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*.

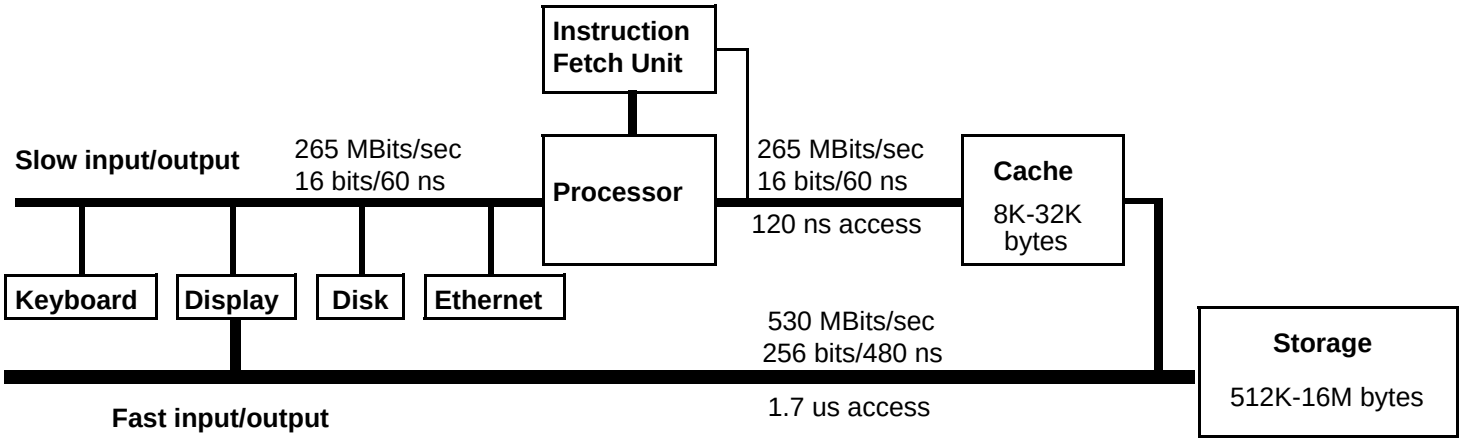

Figure 1: Dorado block diagram

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.

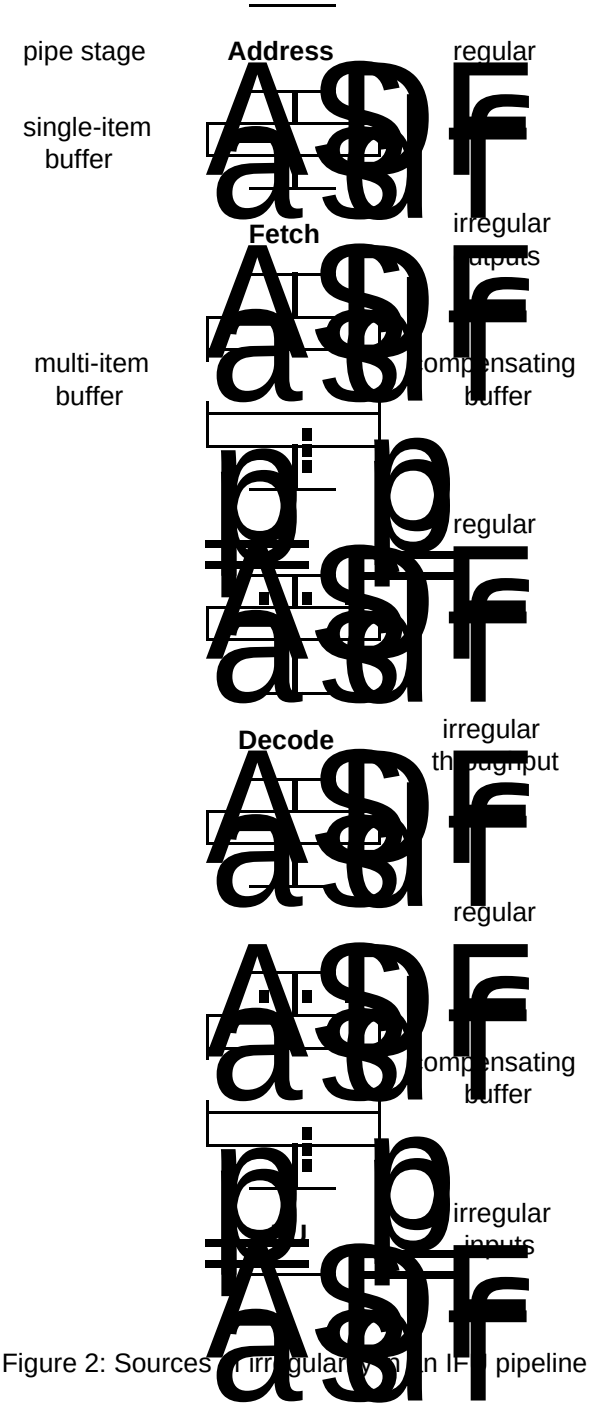

Figure 2: Sources of irregularity in an IFU pipeline

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.

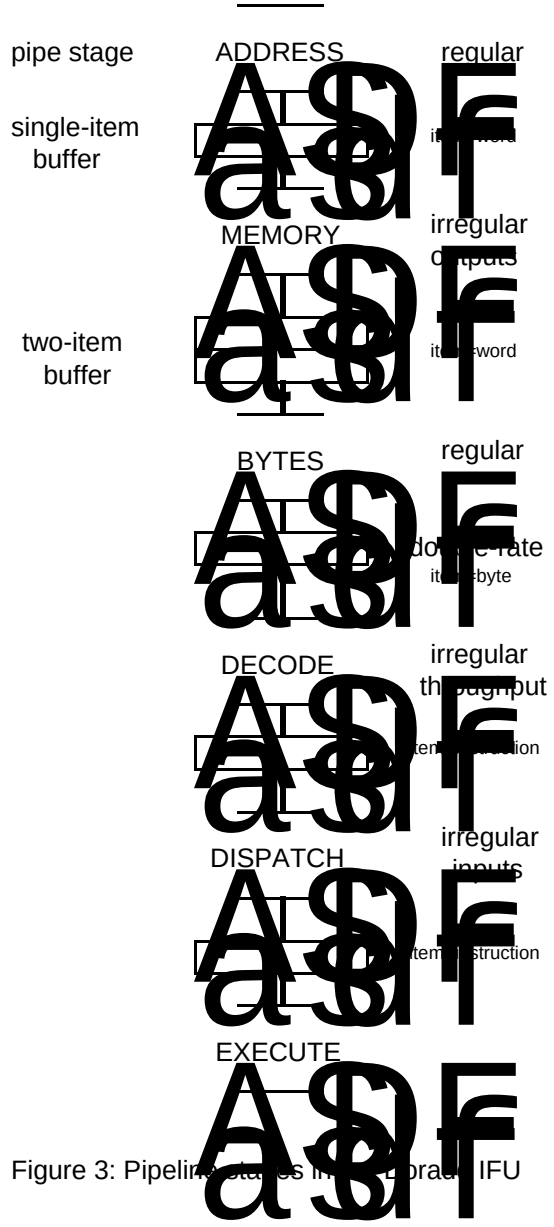

Figure 3: Pipeline stages in the broad IFU

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.

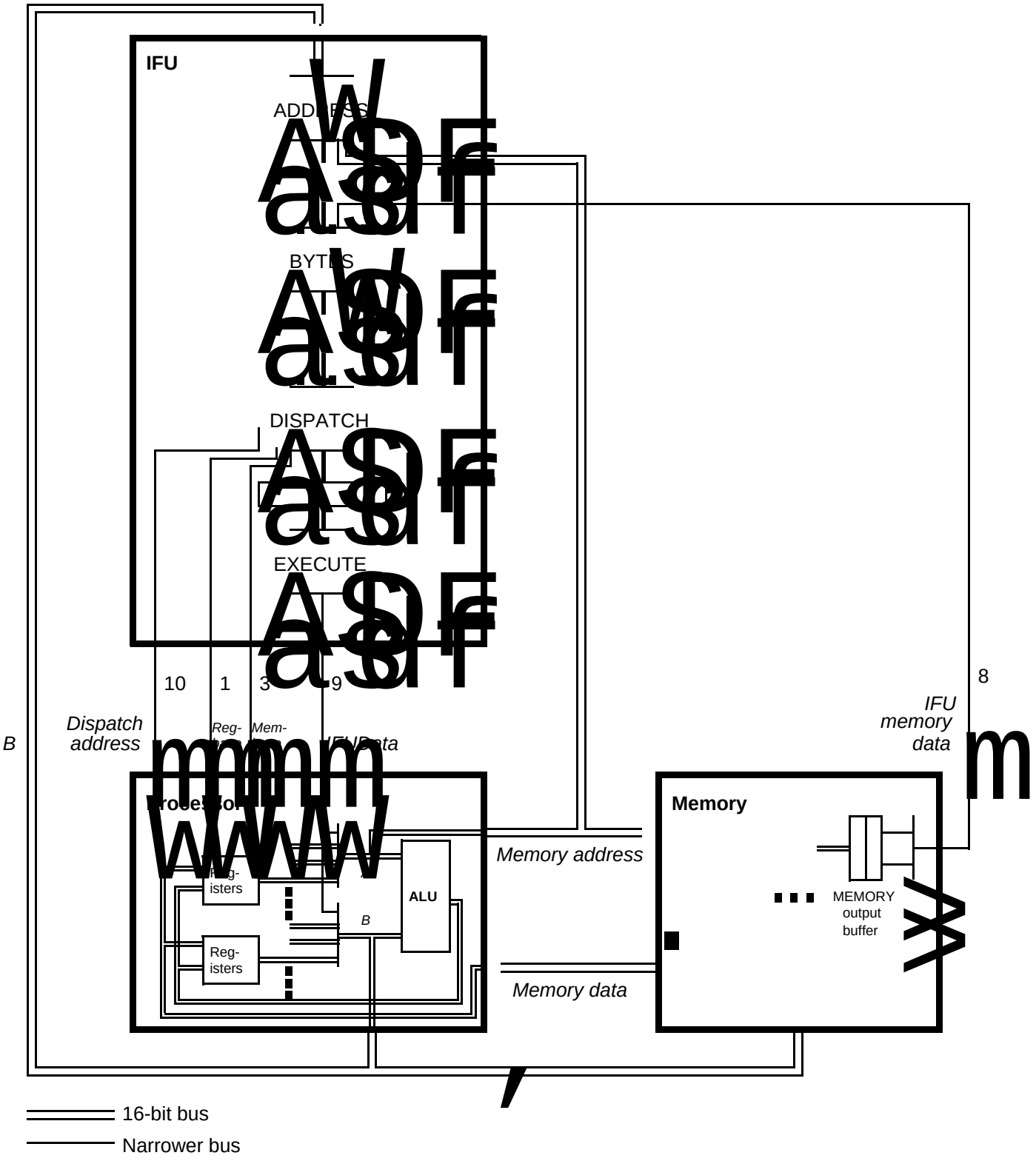

Figure 4: Busses between the IFU and the processor/memory

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

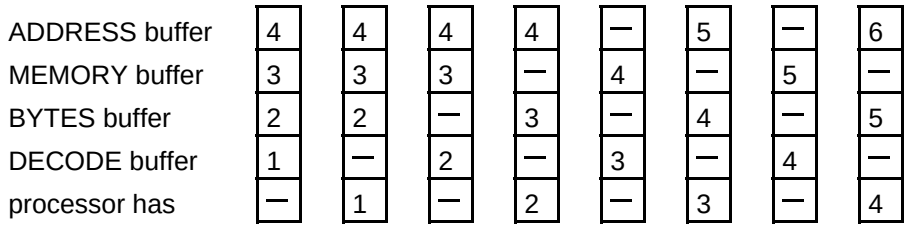

Figure 5a: Cogging with local control and one item buffering

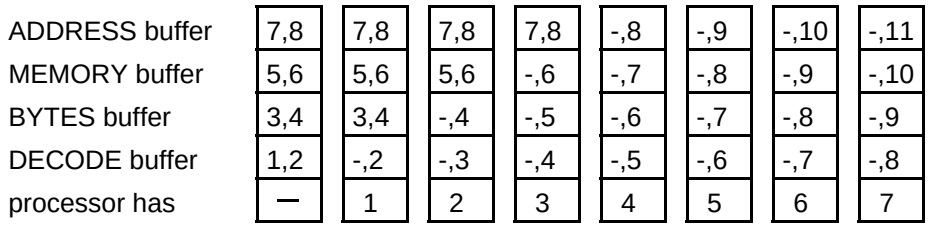

Figure 5b: Smooth operation with local control and two item buffering

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.

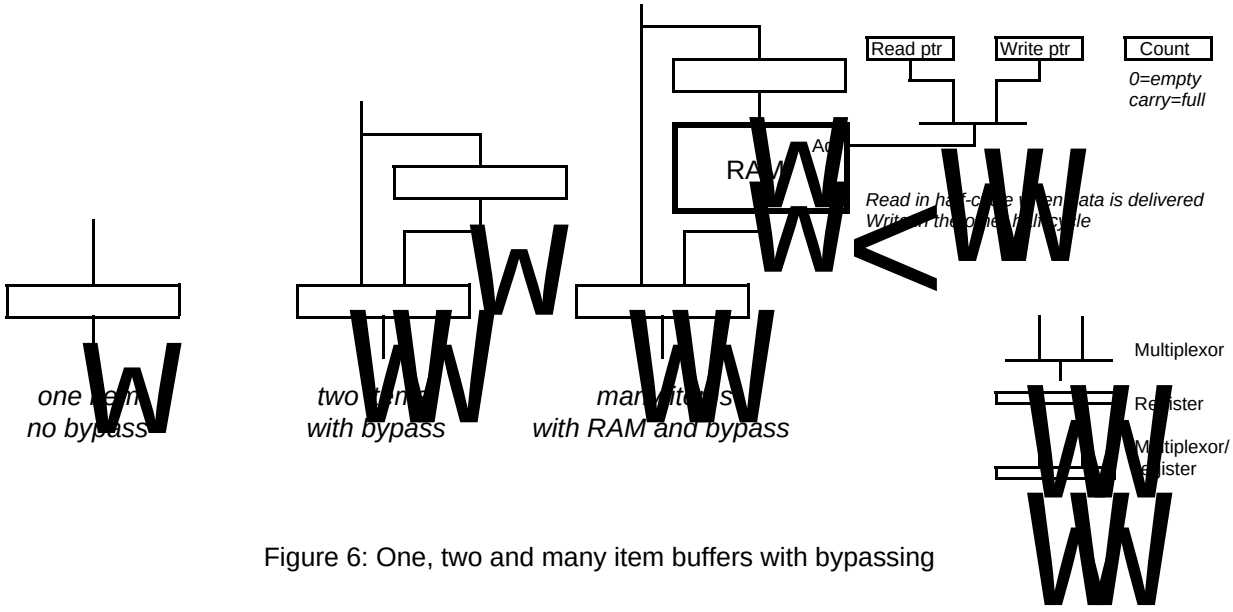

Figure 6: One, two and many item buffers with bypassing

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.

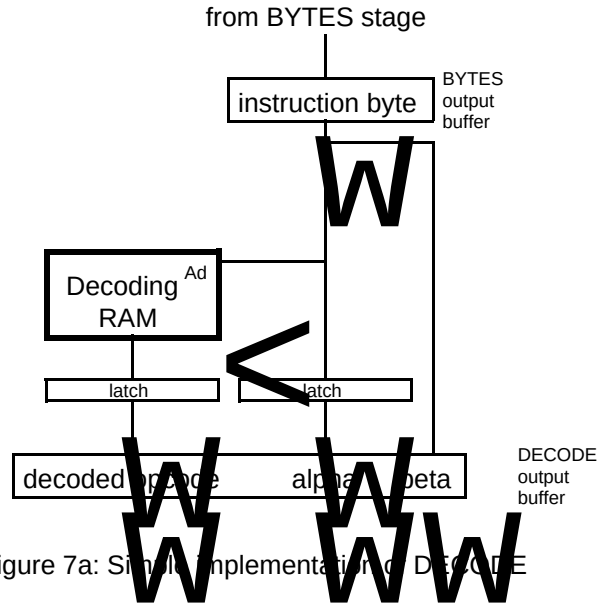

Figure 7a: Simple implementation of DECODE

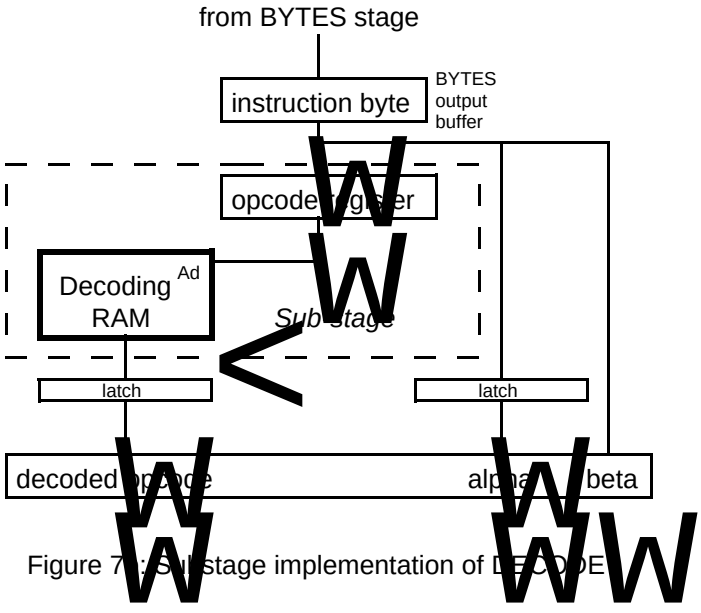

Figure 7b: Sub-stage implementation of DECODE

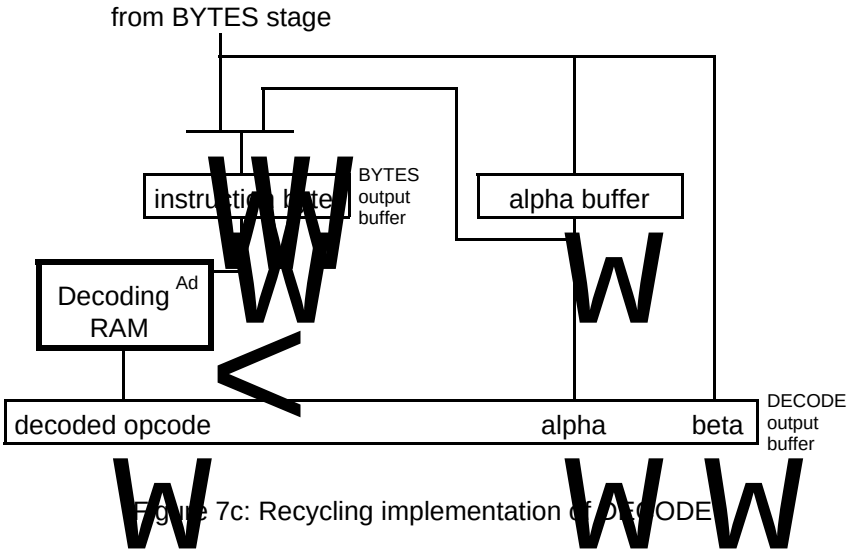

Figure 7c: Recycling implementation of DECODE

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.