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
instructions by pre-fetching, decoding, and preparing later instructions in parallel wit
execution of earlier ones. It dispatches the machine's microcoded processor to the prop
starting address for each instruction, and passes the instruction's fields to the proces
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. Ther
implementations of specialized instruction sets for the Mesa, Lisp, and Smalltalk langua
The IFU is implemented with a six-stage pipeline, and can decode an instruction every 60
Under favorable conditions the Dorado can execute instructions at this peak rate (16 mip

This paper has been submitted for publication.

CR CATEGORIES

6.34, 6.21

KEY WORDS AND PHRASES

cache, emulation, instruction fetch, microcode, pipeline.

C Copyright 1981 by Xerox Corporation.

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
designed to meet the needs of computing researchers at the Xerox Palo Alto Research Cent
These people work in many areas of computer science: programming environments, automated
office systems, electronic filing and communication, page composition and computer graph
design aids, distributed computing, etc. There is heavy emphasis on building working pr
The Dorado preserves the important properties of an earlier personal computer, the Alto
removing the space and speed bottlenecks imposed by that machine's 1973 design. The his
design goals, and general characteristics of the Dorado are discussed in a companion pap
which also describes its microprogrammed processor. A second paper [1] describes the me
system.

The Dorado is built out of ECL 10K circuits. It has 16-bit data paths, 28 bit virtual ac
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
a microinstruction every 60 ns. An instruction of some high level language is performed
executing a suitable succession of these microinstructions; this process is called emula

<==<IfuFig1.press<

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

There are implementations of specialized instruction sets for the Mesa [9], Lisp [12], a
[5] languages, as well as an Alto [13] emulator. The IFU can decode an instruction every
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
outline the space of possible solutions (¶ 2). We then describe the architecture of the
(¶ 3) and its interactions with the processor which actually executes the instructions (
who likes to see concrete details might wish to read these sections in parallel with ¶ 2
section deals with the internals of the IFU, describing how to program it and the details
pipelined implementation (¶ 5). A final section tells how large and how fast it is, and
information about the effectiveness of its various mechanisms for improving performance

2. The problem

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

We illustrate this point with the following fragment of an emulator for a stack-based in
The fragment includes the basic instruction fetch operation and code for two instruction
PushConstant, which pushes the next instruction byte onto the stack, and PushLocalVar, whic
the contents of the local variable addressed by the next byte (relative to a pointer in
localData). The notation is self-explanatory for the most part. Microinstructions are s
semicolons, and parallel operations in the same microinstruction by commas. This code u
special-purpose operations, except that we have compressed the details of the stack mani
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 f
                                       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 operati
presumably established by circuit speeds), it is necessary to do more of the operations
One possibility is to enhance the processor, so that it can do several operations in a s
instruction. 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
                                       for next instruction.
```

This approach is fine as far as it goes, but it is limited to combining independent oper
Fetch and the following retrieval of data, for example, cannot be combined without making
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 o
instructions at once. With n copies, this would run n times as fast if there were no sy
problems; it would also be very simple to implement (though perhaps not cheap). Unfortu
program written in a conventional language and encoded into a conventional instruction s
typically has a great deal of interaction between the successive instructions. For inst
the instruction sequence PushConstant, PushLocalVar, Add. We see from the microcode above
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
are finished; this is not only contention, but dependency of one instruction on the resu

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 hig performance machines such as the CDC 6600 [14] and 7600, the IBM 360/91 [16], the MU5 [4], the Cray-1 [10]; typically only part of the processor is duplicated, often into speciali called functional units. However, with 1977 technology this approach is too expensive f 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 dividing it into parts, each one to be performed by a separate processor or stage. Diff can operate concurrently on successive instructions. In this example, we might have one fetching the instruction (GetInstruction), and another for executing it (PushConstant and P Successive instructions can then execute as follows (where each line represents a "major

```
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 doubled.

2.1 About pipelines

An ideal pipeline has no communication between the stages except when work is passed fro stage to its successor. The unit of work which is passed between stages is called an it 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 o results of a later one (e.g., a conditional branch);

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

processing items in the different stages (e.g., memory fetches may be slow, 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 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 otherwi (let l be the latency);

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

A synchronous, uniform pipeline is one in which each stage takes the same amount of time stages we have l=nt, where t is the time of each stage. With many small stages, t can b small and the throughput high, at the expense of the latency. The only absolute limit t 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 int
tends to be a practical limit also. This limit can be evaded, however (at some cost), b
copies of the hardware, assigning the work to them in round-robin fashion, and selecting
by the same round-robin rule. If a single stage has t=s, such a duplicated stage has t=s
time for multiplexing the results. When this method is used, the copies are usually cal
units.

Usually the main goal is to maximize the throughput; in the absence of dependencies late
unimportant. As dependencies increase, however, latency becomes more important. To se
this is true, consider the backward dependency caused by a conditional branch. Strictly
when a branch instruction is encountered, fetching cannot proceed until the result of th
known. When it is, the target instruction of the branch must traverse the pipe before a
instructions can be completed. If w is the fraction of branch instructions, the average
time will be t+wl. Thus if l=5t (a five stage uniform pipe), a w of 20% will halve the
In this example, of course, it is sensible to make a guess and follow one path, so that
fraction of instructions for which a wrong guess is made; note that w=20% is fairly accu
prediction. Following a guessed path is easy because there are no forward dependencies (
state is never changed by instruction fetching), so that a wrong path can be abandoned w
effects. However, no such shortcut is possible in the case of the Add instruction menti
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 earlie
(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 convention
architecture there are many dependencies among the last three stages, because results ar
being stored into memory or register locations from which operands are fetched. Further
every store operation is regarded as a dependency, there could never be much concurrency
it is necessary to compare the address of each location modified by a store with all the
referenced by earlier stages. Even these dependencies are common enough to be painful;
provision is usually made in such a pipeline for modifying the actions of earlier stages
operands are changed by stores. As a result of all this, pipelining the last three stag
is a complex and expensive business. A fast multi-port cache inside the processor makes
problem much easier, but is not feasible with this technology. An interesting but untri
impose programming restrictions which forbid harmful dependencies; if all the code is ge
compilers this is quite feasible.

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

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

The EU demands instructions from the IFU at an irregular rate, depending on how fast it i absorb the previous ones. A simple machine must completely process an instruction befor demanding the next one. In a machine with multiple functional units, on the other hand, stage in the EU waits until the basic resources required by the instruction (adders, res etc.) are available, and then hands it off to a functional unit for execution. Beyond t operation cannot be described by a single pipeline, and complete execution of the instru be long delayed, but even in this complicated situation the IFU still sees the EU as a si 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 accommodate specific language features, we draw an example from Smalltalk [5]. In this the basic executable operation is applying a function f (called a method) to an object o The address of the code for the function is not determined solely by the static program, depends on a property of the object called its class. There are many implementation tec finding the class and then the function from the object. One possibility is to represen hash table which maps function names (previously converted by a compiler into numbers) i addresses, and to store the address of this table in the first word of the object. The operation of obtaining the hash table address and searching the table for the code addre 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, h

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 occup more bytes, and the first byte determines its essential properties (length, number etc.).

Matters are somewhat simplified if the addresssable unit is the unit delivered by the me instructions are all the same length, and somewhat complicated if instructions may be an 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 a program counter, one memory word at a time.

Fetching data from the code at these addresses. This requires interactions with t machine's memory in general, although recently used code may be cached within the Such a cache looks much like main memory to the rest of the IFU.

Decoding instructions to determine their length and internal structure, and perhap they are branches which the IFU should execute. Decoding changes the representatio the instruction, from one which is compact and convenient for the compiler, to one is convenient for the EU and IFU.

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

Buffering may be introduced between any pair of these stages, either the minimum of one required to separate the stages, or a larger amount to increase the elasticity. Note th be a word early in the pipe (at the interface to the memory), must be an instruction lat (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 taken:

> The instruction length is irregular, as noted in the previous paragraph; hence a u
> flow of instructions to the EU implies an irregular flow of bytes into the decoder,
> versa.
>
> The memory takes an irregular amount of time to fetch data; if it contains a cache
> 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 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-consu relationship. The EU demands a new instruction. If one is ready, the IFU delivers it as suitably formatted bits, and forgets about the instruction. Otherwise the IFU notifies t is not ready; in this case the EU will presumably repeat the request until it is satisfie level of abstraction, hand-off is a synchronized transfer of one data item (a decoded in from one process (the IFU) to another (the EU).

Usually the data in the decoded instruction can be divided into two parts: information a to do, and parameters. If the EU is a microprogrammed processor, for example, what to do conveniently be encoded as the address of a microinstruction to which control should go address), and indeed this is done in the Dorado. Since microinstructions can contain im constants, and in general can do arbitrary computations, it is possible in principle to information in the instruction into a microinstruction address; thus the instructions Pus and PushConstant(4356) could send control to different microinstructions. In fact, howeve instructions are expensive, and it is impractical to have more than a few hundred, or at thousand of them. Hence we want to use the same microcode for as many instructions as p representing the differences in parameters which are treated as data by the microcode. 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 not ready, it can dispatch the EU to a special NotReady location. Here the microcode can background processing it might have, and then repeat the demand for another instruction. same method can be used to communicate other exceptional conditions to the EU, such as a fault encountered in fetching an instruction, or an interrupt signal from an I/O device. Dorado's IFU uses this method (see ¶ 3.4).

Measurements of typical programs [7, 11] reveal that most of the instructions executed a and hence can be handled quickly by the EU. As a result, it is important to keep the cos off low, since otherwise it can easily dominate the execution time for such instructions gets faster, this point gets more important; there are many instructions which the Dorad instance, can execute in one cycle, so that one cycle of hand-off overhead would be 50%. 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 fu independently of the execution unit, which is the master in their relationship. At one can have an IFU which is entirely independent of the EU after it is initialized with a co might also receive information about the outcome of branches); this initialization would on a process switch, complex procedure call, or indexed or indirect jump. At the other an IFU which simply buffers one word of code and delivers successive bytes to the EU; whe buffer is empty, the IFU dispatches the EU to a piece of microcode which fetches another word's worth of code into the buffer. The first IFU must decode instruction lengths, fol and provide the program counter for each instruction to the EU (e.g., so that it can be s return link). The second leaves all these functions to the EU, except perhaps for keepin 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 h the performance improvement of the first one, at one-tenth the cost in hardware. The re be seen by examining the interpreter fragment at the beginning of ¶ 2; half a dozen micr instructions are typically consumed in the clumsy GetInstruction operation, and things get when instructions do not coincide with memory words.

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

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

The branch decision may be based on other historical information. The S-1 [17], for ins in its instruction cache one bit for each instruction, which records whether the instruc last time it was executed. This small amount of partial history reduces the fraction of branch decisions to 5% [Forest Baskett, personal communication]. The MU5 [4] remembers t addresses of the last eight instructions which branched; such a small history leaves 35% branches predicted wrongly, but the scheme allows the prediction to be made before the i is fetched. More elaborate designs [16] follow both branch paths, discarding the wrong the processor makes the branch decision. Each path may of course encounter further bran which in turn may be followed both ways until the capacity of the IFU is exhausted. If e truly followed in parallel, then following n paths will in general require n times as mu and n times as much memory bandwidth as following one path. Alternatively, part or all IFU's resources may be multiplexed between paths to reduce this cost at the expense of ba

2.6 Buffering

As we saw in ¶ 2.2, a pipeline with any irregularities must have buffering to provide el performance at each instant will approximate the performance of the slowest stage at tha this maximizing of the worst performance is highly undesirable. From the enumeration in irregularities in the IFU, we can see that to serve the EU smoothly, there should be a bu the EU and any sources of irregularity, as shown in Figure 2. Similarly, to receive word irregular memory, there should be a buffer between the memory and any sources of irregul Because of the irregularity caused by variable length instructions, a single buffer cann functions. Note that additional regular stages (some are shown in the figure) have no e way or the other.

<==<IfuFig2.press<

The cost of introducing a buffer (in the ECL 10K MSI technology) is the RAM storage to imp
it, a multiplexor to bypass it when it is empty, and its control; see Figure 6 for detai
ensures that the buffer does not increase the latency. In addition, there is typically a
performance penalty: when the pipe is reset, any external resources (the memory in the c
IFU) which have been used to fill the buffers are wasted. If some other processor could
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. I structure follows from the principles of the previous section, though we must admit that in fact proceeded less from general principles than from the goal of delivering one deco instruction per microcycle. This performance requirement dictates an autonomous IFU, and requires careful attention to the details of IFU-EU hand-off. In the Dorado the EU is a m processor with a number of data paths, and a pipelined implementation which allows it to microinstruction every 60 ns; in order to remind the reader of this implementation, we u "processor" to denote the Dorado’s EU. The processor does not have any significant concu visible to the microprogram, however. In particular, all the work done in a given cycle directly by the microinstruction executed in that cycle, although memory references are 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 of decoded instructions, which are from sequential bytes except where the IFU has followe branch. This sequence continues until the processor resets the IFU with another PC, unle or interrupt is detected. For each instruction the IFU supplies a microcode dispatch add which NotReady and all other exceptions are encoded), some bits of initial state for the sequence of field data values, and the PC value for the first byte of the instruction. T 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 Although this model is not specialized to the details of a particular instruction set, g performance depends on adherence to certain rules. The IFU deals only with instructions as variable length byte sequences   byte codes [3, 11]. Variable length instructions pr compaction, since frequent instructions can be small. There is also a performance payof and virtual memory systems, since the compaction enhances locality and thus reduces cach and page faulting. Our experience has shown that byte codes provide a flexible format f languages without favoring a particular one. The choice of eight bits as the grain is a among optimum encoding, the desire to keep code addresses short, and simplicity of the h A larger grain is highly undesirable, both because more than half the instructions can f byte, and because table lookup as a decoding technique is not feasible for units much la eight bits. A finer grain improves code compactness somewhat at the expense of more com length calculation and word disassembly.

The first byte of each instruction, called the opcode, is decoded by full table lookup. followed by as many as two optional data bytes (known as alpha and beta respectively) th passed to the processor with only slight reformatting. Of course the processor is free these bytes as it wishes, but the IFU can only do complex decoding operations on the opco The limitation to three byte instructions reduces hardware complexity at a considerable speed for longer instructions; bytes after the third must be fetched explicitly by the p 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 table. The additional two bits of address come from an instruction-set register. The 2 of the table describe the instruction in sufficient detail for the IFU and the processor jobs, and the opcode byte itself is not passed to the processor. Thus the table lookup the transformation of the instruction; it also governs some minor transformations of the such as sign extension.

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

(e.g., register or mode fields) to be extracted with complete flexibility. Indeed, it i
for such fields to exist explicitly. If single-byte PushConstant instructions for values
any five opcode values can be assigned for this purpose, and the table can produce the v
Furthermore, no sharp distinction is needed between "control" and "data" in the instruct
encoding, since both control information and data values are produced by the same table

Of course nothing is perfect. This scheme may fail when an instruction has many small f
especially if they cross byte boundaries. The PDP-11 and Nova instruction sets are inter
borderline cases: it works quite well to look up the first byte and use the result to se
second lookup in an alternate table lookup, or treatment of the next byte as data. A co
way to describe this is to have the first byte specify either a two byte instruction, or
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
since it is not very useful for the instruction sets we actually use. It is simple, how
easily be added; the only delicate point is that the instruction set register must be sa
exception, or else exceptions must be prohibited before instructions which are decoded w
alternate table. Currently only the processor can change the instruction set, and it no
only when switching from one language to another. This facility is used in the Interlis
tation, for example, since the nucleus of this system is written in BCPL and compiled int
instruction set than the one used for Lisp.

Multiple decoding tables have other uses. In fact, the IFU can be viewed as a rather gen
stream processor. For example, consider the problem of generating halftone values for
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 pixe
sets of pixels are packed into words, $16/p_g$ per word and 16 per word respectively. Thus
binary pixel is generated, it is necessary to keep track of whether a new binary word mu
started (once every 16 binary pixels), and whether a new grey pixel is needed (once ever
binary pixels); in the latter case, a new grey word may be needed. Typical algorithms u
scan-line buffer containing an error value which must be compensated at each binary pixe
IFU can be used to fetch values from this buffer in parallel with the processor. Specia
opcode values can be used to mark the points which require one or more of the special ac
above. The decoding table will dispatch the processor to the special code for these fun
without any processor overhead. A trial implementation using this idea was about twice
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 e
operate in a single 60 ns cycle. For the most part all state is held in the buffers bet
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 p
rate of one per cycle. If there are no cache misses and no collisions with the processo
can accept an address in every cycle and return data words at the same rate two cycles l
under these ideal conditions the memory is not irregular. A double-rate (30 ns) stage (
delivers bytes to the decoder (DECODE), which can accept one opcode byte and one operand
a single cycle, though it requires a full cycle to process an instruction. This arrange
two-byte instructions to pass through the pipe at the rate of one per cycle; longer inst
require two cycles, but are rare. Because DECODE requires a full cycle, the peak rate fo
instructions is still one per cycle. Note that the processor cannot demand instructions
this anyway.

<==<IfuFig3.press<

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

There are two words of buffering after MEMORY, but there is no other buffering except for
minimum single item between stages, contrary to the arguments of ¶ 2.6. This design was
partly to save space, and partly because we did not fully understand the issues in maint
bandwidth. Fortunately the peak bandwidth of the IFU is substantially greater than what
processor is likely to demand for more than a very short interval (see ¶ 6), so that not
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
off from one stage to the next, rather than by establishing separate communication paths
example, a page fault from the memory is indicated by a status bit returned along with t
word; the resulting "page fault value" is propagated through the pipe and decoded into a
dispatch address which is handed to the processor like any ordinary instruction. Each e
its own dispatch address. Interrupts cause a slight complication. The IFU accepts a sig
Reschedule which means "cause an interrupt;" this signal is actually generated by I/O mic
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
find out what interrupt condition has occurred. Since the reschedule item replaces one
instructions in the code, it has a PC value, which is the address of the next instructio
executed. After the interrupt has been dealt with, the IFU will be restarted at that poi

The exceptions may be divided into three classes:

1) the IFU has not (yet) finished decoding the next instruction, and hence is not rea
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 fix
order. Exceptions are communicated only by a dispatch; hence, all exceptions having to
particular opcode must be detected before it is handed off. Thus all the bytes of an in
must have been fetched from memory and be available within the IFU before it is handed o

3.5 Contention and dependencies

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

Although ideally the MEMORY stage is regular, in fact collisions with the processor can h
these irregularities are partially compensated by the two words of buffering after MEMORY
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
Since no attempt is made to detect modifications of code being executed, or to execute b
which depend on the values of variables, the only IFU-processor communication is hand-of
synchronization and resetting of the PC, and these are also the only communication betwe
stages. The IFU is completely reset when it gets a new PC; no attempt is made to follow
one branch path, or to cache information about the code within the IFU. The shortage of
makes the implementation of synchronization rather tricky; see ¶ 5 .

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

same time. The processor actually has access to all the information needed to maintain
but the time required to do this in microcode would be prohibitive (at least one cycle p
instruction).

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

4. IFU-processor hand-off

With a microcoded execution unit like the Dorado's processor, efficient emulation depend
smooth interaction between the IFU and the processor, and on the right kind of concurrenc
processor itself. These considerations are less critical in a low-performance machine,
microcycles are used to execute each instruction, and the loss of a few is not disastrou
performance machine, however, executes many instructions in one or two microcycles. Add
or two more cycles because of a poorly chosen interface with the IFU, or because a very c
pair of operations cannot be expressed in a single microinstruction, slows the emulator
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

Good performance depends on two things:

An adequate set of data busses, so that it is physically possible to perform the f
combinations of independent data transfers in a single cycle. We shall be mainly
with the busses which connect the IFU and the processor, rather than with the inter
details of the latter. These are summarized in Figure 4.

A microinstruction encoding which makes it possible to specify these transfers in
microinstruction. A horizontal encoding does this automatically; a vertical one r
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
microinstruction. Usually folding is done by the microprogrammer, who surveys the opera
be done and the resources of the processor, and arranges the operations in the fewest po
number of microinstructions.

<==<IfuFig4.press<

4.1 How the processor sees the IFU

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

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

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

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.

IFUData is usually delivered to the A bus, one of the processor's two main input busses, it can be sent through the ALU, or used as a displacement in a memory reference. In this encoded in the microinstruction field which controls the contents of this bus, and hence done concurrently with all the other operations of the processor. IFUData can also be del the other main input bus, from which it can be shifted, stored, sent to the other ALU inp output. This operation is encoded in the special function field, where it excludes a la relatively infrequent operations as well as immediate constants and long jumps, all of w 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 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 start fetching instructions from the location addressed by the new PC.

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

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

We illustrate the use of these operations with some examples. First, here is the actual 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 eig 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 inst
```

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

> Any number of microinstructions can be executed to emulate an instruction, i.e., b IFUJumps.
>
> Within an instruction, any number of IFUData requests are possible; see Table 3 for summary of the data delivered to successive requests.
>
> IFUJump and IFUData may be done concurrently. The IFUData will reference the current instruction's data, and then the IFUJump will dispatch the processor to the first m tion of the next instruction (or to NotReady).

Suppose analysis of programs indicates that the most common PushConstant instruction push constant 0. Suppose further that 1 is the next most common constant, and 2 the next beyo and that all other constants occur much less frequently. A lot of code space can probab by dedicating three one-byte opcodes to the most frequent PushConstant instructions, and two-byte instruction for the less frequent cases, as in the PushConstantByte example above opcode byte designates a PushConstantByte opcode and alpha specifies the constant. A third 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, PushC2, PushCB, PushCW. Any five distinct values can be assigned for the opcode bytes of instructions, since the meaning of an opcode is completely defined by its decoding table entries for these instructions are as follows: (N is a constant encoded in the opcode, L instruction length in bytes, and Dispatch is the microcode dispatch address; for details,

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 diff opcodes, for both one and two byte instructions. Only PushConstantWord requires two separ microinstructions.

4.2 Initializing state

A standard method for reducing the size and increasing the usefulness of an instruction parameterize it. For example, we may consider an instruction with a base register field parameterized by that register: the "meaning" of the instruction depends on the contents register. Thus the same instruction can perform different functions, and also perhaps c with a smaller address field. This idea is also applicable to microcode, and is used in For example, there are 32 memory base registers. A microinstruction referencing memory specify one of these explicitly; instead, there is a MemBase register, loadable by the mi which tells which base register to use. Provided the choice of register changes infrequ an economical scheme.

For emulation it presents some problems, however. Consider the microcode to push a loca
variable; the address of the variable is given by the alpha byte plus the contents of th
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
parametric state should be set from the IFU at the start of an instruction, using informa
decoding table. This is in fact done on the Dorado. The decoding table entry for PushLo
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 PushGlo
instruction, with a decoder entry which specifies the same dispatch address, but globalDa
initial value of MemBase. Thus there are two ways in which parameterization saves space
specifying everything in the microinstruction: each microinstruction can be shorter, and
needed. The need for initialization, however, makes the idea somewhat less attractive,
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 itsel
dispatch address. This has a substantial cost in microcode, since typically the number
dispatch addresses is about one-third of the 256 opcodes. If this price is paid and par
eliminated, however, the IFU can be considerably simplified, since not only the decoding
is saved, but also the buffers and busses needed to hand off the parameters to the proce
the parameterization mechanism in the processor itself. On the Dorado, the advantages o
parameterization were judged to be worth the price, but the decision is a fairly close o
current memory base register and the current group of processor registers are parameters
microinstruction which are initialized from the IFU. The IFU also supplies the dispatch a
the same time. The remainder of the information in the decoding table describes the dat
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
important technique for speeding up a microprogram. Often, however, we would like to fo
emulation of two successive instructions, deferring some of the work required to finish
one instruction into the execution of its successor, where we hope for unused resources.
cannot be done in the usual way, since we have no a priori information about what instru
comes next. However, there is a simple trick (due to Ed Fiala) which makes it possible
common cases.

We define for an entire instruction set a small number n of cleanup actions which may be
to the next instruction for completion; on the Dorado up to four are possible, but one m
be the null action. For each dispatch address we had before, we now define n separate o
for each cleanup action. Thus if there were D addresses to which an IFUJump might dispat
are now nD. At each one, there must be microcode to do the proper cleanup action in add
the work required to emulate the current instruction. The choice of cleanup action is s
the microcode for the previous instruction; to make this convenient, the Dorado actually
kinds of IFUJump operations (written IFUJump[i] for i=0, 1, 2, 3), instead of the one desc
above. The two bits thus supplied are ORed with the dispatch address supplied by the IFU
determine the microinstruction to which control should go. To avoid any assumptions abo
pairs of successive instructions can occur, all instructions in the same instruction set
same cleanup actions and must be prepared to handle all the cleanup actions. In spite o
limitation, measurements show that forwarding saves about 8% of the execution time in st
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
how one instruction's memory fetch operation can be finished by its successor, reducing
a PushLocalVar from two microinstructions to one. We use two cleanup actions. One is nu
0), but the other (action 2) finds the top of the stack not on the hardware stack but in
memoryData register. Thus, any instruction can leave the top of stack in memoryData and d
IFUJump[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 le
```

In both cases, the microcode executes IFUJump[2], since the top of stack is left in the m
register, rather than on the stack as it should be. In the case of PushLocalVar[2], the p
tion has done the same thing. Thus, the microcode at this entry point must move that da
stack at the same time it makes the memory reference for the next stack value. The read
that successive Push instructions will do the right thing. Of course there is a payoff o
the first microinstruction of PushLocalVar[0] is not using all the resources of the proces

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
                                        stack.
```

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

4.4 Conditional branches

We conclude our discussion of IFU-processor interactions, and give another example of for
with the example of a conditional branch instruction. Suppose that there is a BranchNotZ
instruction that takes the branch if the current top of the stack is not zero. Assume t
decoding table entry tells the IFU to follow the branch, and specifies the instruction le
first IFUData 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
                                        refills starting at the new location.
```

The most likely case (the top of the stack non-zero) simply makes the test specified by
instruction and does an IFUJump (two cycles). If the value is zero (the IFU took the wron
the microcode computes the correct value for the new PC and redirects the IFU accordingly
cycles, plus the IFU's latency of five cycles; guessing wrong is painful). If we think t
BranchNotZero will usually fail to take the branch, we can program the decoding table to

an ordinary instruction and deliver the branch displacement as IFUData, and reverse the s
test.

A slight modification of the forwarding trick allows further improvement. We introduce
action (say action 1) to do the job of InsFromIFUData above (it must be action 1 or 3, sin
successful test in the Dorado ors a 1 into the next microinstruction address). Now we wr
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 IFUJu
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
                                       refills starting at the new location.
```

Now a branch which was predicted correctly takes only one microinstruction. For this to
processor must keep the IFU from advancing to the next instruction if there is a successf
the IFUJump cycle. Otherwise, the PC and IFUData of the branch instruction would be lost,
cleanup action could not do its job. Note that the first line at EveryInstruction[1] must
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 pri
focus of attention is the pipeline structure, discussed within the framework established
3.3, but in addition we give (in ¶ 5.4) the format of the decoding table, which defines
can be specialized to the needs of a particular instruction set. Figure 3 gives the big
pipeline. Table 1 summarizes the characteristics of each stage; succeeding subsections
row of the table in turn. The first row gives the properties of an ideal stage, and the
table describes departures from this ideal. This information is expanded in the remaind
section; the reader may wish to use the table to compare the behavior of the different s

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

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

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 IFUJump		only	NotReady is default delay; IFUHold is panic delay.
EXECUTE	byte	On IFUData	No output buffer	Reset unnecessary	

Table 1: Summary of the pipeline stages

<==<IfuFig5.press<

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

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

<==<IfuFig6.press<

The RAM requires two counters to act as read and write pointers, and a third to keep trac number of items in the buffer. In addition, it must be effectively two-ported, since in it is necessary to write one item and read an earlier one. In the Dorado two-port RAMS a many places; since no such part is available, they are implemented by running an ordinar twice the machine cycle (both 16x4 and 256x4 RAMS are available which can be read or wri 10 ns), and using a multiplexor to supply the read address in one half-cycle and the wri 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 in the items as they are stored in the inter-stage buffers. As a consequence, resetting 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 by fetched. It is the IFU's handling of jumps which makes this necessary; otherwise it woul remember the initial PC at the end of the pipe, and to increment it by the instruction le instruction goes by. Since no jumps can be executed between the ADDRESS and BYTES states, method is in fact used there. It is especially convenient because BYTES handles one byte so that the PC can be held in a counter which is incremented once per item; later in the

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

Every item also carries a status field, which is used to represent various values that d
correspond to ordinary instructions: empty, page fault, memory error. These are convert
unique dispatch addresses when the item is passed to the processor, as discussed in ¶ 3.

5.1. ADDRESS stage

This stage generates the addresses of memory words which contain the successive bytes of
Unlike the other stages, it has no ordinary input, but instead contains a PC which it inc
two (there are two bytes per memory word) for each successive reference. The PC can also
a pause value which prevents any further memory references until the processor resupplies
with an ordinary PC value. This pause state plays the same role for ADDRESS that an empty
buffer plays for the other stages; hence it is entered whenever this stage is reset. Th
either because of a processor Reset operation (which resets the entire IFU pipe, and is n
during normal execution), or because of a Pause signal from DECODE. Correspondingly, a ne
can be supplied either by a processor PC_ operation, or by a Jump signal from DECODE when
a jump instruction. Any of these operations resets the pipe between ADDRESS and DECODE; th
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.
ADDRESS contends with the processor for the memory address bus; since the IFU has lowest p
it waits until this bus is not being used by the processor. Finally, it is necessary to
space for the resulting memory word: the memory, unlike ordinary IFU stages, delivers its
unconditionally, and hence must not be started unless there is a place to put the result
surveys the buffering in the rest of the pipe, and waits until there are at least two fr
guaranteed; it isn't necessary for these bytes to be in the MEMORY output buffer, since da
buffer will advance into later buffers before the memory delivers the data. It is, howe
to make the most pessimistic assumptions about instruction length and processor demands.
basis, there are seven bytes of buffering altogether: four after MEMORY, two after BYTES, a
after DECODE.

5.2 MEMORY stage

This stage has several peculiarities. Some arise from the fact that most of it is not l
physically a part of the IFU, but instead is shared with the processor and I/O system. A
the previous section, the memory delivers results unconditionally, rather than waiting f
space to be available; ADDRESS allows for this in starting MEMORY. Furthermore, the memory
considerable internal state and cannot be reset, so additional logic is required to disc
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
ranges from two to about 30 cycles (the latter when there is a cache miss). To maintain
bandwidth, the IFU must therefore have more than one item in the MEMORY stage at a time; s
l=2 when the cache hits, and this is the normal case, there is provision for up to two i
MEMORY. A basic principle of pipeline stages is that items emerge in the order they are s
A stage with fixed latency, or one which holds only one item, does this automatically, b
has neither of these properties. Furthermore, its basic function is random access, with
relationship between successive references. Hence if one reference misses and the next
memory is happy to deliver the second result first. To prevent this from happening, the
notifies the memory that it has a reference outstanding when it makes the second one, an
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 outp
in fact two are provided. They are physically packaged with the cache data memory, as i
BYTES stage multiplexing required to produce individual bytes. As a result, a one-byte b
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
however, run twice as fast as the other stages, so that it can deliver two-byte instruct
rate of one per cycle. This means that the multiplexors must look at both words of the
output buffer, which runs only at the normal rate.

BYTES also includes a provision for replacing the first byte coming from memory with a by
from a substitute register within the stage. This feature makes it convenient to procee
breakpoint without removing the one-byte breakpoint instruction from the code; instead t
byte displaced by the breakpoint is loaded into the substitute register (by the microcod
substituted for the break instruction. Since the substitution is done only once, the br
normally when control returns to it. The substitute register is also a convenient way t
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 byte
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
with 256 opcodes each. It takes about two-thirds of a cycle to read these RAMS, with con
which are described below. The form of an entry is outlined in Table 2; parity is also
of this information is passed on directly to the DECODE buffer. The last three fields, h
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
isn't known until late in the DECODE cycle causes serious problems. A simple implementati
DECODE addresses the decoding table directly from the input buffer. If the instruction tu
be one byte long, it is delivered to the output buffer in the normal way. If it is long
output is latched and additional bytes are taken from BYTES until the complete instructio
DECODE ready to be delivered; see Figure 7a. Unfortunately, the length must be known bef
middle of the cycle to handle two-byte instructions at full speed. Figure 7b shows how

problem can be attacked by introducing a sub-stage within DECODE; unfortunately, this dela
reading of the decode table by half a cycle, so that its output is not available togethe
alpha byte. To solve the problem it is necessary to provide a second output buffer for
to feed back its contents into the main buffer if the instruction turns out to be only o
as in Figure 7c. 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 th
of BYTES prevents it from being done in one cycle anyway, space is saved by breaking it i
sub-instructions, each two bytes long; for this purpose a dummy opcode byte is supplied
alpha and beta. Each sub-instruction is treated as an instruction item. The second one
beta and is slightly special: DECODE ignores its dummy opcode byte and treats it as a two
instruction, and DISPATCH passes it on to EXECUTE after the alpha byte has been delivered.

<==<IfuFig7.press<

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

If the Jump field is set, DECODE computes a new program counter by adding an offset to the
the instruction. This offset comes from the alpha byte if there is one, otherwise from
SplitAlpha; it is sign-extended if Sign is true. The new PC is sent back to ADDRESS, as de
5.1, where Pause is also explained. Jump instructions in which the displacement is not e
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 ad
together with the state initialization discussed in ¶ 4.2, from the DECODE output buffer
executes an IFUJump. Because empty is encoded into a NotReady dispatch, the processor take
account of whether the buffer is empty. There are some ugly cases, however, in which DEC
unable to encode an exception quickly enough. In these cases DISPATCH asserts a signal ca
which causes the processor to skip an instruction cycle; this mechanism is rather expens
implement, and is present only because it was essential for synchronization between the
and the memory [1]. Once implemented, however, it is quite cheap for the IFU to use. Th
NotReady dispatch is still preferable, because it gives the microcode an opportunity to d
useful work while waiting.

5.6. EXECUTE stage

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

Jump	Length	N	SplitAlpha	IFUData
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 IFUData

6. Performance

The value of an instruction fetch unit depends on the fraction of total emulation time t (over doing instruction fetching entirely in microcode). This in turn clearly depends o of time spent in executng each instruction. For a language like Smalltalk-76 [5], a typ instruction requires 30-40 cycles for emulation, so that the half-dozen cycles saved by not very significant. At the other extreme, an implementation language like Mesa [9, 11 compiled into instructions which can often be executed in a single cycle; except for fun and block transfers, no Mesa instruction requires more than half a dozen cycles. For th 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 th system functions provided in this single-user system are included. Disk wait time is ex it would tend to bias the statistics. Some adjustments to the raw data have been made t artifacts caused by compatibility with an old Mesa instruction set. Time spent in the p and return instructions (about 15%) has been excluded; these instructions take about 10 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 mach cycle. Together with supporting microcode, these counters provide sufficient precision requires days of execution. It is possible to count a variety of interesting events; so permanently connected, and others can be accessed through a set of multiplexors which pr access to several thousand signals in the machine, independently of normal microprogram

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 t emulator or on the program being executed. The consequences of a particular limitation, depend on how frequently it is encountered in actual execution.

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

A branch correctly executed by the IFU causes a three-cycle gap in the pipeline. Hence i processor spends one cycle executing it and each of its two predecessors, it will see th cycles on the next IFUJump. Additional time spent in any of these three instructions, ho 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 instruct be handled at one per cycle, even with some processor references to memory. A stream of instructions, however (which would consume all the memory bandwidth if handled at full s results in 33% NotReady even if the processor makes no memory references. The reason is IFU cannot make a reference in every cycle, because its buffering is insufficient to abso irregularity in the processor’s demand for instructions. As we shall see, these limitat small practical importance.

6.2 NotReady dispatches

Our measurements show that the average instruction takes 3.1 cycles to execute (includin delays). Jumps are 26% of all instructions, and incorrectly predicted jumps (40% of all 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 the processor's demand for an instruction, i.e., the frequency of NotReady dispatches. I 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 incorrectly predicted jumps. Note that since these NotReady cycles are predictable, unli 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 NotReady dispatches from this cause. This is computed as follows: one cycle in three is 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 times 25, or 2.5%.

The other causes of NotReady account for only 1%. This is also predictable, since more t the instructions are one byte, and the average instruction makes only one memory referen 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. Furthermo line instructions are demanded at less than half the peak rate on the average, and jumps frequent that when the first instruction after a jump is dispatched, the pipe usually co 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 p between the processor and the IFU. Measurements show that the processor obtains a word f memory in 16% of the cycles, and the IFU obtains a word in 32% of the cycles. Thus data supplied by the memory in about half the cycles. The processor actually shuts out the I making its own reference about 20% of the time, since some of its references are rejecte memory and must be retried. The IFU makes a reference for each word transferred, and make unsuccessful references during its misses, for a total of 35%. There is no memory refer 45% of the time.

6.4 Forwarding

The forwarding trick saves a cycle in about 25% of the straight-line instructions, and h up straight-line execution by 8%. Jumps take longer and benefit less, so the speed-up w procedure is 5%. Like the IFU itself, forwarding pays off only when instructions are exe 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 stag
mostly devoted to handling several PC values. DECODE is large because of the decoding tab
RAM chips) and its address drivers and data registers, as well as the branch address calc

Table 5 shows the amount of microcode in the various emulators, and in some functions co
to all of them. In addition, each emulator uses one quarter of the decode table. Of co
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 Fial
reviewed the design, did the microassembler and debugger software, and wrote the manual.
emulators mentioned were written by Peter Deutsch, Willie-Sue Haugeland, Nori Suzuki and
Taft.

References

1. Clark, D.W. et. al. The memory system of a high performance personal computer. Technical Report CSL-81-1, 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 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. Com Architecture, SigArch/IEEE, La Baule, May 1980, 146-160. Also in Technical Report CSL-81-1, Xerox Palo Alto Center, Jan. 1981.
9. Mitchell, J.G. et. al. Mesa Language Manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, Ap
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 197 246.
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 edit Bell and Newell, eds., McGraw-Hill, 1981. Also in Technical Report CSL-79-11, Xerox Palo Alto Research Cen 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
16. Anderson, D.W. et. al. The System/360 Model 91: Machine philosophy and instruction handling. IBM J. R&D 11, 1967, 8-24.
17. Widdoes, L. C. The S-1 project: Developing high performance digital computers. Proc. IEEE Compcon, San Fr 1980, 282-291.