statistics. Some adjustments to the raw data have been made to remove artifacts caused by compatibility with an old Mesa instruction set. Time spent in the procedure call and return instructions (about 15%) has been excluded; these instructions take about 10 times as long to execute as ordinary instructions, and hence put very little demand on the IFU.

The Dorado has a pair of counters which can record events at any rate up to one per machine cycle. Together with supporting microcode, these counters provide sufficient precision that overflow requires days of execution. It is possible to count a variety of interesting events; some are permanently connected, and others can be accessed through a set of multiplexors which provide access to several thousand signals in the machine, independently of normal microprogram execution.

6.1 Performance limits

The maximum performance that the IFU can deliver is limited by certain aspects of its implementation; these limitations are intrinsic, and do not depend on the microcode of the emulator or on the program being executed. The consequences of a particular limitation, of course, depend on how frequently it is encountered in actual execution.

Latency: after the microcode supplies the IFU with a new PC value, an IFU*Jump* will go to *NotReady* until the fifth following cycle (in a few cases, until the sixth cycle). Thus there are at least five cycles of latency before the first microinstruction of the new instruction can be executed. Of course, it may be possible to do useful work in these cycles. This latency is quite important, since every instruction for which the IFU cannot compute the next PC will pay it; these are wrongly guessed conditional branches, indexed branches, subroutine calls and returns, and a few others of negligible importance.

A branch correctly executed by the IFU causes a three-cycle gap in the pipeline. Hence if the processor spends one cycle executing it and each of its two predecessors, it will see three *NotReady* cycles on the next IFU*Jump*. Additional time spent in any of these three instructions, however, will reduce this latency, so it is much less important than the other.

Bandwidth: In addition to these minimum latencies, the IFU is also limited in its maximum throughput by memory bandwidth and its limited buffering. A stream of one-byte instructions can be handled at one per cycle, even with some processor references to memory. A stream of two-byte instructions, however (which would consume all the memory bandwidth if handled at full speed), results in 33% *NotReady* even if the processor makes no memory references. The reason is that the IFU cannot make a reference in every cycle, because its buffering is insufficient to absorb irregularity in the processor's demand for instructions. As we shall see, these limitations are of small practical importance.

6.2 NotReady dispatches

Our measurements show that the average instruction takes 3.1 cycles to execute (including all IFU delays). Jumps are 26% of all instructions, and incorrectly predicted jumps (40% of all conditional jumps) are 10%. The average non-jump instruction takes 2.5 cycles.

The performance of the IFU must be judged primarily on the frequency with which it fails to satisfy the processor's demand for an instruction, i.e., the frequency of *NotReady* dispatches. It is instructive to separate these by their causes:

latency,

cache misses by the IFU,

dearth of memory bandwidth,

insufficient buffering in the IFU.

The first dominates with 16% of all cycles, which is not surprising in view of the large number of incorrectly predicted jumps. Note that since these *NotReady* cycles are predictable, unlike all the others, they can be used to do any background tasks which may be around.

Although the IFU's hit rate is 99.7%, the 25 cycle cost of a miss means that 2.5% of all cycles are *NotReady* dispatches from this cause. This is computed as follows: one cycle in three is a dispatch, and .3% of these must wait for a miss to complete. The average wait is near the maximum, unfortunately, since most misses are caused by resetting the IFU's PC, This yields 33% of .3%, or .1%, times 25, or 2.5%.

The other causes of *NotReady* account for only 1%. This is also predictable, since more than half the instructions are one byte, and the average instruction makes only one memory reference in three cycles. Thus the average memory bandwidth available to the IFU is two words, or three instructions, per instruction processed, or about three times what is needed. Furthermore, straight-line instructions are demanded at less than half the peak rate on the average, and jumps are so frequent that when the first instruction after a jump is dispatched, the pipe usually contains half the instructions that will be executed before the next jump.

6.3 Memory bandwidth

As we have seen, there is no shortage of memory bandwidth, in spite of the narrow data path between the processor and the IFU. Measurements show that the processor obtains a word from the memory in 16% of the cycles, and the IFU obtains a word in 32% of the cycles. Thus data is supplied by the memory in about half the cycles. The processor actually shuts out the IFU by making its own reference about 20% of the time, since some of its references are rejected by the memory and must be retried. The IFU makes a reference for each word transferred, and makes unsuccessful references during its misses, for a total of 35%. There is no memory reference about 45% of the time.

6.4 Forwarding

The forwarding trick saves a cycle in about 25% of the straight-line instructions, and hence speeds up straight-line execution by 8%. Jumps take longer and benefit less, so the speed-up within a procedure is 5%. Like the IFU itself, forwarding pays off only when instructions are executed very quickly, since it can save at most one cycle per instruction.

6.5 Size

A Dorado board can hold 288 standard 16-pin chips. The IFU occupies about 85% of a board; these 240 chips are devoted to the various stages as shown in Table 4.

Function	*Chips*	*%*
ADDRESS-BYTES	40	17
DECODE	86	35
DISPATCH	24	10
EXECUTE	18	8
Processor interface	27	11
Clocks	18	8
Testing	27	11

Table 4: Size of various parts of the IFU

In addition, about 25 chips on another board are part of MEMORY and BYTES. The early stages are mostly devoted to handling several PC values. DECODE is large because of the decoding table (27 RAM chips) and its address drivers and data registers, as well as the branch address calculation.

Table 5 shows the amount of microcode in the various emulators, and in some functions common to all of them. In addition, each emulator uses one quarter of the decode table. Of course they are not all resident at once.

System	*Words*	*Comments*
Mesa	1300	
Smalltalk	1150	
Lisp	1500	
Alto BCPL	700	
I/O	1000	Disk, keyboard, regular and color display, Ethernet
Floating point	300	IEEE standard; there is no special hardware support
Bit block transfer	270	

Table 5: Size of various emulators

Acknowledgements

The preliminary design of the Dorado IFU was done by Tom Chang, Butler Lampson and Chuck Thacker. Final design and checkout were done by Will Crowther and the authors. Ed Fiala reviewed the design, did the microassembler and debugger software, and wrote the manual. The emulators mentioned were written by Peter Deutsch, Willie-Sue Haugeland, Nori Suzuki and Ed Taft.

References

1. Clark, D.W. *et. al*. The memory system of a high performance personal computer. Technical Report CSL-81-1, Xerox Palo Alto Research Center, January 1981. Revised version to appear in *IEEE Transactions on Computers*.
2. Connors, W.D. *et. al*. The IBM 3033: An inside look. *Datamation*, May 1979, 198-218.
3. Deutsch, L.P. A Lisp machine with very compact programs. *Proc 3rd Int. Joint Conf. Artificial Intelligence*, Stanford, 1973, 687-703.
4. Ibbett, R.N. and Capon, P.C. The development of the MU5 computer system. *Comm. ACM* **21**, 1, Jan. 1978, 13-24.
5. Ingalls, D.H. The Smalltalk-76 programming system: Design and implementation. *5th ACM Symp. Principles of Programming Languages*, Tucson, Jan. 1978, 9-16.
6. Intel Corp. *MCS-86 User's Manual*, Feb. 1979.
7. Knuth, D.E. An empirical study of Fortran programs. *Software Practice and Experience* **1**, 1971, 105-133.
8. 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.
9. Mitchell, J.G. *et. al*. *Mesa Language Manual*. Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.
10. Russell, R.M. The CRAY-1 computer system. *Comm. ACM* **21**, 1, Jan. 1978, 63-72.
11. Tanenbaum, A.S. Implications of structured programming for machine architecture. *Comm. ACM* **21**, 3, March 1978, 237-246.
12. Teitelman, W. *Interlisp Reference Manual*. Xerox Palo Alto Research Center, Oct. 1978.
13. 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.
14. Thornton, J.E. *The Control Data 6600*, Scott, Foresman & Co., New York, 1970.
15. Tomasulo, R.M. An efficient algorithm for exploiting multiple arithmetic units, *IBM J. R&D* **11**, 1, Jan. 1967, 25-33.
16. Anderson, D.W. *et. al*. The System/360 Model 91: Machine philosophy and instruction handling. *IBM J. R&D* **11**, 8, Jan. 1967, 8-24.
17. Widdoes, L. C. The S-1 project: Developing high performance digital computers. *Proc. IEEE Compcon*, San Francisco, Feb. 1980, 282-291.

The Memory System of a High-Performance Personal Computer

by Douglas W. Clark[1], Butler W. Lampson, and Kenneth A. Pier

January 1981

ABSTRACT

The memory system of the Dorado, a compact high-performance personal computer, has very high I/O bandwidth, a large paged virtual memory, a cache, and heavily pipelined control; this paper discusses all of these in detail. Relatively low-speed I/O devices transfer single words to or from the cache; fast devices, such as a color video display, transfer directly to or from main storage while the processor uses the cache. Virtual addresses are used in the cache and for all I/O transfers. The memory is controlled by a seven-stage pipeline, which can deliver a peak main-storage bandwidth of 530 million bits per second to service fast I/O devices and cache misses. Interesting problems of synchronization and scheduling in this pipeline are discussed. The paper concludes with some performance measurements that show, among other things, that the cache hit rate is over 99 percent.

A revised version of this paper will appear in *IEEE Transactions on Computers*.

CR CATEGORIES

6.34, 6.21.

KEY WORDS AND PHRASES

bandwidth, cache, latency, memory, pipeline, scheduling, storage, synchronization, virtual memory.

1. Present address: Digital Equipment Corporation, Tewksbury, Mass. 01876.

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

1. Introduction

This paper describes the memory system of the Dorado, a high-performance compact personal computer. This section explains the design goals for the Dorado, sketches its overall architecture, and describes the organization of the memory system. Later sections discuss in detail the cache (¶ 2), the main storage (¶ 3), interactions between the two (¶ 4), and synchronization of the various parallel activities in the system (¶ 5). The paper concludes with a description of the physical implementation (¶ 6), and some performance measurements (¶ 7).

1.1 Goals

A high-performance successor to the Alto computer [18], the Dorado is intended to provide the hardware base for the next generation of computer system research at the Xerox Palo Alto Research Center. The Dorado is 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 is 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 had a major effect on the design.

The Dorado is designed to rapidly execute programs compiled into a stream of *byte codes* [16]; the microcode that does this is called an *emulator*. Byte code compilers and emulators exist for Mesa [6, 13], Interlisp [4, 17], and Smalltalk [7]. An instruction fetch unit (IFU) in the Dorado fetches bytes from such a stream, decodes them as instructions and operands, and provides the necessary control and data information to the emulator microcode in the processor; it is described in another paper [9]. Further support for fast execution comes from a very fast microcycle, and a microinstruction set powerful enough to allow interpretation of a simple byte code in a single microcycle; these are described in a paper on the Dorado processor [10]. There is also a cache [2, 11] which has a latency of two cycles, and which can deliver a 16-bit word every cycle.

Another major goal is to support high-bandwidth input/output. In particular, color monitors, raster scanned printers, and high speed communications are all part of the computer research activities; these devices typically have bandwidths of 20 to 400 million bits per second. Fast devices must not excessively degrade program execution, even though the two functions compete for many of the same resources. Relatively slow devices, such as a keyboard or an Ethernet interface [12], must also be supported cheaply, without tying up the high-bandwidth I/O system. These considerations clearly suggest that I/O activity and program execution should proceed in parallel as much as possible. The memory system therefore allows parallel execution of cache accesses and main storage references. Its pipeline is *fully segmented*: a cache reference can start in every microinstruction cycle, and a main storage reference can start in every main storage cycle.

1.2 Gross structure of the Dorado

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, which in turn contains a cache, a hardware virtual-to-real address map, and main storage. Both the processor and the IFU can make memory references and transfer data to and from the memory through the cache. Slow, or low-bandwidth I/O devices communicate with the processor, which in turn transfers their data to and from the cache. Fast, or high-bandwidth devices communicate directly with storage, bypassing the cache most of the time.

For the most part, data is handled sixteen bits 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. 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).

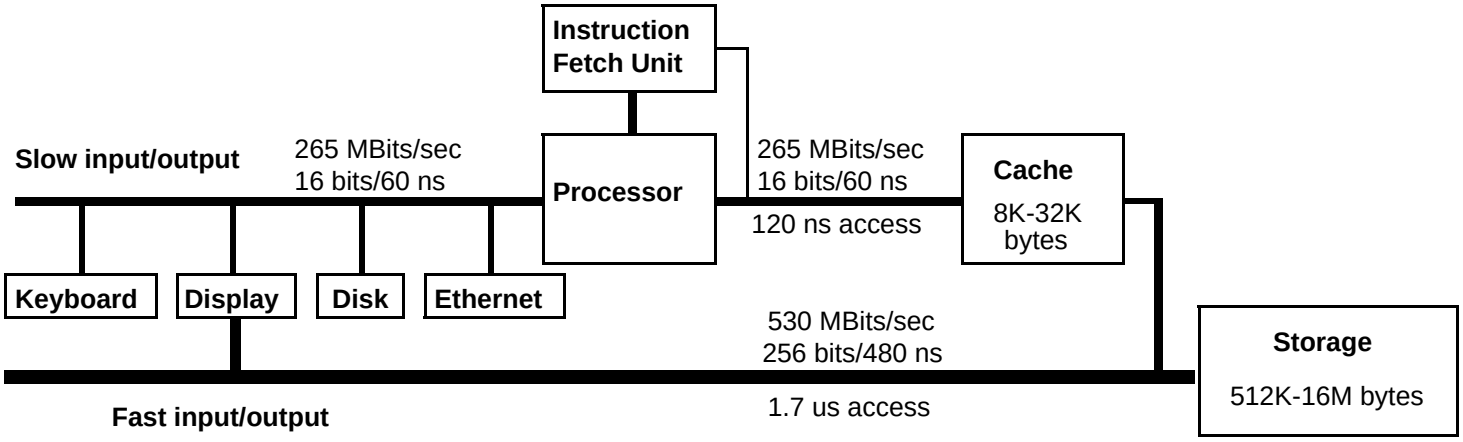


Figure 1: Dorado block diagram

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, originally tried in the TX-2 computer [5] and also used in the Alto [18], works for two main reasons. First, unless a system has both multiple memory busses (i.e., multi-ported memories) and multiple memory modules that can cycle independently, the main factor governing processor throughput is memory contention. Put simply, when I/O devices make memory references, the emulator ends up waiting for the memory. In this situation the processor might as well be working for the I/O device. Second, complex device interfaces can be implemented with relatively little dedicated hardware, since the full power of the processor is available to each device.

This processor sharing is accomplished with 16 hardware-scheduled microcode processes, called *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 so as to serve the highest priority request; control can switch from one task to another on every microinstruction, without any cost in time. 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*, i.e., 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.

Many design decisions were based on the need for speed. Raw circuit speed certainly comes first. Thus, the Dorado is implemented using the fastest commercially available technology that has a reasonable level of integration and is not too hard to package. When our design was started in 1976, the obvious choice was the ECL (Emitter-Coupled Logic) 10K family of integrated circuits. These circuits make it possible for the Dorado to execute a microinstruction in 60 ns; this is the basic cycle time of the machine. Second, 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 two, the instruction fetch unit another. Third, 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, as will be seen in later sections of the paper.

1.3 Memory architecture

The paged virtual memory of the Dorado is designed to accommodate evolving memory chip technology in both the address map and main storage. Possible configurations range from the current 22-bit virtual address with 16K 256-word pages and up to one million words of storage (using 16K chips in both map and storage) to the ultimate 28-bit virtual address with 256K 1024-word pages and 16 million words of storage (using 256K chips). All address busses are wired for their maximum size, so that configuration changes can be made with chip replacement only.

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 (random access memories). 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. The relative values of these parameters are shown in Table 1.

	Cache	*Storage*
Latency^{-1}	15	1
Bandwidth	1	2
Capacity	1	250

Table 1: Parameters of the cache relative to storage

With one exception (the IFU), all memory references are initiated by the processor, which thus acts as a multiplexor controlling access to the memory (see ¶ 1.2 and [10]), and is the sole source of addresses. Once started, however, a reference proceeds independently of the processor. Each one carries with it the number of its originating task, 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 task 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.

Table 2 lists the types of memory references executable by microcode. Figure 2, a picture of the memory system's main data paths, should clarify the sources and destinations of data transferred by these references (parts of Figure 2 will be explained in more detail later). All references, including fast I/O references, specify virtual, not real addresses. Although a microinstruction actually specifies a displacement and a base register which together form the virtual address, for convenience we will suppress this fact and write, for example, *Fetch*(*a*) to mean a fetch from virtual address *a*.

A *Fetch* from the cache delivers data to a 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/O*Read* reference delivers a 16-word *block* of data from storage to the *FastOutBus* (by way of the error corrector, as shown in Figure 2), tagged with the identity of the requesting task; the associated output device is expected to monitor this bus and grab the data when it appears. Similarly, the processor can *Store* one word of data into the cache, or do an I/O*Write* reference which demands a block of data from an input device and sends it to storage (by way of the 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 a map entries to be read and written; these will be discussed later.

The instruction fetch unit is the only device that can make a reference independently of the processor. It uses a single base register, and is treated almost exactly like a processor cache fetch, except that the IFU has its own set of registers for receiving memory data (see [9] for details). In general we ignore IFU references from now on, since they add little complexity to the memory system.

Reference	*Task*	*Effect*
Fetch(*a*)	any task	fetches one word of data from virtual address *a* in the cache and delivers it to FetchReg register
Store(*d, a*)	any task	stores data word *d* at virtual address *a* in the cache
I/O*Read*(*a*)	I/O task only	reads block at virtual address *a* in storage and delivers it to a fast output device
I/O*Write*(*a*)	I/O task only	takes a block from a fast input device and writes it at virtual address *a* in storage
Prefetch(*a*)	any task	forces the block at virtual address *a* into the cache
Flush(*a*)	emulator only	removes from the cache (re-writing to storage if necessary) the 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 2: Memory-reference instructions available to microcode

A cache reference usually *hits*; i.e., it finds the referenced word in the cache. If it *misses* (does not find the word), 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*, and we will generally use these words only for storage operations in order to distinguish them from the references made by the processor. 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.

1.4 Implementation for high performance

Two major forms of concurrency make it possible to implement the memory system's functions with high performance:

> *Physical*: the cache (8K-32K bytes) and the main storage (.5M-32M bytes) are almost independent. Normally, programs execute from the cache, and fast I/O devices transfer to and from the storage. Of course, there must be some coupling when a program refers to data that is not in the cache, or when I/O touches data that is; this is the subject of ¶ 4.
>
> *Temporal*: both cache and storage are implemented by fully segmented pipelines. Each can accept a new operation once per cycle of the memory involved: every machine cycle (60 ns) for the cache, and every eight machine cycles (480 ns) for the storage.

To support this concurrency, the memory has independent busses for cache and main storage addressing (2), data into and out of the cache (2) and main storage (2), and fast input and output (2). The data busses, but not the address busses, are visible in Figure 2. It is possible for all eight busses to be active in a single cycle, and under peak load the average utilization is about 75%. In general, there are enough busses that an operation never has to wait for a bus; thus the problems of concurrently scheduling complex operations that share many resources are simplified by reducing the number of shared resources to the unavoidable minimum of the storage devices themselves.

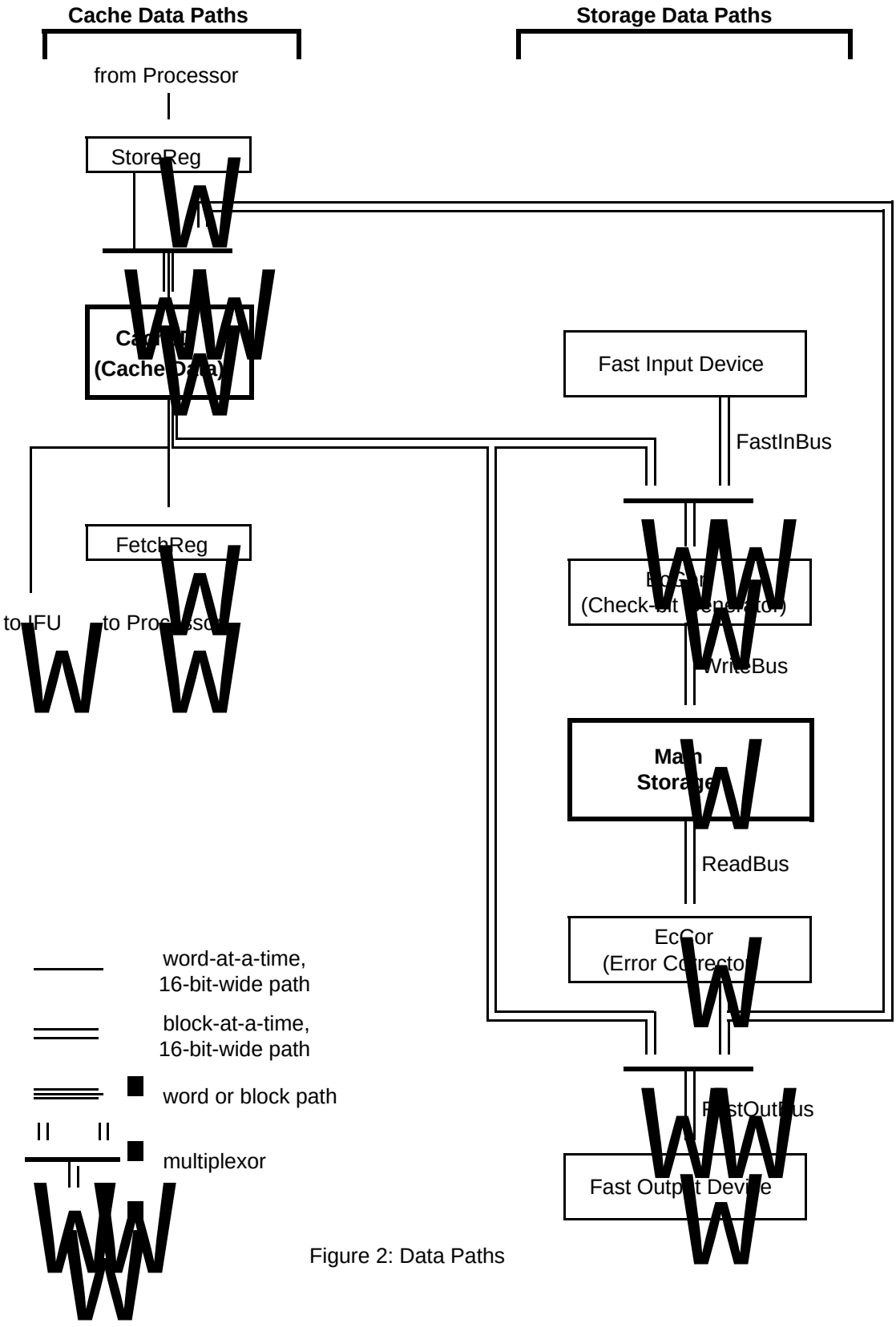


Figure 2: Data Paths

All busses are 16 bits wide; blocks of data are transferred to and from storage at the rate of 16 bits every half cycle (30 ns). This means that 256 bits can be transferred in 8 cycles or 480 ns., which is somewhat more than the 375 ns cycle time of the RAM chips that implement main storage. Thus a block size of 256 bits provides a fairly good match between bus and chip bandwidths; it is also a comfortable unit to store in the cache. The narrow busses increase the latency of a storage transfer somewhat, but they have little effect on the bandwidth. A few hundred nanoseconds of latency is of little importance either for sequential I/O transfers or for delivery of data to a properly functioning cache.

Various measures are taken to maximize the performance of the cache. Data stored there is not written back to main storage until the cache space is needed for some other purpose (the *write-back* rather than the more common *write-through* discipline [1, 14]); this make it possible to use memory locations much like registers in an interpreted instruction set, without incurring the penalty of main storage accesses. Virtual rather than real addresses are stored in the cache, so that the speed of memory mapping does not affect the speed of cache references. (Translation buffers [15, 20] are another way to accomplish this.) This would create problems if there were multiple address spaces. Although these problems can be solved, in a single-user environment with a single address space they do not even need to be considered.

Another important technique for speeding up data manipulation in general, and cache references in particular, is called *bypassing*. Bypassing is one of the speed-up techniques used in the Common Data Bus of the IBM 360/91 [19]. Sequences of instructions having the form

(1) register _ computation1
(2) computation2 involving the register

are very common. Usually the execution of the first instruction takes more than one cycle and is pipelined. As a result, however, the register is not loaded at the end of the first cycle, and therefore is not ready at the beginning of the second instruction. The idea of bypassing is to avoid waiting for the register to be loaded, by routing the results of the first computation directly to the inputs of the second one. The effective latency of the cache is thus reduced from two cycles to one in many cases (see ¶ 2.3).

The implementation of the Dorado memory reflects a balance among competing demands:

for simplicity, so that it can be made to work initially, and maintained when components fail;

for speed, so that the performance will be well-matched to the rest of the machine;

for space, since cost and packaging considerations limit the number of components and edgepins that can be used.

None of these demands is absolute, but all have thresholds that are costly to cross. In the Dorado we set a somewhat arbitrary speed requirement for the whole machine, and generally tried to save space by adding complexity, pushing ever closer to the simplicity threshold. Although many of the complications in the memory system are unavoidable consequences of the speed requirements, some of them could have been eliminated by adding hardware.

2. The cache

The memory system is organized into two kinds of building blocks: pipeline *stages*, which provide the control (their names are in SMALL CAPITALS), and *resources*, which provide the data paths and memories. Figure 3 shows the various stages and their arrangement into two pipelines. One, consisting of the ADDRESS and HITDATA stages, handles cache references and is the subject of this section; the other, containing MAP, WRITETR, STORAGE, READTR1 and READTR2, takes care of storage references and is dealt with in ¶ 3 and 4. References start out either in PROC, the processor, or in the IFU.

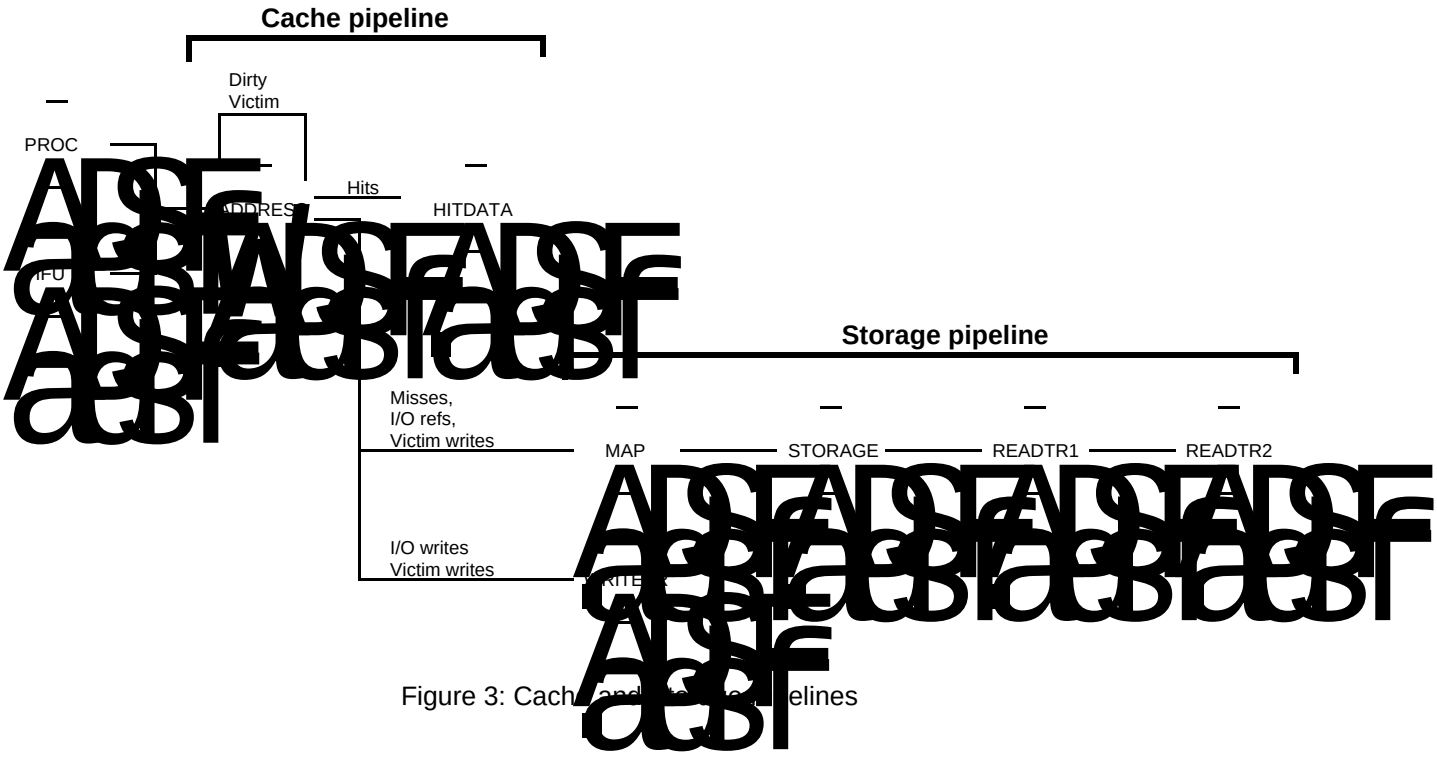


Figure 3: Cache and storage pipelines

The cache pipeline's two resources, *CacheA* and *CacheD*, correspond roughly to its two stages, although each is also used by other stages in the storage pipeline. CacheA stores addresses and associated flags, and contains the comparators which decide whether a given address is currently in the cache. CacheD stores cache data. Figure 4 shows the data paths and memories of these resources. The numbers on the left side of the figure indicate the time at which a reference reaches each point in the pipeline, relative to the start of the microinstruction making the reference.

Every reference is first handled by the 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 as described in ¶ 3.

HITDATA obtains the cache address of the word being referenced from ADDRESS, sends this address to CacheD, which stores 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.

2.1 Cache addressing

Each reference begins by adding the contents of a base register to a displacement provided by the processor (or IFU). A task-specific register holds a 5-bit pointer to a task's current base register. These pointers, as well as the base registers themselves, can be changed by microcode.

Normally the displacement is 16 bits, but by using both its busses the processor can supply a full 28-bit displacement. The resulting sum is the virtual address for the reference. It is divided into a 16-bit *key*, an 8-bit *row* number, and a 4-bit *word* number; Figure 4 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 8 row bits), and it must appear in some column of that row if it is in the cache at all. For each row, 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 to the elements of a set.

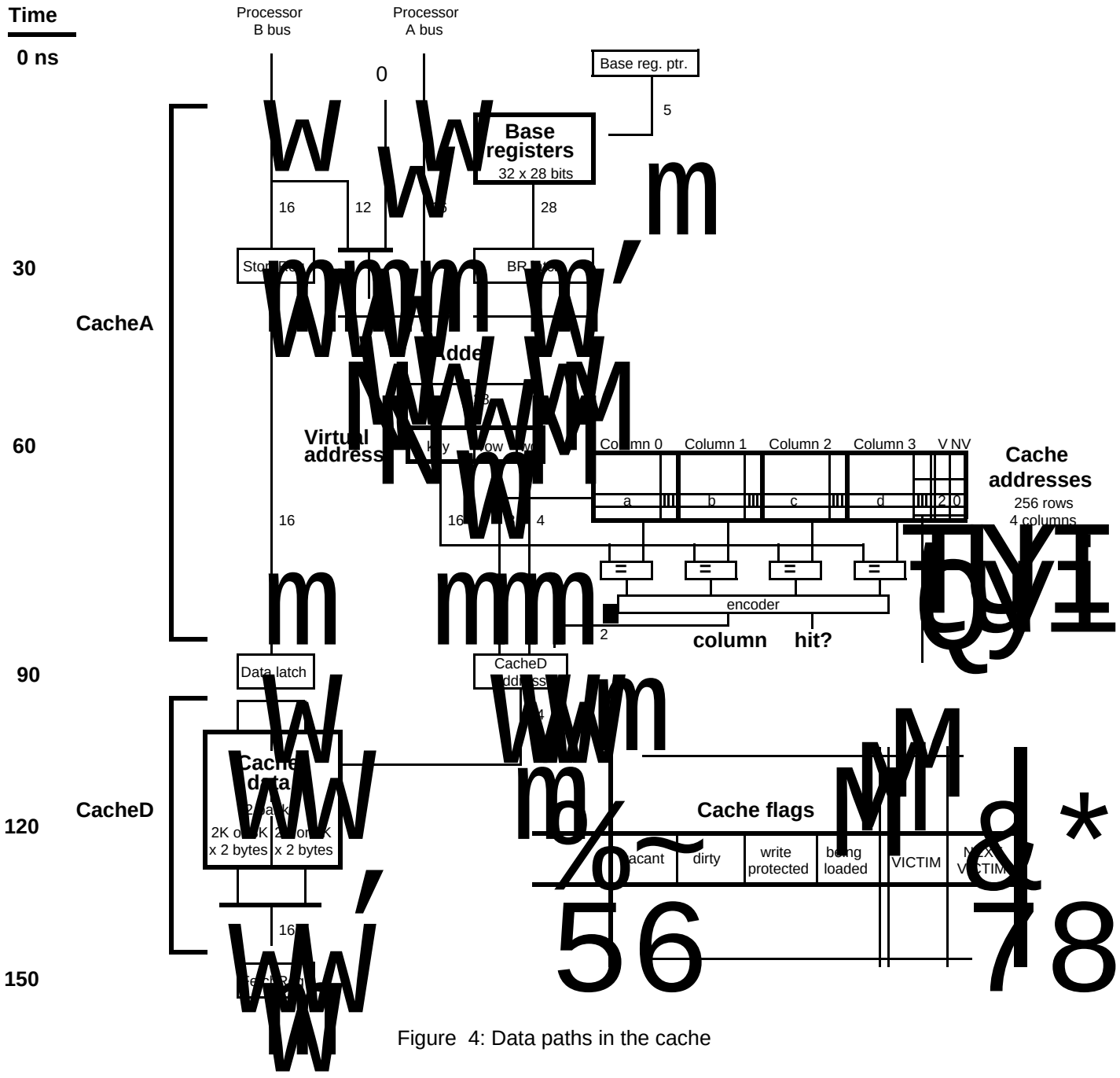


Figure 4: Data paths in the cache

Given this organization, it is simple to determine whether an address is in the cache. Its row bits are used to select a row, and the four keys stored there are compared with the given address. If one of them matches, there is a hit and the address has been located in the cache. The number of the column that matched, together with the row bits, identifies the block completely, and the four word bits of the address select one of the 16 words within that block. If no key matches, there is a

miss: the address is not present in the cache. During normal operation, it is not possible for more than one column to match. The entire matching process can be seen in Figure 4, between 60 and 90 ns after the start of the reference. The cache address latched at 90 contains the row, word and column; these 14 bits address a single word in CacheD. Of course, only the top 16 key bits of the address need be matched, since the row bits are used to select the row, and all the words of a block are present or absent together.

Four flag bits are stored with each cache entry to keep track of its status. We defer discussion of these flags until ¶ 4.

2.2 Cache data

The CacheD resource 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. CacheD is quite simple, and would consist of nothing but a 16K by 16 bit memory were it not for the bandwidth of the storage. To keep up with storage the cache must be able to accept a word every half cycle (30 ns.). Since its memory chips cannot cycle this fast, CacheD is organized in two banks which run a half-cycle out of phase when transferring data to or from the storage. On a hit, however, both banks are cycled together and CacheD behaves like an 8K by 32 bit memory. A multiplexor selects the proper half to deliver into FetchReg. All this is shown in Figure 4.

Figure 4 does not, however, show how FetchReg is made task-specific. In fact, there is a 16-word memory *FetchReg*RAM in addition to the register shown. The register holds the data value for the currently executing task. When a *Fetch* reference completes, the word from CacheD is always loaded into the RAM entry for the task that made the reference; it is also loaded into FetchReg if that task is the one currently running. Whenever the processor switches tasks, the FetchRegRAM entry for the new task is read out and loaded into FetchReg. Matters are further complicated by the bypassing scheme described in the next subsection.

StoreReg is not task-specific. The reason for this choice and the problem it causes are explained in ¶ 5.1.

2.3 Cache pipelining

From the beginning of a cache reference, it takes two and a half cycles before the data is ready in FetchReg, even if it hits and there are no delays. However, because of the latches in the pipeline (some of which are omitted from Figure 4), a new reference can be started every cycle, and if there are no misses the pipeline will never clog up, but will continue to deliver a word every 60 ns. This works because nothing in later stages of the pipeline affects anything that happens in an earlier stage.

The exception to this principle is delivery of data to the processor itself. When the processor uses data that has been fetched, it depends on the later stages of the pipeline. In general this dependency is unavoidable, but in the case of the cache the bypassing technique described in ¶ 1.4 is used to reduce the latency. A cache reference logically delivers its data to the FetchReg register at the end of the cycle following the reference cycle (actually halfway through the second cycle, at 150 in Figure 4). Often the data is then sent to a register in the processor, with a (microcode) sequence such as

(1) *Fetch*(address)
(2) register _ FetchReg
(3) computation involving register.

The register is not actually loaded until cycle (3); hence the data, which is ready in the middle of cycle (3), arrives in time, and instruction (2) does not have to wait. The data is supplied to the computation in cycle (3) by bypassing. The effective latency of the cache is thus only one cycle in this situation.

Unfortunately this sleight-of-hand does not always work. The sequence

(1) *Fetch*(address)
(2) computation involving FetchReg

actually needs the data during cycle (2), which will therefore have to wait for one cycle (see ¶ 5.1). Data retrieved in cycle (1) would be the old value of FetchReg; this allows a sequence of fetches

(1) *Fetch*(address1)
(2) register1 _ FetchReg, *Fetch*(address2)
(3) register2 _ FetchReg, *Fetch*(address3)
(4) register3 _ FetchReg, *Fetch*(address4)
. . .

to proceed at full speed.

3. The storage pipeline

Cache misses and fast I/O references use the storage portion of the pipeline, shown in Figure 3. In this section we first describe the operation of the individual pipeline stages, then explain how fast I/O references use them, and finally discuss how memory faults are handled. Using I/O references to expose the workings of the pipeline allows us to postpone until ¶ 4 a close examination of the more complicated references involving both cache and storage.

3.1 Pipeline stages

Each of the pipeline stages is implemented by a simple finite-state automaton that can change state on every microinstruction cycle. Resources used by a stage are controlled by signals that its automaton produces. Each stage *owns* some resources, and some stages *share* resources with others. 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.

3.1.1 The ADDRESS stage

As we saw in ¶ 2, the ADDRESS stage computes a reference's virtual address and looks it up in CacheA. If it hits, and is not I/O*Read* or I/O*Write*, control is passed to HITDATA. Otherwise, control is passed to MAP, starting a storage cycle. In the simplest case a reference spends just one microinstruction cycle in ADDRESS, but it can be delayed for various reasons discussed in ¶ 5.

3.1.2 The MAP stage

The MAP stage translates a virtual address into a real address by looking it up in a hardware table called the *Map*RAM, and then starts the STORAGE stage. Figure 5 illustrates the straightforward conversion of a virtual page number into a real page number. The low-order bits are not mapped; they point to a single word on the 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; see ¶ 3.1.5.

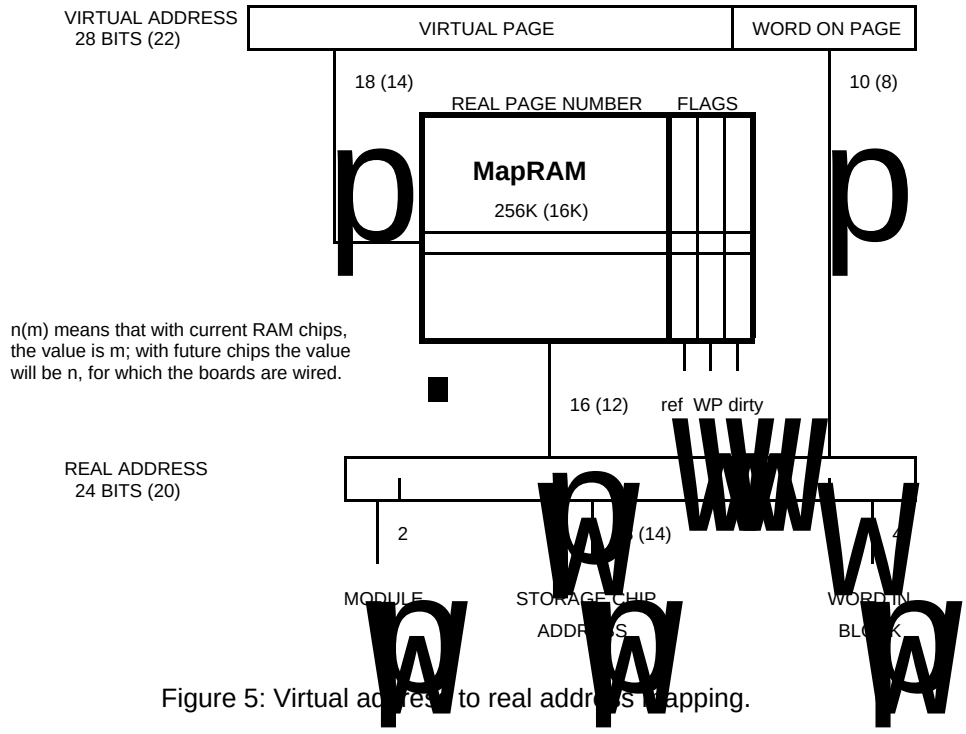


Figure 5: Virtual address to real address mapping.

MAP takes eight cycles to complete. MapRAM outputs are available for the STORAGE stage in the fifth cycle; the last three cycles are used to re-write the flags.

MapRAM entries (including flags) are written by the *MapWrite* reference. They are read by the *MapRead* reference in a slightly devious way explained in ¶ 3.3.

3.1.3 The STORAGE stage

The Dorado's main storage, a resource called *Storage*RAM, is controlled by the STORAGE stage. STORAGE is started by MAP, which supplies the real storage address and the operation type read or write. StorageRAM 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 the StorageRAM take place on word-sized busses called *ReadBus* and *WriteBus*. Block-sized shift registers called *ReadReg* and *WriteReg* lie between these busses and StorageRAM. 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 one word every half-cycle (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. Figure 6 shows one bit-slice of WriteReg, StorageRAM, and ReadReg (neglecting the error correction bits); sixteen such bit-slices comprise one storage *module*, of which there can be up to four. Figure 2 puts Figure 6 in context.

WriteReg and ReadReg are not owned by STORAGE, and it is therefore possible to overlap consecutive storage operations. Furthermore, because the eight-cycle (480 ns) duration of a transport closely matches the 375 ns. cycle time of the 16K MOS RAM chips, it is possible to keep StorageRAM busy most of the time. The resulting bandwidth is one block every eight cycles, or 530 million bits per second. ReadReg is shared between STORAGE, which loads it, and READTR1/2, which shift it. Similarly, WriteReg is shared between WRITETR, which loads it, and STORAGE, which clocks the data into the RAM chips and releases it when their hold time has expired.

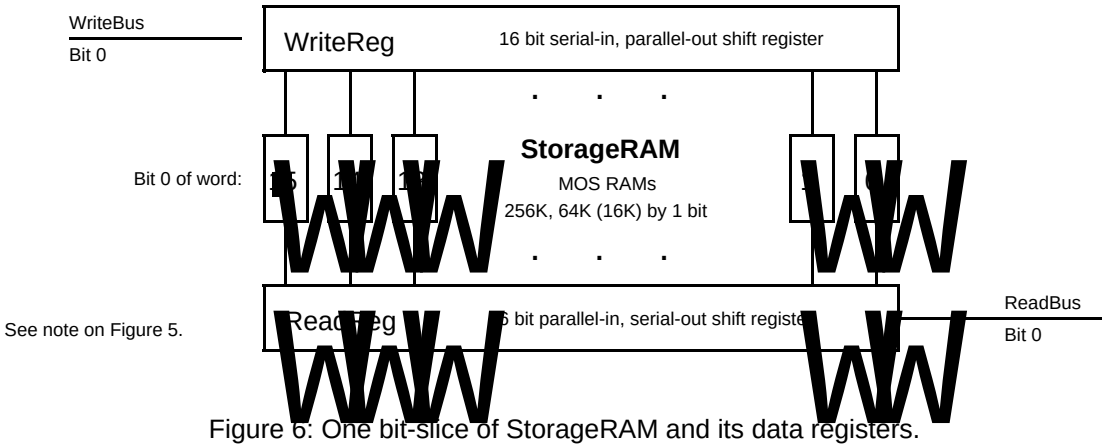


Figure 6: One bit-slice of StorageRAM and its data registers.

Each storage module has a capacity of 256K, 1M, or 4M 16-bit words, depending on whether 16K, 64K, or (hypothetical) 256K RAM chips are used. The two high-order bits of the real address select the module (see Figure 5); modules do not run in parallel. A standard Hamming error-correcting code is used, capable of correcting single errors and detecting double errors in four-word groups. Eight check bits, therefore, are stored with each *quadword*; in what follows we will often ignore these bits.

3.1.4 The WRITETR stage

The WRITETR stage transports a block into WriteReg, either from CacheD or from an input device. It owns *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 explained in ¶ 5.3.1. It runs for eleven cycles on an I/O*Write*, and for twelve cycles on a cache write. As Figure 3 shows, it starts no subsequent stages itself.

3.1.5 The READTR1 and READTR2 stages

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 (see Figure 2). Thus there are at least eleven cycles of activity related to read transport, and controlling the entire transport with a single stage would limit the rate at which read transports could be done to one every eleven cycles. No such limit is imposed by the data paths, since the error corrector is itself pipelined and does not require any wait between quadwords or blocks. To match the storage, bus, and error corrector bandwidths, read transport must be controlled by two eight-cycle stages in series; they are called READTR1 and READTR2.

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 the fault-handling microtask if necessary (see ¶ 3.3); 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 StorageRAM's 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 (¶ 3.3) 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).

3.2 Fast I/O references

We now look in detail at simple cases of the fast I/O references I/O*Read* and I/O*Write*. These references proceed almost independently of the cache, and are therefore easier to understand than fetch and store references, which may involve both the cache and storage.

The reference I/O*Read*(*x*) delivers a block of data from virtual location *x* to a fast output device. Figure 7 shows its progress through the memory system; time divisions on the horizontal axis correspond to microinstruction cycles (60 ns.). At the top is the flow of the reference through the pipeline; in the middle is a time line annotated with the major events of the reference; at the bottom is a block diagram of the resources used. The same time scale is used in all three parts, so that a vertical section shows the stages, the major events, and the resources in use at a particular time. Most of the stages pass through eight states, labelled 0 through 7 in the figure.

The I/O*Read* spends one cycle in the processor and then one in ADDRESS, during which *x* is computed and looked up in CacheA. We assume for the moment that *x* misses; what happens if it hits is the subject of ¶ 4.4. ADDRESS starts MAP, passing it *x*. MAP translates *x* into the real address *r*, and starts STORAGE, passing it *r*; MAP then spends three more cycles rewriting the flags as appropriate and completing the Map*RAM* cycle (¶ 3.1.2). STORAGE does a Storage*RAM* access and loads the 16-word block of data (together with its check bits) into ReadReg. It then starts READTR1

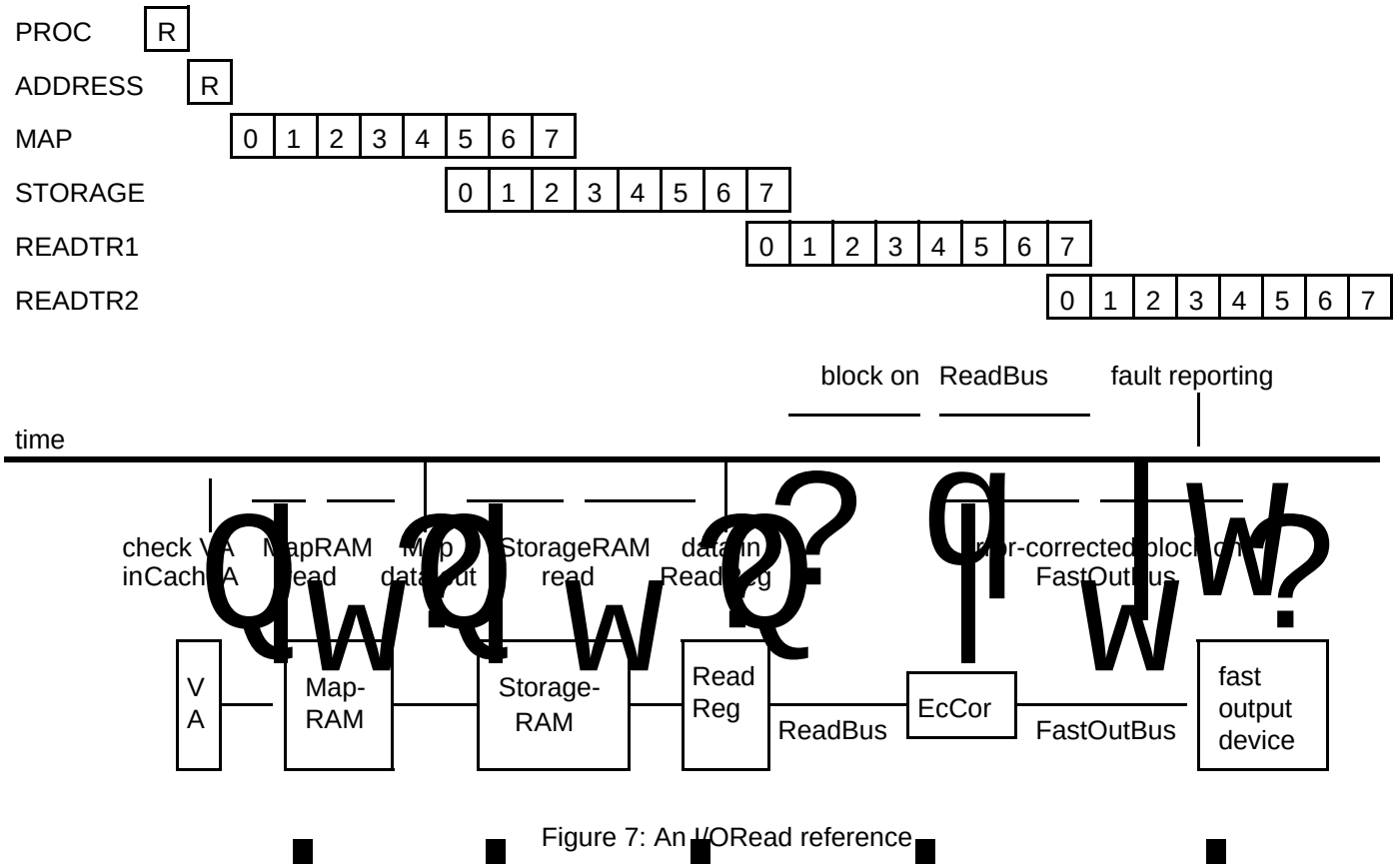


Figure 7: An I/ORead reference

and completes the StorageRAM cycle (¶ 3.1.3). READTR1 and READTR2 transport the data, control the error corrector, and deliver the data to FastOutBus (¶ 3.1.5). Fault reporting, if necessary, is done by READTR2 as soon as the condition of the last quadword in the block is known (¶ 3.3).

It is clear from Figure 7 that an I/O*Read* can be started every eight machine cycles, since this is the longest period of activity of any stage. This would result in 530 million bits per second of bandwidth, the maximum supportable by the memory system. The inner loop of a fast I/O task can be written in two microinstructions, so if a new I/O*Read* is launched every eight cycles, one-fourth of the processor capacity will be used. Because ADDRESS is used for only one cycle per I/O*Read*, other tasks notably the emulator may continue to hit in the cache when the I/O task is not running.

I/O*Write*(*x*) writes into virtual location *x* a block of data delivered by a fast input device, together with appropriate Hamming code check bits. The data always goes to storage, never to the cache, but if address *x* happens to hit in the cache, the entry is invalidated by setting a flag (¶ 4). Figure 8 shows that an I/O*Write* proceeds through the pipeline very much like an I/O*Read*. The difference, of course, is that the WRITETR stage runs, and the READTR1 and READTR2 stages, although they run, do not transport data. Note that the write transport, from FastInBus to WriteBus, proceeds in parallel with mapping. Once the block has been loaded into WriteReg, STORAGE issues a write signal to StorageRAM. All that remains is to run READTR1 and READTR2, as explained above. If a map fault occurs during address translation, the write signal is blocked and the fault is passed along to be reported by READTR2.

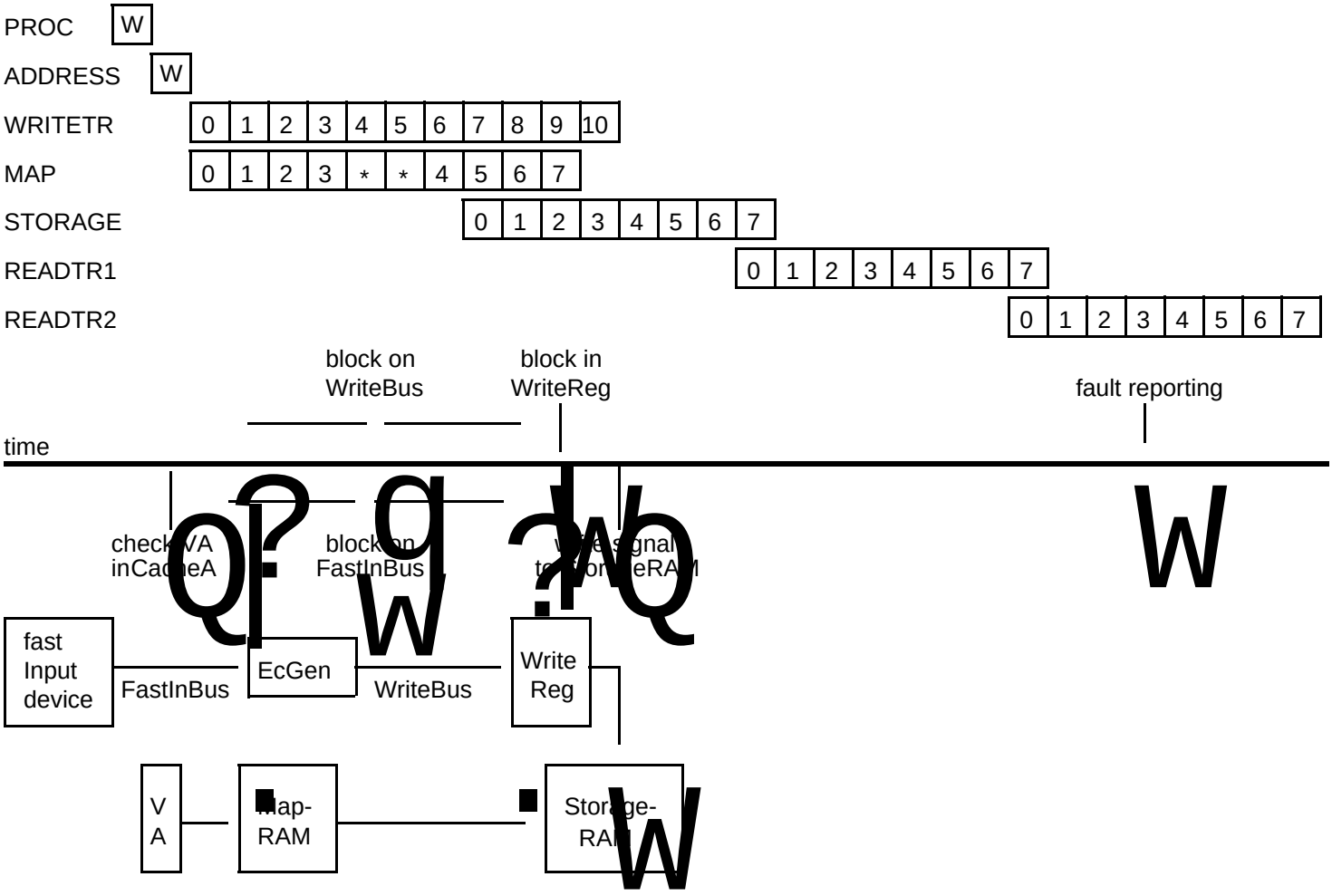


Figure 8: An I/OWrite reference

Figure 8 shows a delay in the MAP stage's handling of I/O*Write*. MAP remains in state 3 for two extra cycles, which are labelled with asterisks, rather than state numbers, in Figure 8. This delay allows the write transport to finish before the write signal is issued to StorageRAM. This synchronization and others are detailed in ¶ 5.

Because WRITETR takes eleven cycles to run, I/O*Write*s can only run at the rate of one every eleven cycles, yielding a maximum bandwidth for fast input devices of 390 million bits per second. At that rate, two of every eleven cycles would go to the I/O task's inner loop, consuming 18 percent of the processor capacity. But again, other tasks could hit in the cache in the remaining nine cycles.

3.3 History and fault reporting

There are two kinds of memory system faults: *map* and *storage*. A map fault is a MapRAM parity error, a reference to a page marked vacant, or a write operation to a write-protected page. A storage fault is either a single or a double error (within a quadword) detected during a read. In what follows we do not always distinguish between the two types.

Consider how a page fault might be handled. MAP has read the MapRAM entry for a reference and found the virtual page marked vacant. At this point there may be another reference in ADDRESS waiting for MAP, and one more in the processor waiting for ADDRESS. An *earlier* reference may be in READTR1, perhaps about to cause a storage fault. The processor is probably several instructions beyond the one that issued the faulting reference, perhaps in another task. What to do? It would be quite cumbersome at this point to *halt* the memory system, deal with the fault, and restart the memory system in such a way that the fault was transparent to the interrupted tasks. Instead, the Dorado allows the reference to complete, while blunting any destructive consequences it might have. A page fault, for example, forces the cache's vacant flag to be set when the read transport is done. At the very end of the pipeline READTR2 wakes up the Dorado's highest-priority microtask, the *fault task*, which must deal appropriately with the fault, perhaps with the help of memory-management software.

Because the fault may be reported well after it happened, a record of the reference must be kept which is complete enough that the fault task can sort out what has happened. Furthermore, because later references in the pipeline may cause additional faults, this record must be able to encompass *several* faulting references. The necessary information associated with each reference, about 80 bits, is recorded in a 16-element memory called *History*. Table 3 gives the contents of History and shows which stage is responsible for writing each part. History is managed as a ring buffer and is addressed by a 4-bit Storage Reference Number or SRN, which is passed along with the reference through the various pipeline stages. When a reference is passed to the MAP stage, a counter containing the next available SRN is incremented. A hit writes the address portion of History (useful for diagnostic purposes; see below), without incrementing the SRN counter.

Entry	*Written by*
Virtual address, reference type, task number, cache column	ADDRESS
Real page number, MapRAM flags, map fault	MAP
Storage fault, bit corrected (for single errors)	READTR2

Table 3: Contents of the History memory

Two hardware registers accessible to the processor help the fault task interpret History: *FaultCount* is incremented every time a fault occurs; *FirstFault* holds the SRN of the first faulting reference. The fault task is awakened whenever FaultCount is non-zero; it can read both registers and clear FaultCount in a single atomic operation. It then handles FaultCount faults, reading successive elements of History starting with History[FirstFault], and then yields control of the processor to the

other tasks. If more faults have occurred in the meantime, FaultCount will have been incremented again and the fault task will be reawakened.

The fault task does different things in response to the different types of fault. Single bit errors, which are corrected, are not reported at all unless a special control bit in the hardware is set. With this bit set, the fault task can collect statistics on failing storage chips; if too many failures are occurring, the bit can be cleared and the machine can continue to run. Double bit errors may be dealt with by re-trying the reference; a recurrence of the error must be reported to the operating system, which may stop using the failing memory, and may be able to reread the data from the disk if the page is not dirty, or determine which computation must be aborted. Page faults are the most likely reason to awaken the fault task, and together with write-protect faults are dealt with by yielding to memory-management software. MapRAM parity errors may disappear if the reference is re-tried; if they do not, the operating system can probably recover the necessary information.

Microinstructions that read the various parts of History are provided, but only the emulator and the fault task may use them. These instructions use an alternate addressing path to History which does not interfere with the SRN addressing used by references in the pipeline. Reading base registers, the MapRAM, and CacheA can be done only by using these microinstructions.

This brings us to a serious difficulty with treating History as a pure ring buffer. To read a MapRAM entry, for example, the emulator must first issue a reference to that entry (normally a *MapRead*), and then read the appropriate part of History when the reference completes; similarly, a *DummyRef* (see Table 3) is used to read a base register. But because other tasks may run and issue their own references between the start of the emulator's reference and its reading of History, the emulator cannot be sure that its History entry will remain valid. Sixteen references by I/O tasks, for example, will destroy it.

To solve this problem, we designate History[0] as the emulator's "private" entry: *MapRead*, *MapWrite*, and *DummyRef* references use it, and it is excluded from the ring buffer. Because the fault task may want to make references of its own without disturbing History, another private entry is reserved for it. The ring buffer proper, then, is a 14-element memory used by all references except *MapRead*, *MapWrite*, and *DummyRef* in the emulator and fault task. For historical reasons, *Fetch*, *Store* and *Flush* references in the emulator and fault task also use the private entries; the tag mechanism (¶ 4.1) ensures that the entries will not be reused too soon.

In one case History is read, rather than written, by a pipeline stage. This happens during a read transport, when READTR1 gets from History the cache address (row and column) it needs for writing the new data and the cache flags. This is done instead of piping this address along from ADDRESS to READTR1.

4. Cache-storage interactions

The preceding sections describe the normal case in which the cache and main storage function independently. Here we consider the relatively rare interactions between them. These can happen for a variety of reasons:

> Processor references that miss in the cache must fetch their data from storage.
>
> A dirty block in the cache must be re-written in storage when its entry is needed.
>
> Prefetch and flush operations explicitly transfer data between cache and storage.
>
> I/O references that hit in the cache must be handled correctly.

Cache-storage interactions are aided by the four flag bits that are stored with each cache entry to keep track of its status (see Figure 4). The *vacant* flag indicates that an entry should never match; it is set by software during system initialization, and by hardware when the normal procedure for loading the cache fails, e.g., because of a page fault. The *dirty* flag is set when the data in the entry is different from the data in storage because the processor did a store; this means that the entry

must be written back to storage before it is used for another block. The *writeProtected* flag is a copy of the corresponding bit in the map. It causes a store into the block to miss and set *vacant*; the resulting storage reference reports a write-protect fault (¶ 3.3). The *beingLoaded* flag is set for about 15 cycles while the entry is in the course of being loaded from storage; whenever the ADDRESS stage attempts to examine an entry, it waits until the entry is not *beingLoaded*, to ensure that the entry and its contents are not used while in this ambiguous state.

When a cache reference misses, the block being referenced must be brought into the cache. In order to make room for it, some other block in the row must be displaced; this unfortunate is called the *victim*. CacheA implements an approximate least-recently-used rule for selecting the victim. With each row, the current candidate for victim and the next candidate, called *next victim*, are kept. The victim and next victim are the top two elements of an LRU stack for that row; keeping only these two is what makes the replacement rule only approximately LRU. On a miss, the next victim is promoted to be the new victim and a pseudo-random choice between the remaining two columns is promoted to be the new next victim. On each hit, the victim and next victim are updated in the obvious way, depending on whether they themselves were hit.

The flow of data in cache-storage interactions is shown in Figure 2. For example, a *Fetch* that misses will read an entire block from storage via the ReadBus, load the error-corrected block into CacheD, and then make a one-word reference as if it had hit.

What follows is a discussion of the four kinds of cache-storage interaction listed above.

4.1 Clean miss

When the processor or IFU references a word *w* that is not in the cache, and the location chosen as victim is vacant or holds data that is unchanged since it was read from storage (i.e., its dirty flag is not set), a *clean miss* has occurred. The victim need not be written back, but a storage read must be done to load into the cache the block containing *w*. At the end of the read, *w* can be fetched from the cache. A clean miss is much like an I/O*Read*, which was discussed in the previous section. The chief difference is that the block from storage is sent not over the FastOutBus to an output device, but to the CacheD memory. Figure 9 illustrates a clean miss.

All cache loads require a special cycle, controlled by READTR1, in which they get the correct cache address from History and write the cache flags for the entry being loaded; the data paths of CacheA are used to read this address and write the flags. This *RThasA* cycle takes priority over all other uses of CacheA and History, and can occur at any time with respect to ADDRESS, which also needs access to these resources. Thus all control signals sent from ADDRESS are inhibited by *RThasA*, and ADDRESS is forced to idle during this cycle. Figure 9 shows that the *RThasA* cycle occurs just before the first word of the new block is written into CacheD. (For simplicity and clarity we will not show *RThasA* cycles in the figures that follow.) During *RThasA*, the *beingLoaded* flag is cleared (it was set when the reference was in ADDRESS) and the *writeProtected* flag is copied from the *writeProtected* bit in MapRAM. As soon as the transport into CacheD is finished, the word reference that started the miss can be made, much as though it had hit in the first place. If the reference was a *Fetch*, the appropriate word is sent to FetchReg in the processor (and loaded into FetchRegRAM); if a *Store*, the contents of StoreReg are stored into the new block in the cache.

If the processor tries to use data it has fetched, it is prevented from proceeding, or *held* until the word reference has occurred (see ¶ 5.1). Each fetch is assigned a sequence number called its *tag*, which is logically part of the reference; actually it is written into History, and read when needed by READTR1. Tags increase monotonically. The tag of the last *Fetch* started by each task is kept in *StartedTag* (it is written there when the reference is made), and the tag of the last *Fetch* completed by the memory is kept in *DoneTag* (it is written there as the *Fetch* is completed); these are task-specific registers. Since tags are assigned monotonically, and fetches always complete in order within a task, both registers increase monotonically. If StartedTag=DoneTag, all the fetches that

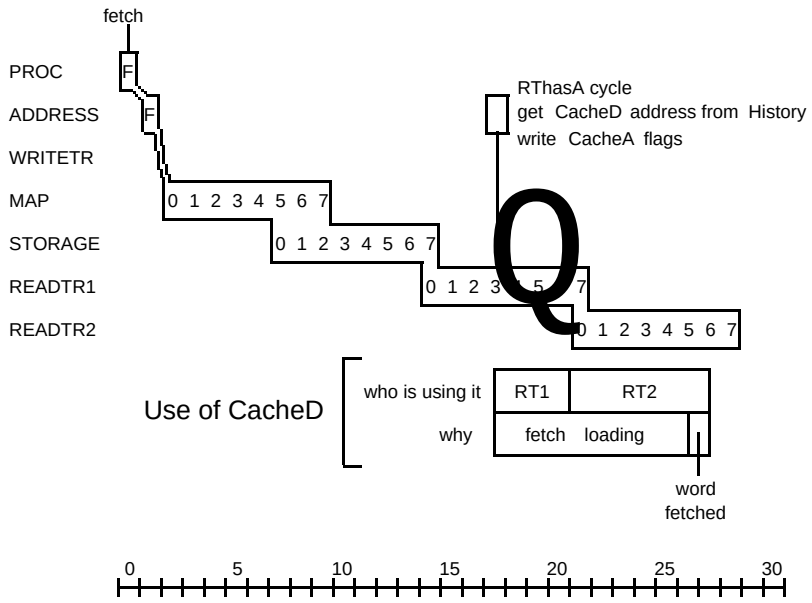


Figure 9: A clean miss

have been started are done, otherwise not; this condition is therefore sufficient to decide whether the processor should be held when it tries to use FetchReg. Because there is only one FetchReg register per task, it is not useful to start another *Fetch* until the preceding one is done and its word has been retrieved. The tags are therefore used to hold up a *Fetch* until the preceding one is done, and thus can be kept modulo 2, so that one bit suffices to represent a tag. *Store* references also use the tag mechanism, although this is not logically necessary.

(Instead of a sequence number on each reference, we might have counted the outstanding references for each task. This idea was rejected for the following rather subtle reason. In a single machine cycle *three* accesses to the counter may be required: the currently running task must read the counter to decide whether a reference is possible, and write back an incremented value; in addition, READTR2 may need to write a decremented value for a different task as a reference completes. Time allows only two references in a single cycle to the RAM in which such task-specific information must be kept. The use of sequence numbers allows the processor to read both StartedTag and DoneTag from separate RAMs, and then the processor and the memory to independently write the RAMs; thus four references are made to two RAMs in one cycle, or two to each.)

Other tasks may start references or use data freely while one task has a *Fetch* outstanding. Cache hits, for example, will not be held up, except during the *RThasA* cycle and while CacheD is busy with the transport. These and other inter-reference conflicts are discussed in more detail in ¶ 5. Furthermore, the same task may do other references, such as *Prefetches*, which are not affected by the tags. The IFU has two FetchReg registers of its own, and can therefore have two fetches outstanding. Hence it cannot use the standard tag mechanism, and instead implements this function with special hardware of its own.

4.2 Dirty miss

When a processor or IFU reference misses, and the victim has been changed by a store since arriving in the cache, a *dirty miss* has occurred, and the victim must be re-written in storage. A dirty miss gives rise to two storage operations: the write that re-writes the victim's dirty block from cache to storage, and the read that loads CacheD with the new block from storage. The actual data transports from and to the cache are done in this order (as they must be), but the storage operations are done in reverse order, as illustrated by a fetch with dirty victim in Figure 10. The figure shows that the victim reference spends eight cycles in ADDRESS waiting for the fetch to finish with MAP (recall that the asterisks mean no change of state for the stage). During this time the victim's transport is done by WRITETR.

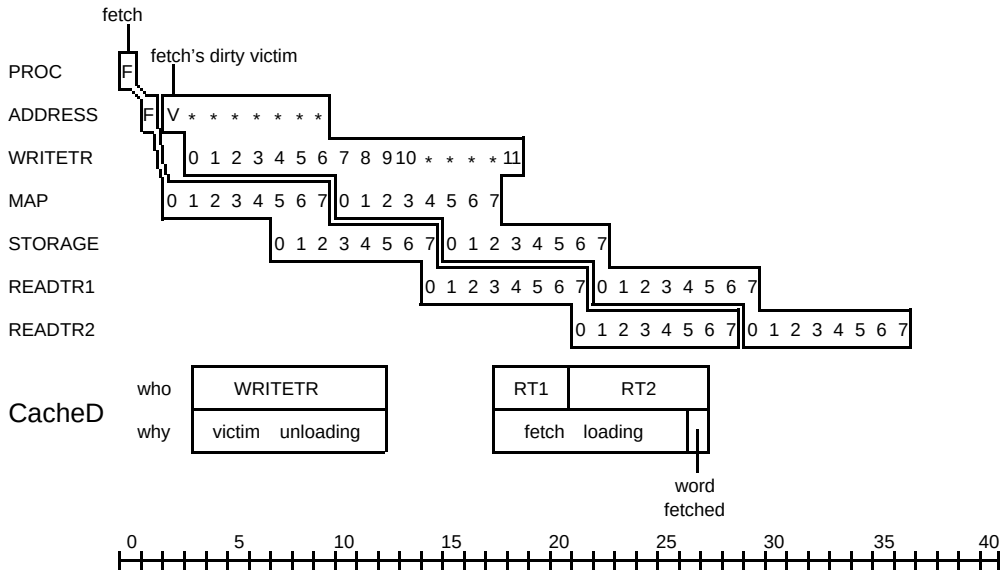


Figure 10: A dirty miss

There are several reasons for this arrangement. As we saw in ¶ 3, data transport to and from storage is not done in lockstep with the corresponding storage cycle; only the proper order of events is enforced. The existence of ReadReg and WriteReg permits this. Furthermore, there is a 12-cycle wait between the start of a read in ADDRESS and the latching of the data in ReadReg. These two considerations allow us to interleave the read and victim write operations in the manner shown in Figure 10. The read is started, and while it proceeds during the 12-cycle window the write transport for the victim is done. The data read is latched in ReadReg, and then transported into the cache while the victim data is written into storage.

Doing things this way means that the latency of a miss, from initiation of a fetch to arrival of the data, is the same regardless of whether the victim is dirty. The opposite order is worse for several reasons, notably because the delivery of the new data, which is what holds up the processor, would be delayed by twelve cycles.

4.3 Prefetch and flush

Prefetch is just like *Fetch*, except that there is no word reference. Also, because it is treated strictly as a hint, map-fault reporting is suppressed and the tags are not involved, so later references are not delayed. A *Prefetch* that hits, therefore, finishes in ADDRESS without entering MAP. A *Prefetch* that misses will load the referenced block into the cache, and cause a dirty victim write if necessary.

A *Flush* explicitly removes the block containing the addressed location from the cache, rewriting it in storage if it is dirty. *Flush* is used to remove a virtual page's blocks from the cache so that its MapRAM entry can be changed safely. If a *Flush* misses, nothing happens. If it hits, the hit location must be marked vacant, and if it is dirty, the block must be written to storage. To simplify the hardware implementation, this write operation is made to look like a victim write. A dirty *Flush* is converted into a *FlushFetch* reference, which is treated almost exactly like a *Prefetch*. Thus, when a *Flush* in ADDRESS hits, three things happen:

the victim for the selected row of CacheA is changed to point to the hit column;

the *vacant* flag is set;

if the *dirty* flag for that column is set, the *Flush* is converted into a *FlushFetch*.

Proceeding like a *Prefetch*, this does a useless read (which is harmless because the vacant flag has been set), and then a write of the dirty victim. Figure 11 shows a dirty *Flush*. The *FlushFetch* spends two cycles in ADDRESS, instead of the usual one, because of an uninteresting implementation problem.

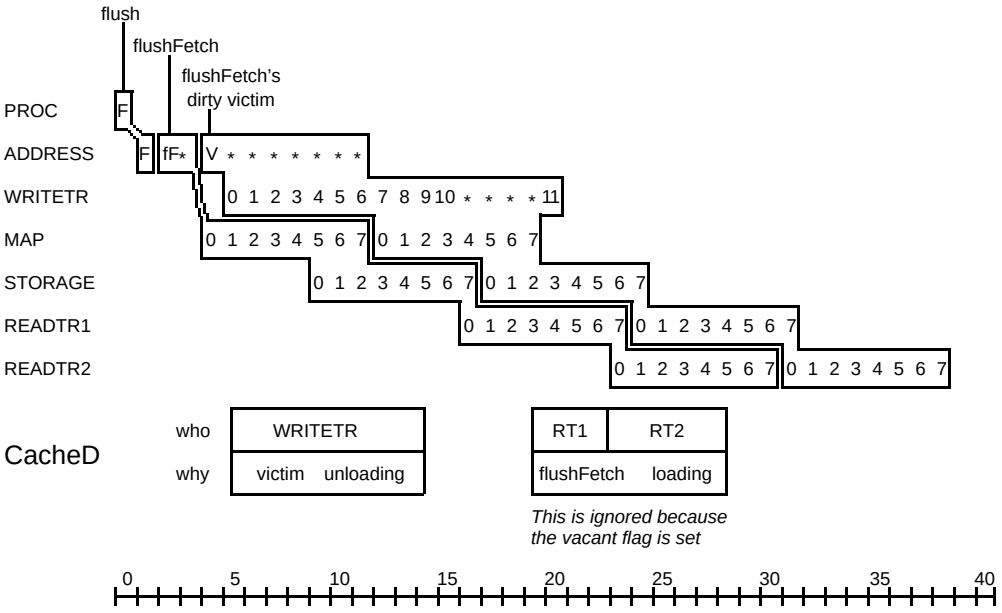


Figure 11: A dirty flush

4.4 Dirty I/ORead

If an I/O*Read* reference hits in a column with *dirty* set, the data must come from the cache rather than from storage. This is made as similar as possible to a clean I/O*Read*, since otherwise the bus scheduling would be drastically different. Hence a full storage read is done, but at the last minute data from the cache is put on FastOutBus in place of the data coming from storage, which is ignored. Figure 12 illustrates a dirty I/O*Read* followed by two clean ones. Note that CacheD is active at the same time as for a standard read, but that it is unloaded rather than loaded. This simplifies the scheduling of CacheD, at the expense of tying up FastOutBus for one extra cycle. Since many operations use CacheD, but only I/O*Read* uses FastOutBus, this is a worthwhile simplification (see ¶ 5.3.4).

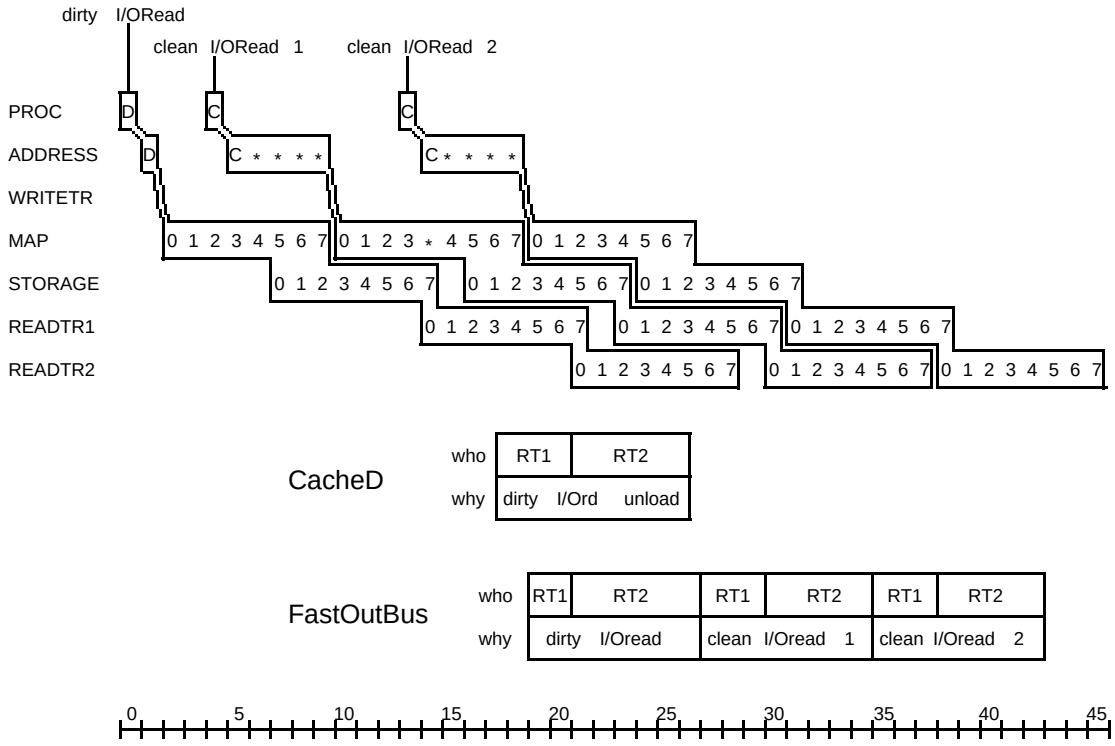


Figure 12: A dirty I/ORead and two clean ones

5. Traffic control

Thus far we have considered memory operations only in isolation from each other. Because the system is pipelined, however, several different operations can be active at once. Measures must be taken to prevent concurrent references from interfering with each other, and to prevent new operations from starting if the system is not ready to accept them. In this section we discuss those measures.

Table 4 lists the resources used by each pipeline stage in three categories: *private* resources, which are used only by one stage; *handoff* resources, which are passed from one stage to another in an orderly way guaranteed not to result in conflicts; and *complex* resources, which are shared among several stages in ways that may conflict with each other. These conflicts are resolved in three ways:

> If the memory system cannot accept a new reference from the processor, it rejects it, and notifies the processor by asserting the *Hold* signal.
>
> A reference, once started, waits in ADDRESS until its immediate resource requirements (i.e., those it needs in the very next cycle) can be met; it then proceeds to MAP or to HITDATA, as shown in Figure 3.
>
> All remaining conflicts are dealt with in a single state of the MAP stage.

We will consider the three methods in turn.

	PROC	ADDRESS	HITDATA	MAP	WRITETR	STORAGE	READTR1	READTR2
Private Resources				MapRAM	WriteBus EcGen FastInBus	StorageRAM		
Handoff Resources					WriteReg	WriteReg ReadReg	ReadReg ReadBus EcCor FastOutBus	ReadBus EcCor FastOutBus
Complex Resources	FetchReg StoreReg History	CacheA History	FetchReg StoreReg CacheD	History	CacheD		CacheA CacheD History	FetchReg StoreReg CacheD History

Table 4: Pipeline resources

5.1 Hold

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.

There are four reasons for the memory system to generate *Hold*.

Data requested before it is ready. Probably the most common type of *Hold* occurs after a *Fetch*, when the data is requested before it has arrived in FetchReg. For a hit that is not delayed in ADDRESS (see below), *Hold* only happens if the data is used early in the very next cycle (i.e., if the instruction after the *Fetch* sends the data to the processor's ALU rather than just into a register). If the data is used late in the next cycle it bypasses FetchReg and comes directly from CacheD (¶ 2.3); if it is used in any later cycle it comes from FetchReg. In either case there will be no *Hold*. If the *Fetch* misses, however, the matching FetchReg operation will be held (by the tag mechanism) until the missing block has been loaded into the cache, and the required word fetched into FetchReg.

ADDRESS *busy*. A reference can be held up in ADDRESS for a variety of reasons, e.g., because it must proceed to MAP, and MAP is busy with a previous reference. Other reasons are discussed in ¶ 5.2 below. Every reference needs to spend at least one cycle in ADDRESS, so new references will be held as long as ADDRESS is busy. A reference needs the data paths of CacheA in order to load its address into ADDRESS, and these are busy during the *RThasA* cycle discussed above (¶ 4.1); hence a reference in the cycle before *RThasA* is held.

StoreReg busy. When a *Store* enters ADDRESS, the data supplied by the processor is loaded into StoreReg. If the *Store* hits and there is no conflict for CacheD, StoreReg is written into CacheD in the next cycle, as Figure 4 shows. If it misses, StoreReg must be maintained until the missing block arrives in CacheD, and so new stores must be held during this time because StoreReg is not task-specific. Even on a hit, CacheD may be busy with another operation. Of course new stores by the same task would be held by the tag mechanism anyway, so StoreReg busy will only hold a *Store* in other tasks. A task-specific StoreReg would have prevented this kind of *Hold*, but the hardware implementation was

too expensive to do this, and we observed that stores are rare compared to fetches in any case.

History busy. As discussed in ¶ 3.3, a reference uses various parts of the History memory at various times as it makes its way through the pipeline. Microinstructions for reading History are provided, and they must be held if they will conflict with any other use.

The memory system *must* generate *Hold* for precisely the above reasons. It turns out, however, that there are several situations in which hardware or time can be saved if *Hold* is generated when it is not strictly needed. This was done only in cases that we expect to occur rarely, so the performance penalty should be small. An extra *Hold* has no logical effect, since it only converts the current microinstruction into a jump-to-self. One example of this situation is that a reference in the cycle after a miss is always held, even though it must be held only if the miss' victim is dirty or the map is busy; the reason is that the miss itself is detected barely in time to generate *Hold*, and there is no time for additional logic. Another example: uses of FetchReg are held while ADDRESS is busy, although they need not be, since they do not use it.

5.2 Waiting in ADDRESS

A reference in ADDRESS normally proceeds either to HITDATA (in the case of a hit) or to MAP (for a miss, a victim write or an I/O reference) after one cycle. If HITDATA or MAP is busy, it will wait in ADDRESS, causing subsequent references to be held because ADDRESS is busy, as discussed above.

HITDATA uses CacheD, and therefore cannot be started when CacheD is busy. A reference that hits must therefore wait in ADDRESS while CacheD is busy, i.e., during transports to and from storage, and during single-word transfers resulting from previous fetches and stores. Some additional hardware would have enabled a reference to be passed to HITDATA and wait there, instead of in ADDRESS, for CacheD to become free; ADDRESS would then be free to accept another reference. This performance improvement was judged not worth the requisite hardware.

When MAP is busy with an earlier reference, a reference in ADDRESS will wait if it needs MAP. An example of this is shown in Figure 10, where the victim write waits while MAP handles the read. However, even if MAP is free, a write must wait in ADDRESS until it can start WRITETR; since WRITETR always takes longer than MAP, there is no point in starting MAP first, and the implementation is simplified by the rule that starting MAP always frees ADDRESS. Figure 13 shows two back-to-back I/O*Writes*, the second of which waits one extra cycle in ADDRESS before starting both WRITETR and MAP.

The last reason for waiting in ADDRESS has to do with the *beingLoaded* flag in the cache. If ADDRESS finds that *beingLoaded* is set anywhere in the row it touches, it waits until the flag is cleared (this is done by READTR1 during the *RThasA* cycle). A better implementation would wait only if the flag is set in the column in which it hits, but this was too slow and would also require special logic to ensure that an entry being loaded is not chosen as a victim. Of course it would be much better to *Hold* a reference to a row being loaded before it ever gets into ADDRESS, but unfortunately the reference must be in ADDRESS to read the flags in the first place.

5.3 Waiting in MAP

The traffic control techniques discussed thus far, namely, *Hold* and waiting in ADDRESS, are not sufficient to prevent all the conflicts shown in Table 4. In particular, neither deals with conflicts downstream in the pipeline. Such conflicts could be resolved by delaying a reference in ADDRESS until it was certain that no further conflicts with earlier references could occur. This is not a good idea because references that hit, which is to say most references, must be held when ADDRESS is busy. If conflicts are resolved in MAP or later, hits can proceed unimpeded, since they do not use later sections of the pipeline.

At the other extreme, the rule could be that a stage waits only if it cannot acquire the resources it will need in the very next cycle. This would be quite feasible for our system, and the proper choice of priorities for the various stages can clearly prevent deadlock. However, each stage that may be forced to wait requires logic for detecting this situation, and the cost of this logic is significant. Furthermore, in a long pipeline gathering all the information and calculating which stages can proceed can take a long time, especially since in general each stage's decision depends on the decision made by the next one in the pipe.

For these reasons we adopted a different strategy in the Dorado. There is one point, early in the pipeline but after ADDRESS, at which *all* remaining conflicts are resolved. A reference is not allowed to proceed beyond that point without a guarantee that no conflicts with earlier references will occur; thus no later stage ever needs to wait. The point used for this purpose is state 3 of the MAP stage, written as MAP.3. No shared resources are used in states 0-3, and STORAGE is not started until state 4. Because there is just one wait state in the pipeline, the exact timing of resource demands by later stages is known and can be used to decide whether conflicts are possible. We now discuss the details.

5.3.1 STORAGE and WRITETR

In a write operation, WRITETR runs in parallel *but not in lockstep* with MAP; see, for example, Figure 10. Synchronization of the data transport with the storage reference itself is accomplished by two things.

> MAP.3 waits for WRITETR to signal that the transport is far enough along that the data will arrive at the StorageRAM chips no later than the write signal generated by STORAGE. This condition must be met for correct functioning of the chips. Figure 13 shows MAP waiting during an I/O*Write*.
>
> WRITETR will wait in its next-to-last state for STORAGE to signal that the data hold time of the chips with respect to the write signal has elapsed; again, the chips will not work if the data in WriteReg is changed before this point. Figure 10 shows WRITETR waiting during a victim write. The wait shown in the figure is actually more conservative than it needs to be, since WRITETR does not change WriteReg immediately when it is started.

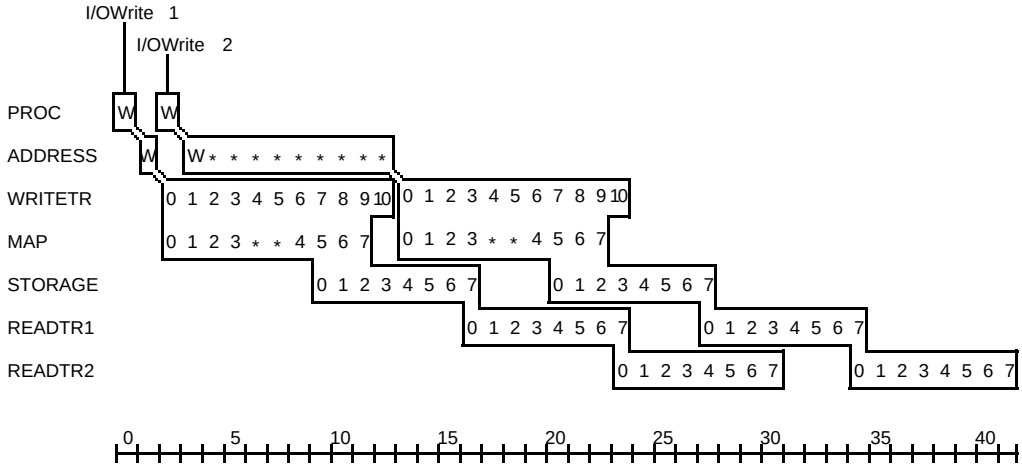


Figure 13: Two I/Owrites

5.3.2 CacheD: consecutive cache loads

Loading a block into CacheD takes 9 cycles, as explained in ¶ 4.1, and a word reference takes one more. Therefore, although the pipeline stages proper are 8 cycles long, cache loads must be spaced either 9 or 10 cycles apart to avoid conflict in CacheD. After a *Fetch* or *Store*, the next cache load must wait for 10 cycles, since these references tie up CacheD for 10 cycles. After a *Prefetch*, *FlushFetch* or dirty I/O*Read*, the next cache load must wait for 9 cycles. STORAGE sends MAP a signal that causes MAP.3 to wait for one or two extra cycles, as appropriate. Figure 14 shows a *Fetch* followed by a *Prefetch*, followed by a *Store*, and illustrates how CacheD conflict is avoided by extra cycles spent in MAP.3 Note that the *Prefetch* waits two extra cycles, while the *Store* only waits one extra.

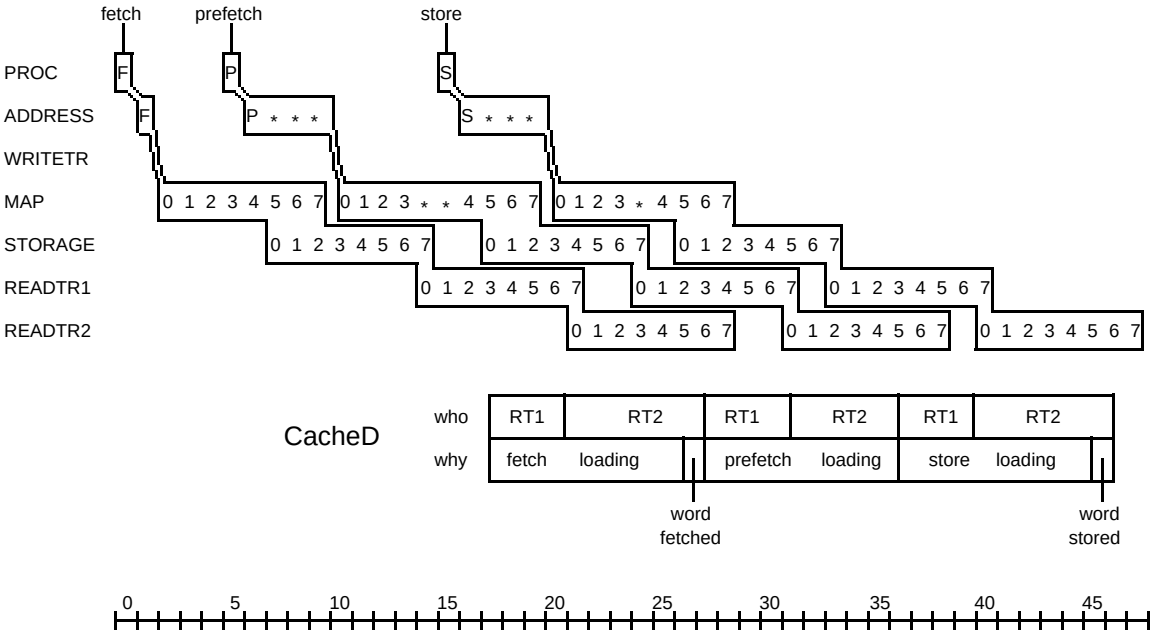


Figure 14: Cache load conflict for CacheD

5.3.3 CacheD: load and unload

The other source of conflict for CacheD is between loading it in a miss read and unloading it in a victim write. This conflict does not arise between a miss read and its own victim, because the victim is finished with CacheD before the read needs it; Figure 10 illustrates this. There is a potential conflict, however, between a miss read and the *next* reference's victim. CacheD is loaded quite late in a read, but unloaded quite early in a write, as the figure shows, so the pipeline by itself will not prevent a conflict. Instead, the following interlock is used. If a miss is followed by another miss with a dirty victim:

> ADDRESS waits to start WRITETR for the victim transport until the danger of CacheD conflict with the first miss is past.
>
> MAP.3 waits while processing the read for the second miss (not its victim write) until WRITETR has been started. This ensures that the second read will not get ahead of its victim write enough to cause a CacheD conflict. Actually, to save hardware we used the same signal to control both waits, which causes MAP.3 to wait two cycles longer than necessary.

Figure 15 shows a *Store* with clean victim followed by a *Fetch* with dirty victim and illustrates this interlock. ADDRESS waits until cycle 26 to start WRITETR. Also, the fetch waits in MAP.3 until the same cycle, thus spending 13 extra cycles there, which forces the fetch victim to spend 13 extra cycles in ADDRESS. The two-cycle gap in the use of CacheD shows that the fetch could have left MAP.3 in cycle 24.

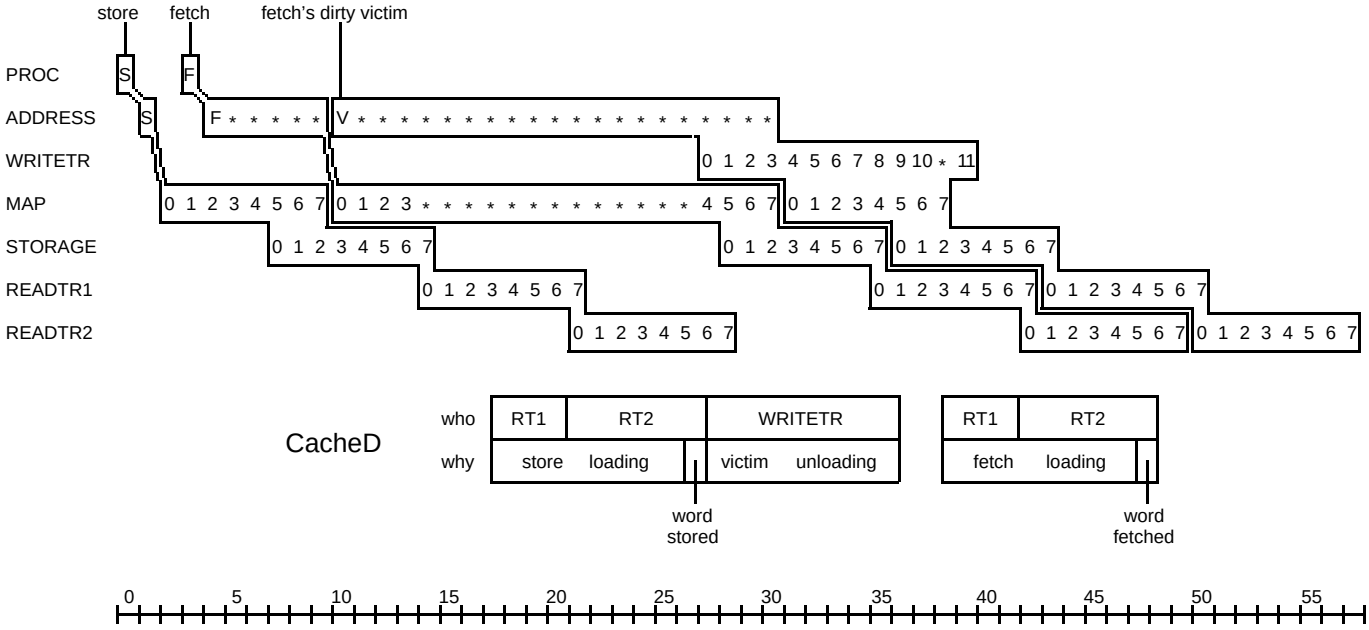


Figure 15: Cache load/unload conflict for CacheD

5.3.4 FastOutBus conflicts

The final reason for waiting in MAP.3 is a conflict over the FastOutBus, used by I/O*Read* references. A dirty I/O*Read* uses FastOutBus one cycle later than an ordinary I/O*Read*, so that it can use CacheD with the same timing as a cache load; see ¶ 4.4. The potential FastOutBus conflict is prevented by delaying an I/O*Read* immediately after a dirty I/O*Read* by one extra cycle in MAP.3. Figure 12 illustrates this, and also shows how a clean I/O*Read* can start every eight cycles and encounter no delay.

6. Physical implementation

A primary design goal of the Dorado as a personal computer is compactness. The whole machine (except the storage) is implemented on a single type of logic board, which is 14 inches square, and can hold 288 16-pin integrated circuits and 144 8-pin packages containing terminating resistors.

Boards slide into zero-insertion-force connectors mounted in two sideplanes, one on each side of the board, which have both bus wiring and interboard point-to-point wiring. Interboard spacing is 0.625 inch, so that the chassis stack of twenty-four board slots is 15 inches high. The entire machine, including cooling and power, occupies about .14 m^3 (4.5 ft^3). There are 192 pins on each side of the board; 8 are used for power connections, and the remainder in pairs for grounds and signals. Thus 184 signals can enter or leave the board.

Main storage boards are the same size as logic boards but are designed to hold an array of MOS RAMs instead of random ECL logic. A pair of storage boards make up a module, which holds 512K bytes (plus error correction) when populated with 16K RAMs, 2M bytes with 64K RAMs, or 8M bytes with (hypothetical) 256K RAMs. There is room for four modules, and space not used for storage modules can hold I/O boards. Within a module, one board stores all the words with even addresses, the other those with odd addresses. The boards are identical, and are differentiated by sideplane wiring.

A standard Dorado contains, in addition to its storage boards, eleven logic boards, including disk, display, and network controllers. Extra board positions can hold additional I/O controllers. Three boards implement the memory system (in about 800 chips); they are called **ADDRESS**, **PIPE**, and **DATA**, names which reflect the functional partition of the system. **ADDRESS** contains the processor interface, base registers and virtual address computation, CacheA (implemented in 256 by 4 RAMs) and its comparators, and the LRU computation. It also generates *Hold*, addresses **DATA** on hits, and sends storage references to **PIPE**.

DATA houses CacheD, which is implemented with 1K by 1 or 4K by 1 ECL RAMs, and holds 8K or 32K bytes respectively. **DATA** is also the source for FastOutBus and WriteBus, and the sink for FastInBus and ReadBus, and it holds the Hamming code generator-checker-corrector. **PIPE** implements MapRAM, all of the pipeline stage automata (except ADDRESS and HITDATA) and their interlocks, and the fault reporting, destination bookkeeping, and refresh control for the MapRAM and StorageRAM chips. The History memory is distributed across the boards: addresses on **ADDRESS**, control information on **PIPE**, and data errors on **DATA**.

Although our several prototype Dorados can run at a 50 nanosecond microcycle, most of the machines run instead at 60 nanoseconds. This is due mainly to a change in board technology from a relatively expensive point-to-point wire-routing method to a cheaper Manhattan routing method.

7. Performance

The memory system's performance is best characterized by two key quantities: the cache hit rate and the percentage of cycles lost due to *Hold* (¶ 5.1). In fact, *Hold* by itself measures the cache hit rate indirectly, since misses usually cause many cycles of *Hold*. Also interesting are the frequencies of stores and of dirty victim writes, which affect performance by increasing the frequency of *Hold* and by consuming storage bandwidth. We measured these quantities with hardware event-counters, together with a small amount of microcode that runs very rarely and makes no memory references itself. The measurement process, therefore, perturbs the measured programs only trivially.

We measured three Mesa programs: two VLSI design-automation programs, called Beads and Placer; and an implementation of Knuth's TEX [8]. All three were run for several minutes (several billion Dorado cycles). The cache size was 4K 16-bit words.

	Percent of cycles:		*Percent of references:*		*Percent of misses:*
	References	*Hold*	*Hits*	*Stores*	*Dirty victims*
Beads	36.4	8.14	99.27	10.5	16.3
Placer	42.9	4.89	99.82	18.7	65.5
TEX	38.4	6.33	99.55	15.2	34.9

Table 5: Memory system performance

Table 5 shows the results. The first column shows the percentage of cycles that contained cache references (by either the processor or the IFU), and the second, how many cycles were lost because they were held. *Hold*, happily, is fairly rare. The hit rates shown in column three are gratifyingly

large all over 99 percent. This is one reason that the number of held cycles is small: a miss can cause the processor to be held for about thirty cycles while a reference completes. In fact, the table shows that *Hold* and hit are inversely related over the programs measured. Beads has the lowest hit rate and the highest *Hold* rate; Placer has the highest hit rate and the lowest *Hold* rate.

The percentage of *Store* references is interesting because stores eventually give rise to dirty victim write operations, which consume storage bandwidth and cause extra occurrences of *Hold* by tying up the ADDRESS section of the pipeline. Furthermore, one of the reasons that the StoreReg register was not made task-specific was the assumption that stores would be relatively rare (see the discussion of StoreReg in ¶ 5.1). Table 5 shows that stores accounted for between 10 and 19 percent of all references to the cache.

Comparing the number of hits to the number of stores shows that the write-back discipline used in the cache was a good choice. Even if *every* miss had a dirty victim, the number of victim writes would still be much less than under the write-through discipline, when every *Store* would cause a write. In fact, not all misses have dirty victims, as shown in the last column of the table. The percentage of misses with dirty victims varies widely from program to program. Placer, which had the highest frequency of stores and the lowest frequency of misses, naturally has the highest frequency of dirty victims. Beads, with the most misses but the fewest stores, has the lowest. The last three columns of the table show that write operations would increase about a hundredfold if write-through were used instead of write-back.

Acknowledgements

The concept and structure of the Dorado memory system are due to Butler Lampson and Chuck Thacker. Much of the design was brought to the register-transfer level by Lampson and Brian Rosen. Final design, implementation, and debugging were done by the authors and Ed McCreight, who was responsible for the main storage boards. Debugging software and microcode were written by Ed Fiala, Willie-Sue Haugeland, and Gene McDaniel. Haugeland and McDaniel were also of great help in collecting the statistics reported in ¶ 7. Useful coments on earlier versions of this paper were contributed by Forest Baskett, Gene McDaniel, Jim Morris, Tim Rentsch, and Chuck Thacker.

References

1. Bell, J. *et. al*. An investigation of alternative cache organizations. *IEEE Trans. Computers* **C-23**, 4, April 1974, 346-351.
2. Bloom, L., *et. al*. Considerations in the design of a computer with high logic-to-memory speed ratio. *Proc. Gigacycle Computing Systems*, AIEE Special Pub. S-136, Jan. 1962, 53-63.
3. Conti, C.J. Concepts for buffer storage. *IEEE Computer Group News* **2**, March 1969, 9-13.
4. Deutsch, L.P. Experience with a microprogrammed Interlisp system. *Proc. 11th Ann. Microprogramming Workshop*, Pacific Grove, Nov. 1979.
5. Forgie, J.W. The Lincoln TX-2 input-output system. *Proc. Western Joint Computer Conference*, Los Angeles, Feb. 1957, 156-160.
6. Geschke, C.M. *et. al*. Early experience with Mesa. *Comm. ACM* **20**, 8, Aug. 1977, 540-552.
7. Ingalls, D.H. The Smalltalk-76 programming system: Design and implementation. *5th ACM Symp. Principles of Programming Languages*, Tucson, Jan. 1978, 9-16.
8. Knuth, D.E. *TEX and METAFONT: New Directions in Typesetting.* American Math. Soc. and Digital Press, Bedford, Mass., 1979.
9. Lampson, B.W. *et. al*. 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.
10. 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.
11. Liptay, J.S. Structural aspects of the System/360 model 85. II. The cache. *IBM Systems Journal* **7**, 1, 1968, 15-21.
12. Metcalfe, R.M., and Boggs, D.R. Ethernet: distributed packet switching for local computer networks. *Comm. ACM* **19**, 7, July 1976, 395-404.

13. Mitchell, J.G. *et. al*. *Mesa Language Manual*. Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.

14. Pohm, A. *et. al*. The cost and performance tradeoffs of buffered memories. *Proc. IEEE* **63**, 8, Aug. 1975, 1129-1135.

15. Schroeder, M.D. Performance of the GE-645 associative memory while Multics is in operation. *Proc. ACM SigOps Workshop on System Performance Evaluation*, Harvard University, April 1971, 227-245.

16. Tanenbaum, A.S. Implications of structured programming for machine architecture. *Comm. ACM* **21**, 3, March 1978, 237-246.

17. Teitelman, W. *Interlisp Reference Manual*. Xerox Palo Alto Research Center, Oct. 1978.

18. 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.

19. Tomasulo, R.M. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. R&D* **11**, 1, Jan. 1967, 25-33.

20. Wilkes, M.V. Slave memories and segmentation. *IEEE Trans. Computers* **C-20**, 6, June 1971, 674-675.